Contention in Shared Memory Algorithms

Cynthia Dwork[1] Maurice Herlihy[2]
Orli Waarts[3]

Digital Equipment Corporation
Cambridge Research Lab

CRL 93/12 August 6, 1993

[1]IBM Almaden Research Center (dwork@almaden.ibm.com)

[2]herlihy@crl.dec.com

[3]IBM Almaden Research Center (orli@cs.stanford.edu). During part of this research the third author was in Stanford University and supported by IBM Graduate fellowship, U.S. Army Research Office Grant DAAL-03-91-G-0102, NSF Grant CCR-8814921, and ONR Contract N00014-88-K-0166.

Abstract

Most complexity measures for concurrent algorithms for asynchronous shared-memory architectures focus on process steps and memory consumption. In practice, however, performance of multiprocessor algorithms is heavily influenced by *contention*, the extent to which processes access the same location at the same time. Nevertheless, even though contention is one of the principal considerations affecting the performance of real algorithms on real multiprocessors, there are no formal tools for analyzing the contention of asynchronous shared-memory algorithms.

This paper introduces the first formal complexity model for contention in multiprocessors. We focus on the standard multiprocessor architecture in which n asynchronous processes communicate by applying *read*, *write*, and *read-modify-write* operations to a shared memory. We use our model to derive two kinds of results: (1) lower bounds on contention for well known basic problems such as agreement and mutual exclusion, and (2) trade-offs between latency (maximal number of accesses to shared variables performed by a single process in executing the algorithm) and contention for these algorithms. Furthermore, we give the first formal performance analysis of counting networks, a class of concurrent data structures implementing shared counters. Experiments indicate that certain counting networks outperform conventional single-variable counters at high levels of contention. Our analysis provides the first formal explanation for this phenomenon.

A preliminary version of this work appeared in the 1993 ACM Symposium on Theory of Computing.

1 Introduction

Most complexity measures for concurrent algorithms for the asynchronous shared-memory model focus on process steps and memory consumption. In practice, however, performance is heavily influenced by *contention*, the extent to which processes access the same location simultaneously. Because of limitations on processor-to-memory bandwidth, performance suffers when too many processes attempt to access the same memory location simultaneously. In shared-bus architectures, for example, simultaneous attempts to access the same shared variable may saturate the bus, resulting in substantial delay [4]. In a network-based architecture, simultaneous attempts to access the same memory module may overload certain network switches, also resulting in delay. The phenomenon of memory contention is well-known to practitioners, and a variety of *ad-hoc* mechanisms are used in practice to reduce contention[1]. Nevertheless, even though contention is one of the principal considerations affecting the performance of real algorithms on real multiprocessors, no formal theoretical tools are available to analyze the contention produced by asynchronous shared-memory algorithms. Consequently, although the standard shared-memory model is useful for developing concurrent algorithms, it does not provide a satisfactory performance model.

This paper introduces for the first time a formal complexity model for contention in shared-memory multiprocessors. We use our model to derive two kinds of results: (1) lower bounds on contention for well known common problems such as agreement and mutual exclusion, and (2) trade-offs between latency (maximal number of accesses to shared variables performed by a single process in executing the algorithm) and contention for these problems. Informally, if you want to reduce contention when concurrency is high, you must pay a certain cost even when concurrency is low, and vice-versa. Moreover, we give for the first time a formal performance analysis of counting networks, a class of concurrent data structures that provide efficient high-concurrency shared counters that has been the subject of much recent research [2, 7, 29, 30, 32].

More specifically, we focus on a multiple instruction/multiple data (MIMD) architecture in which n asynchronous processes communicate by applying

[1] Examples include test-and-test-and-set locks [39], exponential backoff [4, 36], combining networks [25, 37], and clever algorithms for spin locks and barriers [4, 27, 35].

read, *write*, and *read-modify-write* operations to a shared memory. A read-modify-write operation atomically reads a value v from a memory location, writes back $f(v)$, where f is a predefined function, and returns v to the caller. Nearly all modern processor architectures support some form of read-modify-write for interprocess synchronization. Common read-modify-write instructions include *test-and-set*, memory-to-register *swap*, *fetch-and-add* [25], *compare-and-swap* [31], and *load-linked/store-conditional* instructions [13, 41]. Asynchrony means that there is no bound on processes' relative speeds. In real shared-memory multiprocessors, sources of asynchrony include page faults, cache misses, scheduling preemption, clock skew, variation in instruction speeds, and perhaps even processor failure.

In our model, simultaneous accesses to a single memory location are serialized: only one operation succeeds at a time, and other pending operations must *stall*. Our measure of contention is simply the worst-case number of stalls that can be induced by an adversary scheduler. This model (like all complexity models) is an abstraction of how real machines actually behave. Nevertheless, we believe it is accurate enough to make useful comparisons, and simple enough to be tractable. In particular, this model is well-suited for comparing alternative algorithms, and for deriving lower and upper bounds.

We analyze contention in several fundamental shared-memory algorithms. First, we derive tight or, in some cases, nearly tight asymptotic bounds on the contention produced by several classes of counting networks studied in the literature. In each case we show that the contention in the counting network is indeed substantially lower than the contention incurred by the conventional single-variable implementation of a shared counter. Experiments have shown that certain counting networks outperform conventional single-variable counters at high levels of concurrency [7, 29]. Our results explain this phenomenon.

The *consensus* problem [23] is fundamental to synchronization without mutual exclusion and lies at the heart of the more general problem of constructing highly concurrent data structures [28]. We give the first bounds on contention in shared-memory algorithms for consensus. The bounds are tight: $\Theta(n)$ stalls per process, where n is the number of processes participating in the protocol. Bounds for consensus imply lower bounds for a variety of more complex data structures and protocols. The *randomized consensus* problem [1, 5, 6, 8, 10, 11, 19, 38] is a variation of consensus which is required to terminate in finite expected time (instead of finite time). Randomization has a surprising effect: it allows contention to be traded against latency. The contention c can vary from $\Theta(n)$ to $\Theta(1)$, but the latency is at least $(n-1)/c$.

Next we show lower bounds on contention for n-process mutual exclusion, and we show that this problem also has an inherent latency/contention trade-off. In contrast to consensus, mutual exclusion is not required to be non-blocking: processes are allowed to wait for each other. In fact, any solution to the mutual exclusion problem necessarily requires waiting. Intuitively, this will yield a weaker latency/contention trade-off. We define *one-shot* mutual exclusion, a subproblem of mutual exclusion that must be solved by any mutual exclusion protocol, but that does not *a fortiori* require waiting, and show that for any one-shot mutual exclusion algorithm with contention c the latency is at least $\Omega(\log n/c)$.

The remainder of the paper is organized as follows. Section 2 describes our formal model of contention. Section 3 presents our performance analysis of the bitonic, periodic, and linearizable counting networks, and compares their performance with the naïve single shared-variable solution. Section 4 derives lower bounds for contention and latency/contention trade-offs inherent in consensus. Section 5 analyses the latency/contention trade-offs inherent in mutual exclusion. Section 6 closes with a discussion.

2 Model

We consider a multiple instruction/multiple data (MIMD) architecture in which n asynchronous processes communicate by applying *read*, *write*, and *read-modify-write* operations to a shared memory. A read-modify-write operation atomically reads a value v from a memory location, writes back $f(v)$, where f is a predefined function, and returns v to the caller. Asynchrony means that there is no bound on processes' relative speeds.

Algorithms in this model are often viewed as a game played between a set of processes and an adversary *scheduler*. Each process takes two kinds of steps:

1 *Invocation:* A process may initiate a memory operation. Once initiated, an operation is *pending*. A process may have only one pending operation at a time.

2 *Response:* A process may receive the response to a pending memory operation.

An adversary *scheduler* chooses how steps of different processes are interleaved. At each step, the scheduler selects the next process to run. The scheduler can take into account the system history and the processes' internal states.

To model contention, we introduce a third kind of step:

3 *Stall:* A process with a pending operation may be delayed by contention with other processes simultaneously trying to access the same location.

Although the adversary scheduler can dynamically exploit knowledge of the processes' algorithms and states, it is nevertheless subject to the following basic constraint. If process P has an operation pending to a variable v, then P incurs a stall if and only if a process Q, with an operation pending at v, receives a response. Informally, the adversary cannot stall everyone; it must allow one process at a time to succeed. If several processes have operations pending at v and one of them receives a response, then all the others incur a stall.

Stall steps are our measure of contention. An adversary scheduler maximizes contention by maximizing the number of stalls. More formally, an *n-process algorithm* is an algorithm in which up to n concurrent processes may participate. We define the *contention of an n-process algorithm* as the worst case over all executions of the ratio of the number of stalls that can be induced by an adversary scheduler divided by n. The performance of an algorithm may also be limited by conflicts at certain widely-shared memory locations, often called *hot spots* [37]. Thus we define the *variable-contention* of an n-process algorithm to be the worst case number of concurrent accesses to any single variable occurring during an execution of the algorithm. Variable-contention can also be viewed as the contribution of a single variable to the overall contention of the algorithm. Next, the *contention of a concurrent object with concurrency n* is defined as the worst case, over all executions of at most n concurrent processes, of the ratio of the number of stalls occurring over multiple (potentially concurrent) accesses to the object, divided by the number of accesses to the object. The performance of a concurrent object depends mainly on the *amortized* contention, defined as the limit to which the contention of the object goes when the number of accesses goes to infinity.

To illustrate our measures, note that if n processes apply a read-modify-write operation to variable v at the same time, then the cost of completing all these operations, measured in stalls, is at least $n(n-1)/2$, or $O(n^2)$. With n pending operations the adversary can charge $n-1$ stalls, but it must allow one pending operation to return. With the $n-1$ remaining pending operations, the adversary may then charge $n-2$ stalls, and so on. Performance may be worse if processes whose operations have responded return to apply additional operations to v, but the total cost is always $O(n)$ stalls *per access*. Clearly this bound is tight. Therefore, we say that the amortized contention of a read-modify-write operation is $\theta(n)$. In other words, our model implies that the performance of read-modify-write operations degrades linearly with the degree of concurrency, behavior that has been observed experimentally [4]. We assume that concurrent operations applied to the same memory location are not combined (cf. [25]), although it is straightforward to adapt our model to such architectures.

Finally, for any asynchronous algorithm, the *latency* of the algorithm is the maximal number of accesses to shared variables performed by a single process in executing the algorithm.

3 Counting Networks

Many fundamental multiprocessor coordination problems can be expressed as *counting* problems: processors collectively assign themselves successive values from a given range, such as addresses in memory or destinations on an interconnection network. Applications include implementing a shared counter, load balancing, and barrier synchronization. Counting networks are a class of concurrent data structures that can be used to count.

In this section, we give a formal performance analysis of several counting networks. First, we show that the amortized contention of the bitonic counting network [7] is much lower than the conventional solution in which all n processors increment a single shared variable using a read-modify-write operation. This result explains why counting networks outperform single-variable counters in experiments [7, 29]. We also give tight bounds for contention in linearizable counting networks [30], an extension of the standard counting networks in which the order of the values assigned reflects the real-time order of the assignment operations, and nearly tight bounds for the *periodic* counting network [7].

3.1 Brief Review

This section gives a brief informal review of counting networks. For more details, see [7].

A *counting network*, like a sorting network [15], is a directed graph whose nodes are simple computing elements called *balancers*, and whose edges are called *wires*. Each *token* (input item) enters on an input wire, traverses a sequence of balancers, and leaves on an output wire. Unlike a sorting network, tokens can enter a counting network at arbitrary times, they may be distributed unevenly among the input wires, and they propagate through the network asynchronously.

A *balancer* can be viewed as a computing element with two input wires and two output wires, referred to as the *upper* and *lower* wires. Informally, a balancer is a toggle, sending input tokens alternately to the upper and lower wires.

A *balancing network* of width w is a collection of balancers, where output wires are 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 $y_0, y_1, .., y_{w-1}$ (also unconnected), and containing no cycles. 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 $\sum_{i=0}^{w-1} y_i = m$.

In a MIMD shared-memory multiprocessor, a balancing network is implemented as a data structure, where balancers are records and wires are pointers from one record to another. Each of the machine's n asynchronous processors runs a program that repeatedly traverses the data structure, each time shepherding a new token through the network. Tokens generated by processor P enter the network on input wire $P \bmod w$, and a processor can push at most one token through the network at any time. Thus, the limitation on the number of concurrent processors translates into a limitation on the number of tokens concurrently traversing the network:

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

The *depth* of a balancing network is the maximal depth of any wire, where the depth of a wire is defined as 0 for a network input wire, and $1 + \max_{i \in \{0,1\}} depth(x_i)$ for the output wires of a balancer having input wires x_i, $i \in \{0, 1\}$. A *layer of depth d* is defined as the set of balancers at depth d.

A *counting network* of width w is a balancing network whose outputs $y_0, .., y_{w-1}$ have the *step property* in quiescent states: $0 \leq y_i - y_j \leq 1$ for any $i < j$.

The *bitonic* counting network [7] is a specific counting network that is isomorphic to Batcher's bitonic sorting network [9]. It is constructed recursively as follows: to construct a bitonic network of width $2w$, one first constructs two separate bitonic networks of width w each and then merges their two output sequences using a width $2w$ balancing network called a *merger*. The merger is constructed to guarantee the step property on its outputs in a quiescent state, provided each of its input sequences has the step property. This construction gives a network consisting of $O(log^2 w)$ layers, each consisting of $w/2$ balancers. Note that a single balancer is both a merger and a counter of width 2.

In experiments, the bitonic counting network substantially outperformed conventional techniques for implementing a shared counter, such as spin locks and queue locks on a single shared variable, or software combining trees, on several different multiprocessor architectures [7, 29]. We now present theoretical analysis of this phenomenon.

3.2 Contention in the Bitonic Counting Network

In this section we show tight asymptotic bounds for the amortized contention in the bitonic network. In particular, we show that for a bitonic network of width w with n concurrent processors, the amortized contention of a layer is $\Theta(n/w)$. In other words, the worst case number of stalls occurring at this layer when m tokens traverse the counting network, divided by m, goes to $\Theta(n/w)$ when m goes to infinity. Since a token traverses exactly $\Theta(\log^2 w)$ layers when it traverses the network, the amortized contention of the network is at most $O(\frac{n}{w} \log^2 w)$. (That is, the worst case number of stalls occurring at the network when m tokens traverse the counting network, divided by m, approaches $O(\frac{n}{w}\log^2 w)$ when m goes to infinity.) In a separate argument, we show an execution with amortized contention $\Omega(\frac{n}{w}\log^2 w)$, so the bounds are tight. By comparison, in a single-variable counter, up to n processes may be performing concurrent increments, so one increment has contention $\Theta(n)$.

Having bounds on the amortized contention, the overall performance of the bitonic counting network can now be compared with that of the single-variable solution as follows. The amortized cost of traversing the network is the sum of the number of shared variables a process has to access and the amortized contention. In the single-variable solution, where the network consists of just the one shared variable, this cost is $\Theta(n)$. In the bitonic counting network, our result on the amortized contention shows that the overall amortized cost is $\Theta(\frac{n}{w}\log^2 w)$. This cost is minimized when $w = n$, yielding $\Theta(\log^2 n)$.

Notice that the *temporary* contention of a layer may be quite high. It is always possible to accumulate all n concurrent processors on one balancer. For example, take a bitonic network with eight input wires and eight processors. Let eight tokens traverse it. Two of them must arrive at the rightmost upper balancer; halt them and let the others exit the network. Next re-enter the other six processors. Two of them will reach the contended balancer; halt these two and let the others exit. Now we have accumulated four tokens at one balancer. We can continue in this fashion until all n processors contend for the same balancer, thereby reaching contention of $\Omega(n)$ at that layer. In fact, temporary contention of $\Omega(n)$ can similarly be created for any counting network. Nevertheless, the *amortized* contention remains low. The intuition, which must be proved, is that if the adversary creates locally high contention, it must have let many tokens traverse the network, yielding a low amortized contention.

Henceforth, we consider a bitonic network of width w with n concurrent processors. We will show that the amortized contention of a layer is $O(n/w)$. Since the number of layers is $O(\log^2 w)$ the bound of $O(\frac{n}{w}\log^2 w)$ follows. Recall that on its way through a network of width w, a token first passes through a counting network $C_{\frac{w}{2}}$ of width $\frac{w}{2}$, and then through a merger M_w of width w. If we continue to unwind the recursive construction of $C_{\frac{w}{2}}$, and recall that $C_2 = M_2$ consists of a single balancer, we see that the token passes sequentially through a series of $\log w$ mergers $M_2, M_4, M_8, \ldots, M_w$. It therefore suffices to show that, for any $2 \leq k \leq w$, where k is a power of

2, and any layer ℓ of M_k, a token encounters "on average" at most $O(n/w)$ stalls as it passes through a balancer at layer ℓ of M_k.

More specifically, for any merger M_k in the recursive construction, and any layer ℓ of M_k, we argue as follows. By construction, the number of balancers in layer ℓ is $k/2$. We define $n_k = k\frac{n}{w}$, and partition the tokens arriving at layer ℓ, over the lifetime of the system, into *generations* of size k. We will show that as a group, each generation of tokens at layer ℓ causes $O(n_k)$ stalls to other tokens. It then follows that an average generation receives $O(n_k)$ stalls. (If 10 people each throw 5 balls into the air, and all the balls are caught, then the average person catches 5 balls.) Dividing by the number of tokens in a generation, it follows that the average token passing through ℓ receives $O(\frac{n_k}{k}) = O(n/w)$ stalls.

A layer l of M_k of C_k has the *balancer i-smoothing property* if for every pair of balancers b, b' in l, when C_k is in a quiescent state, the absolute value of the difference between the total number of tokens that have passed through b and the total number of tokens that have passed through b' is bounded by i. A layer l of a balancing network has the *input wire i-smoothing property* if for any two wires w and w', inputs to layer l, when the network is in a quiescent state the total number of tokens that have arrived at level l on wire w and the total number of tokens that have arrived at level l on wire w' differ by at most i. The *output wire balancing property* is defined analogously.

Lemma 3.1 *Fix a network C_k in the recursive construction of C_w, and let M_k be the merger of C_k. Then every layer l of M_k has the balancer 2-smoothing property.*

Proof: We split the proof into two cases, according to whether l is the first layer of M_k or is a later layer.

Claim 3.1 *The first layer of M_k has the balancer 1-smoothing property.*

Proof: Let b and b' be any two balancers of layer l. Since l is the first layer of M_k, both b and b' have one input wire from the upper $C_{k/2}$ and one from the lower $C_{k/2}$. When C_k is in a quiescent state, all the enclosed subnetworks are quiescent, so in particular both copies of $C_{k/2}$ are in a quiescent state and therefore their outputs enjoy the step property. Without loss of generality, let the upper input wire of b be higher than (have smaller index than) the upper input wire of b'. By the construction of M_k, this means that the lower input wire of b is lower than (has greater index than) the lower input wire of b'. Let x and y denote the total number of tokens that entered b on its upper and lower input wires, respectively. Similarly, let z and w denote the number of tokens that entered b' on its upper and lower input wires, respectively. Since the upper input wires come from the upper copy of $C_{k/2}$ we have by the step property that $x \geq z \geq x - 1$; similarly, by the step property of the lower copy of $C_{k/2}$, $w \geq y \geq w - 1$. The total number of tokens that pass through b is $x + y$, while the total passing through b' is $z + w$. From the inequalities we get $x + y \geq z + w - 1$ and $z + w \geq (x - 1) + y$, from which we get a maximum difference of 1, so the claim holds. ∎

Since the balancers at the first layer of M_k have the 1-smoothing property, the output wires at the first layer have the output-wire 1-smoothing property. Consider any two balancers b and b', through which, respectively, c and $c-1$ tokens have passed. Then the number of tokens leaving b on the upper output wire is $\lceil c/2 \rceil$, while the number of tokens that have left on the lower output wire of b' is $\lfloor (c-1)/2 \rfloor$, which differ by at most 1. Moreover, since the output wires of the first layer are precisely the input wires to the second layer, we have that layer 2 of M_k has the input wire 1-smoothing property. In general, if layer l has the input wire 1-smoothing property then it has the balancer 2-smoothing property. The lemma thus follows from the following claim.

Claim 3.2 *In any balancing network, if layer l has the input wire 1-smoothing property, then so does layer $l + 1$.*

Proof: Let b and b' be arbitrary balancers in layer l. Let b receive x_0 and x_1 input tokens on its upper and lower input wires, respectively. Similarly, let b' receive x'_0 and x'_1 tokens on its input wires. The maximum number of tokens leaving on one of b's output wires is at most $\max\{x_0, x_1\}$, while the minimum number of tokens leaving on one of b''s output wires is a least $\min\{x'_0, x'_1\}$. But since layer l has the input wire 1-smoothing property, $\max\{x_0, x_1\} - \min\{x'_0, x'_1\} \leq 1$, so layer l has the output wire 1-smoothing property. Since the output wires of layer l are the input wires of layer $l + 1$, the claim follows. ∎

This completes the proof of the Lemma. ∎

Let M_k be as in the Lemma, and let b be a balancer in layer l of M_k. We say that a token belongs to the *gth generation of tokens arriving at b* if it is either the $(2g-1)$th or the $(2g)$th token to arrive at b. The gth generation of l is the set of gth generation tokens of the balancers in layer l. Note that the gth generation of l has k tokens. We say that by time t, the *gth generation has completed its arrival* at l if for each balancer b_i in l, both tokens of the gth generation have already arrived by that time. Finally, we say that at time t there are f tokens of the gth generation *missing* at layer l if by time t exactly $k - f$ tokens of generation g have arrived at l.

Fact 1: *Let C_k be in a quiescent state, and let g be the maximum generation such that some balancer b in layer l of M_k has received at least one generation g token. Then all balancers in l have received at least one generation $g-1$ token.*

Proof: Let c be the number of tokens that have arrived at b. By Lemma 3.1, layer l has the balancer 2-smoothing property, so every other balancer b' has received at least $c-2$ tokens. If $c = 2g$ then b has received both its generation g tokens and hence every other balancer b' has received at least $2g-2 = 2(g-1)$ tokens, and has therefore completed generation $g-1$. If $c = 2g-1$ then every other balancer in l has received at least $c-2 = 2(g-1)-1$ tokens. Thus in either case, every balancer in l has received at least one generation $g-1$ token. ■

Recall that $n_k = k\frac{n}{w}$. Note that n_k is the maximum number of tokens that can be traversing C_k at any time.

Fact 2: *Let t be the time at which the first gth generation token arrives at l. Then the number of tokens of generations strictly less than $g-1$ stuck at l, plus the number of tokens of generations strictly less than $g-1$ still missing from layer l, is at most n_k.*

Proof: Run the network to quiescence from its state at time t. Let g' be the maximum generation such that some balancer in layer l has received at least one generation g' token. Clearly, $g' \geq g$. By Fact 1, every balancer has received at least one token from generation $g'-1 \geq g-1$. Thus, Fact 2 follows immediately from the fact that at most n_k tokens (the maximum number of tokens in C_k at any time) were involved in moving C_k to a quiescent state. ■

Recall that the number of tokens in the gth generation at l is exactly k. As described above, to complete the proof it is enough to show that the number of stalls caused in layer l of M_k due to the gth generation is $O(n_k)$, since from this it follows that the average (over all generations) number of stalls incurred by a generation is $O(n_k)$, and therefore that the average token incurs $O(\frac{n_k}{k}) = O(n/w)$ stalls at each layer (because a token passes through just one balancer at layer l).

First recall that when a token passes through a balancer, it causes stalls to all tokens that are waiting at this balancer. By *stalls caused at layer l by generation g to generation g'* we refer to stalls incurred by tokens of generation g' when they are waiting at some balancer of layer l and some token of generation g passes. By *stalls caused at layer l between generation g and generation g'* we refer to stalls caused by generation g to generation g', and vice versa. To complete the proof we show:

Lemma 3.2 *Consider the gth generation passing through layer l of M_k. The maximal number of stalls caused between this generation and generations less than or equal to g at this layer is at most $5n_k$.*

Proof: Consider the first token of generation g to arrive at l. Say it arrives at time t. By Fact 2, the total number of tokens of generation less than $g-1$ stuck at l or missing from l is at most n_k. A generation g token can encounter (and hence cause a stall to or be stalled by) (1) these tokens of generation less than $g-1$, (2) generation $g-1$ tokens, and (3) generation g tokens. There are at most n_k tokens of type (1), and at most w_k each of types (2) and (3). The number of stalls occurring between each token of generation g and tokens of generation less than or equal to $g-1$ is at most the number of tokens of these generations that this token encounters at its balancer. Each token of generation less than or equal to $g-1$ can be encountered by up to two tokens of generation g. Each token of generation g can be encountered by at most one token of generation g. Summing, we get $2n_k$, $2w_k$, and w_k for stalls of types (1), (2), and (3), respectively, for a total of at most $5n_k$ stalls. ■

We have shown that the amortized contention endured by a token at any layer is $O(n/w)$. Amortized contention of $\Omega(n/w)$ is easily seen to occur in an execution where on each balancer we have $2n/w$ tokens proceeding in lock step. We have therefore proved the following theorem.

Theorem 3.1 *The amortized contention of a layer of bitonic network of width w and concurrency n is $\Theta(n/w)$.* ∎

Corollary 3.1 *The amortized contention of the bitonic network of width w and concurrency n is $\Theta(\frac{n}{w}\log^2 w)$.* ∎

3.3 Contention in Linearizable Counting Networks

In this section, we observe that the amortized contention of the folded linearizable counting network of width w [30] is also $\Theta(\frac{n}{w}\log^2 w)$. More specifically, a linearizable counting network is a counting network in which the order of the values assigned to processes is consistent with the real-time order of the execution. For example, if process P is assigned a value (leaves the counting network) before process Q requests one (enters the counting network), then process P's value must be less than Q's. Linearizable counting lies at the heart of a number of basic problems, such as concurrent time-stamp generation, concurrent implementations of shared counters, FIFO buffers, snapshots, and similar data structures (e.g. [21, 22, 26]).

Certain counting networks are not linearizable, and there is no linearizable counting network with finite width [30]. Two linearizable counting networks are constructed in [30]. The general idea in their approach is to have tokens first pass through an ordinary (non-linearizing) counting network and then use the resulting value (the value returned by the counter) to select an input wire into an infinite-width linearizer. Thus, if implemented directly in terms of balancers, these networks would have infinite size. However the infinite linearizers can be "folded" onto finite data structures. The *folded* network is a width w by depth d array of *multibalancers*. For this section only, let us define *layer* i of the linearizer to be the set of balancers with lower input wire of depth i. Let $c_{i,j}$ denote a multibalancer in the folded network whose upper input wire is wire i and whose layer is j; similarly, let $b_{i,j}$ denote a balancer in the infinite network whose upper input wire is i and whose layer is j. Then the folded network simulates the original network by simply having $c_{i,j}$ simulate balancers $b_{i,j}, b_{i+w,j}, b_{i+2w,j}$ and so on. Like a balancer, a multibalancer can also be represented as a record with *toggle*, *upper*, and *lower* fields. The *upper* and *lower* fields are still pointers to the neighboring multibalancers or counters, but the *toggle* component is more complex, since it encodes the toggle states of an infinite number of balancers.

Note that, since each balancer in the infinite linearizer is traversed by only two tokens, the linearizer in the infinite construction does not have high contention. However, the folding of the network introduces contention since tokens passing through different balancers in the original network may end up passing through the same multibalancer in the folded network. Here we will argue that this contention is low.

Only two points in the construction of the linearizable counting networks of [30] are necessary for the contention analysis. First, since the input to the linearizer is the output of a counting network, each layer of the folded linearizer has the input wire 1-smoothing property. Second, the tokens at a balancer may be partitioned into generations as follows. Intuitively, we view each "wave" of tokens leaving the non-linearizable counting network as a generation of inputs to the infinite linearizer. Thus, the first generation of tokens to enter layer 1 of the infinite linearizer is the set of tokens entering at wires 1 to w of the infinite linearizer, the next generation of tokens to enter layer 1 is the set of tokens arriving at wires $w+1, \ldots, 2w$, and so on. In general, the gth generation to enter a layer of the infinite network is the set of tokens entering the layer on wires $(g-1)w+1, \ldots, gw$. In the case of the folded network, this translates as follows: a token arriving at a multibalancer $c_{i,j}$ belongs to generation g of layer j if the multibalancer simulates balancer $b_{(g-1)w+i,j}$ for this token. The above two facts immediately imply that generation g tokens encounter at most n tokens from previous generations (because the number of tokens of generations at most $g-1$ missing or stuck at any time can be at most n, the upper bound on the concurrency), and at most w tokens from their own generation (because a generation contains at most w tokens by definition). Thus, Lemma 3.2 can be employed to show that the amortized contention of one layer of the folded network is $O(n/w)$.

Again, amortized contention of $\Omega(n/w)$ per layer occurs in an execution in which all n processors proceed in lock step, and hence the above bound is tight.

3.4 Contention in Other Counting Networks

First, observe that the techniques used to analyze the contention in the bitonic and linearizable counting networks consist of three main ideas:

1 Determine sequences of balancers in the network such that each sequence has the balancer k-smoothing property for some k.

2 Partition the tokens entering each substructure into generations.

3 Compute the number of stalls caused between a generation and its previous generations using the fact that at each substructure, tokens from generation g encounter at most m tokens from generations smaller than $\lceil g - k/2 \rceil$, where m is the number of concurrent processors that can enter the substructure.

It turns out that this method is widely applicable. For example, consider the periodic counting network [7]. It is isomorphic to the balanced periodic sorting network [18]. In particular, a periodic network of width w consists of a sequence of $\log w$ identical subnetworks each of which is of depth $\log w$ and called *Block[w]*. An easy induction on the depth of the layer shows that each block has the output wire $\log w$-smoothing property. Consequently, Claim 3.2 implies that each layer of depth greater than $\log w$ has the input wire $\log w$-smoothing property. Almost identical reasoning to that of Section 3.2 immediately shows that the amortized contention of each layer of the periodic counting network is at most $O(n/w + \log w)$. (To compute the above we need to distinguish between the first block and the later blocks.) Hence the amortized contention of the complete periodic network is $O(\frac{n}{w}\log^2 w + \log^3 w)$ which is minimized when $w = n$, yielding $O(\log^3 w)$.

Block[w] has the output wire $\log w$-smoothing property. Thus, we can guarantee that any counting network can be modified to have low contention by filtering its inputs through Block[w].

4 Consensus

In this section, we give lower bounds for contention and latency-contention trade-offs inherent in consensus. Consensus is fundamental to synchronization without mutual exclusion and hence lies at the heart of the more general problem of constructing highly concurrent data structures [28]. Thus the bounds and trade-offs derived here imply bounds and trade-offs for a variety of more complex data structures and protocols.

The *consensus problem* [17, 20, 23] is a decision problem in which each of n asynchronous processes starts with an input value 0 or 1 not known to the others and runs until it chooses a decision value and halts. The protocol must be *consistent:* no two processes choose different decision values; and *valid:* the decision value is some process' input value. In addition it should satisfy some form of *wait-freedom.* For this matter it is distinguished between *wait-free* consensus in which each process decides after a finite number of its own steps (accesses to shared memory) regardless of other processes' halting failures or relative speeds; and *randomized wait-free* consensus in which each process decides after a finite *expected* number of its own steps regardless again of other processes' halting failures or relative speeds.

It has been shown [17, 34] that on machines that support only reads and writes but no form of read-modify-write, consensus cannot be solved by a wait-free protocol but it does have a randomized wait-free solution [1, 5, 6, 8, 11, 19]. However, as pointed out, most real machines do have some form of read-modify-write. In Section 4.1 we show that even when read-modify-write is available, wait-free consensus has high price: its contention (and hence its cost) is inherently high. In Section 4.2 we show that for randomized wait-free consensus the contention can be traded off against latency.

4.1 Wait-Free Consensus

In this section we show that wait-free consensus has contention of $\Theta(n)$. (Recall that contention of an n-process algorithm is defined as the worst case ratio of the number of stalls that can occur in an execution of the algorithm divided by n.) In addition, the *cost* of an n-process algorithm can be naturally defined as the worst case, over all executions of the algorithm, of the ratio of the sum of the number of stalls and the number of invocations of read-modify-write occurring in the execution, divided by n. It will follow that also the cost of wait-free consensus is $\Theta(n)$.

As has been observed earlier:

Lemma 4.1 *For n processes to do a concurrent read-modify-write to a single location has contention and cost $\Theta(n)$.*

We show that any wait-free consensus protocol in our model has inherently high contention by showing that the adversary can force all n processes to simultaneously access a single shared variable. Intuitively, using a technique introduced by Fischer, Lynch, and Patterson [23], we construct a configuration of the system from which both 0 and 1 are still possible decisions but from which a step by any single process will determine the outcome. Symmetry and commutativity conditions then imply that all steps from this configuration must in fact access the same shared variable.

Following [23], a protocol state is *bivalent* if either decision value is still possible, i.e., the current execution can be extended to yield different decision values. Otherwise it is *univalent*. An *x-valent* state, for $x \in \{0, 1\}$, is a univalent state with eventual decision value x. A *decision step* is an operation that carries a protocol from a bivalent to a univalent state. The following lemma was first proved in [23].

Lemma 4.2 *For every consensus protocol there exists a bivalent initial configuration.*

Theorem 4.1 *Consensus among n processes has contention* $\Theta(n)$.

Proof: For the lower bound, consider the following scenario. By Lemma 4.2, there exists a bivalent initial configuration. Beginning with the system in this configuration, the adversary repeatedly chooses some process that is not about to take a decision step, and allows that process to execute a complete operation (invocation/response pair). This execution cannot proceed forever, since the protocol is wait-free, so eventually the protocol must enter a state where every process is about to execute an operation that will carry the protocol to a univalent state. Since the protocol is still in a bivalent state, there must be some process P about to carry the protocol to a 0-valent state, and some process Q about to carry the protocol to a 1-valent state. If P and Q are not about to execute a read-modify-write operation on the same location, then the 0-valent state in which P's operation precedes Q's is indistinguishable from the 1-valent state in which Q's operation precedes P's, a contradiction. Therefore P and Q must be about to operate on the same location. By symmetry, all n processes are about to operate on the same location. By Lemma 4.1, the adversary can, by scheduling all these processes concurrently, force contention $\Omega(n)$.

For the upper bound, simply initialize a memory location to the distinguished value $\perp$, and have each process execute *compare-and-swap*(location,$\perp$,input). ∎

Observe that in the proof of the above theorem we used the wait-freedom property to construct the critical bivalent configuration from which any step would bring the system to univalence. Randomized consensus is not wait-free, and thus, the proof of Theorem 4.1 breaks down. We address randomized consensus in Section 4.2.

A *concurrent object* is a data structure shared by concurrent processes. A concurrent object X *solves* n-process consensus if there exists a consensus protocol in which the n processes communicate by applying operations to a shared X. A *wait-free* implementation of a concurrent data object is one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes. Theorem 4.1 implies that any wait-free implementation of an object that solves n-process consensus has high contention. This implies, for example, that wait-free implementations of the fetch-and-add, compare-and-swap and the load-linked/store-conditional operations in terms of any other primitive must have high contention.

4.2 Randomized Wait-Free Consensus

The proof of Theorem 4.1 also shows that the variable-contention of wait-free consensus is $\Omega(n)$. (Recall that variable-contention of an algorithm was defined as the worst case number of concurrent accesses to any single variable occurring during the execution of the algorithm.) In this section we show that this lower bound does not hold for randomized wait-free consensus. In fact, we can construct randomized consensus protocols with $O(1)$ variable-contention. Nevertheless, we show that there is a trade-off between variable-contention of an algorithm and its latency.

First we show how to construct a randomized consensus algorithm with low variable-contention. The randomized consensus algorithms in the literature [1, 5, 6, 8, 11, 19] were designed as a way of coping with the impossibility of consensus in a model without read-modify-write [17, 34]. Thus, these algorithms require a weaker communication primitive abstracted in the literature as a *single-writer multi-reader atomic register*. Each such register can be written only by one process, its owner, but all processes can read it. The atomicity property says that reads and writes can be viewed as occurring at a single instant of time. Clearly an algorithm that uses multi-reader atomic registers may have high variable-contention if in some execution all n

readers are concurrently accessing the same register. However, there are well known constructions of single-writer multi-reader registers out of n single-writer *single*-reader atomic registers [33, 40] (a single-reader register can be read by at most one process). Therefore, to achieve randomized consensus with low variable-contention, simply take one of the algorithms that uses a multi-reader register and replace each such register by n single-reader registers. Clearly the resulting algorithm has $O(1)$ variable-contention.

Recall that the latency of the algorithm is the maximum number of steps (accesses to shared memory) that a process must take in any execution. We now show a latency-contention trade-off inherent in randomized consensus. Although randomization was introduced in the literature to replace read-modify-write, our latency-contention tradeoff holds regardless of whether or not read-modify-write is assumed.

Theorem 4.2 *Consider any randomized consensus algorithm. Let p be any process. Let ℓ be the minimum number of variables accessed by p in an execution prefix E in which p has input value 0 and reaches a decision before any of the other processes become active. Let c be the variable-contention in any execution of the algorithm. Then $\ell \geq (n-1)/c$.*

Proof: We actually prove a stronger statement. Let the *preferred path* of p be the series of variables accessed by p in E, *without repetitions*. Thus if p first accesses v then v', and then v again, the preferred path is just v, v'. Let the preferred path be $v_1, v_2, \dots v_\ell$. We show that $\ell \geq \frac{n-1}{c}$. Observe that p must decide 0 in E. Now, consider the following execution E' in which p is initially crashed and all other processes, call them $q_1, \dots, q_{n-1}$ have initial value 1. Note that the decision in E' must be 1. Run process q_1 alone until it accesses the preferred path of p. Note that it must do this, since if not then q_1 cannot distinguish between E' and an extension of E, but in any extension of E it would have had to decide 0. Temporarily suspend q_1 just before it accesses the preferred path.

Now run process q_2 until it too is about to access the preferred path. Note that q_2 may see variables written by q_1, but since these are written by q_1 before it can distinguish E' from an extension of E, q_2 cannot yet make that distinction either. Moreover, q_2 cannot wait to see things that q_1 might write after q_1 has accessed the preferred path because q_1 might have failed. In this way, we continue constructing E', until all $n-1$ processes $q \neq p$ are poised to access the preferred path. Let c_i denote the number of processes poised to access v_i. Then $\sum_{i=1}^{\ell} c_i \geq n-1$. Now, let all $n-1$ processes q access the preferred path together, and the theorem is proved. ∎

Since the latency of an algorithm is no smaller than the worst case number of variables accessed by any single process when it runs in isolation, Theorem 4.2 immediately implies that the contention of a randomized consensus algorithm is at least $(n-1)/\ell$, where ℓ is its latency.

A similar argument [30] shows part (1) of the following theorem.

Theorem 4.3 *The same trade-off holds for any linearizable counting network, randomized or not, using any primitive (including primitives more powerful than simple balancers). For linearizable counting networks, the trade-offs are tight.*

5 Mutual Exclusion

In this section we study contention in solutions to the mutual exclusion problem. In this problem, processes must repeatedly access a critical section in such a way that at any given time there is at most one process in the critical section. A solution must satisfy the following liveness property: in any execution of the protocol in which no process crashes, if any process tries to enter the critical section then eventually some process succeeds to do so.

Like consensus, mutual exclusion is an abstraction of many synchronization problems. The most common example of the need for mutual exclusion in real systems is resource allocation. In contrast to consensus, however, mutual exclusion is not required to be wait-free nor randomized wait-free; rather processes may actually wait for failed processes. (In this case processes may not even terminate with probability 1.) While this feature may not be very desirable, it allows more efficient and simpler implementations. In particular, observe that the lower bounds on the contention and latency-contention trade-off previously derived for consensus may not hold for mutual exclusion: a c-ary tournament tree clearly satisfies the liveness condition with at most $\log_c n$ accesses to shared variables for any single process (in other words, with contention c and latency $\log_c n$), thereby violating the bounds for consensus obtained in Theorems 4.1 and 4.2.

More precisely, our results will be for *one-shot mutual exclusion*, which, informally, allows exactly one of a number of initially competing processes to enter the critical section, with no requirements of the other processes. Clearly such bounds apply for mutual exclusion each time the latter is started from scratch. An algorithm for the one-shot problem clearly satisfies the liveness condition; however, unlike the case for the general mutual exclusion problem, the one-shot problem can be solved (through a tournament tree) without waiting. Thus, a lower bound here is in a sense a lower bound on achieving the liveness condition for mutual exclusion, allowing us to sidestep issues such as how waiting is implemented.[2]

First we show, for any one-shot mutual exclusion algorithm of latency ℓ and variable-contention c, that $\ell \in \Omega(\frac{\log n}{c})$.

Our proof relies on the fact that $\Omega(\log n)$ is lower bound on the time required to compute the logical OR of n values on the CREW PRAM[3] [14], independent of the total number N of processes participating in the computation. The key idea is roughly that a CREW PRAM can simulate an algorithm whose variable-contention is c so that each time c processes access the same shared variable in an execution of the original algorithm, they will access it one by one in c steps in the corresponding execution of the CREW PRAM.

The structure of the argument is as follows. The first step argues that for a CREW PRAM, any algorithm for one-shot mutual exclusion yields, with one additional step, an algorithm for OR. In the second step, we show how to construct a one-shot mutual exclusion algorithm for the CREW PRAM that takes at most $O(c\ell)$ rounds, from any asynchronous algorithm for one-shot mutual exclusion with variable-contention c and latency ℓ. The second step proceeds as follows. First, we say that a specific execution of an asynchronous algorithm is *synchronous* if it can be viewed as if it takes steps in synchronous rounds during which each process that has not yet halted accesses one shared variable and processes accessing the same shared variable (at the same round) succeed (receive responses) in increasing order of process ID. Observe that each input determines exactly one synchronous execution. To complete the second step, given an asynchronous one-shot mutual exclu-

[2]Confusion on this point in an earlier version of this paper was brought to our attention by James Anderson [3].

[3]Concurrent read/exclusive write parallel random access machines. Note that PRAM's are synchronous.

sion algorithm A, we show how to construct a CREW PRAM algorithm each of whose executions simulates the synchronous execution of A that has the same input. Moreover, each round of the synchronous execution of A will be simulated by the corresponding execution of the CREW PRAM using no more than c rounds. Clearly, the resulting CREW PRAM algorithm takes no more than $c\ell'$ rounds, where ℓ' is the maximum number of rounds of the simulated synchronous algorithm. On the other hand, the latter is no larger than A's latency because for each synchronous execution of A, some process must take a step, and hence access a shared variable, in each round.

Combining the two above steps we get that given an algorithm A that achieves one-shot mutual exclusion among n processes with variable contention c and latency ℓ, we can construct a CREW PRAM algorithm that computes the OR of n values in $O(c\ell)$ rounds. Since $\Omega(\log n)$ rounds are necessary for a CREW PRAM to compute the OR of n values, we have $\ell \in \Omega(\log n/c)$.

Definition 5.1 One-shot mutual exclusion *on n processes is defined as follows. There are any number $N \geq n$ of processes. There are n Boolean input variables $x_1, \ldots, x_n$. (These variables can be either in shared memory locations 1 through n, respectively, or for $1 \leq i \leq n$, x_i can be local to p_i. Our results apply to either version of the problem). Let S be the set of indices i such that $x_i = 1$. At the end of each execution of the algorithm there is a unique $i \in S$, such that p_i is a* winner *(that is, p_i is in a special* win *state). If S is empty then there is no winner.*

Theorem 5.1 *Let A be any algorithm for one-shot mutual exclusion, and let c be its variable-contention and ℓ its latency. Then $\ell \in \Omega(\log n/c)$.*

Proof: The next lemma shows that given a one-shot mutual exclusion algorithm for the CREW PRAM, we can get with one additional step an algorithm for OR.

Lemma 5.1 *Let S be a CREW PRAM algorithm for one-shot mutual exclusion on n inputs, running in time $s(n)$. We place no bound on the number of processors, but the mutual exclusion is among $p_1, \ldots, p_n$. Then there is a CREW PRAM algorithm for logical OR on n inputs running in time $s(n)+1$.*

Proof: The algorithm for OR is as follows. Let *result* be a special memory cell that is not used by algorithm S on any input and is initialized to zero. On inputs $x_1, \ldots, x_n$ run $\mathrm{S}(x_1, \ldots, x_n)$. Let process i be the winner, if one exists. Then at step $s(n)+1$ process i writes a "1" into memory location *result*. Since by assumption *result* is initialized to zero we have that *result* will have value 1 if and only if there is a winner to the one-shot mutual exclusion. The definition of one-shot mutual exclusion implies that there is a winner if and only if at least one process started with 1 and hence this PRAM algorithm correctly computes the OR. ■

Next we show that given an asynchronous one-shot mutual exclusion algorithm with variable-contention c and latency ℓ, we can construct a one-shot mutual exclusion algorithm for the CREW PRAM that takes $O(c\ell)$ rounds.

Lemma 5.2 *Let A be any algorithm on n inputs running on an asynchronous shared-memory machine, with variable-contention at most c, with latency ℓ, requiring at most N processes, and requiring at most $m(n) \geq n$ shared variables. Then there exists an algorithm for the synchronous CREW PRAM that requires at most $N + m(n)\max_{1\leq i\leq c}\{\binom{N}{i}\}$ processors and runs in time at most $O(c\ell)$.*

Proof: We have observed that there is exactly one synchronous execution of A for each value of the inputs. Therefore, it is enough to construct a CREW PRAM algorithm S that will simulate executions of A in a step by step fashion such that each execution of S with inputs I will have as its *corresponding* execution of A the synchronous execution of A with inputs I.

For simplicity we first describe an algorithm that runs in $O(c^2\ell)$ rounds. S is constructed as follows. It has a special set of *simulating* processors $P_1, \ldots, P_N$ whose job is principally to simulate, one for one, the processes of A. For clarity, the processors of S will always be denoted by upper case letters, while those of A will be denoted by lower case letters. The additional $m(n)\max_{1\leq i\leq c}\{\binom{N}{i}\}$ *auxiliary* processors are dedicated to resolving write conflicts at the $m(n)$ shared variables of A. Hence, the auxiliary processors are split into $m(n)$ *groups*, one for each of the $m(n)$ shared variables v of A and denote by G_v the group dedicated to location v. The processor in G_v with the smallest index is the *leader* of group G_v.

We let $M[1:m(n)]$ denote the first $m(n)$ locations of the PRAM's shared memory. After each step s of the simulation, for each $1 \leq v \leq m(n)$,

$M[v]$ contains precisely the value of shared variable v after s rounds of the corresponding synchronous execution of A. We define three additional arrays in the PRAM's shared memory: $\mathbf{LOC}[1 : N]$, $\mathbf{INDEX}[1 : N]$, and $\mathbf{FLAG}[1 : m(n)]$. Roughly speaking, when a simulating processor P_i wishes to simulate an access by p_i to a shared variable v of A, it writes the location v into the cell $\mathbf{LOC}[i]$. The **INDEX** array is used to tell processor P_i wishing to access $M[v]$ in a given simulated step, its index among the set of processors that will access $M[v]$ at this step. The **FLAG** array is used in determining, for each shared variable v and each step in the synchronous execution of A, the unique d-tuple of processes, for some $0 \leq d \leq c$, that attempt to access v concurrently in the given step.

All shared variables except possibly the first n cells of memory, are initialized to zero. If the inputs to A are initially in shared memory, then we assume they are initially in the shared memory of S. If the inputs to A are initially known to the processes of A, then we assume they are initially known to the corresponding processors of S.

Each P_i has a special component of its state containing a simulated state of p_i. We prove inductively that for each P_i, $1 \leq i \leq N$, this special component of the state of P_i is the same after $s \geq 0$ simulation steps as the state of p_i after s rounds of the corresponding synchronous execution of A; and that for each $1 \leq v \leq m(n)$, $M[v]$ contains after s simulation steps the contents of v after s synchronous rounds in the corresponding synchronous execution of A. By proper initialization the result clearly holds for $s = 0$. We now show it holds for $s + 1$, assuming it holds for s.

Step $s + 1$ is simulated as follows. First, P_i writes into $\mathbf{LOC}[i]$, the location (shared variable) that p_i accesses in round $s+1$ of the corresponding synchronous execution of A. This takes one PRAM step.

Recall that initially **LOC** is all zeros. If in the simulation of some step, P_i writes a location into $\mathbf{LOC}[i]$, then at the end of the simulation of this step P_i will set $\mathbf{LOC}[i]$ back to zero. Thus, once the simulated p_i has terminated, P_i can terminate as well, and $\mathbf{LOC}[i]$ will have the correct "location" (that is, the null location) for all subsequent steps of the simulation.

Let v be any shared variable. Since A has maximum contention c, at most c processors have written v into the array **LOC**. Each of $\binom{N}{c}$ processors in G_v is assigned a set of c cells of **LOC** to examine to see if the corresponding c processes of A would all attempt to access v in round $s + 1$ of the corresponding synchronous execution of A. If so, then the processor sets $\mathbf{FLAG}[v] := 1$ to indicate that the write-set for v has been found. This takes $c+1$ PRAM steps (c reads and 1 write). In the next PRAM step, all the processors in G_v check $\mathbf{FLAG}[v]$ to see if the write set has been found. If so, then they wait until the write sets of all other variables are found (this takes a predetermined number of rounds, computed below). If not, then each member of a predetermined subset of G_v of size $\binom{N}{c-1}$ examines $c-1$ cells of **LOC** to see if the corresponding $c-1$ processes of A would all attempt to access v in round $s+1$ of the synchronous execution of A, and, if so, sets $\mathbf{FLAG}[v] := v$. Again, all the members of G_v check to see if a write-set has been found. This continues until the write-set is found (or is found to be empty). The write-set can be found in $(c+2)+(c+1)+\ldots+1$ PRAM steps.

Once the write-set is found, the leader of G_v sorts the members of the write-set in increasing order of process id, and writes i's index in this sorted list into $\mathbf{INDEX}[i]$. This takes c PRAM steps.

In the next step, for each v the leader of G_v sets $\mathbf{FLAG}[v] := 0$, and for each i processor P_i resets $\mathbf{LOC}[i] := 0$.

In the last c PRAM steps but one in the simulation of step $s+1$, each P_i simulates p_i's access to v in order, according to $\mathbf{INDEX}[i]$. Finally, in the last PRAM step of the simulation each P_i sets $\mathbf{INDEX}[i] := 0$.

Clearly, at most $O(c^2)$ steps are required for the simulation. Moreover, it is not hard to see that a more careful use of the processors allows the write-set for a shared variable to be determined in $O(c)$ time, making the entire simulation run in $O(c)$ PRAM steps per simulated round of the synchronous execution of A. ∎

Combining the two lemmas we get that given an algorithm A that achieves one-shot mutual exclusion among n processors with variable-contention c and latency ℓ we can construct a CREW PRAM algorithm that computes the logical OR of n values in $O(c\ell)$ rounds. The theorem now follows from the fact that $\Omega(\log n)$ rounds are necessary for a CREW PRAM to compute the logical OR of n values, independent of the number of processes that participate in the computation [14]. ∎

We complete our analysis of mutual exclusion by showing that in any execution of one-shot mutual exclusion there are at least $\Omega(n)$ stalls, and hence the contention is $\Omega(1)$. (The latter does not follow from the above trade-off.)

Theorem 5.2 *One-shot mutual exclusion among n processes has contention* $\Omega(1)$.

Proof: It is enough to show that for each set of k processes where $1 \leq k \leq n$, there is an execution in which there are at least $k-1$ stalls and exactly one of the participating processes enters the critical section. The proof proceeds by induction on k. The base case of $k = 1$ is trivial. Assume the claim holds for k and we will show it holds for $k+1$. Consider a set of $k+1$ processes, $p_1, \ldots, p_{k+1}$. By the inductive hypothesis, there is an execution E of processes $p_1, \ldots, p_k$ in which they incur $k-1$ stalls and exactly one of them enters the critical section. Run process p_{k+1} alone until it is about to access one of the variables, say v, accessed in execution E. Note that it must do this, since otherwise none of the $k+1$ processes $p_1, \ldots, p_{k+1}$ can distinguish between E and execution E' in which p_{k+1} runs in isolation until completion. But in E', p_{k+1} enters the critical section, while in E it cannot do it since one of $p_1, \ldots, p_k$ enters it. Temporarily suspend p_{k+1} just before it accesses v.

Next run processes $p_1, \ldots, p_k$ as in E and let p_{k+1} try to access v at the same time that the processes in E are trying to access it. There is an extension of this execution in which p_{k+1} is stalled and the other processes proceed the same way as in E. Thus, the number of stalls occuring between processes $p_1, \ldots, p_k$ is exactly the same as in E, and hence the total number of stalls increases by one, and we are done. ■

Since in a binary tournament tree the contention is $O(1)$, we have:

Theorem 5.3 *One-shot mutual exclusion among n processes has contention* $\Theta(1)$.

6 Discussion

This paper provides the first formal tools for analyzing contention in shared-memory algorithms. Taking contention into account gives a more realistic model of parallel computation. Similar considerations motivated the recent work done in [12, 16, 24].

In particular, we introduce a formal complexity model for contention in shared-memory multiprocessors and use it to provide the first formal analysis of the contention versus latency-contention trade-offs inherent in basic shared memory problems such as consensus and mutual exclusion. The results match our intuition: wait-free consensus seems to require more contention than (the easier problem of) randomized consensus. Moreover, restricting our attention to variable-contention c, randomized consensus, which is non-blocking, requires provably higher latency than one-shot mutual exclusion, a subproblem (not requiring waiting) of the mutual exclusion problem whose solution must involve waiting.

We also give the first formal performance analysis for counting networks. In particular we show that the amortized contention of the bitonic counting network is low. Our analysis clarifies experimental results showing that the bitonic network outperforms the conventional single-variable solution at high levels of contention. Using the same techniques, similar results are obtained for linearizable counting networks [30] and the periodic counting network [7].

Our work raises many new questions. The contention (and hence the performance) of many well known problems such as reader-writer synchronization, snapshots and approximate agreement is still unknown. Furthermore, the contention of counting networks in general (as opposed to specific constructions of counting networks) is also still a mystery. We believe the techniques developed in this paper can be extended to answer these new questions. We leave these issues for further research.

7 Acknowledgments

The authors are indebted to Serge Plotkin for many helpful discussions, particularly with regard to the definition of the model for contention. We are also grateful to Butler Lampson, for his input on the same subject, and to James Anderson and Faith Fich for their comments on Section 5.

References

[1] K. Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, pages 291–302, August 1988.

[2] E. Aharonson and H. Attiya. Counting networks with arbitrary fan-out. In *Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms*, January 1992.

[3] J. Anderson. Private Communication, June, 1993.

[4] T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[5] J. Aspnes. Time- and space-efficient randomized consensus. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 325–332, August 1990.

[6] J. Aspnes and M.P. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–460, September 1990.

[7] J. Aspnes, M.P. Herlihy, and N. Shavit. Counting networks and multiprocessor coordination. In *Proceedings of the 23rd Symposium on Theory of Computing*, pages 348–358, May 1991.

[8] J. Aspnes and O. Waarts. Randomized consensus in expected $O(n \log^2 n)$ operations per processor. In *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science*, pages 137–146, October 1992.

[9] K.E. Batcher. Sorting networks and their applications. In *Proceedings of AFIPS Joint Computer Conference*, pages 338–334, 1968.

[10] M. Ben-Or. Another advantage of free choice: completely asynchronous agreement protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 27–30, 1983.

[11] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.

[12] M. Choy and A.K. Singh. Adaptive solutions to the mutual exclusion problem. Manuscript.

[13] MIPS Computer Company. The MIPS RISC architecture.

[14] S. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal*, 15(1), February 1986.

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

[16] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, E. Santos, K.E. Schauser, R.M. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, May 1993.

[17] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[18] M. Dowd, Y. Perl, L. Rudolph, and M. Saks. The periodic balanced sorting network. *ACM Transactions on Programming Languages and Systems*, 36(4):738–757, October 1989.

[19] C. Dwork, M.P. Herlihy, S.A. Plotkin, and O. Waarts. Time-lapse snapshots. In *Proceedings of the 1st Israel Symposium on the theory of Computing and Systems*, May 1992, pages 154–170, Lecture Notes in Computer Science #601, Springer-Verlag.

[20] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):228–323, April 1988.

[21] C. Dwork and O. Waarts. Randomized snapshot in linear time. Manuscript.

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

[23] M. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[24] P.B. Gibbons, Y. Matias, and V. Ramachandran. QRQW: Accounting for concurrency in PRAMs and Asynchronous PRAMs. Technical Report, AT&T Bell Laboratories, Murray Hill, NJ, March 1993.

[25] 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.

[26] 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.

[27] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–70, June 1990.

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

[29] M.P. Herlihy, B-H. Lim, and N. Shavit. Low contention load balancing on large-scale multiprocessors. In *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, July 1992.

[30] M.P. Herlihy, N. Shavit, and O. Waarts. Low contention linearizable counting. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, pages 526–535, October 1991.

[31] IBM. System/370 principles of operation. Order Number GA22-7000.

[32] M. Klugerman and C.G. Plaxton. Small-depth counting networks. In *Proceedings of the 24th ACM Symposium on the Theory of Computing*, pages 417–427, May 1992.

[33] M. Li, J. Tromp, and P.M.B. Vitanyi. How to share concurrent wait-free variables. *CWI Technical Report CS-R8916*, 1989.

[34] M.C. Loui and H.H. Abu-Amara. *Memory Requirements for Agreement Among Unreliable Asynchronous Processes*, volume 4, pages 163–183. JAI Press, 1987.

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

[36] R. Metcalfe and D. Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[37] 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.

[38] M. Rabin. Randomized Byzantine generals. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, pages 403–409, 1983.

[39] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.

[40] A.K. Singh, J.H. Anderson, and M.G. Gouda. The elusive atomic register revisited. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 206–221, August 1987.

[41] R.L. Sites. *Alpha Architecture Reference Manual.* Digital Press, Maynard, MA, 1992.