Contention in Shared Memory Algorithms

CYNTHIA DWORK

IBM Almaden Research Center, Yorktown Heights, New York

MAURICE HERLIHY

Brown University, Providence, Rhode Island

AND

ORLI WAARTS

University of California at Berkeley, Berkeley, California

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 shared-memory 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. To illustrate the utility of our model, we use it 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 the length of the critical path (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 contention analysis of a variety of counting networks, a class of concurrent data structures implementing shared counters. Experiments indicate that certain counting networks

---

A preliminary version of this work appeared in *Proceedings of the 25th Annual ACM Symposium on Theory of Computing* (San Diego, Calif., May 16–18). ACM, New York, 1993, pp. 174–183.

Part of this research was performed while M. Herlihy was at Digital Equipment Corporation's Cambridge Research Laboratory and O. Waarts was in Stanford University.

O. Waarts was supported by an IBM Graduate fellowship, U.S. Army Research Office Grant DAAL-03-91-G-0102, National Science Foundation (NSF) grant CCR 88-14921, and Office of Naval Research (ONR) contract N00014-88-K-0166.

Part of this research was performed while O. Waarts was at IBM Almaden Research Center.

Authors' present addresses: C. Dwork, IBM Almaden Research Center, Yorktown Heights, NY, e-mail: dwork@almaden.ibm.com; M. Herlihy, Computer Science Department, Brown University, Box 1910, Providence, RI 02912, e-mail: herlihy@cs.brown.edu; O. Waarts, University of California at Berkeley, Berkeley, CA, e-mail: waarts@cs.berkeley.edu.

outperform conventional single-variable counters at high levels of contention. Our analysis provides the first formal model explaining this phenomenon.

Categories and Subject Descriptors: F.2.m [**Complexity Classes**]: Miscellaneous

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Contention, counting networks, mutual exclusion

1. *Introduction*

Most complexity measures for concurrent shared-memory algorithms focus on process steps and memory consumption. In practice, however, performance of programs on shared-memory multiprocessors 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. The phenomenon of memory contention is well known to practitioners, and a variety of mechanisms are used in practice to reduce contention.[1] Nevertheless, even though contention is one of the principal considerations affecting the performance of real programs on real multiprocessors, no formal theoretical model exists for analyzing the contention of asynchronous shared-memory algorithms.

This paper introduces a formal complexity model for contention in shared-memory multiprocessors. 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. To illustrate the power of the model, we use it to derive two kinds of results: (1) lower bounds on contention for well-known problems such as agreement and mutual exclusion, and (2) trade-offs between the critical path (maximal number of accesses to shared variables performed by a single process in executing the algorithm) and contention. Informally, these trade-offs capture the notion that one can reduce contention when concurrency is high only by paying a cost when concurrency is low, and vice-versa. In addition, we give for the first time a formal contention analysis of a variety of counting networks, a class of low-contention data structures that implement highly-concurrent shared counters.[2]

We focus on 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. Nearly all modern multiprocessor architectures support some form of read-modify-write operation. Examples include *test-and-set*, memory-to-register *swap*, *fetch-and-add* [Gottlieb et al. 1984], *compare-and-swap* [IBM], and *load-linked/store-conditional* instructions [MIPS Computer Company; Digital Equipment Corporation 1992]. Asynchrony

[1] Examples include test-and-test-and-set locks [Rudolph 1983], exponential backoff [Anderson 1990; Metcalfe and Boggs 1976], combining networks [Gottlieb et al. 1984; Pfister and Norton 1985], and clever algorithms for spin locks and barriers [Anderson 1990; Graunke and Thakkar 1990; Mellor-Crummey and Scott 1990].

[2] See, for example, Aspnes et al. [1991], Aharonson and Attiya [1992], Aiello et al. [1994], Busch et al. [1994], Busch and Mavronicolas [1994], Hardavellas et al. [1993], Herlihy et al. [1992], Herlihy et al. [1991], and Klugerman and Plaxton [1992].

means that there is no bound on processes' relative speeds. In modern 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 occur.

To illustrate the use of this model, we analyze contention in several fundamental shared-memory algorithms and data structures. 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 a straightforward single-variable implementation of a shared counter. Recent experiments have shown that the bitonic counting network substantially out-performs conventional single-variable counters at high levels of concurrency [Herlihy et al. 1992]. This paper presents the first formal model of this phenomenon. In addition, we show that our method can be used to analyze the contention of any counting network satisfying certain smoothness properties.

The *consensus* problem [Fischer et al. 1985] is fundamental to synchronization without mutual exclusion and lies at the heart of the more general problem of constructing highly concurrent data structures [Herlihy 1991]. We give the first bounds on contention in shared-memory algorithms for consensus. The bounds are tight: wait-free consensus has contention $\Theta(n)$, 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*[3] is a variation of consensus in which processes are required to reach agreement in finite *expected* time (instead of finite time). Randomization has a surprising effect: it allows contention to be traded against critical path length. The contention c can vary from $\Theta(n)$ to $\Theta(1)$, but the length of the critical path is at least $(n - 1)/c$.

Next we show that n-process mutual exclusion also has an inherent trade-off between the contention and the length of the critical path. In contrast to consensus, mutual exclusion permits processes to wait for one another. In fact, any solution to the mutual exclusion problem necessarily requires waiting. Intuitively, this problem yields a weaker critical-path/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 protocol with contention c the length of the critical path 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 contention analysis of the bitonic, periodic, and linearizable counting networks, and compares their contention with the straightforward single shared-variable solution. It concludes by showing how the same methods can be used to analyze the contention of

[3] See, for example, Abrahamson [1988], Aspnes [1990], Aspnes and Herlihy [1990], Aspnes and Waarts [1992], Ben-Or [1983], Chor et al. [1987], Dwork et al. [1992], and Rabin [1983].

balancing networks in general, provided they have certain smoothness properties. Section 4 derives lower bounds for contention and critical-path/contention trade-offs inherent in consensus. Section 5 analyses the critical-path/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 variable, 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.

Informally, an *invocation* occurs when a process initiates a memory operation. Once initiated, an operation is *pending* until the invoking process receives a response. A process may have multiple pending operations to different shared variables, and a variable may have multiple pending operations from different processes. In a complexity model that does not take contention into account, a process is "charged" only for the number of memory operations it invokes. To capture the effects of contention, however, we need to assign a cost for simultaneous accesses to the same variable. Suppose processes P and Q both have pending operations for variable x. If Q receives a response for x before P, then we will also "charge" P for Q's response. This definition is intended to capture the notion that accesses to x are serialized at some level, and that any response to Q represents a delay for P.[4]

Formally, an *execution* is a finite or infinite sequence of *invocation* and *response* events. Each event is labeled with a process name and a variable in shared memory. An invocation event has the form $\langle P\ inv\ x\rangle$, and a response event $\langle P\ res\ x\rangle$, where P is a process id and x a shared variable. A response *matches* an invocation if their process and variable names agree. An *operation* in an execution is a pair consisting of an invocation and the next matching response.[5] If the matching response does not exist we say that the operation is pending. An execution is *well-formed* if every response event matches a unique preceding invocation event. All executions considered in this paper are assumed to be well formed.

Definition 2.1. An *operation's contention cost* in an execution is the number of response events from the same variable that occur after the operation's invocation, up to but not including the matching response (if it exists). A *process's contention cost* in an execution is the sum of the contention costs of its operations. An *execution's contention cost* is the sum over all processes of the process's contention cost.

Definition 2.2. An *operation's overall cost* is one plus its contention cost. A *process's overall cost* in an execution is the sum of the overall costs of its

[4] A modified model would be required for architectures with hardware combining [Gottlieb et al. 1984]. See Section 6 for possible extensions of our model.

[5] This formal notion of an operation in an execution should not be confused with the informal notion of a memory operation such as *read* or *write*.

operations. An *execution's overall cost* is the sum over all processes of the process's overall cost.

For example, consider the following execution.

$$\begin{array}{lll} P & \text{RMW} & x \\ Q & \text{RMW} & x \\ R & \text{RMW} & x \\ P & \text{Ok} & x \\ Q & \text{Ok} & x. \end{array} \tag{1}$$

Here, processes P, Q, and R issue concurrent *read-modify-write* operations to variable x, and P and Q receive responses. P's response is first, so it has contention cost 0 and overall cost 1. Q is charged one unit of contention cost for P's response, so it has contention cost 1 and overall cost 2. R is charged one unit contention cost for P's response, and one unit contention cost for Q's response, so it has contention cost 2 and overall cost 3. The contention cost of the execution is 3 and its overall cost is 6.

An *n-process protocol* is a program for solving a task in which up to n processes may participate.

Definition 2.3. The *contention of an n-process protocol* is the worst case, over all executions of the protocol, of the ratio of the execution's contention cost divided by n.

For example, consider the protocol $\mathcal{P}$ in which n processes apply a single read-modify-write operation to variable ℓ. It is not hard to see that $\mathcal{P}$ has contention $\Theta(n)$: if all processes execute invocations, then the first response charges one unit of contention cost to all remaining $n - 1$ processes, the second to all the remaining $n - 2$, and so on.

A protocol's performance may also be limited by conflicts at certain widely-shared memory locations, often called *hot spots* [Pfister and Norton 1985].

Definition 2.4. A protocol's *hot-spot contention* is the maximal number of pending operations for any variable in any execution.

Roughly speaking, hot-spot contention measures the contribution of an individual variable to the overall contention of the protocol. For example, the protocol $\mathcal{P}$ described above has hot-spot contention n.

It is also interesting to consider computations that need not terminate. Informally, a *concurrent object* is a data structure shared by n processes. A concurrent object is an instance of an abstract data type, having a concrete representation (typically occupying multiple memory locations), and providing a set of operations (typically invoking multiple memory operations). Unlike protocols, objects may be long-lived: in an infinite execution, a process may apply an unbounded number of operations to an object. We sometimes refer to *object operations* and *memory operations* to distinguish between the primitive operations provided by the memory, and the composite operations provided by objects.

Formally, we model an object operation in an execution as a set of memory operations.

Definition 2.5. An *object's contention in a finite execution* is the contention cost of that execution divided by the number of object operations completed. Its *contention in an infinite execution* is the limit of its contention in the finite prefixes. An object's *amortized contention* is the maximal contention for any infinite execution.

Definition 2.6. An *object's cost in a finite execution* is the overall cost of that execution divided by the number of object operations completed. Its *cost in an infinite execution* is the limit of its cost in the finite prefixes. An object's *amortized cost* is the maximal cost for any infinite execution.

For example, consider an object implemented by a single variable, having operations implemented by a single read-modify-write operation. In the worst-case execution, all n processes have pending operations, and as soon as a process receives a response it immediately issues another invocation. An execution of length k has contention cost $(n - 1) \cdot k$ (overall cost $n \cdot k$), and completes k object operations, so this object has amortized contention at least $n - 1$ (amortized cost at least n). This bound on contention is easily seen to be tight, so the amortized contention of a read-modify-write operation in our model is $\Theta(n)$, implying that the performance of read-modify-write operations degrades linearly with the degree of concurrency, behavior roughly consistent with experimental observation [Anderson 1990].

Definition 2.7. The *critical path length* of a protocol is the maximal number of memory operations performed by any single process in any protocol execution.

In our discussions, it is sometimes convenient to treat contention analysis as a game between the set of processes and an adversary scheduler that seeks to maximize contention.

3. *Counting Networks*

Many fundamental multiprocessor coordination problems can be expressed as *counting* problems: processes 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 contention analysis of several counting networks. First, we show that the amortized contention of the bitonic counting network [Aspnes et al. 1991] is much lower than the conventional solution in which all n processes increment a single shared variable using a read-modify-write operation. This model is consistent with experimental results showing that counting networks perform better than single-variable counters at high levels of concurrency [Herlihy et al. 1992]. We also give tight bounds for contention in linearizable counting networks [Herlihy et al. 1991], an extension of the standard counting networks in which the order of the values assigned reflects the real-time order of the assignment operations. We also give nearly tight bounds for the *periodic* counting network [Aspnes et al. 1991], and show how our methods can be used to analyze the contention of balancing networks in general, provided they have certain smoothness properties. Finally, we show how *any* counting

network can be transformed into a low-contention network by prepending a certain filter network.

3.1. BRIEF REVIEW. This section gives a brief informal review of counting networks. For more details, see Aspnes et al. [1991].

A *counting network*, like a sorting network [Cormen et al. 1990], 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 $x_0, x_1, \ldots, x_{w-1}$ (which are not connected to output wires of balancers), w designated output wires $y_0, y_1, \ldots, y_{w-1}$ (also unconnected), and containing no cycles. (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 tokens received on the wire.) The safety and liveness of the network follow naturally from the above network definition and the properties of balancers, namely, that balancers do not create tokens: $\sum_{i=0}^{w-1} x_i \geq \sum_{i=0}^{w-1} y_i$, and for any finite sequence of m input tokens, within finite time the network reaches a quiescent state, that is, one in which $\sum_{i=0}^{w-1} y_i = m$.

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 $1 + \max_{i \in \{0,1\}} \textit{depth}(x_i)$ 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 $y_0, \ldots, y_{w-1}$ have the *step property* in quiescent states: $0 \leq y_i - y_j \leq 1$ for any $i < j$.

The *bitonic* counting network [Aspnes et al. 1991] is a specific counting network that is isomorphic to Batcher's bitonic sorting network [Batcher 1968]. It is constructed recursively as follows: to construct a bitonic network of width $2k$, one first constructs two separate bitonic networks of width k each and then merges their two output sequences using a width $2k$ balancing network called a *merger*. Each layer of the merger consists of k balancers. Its first layer consists of balancers $b_1, \ldots, b_k$ such that the input wires of b_i are the ith and $k - i + 1$th output wires of the above first and second counting networks of width k, respectively. (Other details of the structure of the merger are not used here and hence omitted from the description.) The merger guarantees the step property on its outputs in a quiescent state, provided each of its input sequences has the step property. This construction gives a counting 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 a MIMD shared-memory multiprocessor, a balancing network is implemented as a data structure in memory. Balancers are records, wires are pointers from one record to another, and each balancer's state is a bit. Tokens generated by process P enter the network on input wire P mod w, and each process

shepherds at most one token through the network at a time. A process traverses the network by atomically complementing the balancer state (a read-modify-write operation) and using the return value to choose which pointer to follow. Contention arises when multiple processes concurrently visit the same balancer.

As reported elsewhere [Aspnes et al. 1991; Herlihy et al. 1992], in experiments the bitonic counting network proved to be substantially more scalable than conventional single-variable techniques such as spin locks, queue locks, or read-modify-write operations. It has roughly the same performance as a software combining tree [Goodman et al. 1989], which is also a low-contention data structure (although not wait-free). It has been observed experimentally that some of this performance difference can be attributed to contention, and some to the serializing effect of locks [Herlihy et al. 1992].

3.2. Contention in the Bitonic Counting Network. In this section, we show tight asymptotic bounds for amortized contention in the bitonic network. In particular, we show that for a bitonic network of width w with n concurrent processes, the amortized contention of a layer is $\Theta(n/w)$. In other words, when m tokens traverse this layer, the worst-case contention cost 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 entire network is at most $O((n/w)\log^2 w)$. (That is, the worst case contention cost of an execution in which m tokens traverse the network, divided by m, approaches $O((n/w)\log^2 w)$ as m goes to infinity.) In a separate argument, we display an execution with amortized contention $\Omega((n/w)\log^2 w)$, so the bounds are tight.

The overall cost of the bitonic counting network can now be compared with that of the single-variable solution. In a single-variable counter, up to n processes may be performing concurrent increments, so one increment has contention cost $\Theta(n)$ and overall cost $\Theta(n)$. 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. Thus, in the bitonic counting network, our amortized contention analysis shows that the amortized cost is $\Theta((n/w)\log^2 w)$. 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 processes on one balancer. For example, take a bitonic network with eight input wires and eight processes. 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 processes. 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 processes 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 at relatively little contention, yielding a low amortized contention.

Henceforth, we consider a bitonic network of width w with n concurrent processes. 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 $O((n/w)\log^2 w)$ follows. Recall that on its way through a network of width w, a token first passes through

a counting network $C_{w/2}$ of width $w/2$, and then through a merger M_w of width w. If we continue to unwind the recursive construction of $C_{w/2}$, 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 incurs "on average" contention cost at most $O(n/w)$ 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 $n_k = k(n/w)$, and partition the tokens arriving at layer ℓ, over the lifetime of the system, into *generations* of size k. We will show (Lemma 3.2.4) that roughly speaking, as a group, each generation of tokens at layer ℓ causes contention cost $O(n_k)$ to all generations combined. It then follows that an average generation is charged cost $O(n_k)$. (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 is charged cost $O((n_k/k)) = O(n/w)$.

A layer ℓ of M_k of C_k has the *balancer i-smoothness 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-smoothness 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 3.2.1. *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*-smoothness 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 3.2.2. *The first layer of M_k has the balancer* 1*-smoothness 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 $C_{k/2}$ (see description of the merger in Section 3.1). 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-smoothness property, the output wires at the first layer have the output-wire 1-smoothness property. (Because, 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-smoothness property. In general, if layer ℓ has the input wire 1-smoothness property then it has the balancer 2-smoothness property. The lemma thus follows from the following claim.

CLAIM 3.2.3. *In any balancing network, if layer ℓ has the input wire* 1*-smoothness property, then so does layer $\ell + 1$.*

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-smoothness property, $\max\{x_0, x_1\} - \min\{x'_0, x'_1\} \leq 1$, so layer ℓ has the output wire 1-smoothness property. Since the output wires of layer ℓ are the input wires of layer $\ell + 1$, 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 3.2.1, layer ℓ has the balancer 2-smoothness 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 $n_k = k(n/w)$. Note that n_k is the maximum number of concurrent 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 contention cost charged in layer ℓ of M_k due to the gth generation is $O(n_k)$, since from this it follows that the average contention cost (over all generations) incurred by a generation is $O(n_k)$, and therefore that the average token incurs contention cost $O(n_k/k) = O(n/w)$ at each layer (because a token passes through just one balancer at layer ℓ).

When a token leaves a balancer, it causes a charge of unit contention cost to all other tokens waiting at this balancer. By *contention charged at layer ℓ between generations g and g',* we refer to the contention costs incurred by tokens of generation g' waiting at a balancer of layer ℓ when a token of generation g leaves the balancer, and to the contention costs incurred by tokens of generation g waiting at a balancer of layer ℓ when a token of generation g' leaves the balancer. To complete the proof we show:

LEMMA 3.2.4. *Consider the gth generation of tokens arriving at layer ℓ of M_k. The maximal contention charged 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 generations 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 charge of contention cost or incur a charge) (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 contention cost charged between each token of generation g and tokens of generations 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, for a sum of $2(n_k + w_k)$ contention cost. Of the at most two generation g tokens at any balancer, at most one can cause a charge of contention cost to the other, for a sum of w_k contention cost over the entire layer. Summing, we get $2n_k$, $2w_k$, and w_k for contention costs of type (1), (2), and (3), respectively, for a total contention cost of at most $5n_k$. □

We have shown that the amortized contention of 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 3.2.5. *The amortized contention of a layer of bitonic network of width w and concurrency n is* $\Theta(n/w)$.

COROLLARY 3.2.6. *The amortized contention of the bitonic network of width w and concurrency n is* $\Theta(n/w \log^2 w)$.

3.3. CONTENTION IN LINEARIZABLE COUNTING NETWORKS. In this section, we observe that the amortized contention of a layer of the folded linearizable counting network of width w [Herlihy et al. 1991] is also $\Theta(n/w)$. 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., [Dwork and Waarts 1993; Ellis and Olson 1988; Gottlieb et al. 1983]).

There is no linearizable counting network with finite width [Herlihy et al. 1991], although linearizable counting constructions based on counting networks are known [Herlihy et al. 1991]. The overall structure is to have tokens first pass through an ordinary (nonlinearizable) 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 j* of the linearizer to be the set of balancers with lower input wire of depth j. 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 Herlihy et al. [1991] 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-smoothness property. Second, the tokens at a balancer may be partitioned into generations as follows: Intuitively, we view

each "wave" of tokens leaving the nonlinearizable 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 3.2.4 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 processes 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-smoothness property for some k.
(2) Partition the tokens entering each substructure into generations.
(3) Compute the contention charged 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 $g - \lceil k/2 \rceil$, where m is the number of concurrent processes that can enter the substructure.

For example, consider the periodic counting network [Aspnes et al. 1991]. It is isomorphic to the balanced periodic sorting network [Dowd et al. 1989]. 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 $\log w$-smoothness property. Consequently, Claim 3.2.3 implies that each layer of depth greater than $\log w$ has the input wire $\log w$-smoothness 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 $O((n/w)\log^2 w + \log^3 w)$ which is minimized when $w = n$, yielding $O(\log^3 w)$.

In general, given a balancing network with certain smoothness properties, one can use the above method to analyze the network's contention. For example,

Theorem 3.4.1. *Let R be a balancing network of width w and depth d with $w/2$ balancers at each layer. Assume that the set of all balancers in R have the balancer*

k-smoothness property for some k. Then the amortized contention of R is $O(((n/w) + k)d)$.

PROOF. Analogously to the proof of Fact 2, it follows:

CLAIM 3.4.2. *Let t be the time at which the first gth generation token arrives at layer* ℓ. *Then the number of tokens of generations strictly less than* $g - \lceil k/2 \rceil$ *stuck at* ℓ, *plus the number of tokens of generations strictly less than* $g - \lceil k/2 \rceil$ *still missing from* ℓ, *is at most n.*

The above claim implies that each token of generation g can encounter at most $n + \lceil k/2 \rceil w$ tokens of generations less than g. Analogously to the proof of Lemma 3.2.4, it thus follows that the maximal contention charged at layer ℓ between generation g and generations less than or equal to g is at most $O(n + kw/2)$. Since each generation consists of w tokens, the above implies that the amortized contention of a layer is $O((n/w) + (k/2))$. Thus the amortized contention of the complete network is $O(d((n/w) + (k/2)))$. □

The smoothness property stated in Theorem 3.4.1 holds with $k = O(\log w)$ [Aiello et al. 1994] for all known optimal counting networks constructions [Klugerman and Plaxton 1992; Aiello et al. 1994], yielding that all known optimal constructions have amortized contention of $O(((n/w) + \log w)\log n)$ (since their depth is $O(\log n)$), which is $O(\log^2 w)$ for $w = n$. It is not known whether such a smoothness property, or any other, holds for counting networks in general. Hence, while it follows from the above method that known constructions have low amortized contention, one must await a better understanding of the deep structure of counting networks, before finding whether counting networks in general have low amortized contention.

Nevertheless, we can transform any counting network of unknown structure, whatever its contention, into a low contention network as follows: The above Block[w] has the output wire log w-smoothness property. Thus, we can transform any counting network to have low contention by filtering its inputs through Block[w].

4. *Consensus*

In this section, we give lower bounds for hot-spot contention in wait-free consensus, as well as a critical-path/contention trade-off inherent in randomized 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 [Herlihy 1991]. 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 task* [Dolev et al. 1987; Dwork et al. 1988; Fischer et al. 1985] 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's input value. A consensus protocol is *wait-free* if each process decides after applying a finite number of memory operations, and it is *randomized wait-free* if each

process decides after applying a finite expected number of memory operations, where interleavings and failures are chosen by an adversary scheduler. (Note that this requirement implies that the protocol terminates with probability one.) Randomized consensus is a *Las Vegas* algorithm: if processes choose, they never choose distinct values.

It is well known that there is no wait-free consensus protocol if processes communicate only by reading and writing shared variables [Dolev et al. 1987; Loui and Abu-Amara 1987]. Fortunately, however, most modern architectures provide some form of read-modify-write operation. In Section 4.1, we show that even when read-modify-write operations are available, wait-free consensus has inherently high hot-spot contention.

By contrast, randomized wait-free consensus does not require read-modify-write operations.[6] In Section 4.2, we show that randomized wait-free consensus does not have inherently high hot-spot contention, but there is an inherent trade-off between hot-spot contention and critical path length, even for protocols that use read-modify-write operations.

4.1. WAIT-FREE CONSENSUS. In this section, we show that wait-free n-process consensus has contention $\Theta(n)$.

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). We now show that any wait-free n-process consensus protocol has contention $\Omega(n)$.

LEMMA 4.1.1. *An n-process protocol with hot-spot contention c has contention at least* $\Omega(c^2/n)$.

PROOF. If c operations are pending, then the first response charges unit contention cost to all other $c - 1$ processes, the next to the remaining $c - 2$ processes, and so on. The total contention cost is $\Omega(c^2)$. Dividing by n to obtain an average per process contention cost yields $\Omega(c^2/n)$. □

COROLLARY 4.1.2. *An n-process protocol with hot-spot contention n has contention at least* $\Omega(n)$.

We show that any wait-free consensus protocol has hot-spot contention $\Omega(n)$ by showing that the adversary can force all n processes simultaneously to access a single shared variable. Using a technique introduced by Fischer et al. [1985], we construct a system configuration from which both 0 and 1 are still possible decisions but from which any step by any process will determine the outcome. Symmetry and commutativity conditions then imply that all processes must be about to apply an operation to the same shared variable.

Following Fischer et al. [1985], a system configuration is *bivalent* if either decision value is still possible, that is, the current execution can be extended to yield different decision values. Otherwise, it is *univalent*. An *x-valent* configuration, for $x \in \{0, 1\}$, is a univalent configuration with eventual decision value x. A *decision step* is an operation that carries a protocol from a bivalent to a

[6] See, for example, Abrahamson [1988], Aspnes [1990], Aspnes and Herlihy [1990], Aspnes and Waarts [1992], Chor et al. [1987], and Dwork et al. [1992].

univalent configuration. The following lemma was first proved in Fischer et al. [1985].

LEMMA 4.1.3. *For every consensus protocol there exists a bivalent initial configuration*.

THEOREM 4.1.4. *Any wait-free n-process consensus protocol has hot-spot contention* $\Omega(n)$ *and contention* $\Omega(n)$.

PROOF. Consider the following scenario. By Lemma 4.1.3, there exists a bivalent initial configuration. Beginning with the system in this configuration, construct an execution by repeatedly choosing any process that is not about to take a decision step, and allow 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 configuration where each process is about to execute an operation that will carry the protocol to a univalent configuration. Since the protocol is still in a bivalent configuration, there exist processes P_0 and P_1 about to carry the system to 0-valent and 1-valent configurations, respectively. Suppose P_0 is about to apply an operation to variable v_0, and P_1 to v_1. We first observe that if Q is any process about to carry the system to a 1-valent configuration, then it must be about to apply a read-modify-write operation to v_0, for otherwise the 0-valent state in which P_0's operation precedes Q's operation is indistinguishable from the 1-valent state in which Q's operation precedes P_0's. By a symmetric argument, if R is about to carry the system to a 0-valent configuration, then it must be about to apply an operation to v_1. Together, these observations show that v_0 and v_1 cannot be distinct. If all processes issue their invocations in this configuration, then the execution has hot-spot contention $\Omega(n)$. Applying Corollary 4.1.2 yields that also the contention is $\Omega(n)$. □

Observe that the proof of Theorem 4.1.1 uses the requirement that the protocol be wait-free to construct the critical bivalent configuration from which any step would bring the system to univalence. Randomized consensus, however, is not wait-free, and so we cannot apply this proof to obtain lower bounds for the randomized problem. We address randomized consensus in Section 4.2.

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 4.1.1 implies that any wait-free implementation of an object that solves n-process consensus has high contention. This result 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. In this section, we show that randomized wait-free consensus need not have $\Theta(n)$ hot-spot contention. Indeed, we can construct randomized consensus protocols with $O(1)$ hot-spot contention. Nevertheless, a reduction in hot-spot contention incurs a cost in the length of the protocol's critical path. We will demonstrate a trade-off between a

protocol's hot-spot contention and the length of its critical path. Note that the contention of a randomized consensus protocol may be unbounded, since execution lengths may be unbounded, but that high-contention executions, like long executions, are increasingly unlikely.

First we observe that randomized consensus is possible with hot-spot contention independent of the number of processes. The randomized consensus protocols in the literature[6] were motivated by a desire to solve consensus without read-modify-write operations, so processes in these protocols communicate by applying read and write operations to shared variables. Moreover, in each of these protocols, a shared variable may be written by only one process, but may be read by multiple processes. Such a variable is called a *single-writer multi-reader atomic register*. Any multi-reader variable may cause high hot-spot contention. However, there are well-known ways to implement single-writer multi-reader atomic registers from single-writer single-reader atomic registers [Li et al. 1989; Singh et al. 1987] (a single-reader register can be read by at most one process). Therefore, one can achieve randomized consensus with low hot-spot contention simply by taking any of the multi-reader protocols and replacing each register by a single-reader implementation, yielding a randomized consensus protocol with $O(1)$ hot-spot contention.[7] Each read operation in the original protocol is now implemented by $\Omega(n)$ reads, so the reduction in contention is accompanied by an increase in the length of the critical path. In general, however, we will show that the trade-off between hot-spot contention and critical path length holds even if the protocol uses read-modify-write operations.

To analyze contention, it is convenient to treat randomized executions as if they were nondeterministic, where each nondeterministic choice represents a step taken with nonzero probability. The next lemma is immediate.

LEMMA 4.2.1. *If some finite nondeterministic execution starting from configuration s leaves the system in configuration s', then an adversary starting in s can reach s' with nonzero probability.*

A *solo* execution by P is one in which only P takes steps.

LEMMA 4.2.2. *If some finite nondeterministic execution leaves the system in configuration s, then process P must decide in some solo execution from s.*

PROOF. Suppose not. By Lemma 4.2.1, an adversary can force the system to s with nonzero probability. Once in s, the adversary can prevent the protocol from terminating by failing all processes but P. □

THEOREM 4.2.3. *Consider any randomized consensus protocol. Let ℓ be the minimum number of distinct variables any process accesses in any execution in which it reaches a decision before any other process takes a step. If the protocol has hot-spot contention c, then $\ell \geq (n-1)/c$.*

PROOF. Consider a configuration where P_1 has initial value 0, and the remaining processes $P_2, \ldots, P_n$ have initial value 1. Let E be any execution in

[7] Together with the proof of Theorem 4.1.4, similar reasoning gives a simple proof that in general a many-process RMW register cannot be constructed from few-process RMW registers. This is because any protocol using only few-process RMW registers has low hot-spot contention, while a many-process RMW register can be used to solve consensus, which has high hot-spot contention.

which P_1 decides before any of the others takes a step (because the protocol is nondeterministic, there may be many such executions). Let P_1's *preferred path* be the sequence of variables it accesses in E, *without repetition*. Thus, if P_1 applies operations to v, v', and then v again, the preferred path is v, v'. Let P_1's preferred path be $v_1, v_2, \ldots, v_\ell$. We claim that $\ell \geq (n-1)/c$. Observe that P_1 must decide 0 in E.

We now inductively construct another execution F. F_1 is the empty execution. For $1 < i \leq n$, let s_{i-1} be the system configuration at the end of F_{i-1}. Lemma 6 implies that there exists a solo execution by P_i, starting from s_{i-1}, in which either (1) P_i decides a value before applying any operations to variables in P_1's preferred path, or (2) P_i invokes an operation on a variable in P_1's preferred path. We claim that (1) is impossible. If P_i decides a value, it must choose 1, since P_1 has taken no steps, and all processes that have taken steps have initial value 1. Moreover, if P_i chooses 1 in some execution, then it does so with nonzero probability starting from s_{i-1}. After P_i decides, P_1 may execute the same steps as in E with nonzero probability, choosing 0. It follows that the adversary can cause the processes to disagree with nonzero probability.

After executing $F_n = F$, the processes $P_2, \ldots, P_n$ have each invoked an operation on some variable in P_1's critical path. If c_i processes are about to apply an operation to v_i, then $\sum_{i=1}^{\ell} c_i \geq n-1$, implying that some $c_i \geq (n-1)/\ell$, and since $c \geq c_i$, it follows that $\ell \geq (n-1)/c$. □

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, referred in the sequel by *weak liveness*: 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 in doing 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, or even randomized wait-free. If a process fails, or is delayed for a long time, then non-faulty processes will either be halted or delayed. Note that lower bounds on contention and critical-path/contention trade-off previously derived for consensus do not hold for mutual exclusion. For example, a c-ary tournament tree clearly satisfies the weak liveness condition with at most $\log_c n$ accesses to shared variables for any single process (in other words, with hot-spot contention c and critical path length $\log_c n$), thereby violating the bounds for consensus obtained in Theorems 4.1.4 and 4.2.3.

Many mutual exclusion algorithms cause processes to incur an unbounded number of memory accesses due to spinning on remote variables. A great deal of work has been done in designing algorithms to reduce the number of remote operations,[8] principally by finding ways to substitute local spinning for remote

[8] See, for example, Anderson [1990], Graunke and Thakkar [1990], Mellor-Crummey and Scott [1990], and Yang and Anderson [1993].

spinning. However, independent of the issue of local versus remote spinning, there is still an inherent synchronization that must be performed by any mutual exclusion algorithm. We analyze the contention costs of this inherent synchronization. Our principal result is a critical path length versus hot-spot contention trade-off for the *one-shot* mutual exclusion problem.

Informally, *one-shot mutual exclusion* allows exactly one of a number of initially competing processes to enter the critical section, with no requirements of the other processes. Clearly any lower bounds for the one-shot problem apply to mutual exclusion each time the latter is started from scratch. A protocol for the one-shot problem clearly satisfies the weak 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 weak liveness condition for mutual exclusion, allowing us to sidestep issues such as how waiting is implemented.

First we show, for any one-shot mutual exclusion protocol with critical path length ℓ and hot-spot contention c, that $\ell \in \Omega(\log n/c)$.

Our proof relies on the fact that $\Omega(\log n)$ is a lower bound on the time required to compute the logical OR of n values on the CREW PRAM[9] [Cook et al. 1986], independent of the total number N of processes participating in the computation. The key idea is roughly that a CREW PRAM can simulate a protocol whose hot-spot contention is c so that each time c processes access the same shared variable in an execution of the original protocol, 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 protocol for one-shot mutual exclusion yields, with one additional step, a protocol for OR. This step uses the mutual exclusion property.

In the second step, we show how to construct a one-shot mutual exclusion protocol for the CREW PRAM that takes at most $O(c\ell)$ rounds, from any asynchronous protocol for one-shot mutual exclusion with hot-spot contention c and critical path length ℓ. The second step proceeds as follows. First, we say that a specific execution of an asynchronous protocol 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 exclusion protocol $\mathcal{A}$, we show how to construct a CREW PRAM protocol each of whose executions simulates the synchronous execution of $\mathcal{A}$ that has the same input. Moreover, each round of the synchronous execution of $\mathcal{A}$ will be simulated by the corresponding execution of the CREW PRAM using no more than c rounds. Clearly, the resulting CREW PRAM protocol takes no more than $c\ell'$ rounds, where ℓ' is the maximum number of rounds of the simulated synchronous protocol. On the other hand, the latter is no larger than $\mathcal{A}$'s critical path length because for each synchronous execution of $\mathcal{A}$, some process must take a step, and hence access a shared variable, in each round.

[9] Concurrent read/exclusive write parallel random access machines. Note that PRAM's are synchronous, and can perform an unlimited amount of local computation in a step.

Combining the two above steps we get that given a protocol $\mathcal{A}$ that achieves one-shot mutual exclusion among n processes with hot-spot contention c and critical path length ℓ, we can construct a CREW PRAM protocol 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 R be the set of indices i such that $x_i = 1$. At the end of each execution of the protocol in which no process crashes, there is a unique $i \in R$, such that p_i is a *winner* (i.e., p_i is in a special *win* state). If R is empty, then there is no winner.

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

LEMMA 5.2. *Let S be a CREW PRAM protocol for one-shot mutual exclusion on n inputs, running in time $s(n)$. We place no bound on the number of processes, but the mutual exclusion is among $p_1, \ldots, p_n$. Then there is a CREW PRAM protocol for logical OR on n inputs running in time $s(n) + 1$.*

PROOF. The protocol for OR is as follows. Let result be a special memory cell that is not used by protocol $\mathcal{S}$ on any input and is initialized to zero. On inputs $x_1, \ldots, x_n$ run $\mathcal{S}(x_1, \ldots, x_n)$. Let process i be the winner, if one exists. Then at step $s(n) + 1$ process i writes a "1" into memory location *result*. Note that the mutual exclusion property implies that there is at most one winner and hence the exclusive write requirement of CREW PRAM is not violated.

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 protocol correctly computes the OR. □

Next we show that given an asynchronous one-shot mutual exclusion protocol with hot-spot contention c and critical path length ℓ, we can construct a one-shot mutual exclusion protocol for the CREW PRAM that takes $O(c\ell)$ rounds.

LEMMA 5.3. *Let $\mathcal{A}$ be any protocol on n inputs running on an asynchronous shared-memory machine, with hot-spot contention at most c, with critical path length ℓ, requiring at most N processes, and requiring at most $m(n) \geq n$ shared variables. Then there exists a protocol for the synchronous CREW PRAM that requires at most $N + m(n)\binom{N}{c}$ processes and runs in time at most $O(c\ell)$.*

PROOF. We have observed that there is exactly one synchronous execution of $\mathcal{A}$ for each value of the inputs. Therefore, it is enough to construct a CREW PRAM protocol $\mathcal{S}$ that will simulate executions of $\mathcal{A}$ in a step-by-step fashion such that each execution of $\mathcal{S}$ with inputs I will have as its *corresponding* execution of $\mathcal{A}$ the synchronous execution of A with inputs I.

$\mathcal{S}$ is constructed as follows: It has a special set of *simulating* processes $P_1, \ldots, P_N$ whose job is principally to simulate, one for one, the processes of $\mathcal{A}$. For clarity, the processes of $\mathcal{S}$ will always be denoted by uppercase letters, while

those of $\mathcal{A}$ will be denoted by lowercase letters. The additional $m(n)\binom{N}{c}$ *auxiliary* processes are dedicated to resolving write conflicts at the $m(n)$ shared variables of A. Hence, the auxiliary processes are split into $m(n)$ *groups*, one for each of the $m(n)$ shared variables v of $\mathcal{A}$ and denote by G_v the group dedicated to location v.

We let $M[1 : m(n)]$ denote the first $m(n)$ locations of the PRAM's shared memory. After each round 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: **LOC**$[1 : N]$, **INDEX**$[1 : N]$, and **FLAG**$[1 : m(n)]$. Roughly speaking, when a simulating process 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 **LOC**$[i]$. The **INDEX** array is used to tell process P_i wishing to access $M[v]$ in a given simulating round, its index among the set of processes that will access $M[v]$ at this round. The **FLAG** array is used in determining, for each shared variable v and each round 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 round.

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 processes 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 rounds 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 rounds 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.

Round $s + 1$ is simulated as follows: First, P_i writes into **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 round, P_i writes a location into **LOC**$[i]$, then at the end of the simulation of this round P_i will set **LOC**$[i]$ back to zero. Thus, once the simulated p_i has terminated, P_i can terminate as well, and **LOC**$[i]$ will have the correct "location" (i.e., the null location) for all subsequent rounds of the simulation.

Let v be any shared variable. Since A has maximum hot-spot contention c, at most c processes have written v into the array **LOC**. Each of the $\binom{N}{c}$ processes P in G_v is assigned a set R of c cells of **LOC** to examine, to see which subset of the corresponding c processes of A would attempt to access v in round $s + 1$ of the corresponding synchronous execution of A. Let $\mathcal{T}$ denote the set of all size c subsets of indices into array **LOC**. Let us impose an ordering on the elements of $\mathcal{T}$ by first listing, for each $R \in \mathcal{T}$, the elements of R in increasing order, and then ordering each pair of lists lexicographically. Note that for every set R' of $1 \leq j \leq c$ indices of **LOC** there is a (lexicographically) smallest element R of $\mathcal{T}$ containing R'. Each process $P \in G_v$ proceeds as follows.

P has a private variable t_P, initially zero. Let $R = \{R_1, \ldots, R_c\}$ be the set of indices in **LOC** for which P is responsible. P reads **LOC**$[R_j]$, $1 \leq j \leq c$. Let R' be the maximal subset of R such that **LOC**$[R_i] = v$ for all $R_i \in R'$. If R' is nonempty, and R is the smallest element of $\mathcal{T}$ containing R', then P sets $t_P = |R'|$. This takes a total of c PRAM steps.

During the next c steps P executes the following loop and resets t_P to zero:

Do $j := c$ to 1;

If $t_P = j \wedge$ **FLAG**$[v] = 0$

then **FLAG**$[v] := 1$ fi od

$t_P := 0$

The loop requires c PRAM steps (recall that t_P is private). If P set **FLAG**$[v] := 1$ during the execution of the loop, then we say that P has found the write set for v and that P is the *leader* of G_v for the current round of the simulation.

If no write set is found there is nothing for the members of G_v to do until the simulation of the next round. Otherwise, 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 **INDEX**$[i]$. This takes c PRAM steps.

In the next step, for each v the leader of G_v sets **FLAG**$[v] := 0$, and for each i process P_i resets **LOC**$[i] := 0$. In the last c PRAM steps but one in the simulation of round $s + 1$, each P_i simulates p_i's access to v in order, according to **INDEX**$[i]$. Finally, in the last PRAM step of the simulation each P_i sets **INDEX**$[i] := 0$. □

THEOREM 5.4. *Let $\mathcal{A}$ be any protocol for one-shot mutual exclusion, and let c be its hot-spot contention and ℓ the length of its critical path. Then $\ell \in \Omega(\log n/c)$.*

PROOF. Combining Lemmas 5.2 and 5.3, we get that given a protocol $\mathcal{A}$ that achieves one-shot mutual exclusion among n processes with hot-spot contention c and critical path length ℓ we can construct a CREW PRAM protocol 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 [Cook et al. 1986]. □

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

THEOREM 5.5. *One-shot mutual exclusion among n processes has contention* $\Theta(1)$.

PROOF. It is enough to show that for each set of k processes where $1 \leq k \leq n$, there is an execution with contention cost at least $k - 1$ 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 contention cost $k - 1$ 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 contention cost incurred among processes $P_1, \ldots, P_k$ is exactly the same as in E, and hence the total contention cost increases by one, and we are done.

Since in a binary tournament tree the contention is $O(1)$, this bound is tight. □

6. *Discussion*

6.1. Summary of Results. This paper provides the first formal tools for analyzing contention in shared-memory algorithms. We believe that a realistic formal model for parallel computation requires taking contention into account. Similar considerations motivated the recent work done in Choy and Singh [1994], Culler et al. [1996], Gibbons et al. [1993], and Gibbons et al. [1994].

In particular, we introduce a formal complexity model for contention in shared-memory multiprocessors and use it to provide the first formal analysis of contention and trade-offs between contention and critical path length 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 hot-spot contention c, randomized consensus, which is nonblocking, requires a provably longer critical path 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 contention 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 [Herlihy et al. 1991] and the periodic counting network [Aspnes et al. 1991]. In addition, we show that our method can be used to analyze the contention of balancing networks in general, conditioned on them having certain smoothness properties.

6.2. Recent Work. For recent works pursuing the model and techniques introduced in this paper the interested reader is referred to Hardavellas et al. [1993] and Busch et al. [1994] (exploring the dependency of contention in counting networks on the width of the balancers being used, where the width of a balancer is the number of its input wires), Aiello et al. [1994] (a study of

contention in balancing networks and in specific constructions including constructions with randomized balancers, that is, balancers which flip coins), Busch and Mavronicolas [1994] (contention analysis of specific counting networks that have different number of input and output wires), and Ben-Dor et al. [1994] (construction of low contention dynamic counter). This list is not exhaustive.

6.3. Possible Extensions of the Model. This model, like all complexity models, represents an abstraction of real architectures. We have chosen not to model many detailed aspects of common multiprocessor architectures. The model could be extended to encompass many of these details, and doing this constitutes very interesting research problems. It remains to be seen whether the resulting increase in model complexity would provide a proportional increase in verisimilitude. Below we specify three of the detailed aspects of common machines not modeled by our model and outline possible extensions. Independent work by Gibbons et al. [1993; 1994] addresses some of these issues, including a memory model that includes pipelining.

For example, we assume that memory accesses are serialized at the granularity of individual locations, when, in practice, serialization occurs at coarser-grained memory modules. Clearly, our model can easily be extended to address this notion. The current simple version however has several virtues. First, it is independent of the particular architecture. Second, the contention measured by our model is a relevant measure for cache coherence overhead, since the number of cache invalidates or updates incurred by a write is often proportional to the number of processes concurrently accessing the location that is being written. Note also that any lower bound in our model is a lower bound also in architectures with memory modules.

Also, for architectures that permit processes to pipeline memory accesses, our contention cost is somewhat of an overkill. For example, one may want to charge a process that has several pending operations, by the maximal contention cost of any of these operations, instead of by their sum (as is currently being done). Formalizing such an extension depends on the architecture and is an interesting research problem.

In addition, we assume that the delay in accessing a variable is proportional to the number of concurrent accesses to the variable, when, in practice, the delay may be sublinear in the number of concurrent accesses (i.e., combining networks), or superlinear (i.e., inefficient spinning protocols). The model can be extended by defining the contention cost of an operation to be some function $f(m)$, where m is the number of responses events from the same variable that occur after the operation's invocation, up to but not including the matching response (in our current model $f(m) = m$). The function f will be sublinear for architectures with combining networks and may be superlinear for the inefficient spinning protocols. Making this point precise needs of course further research.

6.4. Other Open Questions. One interesting open question is whether all counting networks have low contention. Currently, we can suggest a methodology for analyzing other networks as well as general networks with certain smoothness properties, but one does not yet know how to reason about contention for counting networks in general (see Section 3.4). Indeed, the problem of wait-free counting is still only partially explored: note that other well-known data struc-

tures for counting, such as software combining trees [Goodman et al. 1989], are not wait-free, although they display low contention in practice.

This paper raises other questions as well. The contention or critical path/contention trade-offs of many well-known problems such as reader-writer synchronization, snapshots, and approximate agreement, are still unknown. (Note that problems that can be solved using single-writer multi-reader atomic registers, can be solved with very low hot-spot contention, with the price of an increase in the length of the critical path (see Section 4.2). For this type of problems the trade-offs between the hot-spot contention and critical path length are thus especially interesting.[10] It is also unknown whether the lower bounds on the trade-offs between contention and critical path length for randomized consensus and for mutual exclusion are tight.

Another interesting research direction is related to the overall cost (rather than to the contention cost). Loosely stated, in this paper, we compute the overall cost by simply summing up the contention cost plus the number of memory accesses. However, sometimes there are additional factors that affect the overall cost, such as locality of references. One may want to try to compute costs caused by these factors and incorporate them into the overall cost in a similar modular fashion to what is done here with contention cost, i.e. the overall cost will roughly be the sum of the costs caused by each of these factors plus the number of memory accesses.

On a completely different note, recent works [Ajtai et al. 1994; Aspnes and Waarts 1996] introduced a theory of competitiveness for distributed algorithms, meant to provide for distributed algorithms a somewhat more realistic analysis than the standard worst case analysis. Loosely stated, they suggest to compute the performance of a distributed candidate algorithm by comparing it to an optimal specified distributed algorithm that runs on the same schedule. Extending their model/analyses to encompass contention is a very interesting research project.

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. We also thank the anonymous referees for many helpful comments on the presentation of the paper.

REFERENCES

ABRAHAMSON, K. 1988. On achieving consensus using a shared memory. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing* (Toronto, Ont., Canada, Aug. 15–17). ACM, New York, pp. 291–302.

AHARONSON, E., AND ATTIYA, H. 1992. Counting networks with arbitrary fan-out. In *Proceedings of the 3rd ACM–SIAM Symposium on Discrete Algorithms* (Orlando, Fla., Jan. 27–29). ACM, New York, pp. 104–113.

AIELLO, W., VENKATESAN, R., AND YUNG, M. 1994. Coins, weights and contention in balancing networks. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing* (Los Angeles, Calif., Aug. 14–17). ACM, New York, pp. 193–205.

[10] The lower bounds on trade-offs presented in this paper hold even if the protocols use read-modify-write operations.

AJTAI, M., J. A., DWORK, C., AND WAARTS, O. 1994. A theory of competitive analysis for distributed algorithms. In *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science*. IEEE, New York, pp. 401–411.

ANDERSON, T. 1990. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Paral. Dist. Syst. 1*, 1 (Jan.), 6–16.

ASPNES, J. 1990. Time- and space-efficient randomized consensus. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing* (Quebec City, Que. Canada, Aug. 22–24). ACM, New York, pp. 325–332.

ASPNES, J., AND HERLIHY, M. 1990. Fast randomized consensus using shared memory. *J. Algorithms 11*, 3 (Sept.), 441–460.

ASPNES, J., HERLIHY, M., AND SHAVIT, N. 1991. Counting networks and multi-processor coordination. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing* (New Orleans, La., May 6–8). ACM, New York, pp. 348–358.

ASPNES, J., AND WAARTS, O. 1992. Randomized consensus in expected $o(n \log^2 n)$ operations per processor. In *Proceedings of the IEEE Symposium on Foundations of Computer Science.* IEEE, New York.

ASPNES, J., AND WAARTS, O. 1996. Modular competitiveness for distributed algorithms. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing* (Philadelphia, Pa., May 22–24). ACM, New York, pp. 237–246.

BATCHER, K. 1968. Sorting networks and their applications. In *Proceedings of AFIPS Joint Computer Conference* AFIPS Press, pp. 338–334.

BEN-DOR, A., ISRAELI, A., AND SHIRAZI, A. 1994. Dynamic counting. In *Proceedings of the 3rd Israel Symposium on the Theory of Computing and Systems* (June 1994).

BEN-OR, M. 1983. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Que., Canada, Aug. 17–19). ACM, New York, pp. 27–30.

BUSCH, C., HARDAVELLAS, N., AND MAVRONICOLAS, M. 1994. Contention in counting networks. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing* (Los Angeles, Calif., Aug. 14–17). ACM, New York, p. 404.

BUSCH, C., AND MAVRONICOLAS, M. 1994. A depth-contention optimal counting network. Manuscript.

CHOR, B., ISRAELI, A., AND LI, M. 1987. On processor coordination using asynchronous hardware. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, pp. 86–97.

CHOY, M., AND SINGH, A. 1994. Adaptive solutions to the mutual exclusion problem. *Dist. Comput. 8*, 1, 1–17.

COOK, S., DWORK, C., AND REISCHUK, R. 1986. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput. 15*, 1 (Feb.), 87–97.

CORMEN, T., LEISERSON, C., AND RIVEST, R. L. 1990. *Introduction to Algorithms.* MIT Press, Cambridge Mass.

CULLER, D., KARP, R., PATTERSON, D., SAHAY, A., SANTOS, E., SCHAUSER, K., SUBRAMONIAN, R., AND VON EICKEN, T. 1996. Logp: A practical model of parallel computation. *Commun. ACM 39*, 11, 78–85.

DIGITAL EQUIPMENT CORPORATION. 1992. *The Alpha Architecture Handbook*. Digital Press, Maynard, Mass.

DOLEV, D., DWORK, C., AND STOCKMEYER, L. 1987. On the minimal synchronism needed for distributed consensus. *J. ACM 34*, 1 (Jan.), 77–97.

DOWD, M., PERL, Y., RUDOLPH, L., AND SAKS, M. 1989. The periodic balanced sorting network. *ACM Trans. Prog. Lang. Syst. 36*, 4 (Oct.), 738–757.

DWORK, C., HERLIHY, M., PLOTKIN, S., AND WAARTS, O. 1992. Time-lapse snapshots. In *Proceedings of the 1st Israel Symposium on the Theory of Computing and Systems*.

DWORK, C., LYNCH, N., AND STOCKMEYER, L. 1988. Consensus in the presence of partial synchrony. *J. ACM 35*, 2 (April), 228–323.

DWORK, C., AND WAARTS, O. 1993. Randomized snapshot in linear time. Unpublished manuscript.

ELLIS, C., AND OLSON, T. 1988. Algorithms for parallel memory allocation. *J. Paral. Prog. 17*, 4 (Aug.), 303–345.

FISCHER, M., LYNCH, N., AND PATERSON, M. 1985. Impossibility of distributed commit with one faulty process. *J. ACM 32*, 2 (Apr.), 74–82.

GIBBONS, P., MATIAS, Y., AND RAMACHANDRAN, V. 1993. The queue-read queue-write PRAM model: accounting for contention in parallel algorithms. *SIAM J. Comput*., to appear.

GIBBONS, P., MATIAS, Y., AND RAMACHANDRAN, V. 1994. Efficient low-contention parallel algorithms. *J. Comput. Syst. Sci*. to appear.

GOODMAN, J., VERNON, M., AND WOEST, P. 1989. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Mass., Apr. 3–6). ACM, New York, pp. 64–75.

GOTTLIEB, A., GRISHMAN, R., KRUSKAL, C., MCAULIFFE, K., RUDOLPH, L., AND SNIR, M. 1984. The NYU ultracomputer–designing an MIMD parallel computer. *IEEE Trans. Comput. C-32*, 2 (Feb.), 175–189.

GOTTLIEB, A., LUBACHEVSKY, B., AND RUDOLPH, L. 1983. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Prog. Lang. Syst. 5*, 2 (Apr.), 164–189.

GRAUNKE, G., AND THAKKAR, S. 1990. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer 23*, 6 (June), 60–70.

HARDAVELLAS, N., KARAKOS, D., AND MAVRONICOLAS, M. 1993. Notes on sorting and counting networks. In *Proceedings of the 7th International Workshop on Distributed Algorithms* (*WDAG*). In Lecture Notes in Computer Science, Vol. 725. Springer-Verlag, New York, pp. 234–248.

HERLIHY, M. 1991. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst. 13*, 1 (Jan.), 123–149.

HERLIHY, M., LIM, B.-H., AND SHAVIT, N. 1992. Low contention load balancing on large-scale multiprocessors. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures* (July 1992).

HERLIHY, M., SHAVIT, N., AND WAARTS, O. 1991. Low contention linearizable counting. In *Proceedings of the IEEE Symposium on Foundations of Computer Science* IEEE, New York.

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

KLUGERMAN, M., AND PLAXTON, C. 1992. Small-depth counting networks. In *Proceedings of the 1992 ACM Symposium on Theory of Computing* (Victoria, B.C., Canada, May 4–6). ACM, New York, pp. 417–427.

LI, M., TROMP, J., AND VITANYI, P. 1989. How to share concurrent wait-free variables. Tech. Rep. CS-R8916. CWI, Amsterdam.

LOUI, M., AND ABU-AMARA, H. 1987. *Advances in Computing Research*, Volume 4, *Chapter Memory Requirements for Agreement Among Unreliable Asynchronous Processes*. JAI Press, pp. 163–183.

MELLOR-CRUMMEY, J., AND SCOTT, M. 1990. Algorithms for scalable synchronization on shared-memory multiprocessors. Tech. Rep. 342 (April), University of Rochester, Rochester, N.Y.

METCALFE, R., AND BOGGS, D. 1976. Ethernet: distributed packet switching for local computer networks. *Commun. ACM 19*, 7 (July), 395–404.

MIPS COMPUTER COMPANY. The MIPS RISC architecture.

PFISTER, G., AND NORTON, A. 1985. 'hot spot' contention and combining in multistage interconnection networks. *IEEE Transactions on Computers C-34*, 11 (Nov.), 933–938.

RABIN, M. 1983. Randomized byzantine generals. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science* (May). IEEE, New York. pp. 403–409.

RUDOLPH, L. 1983. Decentralized cache scheme for an MIMD parallel processor. In *Proceedings of the 11th Annual Computing Architecture Conference*. pp. 340–347.

SINGH, A., ANDERSON, J., AND GOUDA, M. 1987. The elusive atomic register revisited. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, pp. 206–221.

YANG, J., AND ANDERSON, J. 1993. Fast, scalable synchronization with minimal hardware support. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing* (Ithaca, N.Y., Aug. 15–18). ACM, New York, pp. 171–182.

RECEIVED AUGUST 1993; REVISED OCTOBER 1996; ACCEPTED OCTOBER 1997