A Steady State Analysis of Diffracting Trees

Nir Shavit [*] Eli Upfal [†] Asaph Zemach [‡]

July 15, 1997

Abstract

Diffracting trees are an effective and highly scalable distributed-parallel technique for shared counting and load balancing. This paper presents the first steady-state combinatorial model and analysis for diffracting trees, and uses it to answer several critical algorithmic design questions. Our model is simple and sufficiently high level to overcome many implementation specific details, and yet as we will show it is rich enough to accurately predict empirically observed behaviors. As a result of our analysis we were able to identify starvation problems in the original diffracting tree algorithm and modify it to a create a more stable version. We are also able to identify the range in which the diffracting tree performs most efficiently, and the ranges in which its performance degrades. We believe our model and modeling approach open the way to steady-state analysis of other distributed-parallel structures such as counting networks and elimination trees.

1 Introduction

Diffracting trees [19] are among the most effective and scalable distributed-parallel techniques for shared counting, with a variety of applications to load balancing and concurrent data structure design. Diffracting trees are a special form of the counting networks of Aspnes, Herlihy, and Shavit [4]. They are

[*] MIT and Tel-Aviv University. Supported by National Science Foundation grant CCR-9520298. Contact Author: `shanir@theory.lcs.mit.edu`.

[†] The Weizmann Institute, Israel, and IBM Almaden Research Center, California. Work at the Weizmann Institute supported in part by the Norman D. Cohen Professorial Chair of Computer Science, a MINERVA grant, and a grant from the Israeli Academy of Science.

[‡] Tel-Aviv University. Supported by an Eshkol Fellowship from the Israeli Ministry of Science.

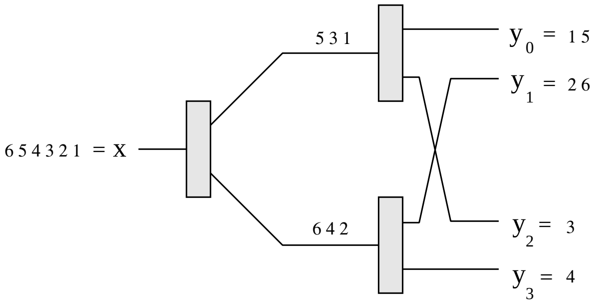

Figure 1: A Simple Counting Tree

constructed from simple one-input two-output computing elements called *balancers* that are connected to one another by wires to form a balanced binary tree. Tokens (processes) arrive on the balancer's input wire at arbitrary times, and are output on its output wires. Intuitively one may think of a balancer as a toggle mechanism (a bit that is repeatedly complemented), that given a stream of input tokens, repeatedly sends one token to the left output wire and one to the right, effectively balancing the number of tokens that have been output. To illustrate this property, consider an execution in which tokens traverse the tree sequentially, one completely after the other. Figure 1 shows such an execution on a tree of width 4. As can be seen, the tree moves input tokens to output wires in increasing order modulo 4. Trees of balancers having this property can easily be adapted to count the total number of tokens that have entered the network. Counting is done by adding a "local counter" to each output wire i, so that tokens coming out of that wire are consecutively assigned numbers $i, i + 4, i + (4 \cdot 2) \ldots$

However, under high loads, the toggle bits, especially the one at the root balancer of the tree, will be hot-spots suffering from contention and sequential bottlenecks that are as bad as that of a centralized counter implementation. Diffracting trees overcome the problem by having a "prism" mechanism in front of the toggle bit of every balancer, allowing independent pairs of tokens to be "diffracted" in separate memory locations in a coordinated manner one to the left and one to the right. A coordinated pair of processors can leave the balancer without either of them having to toggle the shared bit, since each pair of toggles leaves the bit in the same state. The processors need simply to agree between themselves which one would have gotten the "0" bit, and which the "1". The diffraction mechanism uses randomization to ensure high collision/diffraction rates on the prism, and the tree structure guarantees correctness of the output values. Given appropriate hardware primitives, diffracting trees can be implemented in a lock-free manner. In fact, assuming a hardware Fetch&Complement operation allows making diffracting trees wait-free [12], that is, for each increment operation termination is guaranteed in a bounded number of steps even if all other processors fail.

When implementing diffracting trees [13], the following type of questions are of critical importance. Given a typical system with P processors that cycle repeatedly between performing an increment on a diffracting tree based counter and performing some amount of *work*, what are the optimal choices of: (a) the tree "size" (captured by its depth d) relative to the number of processors P. If the tree is too small, it will be overloaded, bringing contention and less parallelism than possible. If it is too deep, the counters at its leaves will not be fully utilized, achieving less than optimal throughput. (b) The prism widths, quantified by `L`, the total number of `prism` locations in the balancers at a given level of the tree. This parameter affects the chances of a successful pairing-off. If it is too large, then processors will tend to miss each other, failing to pair-off and causing contention on the toggle bit. If it is too small, contention and sequential bottlenecking will occur as too many processors will be trying to access the same `prism` locations at the same time.

Finally, even with an optimal choice of tree parameters for a certain maximum number of processors P, there is a wide range of intermediate concurrency levels in which it is unclear what the rate of diffraction will be and hence performance is hard to predict.

In this paper we present the first steady-state combinatorial analysis for diffracting trees, and use it to explain their behavior patterns and answer critical design questions such as the ones posed above. Our model is simple and sufficiently high level to overcome many implementation specific details, and yet as we will show it is rich enough to accurately predict empirically observed behaviors. As a result of our analysis we were able to identify starvation problems in the algorithm of [19] and thus introduce a more stable diffracting balancer algorithm (see section 2.3).

We were also able to identify the range (as a function of P, $work$, d and L) in which the diffracting tree performs most efficiently, and the ranges in which its performance degrades. We show that when $\frac{P}{dL} = O(1)$ and $L \leq 2^d$, the throughput of the system is optimal, and contention is low. With less processors, diffraction probability decreases causing a rise in latency which reaches its peak when $P = d\sqrt{L}$. We further derive performance bounds for very large and very small values of P.

In the final section of this paper we provide a collection of experimental benchmarks that show how accurately our model fits with actual diffracting tree performance.

The closest modeling work related to ours is the amortized contention model of Dwork, Herlihy, and Waarts [9] used in the analysis of counting networks

[9] and of the randomized counting networks by Aiello, Venkatesan, Yung [3]. However, unlike our work, that analysis is directed at modeling and quantifying contention in the face of a *worst case* adversary, not the steady state behaviors of the algorithms in normal (i.e. *common case*) executions.

This paper is organized as follows: Section 2 gives a brief review of tree based counting-networks and contains both the original diffracting tree algorithm, as well as the new code developed based on the analysis. Section 3 introduces the combinatorial model and analyzes the performance of diffracting trees in the steady-state. In Section 4 we present empirical evidence collected through benchmarks on a simulated shared memory machine to support the analysis and Section 5 concludes this paper and lists areas of further research.

2 Counting Trees and Diffraction

Diffracting trees [19] are counting trees, a special form of the counting network data structures introduced by Aspnes, Herlihy and Shavit [5]. They are binary trees of nodes called balancers. A *balancer* is a computing element with one input wire and two output wires. Tokens arrive on the balancer's input wire at arbitrary times, and are output on its output wires. Intuitively one may think of a balancer as a toggle mechanism, that given a stream of input tokens, repeatedly sends one token to the left output wire and one to the right, effectively balancing the output on each wire. We denote by x the number of input tokens ever received on the balancer's input wire, and by y_i, $i \in \{0, 1\}$ the number of tokens ever output on its ith output wire. Given any finite number of input tokens x, it is guaranteed that within a finite amount of time, the balancer will reach a *quiescent* state, that is, one in which the sets of input and output tokens are the same. In any quiescent state, $y_0 = \lceil x/2 \rceil$ and $y_1 = \lfloor x/2 \rfloor$. We will abuse this notation and use y_i both as the name of the ith output wire and as the count of the number of tokens output on that wire.

The diffracting tree is defined as follows. Let k be a power of two, and let us define the counting tree $\textsc{Binary}[2k]$ inductively. When k is equal to 1, the $\textsc{Binary}[2k]$ network consists of a single balancer with output wires y_0 and y_1. For $k > 1$, we construct the $\textsc{Binary}[2k]$ tree from two $\textsc{Binary}[k]$ trees and one additional balancer. We make the input wire x of the single balancer the root of the tree and connect each of its output wires to the input wire of a tree of width k. We then re-designate output wires $y_0, y_1, \ldots, y_{k-1}$ of the tree extending from the 0 output wire as the even output wires $y_0, y_2, \ldots, y_{2k-2}$ of $\textsc{Binary}[2k]$ and the wires $y_0, y_1, \ldots, y_{k-1}$ of the tree extending from the balancer's 1 output wire as the odd output wires $y_1, y_3, \ldots, y_{2k-1}$.

One can extend the notion of quiescence to trees in the natural way, and define a *counting tree* of width w as a of tree balancers, $\textsc{Binary}[w]$, with outputs

```
type balancer is
begin
  lock:   boolean
  toggle: boolean
  next:   array [0..1] of ptr to balancer
end

globals
  Width: integer
  Root : ptr to root of Binary[width] tree

function typical_balancer(b: ptr to balancer) : ptr to balancer
begin
  lock(b->lock)
  i := b->toggle
  b->toggle := not(i)
  unlock(b->lock)
  return b->next[i]
end

function fetch&incr(): integer
begin
  b:= Root
  while not leaf(b)
    b := typical_balancer(b)
  endwhile
  i := increment_counter_at_leaf(b)
  return i * Width + number_of_leaf(b)
end
```

Figure 2: A Shared-Memory tree-based counter implementation

$y_0, .., y_{w-1}$ that satisfy the following *step property*:

> In any quiescent state, $0 \leq y_i - y_j \leq 1$ for any $i < j$.

To illustrate this property, consider an execution in which tokens traverse the tree sequentially, one completely after the other. Figure 1 shows such an execution on a Binary[4] counting tree, the tree moves input tokens to output wires in increasing order modulo w. Trees having this property are called counting trees because they can easily be adapted to count the total number of tokens that have entered the network. Counting is done by adding a "local counter" to each output wire i, so that tokens coming out of that wire are consecutively assigned numbers $i, i + w, \ldots, i + (y_i - 1)w$.

On a shared memory multiprocessor, one can implement a balancing tree as a shared data structure, where balancers are records, and wires are pointers from one record to another. Each of the machine's asynchronous processors can run a program that repeatedly traverses the data structure from the root input pointer to some output pointer, each time shepherding a new token through the network. Pseudo-code for this implementation appears in Figure 2. We do not assume an atomic fetch-and-complement operation, instead, we use a lock to avoid race conditions on the balancer's toggle bit.

Diffracting trees are counting trees whose balancers are of a novel type called *diffracting balancers.* One could easily implement a balancer using a single `toggle` bit. Each processor would toggle the bit inside the balancer, and accordingly decide on which wire to exit. However, if many tokens attempted to pass through the same balancer concurrently, the toggle bit would quickly become a hot-spot. Even if one applied contention reduction techniques such as exponential back-off, the toggle bit would still form a sequential bottleneck. One can overcome this sequential bottleneck based on the following observation:

> *If an even number of tokens passes through a balancer, they are evenly balanced left and right, yet the value of the toggle bit is unchanged.*

Thus, one can allow pairs of colliding tokens to "pair-off" and coordinate among themselves which is diffracted "right" and which diffracted "left". Then they could both leave the balancer without either of them ever having to touch the toggle bit. By performing the collision/coordination decisions in separate locations instead of a global toggle bit, one can increase parallelism and lower contention. However, to guarantee good performance one must make sure that many collisions occur, not an obvious task given the asynchrony in the system.

To achieve this goal, the implementation of the diffracting balancer is based on adding a special `prism` array "in front" of the toggle bit in every balancer. When a token (processor) P enters the balancer, it first selects a location j in `prism` uniformly at random. P tries to "collide" with the previous processor that selected j, and if successful they leave the balancer one to the left and the other to the right. Otherwise, P waits ("spins") for a fixed time `spin` to see whether some other processor R will enter and collide with it by selecting the same location j in `prism`. If no collision occurs within time `spin`, P toggles the shared bit and leaves the balancer accordingly.

2.1 The Original Diffracting Tree Implementation

Figure 3 gives the data structure for diffracting balancers due to Shavit and Zemach [20]. Each balancer record consists of a `toggle` bit (with accompanying

```
type balancer is
  size:   integer
  spin:   integer
  prism:  array [1..size] of integer
  lock:   boolean
  toggle: boolean
  next:   array [0..1] of ptr to balancer
endtype

location: global array[1..NUMPROCS] of ptr to balancer
```

Figure 3: Diffracting balancer data structure

`lock`) and a `prism` array. The `spin` variable holds the amount of a time a processor should delay at this node while waiting to be diffracted, and `next[0..1]` are the two balancers (or counters) which are descendants of this node of the tree. An additional global `location`$[1..n]$ array has an element per processor $p \in \{1 \ldots n\}$ (per processor, not per token), holding the address of the balancer which p is currently traversing. The data structures of a diffracting tree of width 4 are depicted graphically in Figure 4.

Figure 5 gives the implementation of a diffracting tree balancer that was used in [19].

- `MYID` – The ID of the processor executing the code.
- `random(n)` – Returns an integer number in the range $[0, n-1]$.
- `SWAP(a,x)` – Atomically writes `x` to address `a`, and returns the previous value there.
- `C&S(a,p,n)` – Atomically compares the value at address `a` to `p`, if they match, writes `n` to `a` and returns `TRUE`, otherwise returns `FALSE`.
- `T&T&S(l)` – Performs a Test&Test&Set operation [18] on the lock, `l`, returns `TRUE` if the lock was captured.

The counters at the tree's leaves are implemented using a hardware `F&I` operation. The code translates into the following sequence of operations performed by a processor, p, shepherding a token through a balancer, b_0 (see also accompanying illustration in Figure 4). First, p announces its arrival at b_0, by writing b_0 to `location[`p`]` (line 3). It then swaps its own PID for the one written in a randomly chosen location in the `prism` array (line 4-5). Assuming it has read the PID of an existing processor (e.g. r), p attempts to collide with it.

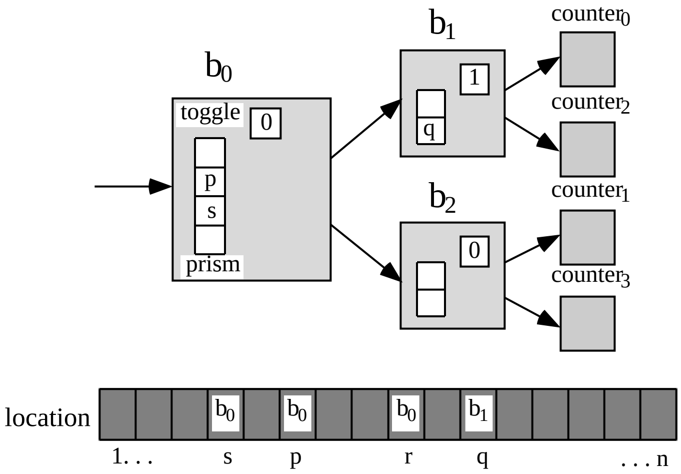

Figure 4: A diffracting tree of width 4

The collision itself is accomplished by performing two compare-and-swap operations. The first removes p from the set of processors waiting at this balancer (thus assuring no other processor will collide with it), the second removes the other processor, completing the diffraction, and allowing p to be diffracted to the `next[0]` balancer (lines 6-8). If the first compare-and-swap fails, it means that some other processor has already managed to collide with p, so p is diffracted to the `next[1]` balancer (line 10). If the first succeeds but the second compare-and-swap fails, it means that the processor with whom p was trying to collide is no longer available, (e.g. if p were trying to collide with q), in which case it goes on to the second part of the algorithm (line 9).

This part starts with p giving some other processor, who may have read its ID from `prism`, time to diffract it. This is done by repeatedly reading the value of `location[`p`] spin` times (lines 12-15). Unless diffracted, p now attempts to acquire the lock on the toggle bit (line 16). If successful, it first removes itself from the set of waiting processors (line 17) and then toggles the bit and exits the balancer (lines 18-21). If it could not remove itself from the set, it follows that some other processor already collided with p, and it exits the balancer, being diffracted to `next[1]` (lines 23-24).

```
function do-node(node: ptr to balancer) returns ptr to balancer
begin
  location[MYID] = node

  rand_place = random(node.size)
  his_id = SWAP(node.prism[rand_place],MYID)
  if C&S(location[MYID],node,EMPTY) then
     if C&S(location[his_id],node,EMPTY) then
        return node.next[0]
     else location[MYID] = node
  else return node.next[1]
  repeat forever
    repeat node.spin times
          if location[MYID] != node
             return node.next[1]
    endrepeat

    if T&T&S(node.lock) then
       if C&S(location[MYID],node,EMPTY) then
          bit_val = node.toggle_bit
          node.toggle_bit = 1 - bit_val
          node.lock = OPEN
          return node.next[bit_val]
       else
          node.lock = OPEN
          return node.next[1]
       endif
    endif
   endrepeat
end
```

Figure 5: Original version of diffraction node algorithm

2.2 The Critical Parameters

As a rule of thumb, when a large number of processors concurrently enter the balancer, the chances for successful collisions in `prism` are high, and contention on the toggle bit is unlikely. When there are few processors, each will spin a short while, reach for the toggle bit and be off, since all "spinning" is done on a locally cached copy of a memory location, it incurs no overhead. However, there is a large range of concurrency levels where there are moderate numbers of processors, and yet it is far from clear what level of diffraction is achieved. Furthermore, it was observed by [19, 21] that too many concurrent processors can also cause performance degradation.

This brings us to the questions most often asked by practitioners implementing diffracting trees [13]. Given a system with P processors that cycle repeatedly between performing an increment on a diffracting tree based counter and performing some amount of *work*, what are the optimal choices of:

1. `d` — The depth, and hence the "size" of the tree (the width is 2^d). If the tree is too small, it will be overloaded, bringing contention and less parallelism than possible. If it is too deep, the counters at its leaves will not be fully utilized, achieving less than optimal throughput.
2. `L` — The total number of `prism` locations at a given level of the tree. At level i there are 2^i `prisms` of size `L`$/2^i$. This parameter affects the chances of a successful pairing-off. If it is too large, then processors will tend to miss each other, failing to pair-off and causing contention on the toggle bit. If it is too small, contention and sequential bottlenecking will occur as too many processors will be trying to access the same `prism` locations at the same time.

It is these and similar questions that our work attempts to address.

2.3 The New Algorithm

We begin by modifying the diffracting tree algorithm presented in Section 2.1.

Touitou [22] reports the following when running a benchmark on the prototype MIT Alewife machine [2]. In his benchmark, processors repeatedly attempt to increment a diffracting tree based counter until some fixed number of increments has been performed. During sufficiently long runs, some processors end up performing all the increments, while all others remain "starving" in the tree. He conjectured that this is caused by processors that were not diffracted and queue up in front of the lock on a toggle bit. The solution was to add a second layer prism between the first layer and the toggle bit, a method which empirically exhibits more stability at the price of slightly increased latency [21]. The combinatorial model of the next section shows that this form of starvation is an inherent phenomenon in the old code due to the fact that processors that do not diffract can leave the balancer only by toggling the shared bit, that is, by passing through a sequential bottleneck. Our analysis shows that in sufficiently long runs one will reach a permanent global state in which processors are piled up at the toggle bits. This would also be true of the method of [21] unless many levels of prisms a re used, resulting in poor latency.

The improved algorithm presented in this article solves this problem by allowing processors to repeatedly return to attempt diffractions on the prism after failing to acquire the toggle bit. It is a dynamic form of the method used by [21], but does not suffer from the same latency increase since it always uses the "right" number of prisms. Figure 6 shows the new algorithm. The `forever` loop has been moved up to encompass the entire diffraction attempt (Instead of being between lines 11 and 12 it is now on line 4). Now if a processor could not diffract the processor whose ID it has read from `prism`, and was not subsequently diffracted by a processor who read its ID, and could not acquire the lock on the toggle bit, then it will go on to make a fresh diffraction attempt, starting the process anew.

The method suggested in [19] to overcome starvation was to allow processors waiting for the toggle bit to rewrite their IDs to `prism` so that later arriving processors might diffract them (this is equivalent to adding the code `node.prism[rand_place]=MYID` between lines 26 and 27 of Figure 5). This offers only a partial remedy, since if many processors wait for the toggle bit, no diffractions occur even if the processors' PIDs are written in the prism array. In any sufficiently long run some processors will get stuck, forever waiting for the toggle bit. In the next section we prove that the new algorithm, when run with the optimal tree of depth d and the optimal prism width $\mathtt{L}/2^i$, does not suffer from this starvation phenomenon.

Figure 6 also gives the code for the dynamic update of the `spin` variable (lines 14-16 and 25-27), a performance enhancement technique that was used both in [19] and here. The spin variable serves both as a delay in which a processor may be diffracted and as a method to exponentially back-off from the toggle bit. `Spin` time is doubled when a processor is diffracted and halved if it captures the lock on the toggle bit. The reasoning behind the update policy is that if a processor is diffracted, it implies there are many other processors in the system, if it has captured the toggle bit, there are probably only a few active processors. With many processors, waiting avoids overloading the toggle bit and has a good chance of yielding a diffraction, with only a few processors, waiting for a diffraction is a waste of time – better to go for the toggle bit directly. `MAXSPIN` is a system dependent constant which defines the maximum amount of time a processor might spin.

The method used in [20] to prove correctness is based on analysis of the

```
function do-node(node: ptr to balancer) returns ptr to balancer
begin
  location[MYID] = node

  forever           /* Moved up to encompass entire algorithm */
     rand_place = random(node.size)
     his_id = SWAP(node.prism[rand_place],MYID)
     if C&S(location[MYID],node,EMPTY) then
        if C&S(location[his_id],node,EMPTY) then
           return node.next[0]
        else location[MYID] = node
     else return node.next[1]

     repeat node.spin times
        if location[MYID] != node then
        /* diffracted? probably a high load better to spin longer */
            if node.spin < MAXSPIN then
               node.spin = node.spin * 2
            endif
            return node.next[1]
        endif
     endrepeat

     if T&T&S(node.lock) then
        if C&S(location[MYID],node,EMPTY) then
           bit_val = node.toggle_bit
           node.toggle_bit = 1 - bit_val
           node.lock = OPEN
          /* toggled? probably a low load better to spin less */
           if node.spin > 1 then
              node.spin = node.spin / 2
           endif
           return node.next[bit_val]
        else
           node.lock = OPEN
           return node.next[1]
        endif
     endif

  endfor
end
```

Figure 6: New version of diffraction node algorithm

different values taken by the elements of the `location` array during the execution of the algorithm. Those methods carry over to the new algorithm with only slight modifications. We will show this for the most important lemma of [20], the rest of the proof can be deduced in a similar manner. The proof is constructed around the pairing of *canceling tokens*, those that leave the balancer through the `return node.next[0]` of line 9, and *canceled* tokens, those that leave the balancer through the `return node.next[1]` of lines 11, 17 or 31. Since all other tokens go through the toggle bit, showing that the number of canceled tokens is equal to the number of canceling tokens is enough to prove that a balance is maintained on the balancer's output wires. We denote by $C\&S_p(\texttt{location}[r], b, \texttt{EMPTY}) = \texttt{true}$, a successful compare-and-swap operation performed by processor p on `location[`r`]`, changing its value from b to `EMPTY`. Similarly, $write_q(\texttt{location}[r] := b)$, denotes the operation of writing b to `location[`r`]` by q.

Lemma 2.1 *Given processors $q \neq r$, if q performs $C\&S_q(\texttt{location}[r], b, \texttt{EMPTY}) = \texttt{true}$, then the token currently shepherded by r through b is a canceled token.*

Proof: Lemmas 5.8 and 5.10 of [20] prove that if q's *C&S* operation, on r's element of `location` was successful, then r was in fact shepherding a token though b at that same time. This token is performing operations in the code somewhere between lines 4 and 34. In order for r's token to leave through one of the `returns` on lines 9 or 28, it must perform a $C\&S_r(\texttt{location}[r], b, \texttt{EMPTY}) = \texttt{true}$ operation. This can't happen since the success of q's operation changed `location[`r`]` to empty. Since r can only shepherd one token at a time, and only r can perform a $write_r(\texttt{location}[r] := b)$, it follows that all subsequent $C\&S_r(\texttt{location}[r], b, \texttt{EMPTY})$ operations must fail. If r's token can't leave through lines 9 or 28, it is a canceled token. ■

3 The Combinatorial Model

We use the following steady-state combinatorial process to model the performance of diffracting trees in the shared memory environment.

The diffracting tree has depth d. The root, which is at level 0, is a balancer with a prism array of L cells, and one toggle bit. Level $i < d$ has 2^i balancers, each with a prism of $\lceil L2^{-i} \rceil$ cells, and one toggle. Each leaf of the tree has one counter.

The combinatorial process works as follows: There are P processors. Each processor proceeds from root to leaf via a sequence of balancers on increasing levels. Once a processor reaches a leaf (which is a counter, on which a fast operation such as a hardware *fetch&increment* is performed) it proceeds to a 'working state'. It returns to the root of the tree after r steps, where r is distributed geometrically $G(\frac{1}{work})$, (i.e., the expected time that a process stays in the 'work' state is *work*). Our model assumes the empirically verified fact that, under equal loads, an operation on a counter is at least as fast as the sequence of operations performed when diffracting on a `prism`. This is true, so long as contention on the counter is not too great.

Each step has two parts. In the first part each processor currently at a balancer chooses a random cell of the prism. If two processors choose the same prism cell, both move to the next level. If only one chooses that cell, it stays at the same level. If more than two choose the same prism cell simultaneously, two move to the next level, the rest stay at their current balancer. In the second part of the step, each processor that it still in that balancer tries to reach the toggle. If at least one processor reaches the toggle, one processor moves to the next balancer, and the toggle changes its state. Each "step" in our model is a simplification of the actual algorithm, since it represents sequences of operations that in practice vary in their execution time in different balancers. Also, we ignore interference between processors, if three processors pick the same location in the prism, we assume two will be diffracted, in reality, the third might interfere with the diffraction of the other two. Another simplification is the assumption that a processor appears in only one place in the prism, this is inaccurate. Since IDs are only erased from the prism as a result of swapping, it is possible for some processor's ID to be written in many places in the prism. Nevertheless, we will show the model is rich enough to accurately predict empirically observed behavior.

3.1 Analysis

Label the nodes of the tree $1, ..., 2^{d+1}-1$ in a breadth first search order, i.e. the 2^i nodes at level i have labels $2^i, ..., 2^{i+1}-1$. Let X_j^t denote the number of processors at node j at time t, let W^t denote the number of processors in the 'work' state at time t, and let τ_j^t denote the state of the toggle of balancer j at time t. Let $\bar{X}^t = (X_1^t, ..., X_{2^{d+1}-1}^t)$, and $\bar{\tau}^t = (\tau_1^t, ..., \tau_{2^{d+1}-1}^t)$. Clearly $\{(\bar{X}^t, W^t, \bar{\tau}^t), \mid t \geq 1\}$ defines a Markov chain. This chain is finite, aperiodic, and irreducible, thus it has a stationary distribution. Our goal is to characterize the performance of the diffracting tree process in the stationary distribution as a function of P, L, d, and *work*.

Let Z_i^t denote the number of processors moving from level $i < d$ to level $i+1$ at step t. Since in the stationary distribution $E[X_j^t] = E[X_j^{t+1}]$, the expected number of processors moving into a balancer in a given step equals the expected number of processors moving out of that balancer. Thus, in the steady state

$$E[Z_i^t] \;=\; E[Z_{i+1}^t] \;=\; E[Z],$$

where $E[Z]$ denote the expected number of processors moving from any one level of the tree to the next level.

The value we are interested in is $E[T]$, the expected number of steps, in the steady state, from the time a processor enters the root of the tree until the time it returns to the root. We first prove a relation between $E[T]$ and $E[Z]$.

Lemma 3.1

$$E[T] = \frac{P}{E[Z]}.$$

Proof: Let Q_i be the probability that in the stationary distribution a given processor at level i proceeds to level $i+1$ of the tree at a given step. Let $E[Y_i]$ denote the expected number of processor at level i. Since $E[Y_i]Q_i = E[Z]$, $Q_i = \frac{E[Z]}{E[Y_i]}$. Let $E[W]$ denote the expected number of processors at the 'work' state in the stationary distribution, then $E[Z] = \frac{E[W]}{work}$. Thus,

$$E[T] = work + \sum_{i=1}^{d} \frac{1}{Q_i} = \frac{E[W]}{E[Z]} + \frac{P - E[W]}{E[Z]} = \frac{P}{E[Z]}.$$

■

We do not have a full characterization of the stationary distribution, but we can obtain a sufficiently good estimate for $E[Z]$. To simplify the presentation we focus on the case in which $work = 1$, which was also studied in the simulations.

Theorem 3.2 *Let $\alpha = \frac{P}{dL}$. In the steady state distribution*

$$E[Z] \geq (1 - o(1))2L\frac{\alpha^2}{\alpha^2 + 2\alpha + 1}.$$

Proof: Since we are interested in a lower bound for $E[Z]$ we can ignore the contribution of the toggles. To approximate the performance of the discrete time Markov chain we study a related continuous time, density dependent jump Markov chain (see [15, Chapers 7-8] or [10, Chapter 11] for detailed discussion of density dependent jump Markov processes and the convergence theorem we use here).

Processors in the continuous Markov process execute the same steps as in the discrete process. The only difference is that in the continuous process the time interval between any two actions of a processor is a random variable exponentially distributed with expectation 1 (instead of deterministically 1 in the discrete process).

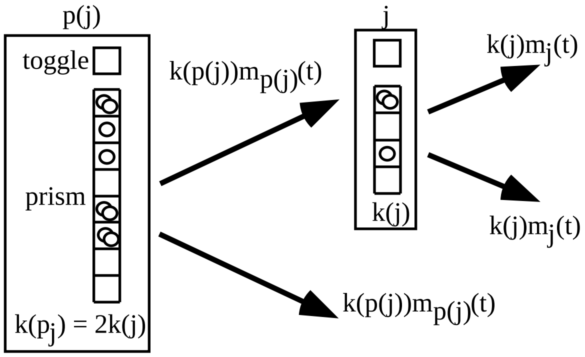

Figure 7: A model of processor advancement in a diffracting tree

Without loss of generality we can assume that no two events occur simultaneously in the continuous process. We need however to carefully define the diffraction process so that the continuous process accurately models the discrete one. When processor p_i is active, it chooses a prism cell and stays there for a random interval of time, till the next time it is active. If the prism cell has no diffraction pair in it at time t we say that the cell is 'free'. If p_i chose a free cell that already has another processor that chose that cell after it became free the two processors are matched, and the cell becomes 'occupied'. After another random time interval the two matched processors are diffracted to the next level of the tree, and the cell becomes free again. Note that processors that chose a cell when it was occupied are not diffracted even if they stay there after it becomes free.

Let $\ell = 2^{d+1} - 1$. The state of the continuous process at time t is defined by three vectors $\bar{s}(t) = (s_1(t), ..., s_\ell(t))$, $\bar{m}(t) = (m_1(t), ..., m_\ell(t))$, and $\bar{x}(t) = (x_1(t), ..., x_\ell(t))$. Where $s_i(t)$ is the fraction of prism cells at node i which are free and have one new processor (a processor that arrived after it became free) at time t, $m_i(t)$ is the fraction of occupied cells at node i at time t, and x_i is the number of processors at node i at time t divided by the number of prism cells at that node.

We formulate a system of differential equations that measures the expected change in the system's state in a short interval of time. We denote the parent of node j by $p(j)$.

For $j = 1, ..., \ell$:

$$\left\{ \begin{array}{rcl} \frac{dx_j}{dt} & = & 2m_{p(i)} - 2m_j \\ \frac{ds_j}{dt} & = & (x_j - 2m_j + 2m_{p(j)})(1 - 2s_j - m_j) - s_j \\ \frac{dm_j}{dt} & = & (x_j - 2m_j + 2m_{p(j)})s_j - m_j \end{array} \right. \tag{1}$$

To see the relation between the above system and the Markov process note that in the continuous model the expected number of events in interval dt among g processors is gdt. Let $k(j)$ denote the number of prism cells at node j, then the change in $x_j(t)$ in interval dt is given by

$$dx_j(t) = \frac{1}{k(j)}(k(p(j))m_{p(j)}(t)dt - 2m_j(t)k(j)dt).$$

Since $k(p(i)) = 2k(i)$ we get the first differential equation above (see illustration in Figure 7). We get the second equation by observing that a total of $k(j)(x_j(t) - 2m_j(t) + 2m_{p(j)}(t))$ processors will choose a cell in node j when they become active. $k(j)(x_j(t) - 2m_j(t) + 2m_{p(j)}(t))dt$ processors become active in time interval dt. Each has probability $(1 - s_j(t) - m_j(t))$ to pick a free cell with no new processors, and probability s_j to transform a free cell with one new processor to an occupied processor. The last term counts free cells with one new processors that loose that processor. Similar derivations give the third relation.

Consider first a deterministic process controlled by the above set of differential equations. A necessary and sufficient condition for $(\bar{s}, \bar{m}, \bar{x})$ to be a fixed point of that process is that for all i $\frac{dx_i}{dt} = 0$, $\frac{ds_i}{dt} = 0$, and $\frac{dm_i}{dt} = 0$. The solution of the above system gives:

$$m_j = \frac{\alpha^2}{\alpha^2 + 2\alpha + 1}.$$

The density dependent jump Markov process satisfies the conditions of Kurtz's convergence theorem (see Theorem 8.1 in [15]). Thus, as $L, d \rightarrow \infty$ the behavior of the Markov process converges to that of the deterministic process. ■

We can now use the above analysis to characterize the performance of the diffracting tree. We need however to add another bound which we ignored above, namely that there are exactly 2^d counters at the leaves of the tree.

Consider the case in which $\frac{P}{dL} \rightarrow \infty$. In that case $E[Z] = MIN[L - o(L), 2^d]$, and $E[T] = MAX[\frac{P}{L}, P2^{-d}]$. If $L < 2^d$ then congestion in the prism cells degrades the performance, if $L > 2^d$ the main congestion is in the counters. In both cases the performance is less than optimal.

If P satisfies $\frac{P}{dL} = O(1)$, then $E[Z] = MIN[O(L), 2^d]$. If $L \leq 2^d$, then $E[Z]$ is linear in L, and $E[T] = O(P/L) = O(d)$, which is optimal up to a constant factor.

As P gets smaller, $\frac{P}{dL} \rightarrow 0$, the diffracting probability decreases, and the performance degrades. If $P = \Omega(d\sqrt{L})$, $E[Z] = \Omega(\frac{P^2}{d^2 L}) \geq \frac{P}{d\sqrt{L}}$, and $E[T] = O(d\sqrt{L})$. If $P = o(d\sqrt{L})$, most of the contribution is from the toggles. As long as $P > d$ at least one processor moves forward in each level, $E[Z] \geq 1$, and $E[T] \leq P$. If $P < d$, $E[T] \leq d$.

Note that when $P = O(dL)$, which is the interesting range, the expected number of processors trying to access a prism cell simultaneously is $O(1)$, and with high probability no more than $\log L$ processors try to reach the same cell simultaneously. These bounds justify our definition of a step, and conform with the experimental results showing that diffracting trees have low memory contention.

The starvation observed in experimenting with the old algorithm can be easily understood when it is analyzed in our model. We can show that in each step in each balancer a constant fraction of the processors reach the toggle, and since in that algorithm processors do not return to the prism to try and diffract again, and given that the toggle processes only one processor at a time, there is congestion built up in the toggle queue.

Finally we comment about adapting the diffracting tree to various ratios between the speed of accessing a prism cell and the speed of the counter. If the counter is ℓ times faster than a diffracting process, we can trim the diffracting tree so that a prism of size ℓ feeds one counter. On the other hand if the speed of a diffraction or a toggle step is ℓ times faster than the counter, each leaf of the full diffracting tree should feed a binary tree of depth $\lceil \log_2 \ell \rceil$ with ℓ counters in the leaves.

4 Experimental Results

In order to verify the validity of our theoretical analysis we ran a set of benchmarks on a simulated distributed-shared-memory multiprocessor similar to the MIT *Alewife* machine [2] developed by Agarwal, et. al. Alewife is a large-scale multiprocessor that supports cache-coherent distributed shared memory and user-level message-passing. The nodes communicate via messages on a two-dimensional mesh network. A Communication and Memory Management Unit on each node holds the cache tags and implements the memory coherence protocol by synthesizing messages to other nodes. Our experiments make use of the shared memory interface only. To simulate the Alewife we used *Proteus*[1], a mul-

[1] Version 3.00, dated February 18, 1993.

```
global
  integer  work  /* the work parameter */
  integer  sum_latency, latencyN, list_place
  diffracting_tree counter
  real avg_latency[], total_latency

per processor code {
    local
      integer latency, start, end, i, j, randw
    forever
       start = current_time()
       i = fetch_and_increment(counter)
       end = current_time()
       atomically {
         latency = end - start
         sum_latency = sum_latency + latency
         latencyN = latencyN + 1
         if latencyN  == 1000 then
            avg_latency[list_place] =
                      sum_latency / latencyN
            list_place = list_place + 1
            sum_latency = 0
            latencyN = 0
         endif
       }
       randw = random(work)
       repeat randw times
         /* nothing */
       endrepeat
       if i > MAXINDEX then quit
    endfor
}

when all processors are done do {
  local
     integer i
  for i = 2 to list_place-1 do
     total_latency = total_latency + avg_latency[i]
  endfor
  total_latency = total_latency / ( list_place - 2 )
}
```

Figure 8: Code for Measuring Fetch&Increment Latency

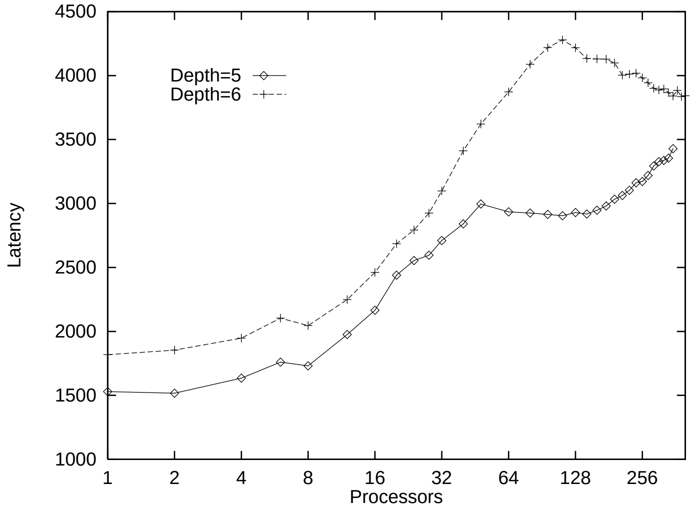

Figure 9: Latency of diffracting trees, logarithmic scale

tiprocessor simulator developed by Brewer, Dellarocas, Colbrook and Weihl [7]. Proteus simulates parallel code by multiplexing several parallel threads on a single CPU. Each thread runs on its own virtual CPU with accompanying local memory, cache and communications hardware, keeping track of how much time is spent using each component. In order to facilitate fast simulations, Proteus does not do complete hardware simulations. Instead, operations which are local (do not interact with the parallel environment) are run uninterruptedly on the simulating machine's CPU and memory. The amount of time used for local calculations is added to the time spent performing (simulated) globally visible operations to derive each thread's notion of the current time. Proteus makes sure a thread can only see global events within the scope of its local time.

In our benchmarks we measured the average latency of processors accessing a distributed Fetch&Increment counter implemented as a diffracting tree with hardware Fetch&Increment counters at its leaves. The average latency is the average number of cycles it takes the counter to deliver an index. In these experiments `work` was very close to 0. In each simulation a counter was accessed between 10,000 and 20,000 times, and the time to deliver an index was measured for each access. The average latency was measured after each 1000 indices delivered, the average of these times is the latency of the counter. In order to take into account start-up times we ignored the latency of the first 2,000 indices delivered. The pseudo-code in Figure 8 illustrates how the measurements were performed. In the code, `current_time` gives the number of cycles since some

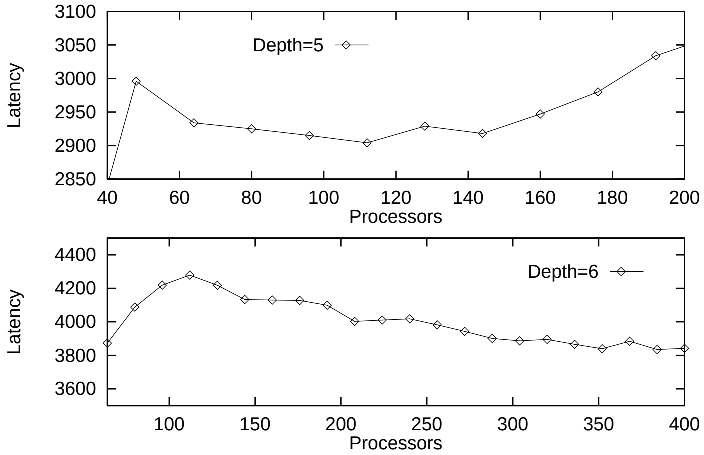

Figure 10: Latency of diffracting trees, linear scale

agreed global event, `fetch_and_increment` is the diffracting tree, `MAXINDEX` is the last index to be delivered and `total_latency` is estimate of the counter's latency. By monitoring the different values in the `avg_latency` array for different values of `i`, we can make sure that the simulation has reached a steady state. The random number function we used was Proteus' `fast_random()` which is an implementation of the ACM Minimal Standard Random Number Generator [17, 8].

We now show how our combinatorial model ties together the choice of diffracting tree parameters depth, d, and prism locations per level, L, to the number of processor, P. A diffracting tree is shown to operate optimally when $P = O(dL)$ and $L = 2^d$ (the number of counters), i.e. the number of processors should be approximately equal to the number of prism locations in the tree. The constant hidden by the O notation is small and depends on a particular machine's ability to handle multiple accesses to the same memory location. This is an expected result and fits well with the saturation model of Aspnes, Herlihy, and Shavit for counting networks [4].

The following figures show how our model accurately predicts the experimental results. Figure 9 shows the latency of diffracting trees five and six levels deep. In these experiments we use binary trees with d levels (meaning 2^d counters) and $L = 2^d$. The graphs have a distinctive shape. The left hand part corresponding to a small number of processors shows a low latency, that increases as more processors are added. When the number of processors is very small, the slope of the graph is low, indicating a nearly constant latency, this fits the term $E[T] = d$. As more processors are added, the slope increases due to the sequential bottleneck at the toggle bits, this fits the term $E[T] = O(P)$ for the range $d \leq P \leq d\sqrt{L}$. There is a local maxima of bad performance reached when $P = O(d\sqrt{L})$, here, as our model predicts, there are too few processors to achieve diffractions, but too many to be processed by the toggle bit. At this point, we approach the bound $E[T] \leq O(d\sqrt{L})$ which is the algorithm's worst case performance. In fact, these results imply that one should avoid using the trees in this range of concurrency. As more processors are added latency decreases linearly, in accordance with the formula $E[T] = O(P/L)$. The close-up graphs in Figure 10, especially the depth 6 tree, show this linear decrease well. The depth 5 tree also shows how latency increases again as concurrency increases. Note that the calibration of our graphs, and hence the phenomena we are modeling, are very fine relative to the changes in latency for other types of data structures. For example, in [19], *combining trees* [11] are shown to have a latency increase by 2500 units over the tested concurrency range, and so the 300 unit change in latency of diffracting trees would be considered almost constant. See [19] for details.

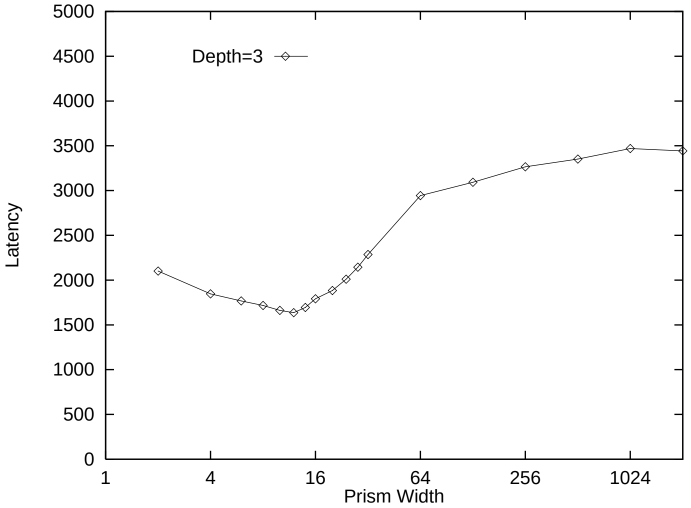

Figure 11: Latency of diffracting trees with different width prisms

Figure 11 shows the effects of changing L, the total number of prism locations in a level of the diffracting tree while keeping the other parameters constant. Here we used a tree of depth 3 and 64 processors, with almost no *work*. The number of counters in the tree does not change, it remains $2^d = 8$. We vary the number of locations in a prism array of a balancer at level i so that there are

$\lceil L/2^i \rceil$ per prism. The left hand side of the graph corresponds to a small L and a large P, this approximates the case where $\frac{P-L}{dL} \rightarrow \infty$. We expect the latency to behave as $E[T] = \frac{P}{L}$, and this is indeed the case. When L is this small diffractions are constantly occurring on the prisms which can't keep up with the flow of new processors. This situation continues up to an optimum point, after which increasing L lowers the chance that two processors will pick the same location in the prism, and thus latency begins to rise. Since the number of diffractions in the tree is $O(\frac{P^2}{dL})$ we get a linear increase in latency. We can expect the rise in latency to taper off when L is large such that no diffractions are occurring, this can be observed in the right hand side of the graph.

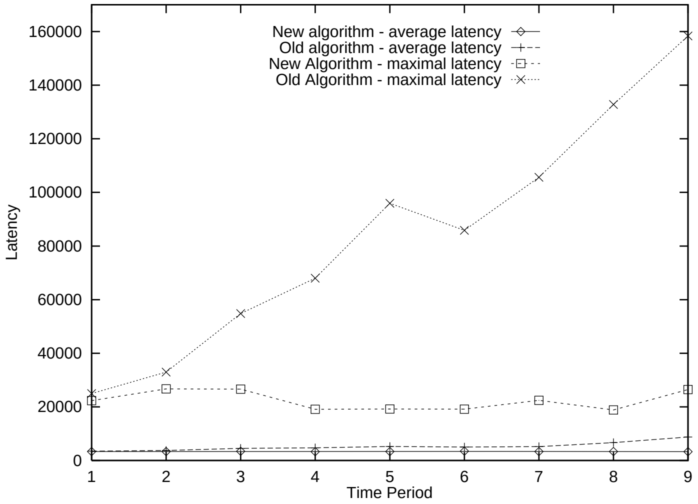

Figure 12: Comparison of old and new diffracting tree algorithms in terms of average and maximal latency.

Finally, we present empirical evidence which indicates that the new algorithm solves the starvation problem discussed in Section 2.3. Figure 12 shows a comparison between the average and maximal latencies of the old and new algorithms. In these experiments we ran a diffracting tree based counter for several thousand increment requests. After every thousand requests we measured the average time for an index to be delivered and the maximum time any single processor waited for an index. As can be seen, maximal times increase rapidly in the old version – an indication of starvation, but remain stable for the new version implying that the starvation problem has indeed been remedied.

5 Conclusions

In this paper we have presented the first analysis of diffracting trees capable of addressing critical design issues. We have identified four ranges of P for which there are specific performance bounds.

- If $\frac{P}{dL} = O(1)$ and $L \leq 2^d$, then the throughput of the system is optimal and the expected latency is $O(d)$, which is optimal up to a constant factor.
- As P gets smaller, the diffracting probability decreases, and the performance degrades, reaching a formerly unnoticed local maxima in latency at about $P = d\sqrt{L}$, where the latency is $O(d\sqrt{L})$.
- For even smaller values of P, processor advancement is mainly due to the toggle bits. Thus when $P = o(d\sqrt{L})$, the expected latency is $\mathrm{MAX}[P, d]$.
- When `work`= $O(1)$ and P is substantially larger than dL, the expected throughput is $L - o(L)$ and if $L < 2^d$ then the expected latency is $\frac{P}{L}$ which is the best one can expect considering the contention on a prism cell. If $L > 2^d$ the main congestion is in the counters. In both cases the performance is less than optimal.

Finally, our model shows that when $P = O(dL)$, which is the optimal range, the contention in the tree is low: the expected number of processors trying to access a prism cell simultaneously is $O(1)$, and with high probability no more than $\log L$ processors try to reach the same cell simultaneously.

We strongly believe our model and modeling approach pave the way to steady-state combinatorial analysis of other distributed-parallel data structures such as counting networks and other diffracting tree based data structures such as elimination trees [21], pools [21], priority queues, and so on.

References

[1] A. Agarwal and M. Cherian. Adaptive Backoff Synchronization Techniques. In *Proceedings of the 16th International Symposium on Computer Architecture*, June 1989.

[2] A. Agarwal et al. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors.* Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[3] B. Aiello, R. Venkatesan and M. Yung. Optimal Depth Counting Networks. personal communication.

[4] J. Aspnes, M.P. Herlihy, and N. Shavit. Counting Networks. *Journal of the ACM*, Vol. 41, No. 5 (September 1994), pp. 1020-1048.

[5] J. Aspnes, M.P. Herlihy, and N. Shavit. Counting Networks and Multi-Processor Coordination. In *Proceedings of the 23rd Annual Symposium on Theory of Computing*, May 1991.

[6] E.A. Brewer, C.N. Dellarocas. PROTEUS *User Documentation*. MIT, 545 Technology Square, Cambridge, MA 02139, 0.5 edition, December 1992.

[7] E.A. Brewer, C.N. Dellarocas, A. Colbrook and W.E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. MIT Technical Report /MIT/LCS/TR-561, September 1991.

[8] D.G. Carta Two Fast Implementations of the "Minimal Standard" Random Number Generator. *CACM*, 33(1), January 1990.

[9] C. Dwork, M. P. Herlihy, and O. Waarts. Contention in shared memory algorithms. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pp. 174-183, May 1993. Expanded version: Digital Equipment Corporation Technical Report CRL 93/12.

[10] S.N. Ethier and T.G. Kurtz. *Markov Processes: Characterization and Convergence.* John Wiley and Sons, 1986.

[11] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent multiprocessors. In *Proceedings of the 3rd ASPLOS*, pages 64–75. ACM, April 1989.

[12] M.P. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

[13] S. Kahan – TERA Computer Company. Personal communication, May 1995.

[14] V.F. Kolchin, B.A. Senast'yanov, and V.P. Chistyakov. *Random Allocation.* V.H. Winston & Sons, Washington D.C. 1978.

[15] T.G. Kurtz. *Approximation of Population Processes.* CBMS-NSF Reginal Conf. Series in Applied Math. SIAM, 1981.

[16] J.M. Mellor-Crummey and M.L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. Technical Report 342, University of Rochester, Rochester, NY 14627, April 1990.

[17] S.K. Park and K.W. Miller. Random number generators: Good ones are hard to find. *CACM*, 31(10),October 1988.

[18] L. Rudolph, Decentralized cache scheme for an MIMD parallel processor. In *11th Annual Computing Architecture Conference,* 1983, pp. 340-347.

[19] N. Shavit and A. Zemach. Diffracting Trees. In *Proceedings of the 6th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 167–174, June 1994.

[20] N. Shavit and A. Zemach. Diffracting Trees. In ACM Transactions on Computer Systems, Nov. 1996.

[21] N. Shavit, and D. Touitou. Elimination Trees and the Construction of Pools and Stacks In *Proceedings of the 7th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 54-63, July 1995.

[22] D. Touitou – Tel-Aviv University. Personal communication, October 1994.