Efficient Low-Contention Asynchronous Consensus with the Value-Oblivious Adversary Scheduler*

Yonatan Aumann

Dept of Mathematics and Computer Science
Bar Ilan University
Ramat-Gan
Israel

Email: aumann@cs.biu.ac.il

Michael A. Bender†

Dept of Computer Science
SUNY Stony Brook
Stony Brook, NY 11794-4400
USA

Email: bender@cs.sunysb.edu
URL: http://www.cs.sunysb.edu/~bender/
Phone: +1 631 632 7835
Fax: +1 631 632 8334

† Contact author.

*An early version of this paper was presented in the 23rd International Colloquium on Automata, Languages, and Programming (ICALP '96) [8].

Efficient Low-Contention Asynchronous Consensus with the Value-Oblivious Adversary Scheduler[†]

Yonatan Aumann[‡] Michael A. Bender[§]

November 27, 2003

Abstract

We consider asynchronous consensus in the shared memory setting. We present the first efficient low-contention consensus algorithm for the weak adversary scheduler model. The algorithm achieves consensus in $O(n \log^2 n)$ *total work* and $O(\log n)$ (hot-spot) contention, both expected and with high probability. The algorithm assumes the *value-oblivious scheduler*, which is defined in the paper. Previous efficient consensus algorithms for weak adversaries suffer from $\Omega(n)$ memory contention.

1 Introduction

1.1 Overview

The Problem. Asynchronous consensus is a central problem in distributed computing. The consensus problem on n processors is defined as follows. Initially, each processor P_i has a private *input value* v_{P_i}. The goal is to agree on a common *consensus value* v, which must be one of the input values. If all the input values are either 0 or 1, then the problem is called *binary consensus*. In this work we consider the asynchronous binary-consensus problem in the shared-memory setting.

The asynchronous consensus problem can be viewed as a game between the processors and an *adversary scheduler*. The processors seek to attain consensus, while the adversary scheduler attempts to prevent consensus by controlling the processors' speeds. Technically, at each instant in time, the adversary determines the next processor to operate.

Asynchronous consensus is impossible deterministically [15, 16, 19, 21] (see Section 1.3). Hence, we seek randomized algorithms. The best known algorithms for the standard (strong) adversary scheduler require expected $\tilde{O}(n^2)$ total work* [13]. Recently, Aspnes [2] proved that this is the best possible. Specifically, [2] proves that any shared-memory protocol for asynchronous consensus

†An early version of this paper was presented in the 23rd International Colloquium on Automata, Languages, and Programming (ICALP '96) [8].

‡Department of Mathematics and Computer Science, Bar-Ilan University, Ramat-Gan, Israel. Email: aumann@cs.biu.ac.il. This work was partially completed while the author was at Harvard University, supported in part by ONR contract ONR-N00014-91-J-1981.

§Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, USA. Email: bender@cs.sunysb.edu. Supported in part by HRL Laboratories, NSF Grants EIA-0112849 and CCR–0208670, and Sandia National Laboratories. This work was partially completed while the author was at Harvard University, supported by NSF grants CCR-97-00365, CCR-95-04436, and CCR-93-13775.

*The notation $g(n) = \tilde{O}(f(n))$ means that $g(n) = O(f(n)\log^c n)$ for some constant c. I.e., g is $O(f)$ up to poly-logarithmic factors. Similarly, $g(n) = \tilde{\Omega}(f(n))$ means that $g(n) = \Omega(f(n)\log^c n)$ for some constant c.

with the standard adversary model requires $\Omega(n^2/\log^2 n)$ total work. Thus, in order to attain sub-$\tilde{O}(n^2)$ work, weaker adversary models must be used. (In general, a weaker adversary model is obtained by imposing restrictions on the information available to the adversary for making its adversarial decisions. The exact definitions are provided later.) Chandra [14] and Aumann [7] provide algorithms that obtain consensus in expected $\tilde{O}(n)$ total work, assuming a weak adversary scheduler ($O(n\log^2 n)$ in [14] and $O(n\log n)$ in [7]). However, both algorithms suffer from $\Theta(n)$ contention on shared-memory registers. In fact, $\Theta(n)$ is not only the worst-case contention, but also the expected contention if the processors operate in a more-or-less synchronized fashion (see Section 1.3). Note that if processors must queue (and busy-wait) for accessing shared-memory variables, then $\Theta(n)$ contention translates to a $\Theta(n)$ additional factor in execution cycles, thus bringing the total number of cycles back to $\tilde{O}(n^2)$.

Our Results. In this paper we present an efficient low-contention consensus algorithm for a weak-adversary-scheduler model. We present a randomized algorithm that obtains consensus in $O(n\log^2 n)$ total work, $O(\log n)$ hot-spot contention, and $O(n\log^3 n)$ contention cost. *Total work* is defined as the total number of instruction steps performed by *all* processors collectively, regardless of the distribution of the instructions among the processors and regardless of the processors' relative speeds and interleaving activity. *Hot-spot* contention is the maximal number of simultaneous accesses to any shared-memory variable. *Contention cost* is an upper bound on the total number of busy-waiting cycles that processors incur throughout the execution. Formal definitions of the performance measures are provided in Section 1.2. The performance bounds are obtained assuming the *Value-Oblivious adversary scheduler*, also defined in Section 1.2. Our algorithm is error free (Las Vegas) and the performance is guaranteed both with high probability and in expectation.

We say that an event E occurs *with high probability* if for any any constant $c > 0$ there exists a proper choice of constants such that $\Pr[E] \geq 1 - n^{-c}$.

Outline. In the remainder of this section we present the formal model and discuss related work. The description of the consensus protocol is divided into four parts. First, in Section 2 we explain how the processors collectively toss $O(\log n)$ *weak random coins*. Next, in Section 3 we show how to use the weak random coins to obtain consensus with high probability. The algorithm thus obtained has a polynomially small failure probability. In Section 4 we provide a construction that allows us to transform the consensus algorithm into one that never fails, thus obtaining a Las Vegas guarantee. In Section 5 we show how to reduce the contention from $O(\log^2 n)$ to $O(\log n)$. Finally, we integrate the results in Section 6 and present open problems.

1.2 The Model

Operations. We assume a system containing n asynchronous processors. The processors communicate via shared memory with *multi-reader/multi-writer* registers. The processors have a set of *atomic operations* that they may execute. The atomic operations include:

1. reading a word from shared memory,
2. writing a word to shared memory, and
3. executing one of a fixed set of *basic computations* (e.g., add, multiply, coin toss, etc.) in local registers.

We emphasize that no atomic operation can both read from and write to shared memory. Thus, no compound operation such as *test & set* or *compare & swap* is atomic.

Asynchrony. The system operates in a sequence of discrete steps. At each step, exactly one processor performs one atomic operation. The ordering of the operations among the processors is arbitrary and is determined by an adversary scheduler. Prior to each step, there is an *enabled operation* associated with each processor that has not yet completed the procedure. The enabled operation is the operation that the processor performs whenever it is granted the right to operate. There are no fairness constraints imposed on the schedule. In particular, all but one processor may fail.

The Value-Oblivious Adversary Scheduler. At each step, the adversary scheduler determines the next processor to operate. In defining the adversary scheduler, we must specify what information is available to it for making this decision. Here, we introduce the *Value-Oblivious adversary scheduler*, which, we believe, faithfully captures asynchronous behavior of real-world systems. Our algorithms assume the Value-Oblivious adversary scheduler.

The Value-Oblivious scheduler is assumed to know the entire state of the system prior to the beginning of the execution of the consensus procedure, including all values stored in all memory cells, all of the processors' input values, and the code of the protocol itself. The definition of the Value-Oblivious scheduler distinguishes between the *functional description* of an operation and the *values description* of an operation. The *functional description* of an operation consists of

1. a description of the type of the operation, e.g., read, write, add, multiply, etc., and

2. a description of the addresses and locations of the operands of the operation.

The *values description* of an operation provides the actual *values* manipulated by the operation. For example, suppose that register R stores the value 'abc', and consider the operation $A[100] \leftarrow R$. In this case, the functional description of the operation says that the operation is a copy operation, copying the value stored in register R to the 100-th location of the array A. The values description of the operation says that the value being copied is 'abc'.

The Value-Oblivious scheduler is provided with the functional description of the operations, but *not* the values description. Specifically, the Value-Oblivious scheduler has knowledge of:

1. the functional description of all operations performed so far, and

2. the functional description of all enabled operations.

The Value-Oblivious adversary is *not* provided with the values description of the past or enabled operations. In particular, the adversary is not (directly) provided with the outcomes of the random coin tosses. The scheduler can gain knowledge of the outcome of a coin toss only insofar as the outcome affects the functional description of the computation (e.g., by affecting a branch decision or the location of an operation's operand).

We claim that the power given to the Value-Oblivious scheduler captures the common sources of asynchronous behavior of real-world systems. That is, the real-world schedule may depend on the functional description of operations, but it does not depend on the values description. To illustrate, common causes of asynchrony such as hardware failures, clock skews, operating system interrupts, cache misses, page faults, and network delays all depend solely on the operations performed, not on the actual values manipulated by the operations. This is due to the fact that on most modern processors common integer ALU operations run in the same number of cycles regardless of the values of the operands.[†] Similarly, the time to run a load or store operation may depend on the

[†]This may not apply to the division operation and to some floating-point operations. Consequently, our algorithms should not use these operations.

location in memory but rarely on the actual data. Accordingly, the scheduler, which is designed to model real-world scheduling dynamics, is provided with the functional description of operations but not the values description.

We note that there may be cases in which this model breaks down, but we believe that such cases are rare. Chandra [14] gives a hypothetical example of hardware where the cache optimization mechanism skips writes that do not change the register value. In this case, the values description may affect the dynamics of the execution, and hence the schedule.

Contention. In defining contention we follow [17]. In [17] each memory-access operation is assumed to consist of two stages: *initiation* (by the processor) and *response* (by the system). Between initiation and response the operation is *pending*. Each initiation event must ultimately have a single matching response event, but the ordering of responses is arbitrary and is determined by the system. In our model, initiation can be correlated with the point in time when the operation is enabled, and the response with the point where it is executed.

For an operation *op* on a shared-memory variable v, the *contention cost* of *op* is the number of response events from v that occur in between the initiation of *op* and its response. Intuitively, the contention cost counts the number of busy-waiting cycles of the operation. The *contention cost* of an execution is the sum of contention costs of all operations in the execution.

The *hot spot contention* of an execution is defined as the maximum, taken over all shared-memory variables v, of the maximum number of simultaneous operations pending on v.

Work. Performance is measured using the *total work* measure. Total work counts the *total number of operations* executed throughout the duration of the protocol, summed up over all processors. Total work does not account for busy waiting due to contention; this quantity is measured by the contention cost.

An asynchronous protocol is said to be *wait free* if any processor can complete the protocol after a finite number of its own steps, regardless of the operations of the other processors [20]. Specifically, a protocol A is *wait free* if the following holds: There exists a function $f_A(n)$ such that for any number of processors n and any processor P that is participating in an n-processor implementation of protocol A, processor P completes its part in the protocol within $f_A(n)$ steps, regardless of the operations of the other processors; the protocol is *expected wait free* if the processor completes within *expected* $f_A(n)$ steps.

The consensus protocol we present here is expected wait free.

1.3 Related Work

Fischer, Lynch, and Paterson [19] proved the impossibility of deterministic asynchronous consensus in a message-passing environment even if only one processor may fail. Loui and Abu-Amara [21], Chor, Israeli, and Li [15], and Dolev, Dwork, and Stockmeyer [16] proved analogous results for the shared-memory setting. The first randomized solution for the shared-memory, wait-free setting was presented by Chor, Israeli, and Li [15]; this algorithm runs in expected $O(n^2)$ total work and assumes a weak adversary scheduler. The first solution for the strong (standard) adversary model was presented by Abrahamson [1], but the work bounds are exponential. Aspnes and Herlihy [4] gave the first polynomial-work algorithm for the strong-adversary model. A series of papers followed, ultimately reducing the complexity to $O(n^2 \log n)$ *total work* [13] and $O(n \log^2 n)$ steps per processor [5]. The reader is referred to [3] for a comprehensive survey of consensus algorithms. Aspnes [2] showed that these last results are the best possible up to polylogarithmic factors by proving that any protocol for the strong-adversary model requires $\Omega(n^2/\log^2 n)$ total work. Thus, in order to obtain sub-$\tilde{O}(n^2)$ performance, weaker adversary models must be used.

We now focus our attention on weak adversary models. Several different weak adversary models exist in the literature. We denote these: the *Oblivious scheduler*, the *Value-Oblivious scheduler*, the *Write-Oblivious scheduler*, and the *Local-Oblivious scheduler*. Appendix A provides a review of the definitions of these schedulers, and the relationships between them. The Oblivious scheduler is the weakest, requiring the adversary to determine the entire schedule in advance. The other three schedulers allow for dynamic choices of the adversary, but differ in the information made available to the adversary for making these choices. Of these three models, the Value-Oblivious scheduler is weakest, and the other two are incomparable.

We now review the results for the weak models.

Aumann, Bender, and Zhang [9] consider the consensus problem in the Oblivious-adversary setting. Their protocol obtains simultaneous agreement on n *words* in $O(n \log n \log\log n)$ total work.

Chor, Israeli, and Li [15] provide an expected $O(n^2)$ total work protocol for the Local-Oblivious scheduler. Abrahamson [1] presents a protocol with expected $O(n^2 \log n)$ steps per processor, also for the Local-Oblivious scheduler.

Chandra [14] provides a protocol that obtains consensus in expected $O(\log^2 n)$ steps per processor, assuming the Write Oblivious scheduler. The consensus may be on inputs of any size, not only binary inputs. Following the first execution of the protocol, subsequent executions can be completed in $O(\log n)$ steps per processor; subsequent executions of binary consensus complete in $O(1)$ steps per processor. Aumann [7] achieves consensus in expected $O(\log n)$ steps per processor, also assuming the Write Oblivious scheduler.[‡] Here too, the consensus may be on inputs of any size, not only binary.

The results of Chandra [14] and Aumann [7] translate to $O(n \log^2 n)$ and $O(n \log n)$ total work, respectively. In addition to total-work bounds, both Chandra and Aumann give upper bounds on the number of steps performed by any individual processor. In this respect, the bounds of Chandra and of Aumann are stronger than those in this paper. However, as mentioned in the introduction, the expected hot-spot contention of both these protocols is $\Omega(n)$, and the expected contention cost is $\Omega(n^2)$. The reason is that in both algorithms in each round there are only a constant number of registers into which all processes write. Thus, for example, if in the first round all processors are synchronous, then there must be at least one register that has a $\Omega(n)$ hot-spot contention and hence $\Omega(n^2)$ contention cost.

Aumann and Kapach-Levy [10] consider the problem of asynchronous consensus using only single-writer/single-reader variables. They provide a binary-consensus algorithm that completes in $O(n \log n \, e^{\sqrt{\log n}})$ total work, assuming the Value-Oblivious scheduler.

It is worthwhile to compare the techniques used in our paper to those used in other papers on asynchronous consensus, and in particular to those of Chandra [14] and Aumann [7]. Most of the other papers are based, following the protocol of Ben-Or [12], on a strategy where processors operate in a sequence of *rounds*. In each round each processor *proposes* a value and reads the proposals of all other processors. If all of the proposals in the round (or the last *two* rounds for some algorithms) agree, then consensus is obtained. If there is no agreement in the round, then the computation advances to the next round in one way or another. In the protocols of Chandra [14] and of Aumann [7], if there is no agreement in the round then the processors toss a shared weak coin to choose their next proposal.

[‡]The protocol as published in [7] contains a bug. However, the bug is easily fixable, and the result still holds. The bug-free protocol shall appear in the full version of the paper (in preparation).

This paper, in contrast, uses a different approach. Initially, processors repeatedly write their input value to random locations in a shared-memory array of size $\Theta(n)$. The array then holds a mix of zeros and ones. The algorithm then gradually modifies the array so that one principal value is stored throughout the array. This convergence is achieved by repeatedly sampling the array and writing back a new value based on the values present in the sample. A two-threshold policy guarantees convergence within $O(\log n)$ phases, with high probability (see [23] for an early use of a two-threshold technique). Following this convergence stage, an additional stage is executed to *guarantee* that convergence has indeed been obtained (the Estimate Counting and Exit-Stage procedures). If not, an alternate consensus procedure is executed, thus ensuring that consensus is obtained unconditionally.

Dwork, Herlihy, and Waarts [17] study contention issues arising in the context of distributed shared-memory protocols. Dwork et al. prove that for any wait-free randomized consensus algorithm with hot-spot contention bounded by c there must be a processor for which the (worst-case) expected number of steps is $\frac{n-1}{c}$. The result is stated for the strong adversary scheduler, but the proof also holds for the Value-Oblivious adversary scheduler. This result means that $\tilde{O}(1)$ hot-spot contention implies the existence of a processor with expected $\tilde{\Omega}(n)$ steps. Thus, any protocol that has low contention, such as the one in the current paper, cannot guarantee low per-processor bounds, which explains our focus on total work as the performance metric.

Saks, Shavit, and Woll [24] obtain a solution that is both wait free in the general case and fast in the near-synchronous case. Using the *interleaving-algorithms* method ([6]) they provide a protocol that with f faulty processors terminates in $O(\log n)$ rounds if $f < \sqrt{n}$ and in $O(\frac{n^3}{n-f})$ rounds, otherwise. The length of the *round* is determined by the speed of the slowest nonfaulty processor.

2 Weak Random Coins

A weak random coin is not necessarily fair, but nonetheless

1. there is at least a constant probability that the coin falls on heads, i.e., gets the value 1, and

2. there is at least a constant probability that the coin fall on tails, i.e., gets the value 0.

There may also be some probability that the weak random coin has an 'undefined' value or that the coin exhibits different outcomes to different viewers. The notion of a *weak random coin* has been used in other agreement protocols (e.g. [4, 18, 13]).

In this section we develop an algorithm for the collective of processors to toss $\Theta(\log n)$ weak random coins. Specifically, we give a procedure by which the processors collectively toss $\beta \log n = \Theta(\log n)$ coins, *half* of which are guaranteed (with high probability) to be weak random coins. The adversary may determine which coins are random and which are not, but the majority are weak random coins.

2.1 The Protocol

Let $d = \lceil \log(n/\log n) \rceil$ and $N = 2^d = O(n/\log n)$ (all logs are to the base 2). The coin-toss procedure employs the following data structure. Associated with each coin r_i, $i = 1, \ldots, \beta \log n$, is a fixed $N \times (d+1)$ array B_i. We view each array B_i as a *butterfly structure* with $d+1$ levels as follows. For $k = 0, \ldots, d$, cells $B_i[k][\cdot]$ constitute the *k-th level* of the butterfly. Each cell $B_i[k][j]$ of level k is said to have two *predecessor* cells in the previous level: $B[k-1][j]$ and $B[k-1][j|_k]$

(here $j|_k$ is the number whose binary representation is identical to that of j except for the k-th bit, which is flipped). We call the cells in level 0 the *inputs* and the cells in level d the *outputs*.

All cells are initialized to 0. The pseudocode of the weak-coin protocol appears in Figure 1. The protocol is performed in *rounds*, which the processors execute repeatedly until the algorithm terminates. A round is a single execution of lines 1-13.

Throughout the protocol, each processor counts the number of operations that it performs. If any processor exceeds n^2 operations, then it exits the entire consensus protocol and enters the Alternate protocol, as described in Figure 10. The use of Alternate ensures that high-probability guarantees can be converted into bounds on the expected work. (The choice of n^2 as the bound is arbitrary. Any polynomial bound that is $\omega(n \log n)$ would suffice.)

WEAK-RANDOM-COINS (for processor P_i)

Repeat

1 **if** total number of operation of processor exceeds n^2 **then**
  run Alternate (Figure 10)
2 $k \leftarrow$ `Read-Clock`
3 **if** $k =$ "undefined" **then goto** line 13
4 **else if** $k > d$ **then exit** // algorithm completed
5 **else if** $k = 0$ **then** // phase 0, fill inputs
6   choose $i \in [1, \ldots, \beta \log n], j \in [1, \ldots, N]$ at random
    // random butterfly B_i and cell $B_i[0][j]$
7   choose $r \in \{0, 1\}$ at random with Pr [r=1] = $1/N$
8   $B_i[0][j] \leftarrow r$
9 **else** // phase $k > 0$
10   **for** $t = 1$ **to** $\log n$ **do**
11   choose $i \in [1, \ldots, \beta \log n], j \in [1, \ldots, N]$ at random
      // random butterfly B_i and cell $B_i[k][j]$
12     $B_i[k][j] \leftarrow B_i[k-1][j] \vee B_i[k-1][j|_{k-1}]$ // logical OR of predecessors
13 `Clock-Update`

Figure 1: **Weak-Random-Coins Procedure.**

The algorithm is composed of $d + 1$ *phases*. For each k, $0 \leq k \leq d$, phase k is devoted to *filling* the k-th level of the butterflies. Thus, in phase 0 the processors write to, or *fill*, input cells of all the butterflies. To do so, each processor repeatedly picks a random input cell in a random butterfly and writes a 0 or a 1 to it, where

$$\begin{aligned} \Pr[\text{writing } 1] &= 1/N \\ \Pr[\text{writing } 0] &= 1 - 1/N. \end{aligned}$$

Phase 0 completes after there have been $O(N \log N) = O(n)$ fillings of input cells.

Phases k, for $k > 0$, are longer; each subsequent phase consists of $O(N \log^2 N) = O(n \log n)$ fillings. To decide which cell to fill during phase k, (as before) a processor picks a random butterfly and a random cell in level k of the butterfly and writes a 0 or a 1 to it. The goal of the procedure is now to propagate 1's from the butterfly's input cells to its output cells. Accordingly, when a processor fills a cell, it writes the logical OR of the values appearing in the two predecessor cells (i.e., two 0's propagate a 0, and a 1 at any of the predecessors propagates a 1).

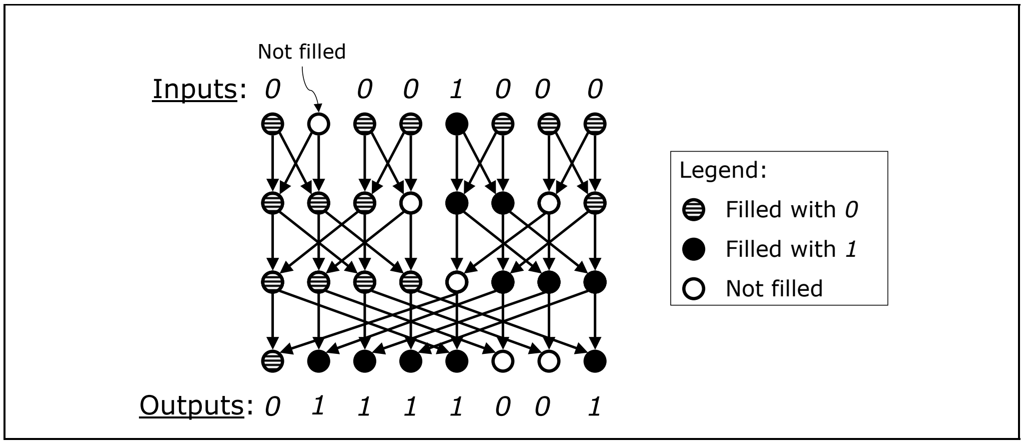

Figure 2: One of the butterflies from the weak coins procedure. There is a constant probability of having a 1 in the input level and a constant probability of having no 1's in the input level. When subsequent levels are filled the 1's propagate. If the adversary knew which inputs were 1's, it could force all outputs to be 0's by stopping processors propagating 1's. Luckily for the algorithm, the adversary is value oblivious.

Each butterfly corresponds to a coin toss. A butterfly with a sufficiently large fraction of 1's in the output corresponds to a 1 coin flip, and a butterfly with a sufficiently large fraction of 0's in the output corresponds to the 0 coin flip. If there are neither enough 1's nor 0's, then the coin is undefined. We show that most of the butterflies have at least a constant fraction of their inputs filled, and for each of these butterflies there is a a constant probability that some input is a 1 and a constant probability that all of the inputs are 0's. The objective is to propagate any 1 input to most or all of the outputs of the butterfly. If all inputs are 0's, then 0's will be propagated to the outputs.

The adversary can try to hinder the algorithm by stopping some 1's from being propagated. If the adversary knew which input cells contained 1's, the adversary could effectively stop all 1's from being propagated to the output cells of a butterfly, thus forcing all outputs to be 0 and causing all coin tosses to be 0. Luckily, the Value-Oblivious adversary cannot be so destructive because it does not know which input cells contain the 1's. We show that for at least half of the butterflies, there is a constant probability that at least 3/4 of the outputs are 1's, and a constant probability that at least 3/4 of the outputs are 0's.

An important issue is how the processors know when to advance from one phase to the next. That is, how do the processors know when enough cells are filled in the current phase, so that the computation may advance to the next? For this task we employ the Phase Clock construction of [11], which we describe below.

2.2 The Phase Clock

Most of the procedures described in this paper operate in a sequence of *phases*. Within each phase the processors perform a specific set of operations associated with the phase. Following a fixed amount of work (e.g., $\Theta(n)$ or $\Theta(n \log n)$) the phase completes, and the next phase begins. Thus, it is necessary to provide a tool that measures the amount of work performed in the phase and indicates when the phase is completed. Aumann and Rabin [11] present a general *Phase Clock* structure, which provides the necessary features. We briefly review the main characteristics of this

Phase Clock. For details the reader is referred to [11].

Properties. The Phase Clock supports two procedures: `Read-Clock` and `Update-Clock`, each of which takes $O(\log n)$ operations. The procedure `Read-Clock` returns a non-negative integer or "undefined". The Phase-Clock procedures guarantee that for any non-negative value, i (up to any exponential bound), there exist time intervals, Strong(i) and Weak(i), such that with high probability:[§]

- Strong(i) $\subseteq$ Weak(i).
- Throughout interval Strong(i) `Read-Clock` necessarily returns the value i.
- Only during interval Weak(i) can `Read-Clock` return the value i.
- Weak(i) ends before Weak($i+1$) starts.
- Strong(i) consists of $\Theta(n)$ invocations of `Update-Clock`. Furthermore, for any α, the clock can be configured such that Strong(i) consists of at least αn invocations of `Update-Clock`.
- $O(n)$ invocations of `Update-Clock` are sufficient to advance from the beginning of Weak(i) to the beginning Weak($i+1$).

Thus, the clock advances from one integral value to the next, staying at each value for $\Theta(n)$ invocations of `Update-Clock`.

Using the Clock. The clock, as described so far, operates in isolation. By interleaving clock updates with other work, as in Figure 1, we obtain a measure of the total work performed in the system.

In our protocol we use a separate Phase Clock for each procedure.

2.3 Obtaining the Outcome of the Coin Flip

We will show that when the Weak-Random-Coin procedure completes, with high probability for at least half of the butterfly arrays B_i, 3/4 of the cells of the output layer share a common value $r \in \{0,1\}$. Furthermore, there is a constant probability that r is 1 and a constant probability that r is 0. Thus, if we were to sample the output layer of these butterflies, we would get a weak-random-coin behavior. However, directly sampling the output layers of the butterflies, would result in high contention. Hence, in order to reduce contention, we "transfer" the values from the output layer of each butterfly B_i, which is of size $\Theta(n/\log n)$, into an array R_i of size $\Theta(n)$. This is achieved by the processors, collectively, reading $\Theta(n)$ random values from the output cells of B_i, and writing them to random locations of R_i. The Transfer procedure is provided in Figure 3

In order to obtain the value of the i-th coin flip, denoted by r_i, a processor samples $\Theta(\log n)$ locations of R_i and takes the majority value.

[§]In fact, the error probability is exponentially low.

```
TRANSFER

Repeat
1   if total number of operation of processor exceeds n^2 then
2       run Alternate (Figure 10)
3   k ← Read-Clock
4   if k = "undefined" then goto line 8
5   else if k > β log n then exit                      // algorithm completed
6   else
7       read random output cell of butterfly B_k
8       write the value to random location of R_k
9   Clock-Update
```

Figure 3: **Transfer Procedure:** Transferring the values from the output layer of the butterflies to the arrays R_i.

2.4 Analysis

2.4.1 Correctness

We denote by *phase* k, $k = 0, \ldots, d$, the time interval when the clock value is k (i.e., Strong(k)).

We view the process of randomly filling cells as a balls-and-bins game. There is a subtle difference, however, because the adversary knows the *next* operation of each processor and can thus decide to stop any given operation. Fortunately, the adversary cannot stop too many cells from being filled.

Lemma 1 *Consider the process where $c_1 n$ balls are thrown to $c_2 n$ bins. Suppose that an adversary can stop at most n of the balls in mid-air. Then, for any c_2 and $\epsilon < 1$, there exists a c_1 such that, for all n, with high probability, at most $\epsilon c_2 n$ bins do not hold a ball at the end of the process.*

Similarly, if $c_1 n \log n$ balls are thrown, then at most $\epsilon c_2 n / \log n$ bins do not hold a ball.

Proof: First consider the situation without the adversary. For a given bin the expected number of balls aimed at the bin is c_1/c_2. Say that a bin is *weakly aimed*, if fewer than $\frac{c_1}{2c_2}$ balls are aimed at it, and *heavily aimed* otherwise. By the Chernoff bounds, the probability that a bin is weakly aimed is $\leq e^{-c_1/8c_2} < \epsilon/4$, for c_1 sufficiently large. Applying the Chernoff bound again, we obtain

$$\Pr\left[\text{the number of weakly aimed bins is } > \epsilon c_2 n/2\right] \leq e^{-\epsilon c_2 n/8},$$

which decays faster than any inverse polynomial in n. If a bin is heavily aimed, then the adversary must stop at least $c_1/2c_2$ balls in mid-air in order to stop the bin from holding a ball. Thus, the adversary can keep empty at most $\frac{n}{c_1/2c_2} \leq \epsilon c_2 n/2$ heavily aimed bins, for c_1 sufficiently large. The total number of bins with no balls is bounded by the number of weakly-aimed bins plus the number of heavily-aimed bins for which the adversary stops all balls in mid-air. Thus, the total number is at most $\epsilon c_2 n$. The proof for the case of $c_1 n \log n$ balls is similar. ∎

Lemma 2 *For any β (where $\beta \log n$ is the number of butterflies), with high probability the following holds:*

1. *The total number of inputs not filled during phase* 0 *is at most* $2n$.

2. *For any $k > 0$ the total number of cells of level k not filled during phase k is at most $n/\log n$.*

Proof: Line 13 is reached at least αn times during phase 0. Thus, line 6 is reached at least αn times. Level 0 contains βn cells, and each time a processor reaches line 6 it chooses a cell at random. After a processor chooses a cell, the adversary can stop the processor from advancing to line 7. However, there are only n processors. Thus, the situation corresponds to that of Lemma 1, with $c_1 = \alpha$ and $c_2 = \beta$. Choose $\epsilon = 2/\beta$ and the result follows.

Now consider phase $k > 0$. Line 13 is reached at least $\alpha_1 n$ times during the phase. There are n processors. Thus, the loop of lines 10-12 is executed at least $(\alpha - 1)n$ times. Hence, line 11 is reached at least $(\alpha - 1)n \log n$ times. After a processor chooses a cell, the adversary can stop the processor from advancing to line 12. Because there are only n processors, the situation corresponds to that the Lemma 1 with $c_1 = \alpha - 1$ and $c_2 = \beta$. Choose $\epsilon = 1$ and the result follows. ∎

For input cell x and output cell y of butterfly B_i, there is exactly one directed path $x = x^{(0)}, x^{(1)}, \ldots, x^{(d-1)}, x^{(d)} = y$ extending from input x to output y.

Definition 1 *Let $x^{(0)}, x^{(1)}, \ldots, x^{(d)}$ be an input-output path in one of the butterflies B_i. We say that input $x^{(0)}$* successfully affects *output $x^{(d)}$ if for each $k = 0, \ldots, d$, cell $x^{(k)}$ is filled during phase k. We say that input cell $x^{(0)}$ is* strongly affecting *if it successfully affects at least $3/4$ of the outputs of B_i.*

Definition 2 *We say that butterfly B_i is* strongly input dependent *if*

1. *at least half of its inputs are strongly affecting, and*
2. *at most $N/4$ writes to its inputs occur after the completion of phase 0.*

The following lemma guarantees that a sufficient number of butterflies are strongly input dependent. We will show in Lemma 4 that if a butterfly is strongly input dependent, then there is enough connectivity between the inputs and the outputs so that the butterfly is a weak coin.

Lemma 3 *There exists a constant β_0 such that for any $\beta > \beta_0$ with high probability at least $\beta \log n/2$ (i.e., half) of the B_i's are strongly input dependent.*

Proof: We use a counting argument. First we obtain a bound on the number of butterflies for which fewer than half of the inputs are strongly affecting. Consider a butterfly B_i. For $k = 0, \ldots, d$, let $F_k^{(i)}$ denote the set of cells of level k in B_i not filled in phase k. Set $F^{(i)} = \bigcup_{k=0}^{d} F_k^{(i)}$. For input x_0 let $D(x_0)$ be the set of outputs not successfully affected by x_0. By definition, if $x_d \in D(x_0)$, then there must be a cell $x \in F^{(i)}$ such that x is on the path from x_0 to x_d. However, for each $x \in F^{(i)}$ exactly N input-output pairs (x_0, x_d) have x on their connecting path. Thus,

$$N|F^{(i)}| = \sum_{\text{inputs } x_0} |D(x_0)| \geq \sum_{x_0 \text{ not strongly affecting}} |D(x_0)| \geq \sum_{x_0 \text{ not strongly affecting}} \frac{N}{4}.$$

(The last inequality is true since, by definition, if input x_0 is not strongly affecting, then $|D(x_0)| \geq N/4$.) Thus, if in B_i there are $N/2$ inputs that are not strongly affecting, then $|F^{(i)}| \geq N/8$. By Lemma 2, $\sum_i |F^{(i)}| \leq 2n + dn/\log n$. Thus, in at most $(8/N)(2n + dn/\log n) \leq c_1 \log n$ of the B_i's (for some constant c_1) fewer than half the inputs are strongly affecting.

Let ℓ_i be the number of cells of level 0 of B_i written after the completion of phase 0. Each processor writes to level 0 at most once after the phase is completed (thereafter it reads the clock and realizes that the phase completed). Thus, $\sum_{i=1}^{\beta \log n} \ell_i \leq n$. Thus, the number of B_i's for which more than $1/4$ of the input cells are written after phase 0 is completed in at most $4n/N \leq c_2 \log n$ (for some constant c_2). Thus, the total number of B_i's that are not strongly input dependent is $\leq c_1 \log n + c_2 \log n$. With $\beta \geq 2(c_1 + c_2)$ the claim follows. ∎

Definition 3 *Consider the state of a butterfly B_i after the procedure completes. We say that B_i is* 1-dominated *(res.* 0-dominated*) if at least $3/4$ of the outputs have value 1 (res. 0).*

Lemma 4 *There is a constant p such that for any strongly input dependent butterfly B_i,*

$$\begin{aligned} \Pr[B_i \text{ is } 1\text{-}dominated] &\geq p, \\ \Pr[B_i \text{ is } 0\text{-}dominated] &\geq p. \end{aligned}$$

Proof: Let t_0 be the time when phase 0 completes. For each input cell x, consider the last write to x before t_0 and all writes after t_0. Denote by $W(x)$ the set of these writes, and set $W = \bigcup_{\text{input } x} W(x)$. Notice that any write to the inputs not in W has no bearing on the values appearing in the outputs. This is because the levels $k > 0$ are only filled after the phase 0 completes.

Since B_i is strongly input dependent, there are at most $N/4$ late writes to level 0. Thus, by definition $|W| \leq N + N/4$. By construction, for any write $w \in W$, $\Pr[\text{value 1 is written}] = 1/N$, independently. Any input x with $W(x) = \emptyset$ (never written) stores the initial value 0. Thus,

$$\Pr[\text{at all times after } t_0 \text{ all inputs store the value } 0] \leq \left(1 - \frac{1}{N}\right)^{\frac{5N}{4}} \leq e^{-5/4}.$$

In this case, all outputs have the value 0, and B_i is 0-dominated.

On the other hand, since B_i is strongly input dependent, $N/4$ of its inputs are strongly affecting and never overwritten after t_0. With probability $\geq 1 - (1 - 1/N)^{N/4} \geq 1 - e^{-1/4}$ at least one of these affecting inputs has the value 1. In this case, at least $3/4$ of the outputs get the value 1, and B_i is 1-dominated. ∎

Obtaining the outcome of the coin flip r_i is achieved by sampling $\Theta(\log n)$ locations of the corresponding array R_i and taking the majority value.

Corollary 5 *With high probability at least half of the values r_i constitute weak-random-coins.*

Proof: By Lemma 3, at least half of the butterflies B_i are strongly-input-dependent. By Lemma 4 for strongly input dependent butterfly B_i, with high probability $3/4$ of the output cells of B_i share the same value b, which with constant probability is 0 and with constant probability is 1. Thus, with high probability the same guarantee also hold for R_i. In this case, with high probability the majority value of any $\Theta(\log n)$ sample of R_i will provide the value b. Thus, for the $\beta \log n/2$ strongly input dependent butterflies B_i, the coin r_i constitutes a weak random coin. ∎

2.4.2 Work and Contention

Lemma 6 *With high probability the total work of the Weak-Random-Coins procedure and of the Transfer procedure is $O(n \log^2 n)$, and the Alternate procedure is not executed.*

Proof: Consider the Weak-Random-Coin procedure. With high probability at most $O(n)$ invocations of the procedure `Clock-Update` are necessary in order to advance the clock by one. Thus, with high probability the total number of times line 13 is performed is $\Theta(dn) = O(n \log n)$. Each round (lines 1-13) takes $O(\log n)$ operations. There are at most n rounds completed after the clock reaches value $d+1$ (one for each processor). Thus, with high probability the total number of operations is $O((n \log n)+n) \cdot O(\log n) = O(n \log^2 n)$.

Similarly, for the Transfer Procedure. With high probability line 8 is executed $O(n \log n)$ times. The entire loop takes $O(1)$ operations. Hence, with high probability the total work is $O(n \log n)$.

In both cases, the Alternate procedure is not executed. ∎

The following well established fact helps bound the hot-spot contention:

Lemma 7 *Suppose that t balls are randomly thrown into m bins such that with high probability $t \le M$ for some M. Then*

1. *for $M = O(m)$, with high probability no bin receives more than $O(\log m)$ balls, and*
2. *for $M = \Omega(m \log m)$, with high probability no bin receives more than $O(M/m)$ balls.*

The proof is by standard Chernoff bounds.

Lemma 8 *In the Weak-Random-Coins procedure and the Transfer procedure, with high probability any memory location, except those of the Phase-Clock, is accessed at most $O(\log n)$ times.*

Proof: Consider the levels of the butterflies in the Weak-Random-Coin procedure. Each level contains $N\beta \log n = O(n)$ locations in total. The first level is accessed in line 8 (for $k=0$), and subsequent levels are accessed in line 12 (for $k = 1, \ldots, d$). With high probability line 8 is performed $O(n)$ times, and line 12 is performed $O(n \log n)$ times for each k. Each of these invocations accesses a random location of the level. Hence, by Lemma 7, with high probability any single locations is accessed $O(\log n)$ times.

Similarly for the Transfer procedure. First, consider the butterfly output cells. For each k, with high probability there are $O(n)$ reads from random output cells of B_k. There are $O(n/\log n)$ outputs to B_k. Hence, by Lemma 7, with high probability each memory location is accessed $O(\log n)$ times. For the arrays R_k. With high probability there are $O(n)$ writes to random locations of R_k. The size of R_k is n. Hence, by Lemma 7, with high probability each memory location is accessed $O(\log n)$ times. ∎

3 The Convergence Procedure

3.1 The Procedure

We now explain how to achieve consensus using the weak-random-coin procedure of the previous section. A pseudocode description appears in Figure 6. The procedure employs a sequence of arrays $C_0, C_1, \ldots, C_{\beta \log n}$ of size γn, where γ is a constant to be determined later. All the cells of arrays C_k are initialized to Λ.

The convergence procedure works in $(\beta \log n + 1) = \Theta(\log n)$ phases (measured by a Phase Clock), each consisting of $\Theta(n \log n)$ operations. In phase 0, the *filling phase*, each active processor P_i repeatedly chooses a random cell of C_0 and copies its private input, v_{P_i}, into the cell. By the end of this phase with high probability most cells of C_0 contain values (which are either 0 or 1).

Phases $k > 0$ are the *convergence phases*. In these phases the goal is to have all values *converge* to a single consensus value $v \in \{0,1\}$. In each phase k processors sample cells of C_{k-1} and write a value to C_k. A naïve method would have processors write the majority value of the sample. If the initial distribution of 1's and 0's in the array C_0 is far from half and half, then the values

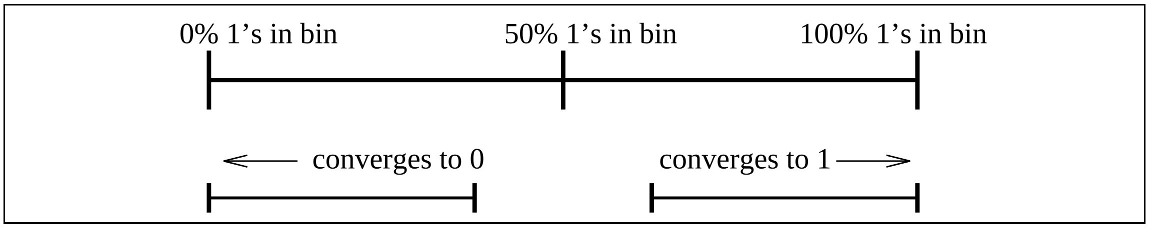

Figure 4: **A naïve solution.** The processors repeatedly sample the array and write the majority value. In the ranges far from the 50% threshold with high probability all the samples have the same majority value and the array converges to one value. Near the 50% threshold, samples can have different majority values and the array may not converge.

will converge to the majority value. However, if the distribution of values is approximately half and half, then this procedure may not converge. (By artfully stopping and starting processors, the adversary could keep the distribution balanced around the 50% threshold.)

Thus, we employ a *two-threshold* policy, as follows. The processors use the values produced by the weak random coin procedure. If the value of the coin is 0 then a sample with 1/3 ones suffices to write back a one. If the value of the coin is 1 then 2/3 ones in the sample are necessary in order to write back a one. Let λ be the fraction of 1's in the sample, and r_k the value of the k-th weak random coin. Then the value, s, written to C_k is

$r_k = 0$	$r_k = 1$
$s = \begin{cases} 1, & \lambda > \frac{1}{3}; \\ 0, & \text{otherwise}; \end{cases}$	$s = \begin{cases} 1, & \lambda > \frac{2}{3}; \\ 0, & \text{otherwise}. \end{cases}$

This policy guarantees that regardless of the initial distribution of values, there is a constant probability that the distribution is far from the threshold. The idea of randomly choosing between two thresholds was first proposed by Rabin [23].

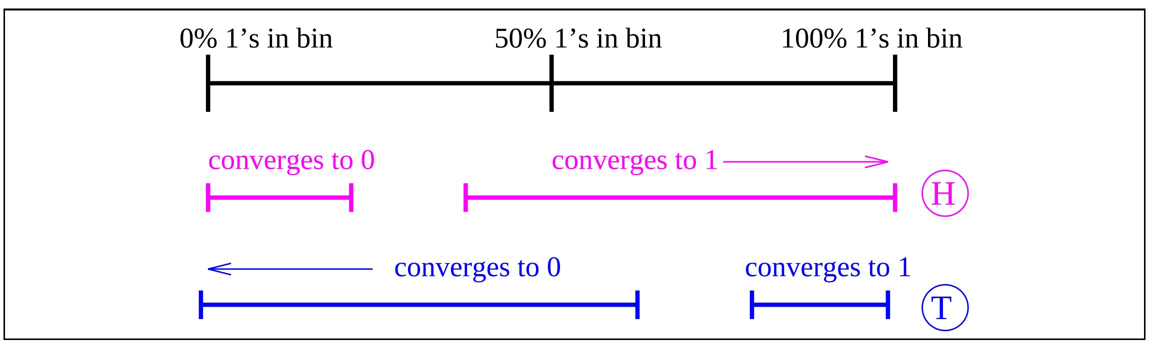

Figure 5: **Fixing the naïve solution.** Before the processors begin converging, they toss a *common weak coin* to decide on the threshold. Now, for any initial number of 1's in the array, the array converges with constant probability. By repeating the converging phase $\Theta(\log n)$ times, the array converges with high probability. Once the array converges, the (overwhelming) majority value will not change.

Convergence (for processor P_i)

repeat
1 **if** total number of operation of processor exceeds n^2 **then**
  run Alternate (Figure 10)
2 $k \leftarrow$ `Read-Clock`
3 **if** $k > \beta \log n$ **then exit** // algorithm completed
4 **else if** $k =$ 'undefined' **then goto** line 14
5 **else if** $k = 0$ **then** // filling phase, $k = 0$
6   $s \leftarrow$ local input v_{P_i} // copy local input
7 **else** // converging phases, $k > 0$
8   sample $b \log n$ locations of array C_{k-1}
9   let λ be the fraction of 1's in the sample
10   Sample $\Theta(\log n)$ locations of R_k and let r_k be the majority value
11   $s \leftarrow \left(\lambda > \frac{1+r_k}{3}\right)$ // threshold decision
12 choose $j \in [1, \ldots, \gamma n]$ at random
13 $C_k[j] \leftarrow s$ // write s in random cell of array C_k
14 `Clock-Update`

Figure 6: **The Convergence Procedure.**

3.2 Analysis

3.2.1 Correctness

Recall that the arrays are of size γn and with high probability the clock remains in each value for at least αn invocations of `Clock-Update`, where α can be tuned to need.

Lemma 9 *For any γ there exists an α such that with high probability for each k at least* 0.9 *of the cells of C_k are written during phase k.*

Proof: Line 14 is reached at least αn times during the phase. Thus, line 12 is reached at least the same number of times. There are n processors and the adversary can stop any processor from advancing from line 12 to line 13. Thus, the situation corresponds to that of Lemma 1 with $c_1 = \alpha$ and $c_2 = \gamma$. Choose $\epsilon = 0.1$ and the result follows for sufficiently large α. ∎

For $1 \leq k \leq \beta \log n$, let m_k be the fraction of cells in C_k that store the value 1 when phase $k+1$ starts. Recall that the sample in line 8 is of size $b \log n$.

Lemma 10 *There exist sample-size parameter b and array-size parameter γ such that for all $k > 0$ the following holds. If $m_k \geq 1/2$ (res. $m_k < 1/2$) and $r_k = 0$ (res. $r_k = 1$) then with high probability all the writes to C_k during phase k are with the value* 1 *(res.* 0*).*

Proof: Suppose $m_k \geq 1/2$ and $r_k = 0$. Let I_k be the time interval when the clock reads k. We prove that with high probability all writes within I_k (to array C_k) output the value 1. There are at most n tardy writes to C_{k-1} during I_k. Thus, throughout I_k the number of 1's in C_k is at least $\gamma m_k - n$. Hence, with γ sufficiently large, if $m_k \geq 1/2$, then *throughout* the interval I_k the fraction of 1's in C_{k-1} is at least 0.4. Thus, by the Chernoff bounds, with high probability any

$b \log n$ sample taken during this interval finds at least $1/3$ of the cells with value 1. Since $r_k = 0$, it outputs the value 1. The argument for $m_k < 1/2$ and $r_k = 1$ is analogous. ∎

Lemma 11 *For β sufficiently large, with high probability, after the completion of the Consensus Procedure at least 90% of the cells in the last array $C_{\beta \log n}$ are filled, and all filled cells store the same value $v \in \{0, 1\}$.*

Proof: By Corollary 5, with high probability for at least half the values of k, r_k constitutes a weak-random coin. For these values of k, there is a constant probability that r_k is 1 and a constant probability that r_k is 0. Thus, for these values of k, by Lemma 10, there is a constant probability that all writes to C_k are with an identical value. If this happens, then in all subsequent phases, all writes will also be the same value. Thus, with high probability after $\beta \log n$ phases all writes are with the same value. By Lemma 9 at least 90% of the cells of $C_{\beta \log n}$ hold this value. ∎

The following lemma hold unconditionally.

Lemma 12 *Any value (other than Λ) appearing in $C_{\beta \log n}$ is one of the original input values (i.e. there is a processor P_i for which this value is its private input value v_{P_i}) .*

Proof: In the first phase the array C_0 is filled only with input values. From then on, values are only copied from one array to the next. ∎

Thus, with high probability, the value appearing in the majority of the cells of $C_{\beta \log n}$ is a consensus value, and processors can sample this array to obtain the consensus value. However, there is still a low probability that consensus is not achieved. In the next section we show how to guarantee that consensus is obtained with no error probability.

3.2.2 Work and Contention

Lemma 13 *With high probability the total work for the convergence procedure is $O(n \log^2 n)$ and the Alternate procedure is not executed.*

Proof: The algorithm completes when the value of the clock reaches $\beta \log n + 1$. With high probability $O(n)$ invocations of the procedure `Clock-Update` suffice in order to advance the clock by one. Thus, with high probability the total number of times that line 14 is performed is $O(n \log n)$. Each round (lines 1-14) takes $O(\log n)$ operations. There are at most n rounds completed after the clock reaches value $c \log n + 1$ (one for each processor). Thus, with high probability the total number of operations is $O(n \log n + n) \cdot O(\log n) = O(n \log^2 n)$. In this case, the Alternate procedure is not executed. ∎

Lemma 14 *In the Convergence protocol, with high probability any memory location, except those of the Phase-Clock, is accessed at most $O(\log n)$ times.*

Proof: Consider the arrays C_k. For each k, the array C_k is accessed in line 13 during phase k and in line 8 during phase $k+1$. Line 13 accesses one random location and line 8 accesses $O(\log n)$ random locations. With high probability line 14 is executed $O(n)$ times for each k, and hence also lines 8 and 13. Hence, by Lemma 7, with high probability each location is accessed $O(\log n)$ times.

Consider the arrays R_k. For each k, array R_k is accessed during phase k in Line 10, which accesses $O(\log n)$ random locations. With high probability line 10 is executed $O(n)$ times during the phase. Hence, by Lemma 7, with high probability each location is accessed $O(\log n)$ times. ∎

4 Guaranteeing Consensus

The algorithm described so far has a polynomially-small probability of failure. In this section we show how to transform the algorithm into a fault-free algorithm in the Las Vegas sense. We do so by augmenting the two stages presented so far with an extra stage, where processors either exit with a *proof* that consensus is reached or else discover a failure and run an alternate consensus protocol.

At first inspection it seems that this Las Vegas addition is in itself a consensus protocol, since the processors apparently agree on whether consensus has been obtained. In fact, the processors do not have to reach consensus and thus we avoid circular reasoning. In particular, some processors may run the alternate protocol while others have a proof of consensus. The guarantee is that if any processor thinks that consensus has been obtained with value v, then any other processor that runs the alternate consensus protocol will ultimately decide on the value v.

4.1 The Write-Once Register

The algorithm relies on a new building block, which we call the *Write-Once-Register (WOR)*. A WOR is a deterministic data structure that has the property that it can be successfully written only once by any single processor. If two processors attempt to write to a WOR simultaneously, then the WOR obtains a "garbage" value. Processors arriving after the WOR is successfully written, cannot change the value of the WOR and can recognize whether the WOR contains a "garbage" value or a valid one. We show how to construct a WOR from three elementary read/write registers having only atomic reads and writes.

Let w_1, w_2, and w_3 be the three atomic read/write registers constituting the WOR w. Each w_i has two fields: $w_i.value$ and $w_i.P\text{-}id$. When a processor writes to any of the registers, it attaches its unique identification number (*P-id*) to the value it writes. The three registers are initialized to Λ. The WOR is said to *hold a value* iff all three registers of the WOR store identical values, including identical *P-id*'s. If the three registers store different values, the value of the WOR is defined to be "garbage". To write to a WOR, the processor writes to the three registers in sequence, checking before each write that no other processor has attempted to write the same WOR. A full description of the WOR procedures is provided in Figure 7.

Lemma 15 *Consider a WOR w and suppose that each processor attempts to write to the WOR at most once. Suppose that two separate reads of w return the values v_1 and v_2. If $v_1, v_2 \notin \{\Lambda, \text{“garbage”}\}$ then $v_1 = v_2$.*

Proof: Assume the contrary. Without loss of generality suppose processor P_1 writes v_1 and that processor P_2 writes v_2. Also, without loss of generality assume that P_1 writes to w_1 before P_2. Then, for P_1 to write w_3, it must be that when P_1 reads w_1 in line 9, it finds $w_1 = (v_1, P_1)$. Therefore, P_2 must reach line 4 after P_1 reaches line 9. Thus, when P_2 reaches line 5 P_1 has already written w_2. The test in line 6 must fail, and P_2 aborts in line 7. Thus, v_2 can never be written in all three registers. ∎

4.2 The Estimate Counting Procedure

We now present a procedure by which processors are either convinced that consensus is achieved or else are warned of a failure and run the Alternate protocol. The procedure is composed of an *estimate counting stage* and then an *exit stage*. The pseudocode appears in Figures 8 and 9.

Write-WOR(v, P_{id})

read w_1, w_2, and w_3
if $w_1 \neq \Lambda$ **or** $w_2 \neq \Lambda$ **or** $w_3 \neq \Lambda$ **then** // if another processor is writing
    abort
write $w_1 \leftarrow (v, id)$
read w_1, w_2, and w_3
if $w_1 \neq (v, id)$ **or** $w_2 \neq \Lambda$ **or** $w_3 \neq \Lambda$ **then** // if another processor is writing
    abort
write $w_2 \leftarrow (v, id)$
read w_1, w_2, and w_3
if $w_1 \neq (v, id)$ **or** $w_2 \neq (v, id)$ **or** $w_3 \neq \Lambda$ **then** // if another processor is writing
    abort
write $w_2 \leftarrow (v, id)$

Read-WOR()

read w_1, w_2, and w_3 in order
if $w_1 = w_2 = w_3$ **then**
    return $w_1.value$
else
    return "garbage"

Figure 7: **The Write-Once Register (WOR).**

Let C_ℓ be the *last array of the consensus protocol* from Section 3, where $\ell = \Theta(\log n)$. In Section 3 we showed that with high probability a consensus value appears in 90% of the γn cells of the array C_ℓ. To obtain a *proof* that this consensus is obtained, the processors collectively estimate the number of 1's and 0's in this array. The procedure employs a butterfly structure W having WOR registers at the first and last levels and regular registers at intermediate levels. Each level of W has γn cells and there are $\log \gamma n + 1$ levels. Nodes of intermediate levels are initialized to 0.

The processors fill the levels of W in sequence. To fill the input level, processors copy the values from C_ℓ, substituting a -1 for 0 in C_ℓ and a 0 for a Λ in C_ℓ. In subsequent levels, node $W[k][j]$ is written with the *sum* of the values appearing in its two predecessors. A garbage value is treated as a 0.

4.3 The Exit-Stage Procedure

Once the last level of W is filled, the processors obtain the consensus value by running the *Exit Stage* procedure, presented in Figure 9. The processors obtain the consensus value by searching for an output having an absolute value greater than $\gamma n/2$. We call such an output a *witness output*. The sign (positive/negative) of the witness output determines the consensus value. A positive witness testifies that most of the cells of C_ℓ have value 1 and hence the processor exits with 1 as the consensus value. Similarly, a negative witness output testifies that most of the cells of C_ℓ have value 0 and hence the processor exits with the value 0. We prove that no two processors will find witnesses with opposite signs.

```
Estimate-Counting (for processor P_i)

repeat
1     if total number of operations of the processor exceeds n^2 then
          run Alternate (Figure 10)
2     k ← Read-Clock
3     if k > log γn then
4         exit                                              // algorithm completed
5     else if k = "undefined" then goto line 15
6     for i = 1 to log n do
7         choose j ∈ [1, ..., γn] at random
8         if k = 0 then                                     // fill inputs
9             if C_ℓ[j] = Λ then                            // cell C_ℓ[j] not filled
10                W[0][j] ← 0                               // fill the substitute 0 for Λ
11            else                                          // C_ℓ[j] has a value
12                W[0][j] ← 2C[c log n][j] − 1              // substitute a −1 for 0
13        else                                              // subsequent levels
14            W[k][j] ← W[k − 1][j] + W[k − 1][j^(k)]       // sum of predecessors
15    Clock-Update
```

Figure 8: **The Estimate-Counting Procedure.**

4.4 The Alternate Procedure

The Alternate procedure (Figure 10) is reached from any of the other procedures if and when something goes wrong. Specifically, the procedure is executed by any processor that performs more than n^2 operations in any other procedure. This is a low probability event, which indicates that something wrong has happened.

In the Alternate procedure the processors participate in an alternate consensus protocol, implemented using any existing polynomial time protocol (e.g. [4]). Note that one processor may detect a failure, while another exits the Exit Stage with a value v. In this case we guarantee that all processors that participate in the alternate procedure start the process with v as their private input value. The Candidate Value Array V, initialized to Λ, helps to provide this guarantee.

4.5 Analysis

4.5.1 Correctness

Lemma 16 *For any $\epsilon > 0$, there exists an array-size parameter γ sufficiently large such that the following holds. With high probability at most an ϵ-fraction of the inputs and an ϵ-fraction of the outputs obtain "garbage" values.*

Proof: Consider an input $w = W[0][j]$. This is a WOR register. Thus, w obtains the value "garbage" only if during the first time that it is written, two different processors try to write to it simultaneously (otherwise the second processor notices that the w is already written and aborts). Specifically, it must be the case that two processors choose w in line 7 (of the estimate counting algorithm) and perform lines 8–12 simultaneously and that this is the first time lines 8–12 are

EXIT-STAGE (for processor P_i)

Repeat

if total number of operations of processor exceeds n^2 **then**
    run Alternate (Figure 10)
choose $j \in [1, \ldots, \gamma n]$ at random // choose random output
$w \leftarrow W[\log \gamma n][j]$
if $|w| > \frac{1}{2}\gamma n$ **then** // if witness output
    $V[i] \leftarrow \frac{1}{2} + \frac{w}{2|w|}$ // write value as candidate exit value
    $w' \leftarrow W[\log \gamma n][j]$ // reread output
    if $w = w'$ **then** // if output value $W[\log \gamma n][j]$ unchanged
        exit with consensus value $v = \frac{1}{2} + \frac{w}{2|w|}$

Figure 9: **Exit-Stage Procedure**

ALTERNATE (for processor P_i)

for all $j \in \{1, \ldots, \gamma n\}$ in order **do**
    overwrite $W[\log \gamma n][j]$ with "garbage"[a] // erase all outputs
$s \leftarrow v_{P_i}$ // use private input
for $j = 1$ **to** γn **do** // scan the array V
    if $V[j] \neq \Lambda$ **then** // a processor exited with value $V[j]$
        $s \leftarrow V[j]$ // use this value
run agreement protocol of [4] using s as the private input value.

[a] Here, the WOR write procedure *is not followed*. The WOR is overwritten regardless of what is already there.

Figure 10: **Alternate Procedure**

applied to cell w. We call such an input w a *clobbered input*. Observe that the adversary can stop processors from advancing from line 7 to lines 8–12. We bound on the number of clobbers that the adversary can produce.

An input cell is in one of three states: *empty*, *successfully written*, or *clobbered*. We call each execution of lines 7–11 a *round*. We say that a round is *effective* if the input cell that it chooses in line 7 was never selected before. We say that a round *clobbers* if it chooses an input cell for which the effective round that selected it is still active. We say that the round is *neutral* if it chooses an input that was already selected by a round no longer active.

Set $\epsilon' = \epsilon/3$. Let t_0 be the first time when $(1-\epsilon')\gamma n$ inputs are chosen (in line 7). Consider the nonneutral rounds before t_0. At any given time before t_0 there are at most n active rounds (because there are only n processors), and at least $\epsilon'\gamma n$ cells not yet chosen. Thus, for a nonneutral round R before t_0,

$$\Pr[\text{round } R \text{ clobbers} \mid \text{round } R \text{ is not neutral}] \leq \frac{n}{\epsilon'\gamma n} = \frac{1}{\epsilon'\gamma}. \tag{1}$$

Moreover, for any two rounds, the probabilities that they clobber are negatively correlated, and so the Chernoff bounds still apply. Let g be the number of effective rounds started before t_0, and let h be the number of clobbering rounds started before t_0. By Eq. (1) and the Chernoff bound,

$$\Pr\left[\frac{h}{g+h} \geq \frac{2}{\epsilon'\gamma}\right] \leq \left(\frac{e}{4}\right)^{(s+h)/(\epsilon'\gamma)} . \tag{2}$$

By definition $g = (1-\epsilon')\gamma n$. Thus, for sufficiently large γ with high probability up until t_0 there are at most $h = \epsilon'\gamma n$ clobber rounds. At most n of these effective rounds do not complete before t_0. Thus, at least $g - h - n \geq (1-3\epsilon')\gamma n = (1-\epsilon)\gamma n$ cells are successfully written, as claimed. A similar proof applies to the outputs. ∎

Recall that the clock remains in each integral value for at least αn clock updates.

Lemma 17 *For any γ there exists a clock parameter α such that with high probability the following holds. For any $0 \leq k \leq \log \gamma n$, at most $n/\log n$ of the cells of level k are not filled during round k.*

Proof: The proof is similar to the proof of Lemma 2. In each round, with high probability line 15 is reached at least αn times. Thus, lines 7–14 are invoked at least $\alpha n \log n$ times. In line 7 cell j is chosen at random. The adversary can stop at most n processors from continuing from line 7 to lines 8–14. Thus, by Lemma 1, setting $c_2 = \gamma$ and $\epsilon = 1/\gamma$, there exists an $\alpha = c_1$ such that with high probability at most $n/\log n$ are not filled in the round. ∎

Recall that a *witness output* has an absolute value greater than $\gamma n/2$.

Lemma 18 *1. With array-size parameter γ and clock parameter α sufficiently large with high probability at least half of the outputs are witness outputs.*

2. All witness outputs are either all positive or all negative.

Proof:

(1) Set $d = \log \gamma n + 1$. Consider a cell $c = C_\ell[j_0]$ from the converging array (Figure 6) and an output cell $w = W[d][j_1]$ from the estimate counting butterfly (Figure 8).

There is exactly one directed path $c, w_0, \ldots, w_d = w$ from c to w. Suppose that for all rounds k, cell w_k is filled during round k and that neither w_0 nor w have the value "garbage." In this case we say that c *affects* w. If c affects w and $c = 1$, then c contributes 1 to the sum stored in w. If c affects w and $c = 0$, then c contributes -1 to the sum stored in w. By Lemma 11 with high probability 0.9 of the cells of C_ℓ store the same value. In this case, if w is affected by more than 0.9 of the c's, then it is a witness. Using a counting argument we show that at least half the w's are affected by 0.9 of the c's.

For each output w, let $I(w)$ be the set of cells $C_\ell[j]$, such that $C_\ell[j]$ does not affect w. For level k, let $F^{(k)}$ be the set of cells not filled during round k. Let G_{in} be the set of inputs having the value "garbage" and G_{out} be the set outputs having the value "garbage". For any output w and cell $c = C_\ell[j]$, such that c does not affect w (i.e., $c \in I(w)$), one of the following must hold:

1. $w \in G_{\text{out}}$;
2. $W[0][j] \in G_{\text{in}}$;
3. there exists a k and $w_k \in F^{(k)}$ such that w_k is on the path from $W[0][j]$ to w.

For each $w_k \in F^{(k)}$ there are at most γn pairs (c, w) such that w_k is along the path from c to w. Let B be the set of outputs, w, such that $|I(w)| \geq 0.1\gamma n$ (these are the "bad" outputs). Thus, with Lemmas 17 and 16, we get

$$\sum_{w \in B} 0.1\gamma n \leq \sum_{w \in B} |I(w)| \leq \sum_{w \text{ output}} |I(w)| \leq \left(\gamma n \sum_{k=0}^{d} |F^{(k)}|\right) + |G_{\text{in}}| + |G_{\text{out}}| \leq \gamma n \left(\frac{dn}{\log n} + 2\epsilon\right).$$

Thus, for sufficiently small ϵ, $|B| \leq \gamma n/2$.

(2) Suppose output w is a witness. Assume without loss of generality that w is positive. Thus, there must have been at least $\gamma n/2 + 1$ inputs with the value 1. Inputs are WOR registers. Thus, at no time can there be $\gamma n/2$ inputs with value 0. (Note that this claim holds unconditionally). ∎

The following corollary and the next two lemmata hold unconditionally.

Corollary 19 *There exists a value $b \in \{0, 1\}$ such that:*

1. *b is the only value possibly written to any entry of the array V (line 5 of the Exit-Stage),*
2. *All processes successfully terminating the Exit Stage at line 8 exit with the the consensus value b,*
3. *b is one of the original input values.*

Proof: By Lemma 18, there are no two witnesses with opposite signs. If all witnesses are positive then set $b = 1$ and otherwise $b = 0$.

1. Entry $V[j]$ is only written in line 5 of the Exit Stage. In this case, it is gets value 1 if the witness is positive and value 0 if the witness is negative. Since all witness have the same sign, all writes to V are with the value b.
2. The exit value in line 8 is the same as that written in line 5 to $V[j]$.
3. By Lemma 12 any value appearing in $C_{\beta \log n}$ (other than Λ) is an input value. Thus, if all inputs are 1 then there is no negative value in the first level of the array W ($W[0][\cdot]$), and hence no negative witness, and hence $b = 1$. Similarly if all inputs are 0, $b = 0$.

∎

This completes the proof that with high probability consensus is achieved. We now consider the low-probability event of failure and prove that consensus is still obtained.

Lemma 20 *Any value s used as an input value to the alternate consensus protocol in line 7 of the Alternate procedure is one of the original input values.*

Proof: In line 3 of the Alternate procedure s is set to be V_{P_i}. This value can only change in line 6 to a value $V[j]$ for some j. In this case, by Corollary 19, $V[j]$ is an input value. ∎

Lemma 21 *If any processor exits the Exit Stage procedure in line 8 with a consensus value v, then any processor participating in the Alternate procedure enters the alternate consensus protocol with $s = v$.*

Proof: Suppose P_1 exits with the value v and P_2 participates in the alternate execution. Let $x = W[\log \gamma n][j]$ be the witness output found by P_1. Processor P_1 reads x twice (lines 3 and 6). In lines 1–2 of the Alternate-Route processor P_2 erases all outputs. Thus, P_1 must have read w for the second time before P_2 reached line 4 of the Alternate-Route . By this time, P_1 has already written the value v in $V[1]$. Thus, at line 5, P_2 finds the value v in V, and uses it as its input value for the alternate protocol (lines 6-7). By Corollary 19 no other value can be written to any other entry in V. ∎

4.5.2 Work and Contention

Lemma 22 *With high probability the Estimate-Counting and the Exit Stage procedures complete in $O(n \log^2 n)$ total work and the Alternate procedure is not executed.*

Proof: Consider the Estimate Counting procedure. The procedure completes when the value of the clock reaches $\log \gamma n = O(\log n)$. With high probability at most $O(n)$ invocations of `Clock-Update` are necessary to advance the clock by one. Thus, the total number of times line 15 is performed is $O(n \log n)$. Each round (lines 1-15) takes $O(\log n)$ operations. At most n rounds complete after the clock reaches value $\log \gamma n$ (one for each processor). Thus, with a high probability the total number of operations is $O((n \log n) + n) \cdot O(\log n) = O(n \log^2 n)$ and the Alternate-Route is not reached.

Consider the Exit-Stage procedure. By Lemma 18, with high probability at least half of the outputs are witnesses. By Lemma 16 at most ϵ ($\epsilon < 1/4$) of these witness output may change its value (to "garbage"). Thus, there is a constant probability of a processor executing the Exit Stage procedure to hit (line 3) on such a witness output which does not change its value. In this case, the processor will exit in line 8. Hence, with high probability every processor completes within $O(\log n)$ rounds of lines 1-7. Each round takes $O(1)$ steps, for a total of $O(n \log n)$.

The Alternate-Route procedure may be reached in two case:

1. Less than a constant fraction of the outputs are such that they are witnesses and their value does not change. By Lemmata 18 and 16, this event has polynomially low probability.

2. At least a constant fraction of the outputs are non-changing witnesses, but the processor fails to find such an output in n^2 operations. Each round of lines 1-7 takes $O(1)$ operations. Thus, the probability of this event is $\leq 2^{-\Omega(n)}$.

∎

Lemma 23 *In the Estimate-Counting and Exit Stage procedures with high probability any memory location, except those of the Phase-Clock, is accessed at most $O(\log n)$ times.*

Proof: Consider the Estimate-Counting procedure. The array C_ℓ is accessed in lines 9 and 12 during the first phase. The first level of the butterfly is accessed in lines 10 and 12, also during the first phase. For subsequent levels, level k is accessed in line 14, during phases k and $k+1$. In each round (lines 1–15), each of the above mentioned lines is executed $O(\log n)$ times. With high probability each phase consists of $O(n)$ rounds of lines 1-15. Thus, the total number of accesses to any single level is $O(n \log n)$, and these are to random locations. Hence, by Lemma 7, with high probability each location is accessed $O(\log n)$ times.

Consider the Exit-Stage procedure. By Lemma 18, with high probability at least half of the outputs are non-changing witnesses. In this case, with high probability for each processor, after at most $O(\log n)$ rounds of the Exit-Stage procedure, it finds a non-changing witness output and exits. Each round accesses $O(1)$ memory locations and there are n processors. Hence, by Lemma 7, with high probability each location is accessed $O(\log n)$ times. ∎

5 Contention on the Phase Clock

In this section we analyze the contention on the Phase Clock, and show how its hot-spot contention can be reduced to $O(\log n)$. In order to do so, we must first describe the structure of the Phase Clock, as presented in [11].

The Aumann-Rabin Phase Clock. The Phase Clock as described in [11] is composed of three arrays A_0, A_1, and A_2, each containing $cn = \Theta(n)$ cells. The clock arrays drive each other in a circular fashion. Initially, all cells of A_0 are initialized to 0, those of A_1 to 1 and those of A_2 to 2. Now, the value 2, in A_2, will start driving the value of A_0 from 0 to 3. This, in turn, drives the value of A_1 from 1 to 4, and so forth. The Phase Clock supports two procedures: `Read-Clock`, which returns the current value of the clock, and `Update-Clock`, which allows processors to participate in advancing the clock. The procedures are presented in Figure 11.

Data Structure: 3 arrays, A_0, A_1, A_2, each with cn cells; A_j initialized to j.

Update-Clock
for $j = 0, 1, 2$ **do**
    sample $\Theta(\log n)$ cells of A_j
    if at least 80% of the sampled cells equal i **then**
        write $j+1$ to $A_{(j+1) \bmod 3}$

Read-Clock
sample $\Theta(\log n)$ cells of A_0
if at least 70% of the sampled cells share the same value i **then**
    return $i/3$
else
    return "undefined"

Figure 11: **Phase Clock of Aumann-Rabin [11].**

Reducing the Contention. Using the Phase Clock as described so far results in $O(\log^2 n)$ hot-spot contention. The reason is that during each phase each array is accessed $\Omega(n \log n)$ times, and the same three arrays are reused again and again for all $O(\log n)$ phases. In order to reduce the contention to $O(\log n)$ we "unfold" the clock construction, as follows. Suppose we want to count for k phases. Then, instead of reusing the same three arrays, we keep $3k$ distinct arrays, $A_0, \ldots, A_{3k-1}$. All arrays are initialized to 0. The arrays drive each other in a sequential manner, with A_0 driving the values written in A_1 from 0 to 1, which, in turn, drives the values in A_2 to 2, and so forth. The *value* of the clock is defined as the highest number i, for which the array A_{3i} is *filled*, i.e., at least 70% of the nodes hold the value $3i$. The resulting code for `Read-Clock` and `Update-Clock` is provided in Figure 12. As can be seen, the code for the unfolded clock is analogous to the original one, except that in each phase the processors operate on a different triplet of array ($A_{3\ell}$, $A_{3\ell+1}$ and $A_{3\ell+2}$ for phase ℓ). The only difference is that processors must first determine the triplet of arrays on which to operate. In the `Update-Clock` procedure, the processor does this by first performing a `Read-Clock` to find the current phase. For the `Read-Clock` procedure, the processor samples the arrays in an increasing order, until it finds the first array (whose index divides by 3) that is not filled. For efficiency reasons the search for the last filled array does not begin with A_0, rather from the last array already known to be filled. This guarantees that while some invocations of `Read-Clock` may take $O(k \log n)$ steps, the amortized cost is $O(\log n)$.

Data Structure: $3k$ arrays, $A_0, \ldots, A_{3k-1}$, each with cn cells; all initialized to 0.

Update-Clock

$\ell \leftarrow$ `Read-Clock`
for $j = 0, 1, 2$ **do**
    sample $\Theta(\log n)$ cells of $A_{3\ell+j}$
    if at least 80% of the sampled cells equal $3\ell + j$ **then**
        write $3\ell + j + 1$ to $A_{3\ell+j+1}$

Read-Clock

Static Variable: ℓ - most recent known clock value (remembered from previous execution); initialized to 0

repeat
sample $\Theta(\log n)$ cells of $A_{3(\ell+1)}$
if at least 70% of the sampled cells share the same value $3(\ell + 1)$ **then**
    return $\ell = \ell + 1$
else
    return ℓ
until $\ell = k + 1$

Figure 12: **"Unfolded" Phase Clock.**

We thus obtain behavior analogous to that of the original [11] clock. In particular, with high probability advancing the clock from one value to the next takes $\Theta(n)$ invocations of `Update-Clock`.

Lemma 24 *Suppose that the Unfolded Phase Clock is used such that each invocation of* `Read-Clock` *is coupled with an invocation of* `Update-Clock`. *Then, with high probability any shared-memory location is accessed* $O(\log n)$ *times.*

Proof: Consider an array A_i, $i = 3\ell + j, 0 \leq j \leq 2$. The array A_i may be accessed in the following cases:

1. During phases ℓ and $\ell - 1$, invocations of `Update-Clock` and `Read-Clock` reading and writing A_i. With high probability there are $\Theta(n)$ invocations of `Update-Clock` before the any phase is over. Since each `Read-Clock` is coupled with a `Update-Clock`, with high probability there are also only $O(n)$ invocations of `Read-Clock` during these phase. Thus, the total number of such invocations is $O(n)$. Each such invocation accesses at most $O(\log n)$ random location of the array.

2. Invocations by tardy processors, acting after the completion of phase ℓ. Each processor performs one such invocation before realizing that the phase is over. Thus, the total number of such tardy invocations is $O(n)$. Each such invocation accesses at most $O(\log n)$ random location of the array.

Thus, with high probability there are a total of at most $O(n \log n)$ accesses to any array A_i, and these are to random locations. Hence, by Lemma 7, the result follows. ∎

Corollary 25 *In the procedures Weak-Random-Coins, Transfer, Convergence, Estimate-Counting, and Exit-Stage, with high probability any shared memory location of the Phase Clocks is accessed* $O(\log n)$ *times.*

6 Conclusions

We thus obtain the following:

Theorem 1 *The procedures described in Sections 2-5 together guarantee consensus. With high probability the total work is* $O(n \log^2 n)$*, the hot spot contention is* $O(\log n)$ *and the contention cost is* $O(n \log^3 n)$*. The same bounds also hold for the expected total work, hot-spot contention and contention cost, respectively.*

Proof: First, we prove correctness. If no processor exits the Exit-Stage procedure in line 8, then all processors participate in the Alternate procedure and correctness follows from that of the protocol of [4]. Otherwise, by Corollary 19 all processors exiting in line 8 of the Exit Stage, have the same consensus value v, which, by Lemma 21, is also the input value for all processors participating in the Alternate procedure. Thus, v must also be the output of all processors participating in the Alternate procedure.

The high-probability bound for total work follows from Lemmata 6, 13, and 22. The high-probability bound for the hot-spot contention follows from Lemmata 8, 14, 23, and Corollary 25. For the contention cost, note that for any shared-memory variable v, if the total number of accesses to v is c then the contention cost of any operation on v is also bounded by c. With high probability any memory location is accessed at most $O(\log n)$ times, and hence with high probability the contention cost of any operation is $O(\log n)$. With high probability the total number of operations is $O(n \log^2 n)$. Hence, with high probability the total contention cost is $O(n \log^3 n)$.

Next, we prove the bounds on the expected total work. With high probability the total work is $O(n \log^2 n)$. Consider the low-probability event that the work is more than $O(n \log^2 n)$. This event occurs with probability $< n^{-c}$, where c can be set to any constant. In this low probability event, the expected total work is still bounded by $O(n^4)$ per processor, based on the complexity of the alternate protocol which uses [4]. Thus, setting $c > 5$, the low probability event adds at most a constant to the total work, which thus remains $O(n \log^2 n)$. Similarly, the hot-spot contention in the low probability event is bounded by $O(n)$, and hence the expected hot-spot contention remains $O(\log n)$. The expected contention cost in the low probability event is bounded by $O(n \cdot n^4)$, and hence the overall contention cost remains $O(n \log^3 n)$. ∎

Open Problems. Several problems remain open:

- Can the total work be further reduced while keeping the contention low? Specifically, can the total work be reduced to $O(n \log n)$ without increasing the asymptotic hotspot contention?

- Can the contention cost be reduced?
- Is it possible to obtain analogous low contention results for one of the stronger adversary schedulers (the Write Oblivious or the Local Oblivious)?

Acknowledgements. We are grateful to the two anonymous referees for many helpful and constructive comments. The presentation of this paper has benefited much from their comments.

References

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

[2] J. Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *Journal of the ACM*, 45(3):415–450, 1998.

[3] J. Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, September 2003.

[4] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, September 1990.

[5] J. Aspnes and O. Waarts. Randomized consensus in expected $O(N \log^2 N)$ operations per processor. *SIAM Journal on Computing*, 25(5):1024–1044, October 1996.

[6] H. Attiya, N. Lynch, and N. Shavit. Are wait-free algorithms fast? In *Proceedings of the 31st Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 55–64, 1990.

[7] Y. Aumann. Efficient asynchronous consensus with the weak adversary scheduler. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 209–218, 1997.

[8] Y. Aumann and M. Bender. Efficient asynchronous consensus with the value-oblivious adversary scheduler. In *Proceedings of the 23rd International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 622–633, 1996.

[9] Y. Aumann, M. A. Bender, and L. Zhang. Efficient execution of nondeterministic parallel programs on asynchronous systems. *Information and Computation*, 139(1):1–16, November 1997.

[10] Y. Aumann and A. Kapah-Levy. Cooperative sharing and asynchronous consensus using single-reader single-writer registers. In *Proceedings of the 10th ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 61–70, 1999.

[11] Y. Aumann and M. O. Rabin. Clock construction in fully asynchronous parallel systems and pram simulation. *Theoretical Computer Science*, 128:3–30, 1994.

[12] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, 1983.

[13] G. Bracha and O. Rachman. Randomized consensus in expected $O(n^2 \log n)$ operations. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WDAG)*, volume 579 of *Lecture Notes in Computer Science*, pages 143–150, 1991.

[14] T. D. Chandra. Polylog randomized wait-free consenus. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 166–175, 1996.

[15] B. Chor, A. Israeli, and M. Li. Wait-free consensus using asynchronous hardware. *SIAM Journal of Computing*, 23(4):701–712, August 1994.

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

[17] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, November 1997.

[18] C. Dwork, D. Shmoys, and L. Stockmeyer. Flipping persuasively in constant time. *SIAM Journal of Computing*, 19(3):472–499, 1990.

[19] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[20] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[21] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[22] C. Martel, A. Park, and R. Subramonian. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proceedings of the 31st Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 590–599, 1990.

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

[24] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus - making resilient algorithms fast in practice. In *Proceedings of the 2nd ACM-SIAM Annual Symposium on Discrete Algorithms (SODA)*, pages 351–362, 1991.

Appendix

A Weak Adversary Schedular Models

In this section we provide a review of the different weak adversary scheduler models, and provide a comparison of their relative strengths. For comparison, it is worthwhile to recall the definition of the strong scheduler:

The Strong Scheduler: At each point in time, the strong scheduler chooses the next processor to operate. If a processor is granted the right to operate, the processor performs its enabled atomic operation. The adversary scheduler can base its choice on the entire history of the state of the system including the protocol, the processors' input values, the history of all operations performed, and all values stored anywhere in the system's memory. In particular, the outcome of a coin toss is available to the scheduler immediately after the coin is tossed, even if the outcome is only stored in private memory. The only restriction on the scheduler's knowledge is that the scheduler cannot foresee the outcome of future coin tosses.

The weak adversary models place limitations on the information available to the scheduler. These limitations generally prevent the scheduler from learning certain information about the outcomes of random coin tosses. The code of the protocol and the processors' input values are available to the scheduler in all weak models.

Oblivious Scheduler: This scheduler has no knowledge of the outcomes of coin tosses, ever. Thus, effectively the entire schedule must be determined in advance. The Oblivious adversary is the standard adversary used in the APRAM model (e.g. [22]).

Value-Oblivious Scheduler: This scheduler is used in this paper and in [10]. The scheduler does not get direct access to outcomes of coin tosses. Rather, the scheduler can learn the outcome of a coin toss only insofar as the outcome affects the dynamics of the execution, i.e., what operations are executed and what memory locations are accessed. Enabled operations are known to the scheduler.

Write-Oblivious Scheduler: This scheduler is used in [14] and [7]. The outcome of a coin toss is hidden from the scheduler until some processor *reads* its value. Writing the outcome of a coin toss does not reveal its value, and similarly copying this value from private memory to public memory does not reveal its value. Enabled operations are known to the scheduler. The rational for this scheduler is similar to that of the Value-Oblivious scheduler. Namely, as long as no processor reads the value, the value cannot affect the dynamics of the execution and hence does not affect the schedule.

Local-Oblivious Scheduler: This scheduler is used in [15] and [1]. The scheduler knows the outcome of a coin toss as soon as it is written to shared memory. However, local operations based on the outcome of the coin toss are hidden from the scheduler. Also, if the value written to shared memory depends on the outcome of a coin toss (e.g., it is x if *heads* and y if *tails*), then the scheduler does not know the value until *after* it is written to shared memory. (In [1] the model is described as allowing an atomic *p-write* operation, i.e., *write* with probability p and *do nothing* with probability $1 - p$. For the algorithms of [15] and [1] this model is equivalent to Local-Oblivious.)

The Oblivious Scheduler is strictly weaker from all other schedulers mentioned here. By definition, the Value-Oblivious scheduler is weaker than the Write-Oblivious scheduler. The Value-Oblivious scheduler is also weaker than the Local-Oblivious scheduler. The reason is that it is possible to convert any protocol for the Local-Oblivious into a protocol for Value-Oblivious, by replacing any sequence of randomized local steps resulting in a specific output value v by a corresponding sequence of deterministic steps, coin tosses, and arithmetic operations resulting in the same output value. The detailed are beyond the scope of this paper. The Write-Oblivious and the Local-Oblivious schedulers are not directly comparable. On the one hand, the Local-Oblivious scheduler knows the outcome of a coin toss immediately upon writing it to shared memory, unlike the Write-Oblivious scheduler, which hides the value until it is read. On the other hand, local operations using the outcome of a coin toss would reveal its value to the Write-Oblivious scheduler, whereas this information is hidden from the Local-Oblivious scheduler.