Diffracting Trees

(PRELIMINARY VERSION)

Nir Shavit* Asaph Zemach*

Abstract

Shared counters are among the most basic coordination structures in multiprocessor computation, with applications ranging from barrier synchronization to dynamic load balancing. Introduced in this paper are *diffracting trees*, novel distributed-parallel data structures for shared counting. Diffracting trees combine a randomized coordination method together with a combinatorial data structure, to yeild a logarithmic depth counter that improves on the log^2 depth of counting networks, and overcomes the resiliency drawbacks of combining trees. Empirical evidence collected on a simulated distributed shared-memory multiprocessor shows that diffracting trees *substantially* outperform both combining trees and counting networks, currently the most effective known methods for shared counting. Not only do diffracting trees have higher throughput and lower latency, but unlike any known technique, their latency remains almost constant as the number of processors increases.

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 [20], index distribution, shared program counters [21] and concurrent data structures (see also [12, 14, 25]). In its purest form, a counter is an object that holds an integer value and provides a *fetch&inc* operation, incrementing the counter and returning its previous value. Given that the majority of current multiprocessor designs do not provide specialized hardware support for efficient counting, there is a growing need to develop effective software based counting techniques.

The simplest way to implement a counter is to place it in a spin-lock protected critical section, adding an exponential-backoff mechanism [1, 4, 15] or a queue lock as devised by Mellor-Crummey and Scott [20] or Anderson [4] to reduce contention. Unfortunately, such centralized methods are inherently non-parallel and cannot hope to scale well.

A recent survey of counting techniques by Herlihy, Lim, and the present author [16] 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 Software Combining Trees of Yew, Tzeng, and Lawrie [26] and Goodman, Vernon, and Woest [13], and the Counting Networks of Aspnes, Herlihy, and the present author [5], both meet the above criteria, and indeed were found by [16] to be the most effective methods for concurrent counting.

A combining tree is a distributed data structure with a shared counter at its root. Processors combine their increment requests going up the tree from the leaves to the root, eliminating the need for all to actually reach the counter. A Bitonic counting network is a distributed data structure having a layout isomorphic to a Bitonic sorting network [6] with a local counter at the end of each output wire. The network has width $w \ll n$ and depth $\frac{1}{2}\log^2 w$.[1] Combining trees have logarithmic depth and the desirable property that the unavoidable "collisions" of processors at their nodes are utilized to increase parallelism, but introduce high dependency among processes and cannot withstand even a single processor failure. Counting networks on the other hand, support complete independence among requesting processes and are highly fault tolerant, but have $\log^2$ depth and do not make use of the collisions at their nodes.

This paper introduces *diffracting trees*, a new distributed-parallel technique for shared counting that enjoys the benefits of each of the above methods and avoids many of their drawbacks. In a manner similar to counting networks, diffracting trees 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

*Department of Computer Science, School of Mathematics, Tel-Aviv University, Tel-Aviv 69978, Israel. Contact: shanir@math.tau.ac.il

This work was supported by a Digital Equipment Corporation ERP Equipment Grant

Keywords: Shared Counters, Counting Networks, Load Balancing, Concurrent Data Structures, Randomization.

SPAA 94 - 6/94 Cape May, N.J, USA
© 1994 ACM 0-89791-671-9/94/0006..$3.50

[1] Klugerman and Plaxton [18] have designed elegant combinatorial constructions of counting networks with depth close to $O(\log w)$, unfortunately though, at this point in time the constants involved are "exponentially large".

the tree sequentially, one completely after the other. Figure 1 shows such an execution on a tree of width 4. As can be seen, the tree moves input tokens to output wires in increasing order modulo 4. Trees of balancers having this property can easily be adapted to count the total number of tokens that have entered the network. Counting is done by adding a "local counter" to each output wire i, so that tokens coming out of that wire are consecutively assigned numbers $i, i+4, i+(4\cdot 2)\ldots$

A clear advantage of a tree over a network is its depth which is logarithmic in w. However, it seems that we are back to square one since the root of the tree will be a hot-spot 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 bit toggled by each passing token. We are able to overcome the problem based on the following simple observation: an even number of tokens passing through a balancer leave the toggle bit unchanged. This means that if one could have independent pairs of tokens diffracted in a coordinated manner one to the left and one to the right, they could leave the balancer without ever having to toggle the shared bit. The idea behind *diffracting trees* is to create such a "prism" mechanism, in front of the toggle bit of every balancer. By distributing the prism over many locations , and ensuring that each pair of tokens uses a different location, we would get a highly parallel balancer with very low contention. The diffraction mechanism uses randomization to ensure high collision/diffraction rates on the prism, and the tree structure guarantees correctness of the output values. *Diffracting trees* thus combine the the high degree of parallelism and fault-tolerance of the counting networks with the logarithmic depth and beneficial utilization of "collisions" of a combining tree.

We compared the performance of diffracting trees with the above techniques on a simulated distributed shared-memory multiprocessor using the well accepted Proteus Parallel Hardware Simulator [8, 7]. We found that diffracting trees *substantially* outperform both combining trees and counting networks, currently the most effective known methods for shared counting. Not only do they have higher throughput and lower latency, but in fact, their latency remains almost constant as the number of processors increases.

Diffracting trees can also be used to create a highly parallel centralized job queue implementation or as a general load balancing tool (dropping the counters at the end of the output lines). Unlike the distributed job queues of Rudolph, Slivkin, and Upfal [24] and the randomized $\log w$ smoothing networks of Aiello, Venkatesan, and Yung [3] which give only a probabilistic correctness guarantee that the load will be balanced, diffracting trees always correctly balance the number of tokens on their output lines. We believe diffracting trees will prove to be an effective and useful technique in other application areas, and are currently testing a message passing version of the technique that can be used both on multiprocessors and computer networks.

This preliminary version includes a description of counting trees, a shared memory implementation of diffracting balancers and an evaluation of their performance. The outlines of the correctness proofs can be found in the appendices.

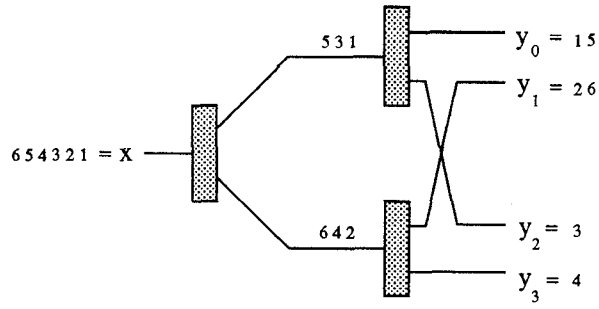

Figure 1: A Simple Counting Tree

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 in [5]. 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 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 in [5]. 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.

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. Figure 1 shows such an execution on a BINARY[4] counting tree 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 consecutively assigned numbers $i, i+w, \ldots, i+(y_i-1)w$. Code for implementing such a counter can be found in Figure 2.

In our implementation we will be using a counting tree called BINARY[w], which we define below. 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[$2k$] network 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 A.6 in Appendix A proves that BINARY[$2k$] is indeed a counting tree.

```
type balancer {
  lock:   boolean
  toggle: boolean
  next:   array [0..1] of ptr to balancer
}

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

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

function fetch&incr(): integer
  b:= root
  while not leaf(b)
    b := balancer(b)
  endwhile
  i := increment_counter_at_leaf(b)
  return i * width + b->number
```

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

3 Diffraction Balancing

Diffracting trees are counting trees whose balancers are of a novel type called *diffracting balancers*. In the typical implementation of balancers (as in Figure 2), each processor shepherding a token through the tree toggles the bit inside the balancer, 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. One can overcome this sequential bottleneck based on 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 pairs of colliding tokens to "pair-off" and coordinate among themselves which is diffracted "right" and which diffracted "left", they could both leave the balancer without either of them ever having to touch the toggle bit. By performing the collision/coordination decisions in separate locations instead of a global toggle bit, we will hopefully increase parallelism and lower contention. However, we must guarantee that many such collisions occur, not an obvious task given the asynchrony in the system.

On a high level, our implementation of the above is based on adding a special `prism` array "in front" of the toggle bit in every balancer. When a token (processor) P enters the balancer, it first selects a location, L, in `prism` uniformly at random. P tries to "collide" with the previous processor to select L, or, by waiting for a fixed time, with the next processor to do so. If a collision occurs, both processors leave the balancer on separate wires without ever attempting to toggle the bit.

Figure 3 gives the diffracting balancer data structure and contains the code for this type of balancer. Three synchronization operations are used in the implementation code:

- `register_to_memory_swap(addr,val)` writes `val` to address `addr`, and returns the previous value there,
- `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 returns `FALSE`, and
- `test_and_set(addr)` writes `TRUE` to address `addr` and returns the previous value.

All three primitives can be implemented in a lock-free manner using the fashionable load-linked/store-conditional operations available on standard architectures [10, 19].

The code also uses two functions: (a) `random(i,j)` returns a random number between `i` and `j`; (b) `not_empty(i)` returns `TRUE` if `i` is the PID of some processor and `FALSE` otherwise.

The code translates into the following sequence of operations performed by a process shepherding a token through a balancer. In Phase 1 of the code the processor announces its arrival at the balancer, by writing to the global `location` array. It then swaps its own PID for the one written in a randomly chosen location in the `prism` array. Assuming it has read the PID of an existing processor, it attempts to collide with it. The collision itself is accomplished by performing two compare-and-swap operations. The first erases this processor from the list of processors waiting at this balancer (thus assuring no other processor will collide with it), the second erases the other processor, completing the diffraction, and allowing the process to be diffracted to the `b->next[0]` balancer. If the first compare-and-swap fails, it means that some other processor has already managed to collide with it, and the processor is diffracted to the `b->next[1]` balancer. If the first succeeds but the second compare-and-swap fails, it means that the processor with whom it was trying to collide is no longer available, in which case it goes on to phase 2.

In Phase 2 the processor repeatedly checks to see if it has been diffracted by another processor. After spinning `spin`

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

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

function diff-bal(b: ptr to balancer): ptr to balancer

  /* phase 1 */

  location[mypid] := b
  place := random(1,b->size)
  him := register_to_memory_swap(b->prism[place],mypid)
  if not_empty(him) then
     if compare_and_swap(location[mypid],b,EMPTY) then
         if compare_and_swap(location[him],b,EMPTY) then
             return b->next[0]                          (a)
         else location[mypid] := b
     else return b->next[1]                             (b)
  endif

  /* phase 2 */
  forever
     repeat b->spin times
        if location[mypid] <> b then
           return b->next[1]                            (b)
     endrepeat
     if test_and_set(b->lock) then
        if compare_and_swap(location[mypid],b,EMPTY)
        then
           i := b->toggle
           b->toggle := not(i)
           unlock(b->lock)
           return b->next[i]                            (c)
        else
           unlock(b->lock)
           return b->next[1]                            (b)
        endif
     endif
  endfor
```

Figure 3: Code for traversing a diffracting balancer

times, giving some other processor a chance to diffract it, the processor attempts to get the toggle bit. If successful, it first removes itself from the list of waiting processors and then toggles the bit and exits the balancer. If it could not remove itself from the list, it follows that some other processor already collided with it, and it exits the balancer, being diffracted to `b->next[1]`. If the toggle bit could not be seized, the process resumes spinning. Appendix B contains the formal correctness proof for this algorithm.

3.1 Some implementation details

When a large number of processors concurrently enter the balancer, the chances for successful collisions in `prism` are high, and contention on the toggle bit is unlikely. When there are few processors, each will spin a short while, reach for the toggle bit and be off, since all spinning is done on a cached copy of the value of `location[mypid]` it incurs no overhead. The only case where a processor is repeatedly making accesses to memory, is when no other processor cancels (by diffracting) it, and it is constantly reaching 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 effects 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 toggle bit. If it is too low, contention will occur on the array `prism` as too many processors will be trying to access it at the same time.

2. `spin` — If this value is too low, processors will not have a chance to pair-off, and contention will occur on 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 `size = 8,4,2,1,1` for the various levels of a width 32 tree. We also employed a form of exponential backoff on the balancer's toggle bit. Each processor kept a local copy of the diffracting balancer's `spin` variable, and doubled it each time it could not seize the lock, thus increasing the amount of time it waited to be collided with. The value of the local copy was not retained between calls. In order to maximize the distribution of the balancer's data structure the `prism` array was actually an array of pointers to different modules of memory. The random number function we used was Proteus' `fast_random` which is an implementation of the ACM Minimal Standard Random Number Generator [22, 9].

4 Performance

We evaluated the performance of counting 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 [2]. Our simulations were performed using *Proteus*[2], a multiprocessor simulator developed by Brewer, Dellarocas, Colbrook and Weihl [8]. In this abstract we present the results of running a benchmark called *Index-Distribution*. Index-distribution, is a load

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

balancing technique, in which processors dynamically choose loop iterations to execute in parallel. As mentioned in [16], 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()
          repeat random(0,work) times
                /* nothing */
          endrepeat
          goto loop
```

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

We ran the benchmark several times, varying the number of processors participating in the simulation, and the amount of `work` done. Each time we measured:

Latency The average amount of time between the moment `get_next_index` was called, and the time it returned with a new index. This was measured by taking the time at the beginning of each call, the time at the end, and adding the difference to a global counter, which was then divided by the number of increments performed.

Throughput The average number of indices distributed in a one million cycle period. This cycle count includes the time spent spinning in the work loop. It was measured by marking the time after the first 100 increments where performed, and then measuring, t, the time it took to make d more increments. The throughput is: $10^6 d/t$.

We compared a collection of the fastest known software counting techniques with respect to the index-distribution benchmark.

ExpBackoff A lock using *Test&Test&Set* with exponential backoff [4, 15].

MCS The MCS lock of [20]. Each processor locks the shared counter, increments it, and then unlocks it. The code was taken directly from the article, and implemented using atomic *Swap* and *Compare&Swap* operations.

CTree *Fetch&Inc* using an optimal depth combining tree. We implemented the software combining tree protocol of Goodman et al. [13], modified according to [16]. Optimal depth means that when p processors participate in the simulation, a tree of depth $\lceil \log_2 p \rceil$ is used.

CNet A BITONIC counting network of width 32 [16]. The toggle bit was implemented using a short critical section.

DTree A Diffracting Tree of width 32.

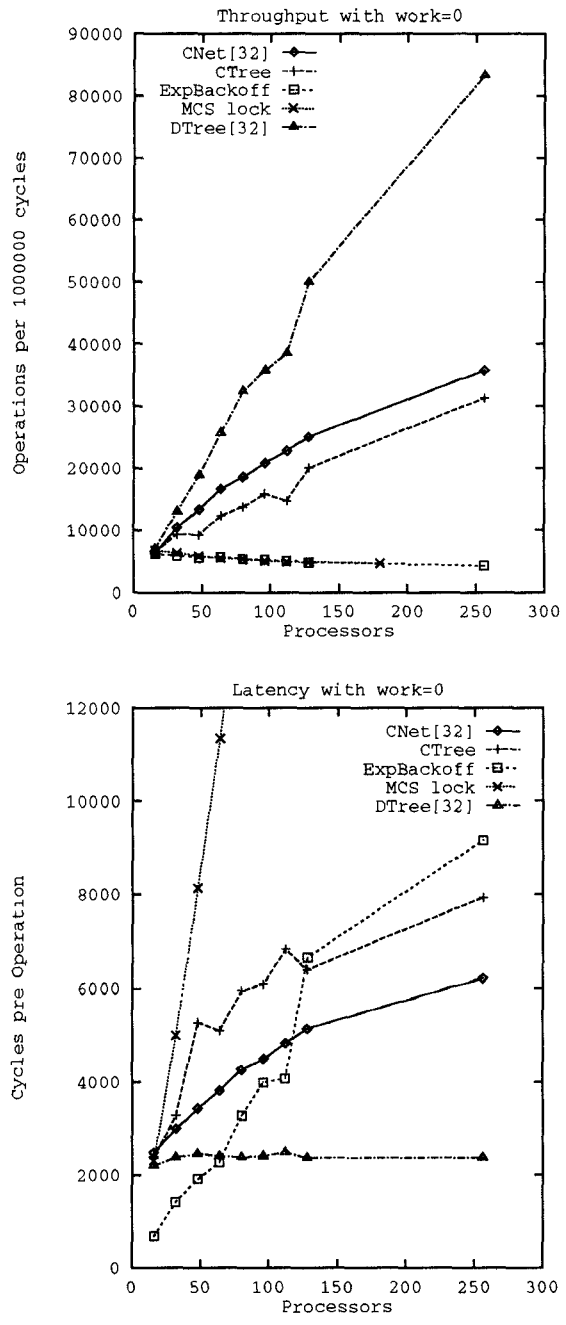

Figure 4: Throughput and Latency of Major Counting Techniques

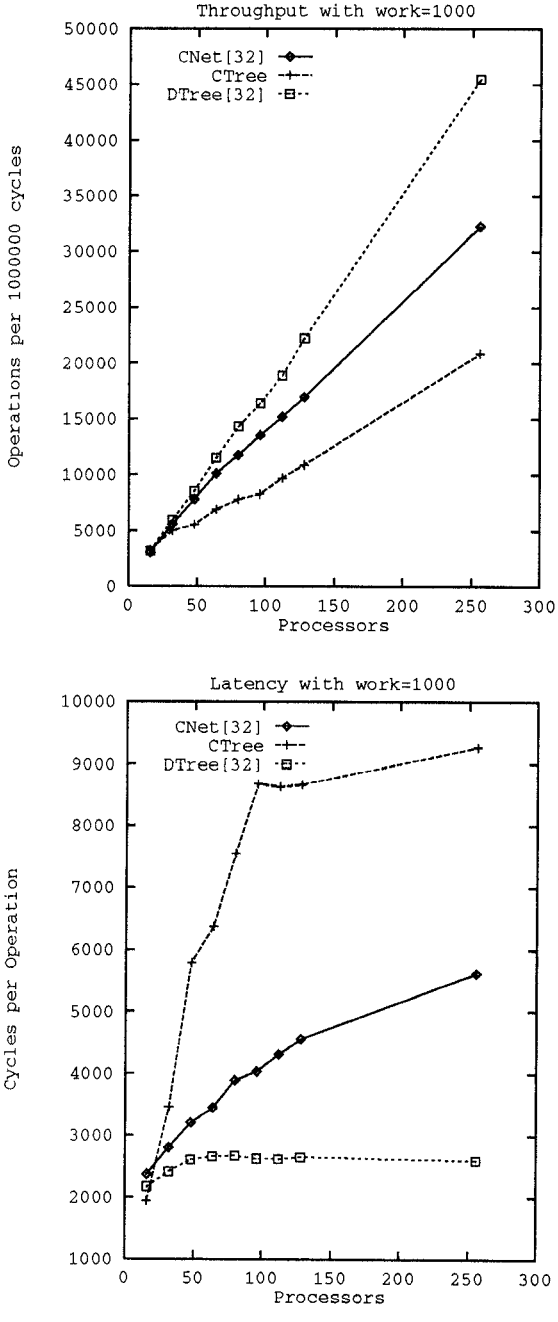

Figure 5: Throughput and Latency with **work**=1000

The graphs in Figure 4 show the latency and throughput of the various counting methods. They confirm previous findings and in particular, they agree with the results of [16] on *ASIM*, the *Alewife* machine simulator [2]. We have not included results of various optimizations such as building counting networks from balancers with four or more input/output wires. Elsewhere [11], and in our simulations, these have been shown to give performance improvements of at most 25%, when **work**=0 and almost no improvement as **work** increases. It is clear from these graphs that the MCS lock and the exponential backoff lock do not scale well, latency grows quickly, and throughput diminishes. This is not surprising, since both are methods for eliminating contention but do not support parallelism.

We therefore concentrate on the latency and throughput results of the three distributed-parallel techniques: combining trees, bitonic counting networks and diffracting trees. The graphs in Figures 4 and 5 show that diffracting trees give consistently better throughput than the other methods and that in terms of latency they scale extremely well, tending to hand out numbers in almost constant time (on average).

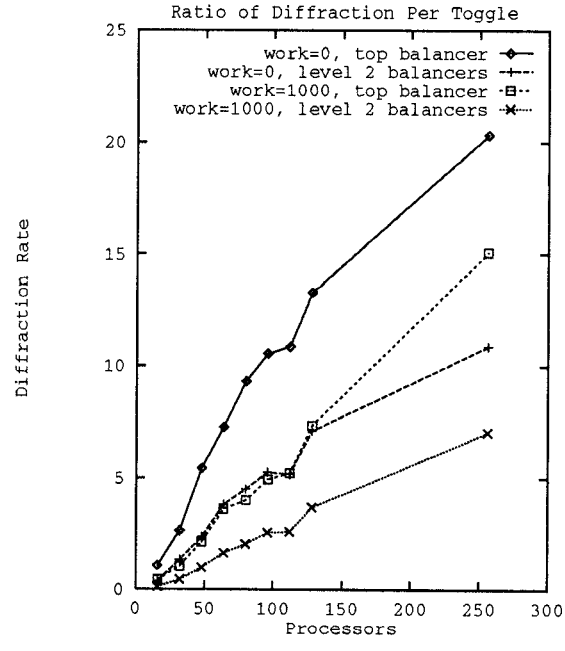

Figure 6: Diffraction Rate

The high throughput is explained by the parallelism, due to the optimized rate of successful collisions in the **prism** arrays of balancers. we claim that 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. The justification of this claim is provided by the graph in Figure 6 which shows the diffracting rate of tokens in the top two balancers. The diffracting rate is the number of diffracted processors, leaving the balancer without toggling the bit, divided by the number of those that did toggle. For

example, when the diffraction rate is 20, it means that for every token toggling the bit, 20 will be diffracted. If we measure a diffraction rate of r at some balancer, we conclude, that if till now t tokens have entered it, then the total number of tokens toggling the bit, τ, is given by $\frac{t}{r+1}$ and the total number diffracted is $\frac{tr}{r+1}$. The diffraction graph indicates a linear a relationship of the form $r = cp$, where p is the number of processors participating in the simulations, and c is some constant. We can now restate the previous formula, and say that $\tau \approx \frac{t}{cp+1}$. Our measurements indicate this predictor to be accurate to within 15% when the number of processors is above 64. During sufficiently short time periods, t is bounded by p, since only p tokens can exists simultaneously, and we get $\tau \leq \frac{p}{cp+1} \leq \frac{1}{c}$. The number of processors reaching for the toggle bit simultaneously (the contention on it) is thus bounded by the constant $\frac{1}{c}$ (We measured this constant at about 8.5 for the top-level balancer, when `work` was 0). The diffracting balancer thus tends to keep the contention at the toggle bit constant, regardless of the total number of processors in the system, an indication of the robustness of the construction.

There is an interesting resemblance between the shape of the throughput graphs of the combining tree and the diffracting tree. This is probably the result of both trees having a similar coordination property: combining collided requests in one and successful diffracting of collided tokens in the other. As shown above, as the number of processes in the tree increases, the likelihood of such collisions happening grows. Counting networks do not gain a performance advantage (and even loose performance) from this phenomena, since they are designed only to minimize the number of collisions on individual toggle variables, not to take advantage of them. When processors fail to meet at the nodes, performance of both the tree methods diminishes, but like counting networks, diffracting trees still have an advantage given the low contention on the toggle variables.

The low latency of diffracting trees is due in part to their low depth. While combining trees have a depth of $\log n$ where n is the number of processes, and counting networks have a fixed depth of $1/2 \log^2 w$ where w is the width of the network, diffracting trees have a depth of only $\log w$. For example, with 256 processors, the combining tree rises to depth 8, the width 32 counting networks have depth 10, whereas diffracting trees have depth of only 5. The low latency of diffracting trees remains almost constant as the number of processes increases, since the high rate of successful collisions works in their favor by lowering contention and thus ensuring that most processes take order of $\log w$ steps.

The graphs in Figure 5 show that, as the amount of work between accesses to the shared counter increases, the throughput of all methods decreases, as would be expected. The latency of the counting network remains unchanged, that of the combining tree initially grows rapidly and then seems to begin evening out. The latency of the diffracting trees evens out much sooner, and even slightly diminishes as the number of processors increases. We can conclude that diffracting trees maintain their superiority even in the face of substantial work loads. When the number of processors is small (less than 64), the latency of all methods, including our own, is greater than that of the exponential backoff lock. We are currently studying adaptive versions of the algorithm, (able to shrink or grow in order accommodate different numbers of processors) that promise to address this difficulty.

In summary, diffracting trees enjoy both the parallelism of counting networks and the high coordination of combining trees. One must also remember that like counting networks and unlike combining trees, diffracting trees can be made "lock-free," that is, guarantee progress even if processors fail.

5 Conclusions

Current approaches to multiprocessor computing are geared towards providing performance speedup for input/output problems (as in numerical computing) that are solved using parallelized sequential alogorithms. This, in our view, is the reason why one can program distributed and asynchronous multiprocesor machines using tight synchronization paradigms like critical sections and barriers, and still observe good performance. However, future roles of multiprocessor machines will include application areas in industrial production control, aircraft flight control, and eventually even robot brains. These will require general purpose multitasking capabilities and on-line control of high speed asynchrnous data arriving from multiple sources, and are inherently asynchronous and distributed in nature. They will gain more from methods that promote coordination between processors without actually synchronizing them.

Combining Trees are an example of a data structure that achieves parallelism by synchronizing multiple accesses to a single location. Diffracting Trees, on the other hand, achieve parallelism by distributing requests to different locations, assuring correctness through coordination. They are an example of a novel *Distributed-Coordinated* (*DisCo* if you will) approach to concurrent data structure design. We believe this novel approach (which includes lock-free and wait-free methods, but does not prohibit use of locks) will better bridge the gap between the applications and the machines which run them.

References

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

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

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

[4] 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.

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

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

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

[8] 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.

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

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

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

[12] E. Freudenthal and A. Gottlieb. Process Coordination with Fetch-and-Increment. In *Proceedings of the 4th International Conference on Architecture Support for Programming Languages and Operating Systems*, April 1991, Santa Clara, California. To appear.

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

[14] 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.

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

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

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

[18] M. Klugerman and C.G. Plaxton. Small-depth Counting Networks. 1992 ACM Symposium on the Theory of Computing.

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

[20] 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.

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

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

[23] 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.

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

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

[26] 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.

A A proof that counting-trees count

Following [5], let the state of a balancer at a given time be defined as the collection of tokens on its input and output wires. For the sake of clarity we will assume that tokens are all distinct. We can now formally state the properties of a balancer:

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 are not connected to output wires of balancers), 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 counting tree reaches a quiescent state, then its outputs, $y_0, \ldots, y_{w-1}$ have the stop property. We present the following useful lemmas due to [5].

Lemma A.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 A.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 A.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 A.4 *let $x_0, x_1, \ldots, x_n$ and $y_0, y_1, \ldots, y_n$ be two arbitrary sequences having the step propery. Then if*

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

then the sequence

$$x_0, y_0, x_1, y_1, \ldots, x_n, y_n$$

has the step property.

Proof outline: There are two cases:

1. $\sum_{i=0}^{n} y_i = \sum_{i=0}^{n} x_i$, in this case, by Lemma A.2 , both sequences are identical, and the proof is trivial.
2. $m = \sum_{i=0}^{n} y_i = \sum_{i=0}^{n} x_i + 1$, in this case, Lemma A.3, applies. We know from Lemma A.1 that $x_i = \lceil \frac{m-1-i}{k} \rceil$ and $y_i = \lceil \frac{m-i}{k} \rceil$, this means that $\forall i, x_i = y_{i+1}$. The joint sequence has the form $x_0 = y_0 = x_1 = y_1 = \ldots = x_{j-1} = y_{j-1} = x_j = y_j - 1 = x_{j+1} - 1 = y_{j+1} - 1, \ldots, = x_n - 1, y_n - 1$. This sequence has the step property.

■

Theorem A.5 *The outputs of* BINARY[w] *have the step property in any quiescent state.*

Proof outline: The proof is by induction. If $w = 1$ then we are dealing with a BINARY[2] counting tree. This tree has two outputs, and is therefore, simply, a balancer. By definition, the outputs of a balancer have the step property. 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 big tree of width $2k$, is actually one root balancer whose two outputs are connected to small trees of width k. The even leaves of the the big 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 the number of inputs to the left and right small trees differ by at most one. By Lemma A.4, the outputs of BINARY[$2k$] have the step property. ■

Building on the work of [5] the following theorem is now immediate.

Theorem A.6 *A* BINARY[w] *tree counts.*

B A proof of the diffracting balancer implementation

This section outlines the proof that the shared memory implementation of a diffracting balancer is indeed a balancer, *i.e.*, it conforms to the formal definition of a balancer given in Appendix A. The proof makes the implicit assumption that all threads run to completion. For brevity, we will be using *C&S* instead of `compare_and_swap`.

Lemma B.1 *The implementation meets the safety condition.*

Proof outline: In our shared memory implementation, each token represents a thread on some processor. Since the code contains no thread creation commands, no new token can be created in the diffractingbalancer. ■

Lemma B.2 *The implementation meets the liveness condition.*

Proof outline: A single processor accessing the balancer will by the code return within a finite number of steps. If there are several processes executing the balancer code and no processor performs a return then some process must be repeatedly executing in the `forever` loop. By the code this means it must be failing to obtain the lock via the `test_and_set` operation. This in turn implies that the lock must be taken by some other processor, which will within a bounded number of operations return a value. For any number m tokens passing through the balancer, it follows that eventually all of them must exit the loop. ■

In order to prove the balancing property we will require some definitions. Note that Figure 3 marks each `return` point with a letter. A token exiting the diffracting balancer code via the `return` marked `(a)` will be called a *cancelling* token. A token that leaves through a `return` marked `(b)` will be called a *cancelled* token, and one that leaves through `(c)` will be called a *toggling* token. By definition, any two tokens passing through the tree concurrently are being shepherded by processes with different PID. For a token t, PID(t), will denote its PID.

Lemma B.3 *At any time that* `location[PID(`t`)] = b` *there is a process* PID *executing the code of (shephearding a token* t *through) diffracting balancer* `b` *that has not performed its final operation on* `location[PID(`t`)]`.

Proof outline: Initially all `location[PID(`t`)]` locations are empty, so this property holds. Assume that it holds in some state S and let us prove that it holds in any state S' reachable from S following some operation. The only operation that can set a `location[PID(`t`)]` to b is by process PID and in its next step it is still a executing the code of balancer b. All `return` events are conditional on a final operation that either tests that `location[mypid]` is not b or explicitely sets it to `EMPTY`. ■

Lemma B.4 *The number of cancelled tokens is equal to the number of cancelling tokens.*

Proof outline: To show this, we have to show that with each token that exited the diffracting balancer code at `(a)` we can associate a unique token that exited at a point marked `(b)`. We first show that for each cancelling token, $\hat{t}$, there is a unique cancelled token t. If $\hat{t}$ is cancelling, then it must have succeeded in both of its *C&S* operations. This means that it has written `EMPTY` to both `location[PID(`$\hat{t}$`)]` and `location[PID(`t`)]`, where t is some other token whose PID is held in $\hat{t}$'s `him` variable. By lemma B.3, when $\hat{t}$ succeeded in its *C&S*(`location[PID(`t`)],b,EMPTY`), process PID(t) had yet to perform it final access to the location. This final access can be either:

1. A read, that will find that `location[PID(`t`)] <> b`; or
2. *C&S*(`location[PID(`t`)],b,EMPTY`), that will fail.

In both cases t is a cancelled token. Token t is unique to $\hat{t}$ since it only reads the array `prism` once.

We now show that for each cancelled token t there exists a unique cancelling token $\hat{t}$. Once again we notice that t, upon entering the diffracting balancer code executes the statement `location[PID(`t`)]:=b` at most twice. A successful *C&S*(`location[PID(`t`)],b,EMPTY`) by t itself, must precede the second such write. Thus, since all erasing of `b` from `location[PID(`t`)]` is done using a *C&S* operation only one token $\hat{t}$ can succeed. Successful writing into another process' location implies by the code that $\hat{t}$ returns as a cancelling token. ■

We can now complete the proof of Theorem B.5.

Theorem B.5 *The diffracting blancer code of figure 3 implements a balancer.*

Proof outline: Lemmas B.1 and B.2 prove token safety and liveness. We now assume that the diffracting balancer is in a quiescent state and proove that $y_0 = \lceil m/2 \rceil$ and $y_1 = \lfloor m/2 \rfloor$. Each token exiting the diffracting balancer code must be either cancelled, cancelling or toggling. Each cancelled token increments y_1 by one, and each cancelling token increments y_0 by one. Lemma B.4 allows us to ignore these tokens, and count only the toggling tokens which by the proofs of [5] for a toggle based balancer are properly balanced. ■