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 here a combinatorial problem that is motivated by a genre of architecture-independent scheduler for parallel computations. Such schedulers are often used, for instance, when computations are being done by a cooperating network of workstations. The results we obtain expose a control-memory tradeoff for such schedulers, when the computation being scheduled has the structure of a complete binary tree. The combinatorial problem takes the following form. Consider, for each integer $N = 2^n$, a family of n algorithms for linearizing the N-leaf complete binary tree in such a way that each nonleaf node precedes its children. For each $k \in \{1, 2, \ldots, n\}$, the kth algorithm in the family employs k FIFO queues to effect the linearization, in a manner specified later (cf., [1], [5] - [7]). In this paper, we expose a tradeoff between the number of queues used by each of the n algorithms — which we view as measuring the *control complexity* of the algorithm — and the *memory requirements* of the algorithms, as embodied in the required capacity of the largest-capacity queue. Specifically, we prove that, for each $k \in \{1, 2, \ldots, n\}$, the maximum per-queue capacity, call it $Q_k(N)$, for a k-queue algorithm that linearizes an N-leaf complete binary tree satisfies

$$\frac{e}{2k}\left(\frac{N}{\log^{k-1} N}\right)^{1/k} \leq Q_k(N) \leq 8k^{1-1/k}\left(\frac{N}{\log^{k-1} N}\right)^{1/k}.$$

Authors' Mailing Addresses:

S.N. Bhatt: Bell Communications Research, 435 South St., Morristown, NJ 07960

F.R.K. Chung: Bell Communications Research, 435 South St., Morristown, NJ 07960

F.T. Leighton: Dept. of Mathematics and Lab. for Computer Science, MIT, Cambridge, MA 02139

A.L. Rosenberg: Dept. of Computer Science, University of Massachusetts, Amherst, MA 01003

1 Introduction

1.1 Overview

We study here the resource requirements of a class of algorithms for scheduling parallel computations. Our main results expose a tradeoff between the two major resources the algorithms consume.

The Computing Environment. We are interested in schedulers that operate in a *client-server mode*, where the processors are the clients, and the server is the scheduler. One encounters such schedulers, for example, in systems that use networks of workstations for parallel computation; cf. [9], [10], [12]. 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. [2] - [4], [8], [11], [13] - [15]. 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 there is no need in this scenario for processors to operate synchronously, either individually or as a group.

The Computational Load. Our particular focus is on dags that are complete binary trees whose edges are oriented from the root toward the leaves. Such dags represent the data dependencies of certain types of branching computations.

Scheduling Regimens and Scheduler Structure. Our goal is to expose a tradeoff between the control complexity of our schedulers and their memory requirements — within the context of the just specified computation load. Toward this end, we must specify enough of the structure of a scheduler to identify its control complexity and memory requirements. 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 tasks that become eligible for execution at a given moment are loaded independently onto the FIFO queues; the task that is assigned to the next requesting processor is chosen from among those that are at the heads of the queues. Of particular interest is the fact that, for our computational load, as one increases the number of queues that make up the scheduler, one has the option of making the overall scheduling algorithm proceed from an *eager* 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. This spectrum of control options is realized by incorporating successively more FIFO queues into the scheduler, from a single queue at the eager end of the spectrum, to a number of queues that is logarithmic in the size of the tree-dag at the lazy end of the spectrum. While the control complexity of the scheduler increases as one incorporates successively more queues, one can show that the memory requirements — as measured by the maximum number of eligible tasks that are awaiting execution — decrease concomitantly. The contribution of this paper is to establish and quantify this control-memory tradeoff rigorously. Specifically, we establish the following upper and lower bounds on the maximum per-queue memory capacity for a k-queue scheduling algorithm scheduling an N-leaf complete binary tree, call this quantity $\mathcal{Q}_k(N)$:[1]

$$\frac{e}{2k}\left(\frac{N}{\log^{k-1} N}\right)^{1/k} \leq \mathcal{Q}_k(N) \leq 8k^{1-1/k}\left(\frac{N}{\log^{k-1} N}\right)^{1/k}.$$

1.2 The Formal Problem

While the formal setting could be framed in terms of general computation dags (directed acyclic graphs) and general scheduling algorithms for dags, we save space by specializing our development to the class of dags that we shall actually be studying, namely, binary tree dags — which represent a class of branching computations. The reader should note that our focus on *binary* trees here is for the sake of definiteness; our results extend easily to any other fixed branching factor.

Binary Tree Dags. A *binary tree dag* (*BT*, for short) is a directed acyclic graph whose node-set is a *prefix-closed* set of binary strings: 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 root is the unique node at level 0). The *width* of a BT is the maximum number of nodes at any level.

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 $\leq n$. There are N nodes at level n of $\mathcal{T}_n$, namely, its leaves, so the width of $\mathcal{T}_n$ is N. See Figure 1.

BTs admit two interpretations that are consistent with the formal problem we study here. The first interpretation would view a BT (since it is a dag) as the data-dependency graph of a computation to be performed. In this scenario, the nodes of the BT represent the tasks to be executed, while its arcs represent computational dependencies among

[1]All logarithms are to the base 2; e is the base of natural logarithms.

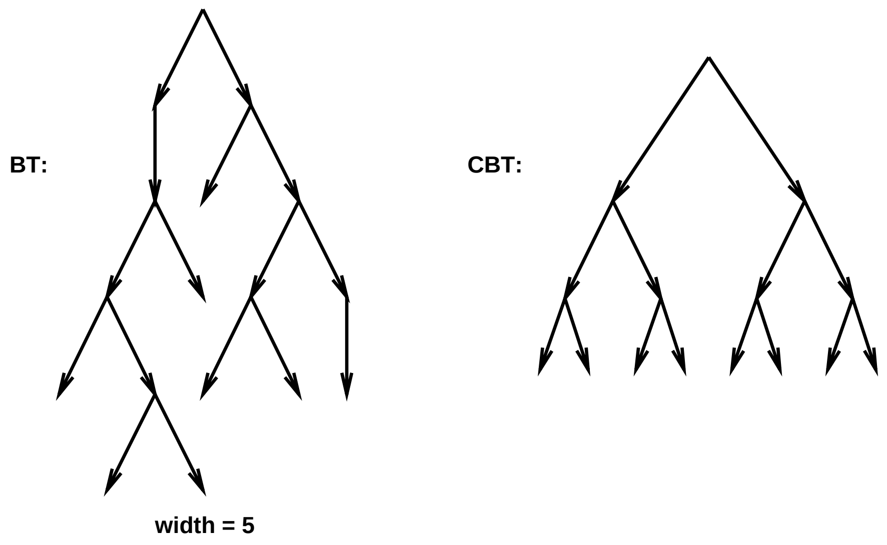

Figure 1: *A width-*5 *BT and a height-*3 *CBT.*

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 [2] - [4], [8] - [15]. The second interpretation would view a BT as a branching process in which each process-node either "dies" after execution (hence is a leaf) or spawns two new processes (which are its children). The second interpretation is a bit less natural in the context of our study than the first, because of our concentrating here on *off-line* scheduling, which implies (and exploits) pre-knowledge of the final shape and size of the fully expanded branching process.

Scheduling BTs. The process of *scheduling* a BT $\mathcal{T}$ proceeds as follows. We are given an endless supply of *enabling tokens* and *execution tokens.* At step 0 of the scheduling process, we place an enabling token *with time-stamp* 0 on the root of $\mathcal{T}$. At each subsequent step, say step $s > 0$, we perform two actions:

- We replace some (one) enabling token by an execution token.
- We place enabling tokens, with time-stamp s, on all children of the just-executed node.

This process continues until all nodes of $\mathcal{T}$ contain execution tokens.

We call a scheduling algorithm *eager* if, at each step, the enabling token we choose to replace has as *small* a time-stamp as possible. We call the algorithm *lazy* if, at each step, the enabling token we choose to replace has as *large* a time-stamp as possible.

One verifies easily that an eager scheduling algorithm executes the nodes of $\mathcal{T}$ in a *breadth-first* manner, while a lazy scheduling algorithm executes the nodes of $\mathcal{T}$ in a *depth-first* manner.

Our real interest is in a class 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.

BT Scheduling and BT Linearization. A *linearization* of the BT $\mathcal{T}$ is a linear ordering of the nodes of $\mathcal{T}$ that is *consistent* with the orientation of the arcs; that is, in the linearization, all arcs of $\mathcal{T}$ point from left to right, so that each nonleaf node precedes its children.

Note that the process of linearizing a BT and the process of scheduling a BT (when it is used to model the dependency structure of computations) are fundamentally isomorphic processes. In both situations, one must keep track of BT-nodes that are eligible to be attended to: these are precisely the ones that have not yet been attended to, but whose parents have. In the scheduling process, "attending to" a node means replacing its enabling token by an execution token and placing enabling tokens on its children; in the linearization process, "attending to" a node means appending it to the current partial linearization and marking its children as eligible for attention.

Having noted the isomorphism between linearizing and scheduling BTs, we shall henceforth talk only about BT-linearization. Our study focuses on the structure of the algorithm that "manages" the linearization process.

Linearizing BTs using Queues: Control vs. Memory. The following formal setup specializes the framework studied in [1], [5] - [7].

A *k-queue linearization of a BT $\mathcal{T}$* is obtained in the following way. One begins by placing the root of $\mathcal{T}$ in one of the k queues. Inductively, a node v of $\mathcal{T}$ is eligible to be laid out in the linearization (resp., executed in the schedule) just when it is at the exit port of one of the k queues. As v is laid out (resp., executed), it is removed from the queue it resided in; simultaneously, each child of v is enqueued on one of the k queues.

The interpretation of multi-queue linearizations in the context of the present study is that the more queues one employs in a linearization algorithm, the more latitude one has to depart from an eager regimen in the direction of a lazy one. This interpretation is particularly apt in the case of multi-queue BT-linearizations, since there exist k-queue BT-linearizations for every positive integer k. Indeed, one readily verifies that there is a unique 1-queue BT-linearization.

Fact 1.1 *The unique* 1*-queue BT-linearization lays out the BT in breadth-first order, i.e., level by level.*

A consequence of the rules for manipulating queues is that a BT-node does not enter a queue until all of its ancestors have already left the queue. This verifies the following simple observation, which is important later.

Fact 1.2 *All nodes that coexist in the k queues at any instant must be* independent *in the BT; i.e., none is an ancestor of another.*

Most obviously, the number of queues used for a particular BT-linearization is a relevant measure of the complexity of the *control* mechanism of the linearization algorithm used. No less relevant, though, is the question of the *memory requirements* of the algorithm, as exposed by the individual and cumulative *capacities* of the queues that implement the algorithm. Specifically, the *capacity of queue $\#q$* in the linearization algorithm is the maximum number of nodes of $\mathcal{T}$ that will ever reside in queue $\#q$ at the same instant. The *cumulative capacities of the queues* is the sum of the capacities of the individual queues. Our goal is to expose a tradeoff between the amount of control in a linearization algorithm and the memory requirements of the algorithm. We accomplish this in the special case when the dag being linearized is a CBT. (Note, however, that our lower bound applies to a broader class of BTs.)

2 A Control-Memory Tradeoff

For all positive integers $N = 2^n$ and $k \leq n$, let $\mathcal{Q}_k(N)$ denote *the minimum per-queue capacity when k queues are used to linearize a CBT having N leaves.* Dually, for all positive integers k and Q, let $\mathcal{N}_k(Q)$ denote *the maximum number of leaves in a CBT that can be linearized using k queues, each of capacity Q.* 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.

An immediate consequence of Fact 1.1 is the following tight specification of $\mathcal{Q}_1(N)$.

Lemma 2.1 *For all positive integers $N = 2^n$, $\mathcal{Q}_1(N) = N/2$.*

We have already mentioned that our goal here is to establish a tradeoff between the control complexity of a CBT-linearization algorithm — as measured by the quantity k — and the memory capacity of the algorithm — as measured by the quantity $\mathcal{Q}_k(N)$. We first present (in Section 2.1) a simple family of CBT-linearization algorithms that at least suggests that such a tradeoff exists. We then state (in Section 2.2) the actual tradeoff, with upper and lower bounds that are close to coincident. Sections 3 and 4 are then devoted, respectively, to proving the upper and lower bounds of the tradeoff.

2.1 A Recursive CBT-Linearization Algorithm

The possibility that there is a control-memory tradeoff for CBT-linearization algorithms is suggested by the following simple family of algorithms. For notational simplicity, when invoking the k-queue version of the following family of algorithms, we assume that the height of the input CBT is divisible by k.

A k-Queue CBT-Linearization Algorithm

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

1. Linearize the top n/k levels of $\mathcal{T}$ using queue $\#k$.
2. For each "leaf" of the tree linearized in step 1, in turn, linearize the CBT rooted at that "leaf" by using queues $\#1$ - $\#(k-1)$ recursively to execute the $(k-1)$-queue version of this algorithm.

See Figure 2

Analyzing the Algorithm. Since each queue is used to linearize (possibly many) CBT(s) of height $(\log N)/k$, no queue needs have capacity greater than $O(N^{1/k})$, uniformly in N and k. (The big-O is needed to compensate for rounding when k does not divide n.) As an immediate consequence, we have:

Fact 2.3 *For all positive integers $N = 2^n$ and $k \leq n$,*

$$\mathcal{Q}_k(N) = O(N^{1/k}), \tag{1}$$

uniformly in N and k.

2.2 The Real Control-Memory Tradeoff

The research described here was motivated by the possible tradeoff suggested in Fact 2.3, i.e., by the possibility that there are lower bounds that match the upper bounds (1). We have verified that there is, indeed, a tradeoff between the quantities k and $\mathcal{Q}_k(N)$, but not exactly the one suggested in the Fact. One aspect of our tradeoff result that we found mildly surprising is that there is a k-queue linearization algorithm that has smaller maximum per-queue capacity than the algorithm presented in Section 2.1. Another surprising aspect is that we obtain upper and lower bounds on $\mathcal{Q}_k(N)$ that differ by only the factor $k^{2-1/k}$. Specifically, we prove the following bounds in the next two sections.

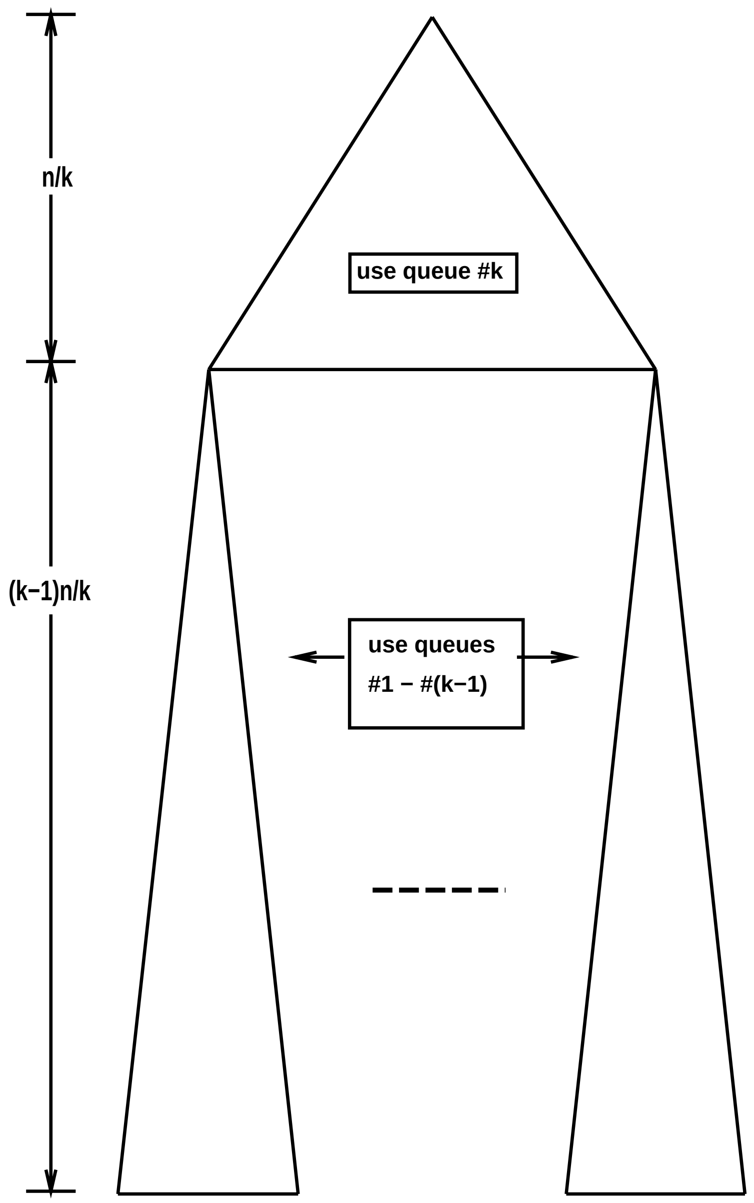

Figure 2: (**a**) *The unique* 1*-queue layout of the height-h CBT.* (**b**) *A capacity saving* 2*-queue layout of the height-h CBT.*

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

$$\frac{e}{2k}\left(\frac{N}{\log^{k-1} N}\right)^{1/k} \le \mathcal{Q}_k(N) \le 8k^{1-1/k}\left(\frac{N}{\log^{k-1} N}\right)^{1/k}. \qquad (2)$$

3 The Upper Bounds in the Tradeoff

This section is devoted to proving the upper bound in (2), via a recursively specified family of linearization algorithms that have better memory requirements than the family of Section 2.1.

Theorem 3.2 *For all positive integers $N = 2^n$ and $k \le n$, the N-leaf CBT $\mathcal{T}_n$ can be linearized with k queues, each of capacity*

$$\mathcal{Q}_k(N) \le 8k^{1-1/k}\left(\frac{N}{\log^{k-1} N}\right)^{1/k}.$$

Since the algorithm that establishes the general case of Theorem 3.2 is somewhat complex, we present first the algorithm for the case of two queues ($k = 2$), which already contains the hardest part of the algorithm. We then proceed to the general k-queue algorithm, which is only moderately more complex than the 2-queue one.

As a purely technical issue, 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 necessary to linearize an N-leaf CBT, we start with a target queue-capacity Q and ask how large a CBT we can linearize using queues of capacity Q. We proceed, therefore, by deriving a lower bound on the quantity $\mathcal{N}_k(Q)$ and inferring therefrom an upper bound on the quantity $\mathcal{Q}_k(N)$.

3.1 The Case $k = 2$

The 2-queue linearization algorithm operates in three phases which we describe now in rough terms. In the first phase, the algorithm uses queue #2 to linearize the top $\lfloor \log(\log Q - 1)\rfloor - 1$ levels of a CBT, leaving the "leaves" from the last level in the queue. In the second phase, the algorithm staggers removing these "leaves" from queue #2 with beginning to use queue #1 to linearize 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 linearize the remainder of the middle

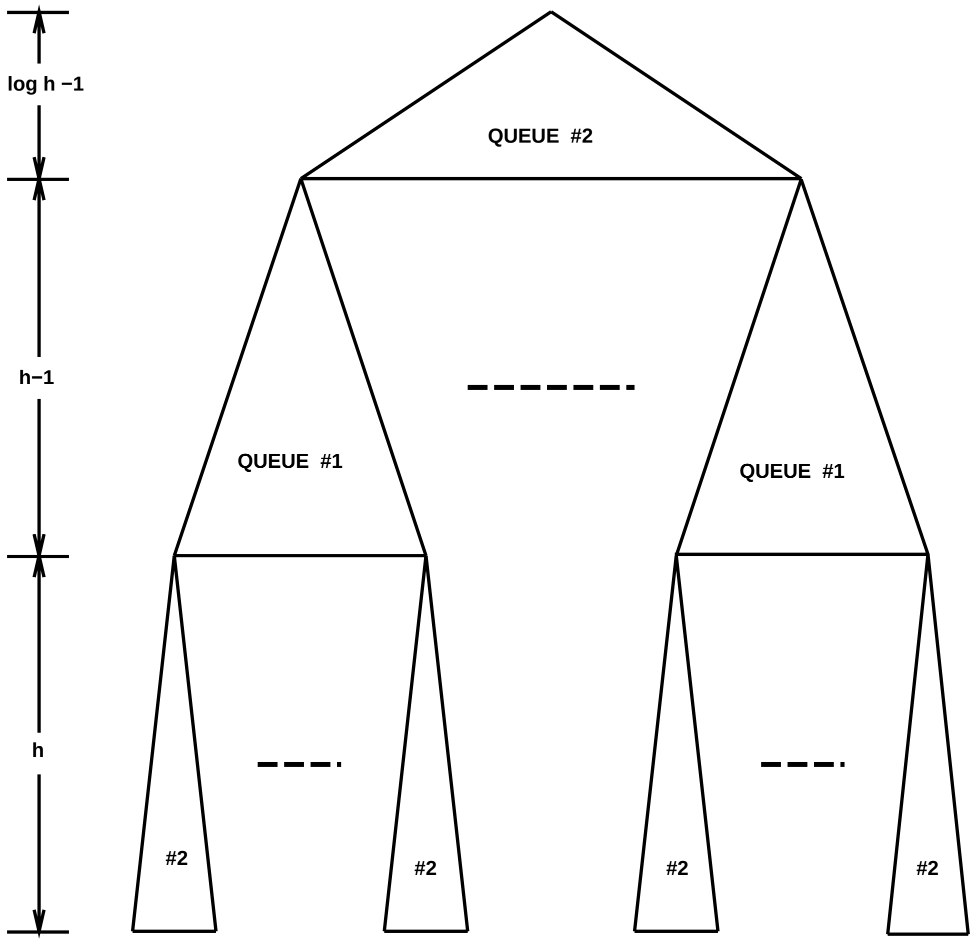

Figure 3: *The target 2-queue layout of the height-h CBT.*

$\log Q - 1$ levels of the CBT with using queue #2 to linearize the bottom $\log Q - 1$ levels. This latter staggering proceeds by having queue #2 linearize a $(\log Q - 1)$-level CBT rooted at each middle-tree "leaf" from queue #1. We then assess the size of the linearized CBT as a function of the queue-capacity Q. To assist the reader in understanding the ensuing technical details, we depict in Figure 3 the ultimate usage pattern of the two queues.

A. The Efficient 2-Queue Linearization Algorithm

Phase 1: The Top of the Tree.

In this phase, we use queue #2 to lay out the top $\lfloor \log(\log Q - 1) \rfloor - 1$ levels of the CBT we are linearizing, using the breadth-first regimen that is the unique way a single queue

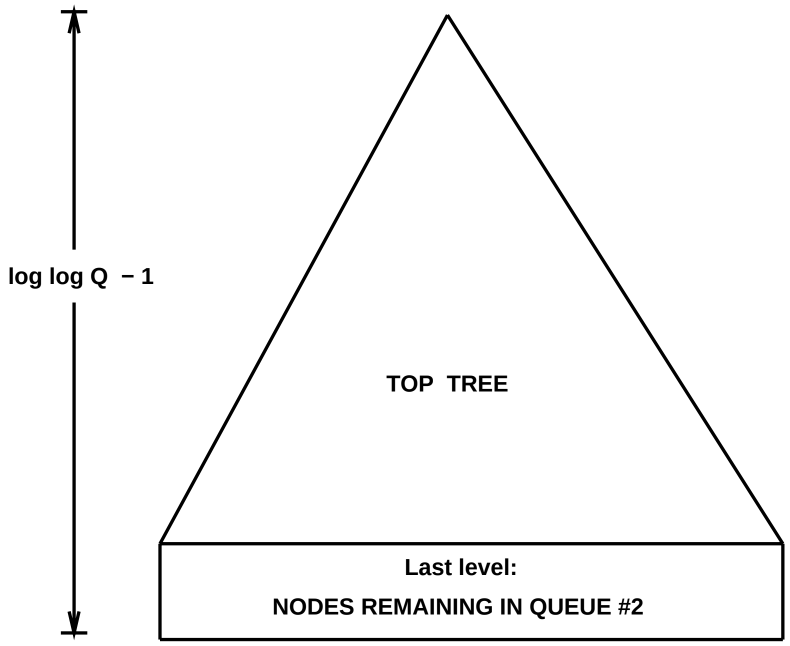

Figure 4: *Laying out the top of the CBT.*

can linearize a CBT; cf. Fact 1.1. At the end of this phase, queue #2 will contain

$$2^{\lfloor \log(\log Q - 1) \rfloor}$$

nodes. We make the transition into Phase 2 of the algorithm by considering each of these nodes 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 lay out the *middle trees* that comprise the next $\log Q - 1$ levels of the CBT we are linearizing. This is the most complicated of the three phases, in that these middle trees get laid out in a staggered manner, in two senses. First, the $2^{\lfloor \log(\log Q - 1) \rfloor}$ middle trees get interleaved in the linearization we are producing. Second, the layout of the middle trees is interleaved with segments of Phase 3, wherein the bottom trees are laid out.

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

Lay out 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.

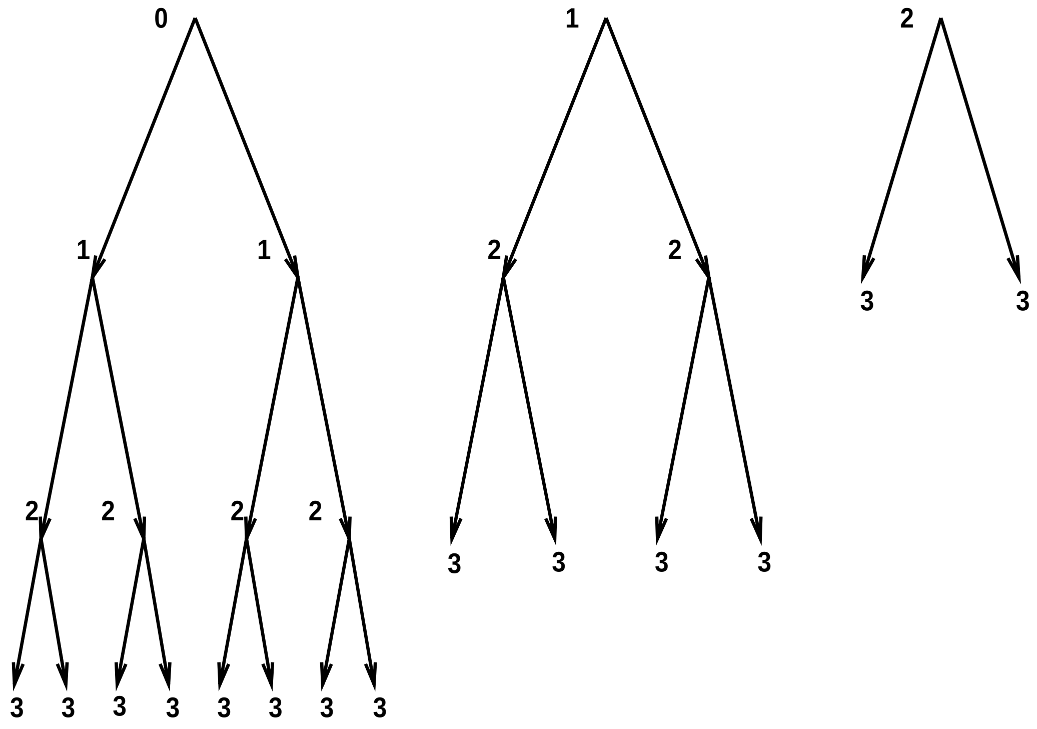

Figure 5: *The initial steps of Phase 2.*

Step 1. Begin the first middle tree.

> **Step 1.1.** Use queue #1 to lay out level 1 of the first middle tree.
>
> **Step 1.2.** Use queue #2 to lay out level 0 of the second middle tree, placing 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 lay out level 2 of the first middle tree.
>
> **Step 2.2.** Use queue #1 to lay out level 1 of the second middle tree.
>
> **Step 2.3.** Use queue #2 to lay out level 0 of the third middle tree, placing 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 lay out level 3 of the first middle tree.
>
> **Step 3.2.** Use queue #1 to lay out level 2 of the second middle tree.
>
> **Step 3.3.** Use queue #1 to lay out level 1 of the third middle tree.
>
> **Step 3.4.** Use queue #2 to lay out level 0 of the fourth middle tree, placing the children of this root in queue #1.

• • •

Step $(\log Q - 2)$**.** Finish the first middle tree; continue the second through next-to-last middle trees; begin the last middle tree.

> **Step** $(\log Q - 2)$**.1.** Use queue #1 to lay out level $\log Q - 2$ of the first middle tree.
>
> **Step** $(\log Q - 2)$**.2.** Use queue #1 to lay out level $\log Q - 3$ of the second middle tree.
>
> • • •
>
> **Step** $(\log Q - 2)$**.**$(\log Q - 2)$**.** Use queue #1 to lay out level 1 of the second from last middle tree.
>
> **Step** $(\log Q - 2)$**.**$(\log Q - 1)$**.** Use queue #2 to lay out level 0 of the last middle tree, placing the children of this root in queue #1.

At this point, queue #2 has been completely emptied, hence is available for reuse. Queue #1, on the other hand, contains fewer than Q nodes. Specifically, queue #1 contains $Q/2$ nodes from the first middle tree, and, in general, it contains only half as many node from the $(i+1)$th middle tree as it does from the ith; in the worst case, of course, there are

$$2^{\lfloor \log(\log Q - 1) \rfloor} = \log Q - 1$$

middle trees, hence $Q - 2$ nodes in queue #1. See Fig. 6.

We have now completely laid out the first middle tree and partially laid out 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 lay out the remaining middle trees. First, though, we initiate Phase 3 in which queue #2 is used to lay out the bottom levels of the CBT being linearized. 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 the first of these subphases — call it Phase 3a — we begin viewing the "leaves" of the middle trees as the roots of *bottom trees* — each being a CBT with $\mathcal{N}_1(Q) = 2Q$ leaves. In the second of these subphases — call it Phase 3b — we continue using the regimen of Phase 2 to lay out the middle trees.

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 lay out a single node — call it node v — from queue #1, and we use queue #2 to lay out a CBT on $2Q$ 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 $\log Q - 2$ of a middle tree. During the subphase, we perform one more step of Phase 2, to extend the layout 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 laid out) has the form:

Step $(\log Q - 1)$**.** Finish the second middle tree; continue the third through last middle trees.

- **Step** $(\log Q - 1)$**.1.** Use queue #1 to lay out level $\log Q - 2$ of the second middle tree.
- **Step** $(\log Q - 1)$**.2.** Use queue #1 to lay out level $\log Q - 3$ of the third middle tree.
- • • •
- **Step** $(\log Q - 1)$**.**$(\log Q - 3)$**.** Use queue #1 to lay out level 2 of the second from last middle tree.

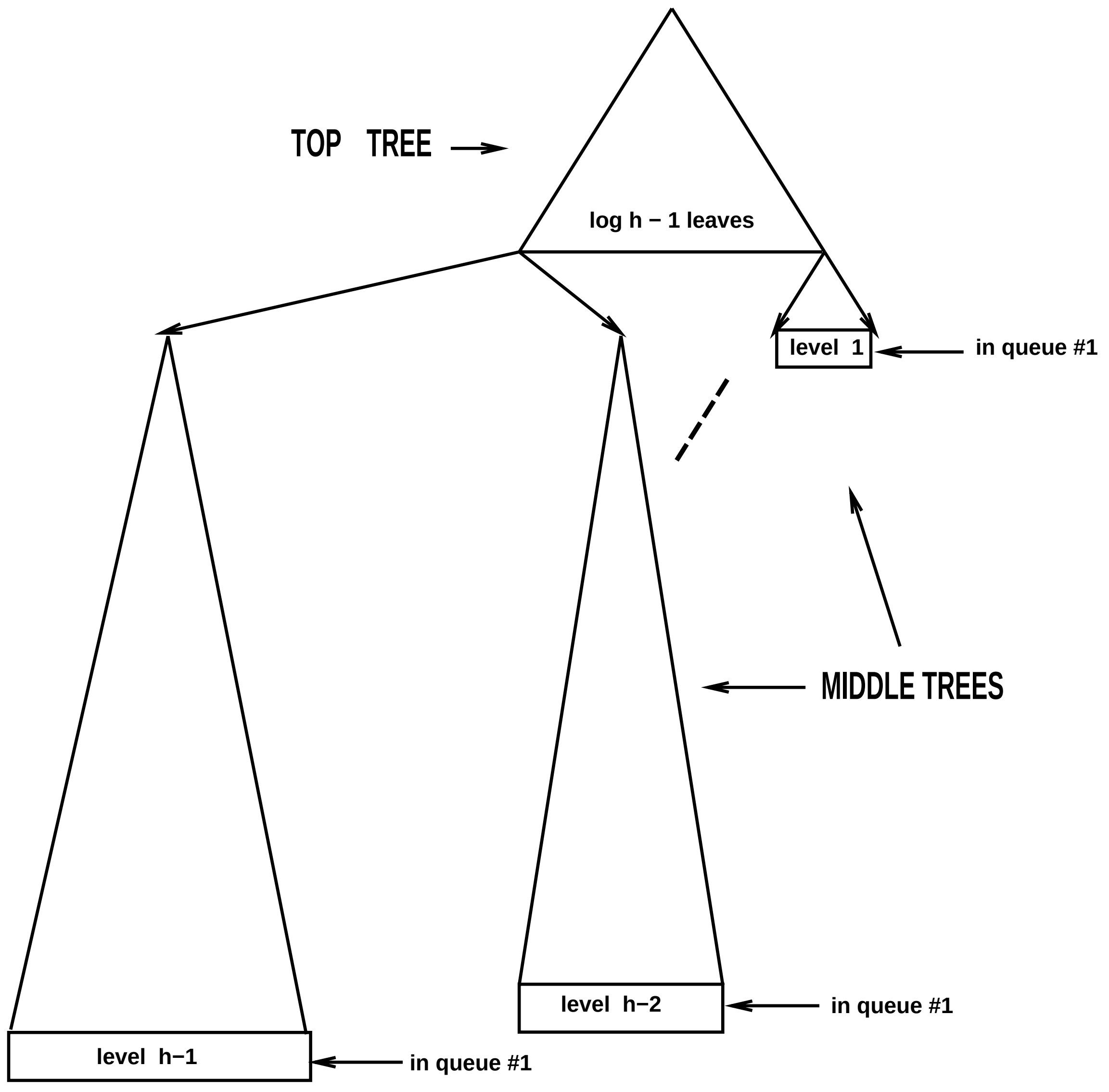

Figure 6: *The layout after Phase 2.*

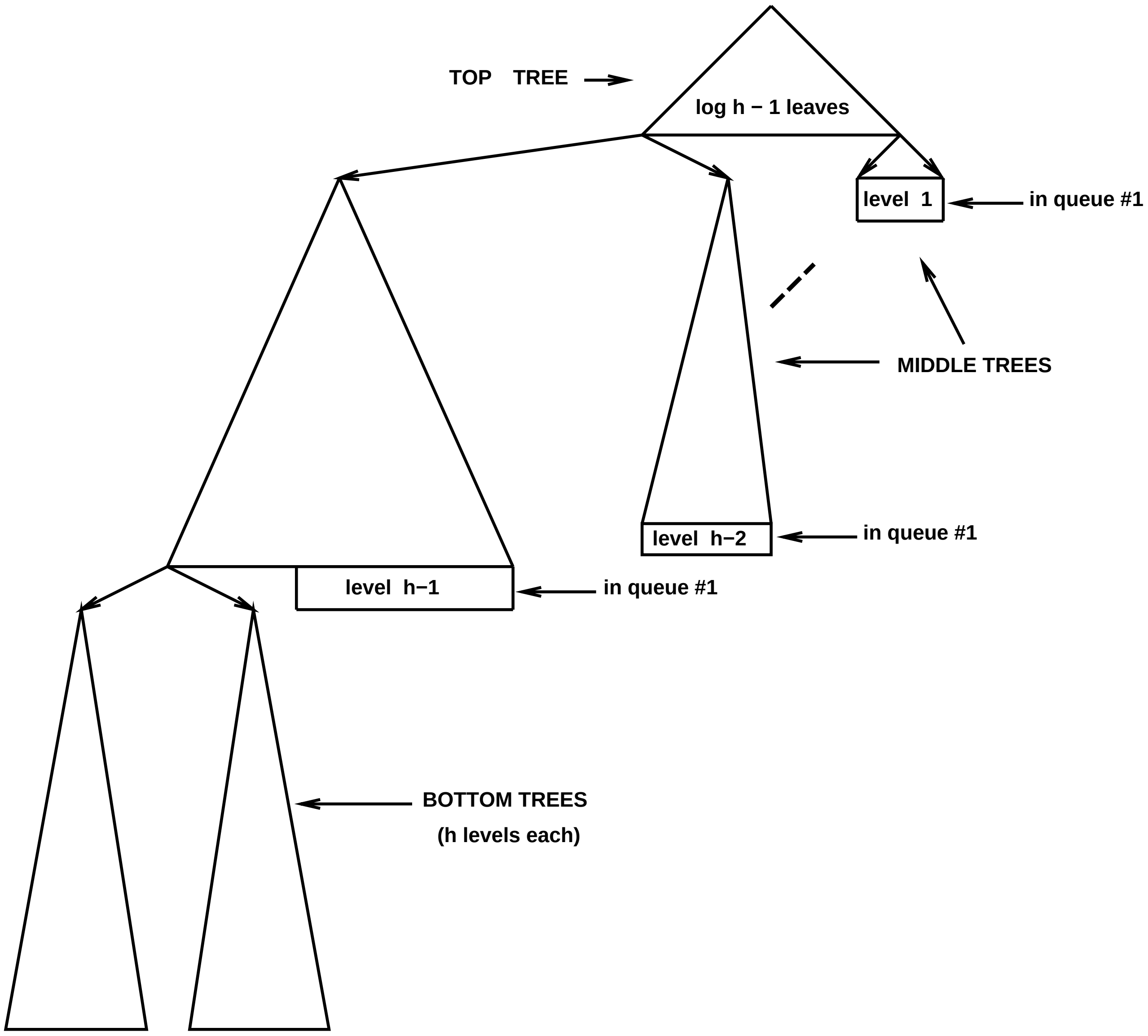

Figure 7: *The layout after Phase 3a begins: the first two bottom trees have been laid out.*

Step $(\log Q - 1)$.$(\log Q - 2)$. Use queue #1 to lay out level 1 of the last middle tree.

See Fig. 8.

B. The Analysis

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

$$2^{\lfloor \log(\log Q - 1) \rfloor} > \frac{1}{4}(\log Q - 1)$$

top-tree leaves, hence, at least $\frac{1}{8}Q(\log Q - 1)$ 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

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

3.2 The Case of General k

We now show how to generalize the 2-queue algorithm to a family of algorithms for arbitrary numbers of queues.

A. The Algorithm

Our general linearization algorithm uses Phases 1, 2, and 3b of the 2-queue algorithm directly. It modifies only Phase 3a, as follows.

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 lay out a single node — call it node v — from queue #1, and we use queues #2 - #k to lay out a CBT on $2Q$ leaves, rooted at node v, *using a recursive invocation of the* $(k-1)$*-queue version of this algorithm.*

Note that the 2-queue algorithm of the previous subsection can, in fact, be obtained via this recursive strategy, from the base case $k = 1$.

As we did earlier, we remark that queues #2 - #k are all available for this recursive call because: (a) queue #2 lays out the last "leaf" of the top tree just before queue #1

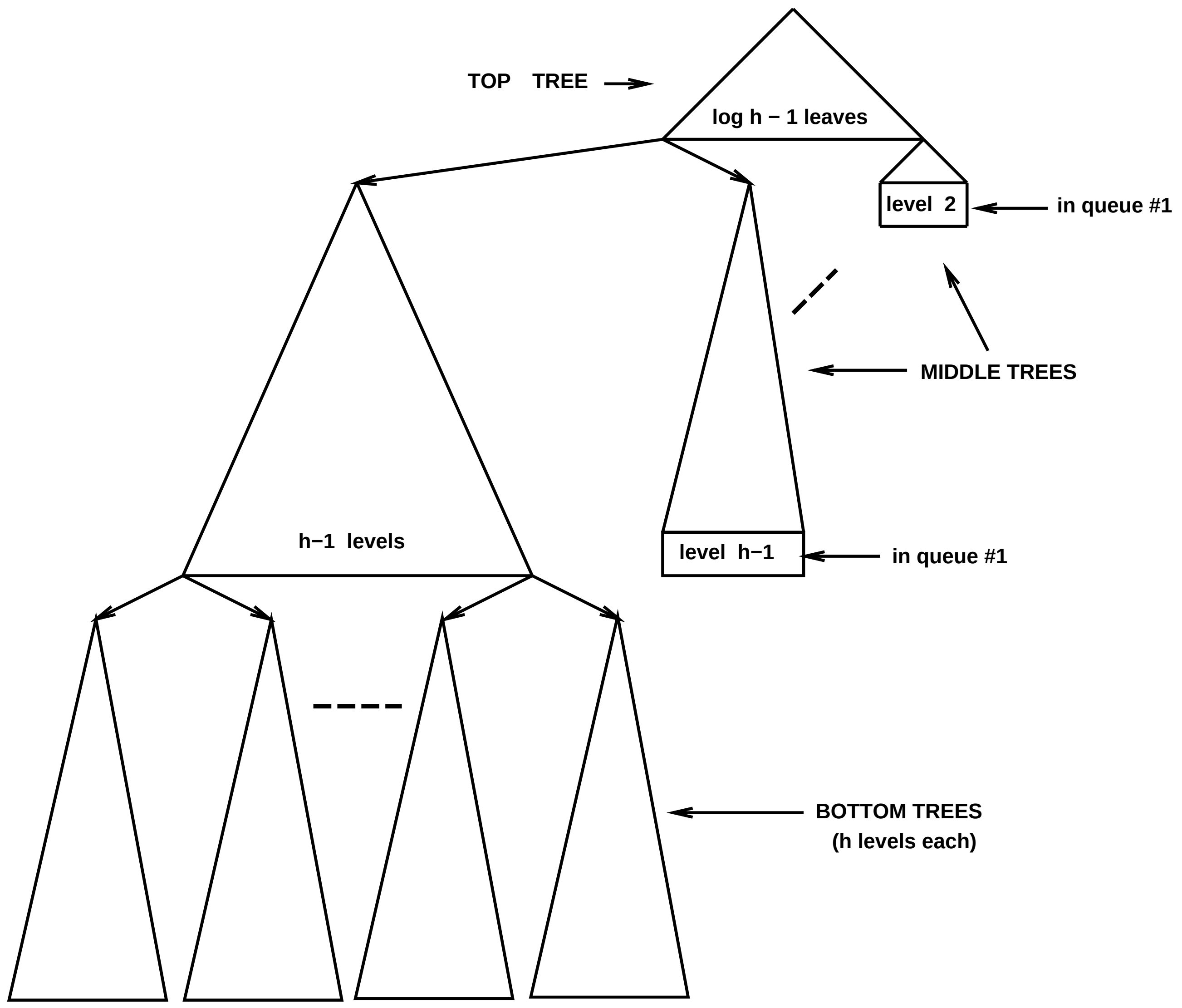

Figure 8: *The layout after Phase 3b begins: the first set of bottom trees have been laid out; the second middle tree has been completed.*

lays out the first "leaf" of the leftmost middle tree, so it is available; (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 generated has, as a function of the given queue-capacity Q. This number is easily seen to be be given by the recursion

$$\begin{aligned} \mathcal{N}_1(Q) &= 2Q \\ \mathcal{N}_{k+1}(Q) &\geq \frac{1}{8}Q(\log Q - 1)\mathcal{N}_k(Q) \end{aligned}$$

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

$$\mathcal{N}_k(Q) = 2Q\left(\frac{1}{8}Q\log Q\right)^{k-1}.$$

For our ends, we invert this relation, to get the sought upper bound on $\mathcal{Q}_k(N)$, namely,

$$\mathcal{Q}_k(N) \leq 8k^{1-1/k}\left(\frac{N}{\log^{k-1} N}\right)^{1/k}.$$

This completes the proof. □ To wit:

$$N \geq 2^{4-3k}Q^k \log^{k-1} Q$$

so that

$$\log N \geq k \log Q$$

so that

$$Q^k \leq 2^{3-4k}k^{k-1}\frac{N}{\log^{k-1} N}.$$

4 The Lower Bounds in the Tradeoff

This section is devoted to proving the lower bound in Theorem 2.1. In fact, we prove this bound as a corollary of the following more general lower bound.

Theorem 4.3 *For all positive integers $N = 2^n$ and $k \leq n$, given any k-queue linearization of a BT having width N and $c\log N$ levels, at least one queue must have capacity at least*

$$\frac{e}{2ck}\left(\frac{N}{\log^{k-1} N}\right)^{1/k}.$$

Corollary 4.1 *For all positive integers $N = 2^n$ and $k \leq n$, given any k-queue linearization of the N-leaf CBT, at least one of the queues must have capacity*

$$\mathcal{Q}_k(N) \geq \frac{e}{2k}\left(\frac{N}{\log^{k-1} N}\right)^{1/k}.$$

Proof of Theorem 4.3. Say that we are given an arbitrary k-queue linearization of a BT $\mathcal{T}$ having width N and $c \log N$ levels (for some constant $c > 0$). Call a level of $\mathcal{T}$ that has N nodes a *wide* level.

We begin by parsing the given linearization into contiguous substrings, by partitioning the linearization process into *phases.* For the purpose of defining the phases, recall that the action of laying out a node (i.e., appending it to the current partial linearization) and loading its children 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 laid out) is laid out and its children loaded on queues. Hence, the first substring in the parsing is just a one-letter string consisting of the root.

Inductively, define Phase $i+1$ to be that part of the process wherein all nodes that were loaded into queues during Phase i are laid out; the Phase ends when the last of these Phase-i "legacies" has been laid out. The substring produced during Phase $i+1$ is, then, the longest substring following the substring produced during Phase i, during which some queue still contains a node that was put there during Phase i.

By a straightforward induction, one establishes that level i of the BT $\mathcal{T}$ must be laid out by the end of Phase i. It follows that

Fact 4.4 *There are at most $c \log N$ phases in the linearization process.*

Fact 4.4 has the following immediate consequence.

Fact 4.5 *There must exist a phase whose associated substring contains at least $N/(c \log N)$ nodes from a wide level of $\mathcal{T}$. Call such a phase* long.

Now look at what happens during a phase of an *optimal* linearization of $\mathcal{T}$, i.e., one that minimizes the capacity of the largest-capacity queue. The phase starts with some nodes residing within the k queues — at most $\mathcal{Q}_k(N)$ per queue. As we noted in Fact 1.2, all of these nodes must be independent in the BT. Moreover, by definition of "phase," all of these nodes must be laid out by the end of the phase. We can, therefore, characterize what the portion of $\mathcal{T}$ that is laid out during a phase looks like.

Fact 4.6 *What is laid out during a phase is a forest of BTs rooted at the $\leq kQ_k(N)$ nodes that resided in the queues at the start of the phase.*

Note now that some BT — call it $\mathcal{T}'$ — in the forest laid out during a *long* phase must contain as nodes at least $N/(ckQ_k(N)\log N)$ of the nodes from the wide level of $\mathcal{T}$. Hence, the width of $\mathcal{T}'$ can be no smaller than this quantity.

Fact 4.7 *At least one of the BTs in the forest must have width no smaller than*

$$\frac{N}{ckQ_k(N)\log N}.$$

Next note that, by definition of "phase," there must be some queue — call it queue $\#m$ — whose sole contributions to the linearization during the long phase are the nodes that started in it at the beginning of the phase. This is because the phase ends when the last node that was created during the previous phase is laid out — so queue $\#m$ is identified as the source of this last node. Now, queue $\#m$ started the phase (as did every queue) with no more than $Q_k(N)$ nodes. As we noted earlier, each of these nodes is the root of a BT generated during the long phase. Of all the nodes laid out during the long phase, only these roots come from queue $\#m$. Now, if we remove from the forest all these queue-$\#m$ nodes, then we partition each BT $\mathcal{T}''$ that is rooted at a queue-$\#m$ node into two BTs, at least one of which must have at least half the width of $\mathcal{T}''$. It follows, therefore, that

Fact 4.8 *The forest generated during the long phase must, after all queue-$\#m$ nodes are removed, contain a BT of width no smaller than*

$$\frac{N}{2ckQ_k(N)\log N}$$

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

We infer immediately the recurrent lower bound

$$Q_k(N) \geq Q_{k-1}\left(\frac{N}{2ckQ_k(N)\log N}\right) \tag{3}$$

whose initial case ($k=1$) is resolved in Lemma 2.1 (for the case of a CBT).

We solve recurrence (3) by induction, making two assumptions. First, we assume for induction that

$$Q_{k-1}(N) \geq \alpha_{k-1}\left(\frac{N}{\log^{k-2} N}\right)^{1/(k-1)} \tag{4}$$

for some quantity α_{k-1} which will be determined later (although we already know from Lemma 2.1 that $\alpha_1 = 1/2$ when $\mathcal{T}$ is an N-leaf CBT). Second, we assume that

$$\log \mathcal{Q}_k(N) = \frac{1}{k} \log N + \text{l.o.t.}.$$

This assumption will turn out to be easy to verify.

We begin by substituting inequality (4) into recurrence (3), to obtain

$$\mathcal{Q}_k(N) \geq \alpha_{k-1} \left(\left(\frac{N}{2ck\mathcal{Q}_k(N) \log N} \right) \left(\frac{k}{(k-1) \log N} \right)^{k-2} \right)^{1/(k-1)}. \tag{5}$$

After some manipulation, inequality (5) becomes

$$\mathcal{Q}_k(N) \geq \left(\frac{k^{k-3}}{2c(k-1)^{k-2}} \right)^{1/k} \alpha_{k-1}^{(k-1)/k} \left(\frac{N}{\log^{k-1} N} \right)^{1/k}. \tag{6}$$

In order to obtain the desired lower bound, we now turn our attention to the following recurrence, which is suggested by inequality (4):

$$\alpha_k = \left(\frac{k^{k-3}}{2c(k-1)^{k-2}} \right)^{1/k} \alpha_{k-1}^{(k-1)/k}, \tag{7}$$

with initial condition $\alpha_1 = 1/2$. Elementary manipulation converts recurrence (7) to the recurrent bound

$$\alpha_k = \left(\frac{1}{2ck} \right)^{1/k} \left(1 + \frac{1}{k-1} \right)^{1/k} \alpha_{k-1}^{(k-1)/k} \geq \left(\frac{1}{2ck} \right)^{1/k} \alpha_{k-1}^{(k-1)/k},$$

so that

$$\alpha_k \geq \frac{1}{2c} \left(\frac{1}{k!} \right)^{1/k} > \frac{e}{2ck}. \tag{8}$$

Combining inequality (6) with inequality (8) via recurrence (7) yields the lower bound of Theorem 4.3. □

Proof of Corollary 4.1. The lower bound of Corollary 4.1 is immediate from that of Theorem 4.3 if one notes that the N-leaf CBT has $\log N + 1$ levels (so $c = 1$). □

Remark. A more careful analysis replaces the recurrent bound (3) by

$$\mathcal{Q}_k(N) \geq \mathcal{Q}_{k-1} \left(\frac{N}{c(k+1)\mathcal{Q}_k(N) \log N} - \mathcal{Q}_k(N) \right),$$

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 $\leq (k+1)\mathcal{Q}_k(N)$ 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 $\leq \mathcal{Q}_k(N)$. What this means is:

Fact 4.9 *During a phase whose substring contains L nodes from a wide level of the BT $\mathcal{T}$, there must be a BT of width at least*

$$\frac{L}{c(k+1)\mathcal{Q}_k(N)} - \mathcal{Q}_k(N)$$

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

ACKNOWLEDGMENTS. The authors thank Marc Snir for helpful comments in the early stages of this research and Li-Xin Gao for helpful discussions in the later stages.

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 Contract OSR-86-0076, DARPA Contract N00014-80-C-0622, Army Contract DAAL-03-86-K-0171, and NSF Presidential Young Investigator Award with matching funds from ATT and IBM; the research of A. L. Rosenberg was supported in part by NSF Grant CCR-90-13184. 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.T. Barrett, L.S. Heath, and S.V. Pemmaraju (1993): Stack and queue layouts of directed acyclic graphs. *1992 DIMACS Workshop on Planar Graphs: Structure and Algorithms*, to appear.

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

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

[4] 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.

[5] 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.

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

[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] 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.

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

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

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