The Art of Multiprocessor Programming

Maurice Herlihy Nir Shavit

July 17, 2006

Contents

Chapter 12

Parallel Counting

12.1 Introduction

A *pool* is a set of objects. As illustrated in Figure 12.1, the Pool interface provides put() and remove() methods that respectively insert and remove items from the pool. Familiar classes such as stacks and queues can be viewed as pools that provide additional ordering guarantees.

One natural way to implement a shared pool is to have each method lock the pool, perhaps by making both put() and remove() synchronized methods. The problem, of course, is that such an approach is too heavy-handed, because the lock itself would become a "hot-spot", that is, a locus of contention. We would prefer an implementation in which Pool methods could work in parallel.

Consider the following alternative. The pool itself is an array of items, where each array element either contains an item, or is **null**. We route threads through two counters. Threads calling put() increment one counter

[0] *Portions of this work are from the book* The Art of Multiprocessor Programming *by Maurice Herlihy and Nir Shavit, published by Morgan Kaufmann Publishers, Copyright 2006 Elsevier Inc. All rights reserved..*

```
public interface Pool {
  public void put(Object x);
  public Object get();
}
```

Figure 12.1: *Pool interface*

to choose an array index into which the new item should be placed. (If that slot is full, the thread waits until it becomes empty.) Similarly, threads calling remove() increment another counter to choose an array index from which the new item should be removed. (If that slot is empty, the thread waits until it becomes full.)

This approach replaces one bottleneck, the lock, with two, the counters. Naturally, two bottlenecks are better than one (think about that claim for a second). But are shared counters necessarily bottlenecks? Can we implement highly-concurrent shared counters? We face two challenges.

1. Can we avoid *memory contention*, where too many threads try to access the same memory location, stressing the underlying communication network and cache coherence protocols?

2. Can we achieve real parallelism? Is incrementing a counter an inherently sequential operation, or is it possible for n threads to increment a counter in time less than it takes for one thread to increment a counter n times?

Queue locks, as seen in Chapter [**?**], alleviate contention, but they cannot provide parallelism, since they force all counter operations to be sequential. Instead, in this chapter we look at two alternative ways to build highly concurrent counters: *combining trees* and *counting networks*.

12.2 Combining Trees

A *combining tree* is a tree of nodes, where each node contains bookkeeping information. The counter's value is stored at the root. Each thread is assigned a leaf, and at most two threads share a leaf. To increment the counter, a thread starts at its leaf, and works its way up the tree to the root. If two threads reach a node at approximately the same time, then they *combine* their increments by adding them together. One thread, the *first* thread, propagates their combined increments up the tree, while the other, *second* thread, waits for the first thread to complete their combined work.

For example, suppose threads A and B share a leaf node. They start at the same time, and their increments are combined at their shared leaf. The first one, say, B, actively continues up to the next level, with the mission of adding 2 to the counter value, while the second, A, waits for B to return from the root with an acknowledgment that A's increment has occurred.

At the next level in the tree, B may combine with another thread C, and advance with the renewed intention of adding 3 to the counter value.

When a thread reaches the root, it adds the sum of its combined increments to the counter's current value. The thread then moves back down the tree, notifying each waiting thread that the increments are now complete.

Combining trees have an inherent disadvantage with respect to locks: each increment has higher *latency*, that is, the time it takes an individual operation to complete. Using a lock, a single getAndIncrement() call takes $O(1)$ time, while using a combining tree, a single getAndIncrement() call takes $O(\log n)$ time. Nevertheless, combining trees are attractive because they promise far better *throughput*, that is, the overall rate at which operations complete. For example, using a queue lock, n getAndIncrement() calls complete in $O(n)$ time, at best, while using a combining tree, under ideal conditions where all threads move up the tree together, n getAndIncrement() calls will complete in $O(\log n)$ time, which is much better than $O(n)$. Of course, actual performance is often less than ideal, a subject we will examine in detail later on, but we emphasize that combining trees, like other techniques we consider later, are intended to increase throughput, not latency.

Combining trees are also attractive because they can be adapted to provide a parallel implementation, not just of getAndIncrement(), but of any read-modify-write method.

12.2.1 Overview

Although the idea behind a combining tree is quite simple, the implementation may, at first glance, seem complicated. To keep the overall (simple) structure from being submerged in (not-so-simple) detail, we split the data structure into two classes: the Tree class manages navigation within the tree, moving up and down the tree as needed, while the CNode class manages each "visit" to a node.

This algorithm uses two kinds of synchronization. Short-term synchronization is provided by synchronized methods of the CNode class. Each method locks the node for the duration of the call to ensure that it can read and write node fields without interference from other threads. The algorithm also requires excluding threads from a node for durations longer than a single method call Such long-term synchronization is provided by a Boolean locked field. When this field is *true*, no other thread is allowed to access the node.

Every tree node has a *combining status*, which defines whether the node is in the early, middle, or late stages of combining concurrent requests.

```
public class Node {
  enum CStatus{IDLE, FIRST, SECOND, RESULT, ROOT};
  boolean locked;  // is node locked?
  CStatus cStatus;  // combining status
  int firstValue , secondValue; // values to be combined
  int result ;        // result of combining
  Node parent;        // reference to parent
   ...
}
```

Figure 12.2: *CNode class fields*

```
enum CStatus{FIRST, SECOND, RESULT, IDLE, ROOT};
```

These values mean the following:

- *FIRST*: One thread has visited this node, and will return to check whether another thread has left a value with which to combine.
- *SECOND*: A second thread has visited this node and stored a value in the node's value field to be combined with the first thread's value, but the combined operation is not yet complete.
- *RESULT*: The two thread's operations have been combined and completed, and the second thread's result has been stored in the node's *result* field.
- *ROOT*: This is a special case to indicate that the node is the root, and must be treated specially.

Figure 12.2 shows the CNode class's other fields.

The combining tree's getAndIncrement() method, shown in Figure 12.3, has four phases. In the *precombining phase* (lines 5 through 8), the Tree class's getAndIncrement() method moves up the tree applying the *precombine()* method to each node. The *precombine()* method returns a Boolean value indicating whether the caller should continue moving up the tree. The *stop* variable is set to the last node visited..

The *precombine()* method is shown in Figure 12.4. Suppose this method is called by thread A. In line 2, A waits until the synchronization status is *FREE*. In line 3, it tests the combining status.

```
1  public int getAndIncrement() {
2    Stack<Node> stack = new Stack<Node>();
3    Node myLeaf = leaf[ThreadID.get() / 2];
4    Node node = myLeaf;
5    // precombining phase
6    while (node.precombine()) {
7      node = node.parent;
8    }
9    Node stop = node;
10   // combining phase
11   node = myLeaf;
12   int combined = 1;
13   while (node != stop) {
14     combined = node.combine(combined);
15     stack.push(node);
16     node = node.parent;
17   }
18   // operation phase
19   int prior = stop.op(combined);
20   // distribution phase
21   while (!stack.empty()) {
22     node = stack.pop();
23     node.distribute(prior);
24   }
25   return prior;
26 }
```

Figure 12.3: *The Tree class's getAndIncrement() method*

- *IDLE*: A sets the status to *FIRST* to indicate that it will return to look for a value for combining. If it finds such a value, A will proceed as the active thread, and the thread that provided that value will be passive. A then returns *true*, instructing the caller to move up the tree.

- *FIRST*: An earlier thread B has recently visited this node, and will return to look for a value to combine. A instructs the caller to stop moving up the tree (by returning *false*), and to start the next phase, computing the value to combine with B. Before it returns, A places

```
1  synchronized boolean precombine() {
2    while (locked) wait();
3    switch (cStatus) {
4      case IDLE:
5        cStatus = CStatus.FIRST;
6        return true;
7      case FIRST:
8        locked = true;
9        cStatus = CStatus.SECOND;
10       return false;
11     case ROOT:
12       return false;
13     default:
14       throw new PanicException("unexpected Node state " + cStatus);
15   }
16 }
```

Figure 12.4: *The precombining phase*

a long-term lock on the node (by setting locked to *true*) to prevent B from proceeding without combining with A's value.

- *ROOT*: If A has reached the root node, it instructs the caller to start the next phase.

Line 13 is a *default* case that is executed if an unexpected status is encountered. It is good programming practice always to provide an arm for every possible enumeration value, even if you "know" it cannot happen. If your are wrong, the program will be easier to debug, and even if you are right, the program may later be changed by someone who does not know as much as you. Always program defensively.

In the *combining phase*, (Figure 12.3, lines 10 through 17), A revisits the nodes it visited in the precombining phase, combining its value with values left by other threads. It stops when it arrives at the node *stop* where the precombining phase ended. Later on, we will need to traverse these nodes in reverse order, so as we go we push the nodes we visit onto a stack.

The CNode class's *combine()* method, shown in Figure 12.5, adds any values left by a recently-arrived passive process to the values combined so far. As before, A first waits until the locked field is *false*. It then sets a long-term lock on the node, to ensure that late-arriving threads do not expect to

```
synchronized int combine(int combined) {
  while (locked) wait();
  locked = true;
  firstValue = combined;
  switch (cStatus) {
    case FIRST:
      return firstValue;
    case SECOND:
      return firstValue + secondValue;
    default:
      throw new PanicException("unexpected Node state " + cStatus);
  }
}
```

Figure 12.5: *The combining phase*

combine with it. If the status is *SECOND*, A adds the other thread's value to the accumulated value, and otherwise it returns the value unchanged.

At the start of the operation phase (lines 18 and 19), A has now combined all method calls from lower-level nodes, and now examines the node where it stopped at the end of the precombining phase (Figure 12.6). If the node is the root, then A carries out the combined getAndIncrement() operations: it adds its accumulated value and returns the prior value. Otherwise, A unlocks the node, notifies any blocked thread, deposits its value and waits for the other thread to return a result after propagating the combined operations toward the root. When the result arrives, A enters the *distribution phase*, propagating the result down the tree.

In the distribution phase (lines 20 through 25), A moves down the tree, releasing locks, and informing passive partners of the values they should report to their own passive partners, or to the caller (at the lowest level).

Pragma 12.2.1 To initialize the combining tree, we create an array *node* of n CNode objects. The root is *node[0]*, and for $0 < i < n$, the parent of *node[i]* is *node[(i-1)/2]*. The leaf nodes are the last $(n+1)/2$ nodes in the array, where thread i is assigned to leaf $i/2$. Figure 12.7 shows the Tree class constructor.

```
synchronized int op(int combined) {
  switch (cStatus) {
    case ROOT:
      int oldValue = result;
      result += combined;
      return oldValue;
    case SECOND:
      secondValue = combined;
      locked = false;
      notifyAll (); // wake up waiting threads
      while (cStatus != CStatus.RESULT) wait();
      locked = false;
      notifyAll ();
      cStatus = CStatus.IDLE;
      return result;
    default:
      throw new PanicException("unexpected Node state");
  }
}
```

Figure 12.6: *The operation phase*

```
public Tree(int size) {
  Node[] nodes = new Node[size - 1];
  nodes[0] = new Node();
  for (int i = 1; i < nodes.length; i++) {
    nodes[i] = new Node(nodes[(i-1)/2]);
  }
  leaf = new Node[(size + 1)/2];
  for (int i = 0; i < leaf.length; i++) {
    leaf[i] = nodes[nodes.length - i - 1];
  }
}
```

Figure 12.7: *Constructor for Tree class*

12.2.2 Performance

It is not at all clear how well combining trees work in practice. As we noted, they do very well under ideal circumstances when every thread's operation

```
void indexBench(int iters, int work) {
 AtomicInteger index = new AtomicInteger(0);
 while (true) {
  index.getAndIncrement();
  Thread.sleep(random() % work);
 }
}
```

Figure 12.8: *Index distribution benchmark*

Figure 12.9: *figure:index*

combines with another. The use of locking, however, means that if one thread fails to combine its operation, it may be blocked waiting for a tree node to be unlocked. Here, we report the results of an experiment testing the combining performance on a simulated 64-processor cache-coherent NUMA architecture. The experiment used the *index distribution benchmark*, shown in Figure **??**. Here, threads apply getAndIncrement() to a shared counter, and use that counter to choose some work to do. This kind of load-balancing is well-suited for a task such as rendering the Mandelbrot Set. Each thread repeatedly picks a rectangle in the image. Because rectangles are independent of one another, they can be rendered in parallel, but because some rectangles take unpredictably longer than others, load-balancing must be dynamic. When a thread runs out of work, it asks for another rectangle, so all threads are busy doing something useful.

In this benchmark, the level of contention for the counter is controlled by adjusting the *work* parameter: the smaller the value, the higher the contention at the counter.

Not surprisingly, combining trees perform best when contention is high. The higher the concurrency, the greater the combining, and the greater the speed-up. Combining trees are less attractive when concurrency is low. At one extreme, if only one thread at a time increments the counter, then it traverses $\log n$ nodes, where n, the maximum number of threads that might ever try to increment the counter, may be much larger than necessary.

Combining Rates The simulation was instrumented to monitor how often requests were combined expressed as a function of the *work* parameter. Figure 12.10 displays the percentage of the arrivals at combining tree nodes that succeed in combining with some other arrival in the index distribution

and job queue benchmarks.

The results show that the rate of combining decreases rapidly as the arrival rate increment requests is reduced. When an increment request misses a chance to combine, it must wait for the earlier request to ascend and descend the tree before it can progress. The effectiveness of combining trees is therefore sensitive to the arrival rate of requests.

Waiting If two requests arrive at a node at about the same time, they are combined, and both increment requests proceed concurrently. If, however, those requests are a little bit off, then the later one must wait for the earlier one to ascend and descend the tree, so they occur sequentially. The *rate* at which method calls combine is central to the performance of combining trees. High combining rates are good, low combining rates are bad.

Since combining increases concurrency, and failure to combine does not, it makes sense for a request arriving at a node to wait for some duration in the hope that another thread will arrive with a request that can be combined. Figure **??** displays combining tree latency (for a high *work* parameter) under three waiting policies: wait 16 cycles, wait 256 cycles, and wait indefinitely. When the number of threads is larger than 64, indefinite waiting is by far the best policy. This follows since a request that moves up the tree without combining holds up later requests for the duration of the traversal. Because the chances of combining are good when contention is high, the simulation showed that when *work* is 0, indefinite waiting is the best strategy for four or more threads.

Robustness An algorithm is *robust* if it performs well under a variety of conditions, such as different work loads, or large variance in request arrival times. Combining trees are not particularly robust. The simulation analyzed robustness in the face of load fluctuations (`work = 500`). For combining trees, as the range of `work` between counter accesses grew, variations in the arrival rates of requests made combining more difficult, and performance degraded.

Combining trees perform best when contention is highest. The higher the level of concurrency, the greater the amount of combining, and the greater the speed-up. On the other hand, combining trees are not as attractive when contention is low.

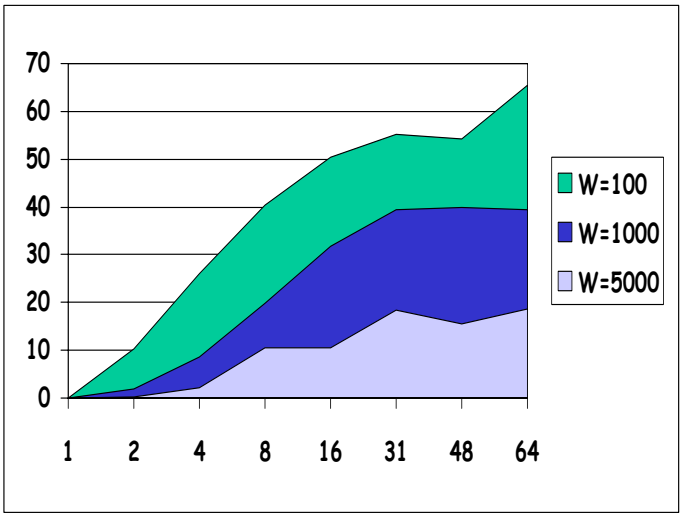
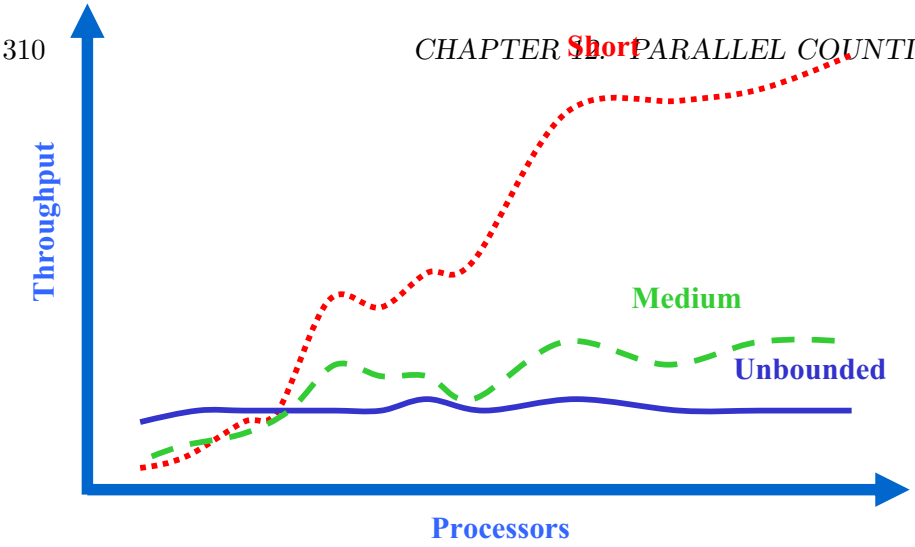


Figure 12.10: *Combining rate (as a percentage) at combining tree nodes in the index distribution benchmark.*

12.3 Counting Networks

A *balancer* is a simple switch with two input wires and two output wires, called the *top* and *bottom* wires (or sometimes the *north* and *south* wires).



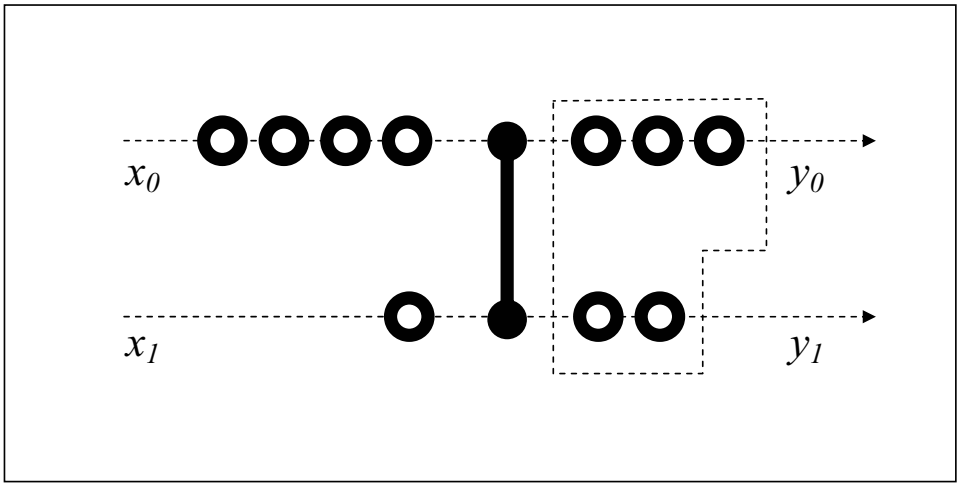


Figure 12.11: *A balancer*

Tokens arrive on the balancer's input wires at arbitrary times, and emerge on its output wires, at some later time. A balancer is a toggle: given a stream of input tokens, it alternates sending one token to the top output wire, and one to the bottom, effectively balancing the number of tokens between the two wires (see Figure 12.11). More precisely, a balancer has two states: *up* and *down*. If the state is *up*, the next token exits on the top wire, otherwise it exits on the bottom wire.

We use x_0 and x_1 to denote the number of tokens that respectively arrive on a balancer's top and bottom input wires, and y_0 and y_1 to denote the number that exit on the the top and bottom output wires. A balancer never

creates tokens: at all times,

$$x_0 + x_1 \geq y_0 + y_1.$$

A balancer is *quiescent* if every token that arrived on an input wire has emerged on an output wire:

$$x_0 + x_1 = y_0 + y_1.$$

A *balancing network* is constructed by connecting some balancers' output wires to other balancers' input wires. A balancing network of width w has input wires $x_0, x_1, .., x_{w-1}$ (not connected to output wires of balancers), and w output wires $y_0, y_1, .., y_{w-1}$ (similarly unconnected). The counting network's *depth* is the maximum number of balancers one can traverse starting from any input wire. We consider only balancing networks of finite depth (meaning the wires do not form a loop). Like balancers, balancing networks do not create tokens:

$$\sum x_i \geq \sum y_i,$$

(We usually drop indexes from summations when we mean to sum over every element in a sequence.) A balancing network is *quiescent* if every token that arrived on an input wire has emerged on an output wire:

$$\sum x_i = \sum y_i.$$

So far, we have described counting networks as if they were switches in a network. On a shared-memory multiprocessor, however, a balancing network can be implemented as an object in memory. Each balancer is an object, whose wires are references from one balancer to another. Each thread repeatedly traverses the object, starting on some input wire, and emerging at some output wire, effectively shepherding a token through the network (see section **??**).

Some balancing networks have unusual properties. The network shown in Figure 12.12 has four input wires and four output wires. Initially, all balancers are *up*. You can check for yourself that if any number of tokens enter the network, in any order, on any set of input wires, then they emerge in a regular pattern on the output wires. Informally, no matter how token arrivals are distributed among the input wires, the output distribution is balanced across the output wires, with the top output wires are "filled" first. So if the number of tokens is a multiple of four, then the same number of tokens emerges from each wire. If there is one excess token, it emerges on

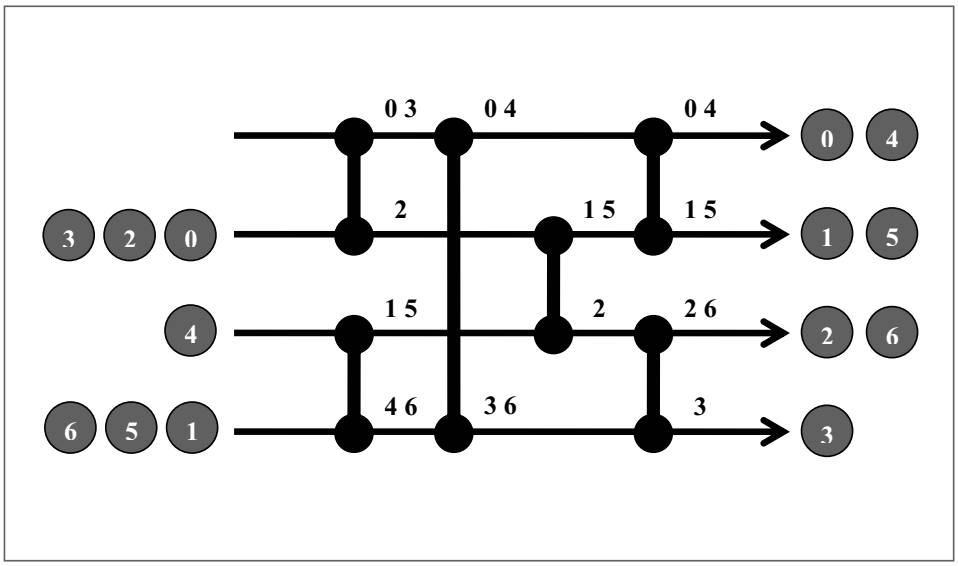


Figure 12.12: *A sequential execution for a* BITONIC [4] *counting network.*

output wire 0, if there are two, they emerge on output wires 0 and 1, and so on. In general, if

$$N = \sum x_i$$

then

$$y_i = (N/w) + (i \bmod w).$$

We call this property the *step property.*

Any balancing network that satisfies the step property is called a *counting network*, because it can easily be adapted to count the number of tokens that have traversed the network. Counting is done by adding a "local counter" to each output wire i, so that tokens emerging on wire are assigned consecutive numbers $i, i + w, \ldots, i + (y_i - 1)w$.

The step property can be defined in a number of ways which we will use interchangeably.

Lemma 12.3.1 *If $y_0, \ldots, y_{w-1}$ is a sequence of non-negative integers, the following statements are all equivalent:*

1. *For any $i < j$, $0 \leq y_i - y_j \leq 1$.*
2. *Either $y_i = y_j$ for all i, j, or there exists some c such that for any $i < c$ and $j \geq c$, $y_i - y_j = 1$.*

3. *If* $m = \sum y_i$, $y_i = \lceil \frac{m-i}{w} \rceil$.

Somtimes we will be interested in a weaker property: a sequence $X = x_0, x_1 \ldots x_{n-1}$ is *k-smooth* if $|x_i - x_j| \leq k$, for any $0 \leq i, j < n$. If X has the step property, it is 1-smooth, but not vice-versa.

12.3.1 The Bitonic Counting Network

In this section we describe how to generalize the counting network of Figure 12.12 to a counting network whose width is any power of 2. We give an inductive construction.

Define the width-$2k$ balancing network MERGER$[2k]$ as follows. It has two sequences of inputs of length k, x and x', and a single sequence of outputs y, of length $2k$. If x and x' both have the step property, then so will y, in any quiescent state.

We define the MERGER$[2k]$ network inductively (see Figures 12.14 and 12.15). When k is equal to 1, the MERGER$[2k]$ network is a single balancer. For $k > 1$, we construct the MERGER$[2k]$ network with input sequences x and x' from two MERGER$[k]$ networks and k balancers. Using a MERGER$[k]$ network we merge the even subsequence $x_0, x_2, \ldots, x_{k-2}$ of x with the odd subsequence $x'_1, x'_3, \ldots, x'_{k-1}$ of x' (that is, the sequence $x_0, \ldots, x_{k-2}, x'_1, \ldots, x'_{k-1}$ is the input to the MERGER$[k]$ network) while with a second MERGER$[k]$ network we merge the odd subsequence of x with the even subsequence of x'. Call the outputs of these two MERGER$[k]$ networks z and z'. The final stage of the network combines z and z' by sending each pair of wires z_i and z'_i into a balancer whose outputs yield y_{2i} and y_{2i+1}.

The MERGER$[2k]$ network consists of $\log 2k$ layers of k balancers each. It provides the step property for its outputs only when its two input sequences also have the step property, which we will ensure by filtering the inputs through smaller balancing networks. Let BITONIC$[2k]$ be the network constructed by passing the outputs from two BITONIC$[k]$ networks into a MERGER$[2k]$ network, where the induction is grounded in the BITONIC$[2]$ network consisting of a single balancer. This construction gives us a network consisting of $\binom{\log 2k+1}{2}$ layers each consisting of k balancers.

A Software Bitonic Network

So far, we have So far, we have described counting networks as if they were switches in a network. On a shared-memory multiprocessor, however, a balancing network can be implemented as an object in memory. Each balancer is an object, whose wires are references from one balancer to another. Each

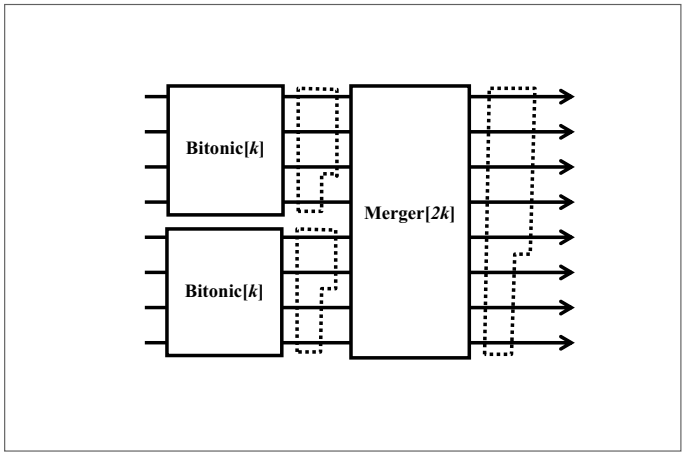


Figure 12.13: *A* Bitonic $[2k]$ *Counting Network*

thread repeatedly traverses the object, starting on some input wire, and emerging at some output wire, effectively shepherding a token through the

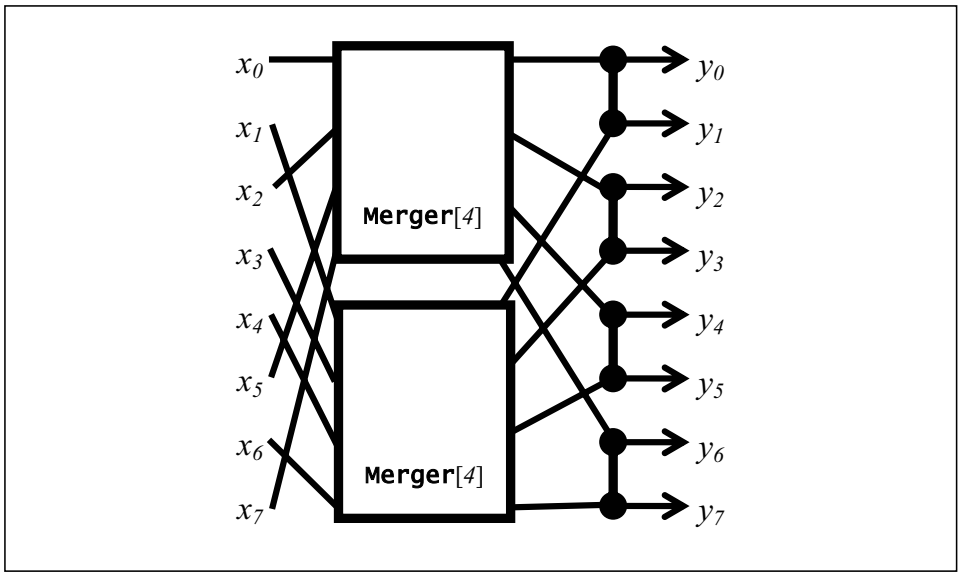


Figure 12.14: *Logical structure of* MERGER [8] *network.*

network. Here, we show how to implement a Bitonic network as a shared-memory data structure.

The Balancer class (Figure 12.16) has a single Boolean field: toggle. The synchronized traverse() method complements the toggle field and returns as output wire either 0 or 1. The Balancer class's traverse() method does not take an argument because the wire on which a token exits a balancer does not depend on the wire on which it enters.

The Merger class (Figure 12.17) has three fields: size is the size (which must be a power of 2), half[] is a two-element array of half-size merger objects, and layer[] is an array of size *size* of balancers implementing the final network layer. If the network has size 2, the half[] array is uninitialized. Otherwise, each element of half[] is initialized to a half-sized Merger network. The layer[] array is initialized so that layer[i] and layer[$size - i - 1$] refer to the same balancer.

The class provides a traverse(w) method, where the argument w is the wire on which the token enters. (For Merger networks, unlike balancers, a token's path depends on its input wire). If the token entered on the lower $size/2$ wires, then it passes through half[0], and otherwise half[1]. A token that emerges from the half-merger subnetwork on wire i then traverses the balancer at layer[i]. This logic is a dynamic expression of the static wiring

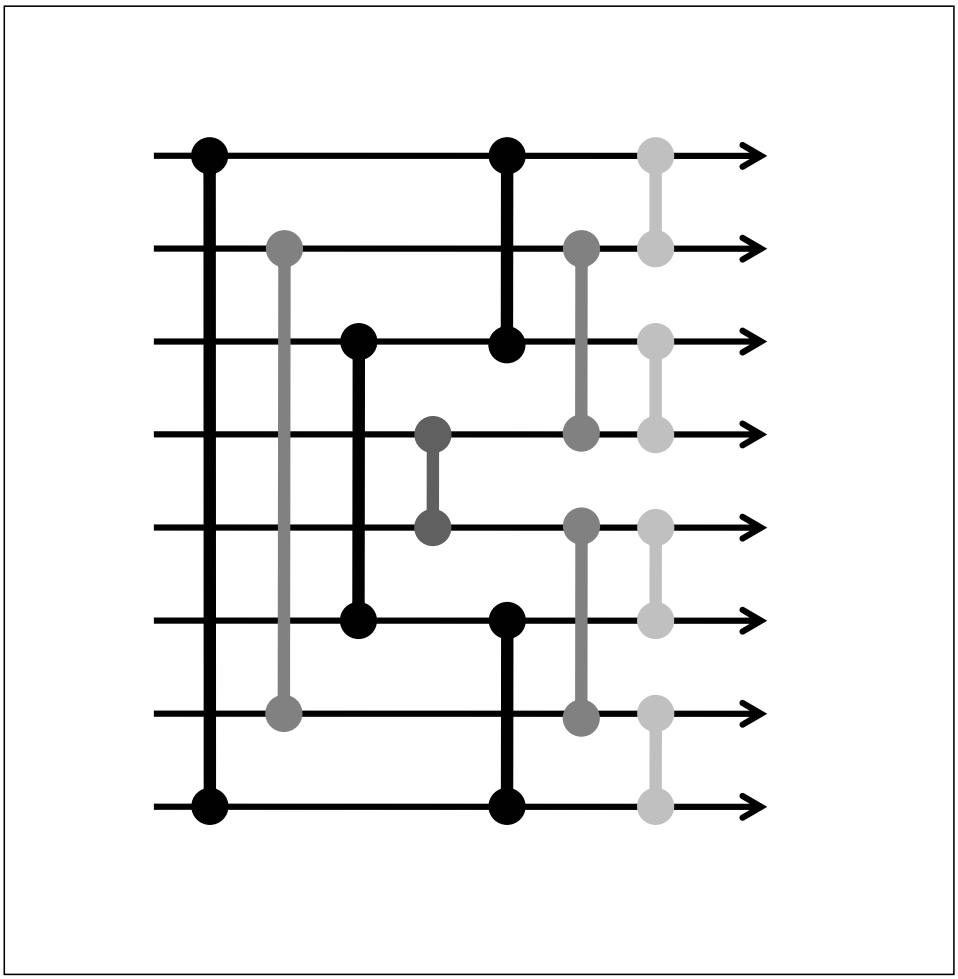


Figure 12.15: *Physical structure of* MERGER[8] *network.*

diagrams shown earlier.

The Bitonic class (Figure 12.18) also has three fields: size is the size (a power of 2), half[] is a two-element array of half-size Bitonic objects, and merger is Merger network of size *size*. If the network has size 2, the half[] array is uninitialized. Otherwise, each element of half[] is initialized to a half-sized Bitonic network. The merger[] array is initialized to a Merger network of size *size*.

The class provides a traverse(w) method, where the argument w is the wire on which the token enters. If the token entered on the lower *size*/2

```
public class Balancer implements Network {
  Boolean toggle = true;
  public synchronized int traverse(int input) {
    try {
      if (toggle) {
        return 0;
      } else {
        return 1;
      }
    } finally {
      toggle = !toggle;
    }
  }
}
```

Figure 12.16: *Balancer code*

wires, then it passes through half[0], and otherwise half[1]. A token that emerges from the half-merger subnetwork on wire i then traverses the final merger network from input wire i.

Proof of Correctness

We now show that BITONIC$[w]$ is a counting network. Before examining the network itself, here are some simple lemmas about sequences with the step property.

Lemma 12.3.2 *If a sequence has the step property, then so do all its subsequences.*

Lemma 12.3.3 *If $x_0, \ldots, x_{k-1}$ has the step property, then its even and odd subsequences satisfy:*

$$\sum_{i=0}^{k/2-1} x_{2i} = \left\lceil \sum_{i=0}^{k-1} x_i/2 \right\rceil \qquad \textit{and} \qquad \sum_{i=0}^{k/2-1} x_{2i+1} = \left\lfloor \sum_{i=0}^{k-1} x_i/2 \right\rfloor$$

Proof: Either $x_{2i} = x_{2i+1}$ for $0 \leq i < k/2$, or by Lemma 12.3.1 there exists a unique j such that $x_{2j} = x_{2j+1} + 1$ and $x_{2i} = x_{2i+1}$ for all $i \neq j$, $0 \leq i < k/2$. In the first case, $\sum x_{2i} = \sum x_{2i+1} = \sum x_i/2$, and in the second case $\sum x_{2i} = \lceil \sum x_i/2 \rceil$ and $\sum x_{2i+1} = \lfloor \sum x_i/2 \rfloor$. ■

```
public class Merger {
  Merger[] half;      // two half−size merger networks
  Balancer[] layer; // final layer
  final int size;
  public Merger(int _size) {
    size = _size;
    layer = new Balancer[size / 2];
    for (int i = 0; i < size / 2; i++) {
      layer[i] = new Balancer();
    }
    if (size > 2) {
      half = new Merger[]{new Merger(size/2), new Merger(size/2)};
    }
  }
  public int traverse(int input) {
    int output = 0;
    if (size > 2) {
      output = half[input % 2].traverse(input / 2);
    }
    return (2 * output) + layer[output].traverse();
  }
}
```

Figure 12.17: *The Merger class*

Lemma 12.3.4 *Let $x_0, \ldots, x_{k-1}$ and $y_0, \ldots, y_{k-1}$ be arbitrary sequences having the step property. If $\sum x_i = \sum y_i$, then $x_i = y_i$ for all $0 \leq i < k$.*

Proof: Let $m = \sum x_i = \sum y_i$. By Lemma 12.3.1, $x_i = y_i = \lceil \frac{m-i}{k} \rceil$. ■

Lemma 12.3.5 *Let $x_0, \ldots, x_{k-1}$ and $y_0, \ldots, y_{k-1}$ be arbitrary sequences having the step property. If $\sum x_i = \sum y_i + 1$, then there exists a unique j, $0 \leq j < k$, such that $x_j = y_j + 1$, and $x_i = y_i$ for $i \neq j$, $0 \leq i < k$.*

Proof: Let $m = \sum x_i = \sum y_i + 1$. By Lemma 12.3.1, $x_i = \lceil \frac{m-1}{k} \rceil$ and $y_i = \lceil \frac{m-1-i}{k} \rceil$. These two terms agree for all i, $0 \leq i < k$, except for the unique i such that $i = m - 1 \pmod{k}$. ■

We now show that the MERGER$[w]$ network preserves the step property.

```
public class Bitonic {
  Bitonic [] half; // two half−size bitonic networks
  Merger merger; // final merger layer
  final int size; // network size
  public Bitonic(int _size) {
    size = _size;
    merger = new Merger(size);
    if (size > 2) {
      half = new Bitonic[]{new Bitonic(size/2), new Bitonic(size/2)};
    }
  }
  public int traverse(int input) {
    int output = 0;
    if (size > 2) {
      output = half[input % 2].traverse(input / 2);
    }
    return merger.traverse(output);
  }
}
```

Figure 12.18: *The Bitonic class*

Lemma 12.3.6 *If* MERGER$[2k]$ *is quiescent, and its inputs* $x_0, \ldots, x_{k-1}$ *and* $x'_0, \ldots, x'_{k-1}$ *both have the step property, then its outputs* $y_0, \ldots, y_{2k-1}$ *have the step property.*

Proof: We argue by induction on $\log k$.

If $2k = 2$, MERGER$[2k]$ is just a balancer, so its outputs are guaranteed to have the step property by the definition of a balancer.

If $2k > 2$, let $z_0, \ldots, z_{k-1}$ be the outputs of the first MERGER$[k]$ subnetwork, which merges the even subsequence of x with the odd subsequence of x', and let $z'_0, \ldots, z'_{k-1}$ be the outputs of the second. Since x and x' have the step property by assumption, so do their even and odd subsequences (Lemma 12.3.2), and hence so do z and z' (induction hypothesis). Furthermore, $\sum z_i = \lceil \sum x_i/2 \rceil + \lfloor \sum x'_i/2 \rfloor$ and $\sum z'_i = \lfloor \sum x_i/2 \rfloor + \lceil \sum x'_i/2 \rceil$ (Lemma 12.3.3). A straightforward case analysis shows that $\sum z_i$ and $\sum z'_i$ can differ by at most 1.

We claim that $0 \leq y_i - y_j \leq 1$ for any $i < j$. If $\sum z_i = \sum z'_i$, then Lemma 12.3.4 implies that $z_i = z'_i$ for $0 \leq i < k/2$. After the final layer of

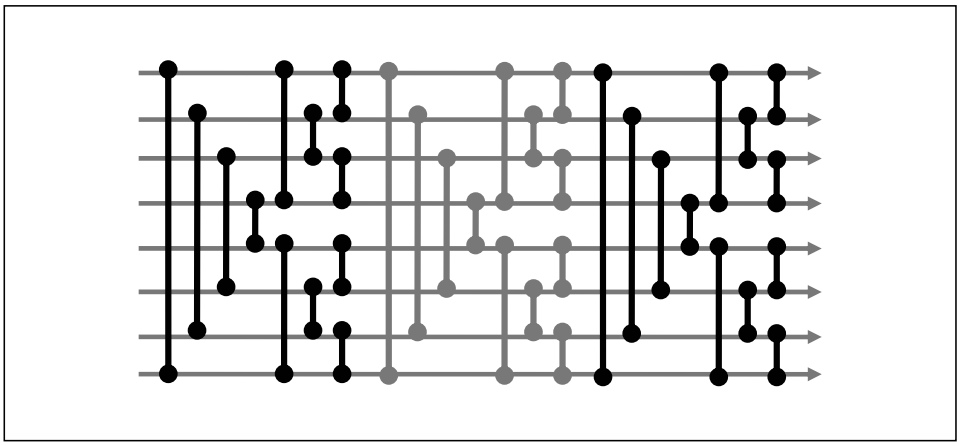


Figure 12.19: *A* PERIODIC [8] *counting network.*

balancers,

$$y_i - y_j = z_{\lfloor i/2 \rfloor} - z_{\lfloor j/2 \rfloor},$$

and the result follows because z has the step property.

Similarly, if $\sum z_i$ and $\sum z'_i$ differ by one, Lemma 12.3.5 implies that $z_i = z'_i$ for $0 \leq i < k/2$, except for a unique ℓ such that z_ℓ and z'_ℓ differ by one. Let $max(z_\ell, z'_\ell) = x + 1$ and $min(z_\ell, z'_\ell) = x$ for some non-negative integer x. From the step property for z and z' we have, for all $i < \ell$, $z_i = z'_i = x + 1$ and for all $i > \ell$ $z_i = z'_i = x$. Since z_ℓ and z'_ℓ are joined by a balancer with outputs $y_{2\ell}$ and $y_{2\ell+1}$, it follows that $y_{2\ell} = x + 1$ and $y_{2\ell+1} = x$. Similarly, z_i and z'_i for $i \neq \ell$ are joined by the same balancer. Thus for any $i < \ell$, $y_{2i} = y_{2i+1} = x + 1$ and for any $i > \ell$, $y_{2i} = y_{2i+1} = x$. The step property follows by choosing $c = 2\ell+1$ and applying Lemma 12.3.1. ■

The proof of the following theorem is now immediate.

Theorem 12.3.7 *In any quiescent state, the outputs of* BITONIC $[w]$ *have the step property.*

A Periodic Counting Network

In this section we show that the Bitonic network is not the only counting network with depth $O(log^2 n)$. We introduce a new counting network with the interesting property that it is *periodic*, consisting of a sequence of identical subnetworks.

We start by defining chains and cochains. Given a sequence $x = \{x_i | i = 0, \ldots, n-1\}$, it is convenient to represent each index (subscript) as a binary string. A *level i chain* of x is a subsequence of x whose indices have the same i low-order bits. For example, the subsequence x^E of entries with even indices is a level 1 chain, as is the subsequence x^O of entries with odd indices. The *A-cochain* of x, denoted x^A, is the subsequence whose indices have the two low-order bits 00 or 11. For example, the *A-cochain* of the sequence $x_0, \ldots, x_7$ is x_0, x_3, x_4, x_7. The *B-cochain* x^B is the subsequence whose low-order bits are 01 and 10.

Define the network BLOCK $[k]$ as follows. When k is equal to 2, the BLOCK $[k]$ network consists of a single balancer. The BLOCK $[2k]$ network for larger k is constructed recursively. We start with two BLOCK $[k]$ networks A and B. Given an input sequence x, the input to A is x^A, and the input to B is x^B. Let y be the output sequence for the two subnetworks, where y^A is the output sequence for A and y^B the output sequence for B. The final stage of the network combines each y_i^A and y_i^B in a single balancer, yielding final outputs z_{2i} and z_{2i+1}. Figure 12.23 describes the recursive construction of a BLOCK [8] network. The PERIODIC $[2k]$ network consists of $\log k$ BLOCK $[2k]$ networks joined so that the i^{th} output wire of one is the i^{th} wire of the next. Figure 12.19 is a PERIODIC [8] counting network [1]

A Software Periodic Network

Here is how to implement the Periodic network in software. We will reuse the Balancer class in Figure 12.16. A single layer of a Block network is implemented by the The Layer class (Figure 12.3.1) of size s joins input wires i and $s - i - 1$ to the same balancer.

In the Block class (Figure **??**), after the token emerges from the intiial Layer network, it passes through one of two half-size BLOCK [] networks (called *north* and *south*).

The Periodic network implementation (Figure 12.22) is very simple: it consists of an array of $\log n$ Block networks, and each token traverses each block in sequence, where the output wire taken on each block is the input wire for its successor.

[1] Despite the apparent similarities between the layouts of the BLOCK $[2k]$ and MERGER $[2k]$ networks, there is no permutation of wires that yields one from the other.

```
public class Layer {
  int size;
  Balancer[] layer;
  public Layer(int size) {
    this.size = size;
    layer = new Balancer[size];
    for (int i = 0; i < size / 2; i++) {
      layer[i] = layer[size-i-1] = new Balancer();
    }
  }
  public int traverse(int input) {
    int toggle = layer[input].traverse();
    int hi, lo;
    if (input < size / 2) {
      lo = input;
      hi = size - input - 1;
    } else {
      lo = size - input - 1;
      hi = input;
    }
    if (toggle == 0) {
      return lo;
    } else {
      return hi;
    }
  }
}
```

Figure 12.20: *Layer network implementation*

Proof of Correctness

In the proof we use the technical lemmas about input and output sequences presented in Section 12.3.1. The following lemma will serve a key role in the inductive proof of our construction:

Lemma 12.3.8 *For $i > 1$,*

1. *The level i chain of x is a level $i-1$ chain of one of x's cochains.*
2. *The level i chain of a cochain of x is a level $i+1$ chain of x.*

```
public class Block {
  Block north;
  Block south;
  Layer layer;
  int size;
  public Block(int size) {
    this.size = size;
    if (size > 2) {
      north = new Block(size / 2);
      south = new Block(size / 2);
    }
    layer = new Layer(size);
  }
  public int traverse(int input) {
    int wire = layer.traverse(input);
    if (size > 2) {
      if (wire < size / 2) {
        return north.traverse(wire);
      } else {
        return (size / 2) + south.traverse(wire - (size / 2));
      }
    } else {
      return wire;
    }
  }
}
```

Figure 12.21: *Block network implementation*

Proof: Follows immediately from the definitions of chains and cochains. ■

As will be seen, the price of modularity is redundancy, that is, balancers in lower level blocks will be applied to sub-sequences that already have the desired step property. We therefore present the following lemma that amounts to saying that applying balancers "evenly" to such sequences does not hurt:

Lemma 12.3.9 *If x and x' are sequences each having the step property, and pairs x_i and x'_i are routed through a balancer, yielding outputs y_i and*

```
public class Periodic {
  Block[] block;
  public Periodic(int size) {
    int logSize = 0;
    int _size = size;
    while (_size > 1) {
      logSize++;
      _size = _size / 2;
    }
    block = new Block[logSize];
    for (int i = 0; i < logSize; i++){
      block[i] = new Block(size);
    }
  }
  public int traverse(int input) {
    int wire = input;
    for (Block b : block) {
      wire = b.traverse(wire);
    }
    return wire;
  }
}
```

Figure 12.22: *Periodic network implementation*

y'_i, then the sequences y and y' each have the step property.

Proof: For any $i < j$, given that x and x' have the step property, $0 \leq x_i - x_j \leq 1$ and $0 \leq x'_i - x'_j \leq 1$ and therefore the difference between any two wires is $0 \leq x_i + x'_i - (x_j + x'_j) \leq 2$. By definition, for any i, $y_i = \left\lceil \frac{x_i + x'_i}{2} \right\rceil$ and $y'_i = \left\lfloor \frac{x_i + x'_i}{2} \right\rfloor$, and so for any $i < j$, it is the case that $0 \leq y_i - y_j \leq 1$ and $0 \leq y'_i - y'_j \leq 1$, implying the step property. ■

To prove the correctness of our construction for PERIODIC $[k]$, we will show that if a block's level i input chains have the step property, then so do its level $i-1$ output chains, for i in $\{0, \ldots, \log k - 1\}$. This observation implies that a sequence of $\log k$ BLOCK $[k]$ networks yields the step property on its outputs.

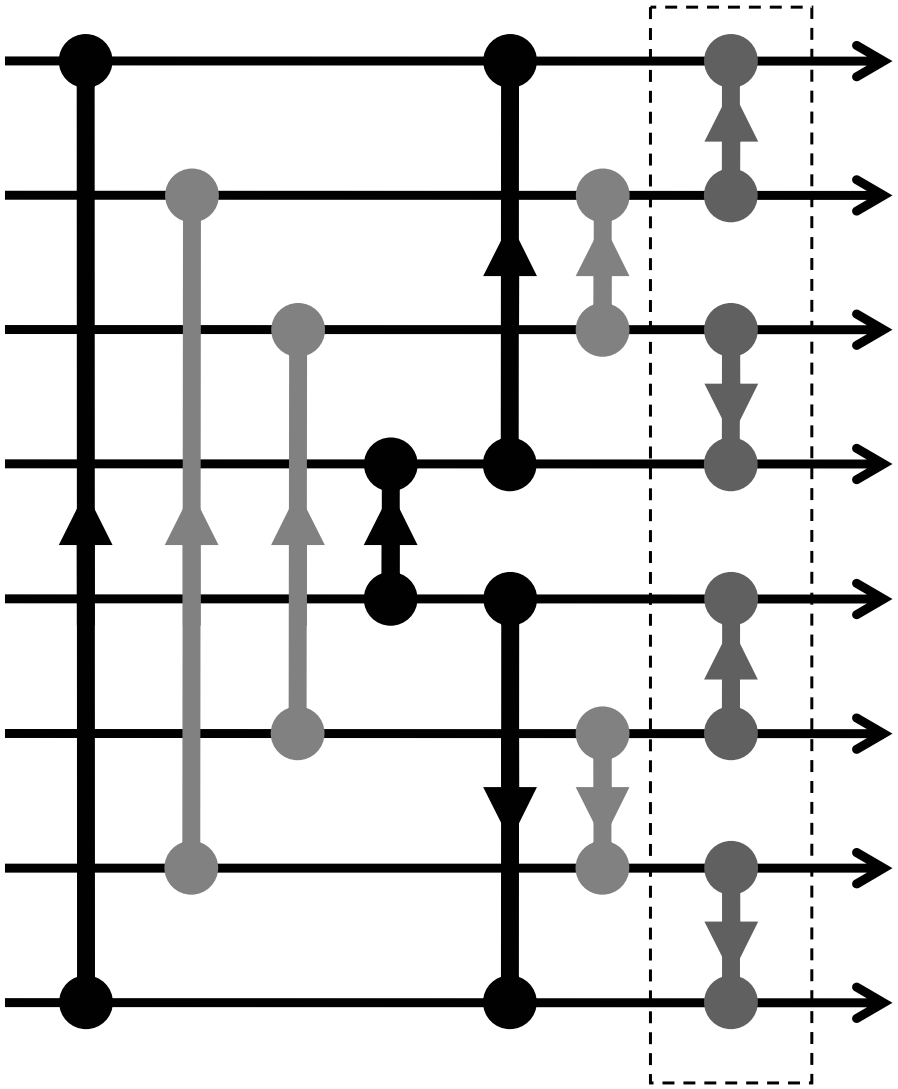


Figure 12.23: *A* Block[8] *balancing network.*

Lemma 12.3.10 *Let* Block[2k] *be quiescent with input sequence x and output sequence y. If x^E and x^O both have the step property, so does y.*

Proof: We argue by induction on $\log k$. The proof is similar to that of Lemma 12.3.6.

For the base case, when $2k = 2$, BLOCK$[2k]$ is just a balancer, so its outputs are guaranteed to have the step property by the definition of a balancer.

For the induction step, assume the result for BLOCK$[k]$ and consider a BLOCK$[2k]$. Let x be the input sequence to the block, z the output sequence of the nested blocks A and B, and y the block's final output sequence. The inputs to A are the level 2 chains x^{EE} and x^{OO}, and the inputs to B are x^{EO} and x^{OE}. By Lemma 12.3.8, each of these is a level 1 chain of x^A or x^B. These sequences are the inputs to A and B, themselves of size k, so the induction hypothesis implies that the outputs z^A and z^B of A and B each has the step property.

Lemma 12.3.3 implies that $0 \leq \sum x_i^{EE} - \sum x_i^{EO} \leq 1$ and $0 \leq \sum x_i^{OE} - \sum x_i^{OO} \leq 1$. It follows that the sum of A's inputs, $\sum x_i^{EE} + \sum x_i^{OO}$, and the sum of B's inputs, $\sum x_i^{EO} + \sum x_i^{OO}$, differ by at most 1. Since balancers do not swallow or create tokens, $\sum z^A$ and $\sum z^B$ also differ by at most 1. If they are equal, then Lemma 12.3.4 implies that $z_i^A = z_i^B = z_{2i} = z_{2i+1}$. For $i < j$,

$$y_i - y_j = z^A_{\lfloor i/2 \rfloor} - z^A_{\lfloor j/2 \rfloor}$$

and the result follows because z^A has the step property.

Similarly, if $\sum z_i^A$ and $\sum z_i^B$ differ by one, Lemma 12.3.5 implies that $z_i^A = z_i^B$ for $0 \leq i < k$, except for a unique ℓ such that z_ℓ^A and z_ℓ^B differ by one. Let $max(z_\ell^A, z_\ell^B) = x + 1$ and $min(z_\ell^A, z_\ell^B) = x$ for some non-negative integer x. From the step property on z^A and z^B we have, for all $i < \ell$, $z_i^A = z_i^B = x + 1$ and for all $i > \ell$ $z_i^A = z_i^B = x$. Since z_ℓ^A and z_ℓ^B are joined by a balancer with outputs $y_{2\ell}$ and $y_{2\ell+1}$, it follows that $y_{2\ell} = x + 1$ and $y_{2\ell+1} = x$. Similarly, z_i^A and z_i^B for $i \neq \ell$ are joined by the same balancer. Thus for any $i < \ell$, $y_{2i} = y_{2i+1} = x + 1$ and for any $i > \ell$, $y_{2i} = y_{2i+1} = x$. The step property follows by choosing $c = 2\ell+1$ and applying Lemma 12.3.1. ■

Supporting Decrements

Can counting networks do anything besides increments? We now show how to extend counting networks to support decrements. We introduce a new kind of token, called an *antitoken*, which we use for decrements. Recall that a token executes a getAndComplement(): it atomically reads the toggle value and complements it, and then departs on the output wire indicated by the old toggle value. Instead, an antitoken complements the toggle value, and then departs on the output wire indicated by the new toggle value.

```
public synchronized int antiTraverse() {
  try {
    if (toggle) {
      return 1;
    } else {
      return 0;
    }
  } finally {
    toggle = !toggle;
  }
}
```

Figure 12.24: *The mAntiTraverse method*

Informally, an antitoken "cancels" the effect of the most recent token on the balancer's toggle state, and vice versa. Surprisingly, antitokens work on any counting network.

Instead of simply balancing the number of tokens that emerges on each wire, we assign a *weight* of $+1$ to each token and -1 to each antitoken. We generalize the step property to require that the sums of the weights of the tokens and antitokens that emerge on each wire have the step property. We call this property the *weighted step property.*

Figure 12.3.1 shows how to implement an antiTraverse() method that moves an antitoken though a balancer. Adding an antiTraverse() method to the other networks is left as an exercise to the reader.

We will need the following "pumping" lemma. Recall that a network state is just the states of the network's balancers.

Lemma 12.3.11 *Let $\mathcal{B}$ be a width-w balancing network of depth d in a quiescent state s. Let $N = 2^d$. If N tokens enter the network on the same wire, pass through the network, and exit, then $\mathcal{B}$ will have the same state after the tokens exit as it did before they entered.*

Proof: Left as an exercise (hint: use induction on the network depth). ■

Lemma 12.3.11 applies to all balancing networks, not just counting networks.

Lemma 12.3.12 *Let $\mathcal{B}$ be a balancing network in a quiescent state s, and suppose a token enters on wire i and passes through the network, leaving the*

network in state s'. If an antitoken now enters on wire i and passes through the network, then the network goes back to state s.

Proof: Immediate from definition of antitoken. ■

These two lemmas imply that sending antitokens through a balancing network always leaves the network in a state reachable by tokens alone.

Theorem 12.3.13 *If balancing network $\mathcal{B}$ is a counting network for tokens alone, then it is also a balancing network for tokens and antitokens.*

Proof: (Sketch) Without loss of generality, we can restrict our attention to sequential executions. If X has the step property, then its *step point* is the unique wire i whose weight is less than wire $i-1$, or wire 0 if all weights are equal.

Let t be the number of tokens that have traversed the network, and a the number of antitokens. We argue inductively that the weights have the step property with step point $t - a \bmod w$. Initially, the step point is zero.

If a token traverses the network, the step point advances by one because the network is a counting network for tokens. Suppose an antitoken traverses the network. By Lemma 12.3.11, the network state is that same as if $N-1$ tokens had entered on that wire and had all traversed the network. Because w, the network width, divides N, the new network state is the same as if N/w additional tokens had emerged on all output wires except for one, from which one fewer token had emerged. The antitoken has the effect of moving the "step" point back one wire. ■

12.4 Adding Networks

The combining tree construction given above is easily generalized to implement getAndAdd() counters. Can we do the same with counting network? Here, the news is bad. We will show that any adding network must have depth at least $n-1$, where n is the maximum number of concurrent tokens. This bound is discouraging because it implies that the size of the network depends on the number of threads (also true for combining trees, but not counting networks), and that the network has inherently high latency.

First, we generalize balancing networks as follows. A *switching network* is a directed graph, where edges are called *wires* and node are called *switches*. Each thread shepherds a *token* through the network. Switches and tokens are allowed to have internal states. A token arrives at a switch via an input wire. In one atomic step, the switch absorbs the token, changes its state

and possibly the token's state, and emits the token on an output wire. Here, for simplicity, we assume that switches have two input and output wires, although the results hold for switches with arbitrary fan-in and fan-out. Note that switching networks are more powerful than balancing networks, since switches can have arbitrary state (instead of a single bit) and tokens also have state. Since are concerned with lower bounds, it is sensible to examine the most powerful (natural) generalization of balancing networks.

We claim that a thread attempting to add any number other than zero must traverse at least $n-1$ tokens, even if it is running by itself, with no other tokens traversing the network. We say that a token is *in front of* a switch if it is on one of the switch's input wires.

Start with the network in a quiescent state q_0, where the next token to run will take value 0. Imagine we have one token t of weight a and $n-1$ tokens $t_1, \ldots, t_{n-1}$ all of weight b, where $b > a$, each on a distinct input wire. Denote by $\mathcal{S}$ the set of switches that t traverses if it traverses the network by starting in q_0.

Lemma 12.4.1 *If we run the $t_1, \ldots, t_{n-1}$ one at a time though the network, we can halt each t_i in front of a switch of $\mathcal{S}$.*

Proof: By induction on i, $1 \leq i \leq n-1$.

Base Case

First, run t_1 by itself. If possible, stop it in front of the first switch of $\mathcal{S}$ that it encounters. If it traverses the network without encountering a switch of $\mathcal{S}$, then it emerges with value 0 (by hypothesis). If we then start up t and have it traverse the network, then it also emerge with value 0, since it did not traverse any balancer visited by t_1. But t and t_1 cannot both return 0, since one has to be ordered after the other. It follows that t_1 stops in front of some balancer of $\mathcal{S}$.

Induction Step

Assume inductively that for some i, $1 < i \leq n-1$, the claim holds for all j, $1 \leq j < i$; that is, we can run $t_1, \ldots, t_{i-1}$ until each one is in front of a balancer of $\mathcal{S}$.

Run t_i though the network. If possible, stop it in front of the first switch of $\mathcal{S}$ that it encounters. If it traverses the network without encountering a switch of $\mathcal{S}$, then it emerges with value $k \cdot b$, for some $0 \leq k < i$ (because it "observes" only the effects of other tokens adding b). If we then start up t and have it traverse the network, then it also emerge with value 0, since it did not traverse any balancer visited by $t_1, \ldots, t_i$. It is not possible to order these operations, however, because if t takes 0, then every other token must

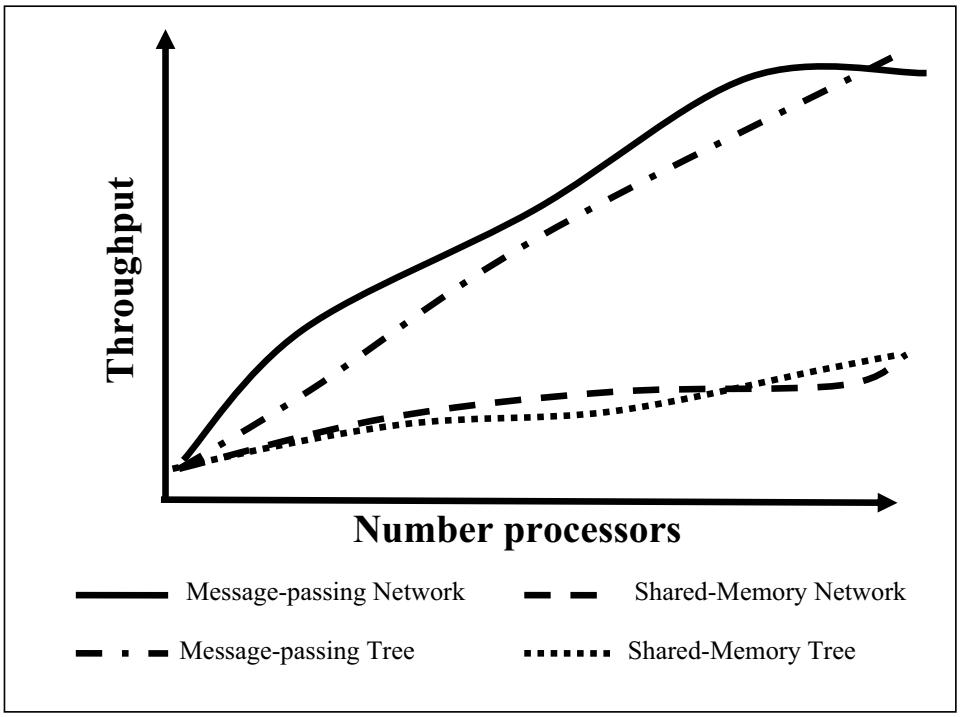


Figure 12.25: *Throughput of Trees versus Networks*

take a value $v_i \equiv a \pmod{b}$, but t_i took a value $v_i \equiv 0 \pmod{b}$, which is impossible because a is not divisible by b. ■

At the end of this construction, $n-1$ tokens are in front of switches of $\mathcal{S}$. Since switches have two input wires, it follows that t's path through the network encompasses at least $n-1$ switches.

12.4.1 Performance

We now review the result of an experiment comparing how counting networks and combining trees perform on a simulated 64-processor cache-coherent NUMA machine. Figure 12.27 compares both shared-memory and message-passing implementations of counting networks and combining trees. For the message-passing implementations, the balancers are mapped onto the processors. For the shared memory implementation, only the toggles are shared, synchronized by queue locks. The message-passing implementation is more efficient because it does not need to use locks or even cache coher-

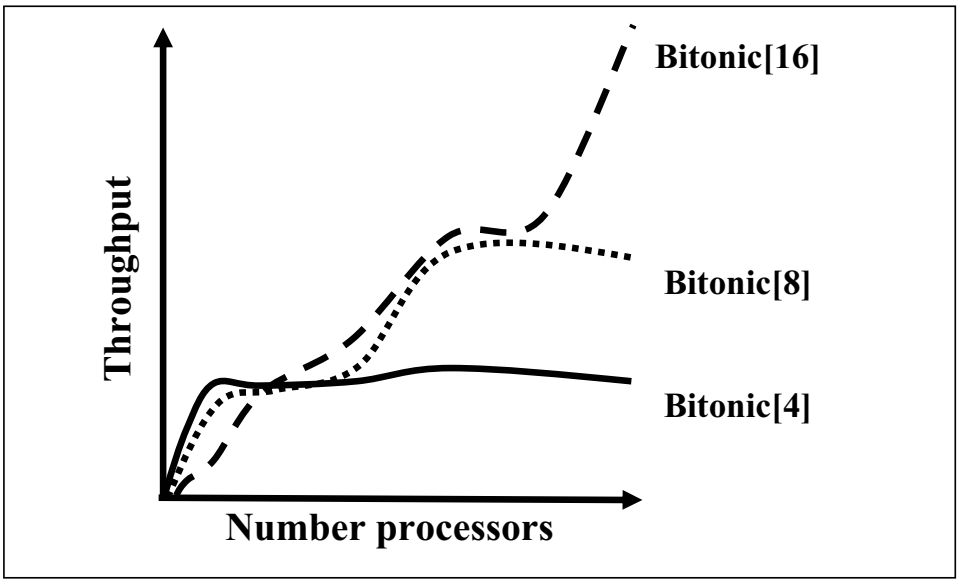


Figure 12.26: *Performance of Counting Network Related to Number of Processors*

ence.

Figure 12.26 shows how throughput varies with the number of processors (threads) and network width. We can see that network throughput rises with the number of threads up to a point, and then it saturates (remains constant or declines). Here is a likely explanation for this behavior.

1. If the number of threads is less than the number of balancers, then the network's pipeline is partly empty, and throughput suffers.

2. If the number of threads is greater than the number of balancers, then threads arrive at the balancers at the same time and have to compete, resulting in contention.

3. Performance should level off when the number of threads is roughly equal to the number of balancers.

The best ratio of balancers to threads is: B[w]= $w \log^2 w = P$

12.5 Exercises

Exercise 12.5.1 Prove Lemma 12.3.11. Let $\mathcal{B}$ be a width-w balancing network of depth d in a quiescent state s. Let $N = 2^d$. If N tokens enter

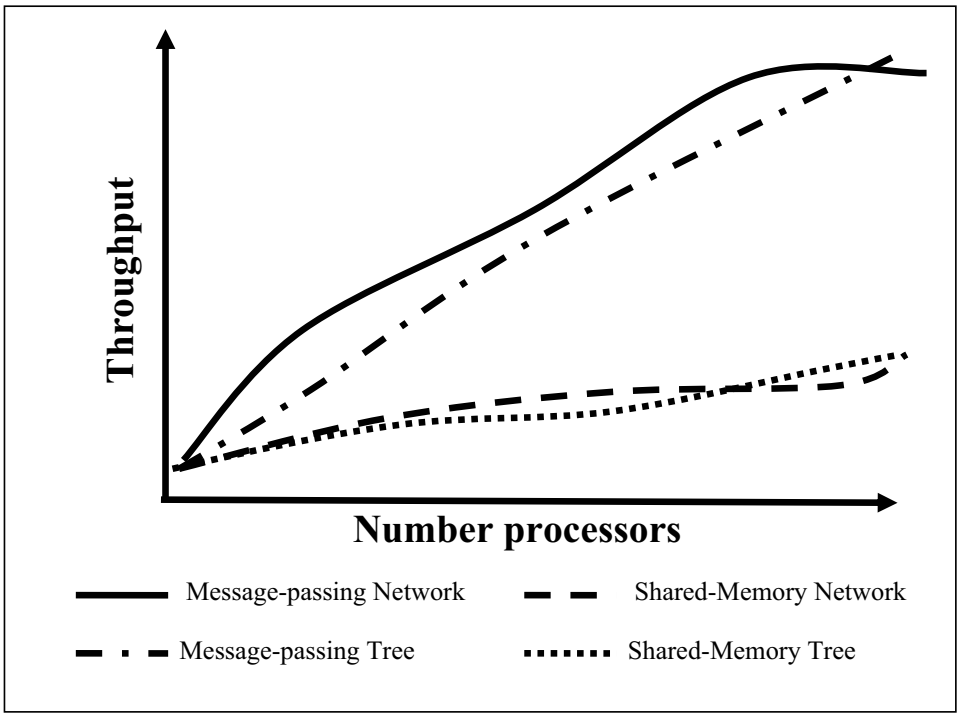


Figure 12.27: *Throughput of combining trees and counting networks*

the network on the same wire, pass through the network, and exit, then $\mathcal{B}$ will have the same state after the tokens exit as it did before they entered.

Solution: We argue by induction on d, the network depth.

When the network depth is 1, the tokens affect only one balancer, $s^d = 2$, and any even number of tokens leaves the balancer in the same state.

Assume the claim for networks of depth $d-1$. If 2^{d-1} tokens enter on one wire, then every balancer at depth less than d has an even number of inputs, but some balancer at depth d may have an odd number. If, however, 2^d tokens enter on the same wire, then every balancer at depth less than d still has an even number of inputs, but all balancers at depth d will have twice the number of inputs, that is, an even number.

Exercise 12.5.2 Let X and Y be k-smooth sequences of length n. A *matching* layer of balancers for X and Y is one where each element of X is joined by a balancer to an element of Y in a one-to-one correspondence.

Prove the following generalization of Lemma 12.3.9: If X and Y are each k-smooth, and Z is the result of matching X and Y, then Z is $(k+1)$-smooth.

Solution: Let x and y be the smallest values in X and Y. Suppose a balancer joins x_i and y_j to produce z_i and z_j. If the balancer is is in a state where it outputs the first token on wire i, then

$$z_i = \left\lceil \frac{x_i + y_j}{2} \right\rceil \qquad \text{and} \qquad z_j = \left\lfloor \frac{x_i + y_j}{2} \right\rfloor.$$

The smallest value that any element in Z can assume is thus

$$\left\lfloor \frac{x+y}{2} \right\rfloor$$

and the largest value is

$$\left\lceil \frac{x+k+y+k}{2} \right\rceil = \left\lceil \frac{x+y}{2} \right\rceil + k \leq \left\lfloor \frac{x+y}{2} \right\rfloor + k + 1.$$

The largest and smallest elements of Z differ by at most $k+1$.

Exercise 12.5.3 Consider a BLOCK $[k]$ network in which each balancer has been initialized to an arbitrary state (either *up* or *down*). Show that no matter what the input distribution is, the output distribution is $(\log k)$-smooth. (Hint: you may use the claim in Exercise 12.5.2.)

Solution: We argue by induction on n. When n is 2, the network is a single balancer, so the output is 1-smooth.

In the BLOCK $[2n]$ network, the output sequences Y^A and Y^B from the two component BLOCK $[n]$ networks are each $\log n$-smooth by the induction hypothesis, and the final EVENODD$[2n]$ layer is a matching for Y^A and Y^B. By Lemma **??**, the output is $(\log n + 1)$-smooth.

Exercise 12.5.4 A *Boolean sorting network* is a network built out of *comparators*, which are gates having two input lines and two output lines. Given Boolean inputs x_0 and x_1, a comparator produces outputs $y_0 = x_0 \vee x_1$, and $y_1 = x_0 \wedge x_1$. In other words, if one input is *true* and the other *false*, the true input is sent to output wire 0.

A *Boolean comparison network*, like a balancing network, is an acyclic network of comparators. It functions as follows: a Boolean value is placed on each of its n input lines. These values pass through each layer of compartors together, finally leaving together on the network output wires.

A Boolean comparison network is a *Boolean sorting network* if, whenever exactly k *true* values appear on the input lines, (1) exactly k *true* values appear on the output lines, and (2) if output line y_i is *true*, so is y_{i-1}. (In other words, all *true* values exit on lower-numbered wires.)

1. Show that if you take a counting network and replace each balancer with a Boolean comparator, the result is a Boolean sorting network.

2. Show that the converse is false: display a Boolean sorting network that is not a counting network.

Exercise 12.5.5 A *smoothing network* is a balancing network that ensures that in any quiescent state, the output sequence $y_0, .., y_{n-1}$ satisfies

$$\text{if} \quad i < j \quad \text{then} \quad |y_i - y_j| \leq 1.$$

Counting network are smoothing networks, but not vice-versa.

Define a *pseudo-sorting network* to be a Boolean sorting network (as defined in Exercise 12.5.4) in which each Binary comparator has been replaced by a balancer.

Let $\mathcal{N}$ be the balancing network constructed by taking a smoothing network $\mathcal{S}$ of width w, a pseudo-sorting network $\mathcal{P}$ also of width w, and joining the i^{th} output wire of $\mathcal{S}$ to the i^{th} input wire of $\mathcal{P}$.

Show that $\mathcal{N}$ is a counting network.

Exercise 12.5.6 A *3-balancer* is a balancer with three input lines and three output lines. Like its 2-line relative, its output sequences have the step property in any quiescent state. Construct a depth-3 counting network with 6 input and output lines from 2-balancers and 3-balancers. Explain why it works.

Exercise 12.5.7 Adapt the argument in Section 12.4 to show that a *linearizable* counting network has depth at least n.

Exercise 12.5.8 Consider the following n-thread counting algorithm. Each thread first uses a bitonic counting network of width n to take a counter value v. It then goes through a *waiting filter*, in which each thread waits for threads with lesser values to catch up.

The waiting filter is an array filter [] of n Boolean values. Define the phase function

$$\phi(v) = \lfloor (v/n) \rfloor \bmod 2.$$

A thread that exits with value v spins on filter $[(v-1) \bmod n]$ until that value is set to $\phi(v-1)$. The thread responds by setting filter $[v \bmod n]$ to $\phi(v)$, and then returns v.

1. Exaplain why this counter implementation is linearizable.

2. Assume the result of Exercise 12.5.7: any linearizable counting network has depth at least n. Explain why the filter [] construction does not contradict this claim.

3. On a bus-based multiprocessor, would this filter [] construction have better throughput than a single variable protected by a spin lock? Explain.

Exercise 12.5.9 If a sequence $X = x_0, \ldots x_{n-1}$ is k-smooth, then the result of passing X through a balancing network is k-smooth.

Solution: We show that the result of passing any two elements of X through a balancer leads to a sequence that is k-smooth, and the lemma follows by induction. Let Z be the result of passing two elements of X through a balancer. The smallest element in Z is greater than or equal to the smallest element in X, and the largest element in Z is lesser than or equal to the largest element in X. Since X is k-smooth, Z is also k-smooth.

12.6 Chapter Notes

The software combining tree presented here is adapted from a proposal by Goodman, Vernon, and Woest. [**?**].

Counting networks were invented by Aspnes, Herlihy, and Shavit [**?**]. They showed that counting networks are related to *sorting networks* [**?**, **?**, **?**] in the following way: every counting network is isomorphic to a sorting network, but not vice-versa. This result implies that the $\Omega(\log w)$ lower bound for sorting network depth holds also for counting networks.

Aharonson and Attiya [**?**] were the first to generalize the notion of a balancer to have different number of input and output wires. This construction was generalized by Busch and Mavronicolas [**?**] to show that the number P^d/w must be an integer, where P is the least common multiple of the different widths of balancers in the network, and d and w are the depth and width of the network, respectively.

Klugerman and Plaxton [**?**] present an explicit counting network construction with depth $O(c^{\log^* w} \log w)$ (for some positive constant c). They also present a randomized counting network construction with depth $O(\log w)$, that counts with extremely high probability. Using this construction they prove the existence of a counting network of depth $O(\log w)$, that matches the lower bound $\Omega(\log w)$. Klugerman [**?**] extends this result and presents a polynomial-time method to construct an $O(\log w)$ depth counting network.

All the above constructions use as a building block the AKS sorting network [**?**] whose depth expression $O(\log w)$ hides huge constants and subsequently all these networks are impractical.

Klugerman and Plaxton [**?**] present an explicit counting network construction with depth $O(c^{\log^* w} \log w)$ (for some positive constant c). They also present a randomized counting network construction with depth $O(\log w)$, that counts with extremely high probability. Using this construction they prove the existence of a counting network of depth $O(\log w)$, that matches the lower bound $\Omega(\log w)$. Klugerman [**?**] extends this result and presents a polynomial-time method to construct an $O(\log w)$ depth counting network. All the above constructions use as a building block the AKS sorting network [**?**] whose depth expression $O(\log w)$ hides huge constants and subsequently all these networks are impractical.

The linear lower bound on adding networks is due to Fatourou and Herlihy [**?**] which extends earlier work by Herlihy, Shavit, and Waarts [**?**]. Lynch, Shavit, Shvartsman, and Touitou [**?**] show that under certain timing conditions certain kinds of counting networks, the *uniform* counting networks are actually linearizable. Mavronicolas, Papatriantafilou, and Tsigas [**?**] extend the analysis of timing conditions for linearizability and provide tighter bounds for such conditions.

Herlihy and Tirthapura [**?**] ask how counting networks behave if they are initialized in an arbitrary state (instead of having all balancers in the *up*) state. They show that the BITONIC $[k]$ and PERIODIC $[k]$ networks, started in arbitrary initial states, remain remarkably smooth, degrading from 1-smooth to $(\log k)$-smooth. Moreover, these bounds are tight. They also show that any balancing network can be made *self-stabilizing*, meaning that if the network is started in an arbitrary initial state, it will eventually act as if it had been started in the usual initial state.

Aiello, Venkatesan, and Yung [**?**] study the behavior of counting networks in which balancers randomly balance outputs. Herlihy and Tirthapura [**?**] study the behavior of counting and smoothing networks whose initial states are chosen at random. They show that BLOCK $[k]$ smoothing network, when started in a random initial state, is $2.36\sqrt{\log(k)}$-smooth with high probability. They also show that the BITONIC $[k]$ and PERIODIC $[k]$ networks when started in random initial states, are $O(\sqrt{\log(w)})$-smooth with high probability.

Aharonson and Attiya [**?**] constructed a counting network of width $w = p2^k$ and depth $O(\log^3(w/p))$ from balancers of width 2 and p. Busch, Hardavellas, and Mavronicolas [**?**] give a construction of width $w = p2^k$ and depth $O(\log^2(w/p))$ using balancers of width 2 and p. Felten, LaMarca, and

Ladner [?] give a construction of width $w = 2^k$ and depth $O(\log^2 w)$ from balancers of width 2^ℓ, as well as a construction of width $w = p2^k$ using also balancers of width p. Busch and Herlihy [?] give a practical construction of arbitrary width w and depth $O(\log^2 w)$ from balancers of width at most the maximum factor of w, where w can be factored in any way (including the prime factorization).

Shavit and Zemach [?], as part of a more general result, present the *counting tree* constructed from 1-input, 2-output balancers. This counting network has the form of a binary tree with input width 1, output width w and depth $logw$.

Aiello, Venkatesan, and Yung [?] present a counting network construction of input width w and output width $w \log w$ and optimal depth $O(\log w)$, using 1-input, 2-output balancers and regular balancers of width 2. This construction uses as a subroutine the AKS network [?] and is therefore not practical. Busch and Mavronicolas [?] present a counting network construction with input width $w = 2^k$ output width $p2^l$ and depth $O(\log^2 w)$, using 2-input, p-output balancers and regular balancers of width 2.

Working from early results by Shavit and Touitou [?], Aiello, Busch, Herlihy, Mavronicolas, Shavit, and Touitou [?] showed that *any* counting network can be adapted to support decrements.

Dwork, Herlihy, and Waarts [?] introduce a formal model for measuring *contention*, the extent to which concurrent processors access the same memory location simultaneously. They found that the amortized contention in the bitonic and periodic counting networks are $\Theta(n \log^2 w/w)$ and $\Theta(n \log^3 w/w)$, respectively. Hardavellas, Damianos, and Mavronicolas [?] improve on the bound of the periodic network and find that its amortized contention is $\Theta(n \log^2 w/w)$.

The experimental results in this chapter are taken from Herlihy, Lim, and Shavit [?].