Guidelines for Scheduling Some Common Computation-Dags for Internet-Based Computing

Arnold L. Rosenberg, *Fellow*, *IEEE*, and Matthew Yurkewych

Abstract—A "pebble game" is developed to model the process of scheduling computation-dags for Internet-based computing (IC, for short). Strategies are derived for scheduling three common, significant families of such dags for IC: *reduction-meshes*, which represent (the intertask dependencies of) computations that can be performed by "up-sweeps" of meshes; *reduction-trees*, which represent "accumulative" computations that can be performed by "up-sweeps" of trees; and *FFT* (Fast Fourier Transform) *dags*, which represent a large variety of convolutional computations. Two criteria are used to assess the quality of a schedule: its memory requirements and its rate of producing tasks that are eligible for allocation to remote clients. These criteria are important because of, respectively, the typically enormous sizes of IC computations and the typical temporal unpredictability of remote clients in IC. In particular, a high production rate of eligible tasks decreases a computation's vulnerability to the gridlock that can occur when a computation stalls pending the return of intermediate results by remote clients. Under idealized assumptions, the schedules derived are optimal in the rate of producing eligible tasks and are either exactly or approximately optimal in memory requirements.

Index Terms—Internet-based computing, grid computing, global computing, Web computing, scheduling, reduction computations, convolutional computations, mesh-structured computations, tree-structured computations.

1 Introduction

Advancing technology has rendered the Internet a viable medium for *collaborative computing*, the use of (many) independent computers to solve a single computational problem. Indeed, Internet-based computing (*IC*, for short), in its several guises—including Grid computing (see, e.g., [5], [9], [10]), global computing (see, e.g., [6]), and Web computing (see, e.g., [17], [21], [24])—promises to become a viable computing platform for many computational problems that are prohibitively consumptive of resources on traditional computing platforms. This paper is devoted to extending the study initiated in [28] of scheduling computations having intertask dependencies for modalities of IC in which a computation is fed by a *(master) Server* to *(remote) Clients*; Grid computing and Web computing typically proceed in this manner.

1.1 The Foci of Our Study

The danger of gridlock in IC. The problem studied in [28] and here results from two sources of *unpredictability* in IC. First, since all communication is over the Internet, it is impossible to predict its timing with complete assurance. Second, the remote Clients that cooperate in an IC computation typically contract to produce results *eventually*, but with no time guarantees. (Indeed, in some modalities of IC, remote Clients cannot be trusted to return results *ever*.) The absence of task completion-and-return timing guarantees from remote clients in IC is largely an annoyance when the tasks comprising the shared workload are mutually independent: The (usually massive) size of the workload ensures that some task is always eligible for execution—hence, for allocation to a Client. However, when the workload's tasks have interdependencies that constrain their order of execution, the lack of timing guarantees creates a nontrivial scheduling challenge, which is discussed from a systems perspective in [20]. Specifically, such dependencies can potentially engender *gridlock* when no new tasks can be allocated for an indeterminate period, pending the execution of already allocated tasks. A variety of "safety devices" have been developed to mitigate this danger (although no device can eliminate it since any "backup" remote Client(s) can be as dilatory as the primary one). Two popular "safety devices" are:

- allocating tasks to multiple Clients—a technique used, e.g., in SETI@home [21];
- deadline-triggered reallocation of tasks—a technique described, e.g., in [5], [20].

One weakness of the former device is its significantly thinning out the remote workforce. More importantly, one weakness shared by both devices is their requiring a (reasonably) reliable model of Clients' computing behaviors. In order to generate such a model, many IC projects—see, e.g., [5], [20], [30], as well as the IC-enabling software developed by Entropia, Inc. (http://www.entropia.com)— monitor either the past history of remote Clients or their current computational capabilities or both. While the resulting snapshots of Clients yield no guarantees of future performance, they at least afford the

• *The authors are with the Department of Computer Science, University of Massachusetts Amherst, Amherst, MA 01003.*
E-mail: {rsnbrg, yurk}@cs.umass.edu.

Manuscript received 24 Mar. 2004; revised 12 July 2004; accepted 2 Sept. 2004; published online 16 Mar. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0106-0304.

Server a basis for estimating such performance. Our study illustrates how the information gleaned from monitoring can be used to lessen the likelihood of gridlock. We proceed under the idealized assumption that such monitoring yields sufficiently accurate predictions of Clients' future performance that the Server can allocate eligible tasks to Clients in an order that makes it likely that tasks will be executed in the order of their allocation. Under this assumption, we use the abstract IC-scheduling (pebble game) model of [28] to seek schedules that decrease the likelihood of gridlock. As in much of the literature on scheduling algorithms (cf. [13], [15]), the model focuses on computations whose intertask dependencies are structured as directed-acyclic graphs (*dags*, for short).[1] Our avenue for minimizing the danger of gridlock is to seek schedules that produce allocation-eligible tasks as fast as possible.

The memory cost of IC. Memory requirements are a significant concern in IC, since computation-dags earmarked for IC are likely to be enormous, and their constituent tasks are likely to be quite large in order to amortize the substantial cost of communicating across the Internet. It is, therefore, essential that IC-oriented schedules for computation-dags be conservative of memory.

Other challenges. The present study focuses on temporal challenges in IC. At least as important are the manifold security-related challenges, which are orthogonal to our study and which we do not address. Three citations exemplify such challenges. Numerous sources use "reputation" to address the danger of malicious content being spread in P2P systems; see, e.g., [1]. For special types of computations, [14] proposes server-side methods that ensure the integrity of clients' submissions. Responding to the problem of malicious anonymous "volunteers" in SETI@home-type environments ([21]), [27] presents an algorithmic approach to matching "volunteers" with their results in such systems.

1.2 The Contributions of Our Study

Our study employs the *Internet-Computing Pebble Game* of [28] to study the problem of scheduling dags for IC. We extend the study in [28] in two ways. 1) We seek schedules that produce eligible tasks optimally fast for three new families of computation-dags that arise in a broad range of significant applications. 2) We add memory conservation to the quality criteria for schedules. The three dag families we study here are:

1. *Reduction-meshes* (or, *pyramid dags*) [7], which represent (say, scientific and engineering) computations whose intertask dependencies have the structure of meshes and that can be executed via sweeps up the meshes.
2. *Reduction-trees* [11], [19], [25], which represent computations whose intertask dependencies have the structure of trees and that can be executed via sweeps up the trees.
3. *FFT dags* [8], [16], which represent convolution-structured computations, including the eponymous Fast-Fourier Transform.

1. Dags and associated notions are defined in Section 2.

Under the idealized assumption that task-execution order follows task-allocation order, we characterize optimal gridlock-avoiding schedules for each dag family and devise at least one concrete optimal schedule. Additionally, we exhibit schedules that either exactly or approximately minimize the memory requirements of computations, even while they minimize the likelihood of gridlock.

Of course, even within our idealized setting, the schedules that are optimal under our formal model neither eliminate the danger of gridlock nor obviate "safety devices" such as those mentioned earlier. Rather, by identifying the characteristics that guarantee optimality in the idealized setting, we provide guidelines for provably enhancing the effectiveness of the "safety devices." And, importantly, these guidelines prescribe actions that are under the control of the IC Server and are independent of the behavior of the Clients! (The proposed scheduling strategy does not address the heterogeneity of Clients explicitly. We assume, rather, that, via its monitoring of Clients, the Server allocates tasks to Clients in a way that recognizes and accommodates their heterogeneity.)

1.3 Related Work

The study most closely related to ours is its immediate predecessor [28]. That source developed the pebble game we use to study IC scheduling. It then used that model to identify classes of schedules that are either exactly or approximately optimal in production rate of allocation-eligible tasks, for computation-dags having the structure of evolving meshes or their close relatives. The main results in [28] identify classes of schedules that are optimal for two-dimensional evolving meshes and within a constant factor of optimal for evolving meshes of higher (finite) dimensionalities. Our study has also benefited from a number of studies that use somewhat different scheduling models to study the memory costs of schedules. The scheduling problem for reduction-meshes is considered in [7] (where reduction-meshes are called "pyramid dags"). Given the ubiquity of tree-structured computations, it is not surprising that there have been myriad studies of scheduling problems on reduction-trees. One of the earliest such studies, [25], focuses on scheduling complete reduction-trees in a way that minimizes memory cost. The study in [3] shows how the memory requirements of complete reduction-tree computations decrease as one proceeds from an "eager" scheduling strategy toward a "lazy" one. The study in [11] minimizes the parallel makespans of complete reduction-tree computations. Many studies, e.g., [12], [18], [22], study the computational complexity of various scheduling problems for reduction-trees. The importance of the Fast Fourier Transform Algorithm [8] has led to a massive body of literature that relates implicitly to FFT dags. The importance of the FFT dag as a network has led to an equally massive body of literature, often under the rubric "butterfly networks," the network-oriented name of the dags. Butterfly networks have truly remarkable properties related to permuting data [2] and to interconnecting communicating sources [4], [23], [26]. Within the context of the present study, the most relevant prior study is [16], which focuses on scheduling (large) FFT dags in a way that minimizes the number of data transfers between adjacent

levels of the memory hierarchy. The differences between our goals and those of the cited sources demand quite different techniques of analysis.

2 A Model for Executing Dags on the Internet

In this section, we present the formal entities that underlie our study: computation-dags and the IC Pebble Game. Much of this material is excerpted from [28].

2.1 Computation-Dags

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, the preceding path cannot have $u_1 = u_n$. 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)*.

The current study focuses on three common, significant classes of computation-dags.

2.1.1 Reduction-Meshes

The *ℓ-level reduction-mesh* $\mathcal{M}_\ell$ is the dag whose nodes comprise the set of ordered pairs of integers $\{\langle x, y\rangle \mid 0 \le x + y < \ell\}$. Each node $v = \langle x, y\rangle$ of $\mathcal{M}_\ell$ has an arc to node $\langle x-1, y\rangle$ whenever $x > 0$ and an arc to node $\langle x, y-1\rangle$ whenever $y > 0$; these account for all of $\mathcal{M}_\ell$'s arcs. The integer $x + y$ is the *level* of node $\langle x, y\rangle$. $\mathcal{M}_\ell$'s ℓ source nodes are the nodes at level $\ell - 1$; $\mathcal{M}_\ell$'s unique sink node is node $\langle 0, 0\rangle$, the sole occupant of level 0. Fig. 1 depicts $\mathcal{M}_5$.

2.1.2 Reduction-Trees

A *reduction-tree* $\mathcal{T}$ is a dag whose nodes comprise a finite *full, prefix-closed* set S of binary strings. Let x be an arbitrary (possibly null) binary string and β an arbitrary bit: $\beta \in \{0, 1\}$. By "full," we mean that $x\beta \in S$ whenever $x\overline{\beta} \in S$;[2] by "prefix-closed," we mean that $x \in S$ whenever $x\beta \in S$. There is an arc from each node $x\beta \in S$ to node x; all arcs of $\mathcal{T}$ arise from such prefix-pairs. The set of nodes of $\mathcal{T}$ that are strings of length $l \ge 0$ comprise *level* l of $\mathcal{T}$. $\mathcal{T}$'s *root* λ, the null string, is the sole node at level 0 and is $\mathcal{T}$'s unique sink; $\mathcal{T}$'s source nodes are called *leaves*. See Fig. 2.

Complete reduction-trees merit special attention since they model computations that occur in so many applications. In a complete reduction-tree $\mathcal{T}$, all leaf-to-root paths have the same length; equivalently, each level l of $\mathcal{T}$ has exactly 2^l nodes. $\mathcal{T}$'s largest level-number is its *height*; we denote the height-h complete reduction-tree by $\mathcal{T}_h$.

2. For $\beta \in \{0, 1\}$, $\overline{\beta} = 1 - \beta$.

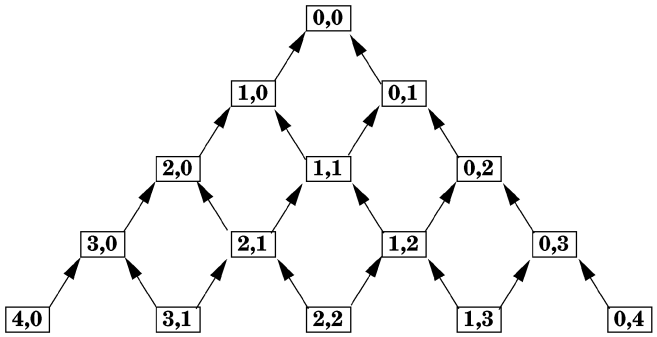

Fig. 1. The 5-level reduction-mesh $\mathcal{M}_5$.

2.1.3 FFT Dags

The *d-dimensional FFT dag* $\mathcal{F}_d$ exposes the data dependencies of the 2^d-input *F*ast *F*ourier *T*ransform algorithm [8]. Each node v of $\mathcal{F}_d$ has the form $v = \langle l, x\rangle$: $l \in \{0, 1, \ldots, d\}$ specifies v's *level*; x is a length-d binary string that specifies v's "position" within level l. Each arc of $\mathcal{F}_d$ connects a node at some level $l > 0$ to a node at level $l - 1$; specifically, each node $\langle l, x\beta y\rangle$ of $\mathcal{F}_d$ has arcs to nodes $\langle l-1, x0y\rangle$ and $\langle l-1, x1y\rangle$. Fig. 3 depicts $\mathcal{F}_3$, highlighting some of the *butterflies* that give FFT dags their network-oriented name. Each butterfly of $\mathcal{F}_d$ is the induced subdag on a quadruple of nodes:

$$\begin{array}{ll} \langle l, \beta_0\beta_1\cdots\beta_{d-l}\cdots\beta_{d-1}\rangle & \langle l, \beta_0\beta_1\cdots\overline{\beta}_{d-l}\cdots\beta_{d-1}\rangle \\ \langle l-1, \beta_0\beta_1\cdots\beta_{d-l}\cdots\beta_{d-1}\rangle & \langle l-1, \beta_0\beta_1\cdots\overline{\beta}_{d-l}\cdots\beta_{d-1}\rangle. \end{array} \tag{2.1}$$

$\mathcal{F}_d$ has arcs from both of the level-l nodes in (2.1) to both of the level-$(l-1)$ nodes; these are all of the out-arcs from the level-l nodes and all of the in-arcs to the level-$(l-1)$ nodes. $\mathcal{F}_d$'s level-d nodes are its sources; its level-0 nodes are its sinks.

2.2 The Internet-Computing Pebble Game

A number of so-called *pebble games* on dags have proven, over the course of several decades, to yield elegant formal analogues of a variety of problems related to scheduling computation-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 [28], whose structure derives from the "no recomputation allowed" pebble game of [29].

2.2.1 The Rules of the Game

The IC Pebble Game on a computation-dag $\mathcal{G}$ involves one player S, the *Server*, and an indeterminate number of players $C_1, C_2, \ldots$, the *Clients*. The Server has access to unlimited supplies of three types of pebbles: EBU (for "eligible-but-unallocated") pebbles, EAA (for "eligible-and-allocated") pebbles, and XEQ (for "executed") pebbles. The Game's moves reflect the successive stages in the "life-cycle" of a task

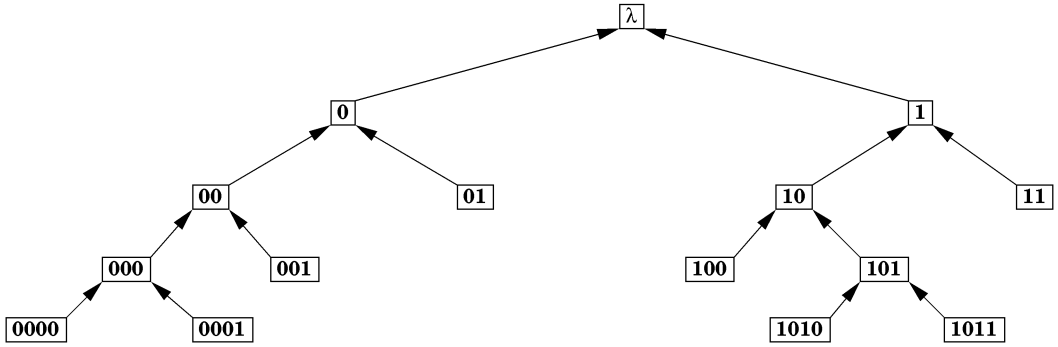

Fig. 2. A sample reduction-tree with eight leaves and five levels.

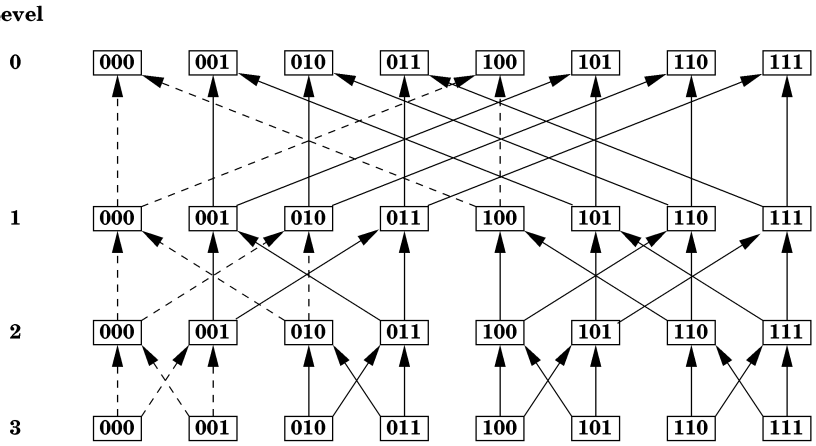

Fig. 3. The three-dimensional FFT dag $\mathcal{F}_3$. The leftmost "butterfly" (c.f. (2.1)) at each level is highlighted via dashed arcs.

in a computation, from eligibility for execution through actual execution. We now present the rules of the Game; these are illustrated and explained in greater detail in [28]. The reader should note how the moves of the Game—notably the Server's dependence on approaches by Clients—expose the danger of a play's being stalled indefinitely by dilatory Clients (what we called "gridlock" in the Introduction).

The I-C Pebble Game

Rule 1. S begins by placing an EBU pebble on every unpebbled source node of $\mathcal{G}$.

/*Unexecuted source nodes are always eligible for execution, having no parents upon whose prior execution they depend.*/

Rule 2. Say that Client C_i approaches S requesting a task. If C_i has previously been allocated a task that it has not completed, then C_i's request is ignored; else, the following occurs:

1. If at least one node of $\mathcal{G}$ contains an EBU pebble, then S gives C_i the task corresponding to one such node and replaces that node's pebble by an EAA pebble.
2. If no node of $\mathcal{G}$ contains an EBU pebble, then C_i is told to withdraw its request, and this move is a no-op.

Rule 3. When a Client returns (the results from) a task-node, S replaces that task-node's EAA pebble by an XEQ pebble. S then places an EBU pebble on each unpebbled node of $\mathcal{G}$ all of whose parents contain XEQ pebbles.

Rule 4. S's goal is ensure that every node of $\mathcal{G}$ *eventually* contains an XEQ pebble.

/*This goal makes sense for every dag $\mathcal{G}$. When $\mathcal{G}$ is infinite, such a modest goal is unavoidable since there may be infinite plays on $\mathcal{G}$ that never pebble certain nodes.*/

2.2.2 The IC Quality of a Play of the Game

As discussed earlier, our goal is to determine how to play the IC Pebble Game in a way that maximizes the likelihood that the Server has a task to allocate whenever it is approached by a Client. Within the framework of the Game, this intuitive goal translates into the formal goal of maximizing the number of nodes that hold EBU pebbles at every step. Hence, the *IC quality of a play of the Game* measures the extent to which the play maximizes the placement rate of EBU pebbles. We term this quality *IC optimality* when the rate is indeed maximized. As in [28], we approach a formal measure of IC quality via a (benign, as argued below) assumption that allows us to simplify our formal framework, at least for large classes of computation-dags, including the ones studied here. We thereby enhance the analyzability of the Game, while retaining its basic structure. Importantly, *our assumption allows a Server to focus on issues that are under its control, rather than depending on Clients' (unpredictable) behavior.*

Our assumption builds on the fact that careful monitoring of Clients, as described in [5], [20], [30] and at http://www.entropia.com, allows a Server to approximate a

situation wherein the (temporal) unpredictability of Clients affects the timing, but not the order of task executions.

Simplifying assumption. *Tasks are executed in the same order as they are allocated.*

Rationale. In the presence of the just-mentioned monitoring of Clients, the Server can enhance the likelihood of, even if not guarantee, the desired execution order.

We see in subsequent sections that this simplification allows the Server an easy avenue for playing the IC Pebble Game optimally, at least for the dag families studied here. It also allows us to simplify the repertoire of pebbles used to play the IC Pebble Game. Specifically, we henceforth *ignore the distinction between* EAA *and* XEQ *pebbles, lumping both together henceforth as* EXECUTED *pebbles*. We then enhance legibility by henceforth referring to EBU pebbles as "ELIGIBLE" pebbles and to a node that holds a pebble of type $T \in$ {ELIGIBLE, EXECUTED} as a "T node."

The preceding simplifications afford us the following conceptually simple, mathematically tractable formalization of our informal scheduling goal. For each step t of a play of the IC Pebble Game on a dag $\mathcal{G}$, let $E^{(t)}$ denote the number of ELIGIBLE pebbles on $\mathcal{G}$'s nodes at step t. (Note that, under our simplifying assumption, the analogous count, $X^{(t)}$, of EXECUTED pebbles at step t satisfies $X^{(t)} \equiv t$ because one new node is executed at each step of the simplified Game.)

We measure the **IC 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.*

2.2.3 The Memory Cost of a Play of the Game

As noted earlier, memory requirements are a significant concern in IC since computation-dags are likely to be massive and their constituent tasks are likely to be quite large. The IC Pebble Game admits a simple measure of memory cost, which is essentially the same as that studied in [7], [25], modulo the slight differences between the IC Pebble Game and the pebble game of those sources. When a node v of a computation-dag $\mathcal{G}$ is executed in a play of the IC Game, the results from v's task must be kept available in order to execute each of v's children (since we do not allow recomputation to regenerate the results). This observation leads us to posit the existence, at each step t of a play of the IC Pebble Game on $\mathcal{G}$, of a "register" (i.e., a result-retaining chunk of memory) that is associated with each EXECUTED node of $\mathcal{G}$ that has a non-EXECUTED child. The number of such *open* EXECUTED nodes at step t is our measure of the memory used at that step.

The **memory cost** *of a play of the IC Pebble Game on a computation-dag $\mathcal{G}$ is the maximum, over all steps t of the play, of the number of open EXECUTED nodes of $\mathcal{G}$.*

2.3 An Intuitive Strategy for Achieving IC Optimality

Intuition suggests that any IC-optimal schedule for a computation-dag $\mathcal{G}$ will be *parent-oriented*, i.e., will execute all parents of each node u in consecutive steps. This "greedy" strategy seems to produce ELIGIBLE nodes as quickly as possible; deviating from the strategy seems to delay such production. It is easy to craft examples that show that parent orientation is *not sufficient* to ensure IC optimality in arbitrary dags. However, we show that parent orientation is both necessary and sufficient for IC optimality in the families of dags studied here.

The IC scheduling problem becomes much more complicated when one seeks schedules that simultaneously minimize memory requirements and maximize IC quality. This complication is highlighted by our exhibiting IC-optimal schedules that are memory-pessimal, and memory-optimal schedules whose IC quality deviates from optimality by an amount that increases with the target dag's size. Indeed, for complete reduction-trees, no schedule optimizes both IC quality and memory requirements. These facts illustrate that intuition alone will not suffice to develop efficient schedules for IC platforms.

3 Optimal Schedules for Reduction-Meshes

This section is devoted to characterizing IC-optimal schedules for reduction-meshes. We exhibit an IC-optimal schedule that also minimizes memory cost.

3.1 The Source of IC Quality

The following analysis tells us how to schedule reduction-meshes IC optimally. Focus on a play of the IC Pebble Game on the ℓ-level reduction-mesh $\mathcal{M}_\ell$. Say that, at step t of the play, each level $l \in \{0, 1, \ldots, \ell - 1\}$ of $\mathcal{M}_\ell$ has $E_l^{(t)}$ ELIGIBLE nodes and $X_l^{(t)}$ EXECUTED nodes. Let c be the smallest level-number for which $E_c^{(t)} + X_c^{(t)} > 0$.

Lemma 3.1. *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 $\mathcal{M}_\ell$ are consecutive.*[3]
2. *Once $E^{(t)}$ is so maximized, we have $c \le E^{(t)} \le c + 1$.*

Proof.

1. Each nonsource ELIGIBLE node of $\mathcal{M}_\ell$ has two EXECUTED parents. Moreover, if the nonsource ELIGIBLE nodes u and v are consecutive on some level l of $\mathcal{M}_\ell$, then they share an EXECUTED parent. This verifies the following system of inequalities.

$$\begin{aligned} &E_l^{(t)} \le X_{l+1}^{(t)} - X_l^{(t)} - 1 \text{ for } l \in \{c, c+1, \ldots, \ell - 2\}; \\ &E_{\ell-1}^{(t)} = \ell - X_{\ell-1}^{(t)}. \end{aligned} \tag{3.1}$$

   If all EXECUTED nodes occur consecutively along a level $l + 1$ of $\mathcal{M}_\ell$, then the inequality involving $E_l^{(t)}$ in (3.1) is an equality. It follows that all inequalities in (3.1) are equalities when the EXECUTED nodes at every level of $\mathcal{M}_\ell$ 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 $E^{(t)}$.

3. 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. We sum the (now) equalities in system (3.1) to obtain an explicit expression for the maximum value of $E^{(t)}$ in terms of $\sum_{i=0}^{\ell-1} X_i^{(t)} = t$, given the current profile of EXECUTED nodes: $E^{(t)} = \sum_{i=c}^{\ell-1} E_i^{(t)} = c + 1 - X_c^{(t)}$. Part (2) now follows because, when the EXECUTED nodes at each level of $\mathcal{M}_\ell$ occur consecutively, we must have $X_c^{(t)} \leq 1$: A larger value would imply that $X_{c-1}^{(t)} + E_{c-1}^{(t)} > 0$. □

3.2 Characterizing IC Optimal Schedules

Theorem 3.1. *A schedule for reduction-meshes is IC optimal if, and only if, it is parent-oriented, i.e., it executes $\mathcal{M}_\ell$'s nodes level-by-level, starting with level $\ell - 1$, and always keeping the* EXECUTED *nodes at each level consecutive.*

Proof. Lemma 3.1.2 indicates that an optimal schedule for executing $\mathcal{M}_\ell$ always keeps c (the level-number of the lowest-numbered level that contains an ELIGIBLE or EXECUTED node) as large as possible for as long as possible. This means that an optimal schedule executes all nodes at level l of $\mathcal{M}_\ell$ before it executes any node at level $l - 1$.

Next, we maximize the number of ELIGIBLE nodes level-by-level during the computation by ensuring that each inequality in (3.1) is an equality, i.e., by executing the nodes on each level of $\mathcal{M}_\ell$ as a consecutive block. By the analysis of Section 3.1, this strategy maximizes $E^{(t)}$ at every step t of the computation; hence it is IC optimal. □

3.3 IC Optimal Schedules Optimize Memory Cost

Lemma 3.2 [7]. *Every schedule for $\mathcal{M}_\ell$ has memory cost $\geq \ell$.*

Lemma 3.2 is proven by counting the *open* EXECUTED nodes at the first step of the IC Pebble Game when every source-to-sink path contains at least one EXECUTED node.

Theorem 3.2. *The IC-optimal schedule that proceeds across each level of $\mathcal{M}_\ell$ is also memory-optimal.*

Proof. After step ℓ, when there are ℓ open EXECUTED nodes on $\mathcal{M}_\ell$, at least one such node is "closed" at every step. By Theorem 3.1, this schedule is IC optimal; by Lemma 3.2, it is also memory-optimal. □

3.4 How Important Is IC Optimality?

We now indicate the importance of Theorem 3.1 and its supporting analysis by exhibiting a "natural" strategy for executing $\mathcal{M}_\ell$ whose IC quality deviates from optimality by a multiplicative amount that increases with the number of EXECUTED nodes. Fig. 4 depicts step t of two schedules for $\mathcal{M}_\ell$, both of which optimize memory cost. The schedule of Fig. 4a proceeds *eagerly*, always executing the newest ELIGIBLE node; the schedule of Fig. 4b proceeds *lazily*, hence IC optimally. Let $E_{(\mathrm{a})}^{(t)}$ (respectively, $E_{(\mathrm{b})}^{(t)}$) denote the number of ELIGIBLE nodes in Fig. 4a (respectively, Fig. 4b); the number of EXECUTED nodes is, of course, t for both schedules. In Fig. 4a, the following nodes have been executed:

- for $1 \leq j < k$, the j nodes of "diagonal" $\ell - j$: $\{\langle \ell - j,\ i\rangle \mid i = 0, 1, \ldots, j - 1\}$;

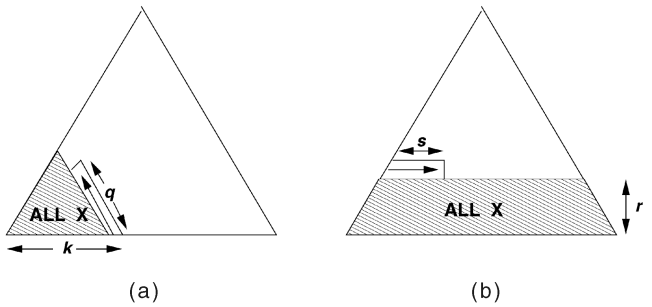

Fig. 4. $\mathcal{M}_\ell$ being executed: (a) eagerly, (b) IC optimally. Nodes in the shaded, "All X" areas are all EXECUTED.

- the "first" q nodes of "diagonal" $(\ell - k)$: $\{\langle \ell - k,\ i\rangle \mid i = k - q, \ldots, k - 1\}$.

Easily then, $t = \binom{k}{2} + q$ and $E_{(\mathrm{a})}^{(t)} \leq \ell - k + 1$ (with equality when $q < k$). In Fig. 4b, the following nodes of $\mathcal{M}_\ell$ have been executed:

- all nodes from levels $\ell - 1,\ \ell - 2, \ldots,\ \ell - r$; s nodes from level $\ell - r - 1$.

To compare $E_{(\mathrm{a})}^{(t)}$ and $E_{(\mathrm{b})}^{(t)}$ perspicuously, we note that $U^{(t)}$, the number of un-EXECUTED nodes at step t, satisfies

$$U^{(t)} \geq \binom{\ell+1}{2} - \binom{k+1}{2}.$$

This means that

$$E_{(\mathrm{b})}^{(t)} \geq \sqrt{2U^{(t)}} \geq \sqrt{2} \cdot \sqrt{\binom{\ell+1}{2} - \binom{k+1}{2}} + O(1)$$

so that

$$\frac{E_{(\mathrm{b})}^{(t)}}{E_{(\mathrm{a})}^{(t)}} \geq \sqrt{\frac{\ell + k - 1}{\ell - k}} + O(1) = \sqrt{1 + \frac{2k-1}{\ell - k}} + O(1). \quad (3.2)$$

The ratio (3.2) grows with k, becoming proportional to $\sqrt{\ell}$ as k approaches ℓ. Indeed, while the eager schedule is executing the last m diagonals of $\mathcal{M}_\ell$ (the nodes of the form $\langle i, k\rangle$ for $i \leq m - 1$), it has access to $\leq m$ ELIGIBLE nodes at each step, whereas the IC-optimal schedule has access to roughly $\sqrt{\ell}$ such nodes.

4 Optimal Schedules for Reduction-Trees

This section is devoted to characterizing IC-optimal schedules for reduction-trees. We exhibit an IC-optimal schedule for *complete* reduction-trees whose memory cost is within a factor of 2 of optimal and show that no IC-optimal schedule has better memory cost.

4.1 The Source of IC Quality

The following analysis tells us how to schedule reduction-trees IC optimally. Focus on a play of the IC Pebble Game on an ℓ-level reduction-tree $\mathcal{T}$ that has S source nodes (i.e., leaves), with S_l source nodes at each level l, for $l \in \{0, 1, \ldots, \ell - 1\}$.[4] Say that at step t of the play, each level $l \in \{0, 1, \ldots, \ell - 1\}$ of $\mathcal{T}$ has $E_l^{(t)}$ ELIGIBLE nodes and

4. Of course, $S_0 = 0$ if $\mathcal{T}$ has more than one level.

$X_l^{(t)}$ EXECUTED nodes. Let c be the smallest level-number such that $E_c^{(t)} + X_c^{(t)} > 0$.

We say that the EXECUTED nodes at level l of $\mathcal{T}$ are *sibling-paired* if there is at most one EXECUTED node $x\beta$ at level l whose sibling node $x\overline{\beta}$ is not EXECUTED.

Lemma 4.1. *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 the* EXECUTED *nodes along each level of $\mathcal{T}$ are sibling-paired.*
2. *Once $E^{(t)}$ is so maximized, we have $E^{(t)} = S - \frac{1}{2}t - \frac{1}{2}\sum_{i=0}^{\ell-1}(X_i^{(t)} \bmod 2)$.*

Proof.

1. Each ELIGIBLE nonsource node of $\mathcal{T}$ has two EXECUTED parents. It follows that, at time t:

$$E_l^{(t)} \le S_l + \left\lfloor \frac{1}{2} X_{l+1}^{(t)} \right\rfloor - X_l^{(t)} \text{ for } l \in \{0, 1, \ldots, \ell - 2\};$$
$$E_{\ell-1}^{(t)} = S_{\ell-1} - X_{\ell-1}^{(t)}. \tag{4.1}$$

   If all EXECUTED nodes at level $l+1$ of $\mathcal{T}$ are sibling-paired, then the inequality involving $E_l^{(t)}$ in (4.1) is an equality. It follows that all inequalities in (4.1) are equalities if all EXECUTED nodes along each level of $\mathcal{T}$ are sibling-paired. Further, forcing all EXECUTED nodes along each level of $\mathcal{T}$ to be sibling-paired may decrease the value of c by rendering new nodes ELIGIBLE at lower-numbered levels. Consequently, this arrangement of EXECUTED nodes at each level maximizes the value of $E^{(t)}$.
2. We sum the (now) equalities in system (4.1) to obtain an explicit expression for the maximum value of $E^{(t)}$, given the current profile of EXECUTED nodes in $\mathcal{T}$. Exploiting the fact that when the EXECUTED nodes at each level of $\mathcal{T}$ are sibling-paired, we must have $X_c^{(t)} \le 1$ (a larger value would imply that $X_{c-1}^{(t)} + E_{c-1}^{(t)} > 0$), we have

$$E^{(t)} = \sum_{i=0}^{\ell-1} E_i^{(t)} = \sum_{i=0}^{\ell-1} S_i - X_0^{(t)} - \sum_{i=1}^{\ell-1} \left\lceil \frac{1}{2} X_i^{(t)} \right\rceil$$
$$= S - \frac{1}{2}\sum_{i=0}^{\ell-1} X_i^{(t)} - \frac{1}{2}\Big(\text{the number of odd } X_i^{(t)}\Big).$$

   Part (2) now follows since $\sum_{i=0}^{\ell-1} X_i^{(t)} = X^{(t)} = t$. □

4.2 Characterizing IC Optimal Schedules

Theorem 4.1. *A schedule for reduction-trees is IC optimal if, and only if, it is parent-oriented, i.e., it always executes a node of a reduction-tree and its sibling in consecutive steps.*

Proof. By Lemma 4.1.1, an allocation of EXECUTED nodes to the levels of $\mathcal{T}$ maximizes the number of ELIGIBLE nodes (for that allocation) if each level of $\mathcal{T}$ has at most one EXECUTED node that is not sibling-paired. Lemma 4.1.2 then implies that one maximizes $E^{(t)}$ by minimizing the number of levels of $\mathcal{T}$ that contain odd numbers of EXECUTED nodes. If we execute sibling-pairs in consecutive steps, then, after every:

- even-numbered step, no level of $\mathcal{T}$ contains an odd number of EXECUTED nodes;
- odd-numbered step, precisely one level of $\mathcal{T}$ contains an odd number of EXECUTED nodes.

Since the IC Pebble Game allows us to execute only one node at a time, no schedule can improve on this behavior of alternating steps in which there is no "odd level" with steps in which there is one "odd level." □

Corollary 4.1. *The strategy of executing the nodes of a reduction-tree level-by-level, proceeding sequentially across each level, is IC optimal.*

Of course, one need not execute nodes level-by-level in order to achieve IC optimality. We next show how to craft alternative IC-optimal schedules to significant benefit.

4.3 Doubly Efficient Schedules for *Complete* Reduction-Trees

We now develop a strategy for scheduling *complete* reduction-trees that is IC optimal and:

- optimal in memory cost *given its IC optimality:* it uses $2h$ registers while executing $\mathcal{T}_h$; no IC-optimal schedule uses fewer registers;
- within a factor of 2 of optimal in memory cost: memory-optimal schedules use $h+1$ registers while executing $\mathcal{T}_h$.

For perspective, the level-by-level schedules of Corollary 4.1 are *pessimal* in memory cost, requiring 2^h registers at the (temporal) midpoint of the execution of $\mathcal{T}_h$.

The doubly efficient schedule. We conserve memory while remaining IC optimal by applying the parent-orientation mandated by Theorem 4.1 *lazily* while executing $\mathcal{T}_h$, in contrast to the *eager* application of the level-by-level strategy. Specifically, we execute the next sibling-pair of ELIGIBLE nodes at a given level of $\mathcal{T}_h$ only when not doing so would require us to leave a sibling-unpaired EXECUTED node at a lower-numbered level. We now present Algorithm Postorder-Execute, which embodies this strategy. The algorithm executes sibling-pairs of $\mathcal{T}_h$ in a modified *postorder* fashion. Fig. 5 illustrates the order in which Algorithm Postorder-Execute executes the sibling-pairs of $\mathcal{T}_3$.

Algorithm Postorder-Execute($\mathcal{T}_h$)
Repeat until **Halt**
  if the root of $\mathcal{T}_h$ is ELIGIBLE for execution
  then EXECUTE the root; **Halt**
  else Determine the lowest level-number l of $\mathcal{T}_h$ that contains an ELIGIBLE sibling-pair;
  EXECUTE the leftmost sibling-pair at level l

Theorem 4.2 *Algorithm Postorder-Execute schedules complete reduction-trees IC optimally. Moreover, it uses $2h$ memory locations while executing $\mathcal{T}_h$, which is optimal among IC-optimal schedules and less than double the memory cost of memory-optimal schedules.*

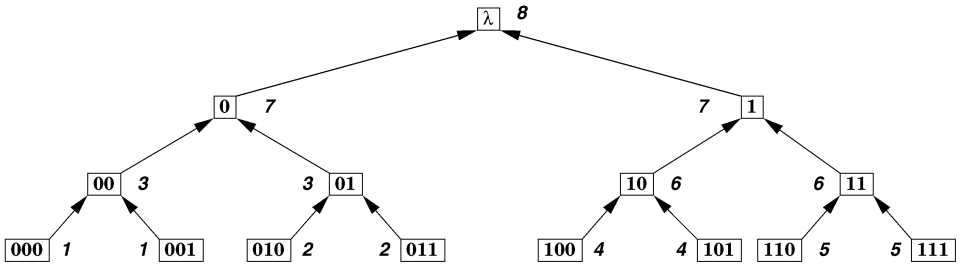

Fig. 5. $\mathcal{T}_3$ being executed by Algorithm Postorder-Execute; italic numbers indicate the execution-order of sibling-pairs.

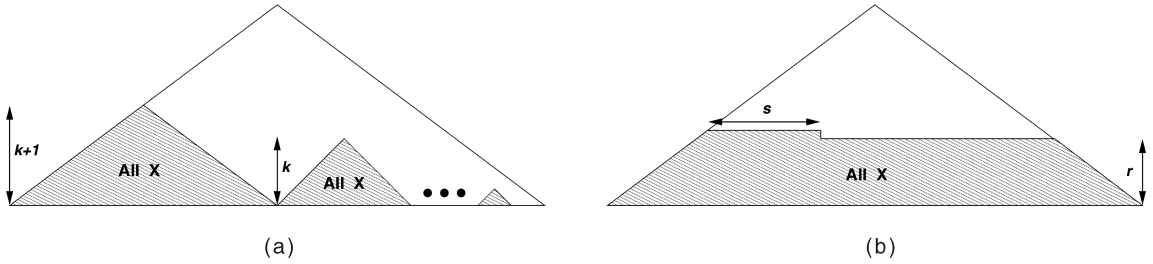

Fig. 6. $\mathcal{T}_h$ being executed: (a) eagerly, (b) IC optimally. Nodes in the shaded "All X" areas are all EXECUTED.

Proof. The algorithm executes ELIGIBLE sibling-pairs in consecutive steps, hence it is IC optimal (Theorem 4.1). We therefore focus on its memory cost. Under its modified postorder execution of $\mathcal{T}_h$, for each level $l = h, h-1, \ldots, 0$, in turn, the algorithm:

1. executes one copy of $\mathcal{T}_{h-l}$, ending with the root of that copy as the only ELIGIBLE node;
2. executes the sibling copy of $\mathcal{T}_{h-l}$, ending with the roots of the two copies of $\mathcal{T}_{h-l}$ as a sibling-pair;
3. executes the sibling-pair, at which point it has executed one copy of $\mathcal{T}_{h-l+1}$.

This reckoning means that the number of registers the algorithm uses while executing $\mathcal{T}_h$ is two more than it uses while executing $\mathcal{T}_{h-1}$. Since this inductive tally begins with the use of two registers while executing $\mathcal{T}_1$, we conclude that the algorithm employs a total of $2h$ registers when executing $\mathcal{T}_h$.

The optimality of the algorithm's memory cost follows from the next two lemmas. First, we compare the algorithm against absolute memory optimality.

Lemma 4.2 [25]. *Every schedule for $\mathcal{T}_h$ has memory cost $\geq h+1$. The pure postorder schedule achieves this cost.*

Next, we show that *no schedule optimizes both IC quality and memory requirements*.

Lemma 4.3. *Every IC-optimal schedule for $\mathcal{T}_h$ has memory cost $\geq 2h$.*

Proof. As with Lemmas 3.2 and 4.2, we consider the number of open EXECUTED nodes on $\mathcal{T}_h$ at the first step when every source-to-sink path contains at least one EXECUTED node. Say that this is step t.

Say, solely for notational convenience, that source node 1^h (a string of h 1s) is the last source node to be executed, i.e., it is the node executed at step t. It follows that, at step $t-1$, node 1^h's sibling, $1^{h-1}0$, is EXECUTED but sibling-unpaired. Since the schedule is IC optimal, node $1^{h-1}0$ must be the *only* sibling-unpaired EXECUTED node of $\mathcal{T}_h$ at step $t-1$. Since all $h-1$ nodes of the form 1^k, for $1 \leq k < h$, are not EXECUTED at step $t-1$, it follows that all of their siblings, the nodes $1^{k-1}0$, must also not be EXECUTED; otherwise, $\mathcal{T}_h$ would have more than one sibling-unpaired EXECUTED node. Since all of $\mathcal{T}_h$'s leaves, save node 1^h, are EXECUTED at time $t-1$, while all nodes of the form $1^{k-1}0$, for $1 \leq k < h$, are not EXECUTED, at least $2h-1$ registers must be in use at step $t-1$. Hence, at least $2h$ registers must be in use at step t. □

4.4 How Important Is IC Optimality?

We now indicate the importance of Theorem 4.1 and its supporting analysis, by showing that there are "natural" strategies for executing $\mathcal{T}_h$ whose IC quality deviates from optimality by an amount that oscillates over time but that periodically is a multiplicative factor that increases with the number of EXECUTED nodes. Fig. 6 depicts step t of two schedules for $\mathcal{T}_h$. The (memory-minimizing [25]) schedule of Fig. 6a executes $\mathcal{T}_h$ *eagerly*; i.e., the newest ELIGIBLE node is always chosen for execution. The schedule of Fig. 6b executes $\mathcal{T}_h$ level-by-level (i.e., *lazily*), hence (by Corollary 4.1) IC optimally. Let $E^{(t)}_{(\mathrm{a})}$ (respectively, $E^{(t)}_{(\mathrm{b})}$) denote the number of ELIGIBLE nodes in Fig. 6a (respectively, Fig. 6b); the number of EXECUTED nodes is, of course, t for both schedules. In Fig. 6a, the following nodes of $\mathcal{T}_h$ have been executed:

- For $0 \leq i \leq k$, one height-i complete subtree (which contains 2^i of $\mathcal{T}_h$'s leaves).

Easily then, $t = \sum_{i=0}^{k}(2^{i+1}-1) = 2^{k+2} - (k+3)$ and $E^{(t)}_{(\mathrm{a})} = 2^h - 2^{k+1} + 1$. In Fig. 6b, the following nodes of $\mathcal{T}_h$ have been executed:

- all nodes from levels $h,\ h-1, \ldots,\ h-r+1$; s nodes from level $h-r$,

where the integers r and $0 \le s < 2^{h-r}$ satisfy $s + \sum_{i=0}^{r-1} 2^{h-i} = t$. In order to compare $E^{(t)}_{(\mathrm{a})}$ and $E^{(t)}_{(\mathrm{b})}$ perspicuously, we note that $U^{(t)}$, the number of unEXECUTED nodes at step t, satisfies

$$U^{(t)} = (2^{h+1} - 1) - (2^{k+2} - (k+3)) > 2(2^h - 2^{k+1}) + k.$$

This means that $E^{(t)}_{(\mathrm{b})} > 2^h - 2^{k+1} + k/2$ so that

$$\frac{E^{(t)}_{(\mathrm{b})}}{E^{(t)}_{(\mathrm{a})}} \ge 1 + \frac{1}{2}\left(\frac{k}{2^h - 2^{k+1} + 1}\right) + O(1). \tag{4.2}$$

The quantity (4.2) grows *superlinearly* with k. Indeed, during the "end game," from step $2^{h+1} - h - 2$ on, the eager schedule has access to only one ELIGIBLE node per step, whereas the IC-optimal schedule has access to at least h/a such nodes for $(1 - 2/a)h$ of these steps. Note that, when the EXECUTED nodes under the eager schedule comprise precisely m complete binary subtrees, then $E^{(t)}_{(\mathrm{b})} - E^{(t)}_{(\mathrm{a})} = m - 1$.

5 Optimal Schedules for FFT Dags

This section is devoted to characterizing IC-optimal schedules for FFT dags. We exhibit an IC-optimal schedule whose memory cost is within 1 of optimal.

5.1 The Source of IC Quality

The following analysis tells us how to schedule FFT dags IC optimally: Focus on a play of the IC Pebble Game on the d-dimensional FFT dag $\mathcal{F}_d$. Say that, at step t of the play, each level $l \in \{0, 1, \ldots, d\}$ of $\mathcal{F}_d$ has $E^{(t)}_l$ ELIGIBLE nodes and $X^{(t)}_l$ EXECUTED nodes. Let c be the smallest level-number such that $E^{(t)}_c + X^{(t)}_c > 0$.

Recalling—cf. Fig. 3 and (2.1)—that $\mathcal{F}_d$ is composed of many *butterflies*, we call pairs of level-ℓ nodes $\langle \ell, x\beta_{d-\ell}y\rangle$ and $\langle \ell, x\overline{\beta}_{d-\ell}y\rangle$ *butterfly partners*. We say that the EXECUTED nodes at level ℓ of $\mathcal{F}_d$ are *butterfly-paired* if there is at most one EXECUTED level-ℓ node whose butterfly partner is not EXECUTED.

Lemma 5.1. *Given the current profile $\langle X^{(t)}_l \mid 0 \le l \le d\rangle$ of* EXECUTED *nodes:*

1. *The aggregate number of* ELIGIBLE *nodes at time t, $E^{(t)} \stackrel{def}{=} \sum_{i=0}^{d} E^{(t)}_i$, is maximized if all* EXECUTED *nodes along each level of $\mathcal{F}_d$ are butterfly-paired.*
2. *Once $E^{(t)}$ is so maximized, we have $E^{(t)} = 2^d - X^{(t)}_0 - \sum_{i=1}^{d}(X^{(t)}_i \bmod 2)$.*

Proof.

1. As indicated in (2.1), each nonsource ELIGIBLE node of $\mathcal{F}_d$ has two EXECUTED parents. Moreover, each ELIGIBLE node shares these parents with its butterfly partner. This verifies the following system of inequalities:

$$\begin{aligned} &E^{(t)}_l \le 2\left\lfloor \frac{1}{2} X^{(t)}_{l+1} \right\rfloor - X^{(t)}_l \quad \text{for } l \in \{0, 1, \ldots, d-1\};\\ &E^{(t)}_d = 2^d - X^{(t)}_d. \end{aligned} \tag{5.1}$$

   If all EXECUTED nodes at level $l+1$ of $\mathcal{F}_d$ are butterfly-paired, then the inequality involving $E^{(t)}_l$ in (5.1) is an equality. It follows that all inequalities in (5.1) are equalities if all EXECUTED nodes along each level of $\mathcal{F}_d$ are butterfly-paired. Further, forcing the EXECUTED nodes at each level to occur as a block 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 $E^{(t)}$.
2. We now obtain part (2) by summing the (now) equalities in system (5.1) and noting that $X^{(t)}_c \le 1$ when all EXECUTED nodes along each level of $\mathcal{F}_d$ are butterfly-paired (since $X^{(t)}_{c-1} + E^{(t)}_{c-1} = 0$). □

5.2 Characterizing IC Optimal Schedules

Theorem 5.1. *A schedule for FFT dags is IC optimal if, and only if, it is parent-oriented, i.e., it always executes a node of $\mathcal{F}_d$ and its butterfly partner in consecutive steps.*

Proof. By Lemma 5.1.1, an allocation of EXECUTED nodes to the levels of $\mathcal{F}_d$ maximizes the number of ELIGIBLE nodes (for that allocation) if each level of $\mathcal{F}_d$ has at most one EXECUTED node that is not butterfly-paired. Lemma 5.1.2 then implies that one maximizes $E^{(t)}$ by minimizing the number of levels of $\mathcal{F}_d$ that contain odd numbers of EXECUTED nodes. If we always execute butterfly partners in consecutive steps, then:

- After every even-numbered node-execution, no level of $\mathcal{F}_d$ contains an odd number of EXECUTED nodes.
- After every odd-numbered node-execution, precisely one level of $\mathcal{F}_d$ contains an odd number of EXECUTED nodes.

Since the IC Pebble Game allows us to execute only one node at a time, no schedule can improve on this behavior of alternating steps in which there is no "odd level" with steps in which there is one "odd level." □

Corollary 5.1. *The strategy of executing the nodes of an FFT dag level-by-level, always executing a node and its butterfly partner consecutively, is optimal in IC quality.*

5.3 Memory-Efficient IC Optimal Schedules

The IC-optimal schedules of Corollary 5.1 have memory cost that is within 1 of optimal.

Lemma 5.2. *Every schedule for $\mathcal{F}_d$ has memory cost $\ge 2^d$.*

Proof. For an arbitrary schedule, consider a snapshot of a play of the IC Pebble Game at the moment when an ELIGIBLE pebble is first placed on level 0 of $\mathcal{F}_d$, say on node v. Note that the subdag of $\mathcal{F}_d$ comprised of all of v's ancestors is a copy of $\mathcal{T}_d$ whose 2^d sources are $\mathcal{F}_d$'s 2^d sources; see Fig. 7. By the rules of the IC Pebble Game, every ancestor of v must be EXECUTED. This means that every one of $\mathcal{F}_d$'s 2^d "columns" must contain at least one

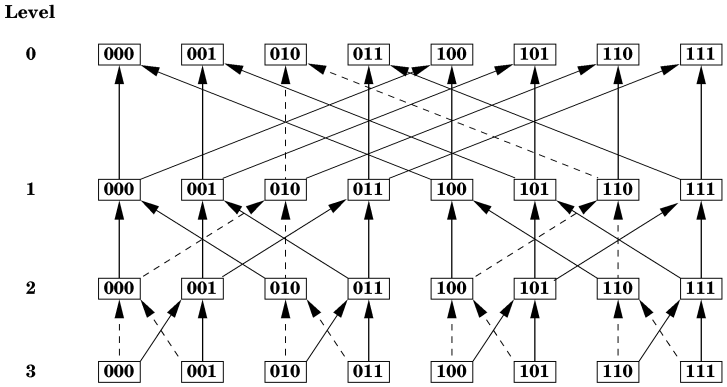

Fig. 7. $\mathcal{F}_3$, with the complete reduction-tree of ancestors of node $\langle 0, 010\rangle$ highlighted via dashed arcs.

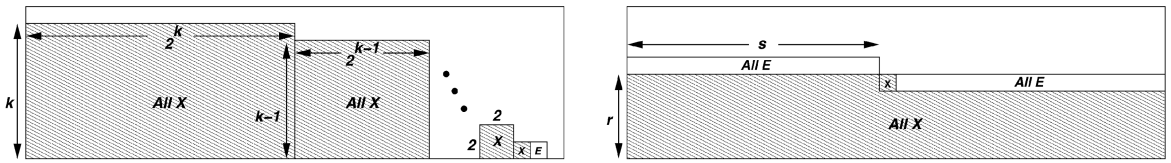

Fig. 8. $\mathcal{F}_d$ being executed: (a) eagerly, (b) IC optimally. Nodes in the shaded "All X" areas are all EXECUTED.

open EXECUTED node. Specifically, in every "column" save the one containing node v, there is an EXECUTED node at some level $i > 0$, while the node at level 0 is not EXECUTED; in the "column" containing v, v's parent's other child is not EXECUTED. □

Theorem 5.2. *Any schedule for $\mathcal{F}_d$ that executes nodes level-by-level, always executing a node and its butterfly partner consecutively, has memory cost $2^d + 1$, hence it is within 1 of optimal in memory cost.*

Proof. The theorem follows by recognizing that exactly $2^d + 1$ registers are employed when both of the following conditions hold:

- All nodes at some level $i \geq 0$ are EXECUTED.
- Some node at level $i + 1$ is EXECUTED while its butterfly partner is not.

For all other steps in the schedule, no more than 2^d registers are required. □

5.4 How Important Is IC Optimality?

We now indicate the importance of Theorem 5.1 and its supporting analysis by showing that there are "natural" strategies for executing $\mathcal{F}_d$ whose IC quality deviates from optimality by an amount that oscillates over time but that periodically is a multiplicative factor that increases with the number of EXECUTED nodes. Fig. 8 depicts step t of two plays for $\mathcal{F}_d$, which have identical memory requirements. The schedule of Fig. 8a executes $\mathcal{F}_d$ *eagerly*, i.e., the newest ELIGIBLE node is always chosen for execution. The schedule of Fig. 8b is the IC-optimal schedule described in Corollary 5.1. Let $E^{(t)}_{(\mathrm{a})}$ (respectively, $E^{(t)}_{(\mathrm{b})}$) denote the number of ELIGIBLE nodes in Fig. 8a (respectively, Fig. 8b); the number of EXECUTED nodes is, of course, t for both schedules. In Fig. 8a, the following nodes of $\mathcal{F}_d$ have been executed:

- For $1 \leq i \leq k$, one i-level sub-FFT dag of $\mathcal{F}_d$ (which contains 2^{i-1} of $\mathcal{F}_d$'s source nodes).

Easily then, $t = \sum_{i=1}^{k} i2^{i-1} = (k-1)2^k + 1$ and $E^{(t)}_{(\mathrm{a})} = 2^d - 2^{k+1} + 1$. In Fig. 8b, the following nodes of $\mathcal{F}_d$ have been executed:

- all nodes from levels $d, d-1, \ldots, d-r+1$; s nodes from level $d-r$,

where the integers r and $0 \leq s < 2^{d-r}$ satisfy $s + r2^d = t$. We see easily that $E^{(t)}_{(\mathrm{b})} = 2^d - (s \bmod 2) \geq 2^d - 1$. We thus have

$$\frac{E^{(t)}_{(\mathrm{b})}}{E^{(t)}_{(\mathrm{a})}} \geq 1 + \frac{1}{2}\left(\frac{2^k - 1}{2^d - 2^{k+1} + 1}\right) + O(1),$$

which grows faster than 2^{k-1}, where the number of EXECUTED nodes is roughly $k2^k$. Notably, from step $(k-1)2^k + 1$ on, the eager schedule must execute a complete binary tree "out-dag," whose root node 1^d is its sole source, and whose leaves are its 2^d sinks.

6 Conclusion and Projections

Using the IC Pebble Game of [28], we have continued that source's study of the problem of scheduling computations with intertask dependencies on the Internet. The goal of the study is to devise schedules that produce execution-eligible tasks as fast as possible in order to minimize the likelihood of the gridlock that can occur when computational progress cannot be made pending the return of tasks from remote Clients. We have focused on three important families of computation-dags: reduction-meshes (Section 3), reduction-trees (Sections 4.2 and 4.3), and FFT dags (Section 5). For each family, we have devised schedules that provably maximize the production rate of execution-eligible tasks,

even while minimizing the amount of memory needed to achieve such a production rate.

In order to simplify our development while still conveying underlying principles, we have restricted attention to two-dimensional reduction-meshes and to binary reduction-trees. The reader should be able to extend our results to reduction-meshes of arbitrary finite dimensionalities and to reduction-trees of arbitrary finite arities.

The novel goal of our schedules has demanded the development of new techniques for analyzing the performance of schedules on computation-dags. The techniques we use here exploit the uniform structures of the dag families we have studied. It is an inviting challenge to extend our techniques to structurally less uniform families of dags and to dags that evolve in unpredictable (perhaps stochastic) manners.

ACKNOWLEDGMENTS

This research was supported in part by US National Science Foundation Grants CCR-00-73401 and CCF-0342417. The authors have benefited from the comments of anonymous referees. A portion of this work was presented at the 17th International Conference on Parallel and Distributed Computing Systems 2004 (PDCS).

REFERENCES

[1] K. Aberer and Z. Despotovic, "Managing Trust in a Peer-2-Peer Information System," *Proc. 10th Intl Conf. Information and Knowledge Management*, 2001.

[2] V.E. Beneš, "Optimal Rearrangeable Multistage Connecting Networks," *Bell System Technical J.*, vol. 43, pp. 1641-1656, 1964.

[3] S.N. Bhatt, F.R.K. Chung, F.T. Leighton, and A.L. Rosenberg, "Scheduling Tree-Dags Using FIFO Queues: A Control-Memory Tradeoff," *J. Parallel and Distributed Computing*, vol. 33, pp. 55-68, 1996.

[4] S.N. Bhatt, F.R.K. Chung, J.-W. Hong, F.T. Leighton, B. Obrenić, A.L. Rosenberg, and E.J. Schwabe, "Optimal Emulations by Butterfly-Like Networks," *J. ACM*, vol. 43, pp. 293-330, 1996.

[5] R. Buyya, D. Abramson, and J. Giddy, "A Case for Economy Grid Architecture for Service Oriented Grid Computing," *Proc. 10th Heterogeneous Computing Workshop*, 2001.

[6] W. Cirne and K. Marzullo, "The Computational Co-Op: Gathering Clusters into a Metacomputer," *Proc. 13th Int'l Parallel Processing Symp.*, pp. 160-166, 1999.

[7] S.A. Cook, "An Observation on Time-Storage Tradeoff," *J. Computer Systems and Sciences*, vol. 9, pp. 308-316, 1974.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, second ed. Cambridge, Mass.: MIT Press, 1999.

[9] *The Grid: Blueprint for a New Computing Infrastructure*, I. Foster and C. Kesselman, eds. San Francisco: Morgan-Kaufmann, 1999.

[10] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Int'l J. Supercomputer Applications*, 2001.

[11] L.-X. Gao, A.L. Rosenberg, and R.K. Sitaraman, "Optimal Clustering of Tree-Sweep Computations for High-Latency Parallel Environments," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, pp. 813-824, 1999.

[12] M.R. Garey, R.L. Graham, D.S. Johnson, and D.E. Knuth, "Complexity Results for Bandwidth Minimization," *SIAM J. Applied Math.*, vol. 34, pp. 477-495, 1978.

[13] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling Dags on Multiprocessors," *J. Parallel and Distributed Computing*, vol. 16, pp. 276-291, 1992.

[14] P. Golle and I. Mironov, "Uncheatable Distributed Computations," *Proc. RSA Conf. 2001 (Cryptographers' Track)*, 2001.

[15] L. He, Z. Han, H. Jin, L. Pan, and S. Li, "DAG-Based Parallel Real Time Task Scheduling Algorithm on a Cluster," *Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, pp. 437-443, 2000.

[16] J.-W. Hong and H.T. Kung, "I/O Complexity: The Red-Blue Pebble Game," *Proc. 13th ACM Symp. Theory of Computing*, pp. 326-333, 1981.

[17] *The Intel Philanthropic Peer-to-Peer program.* www.intel.com/cure, 2001.

[18] A. Jakoby and R. Reischuk, "The Complexity of Scheduling Problems with Communication Delays for Trees," *Proc. Scandinavian Workshop Algorithmic Theory*, pp. 163-177, 1992.

[19] R.M. Karp, A. Sahay, E. Santos, and K.E. Schauser, "Optimal Broadcast and Summation in the logP Model," *Proc. Fifth ACM Symp. Parallel Algorithms and Architectures*, pp. 142-153, 1993.

[20] D. Kondo, H. Casanova, E. Wing, and F. Berman, "Models and Scheduling Guidelines for Global Computing Applications," *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS '02)*, 2002.

[21] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky, "SETI@home: Massively Distributed Computing for SETI," *Computing in Science and Eng.*, P.F. Dubois, ed., Los Alamitos, Calif.: IEEE CS Press, 2000.

[22] J.K. Lenstra, M. Veldhorst, and B. Veltman, "The Complexity of Scheduling Trees with Communication Delays," *J. Algorithms*, vol. 20, pp. 157-173, 1996.

[23] B.M. Maggs and R.K. Sitaraman, "Simple Algorithms for Routing on Butterfly Networks with Bounded Queues," *SIAM J. Computing*, vol. 28, pp. 984-1004, 1999.

[24] *The Olson Laboratory Fight AIDS@Home Project*, www.fightaidsathome.org, 2001.

[25] M.S. Paterson and C.E. Hewitt, "Comparative Schematology," *Proc. Project MAC Conf. Concurrent Systems and Parallel Computation*, pp. 119-127, 1970.

[26] A.G. Ranade, "Optimal Speedup for Backtrack Search on a Butterfly Network," *Math. Systems Theory*, vol. 27, pp. 85-101, 1994.

[27] A.L. Rosenberg, "Accountable Web-Computing," *IEEE Trans. Parallel and Distributed Systems*, vol. 14, pp. 97-106, 2003.

[28] A.L. Rosenberg, "On Scheduling Mesh-Structured Computations for Internet-Based Computing," *IEEE Trans. Computers*, vol. 53, pp. 1176-1186, 2004.

[29] A.L. Rosenberg and I.H. Sudborough, "Bandwidth and Pebbling," *Computing*, vol. 31, pp. 115-139, 1983.

[30] X.-H. Sun and M. Wu, "GHS: A Performance Prediction and Task Scheduling System for Grid Computing," *Proc. IEEE Int'l Parallel and Distributed Processing Symp.*, 2003.

Arnold L. Rosenberg is a Distinguished University Professor of Computer Science at the University of Massachusetts Amherst, where he codirects the Theoretical Aspects of Parallel and Distributed Systems (TAPADS) Research Laboratory. Prior to joining UMass, he was a professor of computer science at Duke University from 1981 to 1986 and a research staff member at the IBM T.J. Watson Research Center from 1965 to 1981. He has held visiting positions at Yale University and the University of Toronto; he was a Lady Davis Visiting Professor at the Technion (Israel Institute of Technology) in 1994, and a Fulbright Research Scholar at the University of Paris-South in 2000. His research focuses on developing algorithmic models and techniques to deal with the new modalities of "collaborative computing" (the endeavor of having several computers cooperate in the solution of a single computational problem) that result from emerging technologies. He is the author or coauthor of more than 150 technical papers on these and other topics in theoretical computer science and discrete mathematics and is the coauthor of the book *Graph Separators, with Applications*. Dr. Rosenberg is a fellow of the ACM, a fellow of the IEEE, and a Golden Core member of the IEEE Computer Society.

Matthew Yurkewych received the BS degree from the Massachusetts Institute of Technology in 1998. He is a PhD student in computer science at the University of Massachusetts Amherst. Prior to entering graduate school, he worked at Akamai Technologies and CNet Networks as a software engineer.