Lecturer: Nir Shavit

Handout 11

1 Overview

1. 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. It has logarithmic depth and the desirable property that the unavoidable collisions of processors at their nodes are utilized to increase parallelism. However, it introduces high dependency among processes and cannot withstand even a single processor failure.

2. A Bitonic counting network is a distributed data structure having a layout isomorphic to a Bitonic sorting network with a local counter at the end of each output wire. It supports complete independence among requesting processes and is highly fault tolerant, but has $\log^2$ depth and does not make use of the collisions at their nodes.

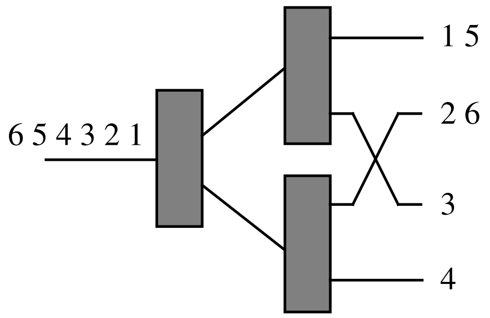


Figure 1: Counting network

2 Diffracting Tree

A distributed parallel technique for shared counting that is constructed, in a manner similar to counting network, from simple one-input two-output computing elements called balancers that are connected to one another by wires to form a balanced binary tree. One can view 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. However, to overcome the problem of sequential bottleneck in the root of the tree, a "prism" mechanism is created in front of the toggle bit to diffract each independent pair, one to the left and one to the right, without even having to toggle the shared bit. By distributing the prism over many locations and ensuring that each pair of tokens use different locations, we can 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 high degree parallelism and fault tolerance of the counting networks with the logarithmic depth, and the beneficial utilization of collisions of a combing tree.

3 Trees That Count

A counting tree balancer is a computing element with one input wire and two output wires. 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 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, the sets of input and output tokens are the same. In any quiescent state, $y_{o=\lceil x/2 \rceil}$ and $y_{1=\lfloor x/2 \rfloor}$.We extent this notion of quiescence to trees and define a counting tree if width w as a balancing tree whose outputs $y_{o,\ldots,}y_{w-1}$satisfies the *step property*:

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

```
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 := typical-balancer(b)
  endwhile
  i := increment_counter_at_leaf(b)
  return i * width + number_of_leaf(b)
```

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

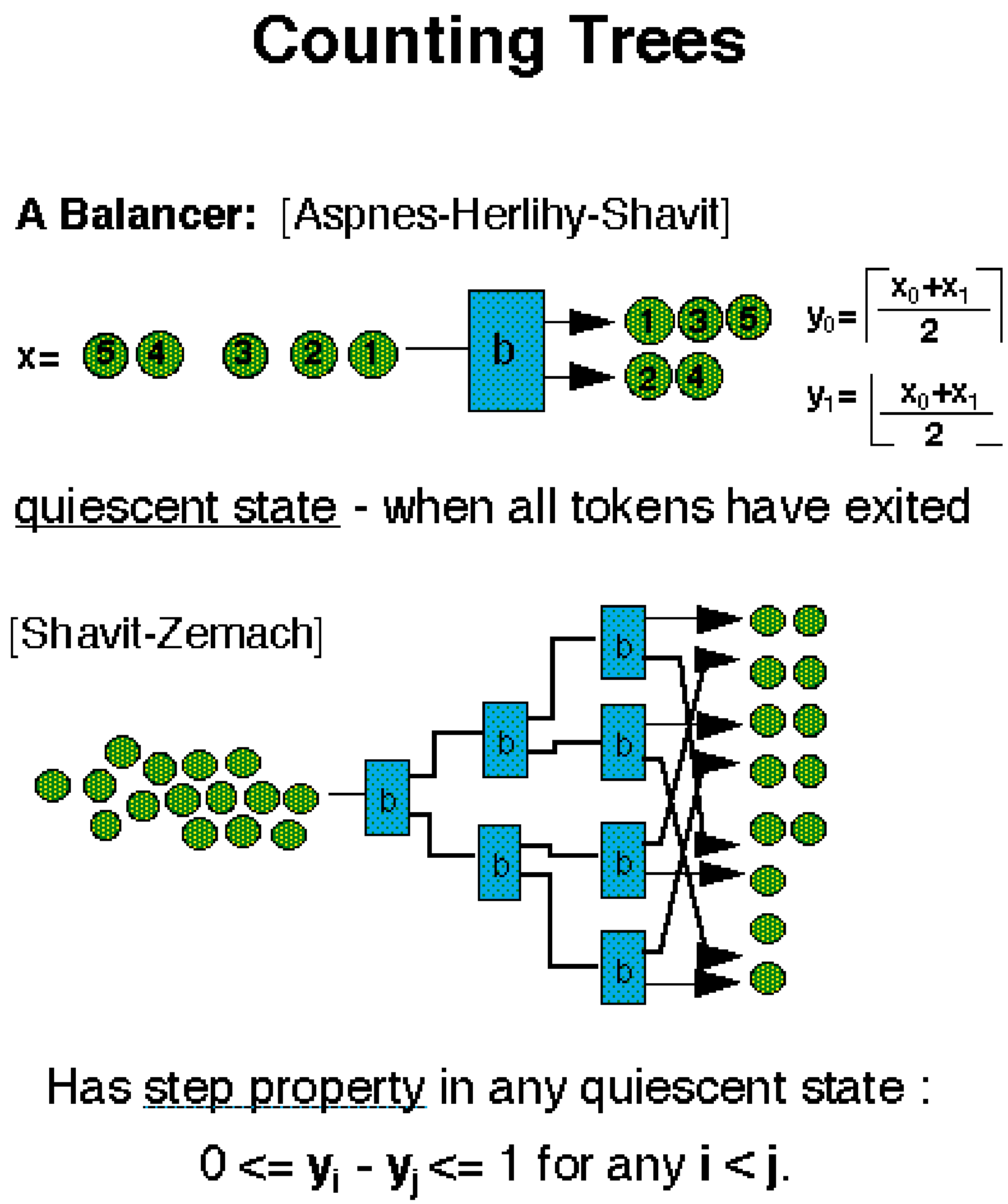


Figure 3: Counting Tree

4 Diffraction Balancing

In the typical implementation of a counting tree balancer, 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 hotspot. Even if one applies contention reduction techniques such as exponential backoff, the toggle bit still forms a sequential bottleneck. We overcome this problem based on the following observation:

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

That is to say if we can 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 can both leave the balancer without even having to touch the toggle bit. By performing the collision/coordination decisions in separate locations instead of a global toggle bit, we will increase parallelism and lower contention.

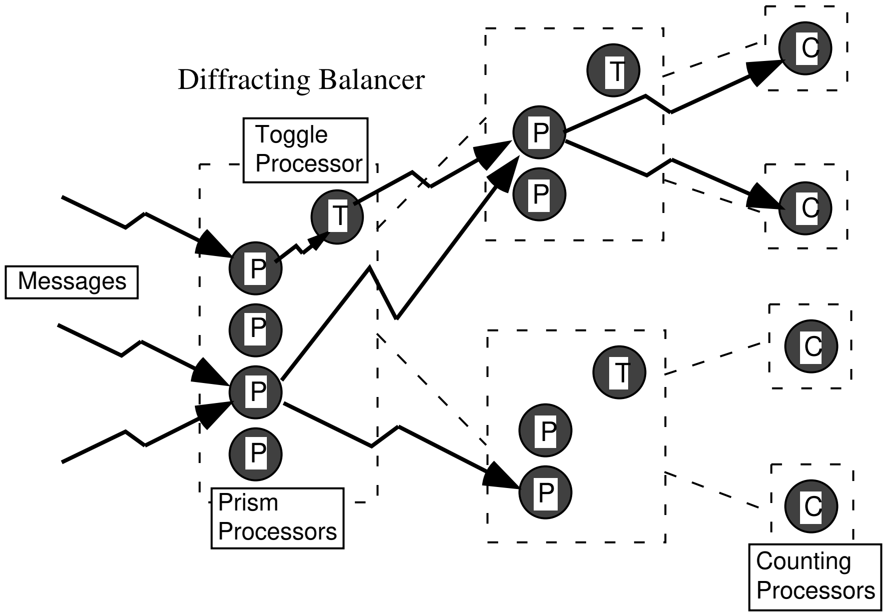


Figure 4: Diffracting balancer

The implementation of our diffracting balancers thus 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 then tries to collide with the previous processor that has selected 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.

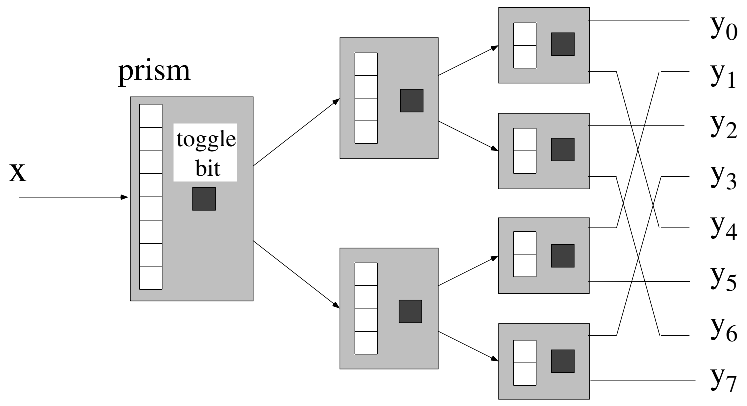


Figure 5: Diffracting balancer

```
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]                       (c)
     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]                       (d)
        else
           unlock(b->lock)
           return b->next[1]                       (e)
        endif
     endif
  endfor
```

Figure 6: Code For Traversing a diffracting balancer

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 manner using the fashionable load-linked/store-conditional operations available on standard architectures. Furthermore, the code of a diffracting tree uses two functions: (a) **random(i,j)** returns a random number between **i** and **j**; (b) **not_empty(i)** returns **TRUE** of **i** is the PID of some processor and **FALSE** otherwise. And the code translates to the following two phases. In phase 1, processor p announces the arrival of its token at balancer b, by writing b to **location**[p]. 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. 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. 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 succeed, the diffraction is successful, and p is diffracted to the **b− >next[0]** balancer. However, if the first compare-and swap fails, it follows that some other processor has already managed to diffract p, so it is directed to the **b− >next[1]** balancer. 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 it goes on toe phase 2.

In phase 2, processor p repeatedly checks to see if it has been diffracted by another processor, by spinning spin times on location[p]. Having given some other processor (the one that read its PID from prism) a chance to diffract it, p attempts to seize the toggle bit. If successful, it first clears its element of location, using compare-and swap, and then toggles the bit and exits the balancer. If the element could not be erased it follows that some other processor already collided with it, and it exits and balancer, being diffracted to b− >next[1]. If the toggle bit could not be seized, the process resumes spinning.

Notice that before accessing the toggle bit or trying to diffract, p clears **location**[p] using compare-and-swap. The use of compare-and-swap operations guarantees that the same processor, p, will not be diffracted twice, and that it will not be diffracted before it gets a chance to exit the balancer. This protect us from situations where some processor q is diffracted by p without noticing. The construction works because it assure 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.

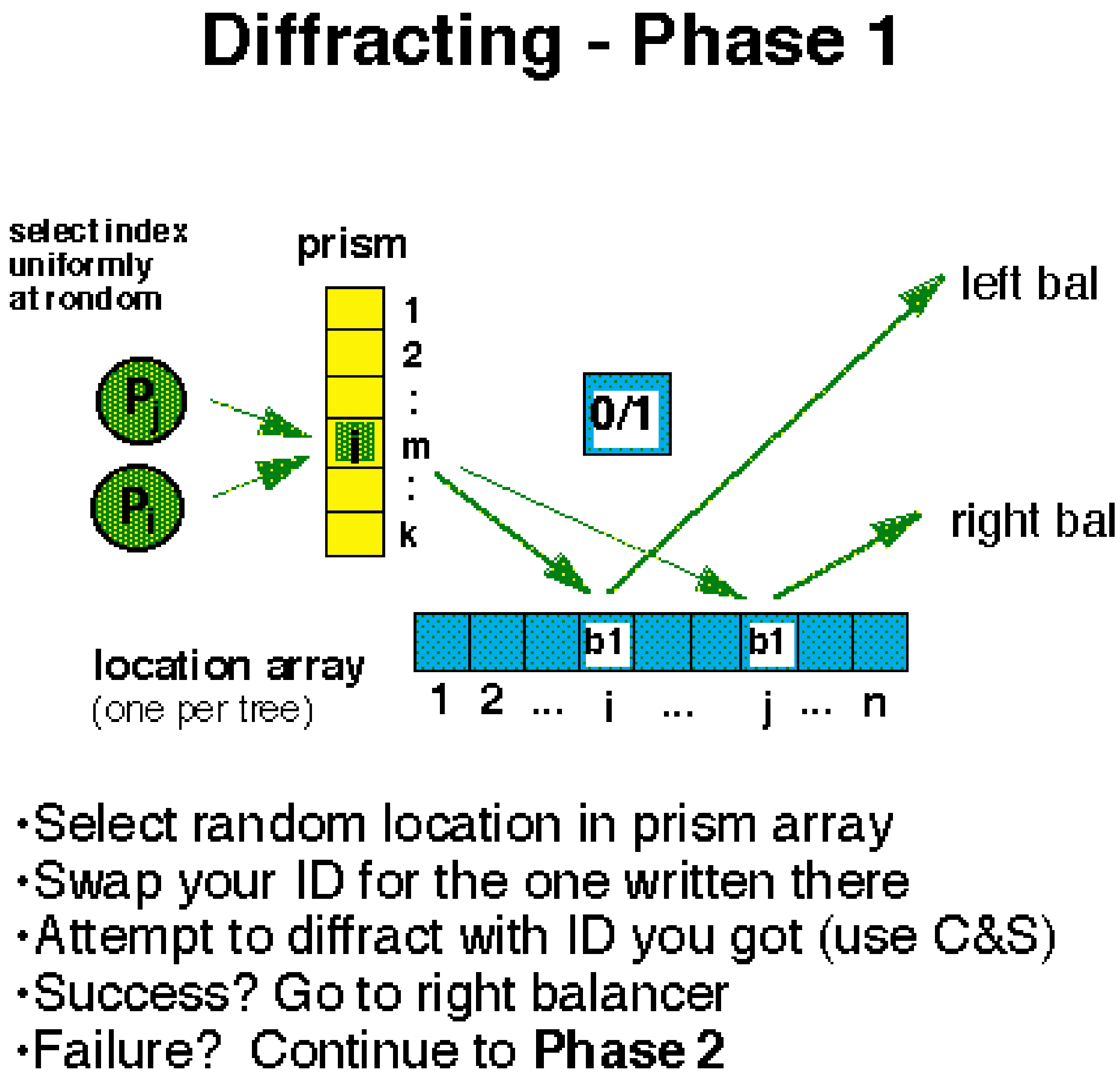


Figure 7: Phase 1

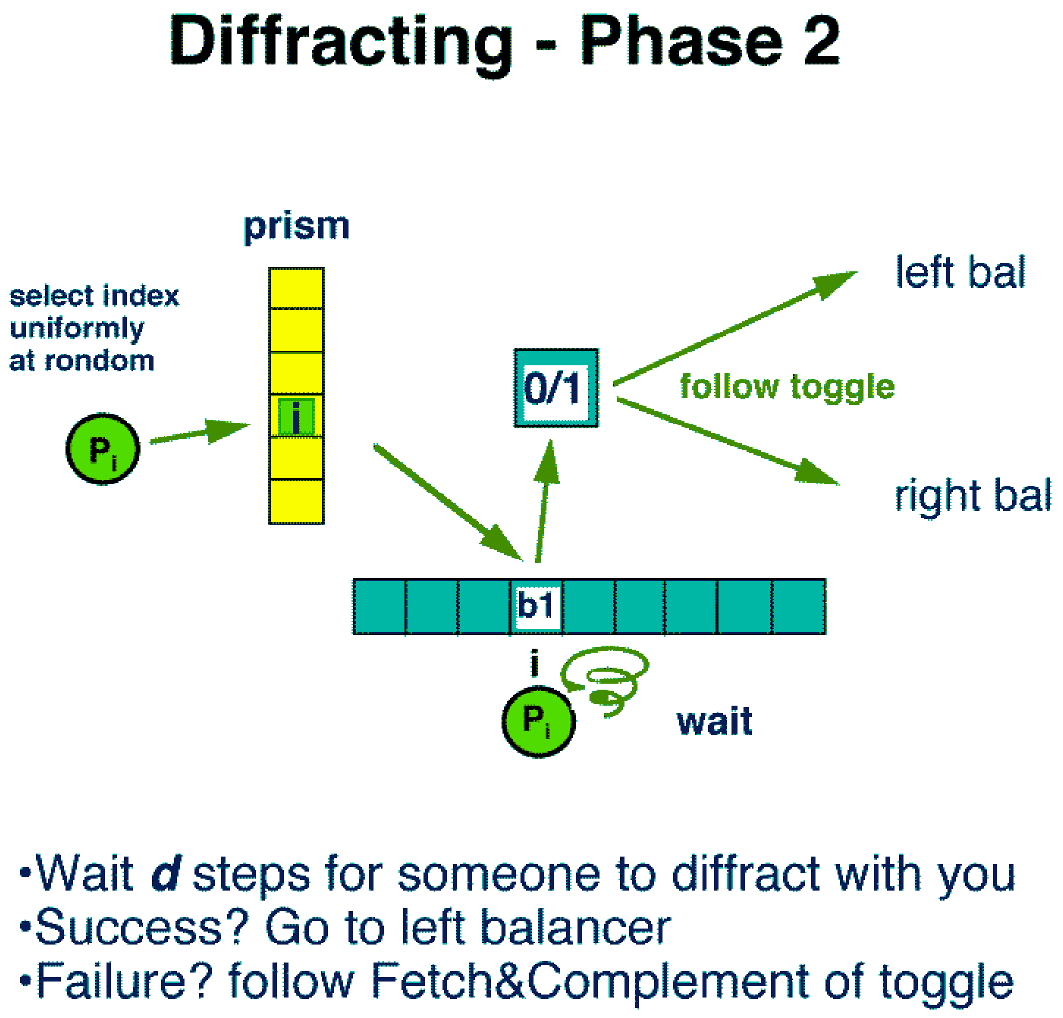


Figure 8: Phase 2

4.1 Some Implementation Details

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

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 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.

5 Performance

The performance of diffracting trees relative to other known methods was evaluated by running a collection of benchmarks on a simulated distributed-shared-memory multiprocessor similar to the MIT *alewife* machine. Two benchmarks were used to test the performance of diffracting trees: index-distribution and job queues.

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, were processors attempt to load a shared counter to full capacity. The date 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 included the **delay( )** time. 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: $10^6 d/t$.

Three software counting techniques mentioned in the class were used to measure the performance.

CTree *Fetch&Inc* using an optimal depth combing tree. Optimal width means that when n processors participate in the simulation, a tree of width $n/2$ will be used.

CNet A standard BITONIC counting network of width 64 with two-input and two-output balancers. The toggle bit was implemented in the standard way using a short critical section.

DTree A diffracting tree of width 32.

Figures 9 and 10 show that diffracting trees give consistently better throughput than the other methods and that in terms of latency they scale extremely well, average latency is

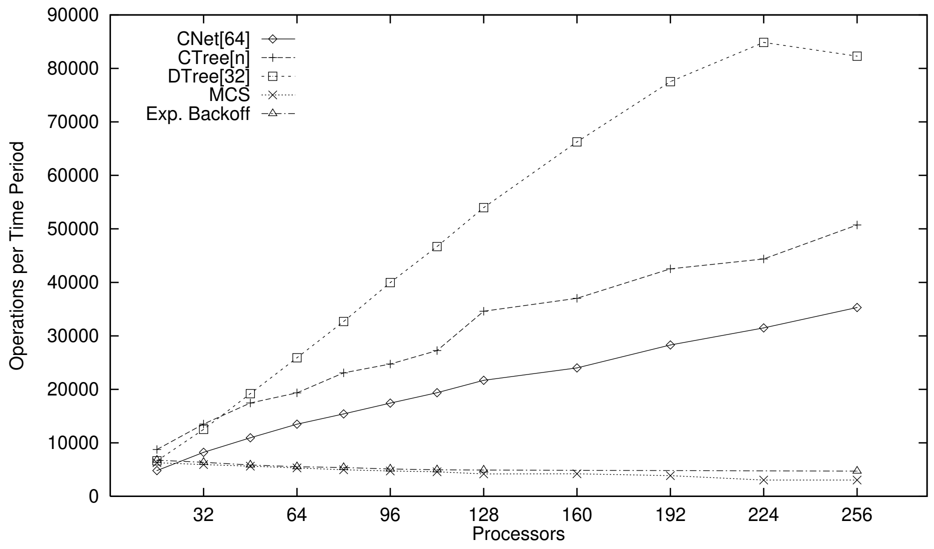


Figure 9: Throughput of various counting methods when work=0

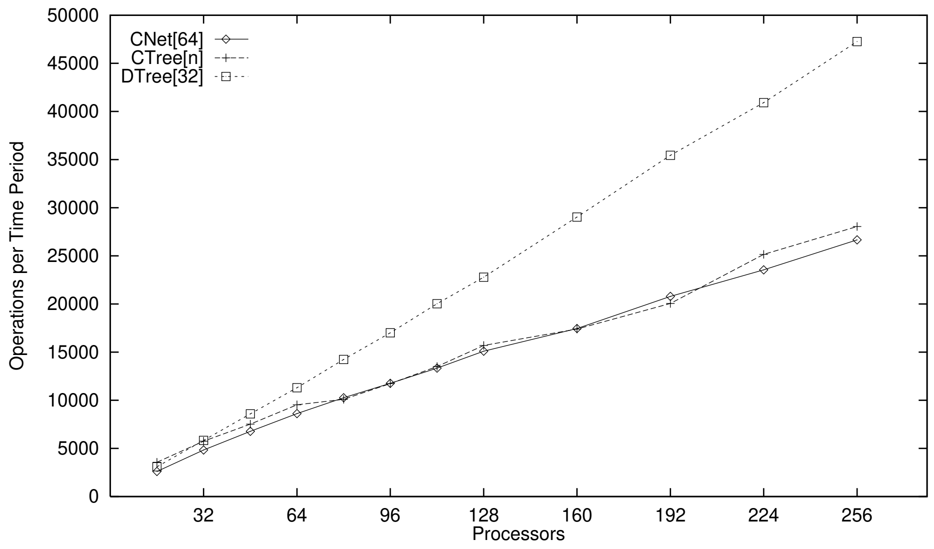


Figure 10: Latency of various counting methods when work=0

unaffected by the level of concurrency. While processors that fail to combine in a combining tree must waste cycles waiting for earlier processors to ascent the tree, processors in a diffracting tree proceed in an almost uninterrupted manner due the the high rate of collisions in the **prism** array. The scalable latency of diffracting tree is due to the constant level of contention on the toggle bit as concurrency raises and their low depth. While optimal combining trees have a depth of log n where n is the number of processes, and counting networks have a fixed depth of $1/2 \log^2 w + 1/2 \log w$ where w is the width of the network. Diffracting tress have a depth of only $\log w$, and high diffraction implies that most processor do traverse the tree in this number of steps i.e. with very little waiting. For example, with 256 processors, the combining tree rises to depth 8, the width 64 counting networks have depth 21, whereas a suitable diffracting tree has a depth of only 5.

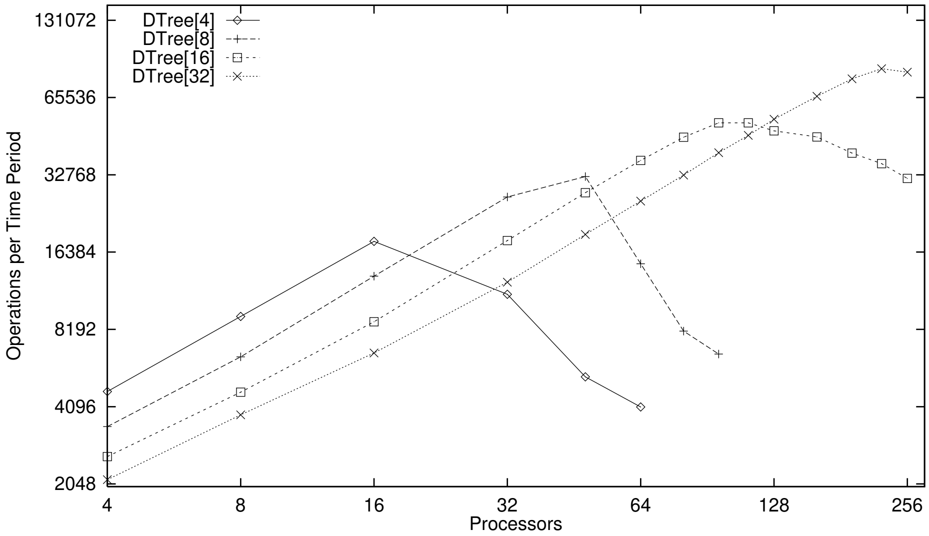


Figure 11:

Figure 11 shows the relationship between diffracting tree size and performance. Choosing a size that is too small or too wide can have negative effects. However, since the interval in which a given width is optimal is increasingly large in most cases the wide tree can be used without fear.

In summary, diffracting tress enjoy both the parallelism of counting networks and the high coordination of combining trees. They outperform both combing tress and counting networks on a simulated distributed shared-memory multiprocessor. Like counting networks and unlike combining trees, they can be made lock-free, that is, guarantee progress even if processors fail.

6 Message Passing Implementation

Message based diffracting trees have potential applications for load balancing and index distribution in a wide range of network architectures, ¿from tightly couple multiprocessors to LANs. The implementation of diffracting trees in a message passing environment is 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 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 a balancer.

6.1 Measuring Performance

The performance of the message passing diffracting trees was tested in simulated network environments, and four types of networks made up of processors, wired and switches were used. Messages are sent by processors, along wired 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.

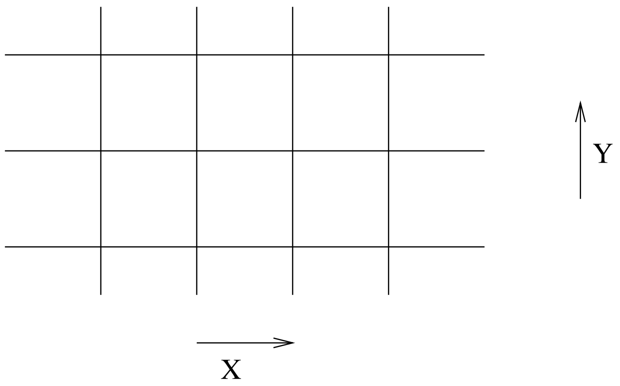


Figure 12: A mesh network

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 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.

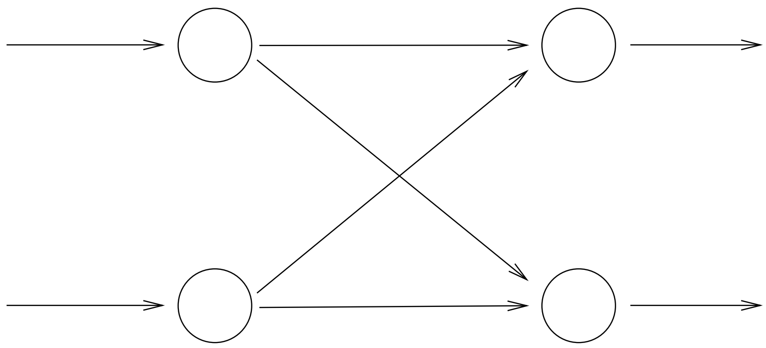


Figure 13: A crossbar switch

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.

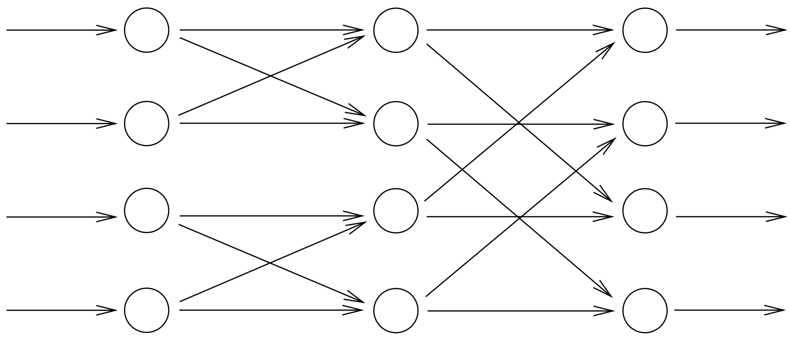


Figure 14: A butterfly network

Butterfly network In this architecture, 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, till 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 message with different sources and destinations to pass through at the same time. This network has a diameter of O(log n).

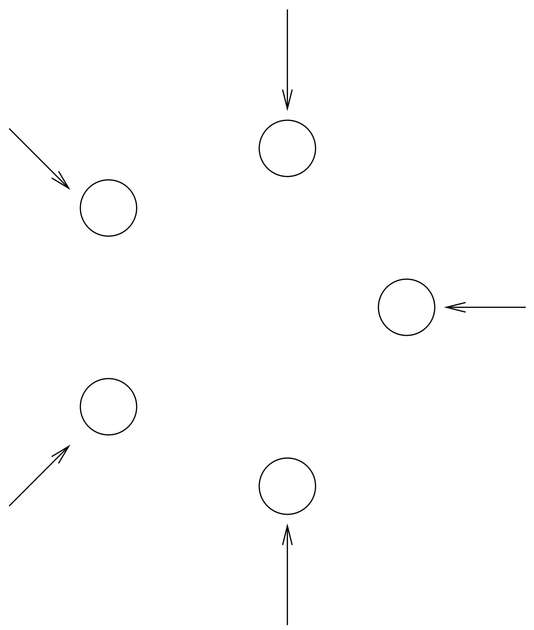


Figure 15: N x N crossbar network

$n \times n$ **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.

6.1.1 A Comparison of Network Topologies

	Low Locality	High Locality
Low Bandwidth	butterfly network	mesh with single wire switches
High Bandwidth	$n \times n$ crossbar	mesh with crossbar switches

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 tress proved the most efficient method in "non-localized" butterfly style networks where locality is not a factor.

6.1.2 Choosing a Waiting Policy

Nodes of a combing tree or **prism** processors in a diffracting tree delay arriving messages to create a time interval in which combining or diffraction can occur. Figure 16 compares combing tree latency when **work** is high, under 3 waiting policies: wait 16 cycles, wait 256 cycles and wait indefinitely. When the number of processors is large than 64, indefinite waiting is by far the best policy. Uncombined message cause the locking of a node until indices are returned, so a large performance penalty is paired for each such message. Because the chances of combing are good at higher arrival rates we found that when **work=0**, simulation using more than four processors justify indefinite waiting, so we used that policy for all combining trees.

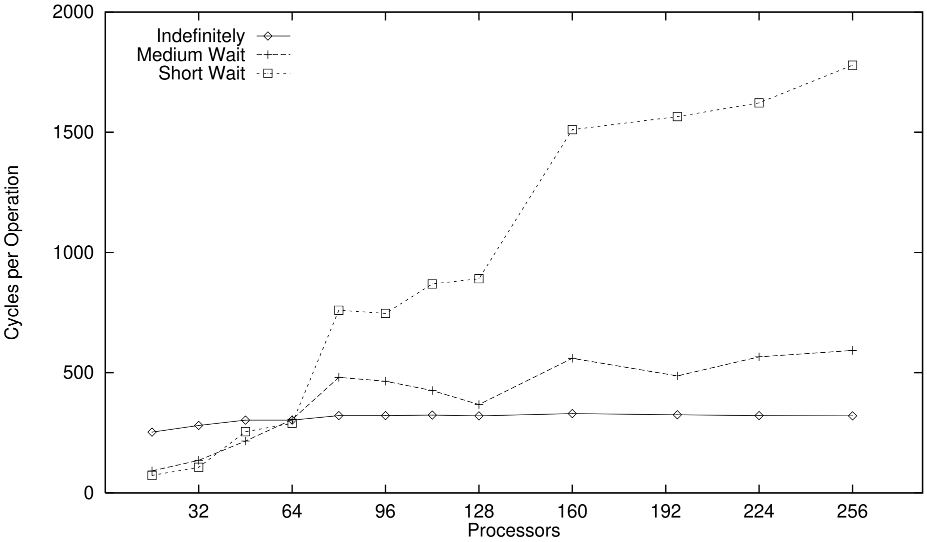


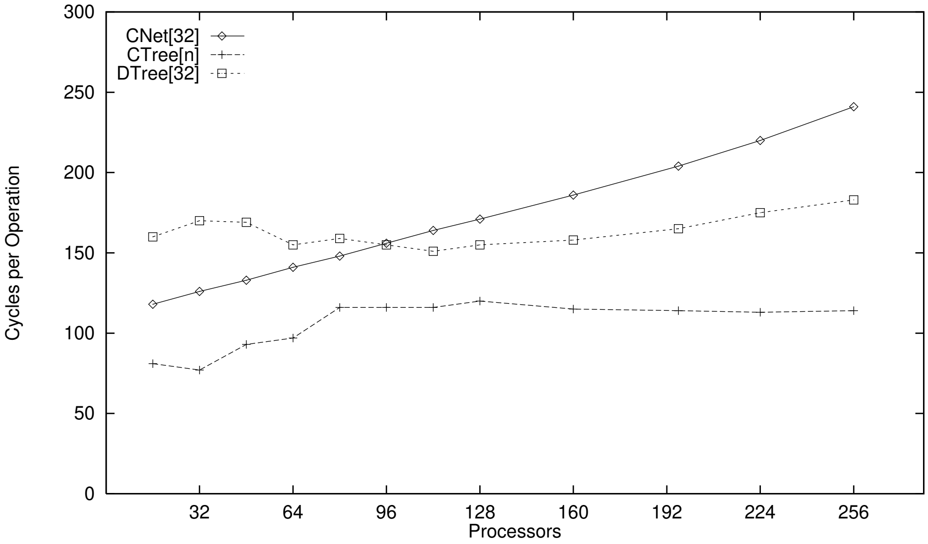
Figure 16:

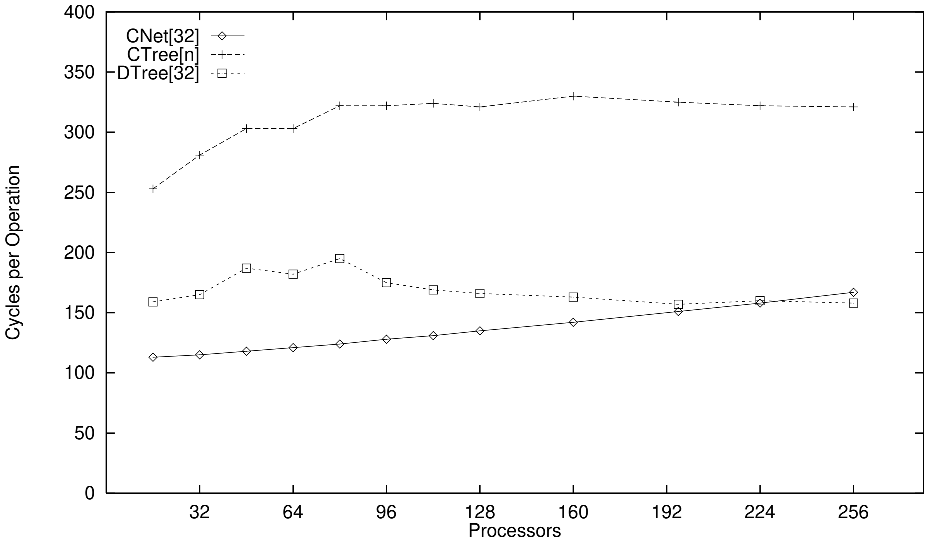
In diffracting tree, high loads favor waiting. However, when arrival rates are low, as in the case when **work** is high or the number or processors in the simulation is small, prism processors should expedite the sending of messages to the toggle processor to reduce latency. As in the shared memory implementation, the best diffracting tree performance was attained when using an adaptive policy to update token delay time as a function of concurrency. THe thread 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 the result of low load, the **spin** time is halved.

6.1.3 Robustness

Combing trees proved to be the least robust of the counting methods we studied and diffracting tress the most robust. For coming trees, as the range of **work** between counter accesses grew, variations in the arrival rates of requests made combing more difficult, and performance degraded. A dramatic example of this can be seen in the tests on the torus mesh network with single wire switches. The need to wait for latecoming processors causes a significant rise in latency which in turn lowers throughput.

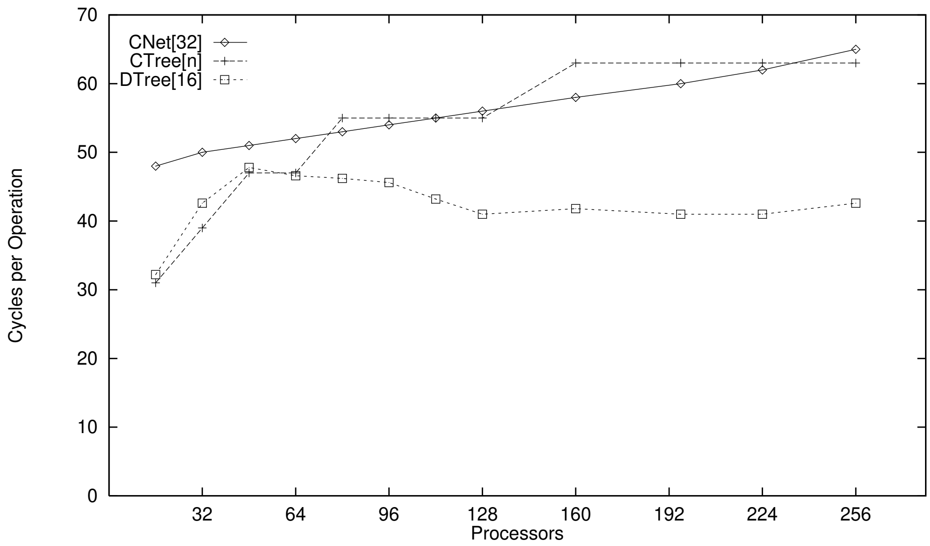
Fluctuations in request arrival times have a lesser effect on diffracting trees and counting networks. The above figures show that for counting networks lower load leads to less contention, latency still raises as concurrency increases, albeit more slowly. In diffracting





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 and transforming into regular balancers that take two messages to traverse.

In terms of 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.



On the other band, combining and diffracting trees make use of the high arrival rate to combine/diffract messages, utilizing the added congestion to increase parallelism. Combing trees handle concurrency by increasing depth, which adds latency with each new level. Diffracting trees are more scalable: a single diffracting tree can often handle a wide range of concurrency levels with little or no performance penalty.

6.1.4 Performance: The Effects of Locality and Bandwidth

Combining tree layout can be optimized to take advantage of network locality. It sends relatively few messages per index delivered which is important of bandwidth is low. For these reasons, combing outperforms all other methods in the mesh network with single wire switches. While a counting network's layout can also be optimized (though to a lesser extent than combining trees), the dynamic flow patterns of diffracting trees make layout optimization much less effective. Figure 17 compares the performance of combing and diffracting trees, with and without layout optimization, and shows that combining trees are less robust-placing them randomly on the mesh causes a drop of nearly 56% in throughput.

Higher bandwidth lessens the need to conserve messages or shorten distances as the added bandwidth helps hide the effects of locality. In the mesh with 5×5 crossbar switches diffracting trees reap the benefits of lower depth: increased throughput and lower latency. Counting networks, like combing 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, cost per message is high and diffracting trees, having the lowest depth, substantially outperform the other methods. In the complete crossbar network, the added bandwidth reduces the cost of messages and all three

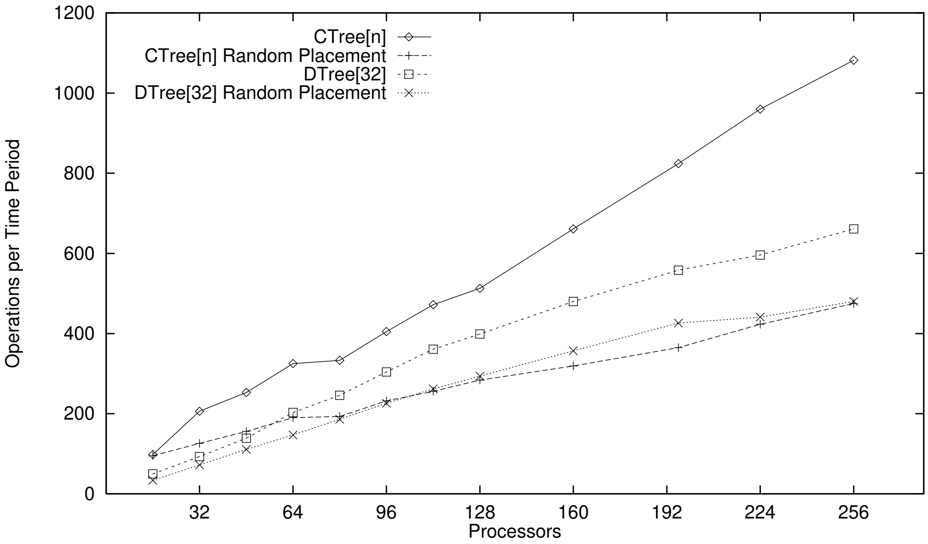


Figure 17:

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 networks, 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. The following tables summarize the optimized widths of the constructions.

Diffracting Tree

	Low Locality	High Locality
Low Bandwidth	8	32
High Bandwidth	16	16

Counting Network

	Low Locality	High Locality
Low Bandwidth	16	32
High Bandwidth	32	32

References

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

[2] 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, New Orleans, Louisiana.