Scheduling Tree-Dags Using FIFO Queues: A Control-Memory Tradeoff

S.N. Bhatt, F.R.K. Chung
F.T. Leighton & A.L. Rosenberg

Computer Science Technical Report 93-48
September, 1993

Scheduling Tree-Dags Using FIFO Queues: A Control-Memory Tradeoff

Sandeep N. Bhatt
Bell Communications Research
Morristown, N.J.

Fan R. K. Chung
Bell Communications Research
Morristown, N.J.

F. Thomson Leighton
MIT
Cambridge, Mass.

Arnold L. Rosenberg
University of Massachusetts
Amherst, Mass.

Abstract. We study a combinatorial problem that is motivated by "client-server" schedulers for parallel computations. Such schedulers are often used, for instance, when computations are being done by a cooperating network of workstations. Our results expose and quantify a control-memory tradeoff for such schedulers, when the computation being scheduled has the structure of a binary tree. (Similar tradeoffs exist for trees of any fixed branching factor.) The combinatorial problem takes the following form. Consider, for integers $k, N > 0$, an algorithm that employs k FIFO queues in order to schedule an N-leaf binary tree in such a way that each nonleaf node of the tree is executed before its children. We establish a tradeoff between the number of queues used by the algorithm — which we view as measuring the *control complexity* of the algorithm — and the *memory requirements* of the algorithm, as embodied in the required capacity of the largest-capacity queue. Specifically, for each integer $k \in \{1, 2, \ldots, \log_2 N\}$, let $\mathcal{Q}_k(N)$ denote the minimax per-queue capacity for a k-queue algorithm that schedules N-leaf binary trees; let $\mathcal{Q}_k^\star(N)$ denote the analogous quantity for *complete* binary trees. We establish the following bounds: For general N-leaf binary trees, for all k,

$$\frac{1}{k}\frac{(2N-1)^{1/k}}{\log N + 1} \le \mathcal{Q}_k(N) \le 2N^{1/k} + 1.$$

For complete binary trees, we derive tighter bounds. We prove that for all constant k,

$$\mathcal{Q}_k^\star(N) = \Theta\left(\frac{N^{1/k}}{\log^{1-1/k} N}\right).$$

For general k, we obtain the following bounds:

$$\frac{1}{k}\frac{N^{1/k}}{(\log N + 1)^{1-1/k}} \le \mathcal{Q}_k^\star(N) \le (4k)^{1-1/k}\frac{N^{1/k}}{\log^{1-1/k} N}.$$

1 Introduction

1.1 Overview

We study the resource requirements of a class of algorithms for scheduling parallel computations. Our main results expose and quantify a tradeoff between two major resources of the algorithms, the complexity of their control mechanisms and their memory requirements.

The Computing Environment. We are interested in schedulers that operate in a *client-server mode*, where the processors are the clients, and the scheduler is the server. One encounters such schedulers, for example, in systems that use networks of workstations for parallel computation; cf. [9], [10], [15]. We restrict attention to algorithms that schedule static *dags* (*directed acyclic graphs* — which model the data dependencies in a computation) in an architecture-independent fashion; cf. [1] - [3], [8], [11], [16] - [18]. One can view schedulers of this type as operating in the following way. (*a*) They determine when a task becomes eligible for execution (because all of its predecessors in the dag have been executed); (*b*) they queue up the eligible, unassigned tasks (in some way) in a FIFO *process queue* (PQ). When a processor becomes idle, it "grabs" the first task on the PQ.

Note that this scenario allows great latitude for processors: a processor can choose to participate or disappear from the computation independently at any time. But, once a processor accepts a task, it must complete the task within "unit time." In other words, the scheduler is clocked, and processors must commit to completing work they accept at a guaranteed rate.

The Computational Load. Our particular focus is on dags that are binary trees whose edges are oriented from the root toward the leaves. Such dags represent the data dependencies of certain types of branching computations. (We concentrate on *binary* tree-dags only for definiteness; our results extend readily to tree-dags of arbitrary fixed branching factor.)

Formally, a *binary tree dag* (*BT*, for short) is a directed acyclic graph whose node-set is a *prefix-closed* set of binary strings; i.e., for all binary strings x and all $\alpha \in \{0,1\}$, if $x\alpha$ is a node of the BT, then so also is x. The null string (which, by prefix-closure, belongs to every BT) is the *root* of the BT. Each node x of a BT has either two *children*, or one child, or no children; in the first case, the two children are nodes $x0$ and $x1$; in the second case, the one child is either node $x0$ or node $x1$; in the last case, node x is a *leaf* of the BT. The arcs of a BT lead from each nonleaf node to (each of) its child(ren). For each $\ell \in \{0, 1, \ldots, n\}$, the node-strings of length ℓ comprise *level* ℓ of the BT (so the

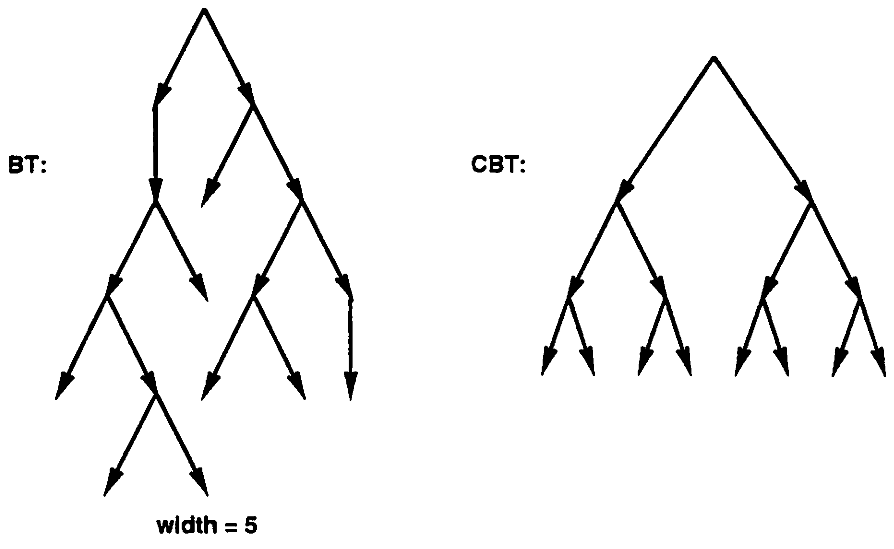

Figure 1: *A width-5, depth-6 BT and an 8-leaf depth-4 CBT.*

root is the unique node at level 0). The *width* of a BT is the maximum number of nodes at any level, and the *depth* of a BT is its number of levels.

The $(N = 2^n)$-*leaf complete binary tree* (*CBT*, for short) $\mathcal{T}_n$ is the BT whose nodes comprise the set of all $2^{n+1} - 1$ binary strings of length $\leq n$; hence the depth of $\mathcal{T}_n$ is $n + 1$. There are N nodes at level n of $\mathcal{T}_n$, namely, its leaves, so the width of $\mathcal{T}_n$ is N. See Fig. 1.

Scheduling Regimens and Scheduler Structure. In order to establish rigorously a tradeoff between the control complexity of our schedulers and their memory requirements, we must specify enough of the structure of a scheduler to quantify these resources. We view a scheduler as using some number of FIFO queues to prioritize tasks that have become eligible for execution: the specific number of queues is our measure of the control complexity of the scheduler; the minimax capacity of the queues is our measure of the scheduler's memory requirements.[1] Roughly speaking, a scheduler uses its queues as follows. As tasks get executed, they produce results that are necessary for the execution of their children. These results are loaded independently onto the FIFO queues of the scheduler; the task that is assigned to the next requesting processor is chosen from among those whose required input resides at the head of some queue.

Multi-queue schedulers are interesting for our computational load because increasing the number of queues within a scheduler allows one to proceed gradually from an *eager*

[1]We recognize that one might wish to use data structures other than FIFO queues (e.g., LIFO stacks) to manage tasks awaiting execution; such alternatives also merit study.

regimen, in which eligible tasks are delayed as little as possible before being assigned for execution, to a *lazy* regimen, in which eligible tasks are delayed as long as possible before being assigned for execution. The following facts about our framework will become clear as our study develops.

- Any scheduler that uses only one queue:
  - observes an eager regimen
  - executes tree-nodes level by level, i.e., essentially in a *breadth-first* manner.
- Lazy scheduling:
  - is an option when the number of queues is commensurate with the length of the longest root-to-leaf path of the tree-dag
  - is characterized by executing tree-nodes essentially in a *depth-first* manner.

(The two uses of the qualifier "essentially" here reflect the fact that inputs for sibling tasks in the tree can be interchanged in the enqueuing process without any concomitant change in the complexity of the scheduler.) While the *control complexity* of a scheduler increases as it incorporates successively more queues, we shall show that the *memory requirements* — as measured by the maximum number of eligible tasks that are awaiting execution — decrease concomitantly. We are able to establish and quantify this control-memory tradeoff rigorously.

The Tradeoff. For positive integers k and N, let $\mathcal{Q}_k(N)$ denote the minimax per-queue memory capacity for a k-queue algorithm that schedules N-leaf binary tree-dags; let $\mathcal{Q}_k^\star(N)$ denote the analogous quantity for *complete* binary trees. We establish the following bounds: For general BTs, for all k:[2]

$$\frac{1}{k}\frac{(2N-1)^{1/k}}{\log N + 1} \leq \mathcal{Q}_k(N) \leq 2N^{1/k} + 1.$$

Hence, the upper and lower bounds differ by roughly a logarithmic factor. For complete binary trees, we derive tighter bounds: for all constant k:

$$\mathcal{Q}_k^\star(N) = \Theta\left(\frac{N^{1/k}}{\log^{1-1/k} N}\right);$$

for general k:

$$\frac{1}{k}\frac{N^{1/k}}{(\log N + 1)^{1-1/k}} \leq \mathcal{Q}_k^\star(N) \leq (4k)^{1-1/k}\frac{N^{1/k}}{\log^{1-1/k} N}.$$

This verifies rigorously an instance of folklore in the scheduling community to the effect that lazy dag-schedulers need less memory than do eager ones.

[2]All logarithms are to the base 2.

1.2 The Formal Problem

BTs, being dags, represent the data-dependency graphs of computations, specifically a class of branching computations. In this scenario, the nodes of the BT represent the tasks, while its arcs represent computational dependencies among the tasks. These dependencies influence any algorithm that schedules the computation represented by the BT, in that a task-node cannot be executed until its parent task-node has been executed. This interpretation is consistent with the static (off-line) scheduling problems studied in [1] - [3], [8], [11], [16] - [18].

Scheduling BTs. The process of *scheduling* a BT $\mathcal{T}$ obeys the following rules. We are given an endless supply of *enabling tokens* and *execution tokens*. Placing an execution token on a node of $\mathcal{T}$ represents the process of executing the task represented by the node. As we execute a node v of $\mathcal{T}$, we place an enabling token on each arc that leaves v, to indicate that the results produced by v are now available; we label each enabling token with a *time stamp* indicating when the token was placed. A node of $\mathcal{T}$ cannot receive an execution token until its incoming arc contains an enabling token. Thus, the process of scheduling a BT proceeds as follows: At step 0 of the scheduling process, we place an execution token on the root of $\mathcal{T}$ (which, of course, has no incoming arc), and we place an enabling token *with time stamp* 0 on each arc leaving the root. At each subsequent step, say step $s > 0$, we perform two actions:

- We place an execution token on some (one) unexecuted node of $\mathcal{T}$ whose incoming arc contains an enabling token.
- We place enabling tokens, with time stamp s, on all arcs that leave the just-executed node.

This process continues until all nodes of $\mathcal{T}$ contain execution tokens. The reader should be able to extrapolate from this BT-specific description to a description of a scheduling process for an arbitrary dag: the major difference is that a node cannot be executed until all of its incoming arcs contain enabling tokens.

We call a scheduling algorithm *eager* if, at each step, it places an execution token on an unexecuted node whose incoming arc contains an enabling token having as *small* a time-stamp as possible; we call a scheduling algorithm *lazy* if, at each step, it places an execution token on an unexecuted node whose incoming arc contains an enabling token having as *large* a time-stamp as possible. One verifies easily that an eager scheduling algorithm executes the nodes of $\mathcal{T}$ level by level, i.e., essentially in a *breadth-first* manner, while a lazy scheduling algorithm executes the nodes of $\mathcal{T}$ essentially in a *depth-first* manner. Our interest here is in a family of scheduling algorithms that form a progression between eager scheduling at one extreme and lazy scheduling at the other. We need more tools to describe this progression formally.

Scheduling BTs using Queues: Control vs. Memory. The reader can verify easily that the process of scheduling a BT $\mathcal{T}$ is "isomorphic" to the process of topologically sorting $\mathcal{T}$, i.e., linearly ordering the nodes of $\mathcal{T}$ so that each nonleaf node precedes its children. Our study focuses on the structure of the algorithm that "manages" the process of scheduling/topologically sorting $\mathcal{T}$. The particular formal framework for our study specializes the framework studied in [4] - [7], [13].

A *k-queue scheduler for a BT* $\mathcal{T}$ proceeds as follows. Initially, at time 0, the scheduler executes the root of $\mathcal{T}$ and enqueues each arc that leaves the root, independently, in one of its k queues. Inductively, a node v of $\mathcal{T}$ is eligible to be executed — i.e., to receive an execution token — just when its entering arc is at the *head* (i.e., the exit port) of some queue. As the scheduler executes a node v, it dequeues the arc that enters v; simultaneously — *as part of the same atomic action* — the scheduler enqueues each arc that leaves v, independently, on one of the k queues. Henceforth, let us denote the k queues of the scheduler as queue #1, ..., queue #k.

A multi-queue BT-scheduler uses its queues to manage the eligible tasks that are awaiting execution. Specifically, enqueuing an arc is equivalent to endowing it with an enabling token; dequeuing an arc is equivalent to placing an execution token on the node the arc enters. The "management" function of the queues is manifest in the fact that only nodes whose incoming arcs reside at the heads of queues can be executed in the next step.

Easily, there exist k-queue BT-schedulers for every positive integer k. A straightforward induction verifies that there is a unique 1-queue BT-scheduler — up to the distinction between "left" and "right" children.

Fact 1.1 *The unique* 1*-queue BT-scheduler executes BT-nodes level by level, i.e., "essentially" in breadth-first order.*

A consequence of the rules for manipulating queues is that the arc that enters a BT-node v does not get enqueued until all of the ancestors of v have already been executed. This verifies the following simple observation, which is important later.

Fact 1.2 *All arcs that coexist in the queues of a multiqueue BT-scheduler at any instant enter nodes that are* independent *in the BT; i.e., none is an ancestor of another.*

We view the number of queues a BT-scheduler uses as its *control complexity*; we view the *worst-case individual capacity* of the queues as measuring the *memory requirements* of the scheduler. In this worldview, the *capacity of queue #q* is the maximum number of arcs of $\mathcal{T}$ that will ever reside in queue #q at the same instant. Clearly, the worst-case *cumulative* capacity of the queues — which some might view as a better measure of a scheduler's memory requirements — is at most k times the worst-case individual capacity. Obtaining bounds on the cumulative capacity that are tighter than the one obtained from this simple observation appears to be quite difficult.

We now turn to the topic of tradeoffs between the amount of control in a BT-scheduler and its memory requirements.

2 A Control-Memory Tradeoff

A Roadmap. Our goal is to verify the tradeoffs that are stated roughly at the end of Section 1.1, between the control complexity of a BT-scheduler — as measured by the quantity k — and the memory requirements of the scheduler — as measured by the quantities $\mathcal{Q}_k(N)$ and $\mathcal{Q}_k^\star(N)$. The possible existence of such a tradeoff is suggested by a family of simple CBT-schedulers (using successively more queues) that we present and analyze in Section 2.1. This family of schedulers, which derives from [7], illustrates that $\mathcal{Q}_k^\star(N) = O(N^{1/k})$, uniformly in N and k. In Section 2.2, we state formally the actual tradeoffs that are the main contribution of our study. Sections 3 and 4 are devoted to proving the upper bounds in the tradeoffs for general BTs and complete BTs, respectively; Section 5 is devoted to proving the lower bounds of the tradeoffs. We note in Section 6.1 that our results extend readily to other classes of tree-based dags. Although we cannot characterize what non-tree-based dags experience such tradeoffs, we close the paper with remarks in Section 6.2 about characteristics of classes of dags that preclude such tradeoffs.

Since our upper and lower bounds derive from recurrences on $\mathcal{Q}_k(N)$ and $\mathcal{Q}_k^\star(N)$ as functions of k and N, it is useful to note the following immediate consequence of Fact 1.1.

Lemma 2.1 *Consider a 1-queue scheduling algorithm for an N-leaf, width-W, depth-D BT.*
(a) The queue of the scheduler must have capacity at least $W \geq (2N-1)/D$.
(b) It suffices for the queue of the scheduler to have capacity $W \leq N$.
(c) In particular, for all $N = 2^n$, $\mathcal{Q}_1^\star(N) = N$.

2.1 A Simple Recursive CBT-Schedule Algorithm

A k-Queue CBT-Scheduler

Input: the $(N = 2^n)$-leaf CBT $\mathcal{T}_n$

1. Schedule the top $\lceil (n+1)/k \rceil$ levels of $\mathcal{T}_n$ using queue #k.
2. For each "leaf" of the tree scheduled in step 1, in turn, schedule the CBT rooted at that "leaf" by using queues #$1, \ldots,$ #$(k-1)$ to execute recursively the $(k-1)$-queue version of this algorithm.

See Fig. 2.

Analyzing the Algorithm. Since each queue is used to schedule (possibly many) CBT(s) of height $(\log N + 1)/k$, no queue need have capacity greater than $\lceil N^{1/k} \rceil$. As an immediate consequence, we have:

Fact 2.1 *For all positive integers $N = 2^n$ and $k \leq n$,*

$$\mathcal{Q}_k(N) \leq \lceil N^{1/k} \rceil. \tag{1}$$

Note that when $k = \log N$ in this family of algorithms, each queue has constant capacity. At this point, the queues are collectively simulating the action of a single stack, executing the CBT in a depth-first, hence lazy, regimen. (One can verify by a simple adaptation of the argument in [12] that the cumulative capacity of the queues when $k = \log N$ cannot be improved by more than a constant factor.)

2.2 The Real Control-Memory Tradeoffs

The possible tradeoff suggested in Fact 2.1 (i.e., the possibility that there exist lower bounds that come close to the upper bounds (1)) does indeed exist. In the next three sections we prove the following bounds, which are refinements of the rough bounds we have stated earlier.

A. The Upper Bound for General BTs

In Section 3, we show that the upper bound (1) holds for arbitrary BTS, to within a factor of 2.

Theorem 2.1 *For all positive integers N and $k \leq \log N$, the minimax per-queue capacity of algorithms that use k queues to schedule an N-leaf BT satisfies*

$$\mathcal{Q}_k(N) \leq 2N^{1/k} + 1 - \frac{(1/2)N^{1-1/k} - 2N^{1/k}}{\lfloor N^{1-1/k} \rfloor}. \tag{2}$$

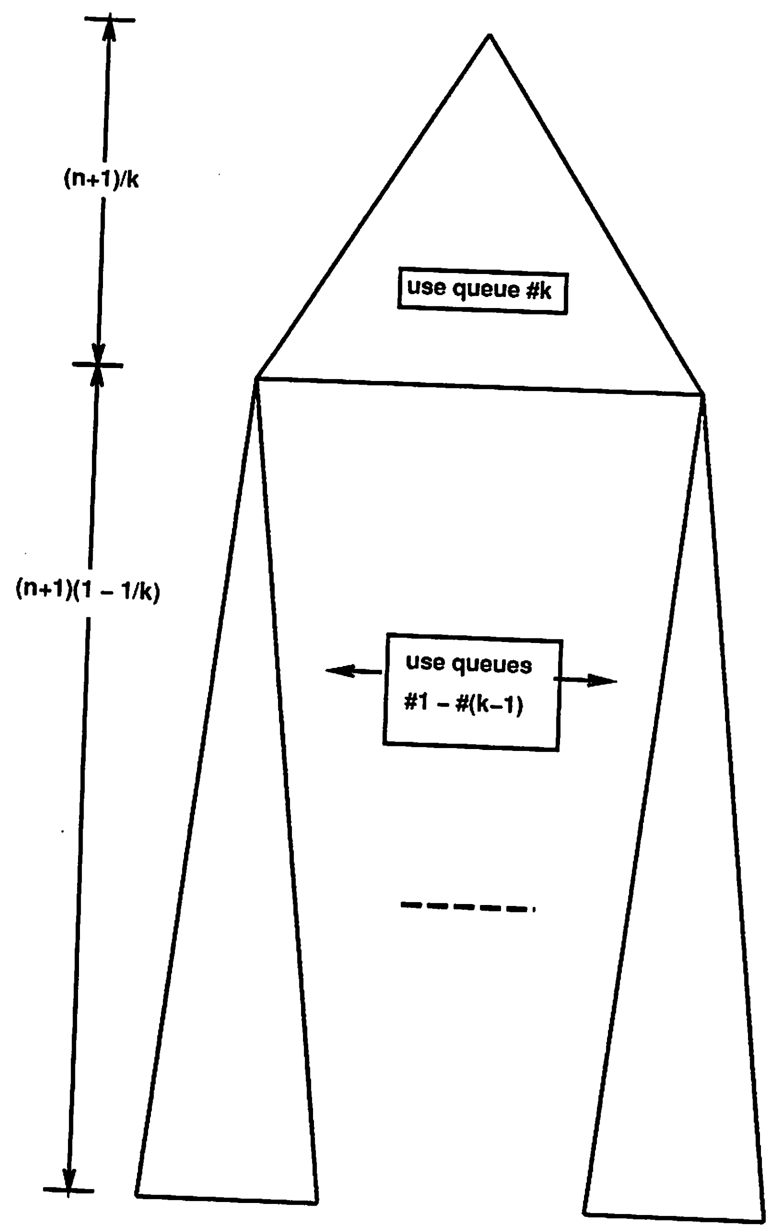

Figure 2: *An indication of queue utilization in a capacity-saving two-queue schedule for the* 2^n*-leaf CBT.*

B. The Upper Bound for Complete BTs

In Section 4, we demonstrate that, somewhat surprisingly, the upper bound (1) can be improved for complete BTs.

Theorem 2.2 *For all positive integers $N = 2^n$ and $k \le n$, the minimax per-queue capacity of algorithms that use k queues to schedule the N-leaf CBT satisfies*

$$\mathcal{Q}_k^\star(N) \le (4k)^{1-1/k} \frac{N^{1/k}}{\log^{1-1/k} N}. \tag{3}$$

C. The Lower Bounds

Our lower bounds for scheduling BTs derive from the following bound which is proved in Section 5. For positive integers k, W, and D, let $\mathcal{Q}_k'(W, D)$ denote *the minimax per-queue capacity when k queues are used to schedule a BT having width W and depth $\le D$.*

Theorem 2.3 *Let $\mathcal{T}$ be an N-leaf BT having width W and depth D. For all $k \le \log N$: given any k-queue algorithm that schedules $\mathcal{T}$, at least one queue must have capacity*

$$\mathcal{Q}_k'(W, D) \ge \left(\frac{1}{k!2^{k-1}}\right)^{1/k} \frac{W^{1/k}}{D^{1-1/k}}. \tag{4}$$

Since $W \ge (2N-1)/D$ for an N-leaf BT having width W and depth D, Theorem 2.3 immediately yields

Corollary 2.1 *(The Lower Bound for General BTs)*
Let $\mathcal{T}$ be an N-leaf, depth-D BT. For all $k \le \log N$: given any k-queue algorithm that schedules $\mathcal{T}$, at least one queue must have capacity no smaller than

$$\left(\frac{1}{k!2^{k-1}}\right)^{1/k} \frac{(2N-1)^{1/k}}{D} \ge \frac{1}{k}\frac{(2N-1)^{1/k}}{D}.$$

Since there exist N-leaf BTs of depth $\log N + 1$, it follows that

$$\mathcal{Q}_k(N) \ge \frac{1}{k}\frac{(2N-1)^{1/k}}{\log N + 1}.$$

Since an $(N = 2^n)$-leaf CBT has width $W = N$ and depth $D = n + 1$, we get a stronger corollary of Theorem 2.3 for CBTs.

Corollary 2.2 ***(The Lower Bound for Complete BTs)***
For all positive integers $N = 2^n$ and $k \leq n$: given any k-queue algorithm that schedules the N-leaf CBT, at least one of the queues must have capacity

$$\mathcal{Q}_k^*(N) \geq \frac{1}{k} \frac{N^{1/k}}{(\log N + 1)^{1-1/k}}.$$

Corollaries 2.1 and 2.2 yield the claimed lower bounds on $\mathcal{Q}_k(N)$ and $\mathcal{Q}_k^*(N)$, respectively.

D. The Fixed-k Tradeoff for Complete BTs

It is worth remarking that for fixed k, the bounds of Theorems 2.2 and Corollary 2.2 are within constant factors of each other.

Corollary 2.3 *For any fixed constant k, there exist positive constants c_1 and c_2 such that, for all $N = 2^n$,*

$$c_1 \frac{N^{1/k}}{\log^{1-1/k} N} \leq \mathcal{Q}_k(N) \leq c_2 \frac{N^{1/k}}{\log^{1-1/k} N}.$$

3 The Upper Bounds for General BTs

This section is devoted to proving the upper bounds of Theorem 2.1, via a recursive family of BT-scheduling algorithms that match the memory requirements of the CBT-scheduling algorithms of Section 2.1 (to within constant factors). Let us be given an N-leaf BT $\mathcal{T}$.

Preprocessing. Label each arc of $\mathcal{T}$ with the number of leaves in the sub-BT of $\mathcal{T}$ rooted at the node that the arc enters. This can be accomplished using either an $O(N)$-time sequential algorithm or an $O(\log N)$-time parallel algorithm, depending on the resources available to the scheduler.

3.1 The k-Queue Algorithm

Use queue #k to start executing the nodes of $\mathcal{T}$, from the root, level by level. As the scheduler executes a node v that has two children, it scans the labels of the arcs leaving v.

- If the label of one arc leaving v is $> \lfloor \frac{1}{2} N^{1-1/k} \rfloor$, and the label of the other arc leaving v is $\leq \lfloor \frac{1}{2} N^{1-1/k} \rfloor$, then the scheduler
  1. enqueues the arc leading to the big subtree in queue #k
  2. immediately schedules the smaller subtree, using queues #$1, \ldots,$ #$(k-1)$ in a recursive invocation of the $(k-1)$-queue version of this algorithm.
- If the labels of both arcs leaving v are $\leq \lfloor \frac{1}{2} N^{1-1/k} \rfloor$, and the label of the arc entering v is $> \lfloor \frac{1}{2} N^{1-1/k} \rfloor$, then the scheduler immediately schedules the subtree rooted at node v, using queues #$1, \ldots,$ #$(k-1)$ in a recursive invocation of the $(k-1)$-queue version of this algorithm.

3.2 Analyzing the k-Queue Algorithm

In order to assess the memory requirements of the algorithm, we isolate the portions of the scheduled tree $\mathcal{T}$ that are executed under the control of each of the k queues. Specifically, in order to bound the capacity of queue #k, we prune it by removing all subtrees that are processed using queues #$1, \ldots,$ #$(k-1)$. We claim that the pruned version of $\mathcal{T}$ has no more than

$$2N^{1/k} + 1 - \frac{(1/2)N^{1-1/k} - 2N^{1/k}}{\lfloor N^{1-1/k} \rfloor}$$

leaves. In order to see this, note that the "leaves" of the pruned version of $\mathcal{T}$ all have incoming arcs with labels $> \lfloor \frac{1}{2} N^{1-1/k} \rfloor$, meaning that the subtree rooted at each of these "leaves" has $> \lfloor \frac{1}{2} N^{1-1/k} \rfloor$ leaves. Since $\mathcal{T}$ has N leaves in all, this bound on the number of leaves in each of the removed subtrees implies that there are fewer than

$$\frac{N}{\lfloor (1/2)N^{1-1/k} \rfloor} \tag{5}$$

removed subtrees. But, the roots of these subtrees comprise the "leaves" of the pruned version of $\mathcal{T}$, and it is the pruned version which is the tree scheduled using queue #k. The claimed capacity of queue #k now follows from Lemma 2.1 and elementary bounds for the fraction (5).

In order to bound the capacities of queues #$1, \ldots,$ #$(k-1)$, note that each recursive invocation of the algorithm using those queues schedules a BT having no more than $\lfloor N^{1-1/k} \rfloor$ leaves. It follows that

$$\mathcal{Q}_k(N) \leq \max\left(2N^{1/k} + 1 - \frac{(1/2)N^{1-1/k} - 2N^{1/k}}{\lfloor N^{1-1/k} \rfloor},\ \mathcal{Q}_{k-1}(\lfloor N^{1-1/k} \rfloor)\right),$$

whence the Theorem.

4 The Upper Bounds for Complete BTs

This section is devoted to proving the upper bound of Theorem 2.2, via a recursive family of CBT-scheduling algorithms that have better memory requirements than the family of schedulers of Section 2.1.

For purely technical reasons, our algorithmic strategy inverts the question we really want to solve. Specifically, instead of starting with a target number N of leaves and asking how small a queue-capacity is sufficient to schedule an N-leaf CBT, we start with a target queue-capacity Q and ask how large a CBT we can schedule using queues of capacity Q. We proceed, therefore, by considering the quantity $\mathcal{N}_k(Q)$ which denotes *the maximum number of leaves in a CBT that can be scheduled using k queues, each of capacity Q*. By deriving a lower bound on the quantity $\mathcal{N}_k(Q)$, we can infer an upper bound on the dual quantity $\mathcal{Q}_k^\star(N)$. In order to avoid a proliferation of floors and ceilings in our calculations, we assume henceforth that Q is a power of 2; this assumption will be seen to affect only constant factors.

Since the algorithm that establishes the general case of Theorem 2.2 is somewhat complex, we present first the algorithm for the case of two queues ($k = 2$), which already exposes the subtlest ideas in the general algorithm.

4.1 The Case $k = 2$

Our two-queue CBT-scheduling algorithm operates in three phases which we describe now in rough terms. Let $\lambda(Q) =_{def} \lfloor \log(\log Q - 1) \rfloor$. In the first phase, the algorithm uses queue #2 to schedule (the execution of) the top $\lambda(Q) + 1$ levels of a CBT, retaining the "leaves" from the last level in the queue. In the second phase, the algorithm staggers executing these "leaves" (hence, removing them from queue #2) with beginning to use queue #1 to schedule the middle $\log Q - 1$ levels of the CBT. By the end of the second phase, queue #2 has been emptied, hence is available for reuse. In the third phase, the algorithm staggers using queue #1 to schedule the remainder of the middle $\log Q - 1$ levels of the CBT with using queue #2 to schedule the bottom $\log Q$ levels. This latter staggering proceeds by having queue #2 schedule a Q-leaf CBT rooted at each middle-tree "leaf" from queue #1. To assist the reader in understanding the ensuing detailed description of the algorithm, we depict in Fig. 3 the ultimate usage pattern of the two queues.

A. An Efficient Two-Queue CBT-Scheduling Algorithm

Phase 1: The Top of the Tree.

In this phase, we use queue #2 to schedule (the execution of) the top $\lambda(Q) + 1$ levels

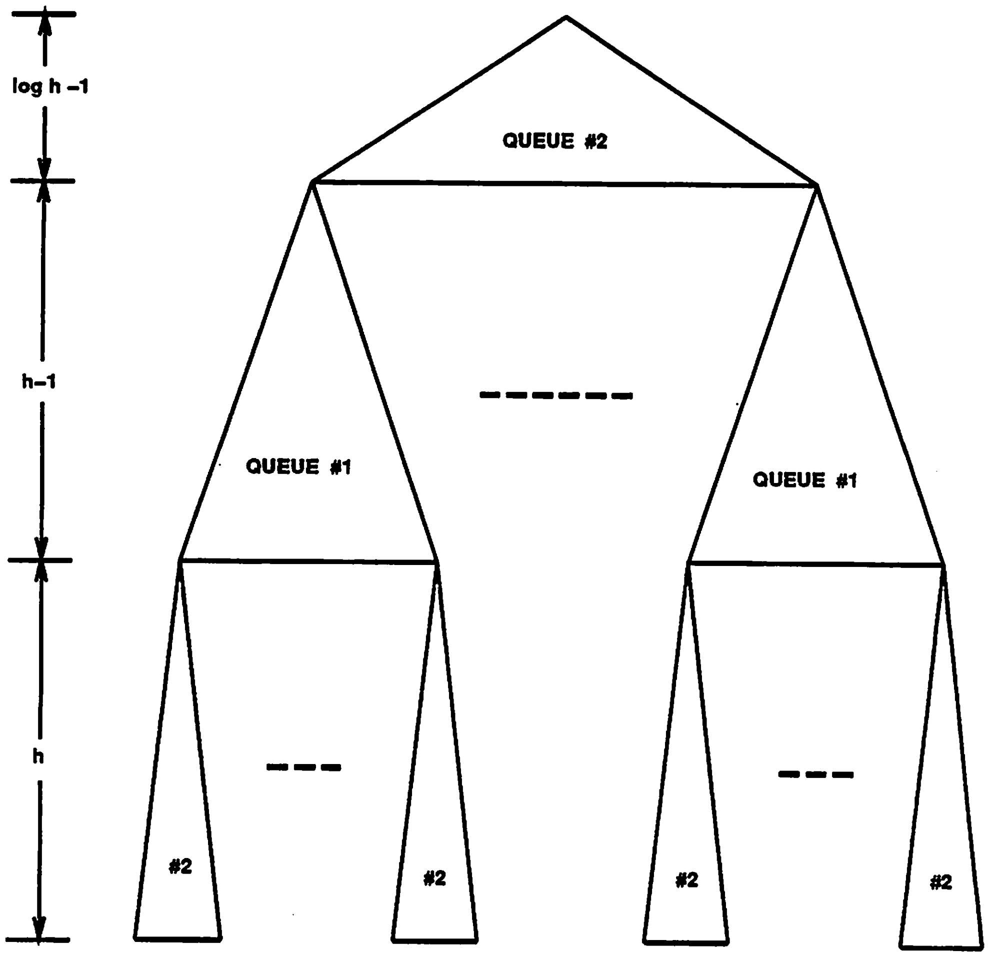

Figure 3: *The target utilization of two capacity-Q queues when scheduling the execution of a "big" CBT;* $h = \log Q$.

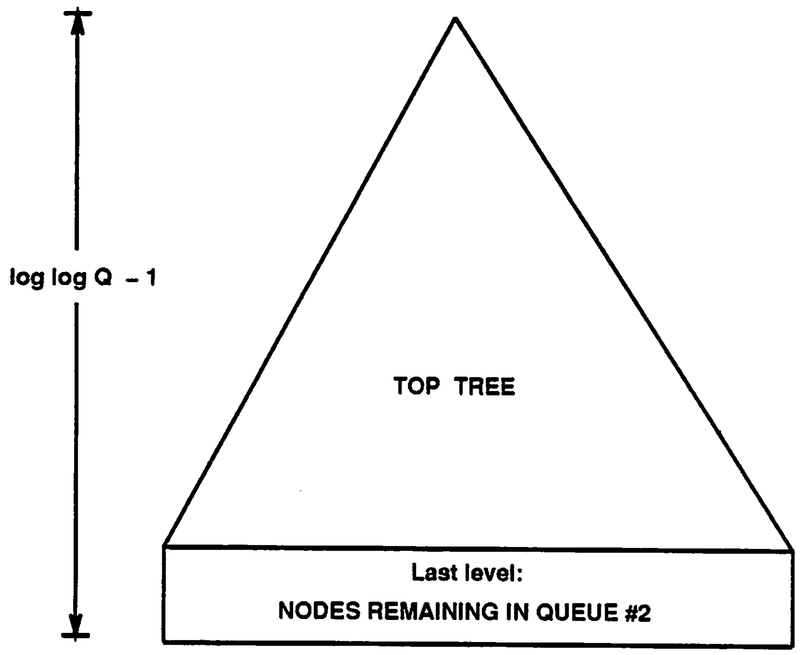

Figure 4: *Scheduling the execution of the top of the CBT.*

of the CBT we are scheduling, using the breadth-first regimen that is the unique way a single queue can schedule a CBT (cf. Fact 1.1). At the end of this phase, queue #2 will contain $2^{\lambda(Q)}$ nodes. We make the transition into Phase 2 of the algorithm by considering each node in queue #2 as the root of a "middle" CBT (which will have $Q/2$ "leaves"). See Fig. 4.

Phase 2: The Middle of the Tree.
In this phase, we use queue #1 to schedule (the execution of) the *middle trees* that comprise the next $\log Q - 1$ levels of the CBT we are scheduling. This is the most complicated of the three phases, in that these middle trees get executed in a staggered manner, in two senses. First, the executions of the $2^{\lambda(Q)}$ middle trees get interleaved in the schedule we are producing. Second, the execution of the middle trees is interleaved with segments of Phase 3, wherein the bottom trees are executed.

We describe first the initial portion of Phase 2, i.e., the portion before the phase gets interrupted by segments of Phase 3.

Execute the first node from queue #2, which is level 0 (i.e., the root) of the first middle tree; place the children of this node in queue #1. Next, proceed through the following iterations; see Fig. 5.

Step 1. Begin the first middle tree.

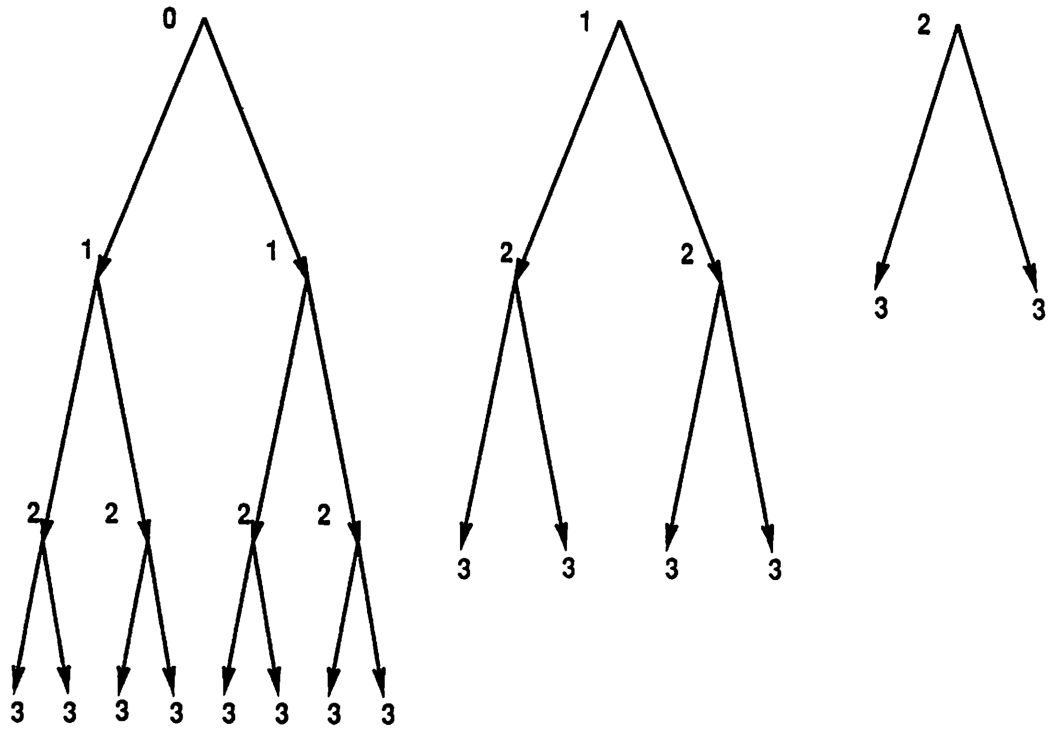

Figure 5: *The initial steps of Phase 2.*

Step 1.1. Use queue #1 to schedule the execution of level 1 of the first middle tree.

Step 1.2. Execute the root (level 0) of the second middle tree (removing that node from queue #2); place the children of this root in queue #1.

Step 2. Continue the first middle tree; begin the second middle tree.

Step 2.1. Use queue #1 to schedule the execution of level 2 of the first middle tree.

Step 2.2. Use queue #1 to schedule the execution of level 1 of the second middle tree.

Step 2.3. Execute the root (level 0) of the third middle tree (removing that node from queue #2); place the children of this root in queue #1.

Step 3. Continue the first and second middle trees; begin the third middle tree.

Step 3.1. Use queue #1 to schedule the execution of level 3 of the first middle tree.

Step 3.2. Use queue #1 to schedule the execution of level 2 of the second middle tree.

Step 3.3. Use queue #1 to schedule the execution of level 1 of the third middle tree.

Step 3.4. Execute the root (level 0) of the fourth middle tree (removing that node from queue #2); place the children of this root in queue #1.

● ● ●

Step $(2^{\lambda(Q)} - 1)$. Finish the first middle tree; continue the second through next-to-last middle trees; begin the last middle tree.

Step $(2^{\lambda(Q)} - 1)$.1. Use queue #1 to schedule the execution of level $\log Q - 2$ of the first middle tree.

Step $(2^{\lambda(Q)} - 1)$.2. Use queue #1 to schedule the execution of level $\log Q - 3$ of the second middle tree.

● ● ●

Step $(2^{\lambda(Q)} - 1).(2^{\lambda(Q)} - 1)$. Use queue #1 to schedule the execution of level 1 of the second from last middle tree.

Step $(2^{\lambda(Q)} - 1).(2^{\lambda(Q)})$. Execute the root (level 0) of the last middle tree (removing that node from queue #2); place the children of this root in queue #1.

After the ith step of Phase 1, the following progress has been made: $i+1$ of the nodes that began the Phase in queue #2 have been executed; the first $i+1$ levels (i.e., levels $0, \ldots, i$) of the first middle tree have been executed, and level $i+1$ of the tree resides queue #1; the first i levels of the second middle tree have been executed, and level i of the tree resides in queue #1 (behind the nodes from the first middle tree); the first $i-1$ levels of the third middle tree have been executed, and level $i-1$ of the tree resides in queue #1 (behind the nodes from the second middle tree); and so on. When Phase 1 is completed (i.e., after the $(2^{\lambda(Q)}-1)$th step of the Phase), all $2^{\lambda(Q)}$ of the nodes that began the Phase in queue #2 have been executed, so queue #2 is completely emptied, hence is available for reuse. Queue #1, on the other hand, contains some number of nodes that is guaranteed to be less than Q. Specifically, queue #1 contains

$$2^{2^{\lambda(Q)}} \leq Q/2$$

nodes from the first middle tree, and, in general, it contains only half as many nodes from the $(i+1)$th middle tree as it does from the ith; there are, of course,

$$2^{\lambda(Q)} \leq \log Q - 1$$

middle trees, hence

$$\sum_{i=1}^{2^{\lambda(Q)}} 2^i \leq Q - 2$$

nodes in queue #1. See Fig. 6.

We have now completely executed the first middle tree and partially executed all the other middle trees. Ultimately, we shall continue to use queue #1 in the same interleaved, power-of-2 decreasing manner as described here, to schedule the remaining middle trees for execution. First, though, we initiate Phase 3 in which queue #2 is used to schedule the bottom levels of the CBT for execution. It is important to begin Phase 3 now, because some of the contents of queue #1 must be unloaded at this point, in order to make room for the remaining levels of the remaining middle trees.

Phase 3: The Bottom of the Tree.
Phase 3 is partitioned into two subphases. In Phase 3a, we begin viewing the "leaves" of the middle trees as the roots of *bottom trees* — each being a CBT with $\mathcal{N}_1(Q) = Q$ leaves. In Phase 3b, we continue using the regimen of Phase 2 to schedule the middle trees.

Phase 3a. This subphase is active whenever the nodes at the front of queue #1 come from level $2^{\lambda(Q)}$ of a middle tree (which is the last level to enter queue #1). During the subphase, we iteratively execute a single node — call it node v — from queue #1, and we

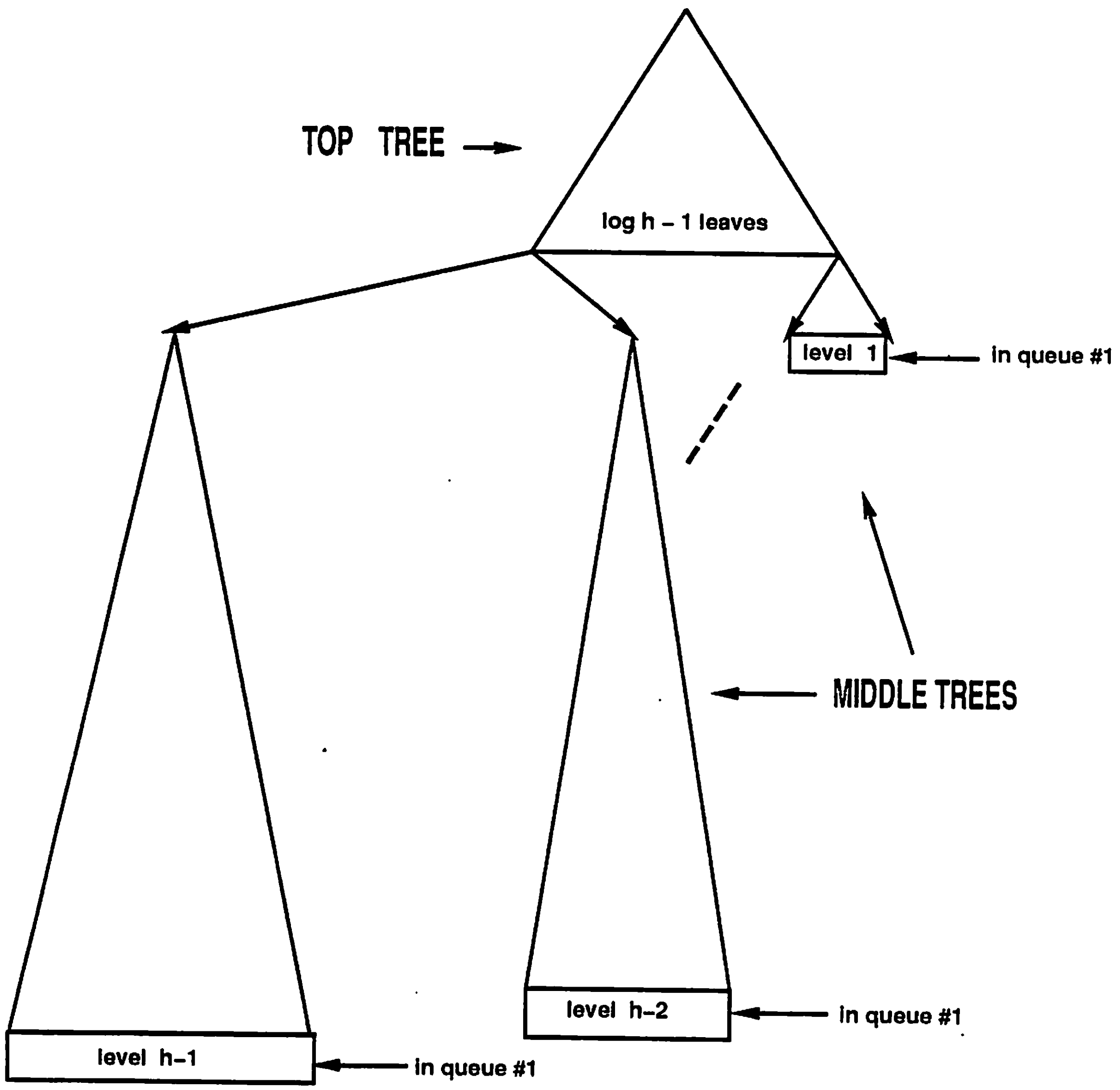

Figure 6: *The situation after Phase 2.*

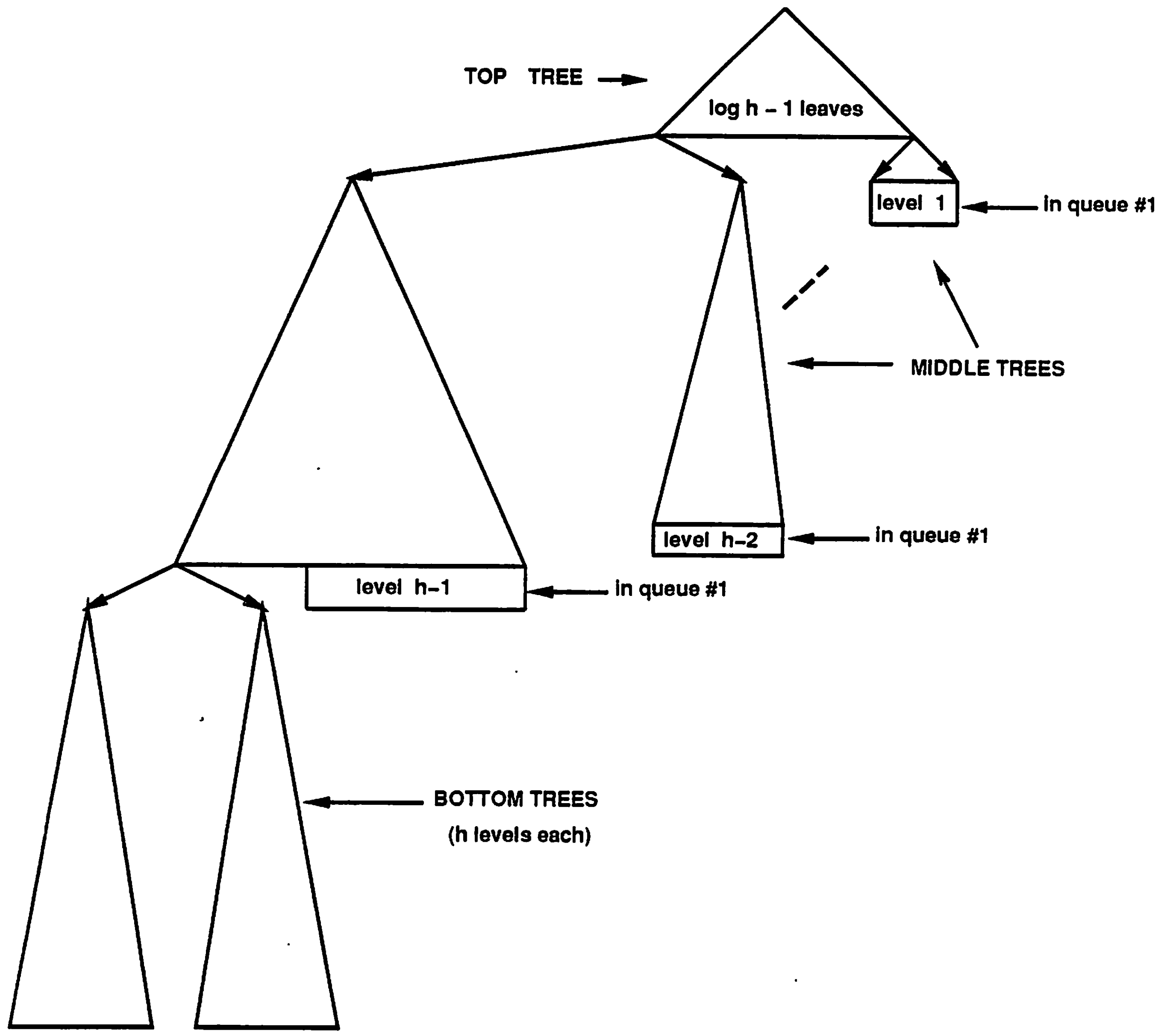

Figure 7: *The situation after Phase 3a begins: the first two bottom trees have been executed.*

use queue #2 to schedule the CBT on Q leaves rooted at node v (using the breadth-first regimen, of course). See Fig. 7.

Phase 3b. This subphase is active whenever the nodes at the front of queue #1 *do not* come from level $2^{\lambda(Q)}$ of a middle tree. During the subphase, we perform one more step of Phase 2, to extend the executed segment of the middle tree. To illustrate our intent, the instance of Subphase 3b that is executed immediately after the first round of executions of Subphase 3a (wherein the leftmost $Q/2$ bottom trees are executed) has the following form.

Step $(2^{\lambda(Q)}+1)$**.** Finish the second middle tree; continue the third through last middle trees.

- **Step** $(2^{\lambda(Q)}+1)$**.1.** Use queue #1 to schedule the execution of level $2^{\lambda(Q)}$ of the second middle tree.
- **Step** $(2^{\lambda(Q)}+1)$**.2.** Use queue #1 to schedule the execution of level $2^{\lambda(Q)}-1$ of the third middle tree.

  • • •

- **Step** $(2^{\lambda(Q)}+1).(2^{\lambda(Q)}-1)$**.** Use queue #1 to schedule the execution of level 2 of the second from last middle tree.
- **Step** $(2^{\lambda(Q)}+1).(2^{\lambda(Q)})$**.** Use queue #1 to schedule the execution of level 1 of the last middle tree.

See Fig. 8.

B. The Analysis

Correctness being (hopefully) clear, we need only assess how much CBT we are getting for given queue-capacity Q. There are

$$2^{\lambda(Q)} > \frac{1}{2}(\log Q - 1)$$

top-tree leaves, hence, at least $\frac{1}{4}Q(\log Q-1)$ middle-tree leaves, hence at least $\frac{1}{4}Q^2(\log Q-1)$ CBT leaves. It follows that

$$\mathcal{N}_2(Q) \geq \frac{1}{4}Q^2(\log Q - 1).$$

Inverting this inequality to obtain the desired upper bound on $\mathcal{Q}_2^*(N)$, we find that

$$\mathcal{Q}_2^*(N) \leq 2\left(\frac{N}{\log N}\right)^{1/2}.$$

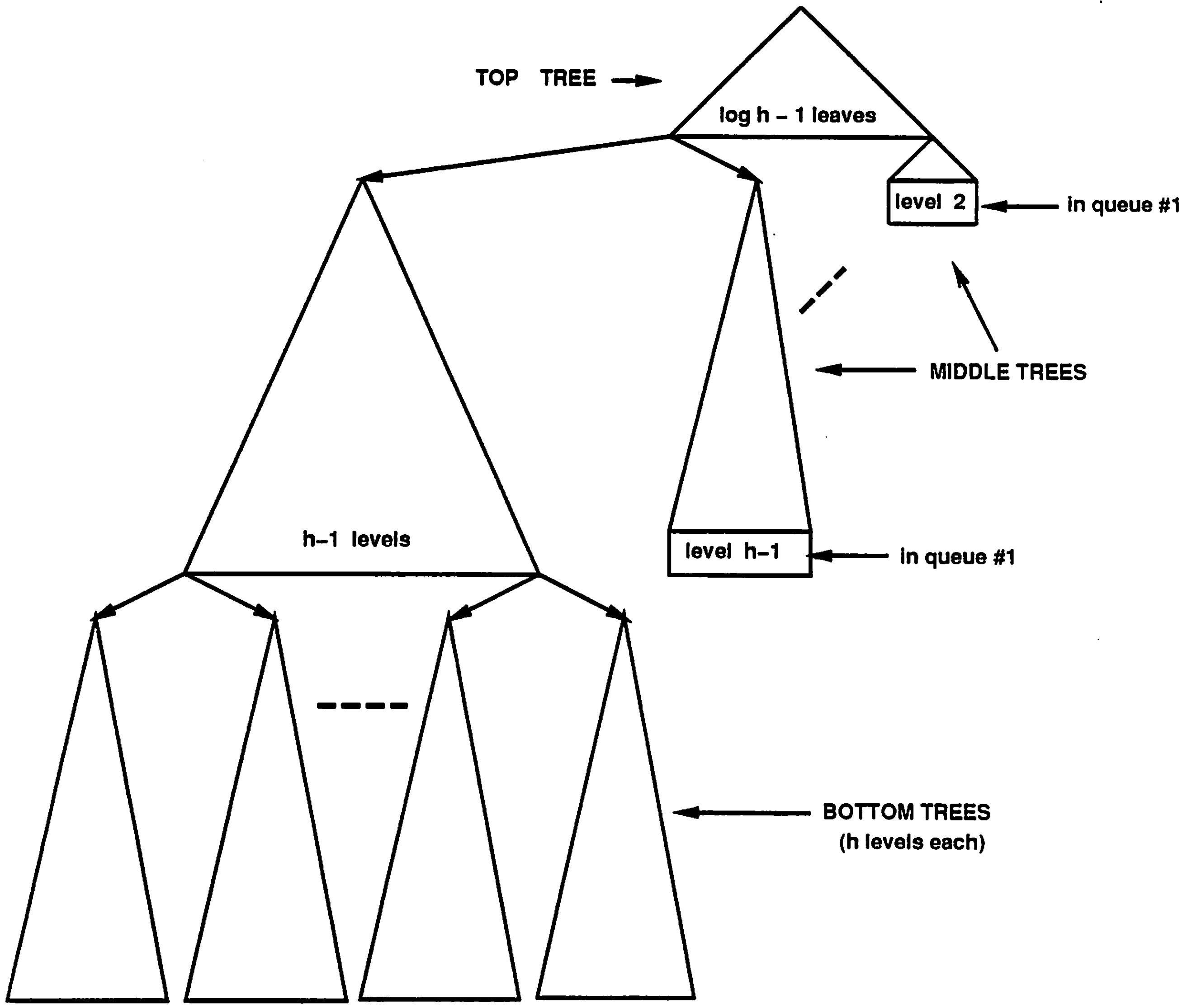

Figure 8: *The situation after Phase 3b begins: the first set of bottom trees have been executed; the second middle tree has been completed.*

4.2 The Case of General k

We now show how to generalize our two-queue CBT-scheduler to obtain multiqueue CBT-schedulers for arbitrary numbers of queues.

A. The Algorithm

Our k-queue CBT-scheduling algorithm uses Phases 1, 2, and 3b of our two-queue scheduling algorithm directly. It modifies only Phase 3a, in the following way.

Phase 3a. This subphase is active whenever the nodes at the front of queue #1 come from level $\log Q - 2$ of a middle tree (which is the last level to enter queue #1). During the subphase, we iteratively execute a single node — call it node v — from queue #1, and we use queues #2, ..., #k to schedule a CBT on Q leaves, rooted at node v, *using a recursive invocation of the $(k-1)$-queue version of this algorithm.*

Note that the two-queue CBT-scheduler of the previous subsection can, in fact, be obtained via this recursive strategy, from the base case $k = 1$.

As with the case $k = 2$, queues #2, ..., #k are all available for this recursive call because: (a) the last "leaf" of the top tree is extracted from queue #2 for execution just before the first "leaf" of the leftmost middle tree is extracted from queue #1 for execution; (b) queues #3, ..., #k are not used at all with the top or middle trees above this level of the final CBT.

B. The Analysis

Correctness being (hopefully) obvious, we need consider only how many leaves the CBT we have scheduled has, as a function of the given queue-capacity Q. Because each recursive call to our CBT-scheduler generates its own "top" tree and its own set of "middle" trees, but uses one fewer queue than the previous call, it is not hard to verify that this number is given by the recurrence

$$\begin{aligned}\mathcal{N}_1(Q) &= Q \\ \mathcal{N}_{k+1}(Q) &\geq \frac{1}{4}Q(\log Q - 1)\mathcal{N}_k(Q)\end{aligned}$$

Easily, this system yields the following solution, which holds for all $k \geq 1$.

$$\mathcal{N}_k(Q) \geq Q\left(\frac{1}{4}Q\log Q\right)^{k-1}.$$

For our ends, we invert this relation, to get the sought upper bound on $Q_k^*(N)$. □

5 The Lower Bounds in the Tradeoff

This section is devoted to proving the lower bound in Theorem 2.3.

Let us be given an N-leaf BT $\mathcal{T}$ having width W and depth D. Call a level of $\mathcal{T}$ that has W nodes a *wide* level. Consider the action of an arbitrary k-queue schedule for executing $\mathcal{T}$.

Our analysis of the given schedule is based on parsing the sequence of node-executions prescribed by the schedule into *phases.* Recall that the action of executing a node and loading its outgoing arcs into queues is a single atomic action.

Define *Phase* 0 to be that part of the process wherein the root of $\mathcal{T}$ (which must be the first node in the schedule) is executed and its outgoing arcs loaded onto queues.

Inductively, define *Phase* $i+1$ to be that part of the process that completes the execution of all nodes whose incoming arcs were loaded into queues during Phase i. In other words, Phase $i+1$ continues as long as some queue still contains an arc that was put there during Phase i; the Phase ends when the last of these Phase-i "legacies" has been executed.

One verifies by a straightforward induction that all nodes on level i of $\mathcal{T}$ must have been executed by the end of Phase i. The following Fact is an immediate consequence of this inference.

Fact 5.1 *There are at most D phases in the sequence of node-executions.*

Fact 5.1 has the following corollary.

Fact 5.2 *There must be a phase during which at least W/D nodes from a wide level of $\mathcal{T}$ are executed. Call such a phase* long.

Now look at what happens during a phase of an execution of $\mathcal{T}$ that minimizes the capacity of the largest-capacity queue. The phase starts with some arcs residing within the k queues — at most $\mathcal{Q}'_k(W,D)$ per queue. As we noted in Fact 1.2, all of the nodes entered by these arcs are independent in $\mathcal{T}$. Moreover, by definition of "phase," all of these nodes must be executed by the end of the phase. We can, therefore, characterize what the portion of $\mathcal{T}$ that is executed during a phase looks like.

Fact 5.3 *The nodes that are executed during a phase of the execution of $\mathcal{T}$ form a forest of BTs rooted at the $\leq k\mathcal{Q}'_k(W,D)$ nodes whose incoming arcs resided in the k queues at the start of the phase.*

Let us henceforth focus on a specific long phase in the given schedule. Now, the widest BT in the forest executed during this phase is at least as wide as the average BT in the forest. Combining Fact 5.2 with Fact 5.3, the average width of a BT in this forest must be at least $W/(kD\mathcal{Q}'_k(W,D))$, since nodes that reside on the same level of $\mathcal{T}$ must reside on the same level of any sub-BT of $\mathcal{T}$ that contains them. We conclude, therefore, the following bound.

Fact 5.4 *At least one of the BTs in the forest of nodes executed during a long phase must have width no smaller than*

$$\frac{W}{kD\mathcal{Q}'_k(W,D)}.$$

Next, note that, by definition of "phase," there must be some queue — say, queue $\#m$ — whose sole contributions to the set of nodes executed during our long phase are the nodes whose incoming arcs reside in this queue at the beginning of the phase. This is because the phase ends when the last node whose incoming arc was enqueued during the previous phase is executed; we are identifying queue $\#m$ as the source of this last incoming arc. Now, queue $\#m$ started the phase (as did every queue) with no more than $\mathcal{Q}'_k(W,D)$ arcs. As we noted earlier, each of these arcs enters the root of a BT whose nodes are executed during the long phase. Of all the nodes executed during the long phase, only these root nodes have incoming arcs that were enqueued in queue $\#m$. Therefore, if we remove all these queue $\#m$-nodes from the forest, then we partition each BT $\mathcal{T}'$ that is rooted at a node whose incoming arc came from queue $\#m$ into two BTs, call them $\mathcal{T}'_1$ and $\mathcal{T}'_2$. Clearly, at least one of $\mathcal{T}'_1$ and $\mathcal{T}'_2$ has at least half the width of $\mathcal{T}'$. It follows, therefore, that

Fact 5.5 *The forest of nodes executed during the long phase must, after all the nodes from queue #m are removed, contains a BT of width no smaller than*

$$\frac{W}{2kD\mathcal{Q}'_k(W,D)}$$

that is scheduled for execution by only $k-1$ of the queues.

We infer from Fact 5.5 the recurrent lower bound

$$\mathcal{Q}'_k(W,D) \geq \mathcal{Q}'_{k-1}\left(\frac{W}{2kD\mathcal{Q}'_k(W,D)},\ D\right) \tag{6}$$

whose initial case ($k=1$)

$$\mathcal{Q}'_1(W,D) = W \tag{7}$$

is resolved in Lemma 2.1. We solve this recurrence by induction. Specifically, note that the expression in inequality (4) reduces to equation (7) for the case $k = 1$. Direct calculation verifies that inequality (4) is "preserved" by recurrence (6); i.e., if we assume, for induction, that

$$\mathcal{Q}'_{k-1}(W,D) \geq \left(\frac{1}{(k-1)!2^{k-2}}\right)^{1/(k-1)} \frac{(DW)^{1/(k-1)}}{D},$$

then an application of recurrence (6) yields inequality (4), which is the lower bound of Theorem 2.3. □

6 Closing Remarks

We close the paper with some observations on directions for extending our work and on directions in which extensions are impossible. The extensions we know of (Section 6.1) concern queue-based scheduling algorithms for tree-dags; the impossibility results we know of (Section 6.2) concern queue-based scheduling algorithms for dags whose underlying graphs are not trees. A topic that we have not considered, which might be fruitful, is the possible existence of control-memory tradeoffs that might arise with schedulers that use other data structures (e.g., stacks) to manage tasks awaiting execution.

6.1 Extending Our Results on Scheduling Tree-Dags

The results we have reported here can be extended in a variety of ways.

A. A Better Lower Bound

A more careful analysis replaces the recurrent bound (6) by

$$\mathcal{Q}'_k(W,D) \geq \mathcal{Q}'_{k-1}\left(\frac{W}{(k+1)D\mathcal{Q}'_k(W,D)} - \mathcal{Q}'_k(W,D),\ D\right),$$

which solves to a marginally larger lower bound (by a constant factor). The reasoning behind this better recurrence is as follows. If we remove the nodes that came from queue $\#m$, we have left $\leq (k+1)\mathcal{Q}'_k(W,D)$ trees, each of which is generated by only $k-1$ of the queues. Since each of the nodes that came from queue $\#m$ could, in fact, come from a wide level of the big BT, removing these nodes could decrease the number of wide-level nodes laid out during this phase by $\leq \mathcal{Q}'_k(W,D)$. What this means is:

Fact 6.1 *The forest of BTs from a phase during which L nodes from a wide level of $\mathcal{T}$ are executed must contain a BT of width no smaller than*

$$\frac{L}{(k+1)\mathcal{Q}'_k(W,D)} - \mathcal{Q}'_k(W,D)$$

that is generated by only $k-1$ of the queues.

B. Extensions to Broader Classes of Tree-Dags

Arbitrary Fixed Node-Degrees. We have focussed here on *binary* tree-dags solely for the sake of definiteness; our results extend to tree-dags of any fixed branching factor, with only clerical modifications.

Root-to-Leaf Tree-Dags. We have focussed here on tree-dags whose arcs point from the root toward the leaves, thereby modeling a class of branching computations. The control-memory tradeoff that we have proved obtains also for tree-dags whose arcs point from the leaves toward the root, such as are used in many evaluative computations (e.g., evaluating arithmetic expressions or computing parallel-prefixes). This fact can be proved by fleshing out the details of the following indirect argument.

The formal framework of our study emerged from [4] - [7], [13] wherein queues are used to topologically sort graphs and dags. Indeed, the processes of topologically sorting a dag and scheduling it (in our sense) are isomorphic processes.[3] In order to make this isomorphism formal, one must complicate our framework slightly, to accommodate nodes whose in-degrees exceed unity. Two changes are required:

- When scheduling a general dag using (enabling and execution) tokens, a node v becomes eligible for execution when all arcs that enter v contain enabling tokens.
- When scheduling a general dag using queues, a node v becomes eligible for execution when all arcs that enter v are at the "fronts" of queues, in the sense of either being at the heads of queues or being behind other arcs that enter v.

Using insights and results in the cited sources, one can readily prove the following lemma (which does not occur in the sources). Underlying the proof is the fact that, if one takes any topological sort of a dag $\mathcal{G}$ and reverses the linearization of $\mathcal{G}$'s nodes, then one obtains a topological sort of the dag $\overleftarrow{\mathcal{G}}$ which is obtained from $\mathcal{G}$ by reversing the orientation of all arcs.

Lemma 6.1 *If the dag $\mathcal{G}$ can scheduled by a k-queue algorithm whose queues each have capacity $\leq C$, then so also can the dag $\overleftarrow{\mathcal{G}}$.*

[3]We noted this isomorphism earlier, in Section 1.2, for tree-dags.

6.2 Remarks on Scheduling General Dags

The control-memory tradeoff that we have exhibited in this paper is interesting because of its nonlinearity: roughly speaking, the *exponent* in the expression for the memory requirements decreases linearly with the increase in the number of queues. It is an inviting challenge to discover other classes of dags that admit nonlinear control-memory tradeoffs and, hopefully, to characterize the properties of those dags that enable such tradeoffs. While we have been unable to find either such classes or such properties, we have discovered two simple properties that preclude such tradeoffs.

A. Pebbling Number

Fact 6.2 *The cumulative capacity of the queues in a k-queue scheduling algorithm for a dag $\mathcal{G}$ can be no less than the "pebbling number" of $\mathcal{G}$, in the sense of [12] and its numerous successors.*

The validity of this principle is clear from our formulation of dag-scheduling in terms of (a nonstandard) type of pebble game. It is this principle that assures us (via the results in [12]) that we should not consider BT-schedulers with more than $\log N$ queues to schedule N-leaf BTs.

B. Separator Size

Fact 6.3 *The cumulative capacity of the queues in a k-queue scheduling algorithm for a dag $\mathcal{G}$ can be no less than the size of the smallest (arc-)separator of the dag into two disjoint subdags.*

The validity of this principle is clear once one notices that the (arc-)boundary between the sets of executed and unexecuted nodes of a dag — which must coreside in the queues of the scheduling algorithm — forms an (arc-)separator of the dag. It is this principle that assures us that mesh-pyramids do not admit nonlinear control-memory tradeoffs. We illustrate the argument for two- and three-dimensional mesh-pyramids.

The N-sink two-dimensional mesh-pyramid $\mathcal{M}_N^{(2)}$ has nodes $\{\langle i,j\rangle \mid 0 \leq i+j < N\}$; its arcs lead from each node $\langle i,j\rangle$, where $i+j < N-1$, to nodes $\langle i+1,j\rangle$ and $\langle i,j+1\rangle$. Easily, $\mathcal{M}_N^{(2)}$ can be executed, level by level, by a 1-queue scheduler whose queue has capacity $2N-2$. Since the smallest bisector of $\mathcal{M}_N^{(2)}$ must "cut" a number of arcs proportional to N (cf. [14]), the queues of any multiqueue scheduler must (collectively) contain this many arcs at the moment when precisely half the nodes of $\mathcal{M}_N^{(2)}$ have been executed.

The ($N = \frac{1}{2}n(n+1)$)-sink three-dimensional mesh-pyramid $\mathcal{M}_N^{(3)}$ has nodes $\{\langle i,j,k\rangle \mid 0 \leq i+j+k < n\}$; its arcs lead from each node $\langle i,j,k\rangle$, where $i+j+k < n-1$, to nodes $\langle i+1,j,k\rangle$, $\langle i,j+1,k\rangle$ and $\langle i,j,k+1\rangle$. $\mathcal{M}_N^{(3)}$, being nonplanar, cannot be executed by any 1-queue scheduler [7]. Easily, however, there is a 2-queue scheduler that executes $\mathcal{M}_N^{(3)}$ face by face, as follows. The scheduler uses queue #1 to execute face $k = 0$ of $\mathcal{M}_N^{(3)}$ "level by level;" while executing the nodes, the scheduler fills up queue #2 with the arcs that lead from face $k = 0$ to face $k = 1$. Inductively, the scheduler executes face $k = r$, where $r > 0$, "level by level," using queue #1 for the arcs that lie within that face, and emptying queue #2 of the arcs that come from face $k = r-1$; additionally, if $r < n$, while executing the nodes, the scheduler fills up queue #2 with the arcs that lead from face $k = r$ to face $k = r+1$. The capacity of queue #1 in this algorithm is $2n-2$ (which is achieved just before the algorithm executes the last "row" of face $k = 0$); the capacity of queue #2 is $\frac{1}{2}n(n+1)$ (which is achieved just after the algorithm executes the last "row" of face $k = 0$). Since the smallest bisector of $\mathcal{M}_N^{(3)}$ must "cut" a number of arcs proportional to n^2 (cf. [14]), the queues of any multiqueue scheduler must (collectively) contain this many arcs at the moment when precisely half the nodes of $\mathcal{M}_N^{(3)}$ have been executed.

ACKNOWLEDGMENTS. The authors thank Marc Snir for helpful comments in the early stages of this research, Li-Xin Gao for helpful discussions in the later stages, and Lenny Heath and Sriram Pemmaraju for helpful comments on the presentation.

The research of S. N. Bhatt was supported in part by NSF Grants MIP-86-01885 and CCR-88-07426, by NSF/DARPA Grant CCR-89-08285, and by Air Force Grant AFOSR-89-0382; the research of F. T. Leighton was supported in part by Air Force Grant AFOSR F49620-92-J-0125, and by ARPA Contracts ARPA N00014-91-J-1698, and ARPA N00014-92-J-1799; the research of A. L. Rosenberg was supported in part by NSF Grants CCR-90-13184 and CCR-92-21785. A portion of this research was done while S. N. Bhatt, F. T. Leighton, and A. L. Rosenberg were visiting Bell Communications Research.

References

[1] A. Gerasoulis and T. Yang (1992): A comparison of clustering heuristics for scheduling dags on multiprocessors. *J. Parallel and Distr. Comput.*

[2] A. Gerasoulis and T. Yang (1992): Scheduling program task graphs on MIMD architectures. Typescript, Rutgers Univ.

[3] A. Gerasoulis and T. Yang (1992): Static scheduling of parallel programs for message passing architectures. *Parallel Processing: CONPAR 92 - VAPP V. Lecture Notes in Computer Science 634*, Springer-Verlag, Berlin, pp. 601-612.

[4] L.S. Heath, F.T. Leighton, A.L. Rosenberg (1992): Comparing queues and stacks as mechanisms for laying out graphs. *SIAM J. Discr. Math. 5*, 398-412.

[5] L.S. Heath and S.V. Pemmaraju (1992): Stack and queue layouts of posets. Tech. Rpt. 92-31, VPI.

[6] L.S. Heath, S.V. Pemmaraju, A. Trenk (1993): Stack and queue layouts of directed acyclic graphs. In *Planar Graphs* (W.T. Trotter, ed.), American Mathematical Society, Providence, R.I., 5-11.

[7] L.S. Heath and A.L. Rosenberg (1992): Laying out graphs using queues. *SIAM J. Comput. 21*, 927-958.

[8] S.J. Kim and J.C. Browne (1988): A general approach to mapping of parallel computations upon multiprocessor architectures. *Intl. Conf. on Parallel Processing 3*, 1-8.

[9] M. Litzkow, M. Livny, M. Matka (1988): Condor - A hunter of idle workstations. *8th Ann. Intl. Conf. on Distributed Computing Systems.*

[10] D. Nichols (1990): *Multiprocessing in a Network of Workstations.* Ph.D. thesis, CMU.

[11] C.H. Papadimitriou and M. Yannakakis (1990): Towards an architecture-independent analysis of parallel algorithms. *SIAM J. Comput. 19*, 322-328.

[12] M.S. Paterson and C.E. Hewitt (1970): Comparative schematology. *Project MAC Conf. on Concurrent Systems and Parallel Computation*, ACM Press, 119-128.

[13] S.V. Pemmaraju (1992): *Exploring the powers of stacks and queues via graph layouts.* Ph.D. Thesis, Virginia Polytechnic Inst.

[14] A.L. Rosenberg (1979): Encoding data structures in trees. *J. ACM 26*, 668-689.

[15] 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.

[16] J. Yang, L. Bic, A. Nicolau (1991): A mapping strategy for MIMD computers. *Intl. Conf. on Parallel Processing 1*, 102-109.

[17] T. Yang and A. Gerasoulis (1991): A fast static scheduling algorithm for dags on an unbounded number of processors. *Supercomputing '91*, 633-642.

[18] T. Yang and A. Gerasoulis (1992): PYRROS: static task scheduling and code generation for message passing multiprocessors. *6th ACM Conf. on Supercomputing*, 428-437.