Counting Networks and Multi-Processor Coordination (Extended Abstract)

James Aspnes* Maurice Herlihy† Nir Shavit‡

Digital Equipment Corporation
Cambridge Research Lab

CRL 90/11 September 18, 1991

Abstract

Many fundamental multi-processor coordination problems can be expressed as *counting problems:* processes must cooperate to assign successive values from a given range, such as addresses in memory or destinations on an interconnection network. Conventional solutions to these problems perform poorly because of synchronization bottlenecks and high memory contention.

Motivated by observations on the behavior of sorting networks, we offer a completely new approach to solving such problems. We introduce a new class of networks called *counting networks*, i.e., networks that can be used to count. We give a counting network construction of depth $\log^2 n$ using $n \log^2 n$ "gates," avoiding the sequential bottlenecks inherent to former solutions, and having a provably lower contention factor on its gates.

Finally, to show that counting networks are not merely mathematical creatures, we provide experimental evidence that they outperform conventional synchronization techniques under a variety of circumstances.

©Carnegie Mellon University, Digital Equipment Corporation, and Interational Business Machines Corporation 1991. All rights reserved.

* Carnegie Mellon University.
† DEC Cambridge Research Lab.
‡ IBM Almaden Research Center.

1 Introduction

Many fundamental multi-processor coordination problems can be expressed as *counting problems:* processors collectively assign successive values from a given range, such as addresses in memory or destinations on an interconnection network. In this paper, we offer a completely new approach to solving such problems, by introducing *counting networks*, a new class of networks that can be used to count.

Counting networks, like sorting networks [2, 4, 5], are constructed from simple two-input two-output computing elements called *balancers*, connected to one another by wires. However, while an n input sorting network sorts a collection of n input values only if they arrive together, on separate wires, and propagate through the network in lockstep, a counting network can count any number $N \gg n$ of input values even if they arrive at arbitrary times, are distributed unevenly among the input wires, and propagate through the network asynchronously.

Figure 2 provides an example of an execution of a 4-input, 4-output, counting network. A balancer is represented by two dots and a vertical line (see Figure 1). Intuitively, a balancer is just a toggle mechanism [1], repeatedly sending the inputs it receives, one to the left and one to the right. It thus balances the number of values on its output wires. In the example of Figure 2, input values arrive on the network's input lines one after the other. For convenience we have numbered them by the order of their arrival (these numbers are *not* used by the network). As can be seen, the first input (numbered 1) enters on line 2 and leaves on line 1, the second leaves on line 2, and in general, the Nth value will leave on line N mod 4. (The reader is encouraged to try this for him/herself.) Thus, if on the ith output line the network assigns to consecutive outputs the numbers $i, i+4, i+2\cdot 4, ..$, it is *counting* the number of input values without actually passing them all through a shared computing element!

Counting networks achieve a high level of throughput by decomposing interactions among processes into pieces that can be performed in parallel. This decomposition has two performance benefits: It eliminates serial bottlenecks and reduces memory contention. In practice, the performance of many shared-memory algorithms is often limited by conflicts at certain widely-shared memory locations, often called *hot spots* [19]. Reducing hot-spot conflicts has been the focus of hardware architecture design [1, 8, 12, 14, 11] and experimental work in software [3, 9, 10, 16, 20].

Counting networks are also *non-blocking:* processes that undergo halting failures or delays while using a counting network do not prevent other processes from making progress. This property is important because existing shared-memory architectures are themselves inherently asynchronous; process step times are subject to timing uncertainties due to variations in instruction complexity, page faults, cache misses, and operating system activities such as preemption or swapping.

We show a depth $\log^2 n$ construction of a counting network, using $n \log^2 n$ balancers, and argue that our construction produces low levels of contention; we feel that many other concurrent shared-memory algorithms would benefit from a similar contention analysis.

To illustrate the utility of counting networks, we show how to construct highly concurrent implementations of two common data structures: shared counters and producer/consumer buffers. A *shared counter* is simply an object that issues the numbers 1 to n in response to n requests by processes. Shared counters are central to a number of shared-memory synchronization algorithms (e.g., [6, 12, 15, 20]). A *producer/consumer buffer* is a data structure in which items inserted by a pool of producer processes are removed by a pool of consumer processes. Compared to conventional techniques such as spin locks or semaphores, our counting network implementations provide higher throughput, less memory contention, and better tolerance for failures and delays.

[1] It is easy to implement a balancer using a *Compare & Swap*, *Test & Set*, or a randomized consensus primitive.

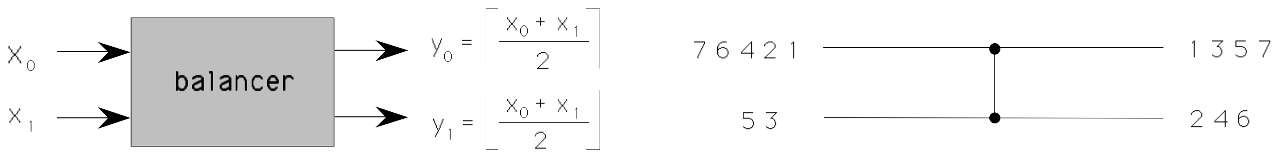

Figure 1: A Balancer.

Our analysis of the counting network construction is supported by experiment. In the appendix, we compare the performance of several implementations of shared counters and producer/consumer buffers on an eighteen-processor Encore MultiMax. When the level of concurrency is sufficiently high, the counting network implementations outperform conventional implementations based on spin locks, sometimes dramatically.

In summary, counting networks represent a new class of concurrent algorithms. They have a rich mathematical structure, they provide effective solutions to important problems, and they perform well in practice. We believe that counting networks have other potential uses, for example as interconnection networks [21] or as load balancers[18], and that they deserve further attention.

2 Networks that Count

2.1 Counting Networks

Counting networks belong to a larger class of networks called balancing networks, constructed from wires and computing elements called balancers, in a manner very similar to that in which comparison networks [5] are constructed from wires and comparators. We begin by describing balancing networks.

A *balancer* is a computing element with two input wires and two output wires[2] (see Figure 1). Tokens repeatedly arrive on one of the balancer's input wires, at arbitrary times, and are repeatedly output on its output wires. Intuitively, one may think of a balancer as a toggle mechanism, that given a stream of input tokens, repeatedly sends one token to the upper output wire and one to the lower, effectively balancing the number of tokens on its output wires. We denote by x_i, $i \in \{0, 1\}$ the number of input tokens ever received on the balancer's ith input wire, and similarly by y_i, $i \in \{0, 1\}$ the number of tokens ever output on its ith output wire. Throughout the paper we will abuse this notation and use x_i (y_i) both as the name of the ith input (output) wire and a count of the number of input tokens received on the wire.

Let the state of a balancer at a given time be defined as the collection of tokens on its input and output wires. We can now formally state the safety and liveness properties of a balancer:

1. In any state, $x_0 + x_1 \geq y_0 + y_1$ (i.e. a balancer never creates output tokens).

[2] In Figure 1 as well as in the sequel, we adopt the notation of [5] and and draw wires as horizontal lines with balancers stretched vertically.

2. Given any finite number of input tokens $m = x_0 + x_1$ to the balancer, it is guaranteed that within a finite amount of time, it will reach a *quiescent* state, that is, one in which $x_0 + x_1 = y_0 + y_1 = m$ (i.e. a balancer never swallows input tokens).

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

4. In any quiescent state the set of input tokens and output tokens are the same.

A *balancing network* of width w is a collection of balancers, where output wires connected to input wires, having w designated input wires $x_0, x_1, .., x_{w-1}$ (which are not connected to output wires of balancers), w designated output wires $y_0, y_1, .., y_{w-1}$ (similarly unconnected), and containing no cycles. Let the state of a network at a given time be defined as the union of the states of all its component balancers. The safety and liveness of the network follow naturally from the above network definition and the properties of balancers, namely, that it is always the case that $\sum_{i=0}^{w-1} x_i \geq \sum_{i=0}^{w-1} y_i$, and for any finite sequence of m input tokens, within finite time the network reaches a *quiescent* state, i.e. one in which $\sum_{i=0}^{w-1} y_i = m$.

It is important to note that we make no assumptions regarding the timing of token transitions from balancer to balancer in a balancing network— its behavior can be viewed as a completely asynchronous process, and is defined in the usual way by a schedule.

To give the reader a feeling of what the above abstraction might represent, consider an implementation on a shared memory multiprocessor. A balancing network is implemented as a shared data structure, where balancers are records and wires are pointers from one record to another. Each of the machine's asynchronous processors runs a program that repeatedly traverses the data structure from some input pointer to some output pointer, each time shepherding a new token through the network.

We define the *depth* of a balancing network to be the maximal depth of any wire, where the depth of a wire is defined as 0 for a network input wire, and $\max(\mathit{depth}(x_0), \mathit{depth}(x_1)) + 1$ for the output wires of a balancer having input wires x_0 and x_1.

A *counting network* of width w is a is a balancing network whose outputs $y_0, .., y_{w-1}$ have the following additional *step property* in quiescent states:

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

To illustrate this property, consider an execution in which tokens traverse the network sequentially, one completely after the other. Figure 2 shows such an execution on a COUNTER[4] network which we will define formally in Section 3. As can be seen, the network moves input tokens to output tokens in increasing order modulo w. Balancing networks having this property are called counting networks, because we can easily construct from them counters which count the total number of tokens that have passed through, or are currently in, the network. Counting is done by adding a "local counter" to each output wire i, so that tokens coming out of that wire are consecutively assigned numbers $i, i + w, i + 2w, .., i + (y_i - 1)w$. (This application is described in greater detail in Section 4.)

The step property can be defined in a number of ways which we will use interchangeably. The connection between them is stated in the following lemma:

Lemma 2.1 *If $y_0, \ldots, y_{w-1}$ is a sequence of non-negative integers, the following statements are all equivalent:*

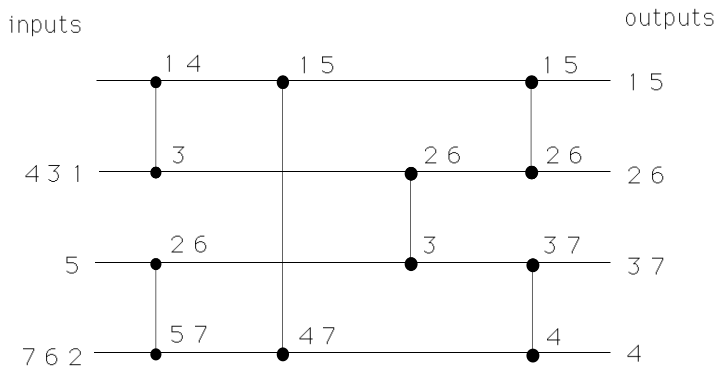

Figure 2: A sequential execution for a COUNTER[4] counting network.

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

It is the third form of the step property that makes counting networks usable as counters.

The requirement that the outputs of a quiescent counting network have the step property might appear to tell us very little about the behavior of a counting network during an asynchronous execution, but in fact it is surprisingly powerful. The reason is that even in a state in which many tokens are passing through the network, if no new tokens arrive the network must eventually settle into a quiescent state. This fact constrains the behavior of the network, and makes it possible to prove such important properties as the following:

Lemma 2.2 *Suppose that in a given execution, a counting network with outputs* $y_0, \ldots y_{w-1}$ *is in a state where* m *tokens have entered the network and* m' *tokens have left it. Then there exist non-negative integers* d_i, $0 \leq i < w$, *such that* $\sum_{i=0}^{w-1} d_i = m - m'$ *and* $y_i + d_i = \lceil \frac{m-i}{w} \rceil$.

2.2 Counting vs. Sorting

Given a balancing network and a comparison network, we will say that they are *isomorphic* if one can be constructed from the other by replacing balancers by comparators or vice versa. The counting network in this paper is isomorphic to the Bitonic sorting network of Batcher [4]. To see that constructing counting networks is a challenging task, consider the following theorem:

Theorem 2.3 *If a balancing network counts, then its isomorphic comparison network sorts, but not vice versa.*

Proof outline: The balancing networks isomorphic to the EVEN-ODD or INSERTION sorting networks [5] are not counting networks.

To prove the other direction, we construct a mapping from the comparison network transitions to the isomorphic balancing network transitions, so that if the balancing network counts, the comparison network sorts.

By the 0-1 principle [5], a comparison network which sorts all sequences of 0's and 1's correctly sorts all sequences. Take any arbitrary sequence of 0's and 1's as inputs to the comparison network, and for the balancing network place a token on each 0 input wire and no token on each 1 input wire. If we run both networks in lockstep, the balancing network will simulate the comparison network.

On every gate where two 0's meet in the comparison network, two tokens meet in the balancing network, so two 0's leave on each wire in the comparison network, and both tokens leave in the balancing network. On every gate where two 1's meet in the comparison network, no tokens meet in the balancing network, so two 1's leave on each wire in the comparison network, and no tokens leave in the balancing network. On every gate where a 0 and 1 meet in the comparison network, the 0 leaves on the lower wire and the 1 on the upper wire, while in the balancing network the token leaves on the lower wire, and no token on the upper wire.

If the balancing network is a counting network, i.e., it has the step property, then the comparison network must have sorted the input sequence of 0's and 1's. ■

2.3 Verifying That a Network Counts

The 0-1 law for comparison networks allows one to verify a supposed sorting network by testing it on a relatively small range of possible executions, namely, those generated by input sequences of zeroes and ones. Does a similar law exist for counting networks? The answer is mixed: on the one hand, it is possible to show that a counting network can be tested by considering only a finite subset of its infinitely many possible executions. On the other hand, the size of that finite subset is dependent on the network's depth, and therefore may be very large.

We first prove that in testing a network, one need only consider sequential executions, that is, executions in which tokens enter and leave the network one completely after the other.

Theorem 2.4 *If a balancing network maintains the step property in all sequential executions, it maintains it in all executions.*

Thus the problem of testing a supposed counting network is reduced from examining all possible executions to examining all sequential executions. The problem can be reduced further by regarding the network as a finite-state automaton. Suppose we have a width-w network with a total of m balancers. If the network is quiescent, we can describe its state completely by specifying for each balancer which of its outputs the next token to arrive will appear on; thus the network has at most 2^m reachable quiescent states. If we consider only sequential executions, we can treat the network as a finite-state machine whose states are the quiescent states and whose transitions correspond to running a token through the network starting at some input-stage balancer. In this representation, an execution may be described by specifying the sequence of input-stage balancers on which the tokens are introduced.

Lemma 2.5 *Let b be a sequence of input tokens of length n which takes the network from a reachable state q back to the same state q. Then if the network counts all sequences of up to $2n + 2^m$ tokens, the length of b is a multiple of w and exactly $\frac{|b|}{w}$ tokens leave on each output wire.*

Based on the above lemma, we can now prove that

Theorem 2.6 *If a width-w balancing network with m balancers counts in all sequential executions in which up to $3 \cdot 2^m$ tokens pass through the network, it is a counting network.*

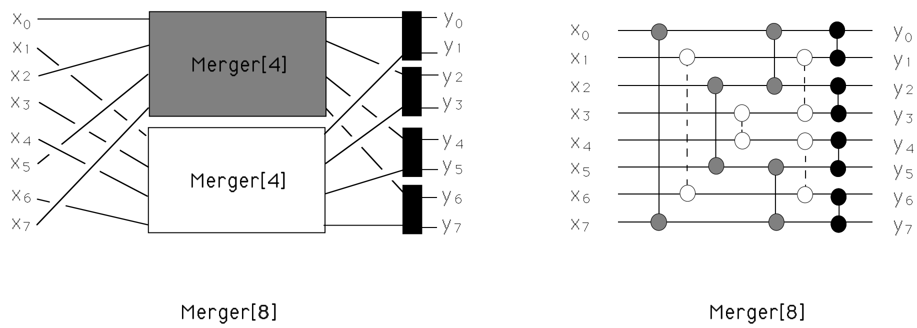

Figure 3: A MERGER [8] balancing network.

Proof outline: By Theorem 2.4 it is enough to show that the network guarantees the step property in sequential executions. Thus we may regard the network as a finite-state machine as in the preceding lemma.

Consider an input sequence a of length greater than 2^m. By the Pigeonhole Principal there exists some subsequence b of length at most 2^m such that $a = a_0ba_1$ and the state of the network after a_0 and a_0b is the same. Thus we can remove b without affecting the behavior of the network on a_0a_1. Since Lemma 2.5 tells us that b contributes an equal number of tokens to each output, the network's output on a_0ba_1 will have the step property if and only if its output on a_0a_1 does. Repeating such contractions will eventually yield an input sequence of length less than 2^m, for which the network guarantees the step property. ■

Finally, we give a lower bound on the number of tokens required by a test as in Theorem 2.6 [3] Let us construct a would-be counting network of the following form. Take two counting networks of width w, labeling their outputs as $a_0 \ldots a_{w-1}$ and $b_0 \ldots b_{w-1}$, respectively. Combine the two networks by running a balancer between a_0 and b_{w-1} and a second balancer between b_0 and a_{w-1}. Now construct a k stage periodic balancing network of width $2w$ by joining k copies of the above network, the outputs of each stage connected to the corresponding inputs of the next. We can now prove that:

Lemma 2.7 *A periodic balancing network with k stages, constructed as above, will count in all executions involving up to $O(2^k w)$ tokens, but is not a counting network.*

3 A Bitonic Counting Network

Counting networks, of course, would not be interesting if we could not exhibit an example of one. In this section we describe how to construct a counting network whose width is any power of 2. The layout of this network is isomorphic to Batcher's Bitonic sorting network [4, 5], though its behavior and correctness arguments are completely different. We give an inductive construction, as this will later aid us in proving its correctness.

[3] A similar counter example can be constructed having any width, not just a power of 2.

Define the width w balancing network MERGER$[w]$ as follows. It has two sequences of inputs of length $w/2$, x and x', and a single sequence of outputs y, of length w. MERGER$[w]$ will be constructed to guarantee that in a quiescent state where the sequences x and x' have the step property, y will also have the step property, a fact which will be proved in the next section.

We define the network MERGER$[w]$ inductively (see example in Figure 3). Since w is a power of 2, we will repeatedly use the notation $2k$ in place of w . When k is equal to 1, the MERGER$[2k]$ network consists of a single balancer. For $k > 1$, we construct the MERGER$[2k]$ network from 2 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 $x'_1, x'_3, \ldots x'_{k-1}$ (i.e. the input to the MERGER$[k]$ network is $x_0, \ldots x_{k-2}, x'_1, \ldots x'_{k-1}$) 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 lines z_i and z'_i into a balancer whose outputs yield y_{2i} and y_{2i+1}.

The MERGER$[w]$ network consists of $\log w$ layers of $w/2$ balancers each. This MERGER$[w]$ network guarantees the step property on its outputs only when its odd and even input subsequences also have the step property— but we can guarantee this by providing those inputs as the outputs of smaller counting networks. We define COUNTER$[w]$ to be the network constructed by passing the outputs from two COUNTER$[w/2]$ networks into a MERGER$[w]$ network, where the induction is grounded in the COUNTER$[1]$ network which contains no balancers and simply passes its input directly to its output. This construction gives us a network consisting of $\binom{\log w+1}{2}$ layers each consisting of $w/2$ balancers.

3.1 Proof of Correctness

In this section we show that COUNTER$[w]$ is a counting network. Before examining the network itself, we present some simple lemmas about the step property.

Lemma 3.1 *If a sequence has the step property, then so do all its subsequences.*

Lemma 3.2 *If $x_0, \ldots, x_{k-1}$ has the step property, then*

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

Lemma 3.3 *Let $x_0, \ldots, x_{k-1}$ and $y_0, \ldots, y_{k-1}$ be arbitrary sequences having the step property. If $\sum_{i=0}^{k-1} x_i = \sum_{i=0}^{k-1} y_i$, then $x_i = y_i$ for all $0 \le i < k$.*

Lemma 3.4 *Let $x_0, \ldots, x_{k-1}$ and $y_0, \ldots, y_{k-1}$ be arbitrary sequences having the step property. If $\sum_{i=0}^{k-1} x_i = \sum_{i=0}^{k-1} y_i + 1$, then there exists a unique j, $0 \le j < k$, such that $x_j = y_j + 1$, and $x_i = y_i$ for $i \ne j$, $0 \le i < k$.*

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

Lemma 3.5 *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 outline: 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 3.1), 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 3.2). 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 3.3 implies that $z_i = z'_i$ for $0 \leq i < k/2$. After the final layer of 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 3.4 implies that $z_i = z'_i$ for $0 \leq i < k/2$, except for a unique j such that z_j and z'_j differ by one. The difference $0 \leq y_i - y_j \leq 1$ for any $i < j$ can be expressed as the difference between earlier and later terms either of z or of z', and the result follows because these two sequences both have the step property. ■

The proof of the following theorem is now immediate.

Theorem 3.6 *In any quiescent state, the outputs of* COUNTER$[w]$ *have the step property.*

4 Applications

We illustrate the utility of counting networks by constructing highly concurrent implementations of three common data structures: shared counters, producer/consumer buffers, and barriers. In Section 5 we give some experimental evidence that that counting network implementations have higher throughput than conventional implementations when contention is sufficiently high.

4.1 Shared Counter

A *shared counter* [6, 12, 7, 15, 20] issues the numbers 0 to $n-1$ in response to the first n requests it receives. To construct the counter, start with an arbitrary width-w counting network. Associate an integer cell c_i with the i^{th} output wire. Initially, c_i holds the value i. A process requests a number by traversing the counting network. After it exits the network on wire i, it atomically adds w to the value of c_i and returns c_i's previous value.

Lemma 2.2 implies that:

Lemma 4.1 *Let x be the largest number yet returned by any operation on the counter. Let S be the set of numbers less than x which have not been returned by any operation on the counter. Then*

1. *The size of S is no greater than the number of operations still in progress.*

2. *If $y \in S$, then $y \geq x - w|S|$.*

3. *Each number in S will be returned by some operation in time $\Delta \cdot d + \Delta_c$, where d is the depth of the network, Δ is the maximum gate delay, and Δ_c is the maximum time to update a cell on an output wire.*

4.2 Producer/Consumer Buffer

A *producer/consumer buffer* is a data structure in which items inserted by a pool of m producer processes are removed by a pool of m consumer processes. The buffer algorithm used here is essentially that of Gottleib, Lubachevsky, and Rudolph [12]. The buffer is an n-element circular array. There are two m-process counting networks, a *producer* network, and a *consumer* network. A producer starts by traversing the producer network, leaving the network with value i. It then atomically inspects the i^{th} buffer element, and, if it is $\perp$, replaces it with the produced item. If that position is full, then the producer waits for the item to be consumed (or returns an exception). Similarly, a consumer traverses the consumer network, exits on wire j, and if the j^{th} position holds an item, atomically replaces it with $\perp$. If there is no item to consume, the consumer waits for an item to be produced (or returns an exception).

Lemma 2.2 implies that:

Lemma 4.2 *Suppose m producers and m' consumers have entered a producer/consumer buffer built out of counting networks of depth d and maximum gate delay Δ. Assume that the time to update each b_i once a process has left the counting network is negligible. Then if $m \leq m'$, every producer leaves the network in time $2d\Delta$ and the network reaches a quiescent state. Similarly if $m \geq m'$, every consumer leaves the network in time $2d\Delta$ and the network reaches a quiescent state.*

5 Performance

The following is a summary of the more complete performance analysis provided in the full paper.

We consider the performance of the network when each processor is assigned a fixed input wire, ensuring that the number of input tokens that can arrive simultaneously at an input wire is bounded. The network *saturation* S is defined to be the expected number of tokens at each balancer. For the COUNTER network, $S = 2n/wd$. The network is *oversaturated* if $S > 1$, and *undersaturated* if $S < 1$. This measure is motivated by the assumption that in a sufficiently long computation, tokens are likely to be spread through the network in an approximately uniform distribution.

Define the *contention* at a balancer at a given time to be the number of tokens pending on its input wires. An oversaturated network represents a full pipeline, hence its throughput is dominated by the per-balancer contention, not by the network depth. If a balancer with S tokens makes a transition in time $\Delta(S)$, then approximately w tokens emerge from the network every $\Delta(S)$ time units, yielding a throughput of $w/\Delta(S)$. Δ is an increasing function whose exact form depends on the particular architecture, but similar measures of degradation have been observed in practice to grow linearly or worse [3, 16]. The throughput of an oversaturated network is therefore maximized by choosing w and d to minimize S, bringing it as close as possible to 1.

The throughput of an undersaturated network is dominated by the network depth, not by the per-balancer contention, since the network pipeline is partially empty. Every $O(1/S)$ time units, w tokens leave the network, yielding throughput $O(wS)$. The throughput of an undersaturated network is therefore maximized by choosing w and d to increase S, bringing it as close as possible to 1.

We implemented several data structures employing counting networks, as well as more conventional implementations using spin locks (which can be considered degenerate counting networks of width one). These implementations were done on an Encore Multimax, using Mul-T [13], a parallel dialect of Lisp. The spin lock is a simple "test-and-test-and-set" loop [17] written in assembly language, and provided by the Mul-T run-time system. Each balancer is protected by a single spin lock.

We compare four shared counter implementations, counting networks of widths 16, 8, and 4, and a conventional spin lock implementation. For each network, we measured the elapsed time necessary for a 2^{20} (approximately a million) tokens to traverse the network, controlling the level of concurrency.

The width-16 network has 80 balancers, the width-8 network has 24 balancers, and the width-4 network has 6 balancers. In Figure 5 the horizontal axis represents the number of processes executing concurrently. The vertical axis represents the elapsed time (in seconds) until all 2^{20} tokens had traversed the network. With no concurrency, the networks are heavily undersaturated, and the spin lock's throughput is the highest by far. As saturation increases, however, so does the throughput for each of the networks. The width-4 network is undersaturated at concurrency levels less than 6. As the level of concurrency increases from 1 to 6, saturation approaches 1, and throughput increases as the elapsed time decreases. Beyond 6, saturation increases beyond 1, and throughput eventually starts to decrease. The other networks remain undersaturated for the range of the experiment; their throughputs continue to improve. Notice that as the level of concurrency increases, the spin lock's throughput degrades in an approximately linear fashion.

5.1 Producer/Consumer Buffers

Next, we compare the performance of several producer/consumer buffers. Each implementation has 8 producer processes and 8 consumer processes. We consider buffers with networks of width 8, 4, and 2. The width-2 implementation is simply a pair of counters protected by spin locks. As a final control, we tested a circular buffer protected by a single spin lock, a structure that permits no concurrency between producers and consumers. Figure 5 shows the time in seconds needed to produce and consume 2^{20} tokens. Not surprisingly, the single spin-lock implementation is much slower than any of the others. The width-2 network is heavily oversaturated, the bitonic width-4 network is slightly oversaturated, while the others are undersaturated.

6 Acknowledgments

Orli Waarts made many important remarks and observations. Our thanks to Heather Woll, Eli Gafni and Shanghua Teng for helpful discussions. The first and third authors also wish to thank David for being quiet during phone calls.

References

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

[2] M. Ajtai, J. Komlos and E. Szemeredi. An $O(n \log n)$ sorting network. In *Proceedings of the 15th ACM Symposium on the Theory of Computing*, 1-9, 1983.

[3] T.E. Anderson. The performance implications of spin-waiting alternatives for shared-memory multiprocessors. Technical Report 89-04-03, University of Washington, Seattle, WA 98195, April 1989. To appear, IEEE Transactions on Parallel and Distributed Systems.

[4] K.E. Batcher. Sorting networks and their applications. In *Proceedings of AFIPS Joint Computer Conference*, 32:338-334, 1968.

[5] T.H. Cormen, C.E. Leiserson, and R. L. Rivest. Introduction to Algorithms. MIT Press, Cambridge MA, 1990.

[6] C.S. Ellis and T.J. Olson. Algorithms for parallel memory allocation. *Journal of Parallel Programming*, 17(4):303–345, August 1988.

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

[8] G.H. Pfister et al. The IBM research parallel processor prototype (RP3): introduction and architecture. In *International Conference on Parallel Processing*, 1985.

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

[10] J. Goodman, M. Vernon, and P. Woest. A set of efficient synchronization primitives for a large-scale shared-memory multiprocessor. In *3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[11] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer – designing an mimd parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1984.

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

[13] D. Kranz, R. Halstead, and E. Mohr. "Mul-T, A High-Performance Parallel Lisp", *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989, pp. 81–90.

[14] C.P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. In *Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1986.

[15] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[16] J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. Technical Report Technical Report 342, University of Rochester, Rochester, NY 14627, April 1990.

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

[18] D. Peleg and E. Upfal. The token distribution problem. In *27th IEEE Symposium on Foundations of Computer Science*, October 1986.

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

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

[21] U. Vishkin. A parallel-design distributed-implementation (PDDI) general purpose computer. *Theoretical Computer Science*, 32:157–172, 1984.

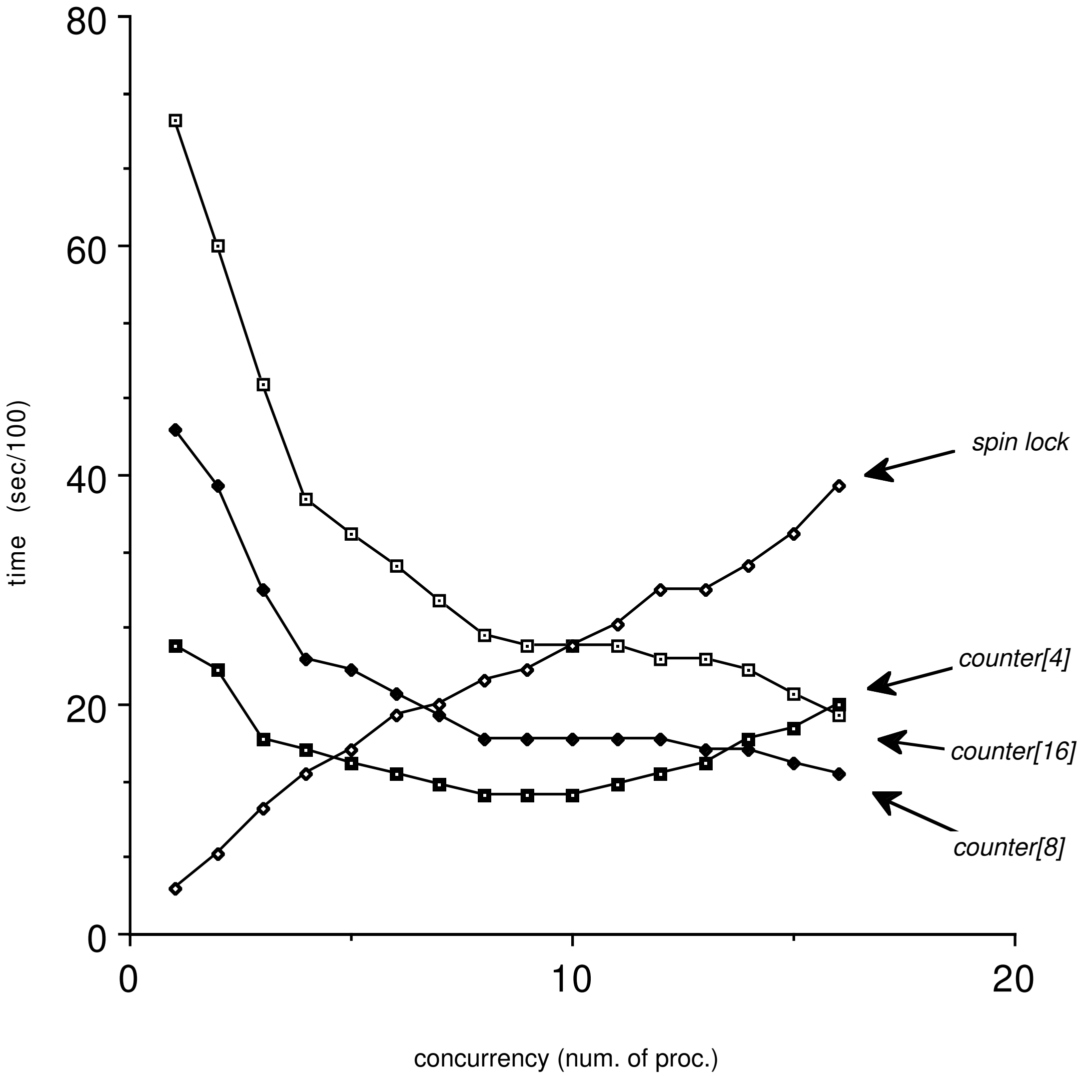

Figure 4: Shared Counter Implementations

	spin	2	4	8
time (secs)	57.74	17.51	10.44	14.25

Figure 5: Producer/Consumer Buffer Implementations