Diffracting Trees

Nir Shavit[†‡] Asaph Zemach[†]

August 15, 1996

Abstract

Shared counters are among the most basic coordination structures in multiprocessor computation, with applications ranging from barrier synchronization to concurrent-data-structure design. This paper introduces *diffracting trees*, novel data structures for shared counting and load balancing in a distributed parallel environment. Empirical evidence, collected on a simulated distributed shared-memory machine and several simulated message passing architectures, shows that diffracting trees scale better and are more robust than both combining trees and counting networks, currently the most effective known methods for implementing concurrent counters in software. The use of a randomized coordination method together with a combinatorial data structure overcomes the resiliency drawbacks of combining trees. Our simulations show that to handle the same load, diffracting trees and counting networks should have a similar width w, yet the depth of a diffracting tree is $O(\log w)$, whereas counting networks have depth $O(\log^2 w)$.

Diffracting trees have already been used to implement highly efficient producer/consumer queues, and we believe diffraction will prove to be an effective alternative paradigm to combining and queue-locking in the design of many concurrent data structures.

[*]A preliminary version of this work appeared in the *Proceedings of the Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1994.

[†]Department of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel.

[‡]Contact Author: E-mail: shanir@math.tau.ac.il.

1 Introduction

It is hard to imagine a program that doesn't count something, and indeed, on multiprocessor machines shared counters are the key to solving a variety of coordination problems such as barrier synchronization [40], index distribution, shared program counters [41] and the design of concurrent data structures such as queues and stacks (see also [19, 22, 47]). In its purest form, a counter is an object that holds an integer value and provides a *fetch-and-increment* operation, incrementing the counter and returning its previous value. Given that the majority of current multiprocessor architectures do not provide specialized hardware support for efficient counting, there is a growing need to develop effective software-based counting methods.

The simplest way to implement a counter is to place it in a spin-lock protected critical section, adding an exponential-back-off mechanism [1, 6, 23] or a queue lock as devised by Anderson [6] and Mellor-Crummey and Scott [40] to reduce contention [20, 49]. Unfortunately, such centralized methods are inherently non-parallel and cannot hope to scale well. This is true also of hardware supported fetch-and-increment operations unless the hardware itself employs one of the parallel methods described below.

A recent survey of counting techniques by Herlihy, Lim, and Shavit [24] suggests that scalable counting can only be achieved by methods that are *distributed* and therefore have low contention on memory and interconnect, and are *parallel*, and thus allow many requests to be dealt with concurrently. The combining trees of Yew, Tzeng, and Lawrie [49] and Goodman, Vernon, and Woest [21], and the counting networks of Aspnes, Herlihy, and Shavit [7], both meet the above criteria, and indeed were found to be the most effective methods for concurrent counting in software.

A combining tree is a distributed binary-tree based data structure with a shared counter at its root. Processors combine their increment requests going up the tree from the leaves to the root, and propagate the answers down the tree, thus eliminating the need for all processors to actually reach the root in order to increment the counter. For n processors optimal combining trees have $O(\log n)$ depth and the desirable property that the unavoidable "collisions" of processors at their nodes are utilized to increase parallelism. At peak performance a combining tree would have a throughput of $n/2 \log n$ indices per time step, that is, n indices are returned every $2 \log n$ steps. However, this throughput is highly dependent on processor timings, and a single processor's delay or failure can delay all others indefinitely.

A Bitonic counting network [7] is a distributed data structure having a layout isomorphic to Batcher's Bitonic sorting network [8], with a "local counter" at the end of each output wire. Unlike queue-locks and combining trees which are based on a single counter location handing out indices, counting networks have a collection of w separate counter locations. To guarantee that indices handed out by the w separate counters are not erroneously "duplicated" or "omitted," one adds a special network coordination structure to be traversed by processes before accessing the

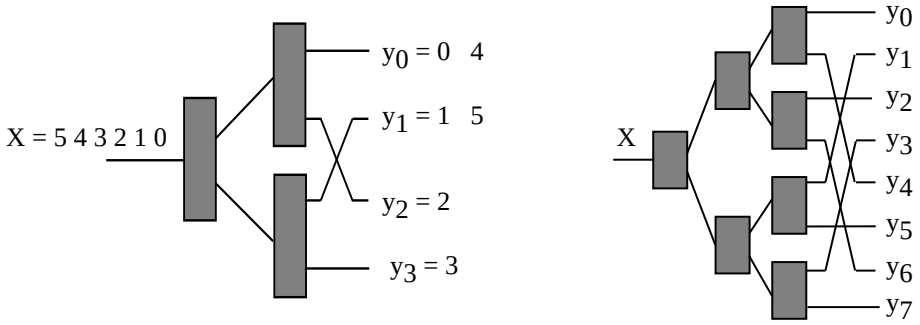

Figure 1: Two Simple Counting Trees

counters. Bitonic counting networks have width $w < n$ and depth $O(\log^2 w)$. Unlike combining trees, counting networks support complete independence among requests and are thus highly fault tolerant. At peak performance their throughput is w, as w indices are returned per time step by the independent counters. Unfortunately, counting networks suffer a performance drop-off due to contention as concurrency increases, and the latency in traversing them is a high $O(\log^2 w)$. There is a wide body of theoretical research analyzing the performance of counting networks and attempting to improve on their $O(\log^2 w)$ depth [2, 5, 7, 12, 13, 18, 27, 32, 33]. The most effective is the elegant combinatorial design due to Klugerman and Plaxton [32, 33] of depth close to $O(\log w)$. Unfortunately, the "exponentially large" constants involved make these constructions impractical.

This paper introduces *diffracting trees*, a new distributed technique for shared counting, enjoying the benefits of the above methods and avoiding many of their drawbacks. Diffracting trees, like counting networks [7], are 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 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 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. The left-hand side of Figure 1 shows such an execution on a tree of width 4. As can be seen, if the output wires are arranged correctly, the tree will move 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. As in the case of counting networks, counting is done by adding a local counter to each output wire i, so that tokens coming out of that wire are assigned numbers $i, i+4, i+(4\cdot 2)\ldots$

A clear advantage of the tree over a counting network is its depth which is logarithmic in w. This means that it can support the same kind of throughput to w independent counters with much

lower latency. However, it seems that we are back to square one since the root of the tree will be a "hot spot" [20, 42] and a sequential bottleneck that is no better than a centralized counter implementation. This would indeed be true if one were to use the accepted (counting network) implementation of a balancer – a single location with a bit toggled by each passing token. The problem is overcome based on the following simple observation: if an even number of tokens pass through a balancer they leave the toggle bit state unchanged. Thus, if one could have pairs of tokens collide and then diffract in a coordinated manner one to the left and one to the right, both could leave the balancer without ever having to toggle the shared bit. This bit will only be accessed by processors that did not collide. *Diffracting trees* implement this approach by adding a software "prism" in front of the toggle bit of every balancer (see Figure 3). The prism is an inherently distributed data structure that allows many diffractions to occur in parallel. Processors select prism locations uniformly at random to ensure load balancing and high collision/diffraction rates. The tree structure guarantees correctness of the output values. *Diffracting trees* thus combine the the high degree of parallelism and fault-tolerance of counting networks with the beneficial utilization of "collisions" of a combining tree.

We compared the performance of diffracting trees to the above methods in simulated shared memory and message passing environments. The Proteus Parallel Hardware Simulator [10, 11] of Brewer, Dellarocas, Colbrook and Weihl was used to evaluate performance in a shared memory architecture similar to the Alewife machine of Agarwal, Chaiken, Johnson, Krantz, Kubiatowicz, Kurihara, Lim, Maa, and Nussbaumet [3]. Netsim, part of the Rice Parallel Processing Testbed [15, 29] developed by Covington, Dwarkadas, Jump, Sinclair, and Madala was used for testing in message passing architectures. We found that, in shared-memory systems, diffracting trees substantially outperform both combining trees and counting networks, currently the most effective known methods for shared counting. They scale better, giving higher throughput over a large number of processors, and are more robust in terms of their ability to handle unexpected latencies and differing loads. Note also that like counting networks but unlike combining trees, diffracting trees can be implemented in a wait-free [26] manner (given the appropriate hardware primitives). By this we mean that for each increment operation termination is guaranteed in a bounded number of steps independently of the pace or even a possible halting failure of all other processors. In message passing environments, we analyzed the effects of network bandwidth and locality on these distributed data structures. We found that in low bandwidth mesh networks combining trees can be optimally placed so that they are by far the most effective method, but only when the load is very high. A drop in the load immediately results in poor combining and the performance falls below that of the more robust diffracting tree. In other architectures where locality plays a lesser role or where wider bandwidth is available, all the methods have comparable behavior. In a butterfly type network, which has no locality and low bandwidth, diffracting trees substantially outperform other methods.

In summary, we believe diffraction will prove to be an effective alternative paradigm to combining in the design of many concurrent data structures and algorithms for multi-scale computing.

This paper is organized as follows: Section 2 describes tree counting networks with the BINARY[w] layout and introduces diffracting trees, Section 3 gives the shared memory implementation of diffracting trees and performance results on Proteus, Section 4 has the message passing implementation and results of Netsim simulations, Section 5 contains formal correctness proofs for all our constructions, and Section 6 concludes this paper and lists areas of further research.

2 Trees that Count

We begin by introducing the abstract notion of a counting tree, a special form of the counting network data structures introduced by Aspnes, Herlihy, and Shavit [7]. A counting tree *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 number 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 the wire.

A *balancing tree* of width w is a binary tree of balancers, where output wires of one balancer are connected to input wires of another, having one designated root input wire and w designated output wires: $y_0, y_1, .., y_{w-1}$. Formal definitions of the properties of balancing networks can be found elsewhere [7]. 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. In a message passing architecture "tokens" would be implemented as messages, and balancers would be processors that receive messages and send them left or right in a balanced way.

We extend the notion of quiescence to trees in the natural way, and define a *counting tree* of width w as a balancing tree whose outputs $y_0, .., y_{w-1}$ 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. The left-hand side of Figure 1 shows such an execution on a BINARY[4] type counting tree (width 4) which we define formally below. As can be seen, the network moves input tokens to output wires in increasing order modulo w. Balancing 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 assigned numbers $i, i+w, \ldots, i+(y_i-1)w$. Code for implementing a simple counting tree can be found in Figure 2. The `increment_counter_at_leaf()` call (line 7 of `fetch&incr`) hides the implementation of a simpler form of counting operation, either one that employs a software lock or through a hardware fetch-and-increment operation.

We use a counting tree called BINARY[w], defined as follows. Let w be a power of two, and let us define the counting tree BINARY[$2k$] inductively. When k is equal to 1, the BINARY[$2k$] tree consists of a single balancer with output wires y_0 and y_1. For $k > 1$, we construct the BINARY[$2k$] tree from two 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 redesignate 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 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 $y_1, y_3, \ldots, y_{2k-1}$. Theorem 5.7 in Section 5.1 proves that BINARY[$2k$] is indeed a counting tree.

To informally understand why BINARY[$2k$] has the step property in a quiescent state, assume inductively that BINARY[k] has the step property in a quiescent state. The root balancer passes at most one token more to the BINARY[k] tree on its "0" (top) wire than on its"1" (bottom) wire. Thus, the tokens exiting the top BINARY[k] subtree have the shape of a step differing from that of the bottom subtree on exactly one wire j among their k output wires. The outputs of the BINARY[$2k$] are a perfect shuffle of the wires stemming from the two subtrees, and it easily follows that the two step-shaped token sequences of width k will form a new step of width $2k$ where the possible single excess token resides in the higher of the two wires j, that is, the one stemming from the top BINARY[k] tree.

2.1 Diffraction Balancing

Consider implementing a BINARY[w] tree using the standard balancer implementation, as in Figure 2. Each processor shepherding a token through the tree toggles a bit inside each balancer encountered, and accordingly decides on which wire to exit. If many tokens attempt to pass through the same balancer concurrently, the toggle bit quickly becomes a hot-spot. Even if one applies contention reduction techniques such as exponential backoff, the toggle bit still forms a sequential bottleneck. Contention would be greatest at the root balancer through which all tokens

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

constants
width: global integer
root : global ptr to root of Binary[width] tree
```

```
1  function typical-balancer(b: ptr to balancer) : ptr to balancer
2  begin
3    lock(b->lock)
4    i := b->toggle
5    b->toggle := not(i)
6    unlock(b->lock)
7    return b->next[i]
8  end
```

```
1  function fetch&incr(): integer
2  begin
3    b:= root
4    while not leaf(b)
5      b := typical-balancer(b)
6    endwhile
7    i := increment_counter_at_leaf(b)
8    return i * width + number_of_leaf(b)
9  end
```

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

must pass. To overcome this difficulty we make use of the following observation:

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

If we could find a method that allows separate pairs of tokens arriving at a balancer to "collide" and coordinate among themselves which is diffracted "right" and which diffracted "left", both could

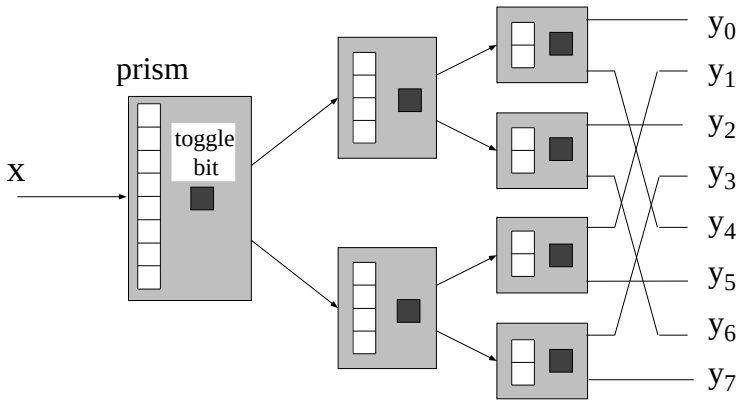

Figure 3: A Diffracting Tree

leave the balancer without either having to touch the toggle bit. This potential hot-spot would only be accessed by those processors that did not manage to collide. By performing the collision/co-ordination decisions independently in separate locations instead of at a single toggle bit, we will hopefully increase parallelism and lower contention. However, we must guarantee that many such collisions occur, not an obvious task given the inherent asynchrony in the system.

Our *diffracting balancer* data structure is based on adding a special `prism` array "in front" of the toggle bit in every balancer. When a token T enters the balancer, it first selects a location, l, in `prism` uniformly at random. T tries to "collide" with the previous token to select l, or, by waiting for a fixed time, with the next token to do so. If a collision occurs, both tokens leave the balancer on separate wires, otherwise the undiffracted token T toggles the bit and leaves accordingly. Figure 3 shows a diffracting tree of width 8.

The next two sections discuss how diffracting trees are implemented in the two major parallel programming paradigms: shared memory (Section 3) and message passing (Section 4).

3 A Shared Memory Implementation

In shared memory a diffracting tree is implemented by a BINARY[w] tree of balancer records. Each processor that wishes to increment the counter, shepherds a token though the tree by executing a program that reads and writes to shared memory. Each balancer record consists of a `toggle` bit (our implementation uses a spin-lock to allow atomic toggling of this bit), and a `prism` array. Additionally, each balancer holds the size of its `prism` array in the variable `size`, the addresses of its descendant balancers (or counters) in `next` and an additional field, `spin`, detailed below. 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.

Figure 4 gives the diffracting balancer data structure and code and Figure 5 illustrates an actual run of the algorithm (detailed below). Three synchronization operations are used in the implementation code:

- `register_to_memory_swap(addr,val)` writes `val` to address `addr`, and returns its previous value,
- `compare_and_swap(addr,old,new)` checks if the value at address `addr` is equal to `old`, and if so, replaces it with `new`, returning `TRUE`, otherwise it just returns `FALSE`, and
- `test_and_set(addr)` writes `TRUE` to address `addr` and returns the previous value.

All three operations can be implemented in a lock-free [25] manner using the load-linked/store-conditional operations available on many modern architectures [16, 39]. On machines like the MIT Alewife [3] that support *full-empty bits* in hardware, the `compare_and_swap` operations can be directly replaced by loads and stores that interact/are-conditioned on the bit [4].

The code translates into the following sequence of operations (illustrated in Figure 5) performed by a process shepherding a token through a balancer. In Phase 1 processor p announces the arrival of its token at balancer b_0, by writing b_0 to `location`[p] (Line 3). Using the routine `random(a,b)`, it chooses a location in the `prism` array uniformly at random and swaps its own PID for the one written there (Lines 4 - 5). Assuming it has read the PID of an existing processor (i.e. `not_empty(him)`), p attempts to diffract it. This diffraction is accomplished by performing two compare-and-swap operations on the `location` array (Lines 6 - 8). The first clears p's element, assuring no other processor will collide with it during the diffraction (this avoids race conditions). The second clears the other processor's element, and completes the diffraction. If both compare-and-swap operations succeed, the diffraction is successful, and p is diffracted to the `b->next[0]` balancer (Line 9). In Figure 5 this might happen if p were trying to diffract r, since examining the location array shows both to be at balancer b_0. If the first compare-and-swap fails, it follows that some other processor has already managed to diffract p, so p is directed to the `b->next[1]` balancer (Line 11). If the first succeeds but the second compare-and-swap fails, then the processor with whom p was trying to collide is no longer available, in which case p goes on to Phase 2, though not before updating `location`[p] to reflect the fact the p is still at b_0 (Line 10). This would happen if, for example, p were trying to diffract q, since q is at balancer b_1 (`location`[q] is b_1, not b_0, causing the second compare-and-swap to fail).

In Phase 2, processor p repeatedly checks to see if it has been diffracted by another processor, by examining `location`[p] `spin` times (Lines 14 - 16). This gives any processor that might have read

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

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

 1  function diff-bal(b: ptr to balancer): ptr to balancer
 2  begin
      /********************* phase 1 **********************/
 3    location[mypid] := b
 4    place := random(1,b->size)
 5    him := register_to_memory_swap(b->prism[place],mypid)
 6    if not_empty(him) then
 7       if compare_and_swap(location[mypid],b,EMPTY) then
 8           if compare_and_swap(location[him],b,EMPTY) then
 9               return b->next[0]
10           else location[mypid] := b
11       else return b->next[1]
12    endif
      /********************* phase 2 **********************/
13    while true
14       repeat b->spin times
15          if location[mypid] <> b then
16             return b->next[1]
17       endrepeat
18       if test_and_set(b->lock) then
19          if compare_and_swap(location[mypid],b,EMPTY)
20          then
21             i := b->toggle
22             b->toggle := not(i)
23             b->lock := FALSE
24             return b->next[i]
25          else
26             b->lock := FALSE
27             return b->next[1]
28          endif
29       endif
30    endwhile
31  end
```

Figure 4: Code for traversing a diffracting balancer

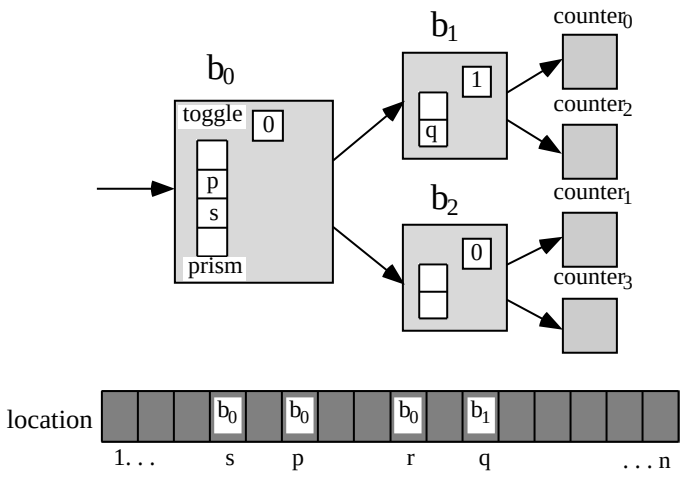

Figure 5: The shared memory implementation of a diffracting tree

p's PID from `prism` time to diffract p. The amount of time is dependent on the value of the `spin` field of each balancer. A higher `spin` value indicates more time is spent waiting to be diffracted. If not diffracted, p attempts to acquire the lock on the toggle bit (Line 18). If successful, p first clears its element of `location`, using compare-and-swap, and then toggles the bit and exits the balancer (Lines 19 - 24). If `location`[p] could not be erased it follows that some other processor already collided with p, and p exits the balancer, being diffracted to `b->next[1]` (Lines 26-27). If the lock could not be seized, processor p resumes spinning.

Notice that before accessing the toggle bit or trying to diffract, p clears `location`[p] using a compare-and-swap operation. The use of compare-and-swap operations guarantees that the same processor p will not be diffracted twice, since success ensures that p has not yet been diffracted. It also guarantees that p will not be diffracted before getting a chance to exit the balancer. This protects us from situations where some processor q is diffracted by p without noticing. The construction works because it assures that for every processor being diffracted left (to `b->next[0]`), there is exactly one processor diffracted right (to `b->next[1]`). Since all other processors go through the toggle bit a balance is maintained. A formal proof is given in Section 5.2.

3.1 Some Implementation Details

The following discussion assumes an implementation on a machine that supports a globally addressable, physically distributed memory model. Each processor has part of the machine's memory adjacent to it, and operates on non-local memory through a network which connects all processors and memory modules. Recently accessed memory is cached locally. Caches are kept up-to-date

through the machine's cache coherency protocol.

On such a machine, when a large number of processors concurrently enter a balancer, the chances for successful collisions in the `prism` are high, and contention on the lock of 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 cached copy of the value of `location[mypid]` it incurs very little overhead. The only case in which a processor repeatedly accesses memory, is when no other processor diffracts it, and it constantly reaches for the lock on the toggle bit. This becomes increasingly unlikely as more processors enter the balancer.

Two parameters are of critical importance to the performance of the diffracting balancer:

1. `size` — This value affects the chances of a successful pairing-off. If it is too high, then processors will tend to miss each other, failing to pair-off and causing contention on the lock of the toggle bit. If it is too low, contention will occur on the `prism` array as too many processors will try to access fewer locations at the same time.

2. `spin` — If this value is too low, processors will not have a chance to pair-off, and there will be contention on the lock of the toggle bit. If it is too high, processors will tend to wait for a long time even though the toggle bit may be free, causing a degradation in performance.

The choice of these parameters is obviously architecture dependent. In our simulations we used for the variable `size` the values 8,4,2,2 and 1, for levels $0, \ldots, 4$ of a width 32 tree respectively. We also used a form of adaptive (exponential) back-off [1] on the `spin` to facilitate rapid access to the toggle bit in reduced load situations. Each processor kept a local copy of the tree's `spin` variables and used them as initial values for the back-off. After each failed attempt at seizing the toggle bit, the processor would double its local `spin` (up to a maximum bound of 128 iterations), thus increasing the amount of time it waited to be diffracted with. However, if the toggle bit was seized, the initial value of `spin` used by the processor in its next pass through this balancer was halved.

Figure 6 shows how these changes are incorporated into the code. A further discussion on the effects of spin optimization is given in Section 4.3. In order to maximize the distribution of the balancer's data structure the elements of the `prism` array were all located to separate modules of memory. Notice that it is possible that some processor will swap another processor's PID from the `prism`, but for some reason not manage to diffract it, despite the fact that both may be at the same balancer. If the second processor's PID is no longer written in the `prism` it will have no chance of being diffracted. To overcome this we enhance performance by giving processors a "second-chance": after spinning at the toggle bit for a while a processor rewrites its PID to the `prism` array and allows itself to be diffracted as in Phase 2 of the code. This increases its chances of being diffracted during a given traversal of a balancer. Correctness of this "second-chance" enhancement follows since the state of the balancer when a token changes from waiting on the toggle to its "second-chance"

waiting on the `prism`, is the same as if it had not yet entered the balancer. (The `location` array entry for it is `EMPTY` and its PID could appear in some entries of the `prism` array but this could as well be the result of accesses to that balancer in earlier tree traversals.) Thus, the correctness proof of the algorithm with the enhancement follows directly from the proof of the original code in Section 5.2, and is left to the interested reader.

3.2 Fault Tolerance

The diffracting tree implementation given in Figure 4 employs the test-and-set operation to lock the balancer's toggle bit. The use of locks is not fault tolerant, if a processor fails inside the critical section it will never release the lock, potentially making further progress impossible. A fault tolerant version of the diffracting tree using a hardware fetch-and-complement operation which atomically flips the value of its argument returning the previous value is described in Figure 7.[1] To complete the fault tolerant construction the local counters at the leaves of the diffracting tree must be made fault tolerant as well. This of course requires the replacement of the locks by a hardware fetch-and-increment operation. (We remind the reader that having hardware support for a fetch-and-increment operation does not obviate the need for the diffracting tree structure, as a single memory location with a hardware fetch-and-increment as a counter would suffer from contention and sequential bottlenecking drawbacks.) The same method can be used to produce fault tolerant counting networks. In fact, replacing the toggling operation with a hardware fetch-and-complement operation would make the diffracting tree and counting network implementations *wait-free* [26]. That is, the number of steps needed to increment the shared counter is bounded by a constant, regardless of the actions of other processors. A formal proof that the implementation in Figure 7 is wait-free is given in Lemma 5.19.

3.3 Performance

We evaluated the performance of diffracting trees relative to other known methods by running a collection of benchmarks on a simulated distributed-shared-memory multiprocessor similar to the MIT *Alewife* machine developed by Agarwal et. al. [3]. Our simulations were performed using *Proteus*[2], a multiprocessor simulator developed by Brewer et. al. [11]. 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

[1] For this purpose a hardware fetch-and-complement is planned to be added to the next version of the Alewife's Sparcle processor [4] as a conditional store operation on a location with a full/empty bit. The new 128 node Alewife machine is due to be operational sometime in 1996.

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

```
    type balancer is
    begin
      shared  /* by all processors */
         prism:  array [1..size] of integer
         lock:   boolean
         toggle: boolean
         next:   array [0..1] of ptr to balancer
         size:   integer
      local   /* each processor has its own copy */
         spin:   integer
    end

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

 1  function diff-bal(b: ptr to balancer): ptr to balancer
 2  begin
      /*************** phase 1 is unchanged ***************/

      /********************** phase 2 **********************/
13    while true
14       repeat b->spin times
15          if location[mypid] <> b then
16             return b->next[1]
17       endrepeat
18       if test_and_set(b->lock) then
19          if compare_and_swap(location[mypid],b,EMPTY)
20          then
21             i := b->toggle
22             b->toggle := not(i)
23             b->lock := FALSE
24             if b->spin > 1 then b->spin := b->spin / 2
25             return b->next[i]
26          else
27             b->lock := FALSE
28             return b->next[1]
29          endif
30       endif
31       if b->spin < MAXSPIN then b->spin := b->spin * 2
32    endwhile
33 end
```

Figure 6: Diffracting balancer with adaptive `spin`

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

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

 1  function wait-free-diff-bal(b: ptr to balancer): ptr to balancer
 2  begin
      /********************* phase 1 **********************/
 3    location[mypid] := b
 4    place := random(1,b->size)
 5    him := register_to_memory_swap(b->prism[place],mypid)
 6    if not_empty(him) then
 7       if compare_and_swap(location[mypid],b,EMPTY) then
 8           if compare_and_swap(location[him],b,EMPTY) then
 9               return b->next[0]
10           else location[mypid] := b
11       else return b->next[1]
12    endif
      /********************* phase 2 **********************/
13    repeat b->spin times
14       if location[mypid] <> b then
15          return b->next[1]
16    endrepeat
17    if compare_and_swap(location[mypid],b,EMPTY) then
18       i := fetch_and_complement(b->toggle)
19       return b->next[i]
20    else
21       return return b->next[1]
22    endif
23  end
```

Figure 7: Code for a fault tolerant diffracting balancer

not perform complete hardware simulations. Instead, operations which are local (do not interact with the parallel environment) are run directly 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.

Two benchmarks were used to test the performance of diffracting trees: index-distribution and job queues.

3.3.1 Index Distribution Benchmark

Index-distribution is a load balancing technique, in which processors dynamically choose loop iterations to execute in parallel. As mentioned elsewhere [24], a simple example of index distribution is the problem of rendering the Mandelbrot Set. Each loop iteration covers a rectangle in the screen. Because rectangles are independent of one another, they can be rendered in parallel, but because some rectangles take unpredictably longer than others, dynamic load-balancing is important for performance. Here is the pseudo-code for this benchmark:

```
Procedure index-dist-bench(work: integer)
  loop:   i := get_next_index()
          delay(random(0,work))
          goto loop
```

In our benchmark, after each index is delivered processors pause for a random amount of time in the range [0,`work`]. When `work` is chosen as 0, this benchmark actually becomes the well known counting benchmark, in which processors attempt to load a shared counter to full capacity.

We ran the benchmark varying the number of processors participating in the simulation (each processor ran only one process), and varying the value of the parameter `work`. In Proteus processes do not begin at exactly the same time, instead, every few cycles a new process begins and this continues until all the processes used in the simulation are running. For this reason the times measured at the start of the simulation are inaccurate and must be ignored. To overcome this problem, we began our measurements after the 100th index was delivered.

The data collected was:

Latency The average amount of time between the moment `get_next_index` was called, and the time it returned with a new index.

Throughput The average number of indices distributed in a one million cycle period. This cycle count includes the `delay()` time. We measured t, the time it took to make d increments. The throughput is $10^6 d/t$.

As a basis for comparison, a collection of the fastest known software counting techniques was used. To make the comparisons fair, the code for each method below was optimized, as was the distribution of the data structures in the machine's memory. The methods are:

ExpBackoff A counter protected by a lock using test-and-test-and-set with exponential backoff [6, 23].

MCS A counter protected by the queue-lock of Mellor-Crummey and Scott [40]. Processors waiting for the lock form a linked list, each pointing to its predecessor. At the "head" of the list is the processor who has the lock. To free the lock, the head processor hands ownership to its successor, and so on, down the list. While waiting for the lock, processors spin locally on their own node in the linked list. The lock has a single "tail" pointer which directs new processors wishing to acquire the lock to the end of the queue. The code was taken directly from Mellor-Crummey and Scott's article [40] and implemented using atomic register-to-memory-swap and compare-and-swap operations.

CTree A counter at the root of an optimal width combining tree using the protocol of Goodman et al. [21] as modified by Herlihy, Lim, and Shavit [24]. A combining tree is a distributed data structure with the layout of a binary tree. Optimal width means that when n processors participate in the simulation, a tree of width $n/2$ is used [24]. Every node of the tree (including the leaves) contains a spin-lock, and the root contains a local counter. Each pair of processors is accorded a leaf. In order to reach the counter at the root, a processor's request to increment the counter must ascend the tree from a leaf. To this end a process attempts to ascend the tree, acquiring the locks in the nodes on its path. If a lock is currently held by another processor or processors, it waits until the lock is freed. If two processors reach the same node and try to acquire the lock at approximately the same time, they combine their increment requests, and only one of them continues to ascend the tree with the combined requests. This eliminates the need for all processors to actually reach the root counter. When a processor acquires the root it increments the counter by the sum of all combined increments, and then descends the tree, unlocking nodes along its path, and handing down results of the increment operation to the processors with which it combined.

CNet The BITONIC counting network of Aspnes, Herlihy, and Shavit [7] of width 64. A Bitonic counting network is a network of two-input-two-output balancers having a layout isomorphic to a Bitonic sorting network [8]. Each processor performing an increment operation travels

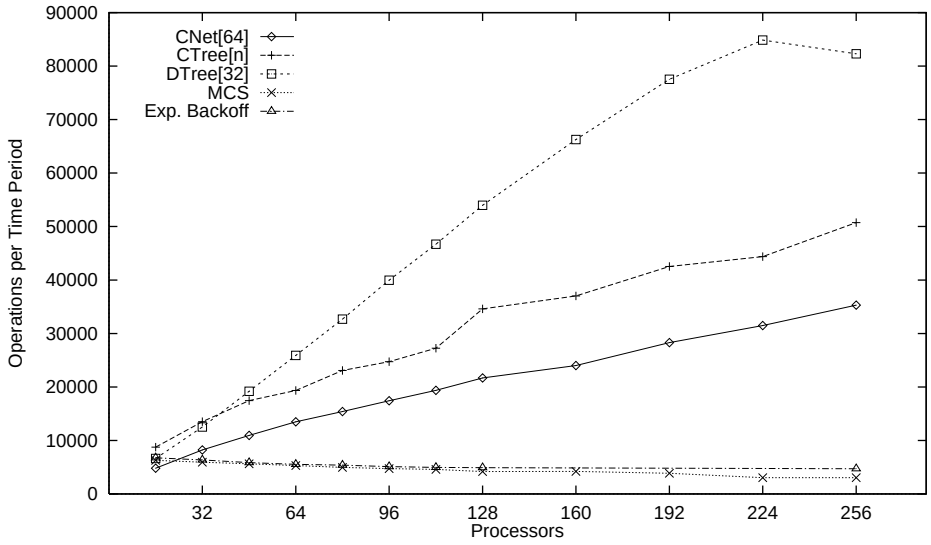

Figure 8: Throughput of various counting methods when `work = 0`

through the network from input wires to output wires toggling the shared bits in the balancers along its path. The code in Figure 2 with the assignment of the root balancer (Line 3 of `fetch&incr`) replaced by the selection of a random input wire to a BINARY[64] amply describes the counting network protocol. The wires/pointers from one balancer to another are cached locally by processors, while the toggle bit in shared memory is protected by a spin-lock with exponential backoff [6, 23]. Each output wire ends in a local counter implemented using a short critical section protected by a test-and-test-and-set lock with exponential backoff [6, 23]. The counting network width of 64 was chosen based on preliminary testing that showed it provides the best throughput/average latency over a range of up to 256 processors. We note that Felten, LaMarca, and Ladner [18] show network designs using higher fan-in/out balancers which can get up to a 25% performance improvement over the Bitonic network.

DTree A Diffracting Tree of width 32.

The graphs in Figures 8 and 9 show the throughput and latency of the various counting methods. Our performance graphs for the known methods other than Diffracting trees conform with previous findings and in particular, agree with the results of Herlihy, Lim and Shavit [24] on *ASIM* [3], the *Alewife* machine hardware simulator[3].

It is clear from these graphs that the MCS lock and the lock with exponential backoff do not

[3]To confirm our findings we reproduced their experiments with Proteus and got nearly identical results. Since the rest of our study uses a 256 processor machine in contrast to their 64, those results are not given here.

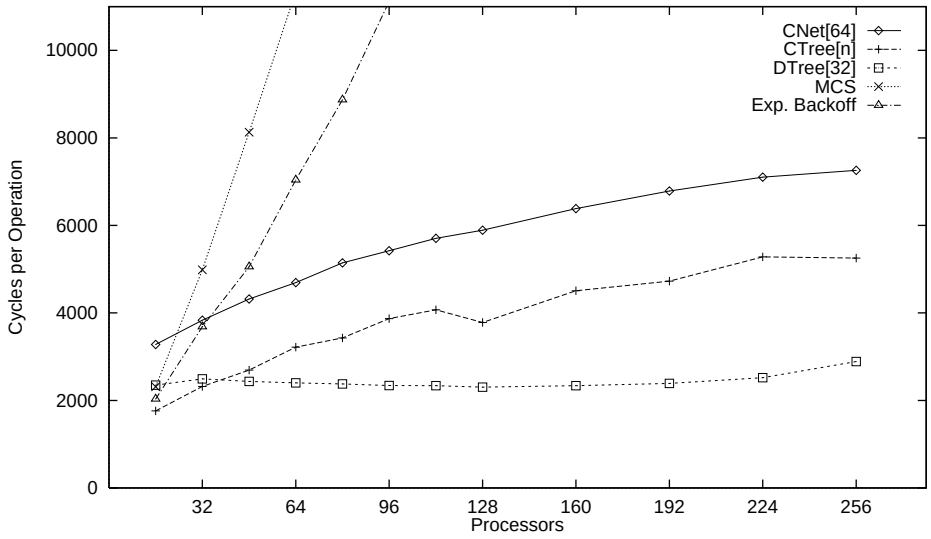

Figure 9: Latency of various counting methods when `work = 0`

scale well: latency grows quickly, and throughput diminishes. This is not surprising, since both are methods for eliminating contention but do not support parallelism. Our results for the MCS lock differ from those of Mellor-Crummey and Scott [40] due to differences in machine architecture. In their BBN Butterfly experiments if two read-modify-write operations are performed on the same memory location (such as register-to-memory-swap on the lock's tail pointer) one will succeed immediately and the other is blocked and retried later. In the cache coherence protocol used by Proteus this results in cache livelocks: both are aborted and retried, possibly several times, explaining the sharp rise in latency seen in Figure 9.

The remainder of the discussion concentrates on the latency and throughput results of the three parallel techniques: combining trees, bitonic counting networks and diffracting trees. The graphs in Figure 8 show that diffracting trees give consistently better throughput than the other methods. In terms of latency Figures 9 and 10 show that they scale extremely well: average latency is unaffected by the level of concurrency.

While processors that failed to combine in a combining tree must waste cycles waiting for earlier processors to ascend the tree, processors in a diffracting tree proceed in an almost uninterrupted manner due to the high rate of collisions in the `prism` array. To estimate the number of useful collisions (those leading to a diffraction) in the prism array, we define the term *diffraction rate* of a balancer, to be the ratio between the number of tokens leaving the balancer by diffraction, to the number of tokens leaving the balancer via the toggle bit. Consider some balancer, b after a sufficiently long run of the algorithm. Suppose l tokens have passed through b, of those, d where

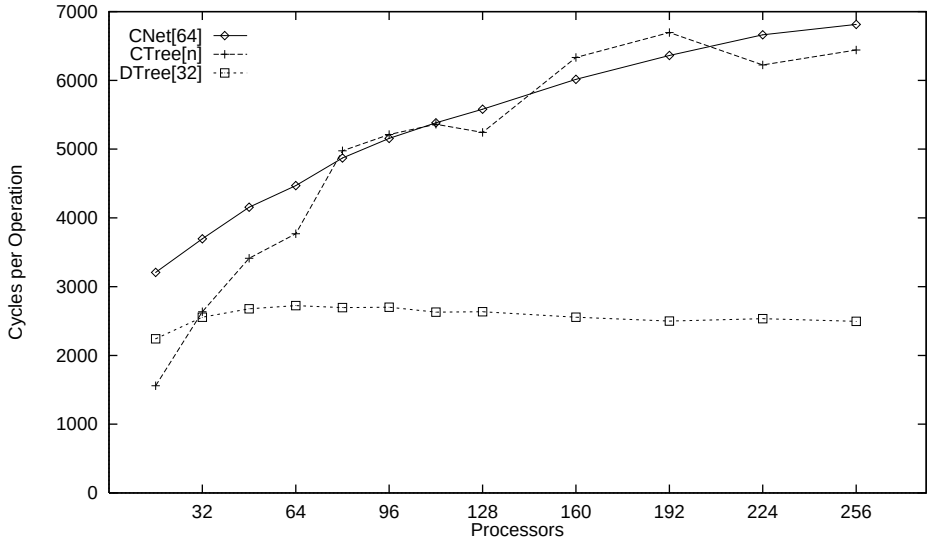

Figure 10: Latency of distributed parallel counting methods when `work = 1000`

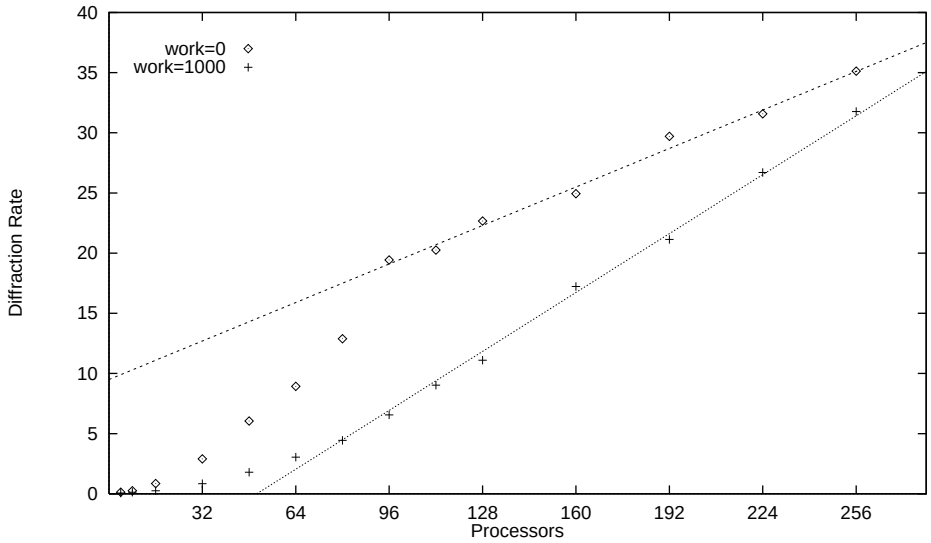

Figure 11: Diffraction Rate

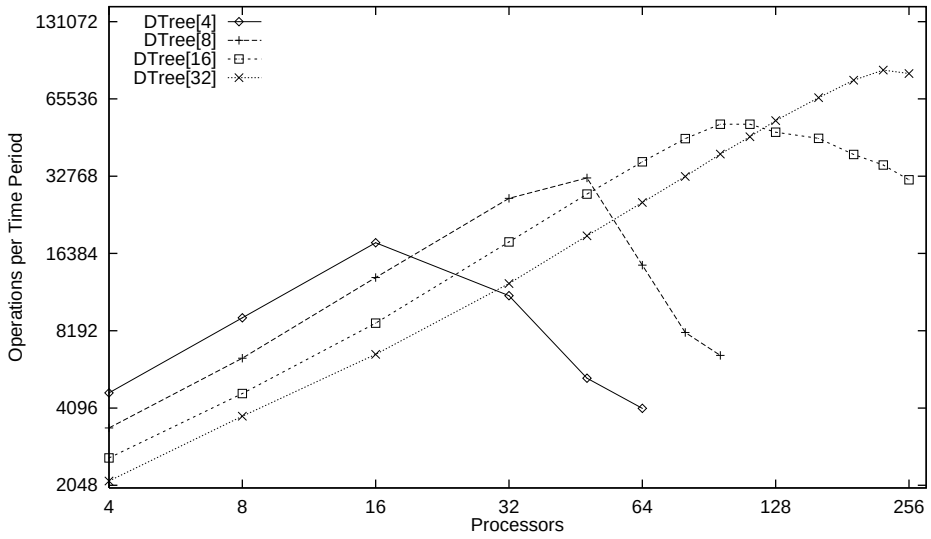

Figure 12: Effects of diffracting tree height on throughput, when `work = 0`

diffracted and $t = l - d$ went through the toggle bit. We define τ, the diffraction rate as: $\tau = d/t$. Figure 11 shows the diffraction rate at the root balancer as a function of the number of processors in the simulation. The graph indicates a linear relationship of the form $\tau \approx an + c$ exists, where n is the number of processors and a and c are some constants. Remembering that $\tau = d/t$, and $d = l - t$, we get $t \approx \frac{l}{an+c+1}$. Let us consider now a short interval of time Δ, during which Δl tokens enter the balancer, $\Delta l \leq n$ since only n tokens can exists simultaneously. If n is large enough, we get $\Delta t \leq \frac{n}{an+c+1} \leq \frac{1}{a}$, where Δt is the number of tokens passing through the toggle bit during Δ. This means that the contention on the toggle bit is bounded by the constant $\frac{1}{a}$ – the number of accesses during Δ. Thus, the level of contention on the toggle bit remains constant as concurrency increases, and in fact, our measurements show that $\frac{1}{a} < 10$ for the root balancer when `work` is 0.

The scalable throughput of diffracting trees is to a large extent a result of their ability to withstand high loads with low contention as explained above, coupled with their low depth. To see why this is so consider the following "back of the envelope" calculation. Optimal depth combining trees [24] have a depth of $\log n/2$ where n is the number of processes. With n of about 256 and assuming time t_{ctree} to traverse/combine in a node, it takes $2(t_{ctree} \log n/2) = 14t_{ctree}$ time to get 256 indices back so its throughput is $18/t_{ctree}$.

A counting network with w counters at its output wires has a fixed depth of $(\log w)(1+\log w)/2$. Unlike the combining tree, tokens traversing the counting network are pipelined in the structure, so as long as there are sufficiently many processors concurrently accessing the network, w indices are returned every t_{cnet} time where t_{cnet} is the balancer traversal time. One would hope this means

that a network of width $w = 16$ could deliver top throughput performance of $16/t_{cnet}$, for, say $1/2w(\log w)(1 + \log w) = 160$ processors. Unfortunately, as empirical testing shows [7, 24], if the counting network is loaded to that extent, t_{cnet} for each balancer tends to degrade (grow) rapidly due to contention and sequential bottlenecking. This explains our preliminary tests that found the counting network with best performance for the range of 256 processors has width 64 and depth 21. Unfortunately, it thus has rather limited pipelining and delivers substantially less than 64 indices every t_{cnet} time. If one assumes an even distribution of processors per level of the counting network, then there could be no more than $256/21$ processors at the counters at any time, giving an average throughput of $256/21t_{cnet} = 12/t_{cnet}$. The experimentally measured throughput for the counting network is accordingly slightly less than that of a combining tree. (One must keep in mind that this is a very crude estimate as the ratio of t_{cnet} to t_{ctree} is a factor in the comparison which is hard to determine.)

A diffracting tree, like a counting network, allows pipelining of requests, has depth $\log w$, and outputs w indices every t_{dtree} time, where t_{dtree} is the time to traverse a diffracting balancer. Though most likely $t_{dtree} > t_{cnet}$, the diffracting balancer, as we explained above, is not susceptible to contention and does not introduce a sequential bottleneck. Thus, loading the tree structure will not significantly increase t_{dtree}. The empirically observed consequence is that a width $w = 32$ and depth $\log w = 5$ diffracting tree can sustain concurrent access by at least 224 processors without a drop in throughput.

Under the reasonable assumption that t_{dtree} for a diffracting balancer is no higher than t_{ctree} for a combining tree node, and given that it is less susceptible to contention and to fluctuations in access times, it becomes clear that the diffracting tree's throughput of $32/t_{dtree}$ is substantially higher than the $18/t_{ctree}$ of the combining tree, as confirmed by the empirical results. Moreover, the diffracting tree's traversal time of $5t_{dtree}$ is much shorter than $14t_{ctree}$ for the combining tree and $21t_{cnet}$ for the counting network, which explains its significantly smaller observed average latency. (This should again be taken with a grain of salt since the ratio t_{cnet} to t_{dtree} is hard to estimate).

For the remainder of the paper we will present either latency or throughput results, but not both, since one can deduce latency from throughput and vice-versa. The reason for this is as follows. Let L be the average latency of a counting method during an interval of t cycles. Each processor can perform t/L fetch-and-increment operations. If n processors are active, we get $T = nt/L$ total operations performed, T is therefore the throughput of the system. Figures 8 and 9 show that whenever there is a significant change in one measure, there is a corresponding change in the other.

Figure 10 shows how latency scales for `work = 1000`. As can be seen, the average latency of the diffracting trees is unaffected by the large variance in increment request arrival times indicating a method that is scalable to both large numbers of processors and different work loads. Scalability of the counting network is likewise unaffected by arrival times, and as before latency increases with

```
Procedure prod-cons-bench(work: integer)
  loop:   w := random(0,work)          /* produce */
          i := increment(NQcounter)
          enqueue w at Queue[i mod n]
          i := increment(DQcounter)
          dequeue w from Queue[i mod n]
          delay(w)                     /* consume */
          goto loop
```

Figure 13: Code for producer/consumer benchmark

concurrency. The combining tree is severely affected by fluctuations in arrival times (see also [24]) and scales poorly.

As seen in Figure 8 the diffracting tree shows a drop in performance when the number of processors goes from 224 to 256. This suggests the need to increase the size of the tree if more processors are to be used. Figure 12 shows the relationship between diffracting tree size and performance. Choosing a tree that is too narrow or too wide can have negative effects. However, since the interval in which a given width is optimal increases with tree size, the wider tree can usually be used without fear. Also, the application of an adaptive scheme for changing diffracting tree size "on the fly" (see for example [37]) will most likely not result in frequent changes among different width trees.

In summary, diffracting trees scale substantially better than the other methods tested as they have small depth and enjoy both the parallelism of counting networks and the beneficial utilization of "collisions" of combining trees.

3.3.2 Producer/Consumer Benchmark

This benchmark simulates a producer/consumer buffer used as a job queue where processors take turns serving as producers and consumers. Each processor produces a job and enqueues it, dequeues a job and performs it, and so on, until N jobs have been performed. The job queue is implemented using an array of n elements, each of which can hold a single job. Processors increment shared counters to choose locations for queue operations. A dequeue operation on the i-th array element will block until some job has been enqueued there. Similarly, enqueues block if the location is full. Since array locations are independent, queue operations can proceed in parallel. The code for this benchmark is given in Figure 13.

To compare the performance of different counters, we measured the time to perform 2000 queue operations. Results appear in Figure 14 for `work = 100` and Figure 15 for `work = 1000`.

The graphs clearly show that a diffracting tree outperforms the other methods by a factor of approximately two when `work` is low, and is about 50% faster when `work` is high. Notice that we used a smaller diffracting tree and counting network here (widths 16 and 32, instead of 32 and 64, respectively) to take advantage of the smaller load, something that can't be done with combining trees. Unlike the index distribution benchmark, here the counting network wins over the combining tree because processor arrival times may be quite far apart, making combining more difficult. Section 4.3 discusses this issue in further detail.

4 Message Passing

The following section describes a realization of diffracting trees in a message passing environment. We studied the performance of the algorithm and compared it to the other parallel methods, in the context of four message passing topologies that differ in their interconnect bandwidth and their utilization of network locality.

4.1 Implementation

The implementation (see Figure 16) is fairly straightforward: instead of the prism array locations and toggle bit, a balancer will consist of a collection of *prism processors* and a *toggle processor*. Shepherding a token through a balancer is accomplished by sending a message to one of the balancer's prism processors (chosen uniformly at random). This processor delays the message for a fixed number of cycles (maintained in the balancer's `spin` field), to allow another token (message) to arrive. If another token arrives, the processor diffracts the two tokens, sending one in a message to the left balancer and the other in a message to the right. If another token did not arrive during this interval, the processor forwards the token to the balancer's toggle processor who decides whether to send it to the left or right balancer based on its internal toggle bit. Counters are implemented using processors that keep an internal counter, increment it when a message arrives, and send the resulting index to the processor who originally requested it. Notice that some processors play two roles (implemented using separate threads): generating requests for indices and participating in the implementation of the data structures.

Figure 17 presents the code and data structure for the message passing implementation. The balancer data type is very similar to the one used for the shared memory version, the `size`, `spin`, and `next` fields are exactly the same, the `toggle` field and `prism` array have been changed from integers to thread IDs (TIDs), as was explained previously. We assume that each thread has a pointer to the balancer it is implementing in its `mybalancer` variable. The following routines are used in the code:

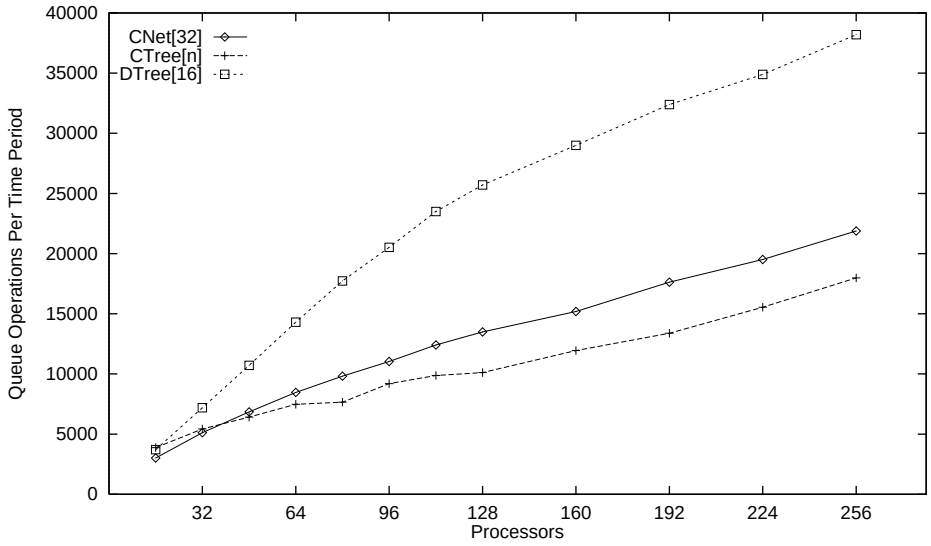

Figure 14: Producer/Consumer benchmark throughput results for `work = 100`

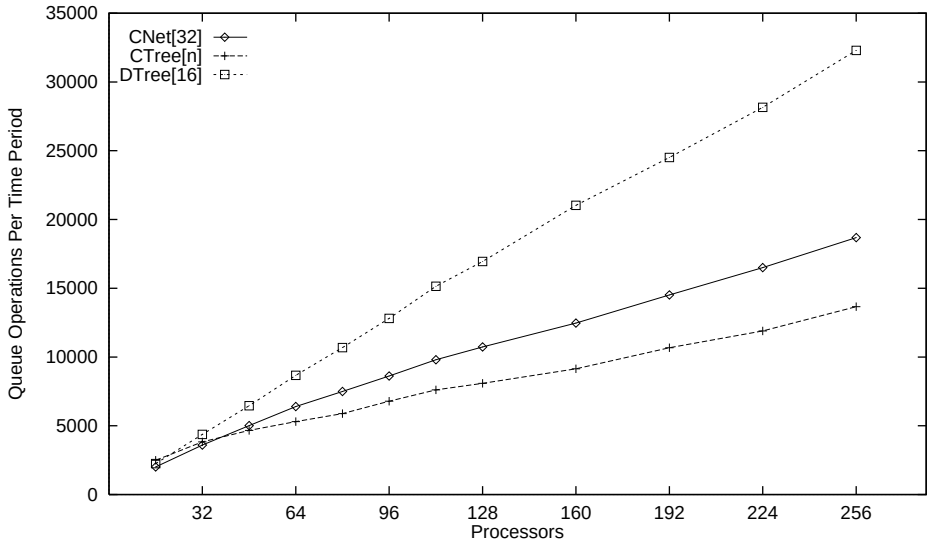

Figure 15: Producer/Consumer benchmark throughput results for `work = 1000`

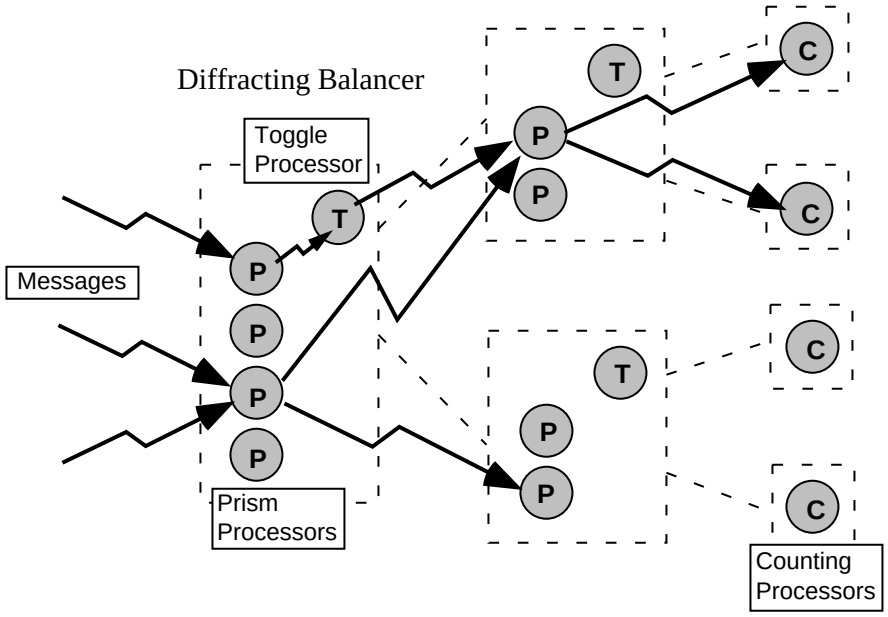

Figure 16: The message passing implementation of a diffracting tree

`dispatch_message(b,m)` Sends the message `m` to a randomly chosen prism processor of balancer `b`.

`receive_message(t)` Waits for a message to arrive, then returns that message. If no message arrives after `t` cycles `NULL` is returned. If `t` has the special value `BLOCK`, the routine waits indefinitely.

`send_message(t,m)` Sends the message `m` to the thread `t`.

`random(a,b)` Returns a random integer in the interval [a,b].

Section 5.3 contains the correctness proof for this implementation.

4.2 Measuring Performance

It has been shown [24] that for the cache-coherent Alewife architecture on which our shared memory counting methods were tested, message passing implementations are significantly faster than shared memory ones. Rather than duplicate those efforts here we choose to focus on performance issues that are common to most message passing systems, ignoring the specific hardware support that might be available in a multiprocessor.

```
   type balancer is
   begin
       size: integer
       spin: integer                          /* How long to wait      */
       prism: array[1..size] of TID           /* prism processors      */
       toggle: TID                            /* toggle processor      */
       next: array[0..1] of ptr to balancer   /* left & right          */
   end

 1 dispatch_message(b: ptr to balancer, m: message)
 2 begin
 3       send_message(b->prism[ random(1,b->size) ],m)
 4 end

 1 CountingProc                         /* code for counting processor */
 2 begin
 3     Vars
 4        Counter : integer
 5     Code
 6        message := receive_message(BLOCK)
 7        send_message(message.source , Counter)
 8        Counter := Counter + 1
 9 end

 1 ToggleProc                            /* code for counting processor */
 2 begin
 3     Vars
 4        ToggleBit: integer
 5     Code
 6        message  := receive_message(BLOCK)
 7        dispatch_message(mybalancer->next[ToggleBit],message)
 8        ToggleBit := not(ToggleBit)
 9 end

 1 PrismProc                            /* code for counting processor */
 2 begin
 3     Code
 4        message0 := receive_message(BLOCK)
          /* wait "spin" cycles for another message to diffract */
 5        message1 := receive_message(mybalancer->spin)
 6        if (message1 <> NULL) then           /* perform diffraction */
 7           dispatch_message(mybalancer->next[0],message0)
 8           dispatch_message(mybalancer->next[1],message1)
 9        else                                 /* send to toggle      */
10           send_message(mybalancer->toggle,message0)
11        endif
12 end
```

Figure 17: Code for message passing diffracting balancer

We tested the performance of the message passing diffracting trees in simulated network environments using Netsim [29]. Netsim is a generic network simulator, developed as part of the Rice Parallel Processing Testbed [15]. The simulation is *event driven*, implying that time progresses from event to event, operations performed between events, which do not interact with the simulated network, take no time. Between a `receive_message()` and a subsequent `send_message()` a process can perform any amount of computation with no performance penalty and no time passing. However, it takes time for a message to travel through the network, and arrive at its destination. Some of the factors which effect this time are the following: the network architecture, the number and size of messages sent, the distance messages must travel to their destinations, and the congestion at network nodes and switches. This type of modeling reflects current trends in computer architecture, where network speeds dominate scalability since they do not improve as fast as processor speeds [28].

Our experiments included four types of networks: a torus mesh network with single wire switches, a torus mesh network with crossbar switches, a butterfly network with crossbar switches, and an $n \times n$ crossbar network. The choice of networks allows the study of two important performance parameters that govern the behavior of highly distributed communication intense control structures: locality and bandwidth. As presented in Table 1 the four types of networks tested cover the various combinations of these two parameters.

Each network is made up of processors, wires and switches. Messages are sent by processors, along wires and are routed by switches along their path to their destination. A wire can accommodate one message at a time, switches may be able to handle more, depending on their construction. Messages arriving at a switch or wire that is busy servicing previous requests, wait at buffers till the network is ready to service them.

	Low Locality	High Locality
Low Bandwidth	Butterfly Network	Mesh with single wire switches
High Bandwidth	$n \times n$ Crossbar	Mesh with Crossbar switches

Table 1: A Comparison of Network Topologies

Torus mesh network with single wire switches This network has a two dimensional mesh topology. Network switches are placed on the grid points of a two dimensional $\sqrt{n} \times \sqrt{n}$ mesh, and each switch interfaces with five components: the four switches around it and the processor local to its grid point. An interface between components uses two wires, one incoming and one outgoing. The switches at the edge of the grid are connected "around the back" to form a torus. The routing used is a simple, shortest path, X coordinate first

algorithm. The switches can support only one message at a time, as can the wires between switches. The diameter of this network is $O(\sqrt{n})$, where n is the number of processors.

Torus mesh network with crossbar switches Except for the construction of the switches, this is exactly the same as the previous network. Here we use 5×5 crossbar switches, this means that a number of messages can pass through a switch at the same time, provided each has a different source, and a different destination. At most 5 messages can pass through such a switch simultaneously.

Butterfly network In this architecture (sometimes called a multi-layer network), processors form the bottom layer of an arrangement of switches, $\log n$ layers deep. Messages are sent from the processors, to the first layer of switches, which forwards them to the next layer, and so on, until $\log n$ layers are passed through. The last layer is connected "around the back" to the processor layer, completing the cycle, and delivering messages to their destination. Each switch is connected to four other switches, two on the layer below it, and two on the layer above. The switches are 2×2 crossbars, allowing two messages with different sources and destinations to pass through at the same time. This network has a diameter of $O(\log n)$.

$n \times n$ **crossbar network** A crossbar network is a switch which provides a dedicated communications channel between any two pairs of processors, giving an $O(1)$ diameter. The switch has n input wires, and n output wires, each pair of which is connected to a processor. It can simultaneously route messages that don't share the same input or output wire, handling at most n concurrent messages.

We ran the index distribution benchmark on each architecture, each time measuring the following:

Latency The average number of cycles to pass between the time a processor sends a message requesting a number, and the arrival time of the message carrying the requested index.

Throughput The number of indices the system can hand out in T cycles. This is calculated using the formula Td/t, where t is the time took the system to hand out d indices. Since Netsim, unlike Proteus, is an event based simulator, there is no need to take in to account start-up times – all processors are ready at the same time.

Since it has already been shown [24] that centralized counting methods do not scale well, we compare only the three distributed-parallel counting methods:

CNet$[w]$ A message based BITONIC counting network, implemented in the obvious way: balancers are threads of control and tokens are messages. For the width of the network w, we tested the

following values: 8, 16 and 32. In each benchmark, results are presented for the best width network.

CTree An optimal depth combining tree. Combining trees are described in Section 3.3.1 The message passing version was implemented by mapping the tree's nodes to threads in the multiprocessor. Each node, upon receiving a message requesting an index, holds that message until combining can be performed; this behavior is optimal as explained below.

DTree[w] A diffracting tree of width $w \in \{4, 8, 16, 32\}$. In each benchmark, results are presented for the best width tree.

In our experiments we varied the number of threads requesting indices. This is the value plotted along the X-axis. In counting networks and diffracting trees, the number of threads implementing the data structure is independent of the number of threads requesting indices, so the size of these structures was kept constant throughout each experiment (graph). For optimal depth combining trees a new level must be added whenever the number of threads requesting indices doubles, so the data structure itself grows during the experiment. In all experiments the number of threads per processor was at most two: one to implement the data structure, and one to request indices. Here, as with the experiments in shared memory, every attempt has been made to optimize the data structures, as we further detail below.

4.3 Results

Overall, combining trees proved to be the most efficient counting method in mesh topologies with low bandwidth switching where locality is a primary performance factor, while diffracting trees proved the most efficient method in "non-localized" butterfly style networks where locality is not a factor. We now discuss these conclusions in detail.

4.3.1 Choosing a Waiting Policy

Nodes of a combining tree or `prism` processors in a diffracting tree delay arriving messages to create a time interval in which combining or diffraction can occur. Figure 18 compares combining tree latency when `work` is high using 3 waiting policies: wait 16 cycles, wait 256 cycles, and wait indefinitely. When the number of processors is larger than 64, indefinite waiting is by far the best policy. This follows since an un-combined token message locks later received token messages from progressing until it returns from traversing the root, so a large performance penalty is paid for each un-combined message. Because the chances of combining are good at higher arrival rates we found that when `work = 0`, simulation using more than four processors justify indefinite waiting. We used this policy for all combining trees.

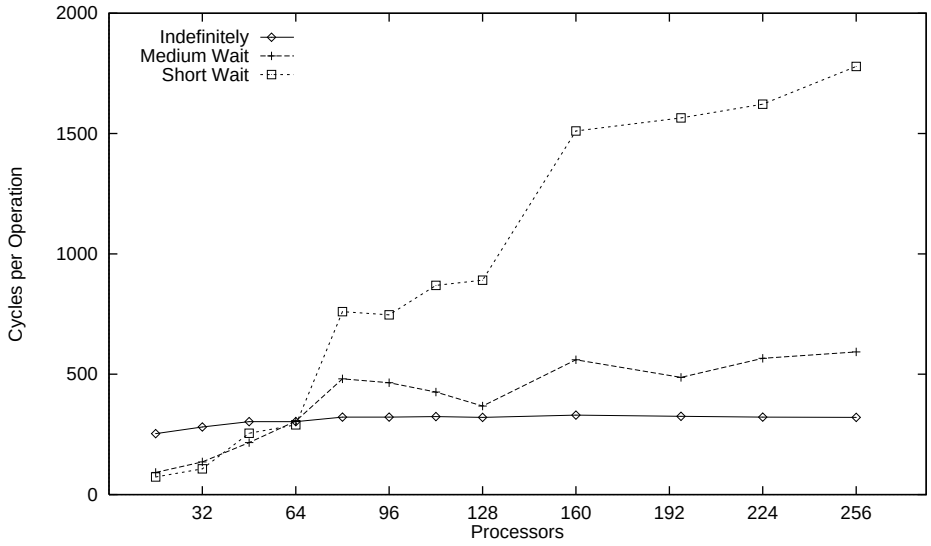

Figure 18: Effects of waiting time policy on combining tree latency in a mesh network with single wire switches when `work = 500`

In diffracting trees, high loads favor waiting. However, when arrival rates are low, as in the case when `work` is high or the number of processors in the simulation is small, prism processors should expedite the sending of messages to the toggle processor to reduce latency. As in the shared memory implementation, the best diffracting tree performance was attained when using an adaptive policy to update token delay time as a function of concurrency. The tree is initialized with a list of values for the `spin` variable. Whenever a thread acting as a prism processor diffracts a message, it doubles its `spin` time since this indicates a high load. If time runs out before diffraction occurs, usually as a result of low load, the `spin` time is halved.

4.3.2 Robustness

For our purposes, an algorithm is considered robust if its performs well under a wide variety of conditions, such as different work loads, or a large variance in request arrival times. Combining trees proved to be the least robust of the counting methods we studied and diffracting trees the most robust. We first analyze robustness in the face of load fluctuations. For combining trees, in all the network architectures we tested, as the range of `work` between counter accesses grew, variations in the arrival rates of requests made combining more difficult, and performance degraded. This conforms with the observations of Herlihy, Lim, and Shavit [24] that combining trees perform poorly (lower percentages of requests are combined) when the load drops. A dramatic example of this can be seen in the tests on the torus mesh network with single wire switches (the same network used for Figure 18) under different workloads (Figures 19 through 20). The need to wait for latecoming

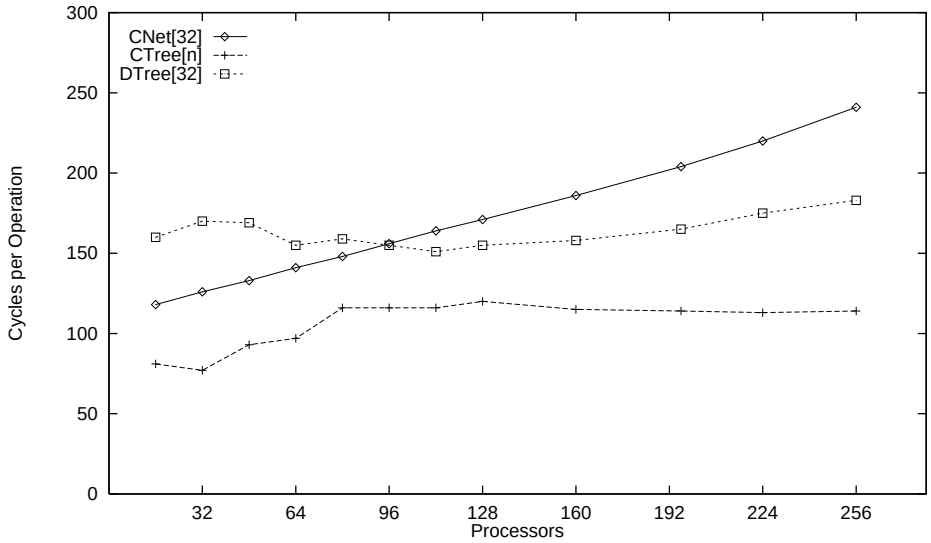

Figure 19: Latency in mesh network with single wire switches, when `work = 0`

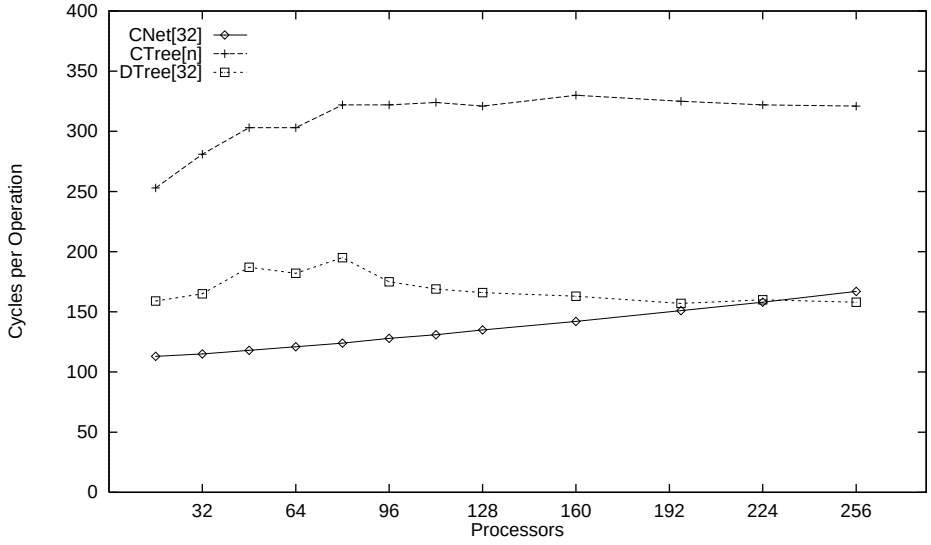

Figure 20: Latency in mesh network with single wire switches, when `work = 500`

processors causes a significant rise in latency which in turn lowers throughput.

Fluctuations in request arrival times have a lesser effect on diffracting trees and counting networks. Comparing Figures 19 and 20 shows that for counting networks lower load leads to less contention, latency still rises as concurrency increases, albeit more slowly. In diffracting trees there

is less diffraction in low load situations, but there is also very little congestion on the toggle bit. In addition diffracting balancers are adaptive, dynamically reducing waiting times at prism processors. When the load is very low, waiting time is reduced to 0 and prism processors immediately forward messages to the toggle processor. In a sense this transforms the diffracting balancer into regular balancer that takes two messages to traverse. This claim is justified by Figure 23 which shows that when the number of processors is small (16), the latency of a width 32 counting network (17 messages to traverse) is about 48 cycles, whereas that a width 16 diffracting tree (10 messages to traverse) is 33 cycles, the ratio of latencies is very close to 17:10. A slight increase in concurrency leads to congestion at the toggle bit, causing a rise in latency, then after a further increase diffractions begin to occur and latency falls again. This gives diffracting trees the characteristic latency curve which appears in all the architectures we tested.

We now consider robustness as load increases. In a counting network, when the load is high there is congestion at the balancers, causing a rise in latency and a lowering of throughput (Figures 19, 21, 22 and 23). On the other hand, combining and diffracting trees make use of the high arrival rate to combine/diffract messages, utilizing the added congestion to increase parallelism (combining requests or avoiding the shared toggle processor). Combining trees handle concurrency by increasing depth, which adds latency with each new level (e.g. Figures 22 and 23). Diffracting trees are more scalable: a single diffracting tree can often handle a wide range of concurrency levels with little or no performance penalty.

4.3.3 Performance: The Effects of Locality and Bandwidth

Combining tree layout can be optimized to take advantage of network locality. The tree thus sends relatively few messages per index delivered which is important if bandwidth is low. For these reasons combining outperforms all other methods in the mesh network with single wire switches (Figure 19). While a counting network's layout can also be optimized (though to a lesser extent than a combining tree), the dynamic flow patterns of diffracting trees make layout optimization much less effective. In our experiments we used the simulated annealing algorithm [30] to attempt to minimize the average distance traveled per message for each data structure. Figure 24 compares the performance of combining and diffracting trees, with and without layout optimization, i.e. once according to the results of the annealing process, and once when threads are randomly distributed among processors in the network. The results show that combining trees are less robust — placing them randomly on the mesh causes a drop of nearly 56% in throughput. Note also that in our experiments, when fewer than n processors participated in the simulation, they were selected in a bottom-up/left-to-right manner, ignoring the advantage that such a fixed distribution gives to localized methods like combining.

Higher bandwidth reduces the need to conserve messages or shorten distances as the added

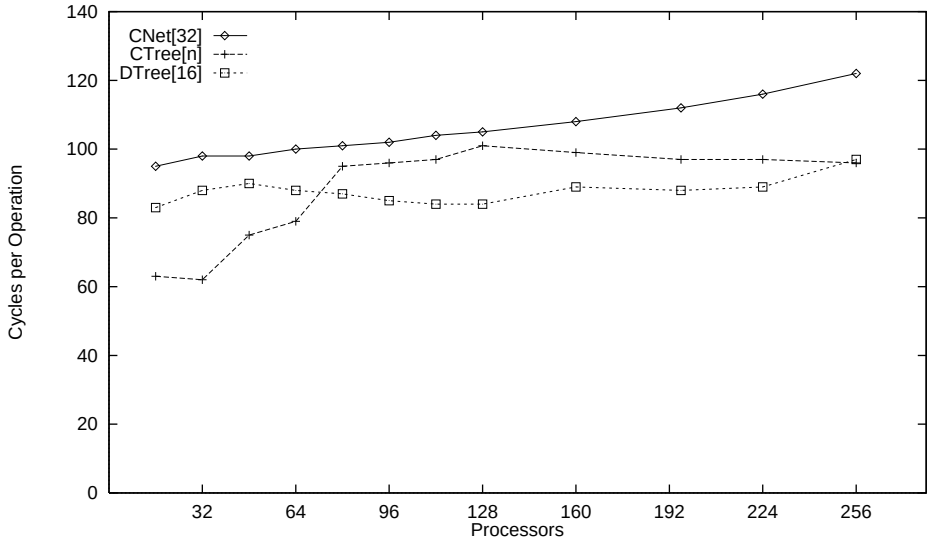

Figure 21: Latency in mesh network with crossbar switches, when `work = 0`

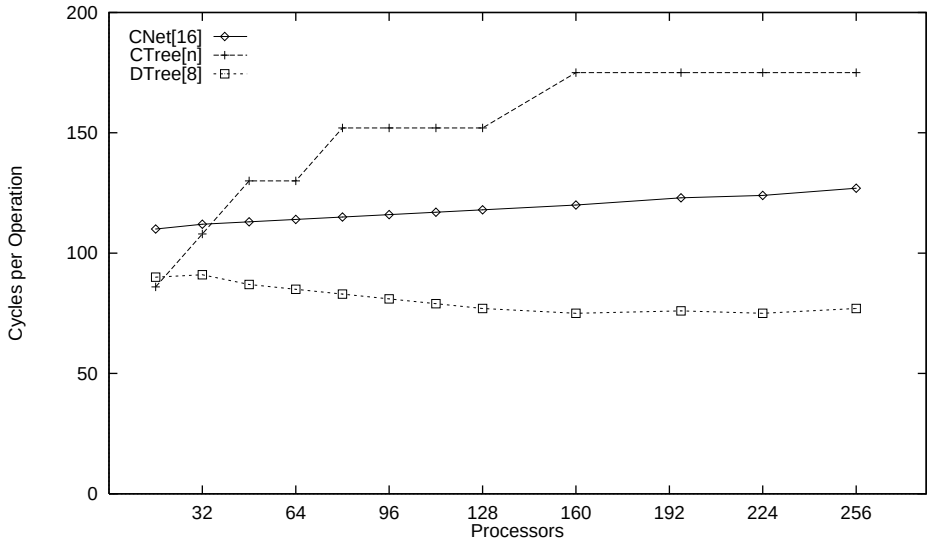

Figure 22: Latency in butterfly network when `work = 0`

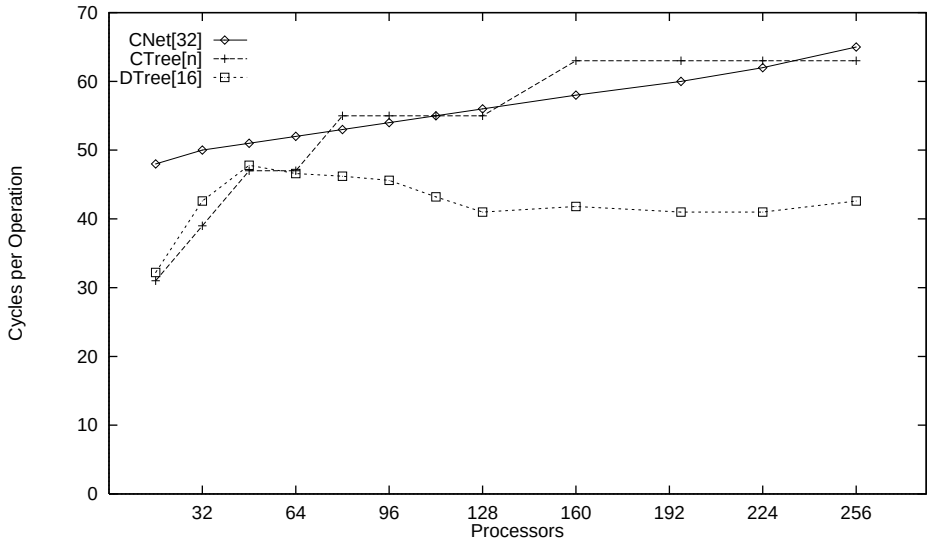

Figure 23: Latency in full crossbar network when `work = 0`

bandwidth helps hide the effects of locality. In the mesh with 5×5 crossbar switches diffracting trees reap the benefits of lower depth: they have increased throughput and lower latency (Figure 21) relative to other methods. Counting networks, like combining trees, gain less from locality, and given balancer contention and relatively high depth, they are the least desirable data structure.

In equidistant network topologies, data structure depth becomes the key performance issue. When bandwidth is low as in the butterfly network (Figure 22), cost per message is high and diffracting trees, having the lowest depth, substantially outperform the other methods. In the complete crossbar network (Figure 23), the added bandwidth reduces the cost of messages and all three methods have roughly similar performance, with the diffracting tree leading in throughput by about 35%.

The appropriate choice of width of a diffracting tree or counting network depends on the properties of the network being used. In equidistant, low bandwidth networks, where depth is the main concern, smaller trees and networks work better. On the other hand a larger data structure is better suited to take advantage of bandwidth, and also tends to spread messages around the entire network, which is useful when congestion is a problem, as in the case of the mesh with single wire switches. Table 2 summarizes the optimized widths of the constructions we present.

Diffracting Tree

	locality	
Bandwidth	Low	High
Low	8	32
High	16	16

Counting Network

	locality	
Bandwidth	Low	High
Low	16	32
High	32	32

Table 2: Width of diffracting tree and counting network per network type.

5 Correctness Proofs

5.1 A Proof that Counting Trees Count

This section contains a formal proof that a counting tree's outputs will achieve the desired step property in any quiescent state. Our formal model for multiprocessor computation follows [7, 36]. First a formal description of a balancer is given, then it is shown that any BINARY counting tree counts, that is, its outputs have the *step property*.

Let the state of a balancer at a given time be defined as the collection of tokens on its input and output wires [7]. 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. For the sake

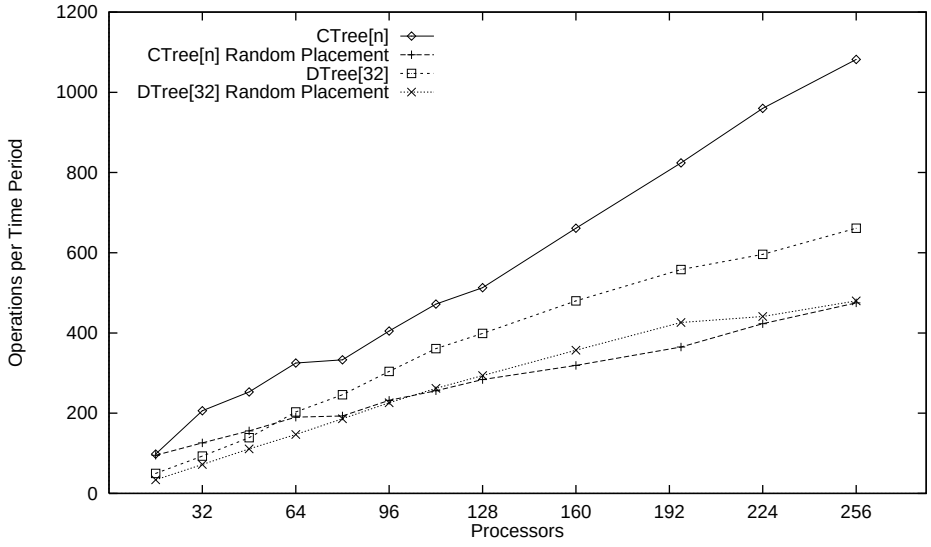

Figure 24: Effects of placement optimization on throughput in mesh network with single wire switches when `work = 0`

of clarity it is assumed that tokens are all distinct. The properties defining a balancer's correct behavior are:

safety In any state $x \geq y_0 + y_1$. (i.e. a balancer never creates output tokens).

liveness Given any finite number of input tokens $m = x$ to the balancer, it is guaranteed that within a finite amount of time, it will reach a *quiescent* state, that is, one in which the sets of input and output tokens are the same.

balancing In any quiescent state, $y_0 = \lceil m/2 \rceil$ and $y_1 = \lfloor m/2 \rfloor$.

As described earlier, a *counting tree* of width w is a binary tree of balancers, where output wires are connected to input wires, having one designated root input wire, x, (which is not connected to an output wire) and w designated output wires $y_0, y_1, .., y_{w-1}$ (similarly unconnected). Let the state of the tree at a given time be defined as the union of the states of all its component balancers. The safety and liveness of the tree follow naturally from the above tree definition and the properties of balancers, namely, that it is always the case that $x \geq \sum_{i=0}^{w-1} y_i$, and for any finite sequence of m input tokens, within finite time the tree reaches a *quiescent* state, i.e. one in which $\sum_{i=0}^{w-1} y_i = m$. It is important to note that we make no assumptions about the timing of token transitions from balancer to balancer in the tree — the tree's behavior is completely asynchronous.

We will show that if a BINARY[w] counting tree reaches a quiescent state, then its outputs, $y_0, \ldots, y_{w-1}$ have the following step property:

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

We present the following useful lemmas due to Aspnes, Herlihy, and Shavit [7].

Lemma 5.1 *If $y_0, \ldots, y_{w-1}$ is a sequence of non-negative integers, the following statements are equivalent:*

1. *For any $i < j$, $0 \leq y_i - y_j \leq 1$.*
2. *If $m = \sum_{i=0}^{w-1} y_i$, then $y_i = \lceil \frac{m-i}{w} \rceil$.*

Lemma 5.2 *Let $x_0, \ldots, x_{k-1}$ and $y_0, \ldots, y_{k-1}$ be arbitrary sequences having the step property. If*

$$\sum_{i=0}^{k-1} x_i = \sum_{i=0}^{k-1} y_i$$

then $x_i = y_i$ for all $0 \leq i < k$.

Lemma 5.3 *Let $x_0, \ldots, x_{k-1}$ and $y_0, \ldots, y_{k-1}$ be arbitrary sequences having the step property. If*

$$\sum_{i=0}^{k-1} x_i = \sum_{i=0}^{k-1} y_i + 1$$

then there exists a unique j, $0 \leq j < k$, such that $x_j = y_j + 1$, and $x_i = y_i$ for $i \neq j$, $0 \leq i < k$.

Using the above we can show that:

Lemma 5.4 *Let $x_0, x_1, \ldots, x_{k-1}$ and $y_0, y_1, \ldots, y_{k-1}$ be two arbitrary sequences having the step property. Then if*

$$0 \leq \sum_{i=0}^{k-1} y_i - \sum_{i=0}^{k-1} x_i \leq 1$$

then the sequence

$$y_0, x_0, y_1, x_1, \ldots, y_{k-1}, x_{k-1}$$

has the step property.

Proof: There are two cases:

1. $\sum_{i=0}^{k-1} y_i = \sum_{i=0}^{k-1} x_i$, in this case, by Lemma 5.2, both sequences are identical, and the proof is trivial.

2. $m = \sum_{i=0}^{k-1} y_i = \sum_{i=0}^{k-1} x_i + 1$, in this case, Lemma 5.3, applies, and the two sequences look like this:

$$\text{X sequence: } \overbrace{aa\ldots a}^{m \oplus k} bb\ldots bb$$

$$\text{Y sequence: } \overbrace{aa\ldots aa}^{m \oplus k+1} b\ldots bb$$

   Where $a = \lceil \frac{m}{w} \rceil$, $b = a - 1$, and $\oplus$ is the modulus operator. The interleaved sequence has the form:

$$\text{Joint sequence: } \overbrace{aa\ldots aa}^{2(m \oplus k)+1} bb\ldots bb$$

   which clearly has the step property.

■

Theorem 5.5 *The outputs of* BINARY$[w]$*, where w is a power of 2, have the step property in any quiescent state.*

Proof: Note that the proof need analyze the tree only in quiescent states, and need not account for concurrent executions. We thus examine the number of tokens that passed through each and every balancer in a diffracting tree once a quiescent state has been reached.

By induction, if $w = 2$ then the tree is a single balancer, and the step property follows by definition. Assume the theorem holds for all trees of width $w \leq k$, and let us prove that it holds for $w = 2k$. According to the construction given in section 2, the large tree of width $2k$, is actually one root balancer whose two outputs are connected to trees of width k. The even leaves of the the large tree are the leaves of the left small tree, and the odd leaves, are the leaves of the right small tree. Since the trees are connected by a balancer, we know that the number of tokens input to the left and right width k trees differ by at most one. By the induction hypothesis this means that they have the step property. By Lemma 5.4, the outputs of BINARY[$2k$] have the step property. ■

A shared counter is an object that allows increment operations that return consecutive integers. In any execution in which m increment operations are performed, all the values $0 \ldots m-1$ are returned, that is, each operation returns an index and there is no duplication or ommission. A counter can be constructed from a BINARY[w] tree by adding local counters to the tree's output wires.

We paraphrase a lemma [7] which equates counting with balancing.

Lemma 5.6 *Consider a* BINARY[w] *tree based shared counter as described above. Let x be the largest number returned by any increment operation on the counter. Let R be the set of of numbers less then x which have not been returned by any increment request. Then:*

1. *the size of R is not greater than the number of operations still in progress.*
2. *if $y \in R$, then $y \geq x - w|R|$.*

Theorem 5.7 *A* BINARY[w] *tree, where w is a power of 2, counts.*

5.2 A Proof of the Shared Memory Implementation

We will first show that the diffracting balancer implementation meets the balancer specification, then show that this is maintained for all balancers in a diffracting tree. The code line numbering in the proof refers to Figure 4.

Let us introduce some notation. A processor that has performed the *write* operation on line 3, is said to have *entered* the balancer. A token has *exited* the balancer, once the return value of its evocation of the `diff-bal` code is decided, *i.e.* no change in the state of any shared object

can cause it to exit on a different wire. Between entering and exiting this processor is said to be *shepherding* a token. Since a processor can shepherd only one token at a time (there is no multiplexing of tokens on a processor), each token is identified uniquely by the pair (p, T) where p is the ID of the shepherding processor and T is the time at which p entered the balancer.

Our proof is based on examining the different values taken by the entries of the `location` array during the lifetime of a balancer. The proof will make strong use of the fact that the `compare_and_swap`, `register_to_memory_swap`, and `test_and_set` operations are atomic, that is, can be assumed to take place at a unique point in time. We assume the machine's shared memory to be a collection of "memory locations," each of which follows the specification of of an atomic register [34]. The operations on each memory location (and therefore the values it takes) can be ordered chronologically, and atomicity assures us that this ordering is well defined. Thus one can draw a time-line of events for each of the memory locations making up the `location` array.

The following is an example of the reasoning used in the proofs. Let R be a memory location. Let processor p perform a *C&S* (compare-and-swap) operation of the form $C\&S_p(R, 1, 0) =$ `true`, where the notation "= `true`" indicates that the operation was successful. It follows that prior to this operation R had the value 1, and following it it has the value 0. If the next operation p performed on R is another successful $C\&S_p(R, 1, 0)$, it follows that between these operations the value of R was changed to 1, and we can surmise that some processor must have performed the operation that changed the value of R between p's *C&S* operations.

The next simple lemma will be proved using the following observation: The only time a token (p, T) changes the value of `location[`r`]` where $r \neq p$, is in the *C&S* operation of line 8.

Lemma 5.8 *Given processors $r \notin \{p, q\}$, if p performs the operation on line 8*

$$C\&S_p(\texttt{location}[r], b, \texttt{EMPTY}) = \texttt{true}$$

and, if the next successful C&S operation on `location`$[r]$ *by some processor q (other then r) is also:*

$$C\&S_q(\texttt{location}[r], b, \texttt{EMPTY}) = \texttt{true}$$

then a new token, shepherded by r, must have entered balancer b between these two C&S operations.

Proof: Following processor p's *C&S* operation, `location`$[r] =$ `EMPTY`. In order for q's *C&S* operation to succeed, some sequence of operations strictly between the two *C&S* operations must have ended in an operation that left `location`$[r] = b$. From the code, the only operations leaving `location`$[r]$ with value b are the $write_r($`location`$[r], b)$ (the write by processor r of the value b to `location`$[r]$) operations performed by processor r on lines 3 or 10. In case the last operation in the sequence was a *write* by r on line 3, then processor r has entered the balancer with a new token

between the *C&S* operations and the claim follows. Otherwise, the last operation was a *write* by r on line 10. Line 10 is reached by processor r by performing the following sequence of operations on `location`$[r]$:

(3) $write_r($`location`$[r], b)$,
(8) $C\&S_r($`location`$[r], b,$ `EMPTY`$) =$ `true`,
(10) $write_r($`location`$[r], b)$.

This implies that the write by r on line 3 preceded the $C\&S_q$ operation. We complete the proof by showing that it strictly followed p's $C\&S_p$ operation. The successful $C\&S_p($`location`$[r], b,$ `EMPTY`$)$ operation is by assumption before the write on line 10. It could not follow r's successful $C\&S_r$ on line 8 since that would cause it to fail, and it cannot be between r's operations on lines 3 and 8 since that would cause r's $C\&S_r$ on line 8 to fail. It follows that it must have preceded r's write operation on line 3. ■

Notice, that the previous lemma does not say how many tokens are shepherded by r between p and q's successful *C&S* operations, only that there is at least one such token. Let us now show that if a balancer is in a quiescent state, the number of tokens output on each wire is balanced to within one. Define three types of tokens: A token exiting the diffracting balancer code via the `return` at line 9 will be called a *canceling* token; one that leaves through a `return` on lines 11, 16 or 27 will be called a *canceled* token, and one that leaves through line 24 will be called a *toggling* token.

Canceling tokens leave the balancer through `b->next[0]` - the first output wire, canceled tokens leave through `b->next[1]` - the second output wire. Toggling tokens may leave through either wire.

Lemma 5.9 *In a quiescent state the toggling tokens have the step property.*

Proof: From the code, the toggle bit is protected by a critical section, and is initialized so that the first token to access it leaves through the first output wire: `b->next[0]`. ■

Showing that the number of canceled tokens is equal to the number of canceling tokens, will prove that the number of tokens added on each output wire is the same, and the balancing property is maintained.

Lemma 5.10 *If* `location[r]`$= b$ *then processor r is shepherding a token (r, T) and currently executing the code of balancer b.*

Proof: Initially the `location` array is set to `EMPTY`. The first operation performed by the process shepherding token (r, T) is setting `location[r]` to b, so this property holds. Examining all the

`return` points from the balancer code, shows that each is preceded by an operation that either sets the value of `location[`r`]` to `EMPTY` or tests that it is `EMPTY` already. Since no other token can cause a write b to `location[`r`]` until (r, T) has exited b, the process shepherding (r, T) must still be executing the code of balancer b. ■

Corollary 5.11 *In a quiescent state, all elements of the* `location` *array are* `EMPTY`*.*

Lemma 5.12 *After a successful* $C\&S_q(\texttt{location}[r], b, \texttt{EMPTY})$ *operation, where* $q \neq r$*, and until the next such operation by* $p \neq r$ *(if there is one), exactly one canceled token shepherded by* r *leaves balancer* b*.*

Proof: Let $t = (r, T)$ be the last of r's tokens to enter b before the $C\&S_q(\texttt{location}[r], b, \texttt{EMPTY})$ operation. Lemmas 5.8 and 5.10 together prove that t exists and entered b after any previous successful $C\&S_s$ operation ($s \neq r$), if there was one. We will now show that t is a canceled token. Assume otherwise, this means that t exited the balancer either on line 9 or on line 24 The operations performed before the return on line 9 are

(1) $write_r(\texttt{location}[r], b)$
(6) $C\&S_r(\texttt{location}[r], b, \texttt{EMPTY}) = \texttt{true}$
(9) return

Those before the return on line 24 are one of either

(1) $write_r(\texttt{location}[r], b)$
(6) $C\&S_r(\texttt{location}[r], b, \texttt{EMPTY}) = \texttt{true}$
(8) $write_r(\texttt{location}[r], b)$
(17) $C\&S_r(\texttt{location}[r], b, \texttt{EMPTY}) = \texttt{true}$
(24) return

or,

(1) $write_r(\texttt{location}[r], b)$
(17) $C\&S_r(\texttt{location}[r], b, \texttt{EMPTY}) = \texttt{true}$
(24) return

Both lines 9 and 24 can be reached by r only after performing a successful $C\&S$ operation on `location[r]`, changing its value from b to `EMPTY`. The write on line 3 occurs by assumption before q's $C\&S$ operation. Using the same reasoning as in the proof of lemma 5.8 it can been seen that q's operation could not have occurred successfully anywhere between this *write* and the subsequent `return` without causing one of r's $C\&S$ operations to fail. Furthermore, q's operation could not have occurred just after the `return` since that would cause it to fail. Thus token t could not have left through either of line 9 or line 24, and must be a canceled token. Let p be the processor to perform the next $C\&S_p(\texttt{location}[r], b, \texttt{EMPTY}) = \texttt{true}$ operation. It remains to be shown that for

each token $t' = (r, T')$ such that t' entered b after t and left before p's *C&S* operation, t' is not a canceled token. Keep in mind that while t' is passing through b it is the only token to change the value of `location[r]`. Examination of the code shows that if no other token changes the value of `location[r]` during the execution, t' will leave the code either on line 9 or line 24. ■

Lemma 5.13 *Any token (r, T) entering balancer b after a $C\&S_p(\texttt{location}[r], b, \texttt{EMPTY}) = \texttt{true}$, and leaving before the next $C\&S_q(\texttt{location}[r], b, \texttt{EMPTY}) = \texttt{true}$, $p, q \neq r$, is not a canceled token.*

Lemma 5.14 *Any token (r, T) leaving the balancer b, before the first $C\&S_p(\texttt{location}[r], b, \texttt{EMPTY}) = \texttt{true}$, $p \neq r$, is not a canceled token.*

Theorem 5.15 *A diffracting balancer b has the step property in any quiescent state.*

Proof: By lemma 5.9 is suffices to show that the number of canceling tokens is equal to the number of canceled tokens. For any processor r, operations performed on `location[r]` are either *C&S* operations by processors other than r, or writes and *C&S* operations by r. Any token whose shepherding process successfully performing a *C&S* operation on another processor's element of the `location` array, is a canceling token. Lemma 5.12 shows us that following every canceling token there is one canceled token, and corollaries 5.13 and 5.14 tell us that there are no other canceled tokens. ■

We define the following notion of progress for a balancer:

Non-blocking In any execution where a non-empty set of tokens are accessing a balancer b, some token will enter b within a finite number of steps.

Theorem 5.16 *The diffracting balancer has the non-blocking property.*

Proof: Phase 1 of the code contains compare-and-swaps, register-to-memory-swaps, and writes, all of which are guaranteed to complete. Since this phase contains no backward branches it will end in a finite number of steps. In Phase 2 processor p reads the `location` array a finite number of times, and then attempts to seize the lock on the toggle bit. If p or any other processor shepherding a token ever manages to seize the lock it will exit the balancer in a finite number of steps, releasing the lock. The only case where p repeatedly fails in entering the critical section is thus if other processors are constantly acquiring the lock and releasing it, i.e. leaving the balancer. ■

A diffracting tree has the form of a binary tree with one shared `location` array used by all diffracting balancer nodes. The following lemma shows that having a shared `location` array does not invalidate balancer behaviour as defined by Theorem 5.15 and Theorem 5.16.

Theorem 5.17 *Any k distinct diffracting balancers $b_1 \ldots b_k$ will have the step property and the non-blocking property even if they share the same* `location` *array.*

Proof: For any processor r, the operations performed on `location[`r`]` are either:

- $write_r$(`location`$[r], b$), on lines 3 and 10; or
- $C\&S_p$(`location`$[r], b,$ `EMPTY`), where p is any processor, on lines 7,8, and 19,

where $b \in \{b_1 \ldots b_k\}$. A processor shepherds only one token at a time through the diffracting tree. Thus, if r is currently shepherding a token through balancer b_i, no processor shepherding a token through any other balancer b_j can cause a change in `location`$[r]$. Obviously, $write_r$(`location`$[r], b$) operations are unaffected by having a shared `location`$[r]$ array since they are performed by r itself. *C&S* operations succeed only if performed by a token in the same balancer b_i, and failed *C&S* operations do not change the value of `location`. The claim thus follows from Theorem 5.15 and Theorem 5.16. ■

Notice that while our implementation of diffracting trees is non-blocking, as proven in Theorem 5.17, it is not starvation free. A token might be blocked forever repeatedly attempting to acquire the lock on toggle bit before succeeding.

Theorem 5.18 *The* BINARY$[w]$ *diffracting tree maintains the non-blocking and counting properties.*

Proof: The non-blocking property of each individual diffracting balancer follows from Theorem 5.17: if some token is always making progress, and there are a finite number tokens, eventually all tokens will exit balancer. Since the tree is an acyclic graph of diffracting balancers, the liveness of the entire tree follows from the liveness of each balancer, as explained at the start of this section. The counting property of the BINARY$[w]$ diffracting tree follows from Theorem 5.17 which proves that a diffracting balancer is, in fact, a balancer. Theorem 5.7 states that a BINARY$[w]$ tree of balancers counts. ■

Lemma 5.19 *Diffracting trees of balancers implemented using a fetch-and-complement toggle operation as in Figure 7 are wait free.*

Proof: Phase one of the code completes in at most 5 shared memory operation: two writes and two compare-and-swaps on the `location` array and one register-to-memory-swap on the `prism` array. In phase two at most `b->spin` reads are performed on `location[mypid]`, and after a single additional fetch-and-complement the processor leaves the balancer. ■

5.3 A Proof of the Message Passing Implementation

The correctness proof given here is based on the following assumptions:

1. Processors don't fail during the execution of the algorithm
2. The network always delivers messages in a finite number of cycles.

A token *enters* a diffracting balancer as soon as it arrives at a prism processor, and *leaves* as soon it is dispatched to the next balancer.

Lemma 5.20 *A message passing diffracting balancer is a balancer.*

Proof: We prove that is meets the balancer specification. *Safety* follows since neither prism processors nor toggle processor create any messages on their own, they only forward message sent to them. *Liveness* follows since prism processors hold tokens for no more than a bounded number of cycles before diffraction occurs or the token is sent to the toggle processor, which dispatches the token immediately. Finally, *balancing* follows since tokens enter the balancer only through the prism processors. Prism processors only dispatch balanced pairs of tokens to other balancers, otherwise the token goes through the toggle processor. Since there is only one toggle processor per balancer it gives centralized control and insures a balanced output. ■

Theorem 5.7 together with Lemma 5.20 give us:

Theorem 5.21 *A* BINARY[w] *tree of message passing diffracting balancers counts.*

6 Discussion

Diffracting trees represent a new class of concurrent algorithms prove to be an effective alternative paradigm to combining in the design of many concurrent data structures.

There is clearly room for experimentation on real machines and networks. One test application will hopefully be the 128-node-Alewife debugger's logging mechanism. The machines' concurrent logging mechanism will have all processors repeatedly write blocks of logged operations onto multiple disks. The counter handing out next-available-disk-block locations will be a test case for a diffracting counter solution to what would otherwise be a hot-spot and a sequential bottleneck [31]. As mentioned earlier, given the hardware *fetch-and-complement* operation to be added to the Sparcle chip's set of conditional load/store operations, one will be able to implement a shared

memory diffracting-tree based counter in a wait-free manner, that is, without any locks. Further enhancements to Alewife's LimitLess protocol will hopefully allow to improve performance even further. The machine is due to become operational in 1996.

We are also developing a version of diffracting trees for non-coherent shared memory machines such as the Cray T3D [48]. A recent paper by Shavit and Touitou [44] introduces "Elimination Trees," a new form of Diffracting trees that can be used to create highly parallel producer/consumer pools and stacks [38, 43]. The algorithms provide superior response (on average just a few machine instructions) under high loads with a guaranteed logarithmic (in w) number of steps under sparse request patterns.

On the more theoretical side, combining trees have the advantage of offering a general *fetch-and-*Φ operation, and it would be interesting to find out if a variant of diffracting could provide such a property. A recent paper by Shavit, Upfal, and Zemach [45] provides a combinatorial model and steady-state analysis that confirm some of the empirical results observed in this paper and offer a collection of improvements, among them a more "stable" diffracting tree algorithm. Our hope is that such modeling will allow one to determine the optimal setting of parameters such as *spin* and prism *width* in a non empirical way. It would also be interesting to formally analyze diffracting tree behavior using newly developed models of contention such as that of Dwork, Herlihy, and Waarts [17].

Finally, it would be interesting to extend the use of diffraction to other forms of counting networks such as those of Felten, LaMarca, and Ladner [18], Aiello, Venkatesan, and Yung [5], and Busch and Mavronicolas [13, 14].

7 Acknowledgments

We wish to thank Dan Touitou for his many insightful observations and the anonymous referees for their many valuable comments. Thanks are also due to Allan Fekete for his careful proof-reading of the final manuscript.

References

[1] A. Agarwal and M. Cherian. Adaptive Backoff Synchronization Techniques. In the *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 396–406, May 1989.

[2] E. Aharonson and H. Attiya. Counting networks with arbitrary fan out. In the *Proceedings of the* 3^{rd} *ACM-SIAM Symposium on Discrete Algorithms,* Orlando, Florida, January 1992. Also: Technical Report 679, The Technion, June 1991.

[3] A. Agarwal, D. Chaiken, K. Johnson, D. Krantz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, and D. Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Scalable Shared Memory Multiprocessors*, Kluwer Academic Publishers, 1991. Also as MIT Technical Report MIT/LCS/TM-454, June 1991.

[4] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors. IEEE Micro, pages 48-61, June 1993

[5] B. Aiello, R. Venkatesan, and M. Yung. Coins, Weights and Contention in Balancing Networks. In the *Proceedings of the Thirteenth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 193–214, August 1994.

[6] T.E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[7] J. Aspnes, M.P. Herlihy, and N. Shavit. Counting Networks and Multi-Processor Coordination. In the *Proceedings of the 23rd ACM Annual Symposium on Theory of Computing*, pp. 348–358, May 1991.

[8] K.E. Batcher. Sorting Networks and their Applications. In the *Proceedings of AFIPS Joint Computer Conference*, pages 338–334, 1968.

[9] R.D. Blumofe, and C.E. Leiserson. Sheduling Multithreaded Computations by Work Stealing. In the *Proceeding of the 35th Symposium on Foundations of Computer Science (FOCS '94)*, pp. 356–368, Nov. 1994.

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

[11] 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.

[12] C. Busch and M. Mavronicolas. A Combinatorial Treatment of Balancing Networks. In *Thirteenth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 206–215, August 1994.

[13] C. Busch and M. Mavronicolas. A Logarithmic Depth Counting Network. Annouced in *Fourteeth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 274, August 1995.

[14] C. Busch and M. Mavronicolas. Load Balancing Networks. Annouced in *Fourteeth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 275, August 1995.

[15] R. G. Covington, S. Dwarkadas, J. R. Jump, J. B. Sinclair, S. Madala. The Efficient Simulation of Parallel Computer Systems. *International Journal in Computer Simulations*, Vol. 1, pp. 31–58, (1991).

[16] Digital Equipment Corporation. Alpha system reference manual.

[17] C. Dwork, M. P. Herlihy, and O. Waarts. Contention in shared memory algorithms. In the *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.

[18] E.W. Felten, A. LaMarca, R. Ladner. Building Counting Networks from Larger Balancers. University of Washington T.R. #93-04-09.

[19] E. Freudenthal and A. Gottlieb. Process Coordination with Fetch-and-Increment. In the *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), SIGOPS Operating Systems Review Special Issue*, page 260, April 1991, Santa Clara, California.

[20] D. Gawlick. Processing 'hot spots' in high performance systems. In the *Proceedings IEEE COMPCON'85*, Feb. 1985.

[21] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent multiprocessors. In the *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 64–75, April 1989.

[22] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[23] G. Graunke and S. Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, 23(6):60–70, June 1990.

[24] M. Herlihy, B.H. Lim and N. Shavit. Low Contention Load Balancing on Large Scale Multiprocessors. In the *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, July 1992, San Diego, CA. Full version available as a DEC TR.

[25] M.P. Herlihy. A methodology for implementing highly concurrent data structures. In the *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, Seattle, WA, March 14-16 1990.

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

[27] M.P. Herlihy, N. Shavit, and O. Waarts. Linearizable Counting Networks. In the *Proceedings of the* 32^{nd} *Annual Symposium on Foundations of Computer Science,* San Juan, Puerto Rico, October 1991, pp. 526-535. Detailed version with empirical results appeared as MIT/LCS technical manuscript 459, November 1991.

[28] K. Hwang. *Advanced Computer Architecture.* McGraw-Hill Computer Engineering Series. ISBN 0-07-031622-8.

[29] J. R. Jump NETSIM Reference Manual. Rice University. Available by `ftp` from `titan.cs.rice.edu` as `/public/parallel/sim.tar.Z`.

[30] S. Kirkpatrick and C. D. Gelatt and M. P. Vecchi. Optimization by simulated annealing. *Science*, Vol. 220, 1983, pages 671–680.

[31] J. Kubiatowicz. Personal communication (February 1995).

[32] M. Klugerman and C.G. Plaxton. Small-depth Counting Networks. In the *Proceedings of the 24th ACM Symposium on Theory of Computing (STOC)*, pp. 417–428, 1992.

[33] M. Klugerman, Small-Depth Counting Networks. Ph.D. Thesis, MIT, 1994.

[34] Nancy A. Lynch. Distributed Algorithms. Morgan Kaufmann, San Mateo, Calif., 1996.

[35] R. Lüling, and B. Monien. A Dynamic Distributed Load Balancing Algorithm with Provable Good Performance. In the *Proceedings of the 5rd ACM Symposium on Parallel Algorithms and Architectures*, pp. 164-173, June 1993.

[36] N.A. Lynch and M.R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Sixth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 137–151, August 1987. Full version available as MIT Technical Report MIT/LCS/TR–387.

[37] B.H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pp. 25–35, 1994.

[38] Udi Manber. On maintaining dynamic information in a concurrent environment *SIAM J. Computing 15(4)*, pages 1130–1142, November 1986.

[39] MIPS Computer Company. The MIPS RISC Architecture.

[40] 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.

[41] J.M. Mellor-Crummey and T.J. LeBlanc. A software instruction counter. In the *Proceedings of the 3rd ACM International Conference On Architectural Support for Programming Languages and Operating Systems*, pages 78–86, April 1989.

[42] G.H. Pfister and A. Norton. 'Hot Spot' contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(11):933–938, November 1985.

[43] L. Rudolph, M. Slivkin, and E. Upfal. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In the *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, pp. 237–245, July 1991.

[44] N. Shavit, and D. Touito. Elimination Trees and the Construction of Pools and Stacks. To appear in the *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, Philadelphia, May 23-26, 1996.

[45] N. Shavit, E. Upfal, and A. Zemach. A Steady-State Analysis of Diffracting Trees. To appear in the *Proceedings of the Eight Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, Padua, Italy, June 1996.

[46] N. Shavit and A. Zemach. Diffracting Trees. In the *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1994.

[47] H.S. Stone. Database applications of the fetch-and-add instruction. *IEEE Transactions on Computers*, C-33(7):604–612, July 1984.

[48] Cray Research. CRAY T3D System Architecture Overview. Available via WWW as `http://www.cray.com/PUBLIC/product-info/mpp/T3D_Architecture_Over`.

[49] P.C Yew, N.F. Tzeng, and D.H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, pages 388–395, April 1987.