A Pebble Game for Internet-Based Computing

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

[1] Google, Inc.
malewicz@google.com
[2] Dept. of Computer Science, University of Massachusetts, Amherst, MA 01003, USA
rsnbrg@cs.umass.edu

Abstract. Advances in technology have rendered the Internet a viable medium for employing multiple independent computers collaboratively in the solution of a single computational problem, leading to the new genre of collaborative computing that we term *Internet-based computing* (*IC*). Scheduling a computation for IC presents challenges that were not encountered with earlier modalities of collaborative computing, especially when the computation's constituent tasks have interdependencies that constrain their order of execution. This paper surveys an ongoing study of (an abstraction of) the scheduling problem for such computations for IC. The work employs a "pebble game on computation-dags," that abstracts the process of allocating a computation's interdependent tasks to participating remote computers. The goal of a schedule, motivated by two related scheduling challenges, is to maximize the production rate of tasks that are eligible for execution. First, in many modalities of IC, remote computers become available at unpredictable times. Always having a maximal number of execution-eligible tasks enhances the utilization of available resources. Second, the fact that remote computers are often not dedicated to this IC computation, hence, may be more dilatory than anticipated, can lead to a type of "gridlock" that results when a computation stalls because (due to dependencies) all execution-eligible tasks are already allocated to remote computers. These motivating challenges raise the hope that the optimality results presented here within an abstract IC setting have the potential of improving efficiency and fault-tolerance in real IC settings.

1 Introduction

A variety of so-called *pebble games* on dags[3] (directed acyclic graphs) have been shown, over the course of several decades, to yield elegant formal analogues of a variety of problems related to scheduling the tasks/nodes of a computation-dag. The basic idea underlying such games is to use tokens (called "pebbles") to model the progress of a computation on a dag: the placement or removal of pebbles of various types—which is constrained by the dependencies modeled by the dag's arcs[4]—represents the changing (computational) status of the tasks represented

[3] Precise definitions of all required notions appear in Section 2.
[4] Pending the definitions of Section 2, one can refer to an algorithms text such as [5] for examples of task interdependencies and their representations via dags.

O. Goldreich et al. (Eds.): Shimon Even Festschrift, LNCS 3895, pp. 291–312, 2006.
© Springer-Verlag Berlin Heidelberg 2006

by the dag's nodes. Pebble games have been used to study problems as diverse as register allocation [17, 3], interprocessor communication in parallel computers [11], "out-of-core" memory accesses [10], and the bandwidth-minimization problem for sparse matrices (which can be formulated as a genre of scheduling problem) [20]. Additionally, pebble games have been shown to model many complexity-theoretic problems perspicaciously; see the survey [18]. The current paper is devoted to surveying ongoing joint work, [19, 21, 16], by the authors and M. Yurkewych (U. Massachusetts), which uses a new pebble game to study the problem of scheduling computation-dags for *Internet-based computing* (*IC*, for short). While this new game shares its basic structure with the "no recomputation allowed" pebble game of [20], it differs markedly from that game in the resource one strives to optimize.

A word about IC will explain the pebble game we study. Advancing technology has rendered the Internet a viable medium for employing multiple independent computers collaboratively in the solution of a single computational problem. A variety of mechanisms have been developed for IC, with "Web-based computing" [14], Peer-to-Peer computing (P2PC) [2, 23], and Grid computing [6, 7] being among the most popular.[5] Most forms of IC—including those just cited—lend themselves naturally to the master-slave computing metaphor, in which a master computer enlists the aid of remote "slave" (or, client) computers to collaborate in the computation of a massive collection of compute-intensive tasks. In rough terms, the differences among the listed modalities are as follows. In "Web-based computing," the remote clients are individuals who allow the master to download a program that will run in background on each client's pc; the clients are typically anonymous and, hence, untrusted; the project usually exists to perform a single computation. Grid computing—so named in analogy with a power grid—typically involves a fixed assemblage of computing sites that contract with one another to share computing resources (possibly, but not necessarily, including computing cycles); Grid members are usually mutually trusted. P2PC often shares with "Web-based computing" the anonymity of remote clients; it usually shares with Grid computing the revolving role of master and client, hence, a lifetime that goes beyond a single computation.

As with all new computing technologies, IC engenders novel scheduling challenges, even while enabling a large variety of computations that could not be handled efficiently by any fixed-size assemblage of dedicated computing agents (e.g., multiprocessors or clusters of workstations). Two related challenges that arise in IC motivate our study. First, in many modalities of IC, remote clients become available (to receive work) at unpredictable times. Second, the fact that remote clients are often not dedicated to the IC computation being performed raises the possibility that some may be slower than anticipated in returning the results from tasks allocated to them. (Indeed, in "Web-based computing," a client may

[5] Definitions and terminology in this fast-evolving field tend to vary from one researcher to another, but the definitions here should convey the essential nature of the three modalities of IC. We put "Web-based computing" in quotes because this specific modality has no generally accepted name.

never return its results.) When the tasks being computed are mutually independent, then (finite) delays by clients are just an annoyance; in particular, delays by "old" clients can never preclude having a new task available for allocation to a new client who becomes available. In contrast, when the tasks being computed have interdependencies that constrain their order of execution, dilatory clients may cause the supply of eligible tasks to be very small at certain times. Indeed, in the limit, an IC computation could occasionally encounter a type of "gridlock" wherein the computation stalls because (due to intertask dependencies) all tasks that are eligible for execution are already in the hands of remote clients. The dual scheduling challenges inherent in the preceding scenarios—to enhance the utilization of remote clients and to prevent "gridlock"—motivated the work we survey here.

As is common in the literature on scheduling (cf. [8, 9]), the studies we survey view the intertask dependencies of the computations being scheduled as having the structure of a dag. The goal of our schedules is to allocate the tasks of a given computation-dag to remote clients in a way that *always maximizes the number of tasks that are eligible for execution.* Although details must await further development and/or reference to [21], we can pictorially hint at the significance of the quality metric we are studying. Imagine that one wants to schedule a computation whose task-dependencies have the structure of the evolving mesh in (the upper left corner of) Fig. 1. If one schedules the dag along its "diagonal levels," as depicted in Fig. 2, then after having executed x tasks, one has roughly $\sqrt{x}$ tasks that are eligible to be the next executed task. In contrast, if one chooses to schedule the dag along its "square shells," as depicted in Fig. 3, then one never has access to more than three tasks that are eligible for execution. This example presents an atypically extreme contrast, but it should suggest that the rate of producing execution-eligible tasks may vary significantly for a given dag depending on the schedule used to execute the dag.

Of course, even if one were able to schedule all dags optimally within our idealized setting, one may not always eliminate the two motivating challenges. However, our scheduling strategies would provide guidelines that would provably improve utilization of remote clients and decrease the likelihood of gridlock—when tasks are executed in the order in which they are assigned to the clients. (One avenue toward achieving the desired order is to monitor the behavior and performance of remote clients, as mandated in [1, 13, 22].) And, importantly, the guidelines we derive prescribe actions that are under the control of the IC master and are independent of the behavior of the remote clients!

Our presentation centers on three topics. In Section 2.2, we define the IC Pebble Game that underlies the theory we are developing. Section 3 presents several results that suggest the range of ways that a given family of dags can fit into our embryonic theory—from not admitting an optimal schedule, at one extreme, to admitting infinitely many such schedules, at the other. Section 4 sketches some of the analyses used to derive optimal schedules for certain very uniform families of dags, notably, those in Fig. 1. Section 5 describes our latest, most exciting work, which establishes a foundation for a decomposition-based

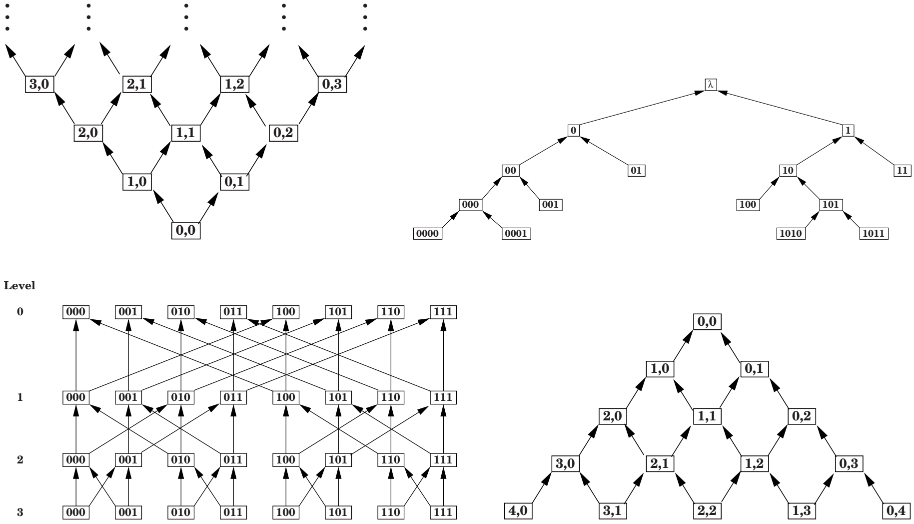

Fig. 1. *Clockwise from upper left: the (2-dimensional) evolving mesh, a (binary) reduction-tree, a (2-dimensional) reduction-mesh (or, pyramid dag), an FFT-dag.*

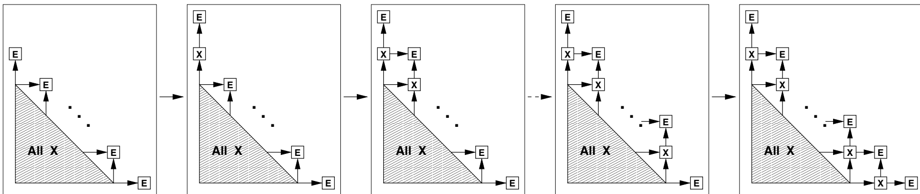

Fig. 2. *Computing a typical diagonal level of the evolving mesh.* "X" *denotes an* EXECUTED *node;* "E" *denotes an* ELIGIBLE *node.*

procedure that derives optimal schedules for a broad range of dags of quite complex, nonuniform structures. Finally, Section 6 describes our response to the fact that some dags admit no optimal schedules within the theory discussed thus far: a batched-scheduling analogue of our theory, within which optimal schedules exist for all families of dags. The major issue in the batched-scheduling setting is how complex (near-)optimal schedules are to derive.

2 A Formal Model for Scheduling Dags for IC

2.1 Computation-Dags

A *directed graph* (*digraph*, for short) $\mathcal{G}$ 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:

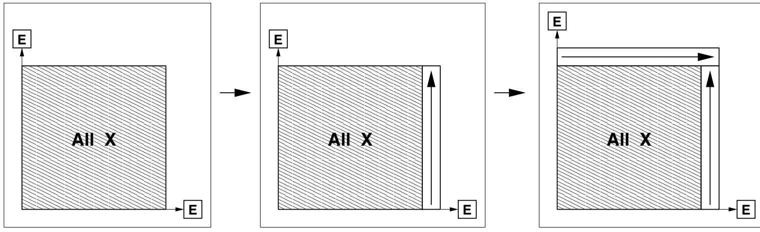

Fig. 3. *A schedule for* $\mathcal{M}_2$ *that traverses* square *levels.* "X" *denotes an* EXECUTED *node;* "E" *denotes an* ELIGIBLE *node. The long arrows indicate sequences of node-executions.*

$$(u_1 \to u_2),\ (u_2 \to u_3),\ \ldots,\ (u_{n-2} \to u_{n-1}),\ (u_{n-1} \to u_n) \qquad (1)$$

A *dag* (short for *directed acyclic graph*) $\mathcal{G}$ is a digraph that has no cycles; i.e., $\mathcal{G}$ cannot contain a path of the form (1) wherein $u_1 = u_n$. 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 of 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.

For each arc $(u \to v) \in A_{\mathcal{G}}$, we call u a *parent* of v and v a *child* of u in $\mathcal{G}$. The transitive extensions of the parent and child relations are, respectively, the *ancestor* and *descendant* relations. Excepting the degenerate dag that has no nodes: every dag has at least one parentless node (which is called a *source*); every finite dag has at least one childless node (which is called a *sink*). The *outdegree* of a node is its number of children. 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 $v \in N_{\mathcal{G}}$ is *incident* to some arc of $\mathcal{G}$, i.e., is either the node u or the node w of some arc $(u \to w) \in A_{\mathcal{G}}$. (Prohibiting "isolated" nodes avoids degeneracies.)

$\mathcal{G}$ is *connected* if, when one ignores the orientation of $\mathcal{G}$'s arcs, there is a path connecting every pair of distinct nodes. A *connected bipartite* dag $\mathcal{H}$ is a **constituent** of $\mathcal{G}$ just when:

1. $\mathcal{H}$ is an *induced subdag* of $\mathcal{G}$: $N_{\mathcal{H}} \subseteq N_{\mathcal{G}}$, and $A_{\mathcal{H}}$ comprises all arcs $(u \to v) \in A_{\mathcal{G}}$ such that $\{u, v\} \subseteq N_{\mathcal{H}}$.
2. $\mathcal{H}$ is *maximal:* the induced subdag of $\mathcal{G}$ on any *superset* of $\mathcal{H}$'s nodes—i.e., any set S such that $N_{\mathcal{H}} \subset S \subseteq N_{\mathcal{G}}$—is not connected and bipartite.

Let $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n$ be connected bipartite dags that are pairwise disjoint, in the sense that $N_{\mathcal{G}_i} \cap N_{\mathcal{G}_j} = \emptyset$ for all distinct indices i and j. The *sum* of these dags, denoted $\mathcal{G}_1 + \mathcal{G}_2 + \cdots + \mathcal{G}_n$, is the bipartite dag whose node-set and arc-set are, respectively, the unions of the corresponding sets of $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n$.

We have not posited the *finiteness* of computation-dags. While the inter-task dependencies in nontrivial computations usually have cycles—caused, say, by loops—it is useful to "unroll" these loops when scheduling the computation's individual tasks. This converts the computation's (possibly modest-size) computation-digraph into a sequence of expanding "prefixes" of what "evolves" into an enormous—often infinite—computation-dag. One typically has better algorithmic control over the "steady-state" scheduling of such computations if one expands these computation-dags to their infinite limits and concentrates on scheduling tasks in a way that leads to a computationally expedient sequence of evolving prefixes.

Fig. 1 displays four dags that are studied in [19, 21]: the mesh-dag in the upper left is an infinite dag (which has no sinks); the other three dags are finite. In Section 5, we outline the (de)composition-based theory of [16], which shows how to construct these four dag-families from bipartite building blocks.

2.2 The Internet-Computing Pebble Game

For brevity, we describe the Internet-Computing (IC) Pebble Game within a "pull"-based scheduling framework, in which remote clients approach the server seeking work; the reader can easily adapt our description to a "push"-based framework, in which the server polls remote clients for availability.

The Idealized IC Pebble Game The IC Pebble Game on a computation-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. We now present the rules of the Game, which simplify those of the original IC Pebble Game of [19, 21].

> Our simplification resides in the assumption that by monitoring remote clients (as mandated in, say, [1, 13, 22]) the Server can enhance the likelihood, if not the certainty, that remotely allocated tasks will be executed in order of their allocation. We idealize by assuming that the Server can ensure this ordering exactly.

Fig. 4 presents the rules of the IC Pebble Game; Fig. 2 illustrates the rules via a succession of moves of the Game on the 2-dimensional evolving mesh.

For each step t of a play of the IC Pebble Game on a dag $\mathcal{G}$, let $X^{(t)}$ denote the number of EXECUTED pebbles on $\mathcal{G}$'s nodes at step t, and let $E^{(t)}$ denote the analogous number of ELIGIBLE pebbles. Of course, $X^{(t)} = t$ in our idealized version of the Game, although this is not true in the original version of [19]

> *We measure the quality of a play of the IC Pebble Game on a dag $\mathcal{G}$ by the size of $E^{(t)}$ at each step t of the play—the bigger $E^{(t)}$ is, the better. Our goal is an* **IC optimal** *schedule, in which, for all steps t, $E^{(t)}$ is as big as possible.*

— S begins by placing an ELIGIBLE pebble on each unpebbled source of $\mathcal{G}$.
/*Unexecuted sources are always eligible for execution, having no parents whose prior execution they depend on.*/
— At each step, S
- selects a node that contains an ELIGIBLE pebble,
- replaces that pebble by an EXECUTED pebble,
- places an ELIGIBLE pebble 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 $\mathcal{G}$ is infinite.*/

Fig. 4. *The Rules of the IC Pebble Game*

The significance of IC quality—hence of IC optimality—stems from the following intuitive scenarios. (1) Schedules that produce ELIGIBLE nodes maximally fast may reduce the chance of a computation's "stalling" because no new tasks can be allocated pending the return of already assigned ones (the "gridlock" of the Introduction). (2) If the Server receives a batch of requests for nodes at (roughly) the same time, then an IC-optimal schedule allows maximally many requests to be satisfied, thereby enhancing the exploitation of clients' available resources.

3 The Boundaries of the Playing Field

The property of IC optimality is so demanding that it is not *a priori* clear that such schedules ever exist! The property demands that there be a *single* schedule Σ for a dag $\mathcal{G}$ such that, at *every* step of the computation, Σ maximizes the number of ELIGIBLE nodes across *all* schedules for $\mathcal{G}$. In principle, it could be that every schedule that maximizes the number of ELIGIBLE nodes at some step t requires that a certain set of t nodes is EXECUTED, while every analogous schedule for step $t+1$ requires that a disjoint set of $t+1$ nodes is EXECUTED. Indeed, there exist (simple) dags that do preclude IC-optimal scheduling for precisely this reason. However, there is a large class of computationally significant dags that can be scheduled IC optimally. In this section, we exhibit, in turn:

- simple dags that admit no IC-optimal schedule;
- a familiar family of dags (evolving meshes), each of which admits a unique strategy for producing IC-optimal schedules; we also show that a natural alternative to this schedule is actually pessimal in IC quality;
- a familiar family of dags (evolving trees) all of whose schedules are IC optimal.

Regrettably, we do not yet know the complexity of determining whether or not a given dag admits any IC-optimal schedule; this is an inviting research challenge.

Dags that admit no IC-optimal schedule. We begin with two recalcitrant dags; the reader can easily produce others.

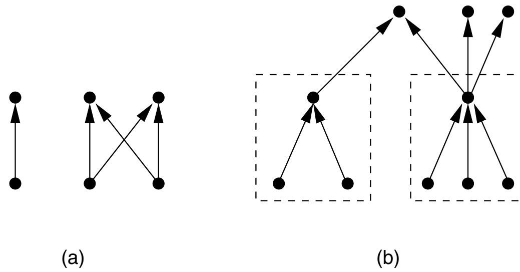

Fig. 5. *Two simple dags that admit no IC-optimal schedule.*

Fig. 5 contains two simple dags that do not admit any IC-optimal schedule — for precisely the reason mentioned in the opening paragraph of the section. For the 2-component dag of Fig. 5(a): in order to maximize the number of ELIGIBLE nodes at time $t = 1$, after one node is EXECUTED, one must begin executing the dag with the (unique) outdegree-1 source; in order to maximize the number of ELIGIBLE nodes at time $t = 2$, after two nodes are EXECUTED, one must begin executing the dag with the two outdegree-2 sources. For the tree-dag[6] of Fig. 5(b): in order to maximize the number of ELIGIBLE nodes at time $t = 2$, after two nodes are EXECUTED, one must begin executing the dag with the subtree in the lefthand dashed box; in order to maximize the number of ELIGIBLE nodes at time $t = 4$, after four nodes are EXECUTED, one must begin executing the dag with the subtree in the righthand dashed box.

Dags with a unique IC-optimal scheduling strategy. Fig. 1 (upper left) depicts the first four levels of the *evolving two-dimensional mesh-dag* $\mathcal{M}_2$. The nodes of $\mathcal{M}_2$ are all ordered pairs of nonnegative integers; its arcs connect each node $\langle v_1, v_2 \rangle$ to its two children $\langle v_1+1, v_2 \rangle$ and $\langle v_1, v_2+1 \rangle$. Node $\langle 0, 0 \rangle$ is $\mathcal{M}_2$'s unique source (often called its *origin*). The kth *diagonal level* of $\mathcal{M}_2$, denoted L_k, is the set of nodes whose coordinates sum to k. While $\mathcal{M}_2$ admits infinitely many IC-optimal schedules, all of them implement the strategy of proceeding along successive diagonal levels, from one end to the other.

Theorem 1 ([19]) (a) *For any schedule that allocates nodes sequentially along successive diagonal levels of* $\mathcal{M}_2$, $E^{(t)} = n$ *whenever* $\binom{n}{2} \le t < \binom{n+1}{2}$.
(b) *For any schedule for* $\mathcal{M}_2$, *if* t *lies in the preceding range, then* $E^{(t)}$ *can be as large as* n, *but no larger.*

[6] A *tree-dag* $\mathcal{T}$ is any dag such that, if one ignores the orientations of $\mathcal{T}$'s arcs, then the resulting graph is a tree (in the graph-theoretic sense).

It follows that $\mathcal{M}_2$'s IC-optimal schedules are precisely the diagonal-threading schedules.

The intuition underlying Theorem 1 resides in the following facts.

- Each row or column of $\mathcal{M}_2$ contains at most one ELIGIBLE node.
- All ancestors (parents, parents of parents, ...) of each ELIGIBLE node of $\mathcal{M}_2$ are EXECUTED.

Theorem 1 asserts that a *lazy* regimen for executing $\mathcal{M}_2$—i.e., one that always executes the *oldest* ELIGIBLE node, say, by proceeding up each diagonal level of $\mathcal{M}_2$—is IC optimal (albeit not uniquely so). In contrast, an *eager* regimen—i.e., one that always executes the *newest* ELIGIBLE node—is actually *pessimal* in IC quality. One implementation of the eager regimen is the "square-shell" schedule depicted schematically in Fig. 3. By fleshing out this schematic depiction to a level of detail commensurate with that of the lazy, "diagonal-level," schedule of Fig. 2, the reader will find that, under the "square-shell" schedule, no more than *three* nodes of $\mathcal{M}_2$ are ever simultaneously ELIGIBLE, in contrast with the ever-growing number of ELIGIBLE nodes promised by Theorem 1 for any lazy schedule.

Dags for which any *schedule is IC optimal.* Consider the *evolving binary out-tree* of Fig. 6. A simple argument shows that *every* valid schedule for the evolving

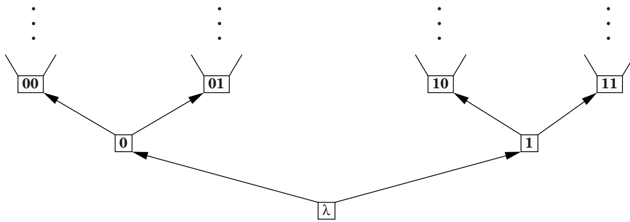

Fig. 6. *An evolving binary out-tree.*

binary out-tree $\mathcal{T}$ is IC optimal. To wit, at every moment during the execution of $\mathcal{T}$, the EXECUTED nodes are the internal nodes of a full binary sub-out-tree $\mathcal{T}'$ of $\mathcal{T}$, and the ELIGIBLE nodes are the leaves of $\mathcal{T}'$. It follows that at every step of any schedule for $\mathcal{T}$, the number of ELIGIBLE nodes is precisely one more than the number of EXECUTED nodes.

The messages of this section.

- There are significant families of dags that admit IC-optimal schedules.
- The disparity between the IC quality of an optimal schedule and that of a natural competitor can be very large.

– No scheduling strategy is going to guarantee IC-optimal schedules for all dags.

The remaining sections summarize our responses to these messages.

4 IC-optimal Schedules for Specific Families of Dags

In this section, we complete the scheduling story for the dags in Fig. 1 by dealing with the three "reductive" dags.

Theorem 2 ([21]) *A schedule for any reduction-mesh, reduction-tree, or FFT-dag is IC optimal if, and only if, it is parent-oriented—i.e., it executes all parents of a node in consecutive steps.*

Since the proofs for the three families of dags share are almost identical in structure (cf. [21]), we sketch the proof only for the reduction-mesh.

The nodes of the *ℓ-level reduction-mesh M_ℓ* comprise the set of ordered pairs of integers $\{\langle x, y\rangle \mid 0 \le x + y < \ell\}$. M_ℓ's arcs connect each node $v = \langle x, y\rangle$ to node $\langle x-1, y\rangle$ whenever $x > 0$ and to node $\langle x, y-1\rangle$ whenever $y > 0$. The integer $x + y$ is the *level* of node $\langle x, y\rangle$. M_ℓ's ℓ source nodes are the nodes at level $\ell - 1$; M_ℓ's unique sink node is node $\langle 0, 0\rangle$, the sole occupant of level 0.

Focus on a play of the IC Pebble Game on M_ℓ. Say that at step t of the play, each level $l \in \{0, 1, \ldots, \ell-1\}$ of M_ℓ has $E_l^{(t)}$ ELIGIBLE nodes and $X_l^{(t)}$ EXECUTED nodes. Let c be the smallest level-number for which $E_c^{(t)} + X_c^{(t)} > 0$.

Claim. *Given the current profile $\langle X_l^{(t)} \mid 0 \le l < \ell\rangle$ of* EXECUTED *nodes:*

1. *The aggregate number of* ELIGIBLE *nodes at time t, $E^{(t)} \stackrel{def}{=} \sum_{i=0}^{\ell-1} E_i^{(t)}$, is maximized if all* EXECUTED *nodes on each level of M_ℓ are consecutive.*[7]
2. *Once $E^{(t)}$ is so maximized, we have $c \le E^{(t)} \le c + 1$.*

Each nonsource ELIGIBLE node of M_ℓ has two EXECUTED parents; any two consecutive nonsource ELIGIBLE nodes share an EXECUTED parent. We thus have the following system of inequalities.

$$\begin{array}{l} E_l^{(t)} \le X_{l+1}^{(t)} - X_l^{(t)} - 1 \ \text{ for } \ l \in \{c, c+1, \ldots, \ell-2\}; \\ E_{\ell-1}^{(t)} = \ell - X_{\ell-1}^{(t)}. \end{array} \qquad (2)$$

1. If all EXECUTED nodes occur consecutively along a level $l + 1$ of $\mathcal{M}_\ell$, then the inequality involving $E_l^{(t)}$ in (2) is an equality. Therefore, all inequalities in (2) are equalities when the EXECUTED nodes at every level occur consecutively. Further, such consecutiveness may decrease the value of c, by rendering new nodes ELIGIBLE at lower-numbered levels. Consequently, this arrangement of EXECUTED nodes maximizes the value of $E^{(t)}$.

[7] Nodes $u_0, u_1, \ldots, u_{k-1}$ are *consecutive* on level l of $\mathcal{M}_\ell$ just when each $u_j = \langle m + j,\ l - m - j\rangle$ for some $0 \le m \le l - k,\ 0 \le j < k$.

2. Summing the (now) equalities in system (2) yields an explicit expression for the maximum value of $E^{(t)}$ in terms of $\sum_{i=0}^{\ell-1} X_i^{(t)} = t$, namely: $E^{(t)} = \sum_{i=c}^{\ell-1} E_i^{(t)} = c+1-X_c^{(t)}$. Part (2) of the claim now follows, because when the EXECUTED nodes at each level of M_ℓ occur consecutively, we must have $X_c^{(t)} \leq 1$: a larger value would imply that $X_{c-1}^{(t)} + E_{c-1}^{(t)} > 0$.

For reduction-meshes, parent-orientation means "level-by-level" execution. For reduction-trees, the phrase means that each tree-node u and its "sibling" (i.e., the node that shares a child with u) must be executed in consecutive steps. For FFT-dags, the phrase means that each node u and its "butterfly partner"(i.e., the node that shares two children with u) must be executed in consecutive steps.

5 Toward a Theory of Scheduling *Composite* Dags

The similarities in the structures of the proofs of Theorem 2 for its three families of dags led us to seek a structure-based explanation of the similarities. We now describe the results of this quest, which has gone far beyond just the motivating explanation.

A hallmark of the nascent scheduling theory of [16] is that it seeks explicit IC-optimal schedules only for connected bipartite dags (which experience has shown is already often quite a challenge). It then uses these bipartite dags as building blocks for constructing complex dags that inherit their IC-optimal schedules from those of the bipartite dags. We outline this development in this section.

The following simple result is quite useful in analyzing scheduling strategies for possible IC optimality. It should allow the reader to intuit the proofs for several of the results that we present.

Lemma 3 ([16]) *If a schedule Σ for a dag $\mathcal{G}$ is altered to execute all of $\mathcal{G}$'s nonsinks before any of its sinks, then the IC quality of the resulting schedule is no less than Σ's.*

5.1 A Sampler of Bipartite Building Blocks

Our study applies to any repertoire of *connected bipartite building-block dags* that one chooses to build complex dags from. For illustration, though, we focus on the following specific building blocks. The following descriptions proceed left to right along successive rows of Fig. 7. For all descriptions, we use the drawings in Fig. 7 to refer to "left" and "right."

The first three dags are named for the letters suggested by their topologies.

W-dags. For each integer $d > 1$, the $(1, d)$-*W-dag* $\mathcal{W}_{1,d}$ has one source and d sinks; 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 identifying (or, merging) the rightmost sink of the former dag with the leftmost sink of the latter. W-dags epitomize "expansive" computations.

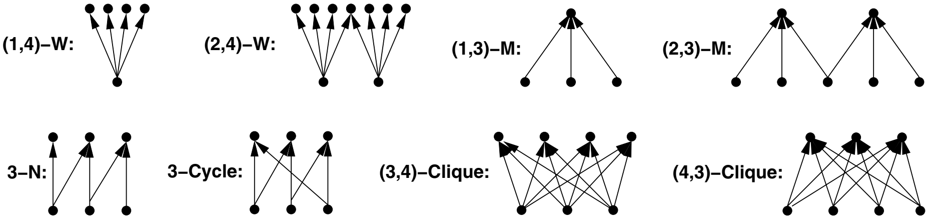

Fig. 7. *The building blocks of semi-uniform dags.*

M-dags. For each integer $d > 1$, the $(1, d)$-*M-dag* $\mathcal{M}_{1,d}$ has d sources and one sink; 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 identifying (or, merging) the rightmost source of the former dag with the leftmost source of the latter. M-dags epitomize "contractive" (or, "reductive") computations.

N-dags. For each integer $s > 0$, the s-*N-dag* $\mathcal{N}_s$ has s sources and s sinks; its $2s - 1$ arcs connect each source v to sink v and to sink $v + 1$ if the latter exists. Specifically, $\mathcal{N}_s$ is obtained from $\mathcal{W}_{s-1,2}$ by adding a new source on the right whose sole arc goes to the rightmost sink. The leftmost source of $\mathcal{N}_s$ has a child that has no other parents; we call this source the *anchor* of $\mathcal{N}_s$.

(Bipartite) Cycle-dags. For each integer $s > 1$, the s-*(Bipartite) Cycle-dag* $\mathcal{C}_s$ is obtained from $\mathcal{N}_s$ by adding a new arc from the rightmost source to the leftmost sink—so that each source v has arcs to sinks v and $v + 1 \bmod s$.

(Bipartite) Clique-dags. For integers $s, s' > 1$, the (s, s')-*(Bipartite) Clique-dag* $\mathcal{Q}_{s,s'}$ has s sources, s' sinks, and an arc from each source to each sink. (We actually deal only with (s, s)-Cliques, which we henceforth denote $\mathcal{Q}_s$; we present general (s, s')-Cliques as an invitation to the reader.)

5.2 Building Dags Via Composition

Our basic technique for constructing complex dags is the following inductively defined operation of *composition*.

– We start with any set **B** of connected bipartite dags; these will serve as our base set.
– Given dags $\mathcal{G}_1, \mathcal{G}_2 \in$ **B**—which could be copies of the same dag with nodes renamed to achieve disjointness—we obtain 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 $\mathcal{G}_1, \mathcal{G}_2$. We rename nodes to ensure that $N_{\mathcal{G}}$ is disjoint from $N_{\mathcal{G}_1}$ and $N_{\mathcal{G}_2}$.
  • We 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.
  • We 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.
– We add the dag $\mathcal{G}$ thus obtained to the base set **B**.

We denote the composition operation by $\Uparrow$ and refer to the resulting dag $\mathcal{G}$ as a *composite dag of type* $[\mathcal{G}_1 \Uparrow \mathcal{G}_2]$. (Note that the structure of $\mathcal{G}$ is not identified uniquely by its type. Our theory does not require knowledge of this detailed structure.) The roles of $\mathcal{G}_1$ and $\mathcal{G}_2$ in creating $\mathcal{G}$ are asymmetric: $\mathcal{G}_1$ contributes sinks, while $\mathcal{G}_2$ contributes sources.

We can now simply illustrate the natural correspondence between the node-set of a composite dag and those of its constituents, via Fig. 1:

- The evolving mesh $\mathcal{M}_2$ is composite of type $\mathcal{W}_{1,2} \Uparrow \mathcal{W}_{2,2} \Uparrow \mathcal{W}_{3,2} \Uparrow \cdots$.
- A binary reduction-tree is obtained by pairwise composing many instances of $\mathcal{M}_{1,2}$ (seven instances in the figure).
- The reduction-mesh M_5 is composite of type $\mathcal{M}_{5,2} \Uparrow \mathcal{M}_{4,2} \Uparrow \mathcal{M}_{3,2} \Uparrow \mathcal{M}_{2,2} \Uparrow \mathcal{M}_{1,2}$.
- The FFT dag is obtained by pairwise composing many instances of $\mathcal{C}_2 = \mathcal{Q}_2$ (twelve instances in the figure).

As hinted at in the preceding description, the composition operation is associative, so we do not have to keep track of the order in which constituent dags are incorporated into a composite dag.

Lemma 4 ([16]) *The composition operation on dags is associative. That is, for all dags* $\mathcal{G}_1$, $\mathcal{G}_2$, $\mathcal{G}_3$, *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* $[\mathcal{G}_1 \Uparrow [\mathcal{G}_2 \Uparrow \mathcal{G}_3]]$.

One can garner intuition for the proof of Lemma 4 from the dags on Fig. 8.

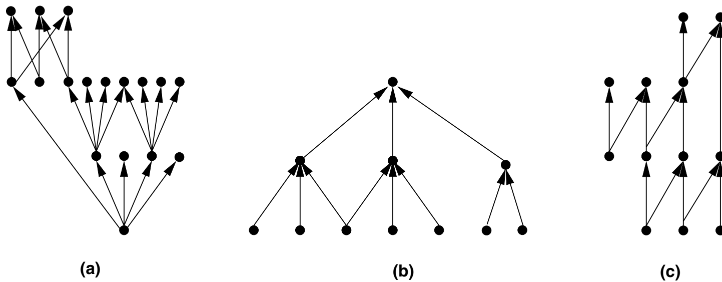

Fig. 8. *A sampler of composite dags, each of which admits an IC-optimal schedule.*

5.3 The Priority Relation $\succeq$

The next ingredient in our scheduling theory is the following relation on bipartite dags. This relation is the mechanism that we can often use to "inherit" an IC-optimal schedule for a composite dag from IC-optimal schedules for its constituents.

Let the disjoint bipartite dags $\mathcal{G}_1$ and $\mathcal{G}_2$, having s_1 and s_2 sources, respectively, admit the IC-optimal schedules Σ_1 and Σ_2, respectively. Say that the following inequalities hold.[8]

$$\begin{aligned}&(\forall x \in [0, s_1])\ (\forall y \in [0, s_2]) : \\ &E_{\Sigma_1}(x) + E_{\Sigma_2}(y) \ \leq\ E_{\Sigma_1}(\min\{s_1, x+y\}) + E_{\Sigma_2}(\max\{0, x+y-s_1\}).\end{aligned} \quad (3)$$

Then we say that $\mathcal{G}_1$ *has priority over* $\mathcal{G}_2$, denoted $\mathcal{G}_1 \succeq \mathcal{G}_2$.

By Lemma 3, the inequalities in (3) basically say that one never decreases IC quality by executing a source of $\mathcal{G}_1$, in preference to a source of $\mathcal{G}_2$, whenever possible.

It is important, both conceptually and algorithmically, that the relation $\succeq$ is transitive. This fact is a bit trickier to prove than one might think at first blush.

Lemma 5 ([16]) *The relation $\succeq$ on bipartite dags is transitive.*

One simple, but consequential application of Lemma 5 is:

Corollary 6 ([16]) *Let $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n$ be pairwise disjoint bipartite dags. If $\mathcal{G}_1 \succeq \mathcal{G}_2 \succeq \cdots \succeq \mathcal{G}_n$, then $\mathcal{G}_1 \succeq (\mathcal{G}_2 + \mathcal{G}_3 + \cdots + \mathcal{G}_n)$.*

5.4 Scheduling $\succeq$-Linear Compositions of Composite Dags

We arrive finally at the first major result of the theory, which provides the sought explanation for the structures of the proofs of Theorem 2 for the three families of dags. More importantly, this result gives structure to our quest for a decomposition-based scheduling theory.

Say that dag $\mathcal{G}$ is a $\succeq$-*linear composition* of the connected bipartite dags $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n$ if:

1. $\mathcal{G}$ is composite of type $\mathcal{G}_1 \Uparrow \mathcal{G}_2 \Uparrow \cdots \Uparrow \mathcal{G}_n$;
2. each $\mathcal{G}_i \succeq \mathcal{G}_{i+1}$, for all $i \in [1, n-1]$.

Dags that are $\succeq$-linear compositions inherit IC-optimal schedules from their constituents.

Theorem 7 ([16]) *Let $\mathcal{G}$ be a $\succeq$-linear composition of $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n$, where each $\mathcal{G}_i$ admits an IC-optimal schedule Σ_i. The schedule Σ for $\mathcal{G}$ that proceeds as follows is IC optimal.*

1. *Σ executes the nodes of $\mathcal{G}$ that correspond to sources of $\mathcal{G}_1$, in the order mandated by Σ_1, then the nodes that correspond to sources of $\mathcal{G}_2$, in the order mandated by Σ_2, and so on, for all $i \in [1, n]$.*
2. *Σ finally executes all sinks of $\mathcal{G}$ in any order.*

The proof of Theorem 7 essentially demonstrates that when a dag $\mathcal{G}$ is a $\succeq$-*linear composition*, then the priority relation $\succeq$ on $\mathcal{G}$'s bipartite constituents is compatible with the executional priorities that are inherent in $\mathcal{G}$'s being a dag.

[8] $[a, b]$ denotes the set of integers $\{a, a+1, \ldots, b\}$.

5.5 Scheduling Composite Dags Via Decomposition

The framework developed thus far in this section is descriptive rather than prescriptive. *If* a computation-dag $\mathcal{G}$ is constructed from bipartite building blocks via composition, and *if* we can identify the "blueprint" used to construct $\mathcal{G}$, and *if* the underlying building blocks are interrelated in a certain way, *then* the strategy prescribed in Theorem 7 produces an optimal schedule for $\mathcal{G}$. We now describe how the algorithmic challenge hidden in the preceding *if*'s is addressed in [16]: given a computation-dag $\mathcal{G}$, how does one apply the preceding framework to it? The algorithms we describe now attempt to decompose $\mathcal{G}$ in order to expose the structure needed to apply Theorem 7. We thereby derive IC-optimal schedules for a large variety of dags.

We describe a suite of algorithms that: (a) reduce any computation-dag $\mathcal{G}$ to its "transitive skeleton" $\mathcal{G}'$, a simplified version of $\mathcal{G}$ that shares the same set of optimal schedules; (b) decompose $\mathcal{G}'$ to determine whether or not it is constructed from bipartite building blocks via composition, thereby exposing a "blueprint" for $\mathcal{G}'$; (c) specify an optimal schedule for any such $\mathcal{G}'$ that is built from building blocks that are interrelated under $\succeq$.

Skeletonizing Input Dags For any dag $\mathcal{G}$ and nodes $u, v \in N_{\mathcal{G}}$, we write $u \Rightarrow_{\mathcal{G}} v$ to indicate that there is a path from u to v in $\mathcal{G}$. An arc $(u \rightarrow v) \in A_{\mathcal{G}}$ is a *shortcut* if there is a path $u \Rightarrow_{\mathcal{G}} v$ that does not include the arc. Of course, removing shortcuts from a dag does not alter internode connectivities. By removing all shortcuts from a dag $\mathcal{G}$, one obtains $\mathcal{G}$'s *(transitive) skeleton* (or, *transitive reduction*). This dag, which is unique, is the smallest subdag of $\mathcal{G}$ that shares $\mathcal{G}$'s transitive closure [5]; we call this dag $\sigma(\mathcal{G})$. One finds in [12] a polynomial-time algorithm that generates $\sigma(\mathcal{G})$ from $\mathcal{G}$. (In fact, a very simple algorithm suffices, that just removes, in turn, each arc $(u \rightarrow v)$ from $\mathcal{G}$ and tests if v is still accessible from u.)

Eliminating shortcuts is a critical first step in our decompsitional scheduling strategy, because dags that are compositions of bipartite dags have no shortcuts.

Since $\mathcal{G}$ shares its node-set with $\sigma(\mathcal{G})$, any schedule that executes one dag also executes the other. This is important because any schedule executes $\mathcal{G}$ as efficiently (in IC quality) as it executes $\sigma(\mathcal{G})$. A special case of this result appears in [19].

Theorem 8 ([16]) *A schedule Σ has the same IC quality when executing a dag $\mathcal{G}$ as when executing $\sigma(\mathcal{G})$. In particular, if Σ is IC optimal for $\sigma(\mathcal{G})$, then it is IC optimal for $\mathcal{G}$.*

Decomposing a Composite Dag Once we have a shortcut-free dag $\mathcal{G}$, we can start trying to decompose it, to find subdags whose composition yields $\mathcal{G}$. We now describe this process.

Selecting a constituent. We begin by selecting any constituent[9] of $\mathcal{G}$ all of whose sources are also sources of $\mathcal{G}$; call the selected constituent $\mathcal{B}_1$ (the notation emphasizing that $\mathcal{B}_1$ is *bipartite*).

> In Fig. 1: Every candidate $\mathcal{B}_1$ for the FFT dag comprises a copy of $\mathcal{C}_2 = \mathcal{Q}_2$ included in levels 2 and 3; every candidate for the reduction-tree comprises a copy of $\mathcal{M}_{1,2}$; the unique candidate for the reduction-mesh comprises $\mathcal{M}_{4,2}$.

Detaching a constituent. We "detach" $\mathcal{B}_1$ from $\mathcal{G}$ by deleting the nodes of $\mathcal{G}$ that correspond to sources of $\mathcal{B}_1$, all incident arcs, and all resulting isolated sinks. We thereby replace $\mathcal{G}$ with a pair of dags $\langle \mathcal{B}_1, \mathcal{G}' \rangle$, where $\mathcal{G}'$ is the remnant of $\mathcal{G}$ remaining after $\mathcal{B}_1$ is detached.

If the remnant $\mathcal{G}'$ is not empty, then we continue the process of selection and detachment. If $\mathcal{G}$ was a composition of bipartite dags, then we produce a sequence of the form

$$\mathcal{G} \implies \langle \mathcal{B}_1, \mathcal{G}' \rangle \implies \langle \mathcal{B}_1, \langle \mathcal{B}_2, \mathcal{G}'' \rangle \rangle \implies \langle \mathcal{B}_1, \langle \mathcal{B}_2, \langle \mathcal{B}_3, \mathcal{G}''' \rangle \rangle \rangle \implies \cdots,$$

that leads ultimately to a complete decomposition of $\mathcal{G}$ into a sequence comprising all of its constituents: $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n$.

We claim that the described process does, indeed, recognize whether or not $\mathcal{G}$ is a composite dag, and, if so, it produces the constituents from which $\mathcal{G}$ is composed (possibly, of course, in an order that differs from their original order of composition).

Theorem 9 ([16]) *Let the dag $\mathcal{G}$ be composite of type $\mathcal{G}_1 \Uparrow \mathcal{G}_2 \Uparrow \cdots \Uparrow \mathcal{G}_n$. The decomposition process produces a sequence $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n$ of constituents of $\mathcal{G}$ such that:*

- *$\mathcal{G}$ is composite of type $\mathcal{B}_1 \Uparrow \mathcal{B}_2 \Uparrow \cdots \Uparrow \mathcal{B}_n$;*
- *$\{\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n\} = \{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n\}$.*

It is fruitful to construct a *super-dag* that abstracts a dag $\mathcal{G}$'s structure, as exposed by the decomposition process. This super-dag, which we denote $\mathcal{S}(\mathcal{B}_1 \Uparrow \cdots \Uparrow \mathcal{B}_n)$, has the constituents $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n$ as its nodes and has an arc from each constituent $\mathcal{B}_i$ to each of the constituents that it is detached from during the decomposition. Fig. 9 depicts the super-dag obtained from decomposing the 3-dimensional FFT dag. Easily that the linearization $\mathcal{B}_1, \ldots, \mathcal{B}_n$ produced by the described decomposition process is a *topological sort* [5] of the super-dag $\mathcal{S}(\mathcal{B}_1 \Uparrow \cdots \Uparrow \mathcal{B}_n)$.

Scheduling a Composite Dag Via Its Super-Dag Our remaining challenge is to determine, given a super-dag $\mathcal{S}(\mathcal{B}_1 \Uparrow \cdots \Uparrow \mathcal{B}_n)$ that is produced by our decomposition process, whether or not there is a topological sort of the super-dag that linearizes the supernodes in an order that honors relation $\succeq$. We now present sufficient conditions for this to occur, verified via a linearization algorithm.

[9] Recall the technical definition of "constituent" from Section 2.1.

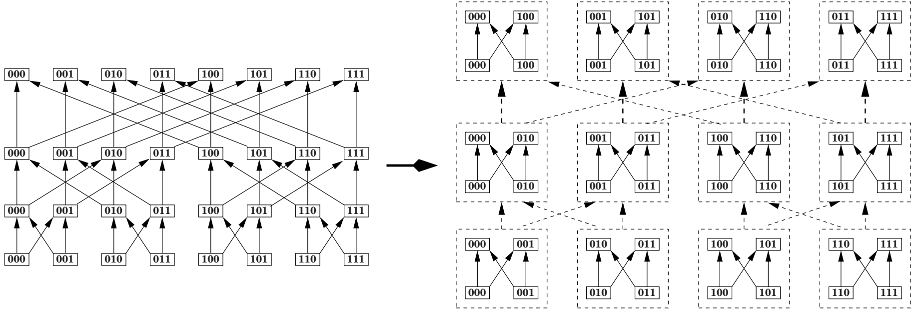

Fig. 9. *The 3-dimensional FFT dag and its associated super-dag.*

Theorem 10 ([16]) *Say that the dag $\mathcal{G}$ is composite of type $\mathcal{B}_1 \Uparrow \cdots \Uparrow \mathcal{B}_n$ and that, for each pair of constituents, $\mathcal{B}_i$, $\mathcal{B}_j$ with $i \neq j$, either $\mathcal{B}_i \succeq \mathcal{B}_j$ or $\mathcal{B}_j \succeq \mathcal{B}_i$. Then $\mathcal{G}$ is a $\succeq$-linear composition whenever the following holds.*

Whenever $\mathcal{B}_j$ is a child of $\mathcal{B}_i$ in $\mathcal{S}(\mathcal{B}_1 \Uparrow \cdots \Uparrow \mathcal{B}_n)$, we have $\mathcal{B}_i \succeq \mathcal{B}_j$.

Theorem 10 is proved via the following algorithm that determines whether or not $\mathcal{G}$ is a $\succeq$-linear composition of the $\mathcal{B}_i$.

1. We begin with a topological sort, $\widehat{\mathcal{B}} \stackrel{\text{def}}{=} \mathcal{B}_{\alpha(1)}, \ldots, \mathcal{B}_{\alpha(n)}$ of $\mathcal{S}_{\mathcal{G}} \stackrel{\text{def}}{=} \mathcal{S}(\mathcal{B}_1 \Uparrow \cdots \Uparrow \mathcal{B}_n)$.
2. We invoke the hypothesis that $\succeq$ is a (weak) order on the $\mathcal{B}_i$'s to reorder $\widehat{\mathcal{B}}$ according to $\succeq$, using a *stable*[10] comparison sort.

Let $\boldsymbol{\mathcal{B}} \stackrel{\text{def}}{=} \mathcal{B}_{\beta(1)} \succeq \cdots \succeq \mathcal{B}_{\beta(n)}$ be the linearization of $\mathcal{S}_{\mathcal{G}}$ produced by the sort. We claim that $\boldsymbol{\mathcal{B}}$ is also a topological sort of $\mathcal{S}_{\mathcal{G}}$. This follows easily because we start with a topological sort of $\mathcal{S}_{\mathcal{G}}$ and employ a stable sort on relation $\succeq$. We conclude that $\mathcal{G}$ is composite of type $\mathcal{B}_{\beta(1)} \Uparrow \cdots \Uparrow \mathcal{B}_{\beta(n)}$. In other words, $\boldsymbol{\mathcal{B}}$ is the desired $\succeq$-linearization of $\mathcal{G}$.

Once we have the decomposition $\boldsymbol{\mathcal{B}}$, we can invoke Theorem 7 to obtain an IC-optimal schedule for $\mathcal{G}$.

6 A Batched Approach to Scheduling

Our development of a dag-scheduling theory for IC is still ongoing: we are making steady progress in both refining and extending the work described in Section 5. Yet, the stringent demands of IC optimality, as reflected in the requirement that a schedule maximize the number of ELIGIBLE nodes *at every step* of a computation, guarantees the existence of simple dags that admit no IC-optimal schedule (cf. Section 3); hence, they preclude this theory from ever being comprehensive.

[10] Stability means that if $\mathcal{B}_i \succeq \mathcal{B}_j$ and $\mathcal{B}_j \succeq \mathcal{B}_i$, then the sort maintains the original positions of $\mathcal{B}_i$ and $\mathcal{B}_j$.

Responding to this fact, we are investigating alternative formulations of the IC-scheduling problem. We have developed one such in [15], and we describe its rudiments in this section.

In the *batched* version of the IC Pebble Game, which abstracts the batched version of the IC-scheduling problem, the Server does not respond to individual requests by Clients as they come in. Instead, it services requests at fixed intervals, hence responds to batches of requests rather than individual ones. This formulation of IC scheduling simplifies the scheduling problem along one axis, while complicating it along another. We now focus on optimizing the production of ELIGIBLE nodes:

1. for a single step of the computation, rather than uniformly for all steps;
2. while executing r (perforce, ELIGIBLE) nodes as a batch, rather than a single node.

The (algorithmic) greed built into this version of IC-scheduling—by the first condition—ensures that there is an optimal solution to every instance of the problem. The complication built into this version—by the second condition—turns out to endow the challenge of finding this optimal solution with the likely computational intractability of NP-hardness. (The solution is easy to find when $r = 1$.) Fig. 10 presents the rules of the 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.*/

Fig. 10. *The Rules of the Batch-IC Pebble Game*

As in earlier sections, we call a node ELIGIBLE (resp., EXECUTED) when it contains an ELIGIBLE (resp., an EXECUTED) pebble, and we talk about executing nodes rather than tasks.

The Batch-IC scheduling problem (BICSO). Our goal is to play the Batch-IC Pebble Game in a way that maximizes the number of ELIGIBLE pebbles on $\mathcal{G}$ at every step of the Game. That is, for each step t of a play of the Game on a dag $\mathcal{G}$, if there are currently e_t ELIGIBLE tasks, and if r_t Clients request tasks, then we want the Server to execute a set of $\min\{e_t, r_t\}$ ELIGIBLE nodes

that will result in the largest possible number of ELIGIBLE tasks 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 $[1, |E|]$.

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

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

In contrast to the search for IC-optimal schedules for dags, every instance of BICSO can be solved! The only question is how hard it is computationally to find a solution. Unfortunately, solving BICSO is likely computationally intractable, even for dags of quite restricted structure.

Theorem 11 ([15]) *BICSO is NP-hard, even when restricted to bipartite dags.*

Of course, results such as Theorem 11 automatically trigger a search for special classes of dags that can be scheduled optimally in polynomial time. Not surprisingly, bipartite tree-dags—*and compositions thereof*—are the first such class that we discovered. The algorithm guaranteed by the following theorem contains a dynamic program as a central component.

Theorem 12 ([15]) *There is a polynomial-time algorithm Σ_{tree} that solves BICSO for any bipartite tree-dag $\mathcal{T}$.*

Theorem 12 is actually more textured than it seems to be at first. On the optimistic side: The theorem gives us more scheduling power than is immediately apparent. Specifically, we show in [15] how to build upon the theorem to solve BICSO for any *composition* of bipartite tree-dags. This is important, since compositions of such tree-dags need not be either leveled or (in their undirected incarnations) cycle-free. On the less-optimistic side: Algorithm Σ_{tree} is computationally rather inefficient: its timing polynomial has high degree. In response, we have sought nontrivial classes of dags for which we could solve BICSO *efficiently*, even if the solution was only *approximate*. We use the word "approximate" here in its usual technical sense: we insist that the number of ELIGIBLE nodes produced by the scheduling algorithm in response to r requests be within a predictable factor of the maximum possible number, given the then-current number of ELIGIBLE nodes.

Our initial success in this quest involved the family **E** of *bipartite expansive-dags*. Each such dag $\mathcal{E}$ 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 at most φ_v other children. Expansive-dags epitomize computations that are "expansive" but may have complex interdependencies. A simple algorithm that we call Algorithm $\Sigma_{\rm exp}$ approximates a solution to BICSO for the family **E**.

Algorithm $\Sigma_{\rm exp}$ implements the following natural, fast heuristic for scheduling a dag $\mathcal{E} \in \mathbf{E}$. For each source v of $\mathcal{E}$, say that there are φ_v nodes that have v as their sole parent and ψ_v nodes that have other parents also. If there are r requests for ELIGIBLE nodes at time t, then $\Sigma_{\rm exp}$ selects the r ELIGIBLE nodes that have the largest associated integers φ_v. Of course, this greedy heuristic may deviate from optimality because it ignores the "bonuses" that may arise from executing ELIGIBLE nodes that are siblings in $\mathcal{E}$, but it does come close to optimality.

Theorem 13 ([15]) *For any instance* $\imath = \langle \mathcal{E}, X, E; r\rangle$ *of BICSO, where* $\mathcal{E} \in \mathbf{E}$*, Algorithm* $\Sigma_{\rm exp}$ *will, in time* $O(|E|)$*, find a solution to BICSO, whose increase in the number of* ELIGIBLE *nodes is at least one-fourth the optimal increase.*

Work continues in trying to extend both Theorems 12 and 13, by expanding the classes of dags for which we can tractably solve, or quickly approximate a solution to BICSO.

7 Conclusions and Projections

We have described two related pebble games that abstract the problem of scheduling computation-dags for Internet-based computing. Both games place an ELIGIBLE pebble (which represents a task's being eligible for execution) on every node all of whose parents contain EXECUTED pebbles (which represents a task's having been executed). At each step: one game selects a single ELIGIBLE pebble to replace with an EXECUTED pebble; the other selects a variable number of ELIGIBLE pebbles to replace (based on the input). With both games, the placement of a new EXECUTED pebble may cause the placement of new ELIGIBLE pebbles. Both games strive, under somewhat different rules, to maximize the number of nodes that contain ELIGIBLE pebbles.

The **IC Pebble Game** takes as input a dag $\mathcal{G}$. It seeks an execution schedule for $\mathcal{G}$ that maximizes the number of nodes that hold ELIGIBLE pebbles *at every step of the game*. We have described the underpinnings of a theory of scheduling under the IC Pebble Game, which builds on the decomposition of an input dag $\mathcal{G}$ into bipartite "building-block" dags. When the decomposition exposes $\mathcal{G}$ to be a *composition of building blocks* that are suitably iterrelated under the *priority relation*, then the theory generates a schedule for $\mathcal{G}$ that is optimal. Ongoing work, some in collaboration with G. Cordasco (U. Salerno), seeks to expand the range of dags that the theory can schedule optimally, both by expanding the repertoire of building blocks that it can deal with [4] and by extending the scope of the priority relation. Other work, some in collaboration with I. Foster and

M. Wilde of Argonne National Laboratory, seeks to assess the impact of the emerging theory on a real IC project.

The **Batched-IC Pebble Game** takes as input a dag $\mathcal{G}$, some e of whose nodes hold ELIGIBLE pebbles, and an integer $r \leq e$ of "requests." It seeks to find a set of r nodes currently holding ELIGIBLE pebbles such that executing those nodes will allow the placement of maximally many new ELIGIBLE pebbles. Results obtained thus far have shown the problem of solving instances of this problem optimally to be NP-hard (with the decision version being NP-complete). The problem is solvable in polynomial time for composite tree-dags, yet not efficiently. The problem is efficiently approximable for certain special classes of dags. Ongoing work here is delving further into the search for efficiently schedulable classes of dags and efficiently approximable classes.

For both pebble games, attempts are also being made to assess the quality of schedules produced by simple heuristics.

Acknowledgment. The research of A. Rosenberg was supported in part by NSF Grant CCF-0342417. It is a pleasure to acknowledge the many contributions of our collaborator, Matt Yurkewych of the University of Massachusetts Amherst.

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. G. Cordasco, G. Malewicz, A.L. Rosenberg (2005): On scheduling expansive and reductive dags for Internet-based computing. Submitted for publication.
5. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein (2001): *Introduction to Algorithms (2nd ed.).* MIT Press, Cambridge, MA.
6. I. Foster and C. Kesselman [eds.] (2004): *The Grid: Blueprint for a New Computing Infrastructure* (2nd ed.) Morgan-Kaufmann, San Francisco, CA.
7. I. Foster, C. Kesselman, S. Tuecke (2001): The anatomy of the Grid: enabling scalable virtual organizations. *Intl. J. High Performance Computing Applications 15*, 200–222.
8. A. Gerasoulis and T. Yang (1992): A comparison of clustering heuristics for scheduling dags on multiprocessors. *J. Parallel Distr. Comput. 16*, 276–291.
9. L. He, Z. Han, H. Jin, L. Pan, S. Li (2000): DAG-based parallel real time task scheduling algorithm on a cluster. *Intl. Conf. on Parallel and Distr. Processing Techniques and Applications*, 437–443.
10. J.-W. Hong and H.T. Kung (1981): I/O complexity: the red-blue pebble game. *13th ACM Symp. on Theory of Computing*, 326–333.
11. J.E. Hopcroft, W. Paul, L.G. Valiant (1977): On time versus space. *J. ACM 24*, 332–337.
12. H.T. Hsu (1975): An algorithm for finding a minimal equivalent graph of a digraph. *J. ACM 22*, 11–16.

13. D. Kondo, H. Casanova, E. Wing, F. Berman (2002): Models and scheduling guidelines for global computing applications. *Intl. Parallel and Distr. Processing Symp.*, 79.
14. E. Korpela, D. Werthimer, D. Anderson, J. Cobb, M. Lebofsky (2000): SETI@home: massively distributed computing for SETI. In *Computing in Science and Engineering* (P.F. Dubois, Ed.) IEEE Computer Soc. Press, Los Alamitos, CA.
15. G. Malewicz and A.L. Rosenberg (2005): On batch-scheduling dags for Internet-based computing. *Euro-Par 2005*. In *Lecture Notes in Computer Science 3648*, Springer-Verlag, Berlin, 262–271.
16. G. Malewicz, A.L. Rosenberg, M. Yurkewych (2005): Toward a theory for scheduling dags in Internet-based computing. *IEEE Trans. Comput.*, to appear. See also: On scheduling complex dags for Internet-based computing. *Intl. Parallel and Distributed Processing Symp.*, 2005.
17. M.S. Paterson, C.E. Hewitt (1970): Comparative schematology. *Project MAC Conf. on Concurrent Systems and Parallel Computation*, ACM Press, 119–127.
18. N.J. Pippenger (1980): Pebbling. In *5th IBM Symp. on Math. Foundations of Computer Science*.
19. A.L. Rosenberg (2004): On scheduling mesh-structured computations for Internet-based computing. *IEEE Trans. Comput. 53*, 1176–1186.
20. A.L. Rosenberg and I.H. Sudborough (1983): Bandwidth and pebbling. *Computing 31*, 115–139.
21. A.L. Rosenberg and M. Yurkewych (2005): Guidelines for scheduling some common computation-dags for Internet-based computing. *IEEE Trans. Comput. 54*, 428–438.
22. X.-H. Sun and M. Wu (2003): Grid Harvest Service: a system for long-term, application-level task scheduling. *IEEE Intl. Parallel and Distributed Processing Symp.*, 25.
23. S.W. White and D.C. Torney (1993): Use of a workstation cluster for the physical mapping of chromosomes. *SIAM NEWS*, March, 1993, 14–17.