Efficient Asynchronous Consensus with the Value-Oblivious Adversary Scheduler

Yonatan Aumann[1⋆] and Michael A. Bender[2⋆⋆]

[1] Department of Mathematics and Computer Science
Bar-Ilan University
Ramat-Gan, Israel
Email: aumann@bimacs.cs.biu.ac.il

[2] Aiken Computation Laboratory
Harvard University
Cambridge, MA 02138
Email: bender@das.harvard.edu

Abstract. We consider the power given to adversary scheduler of an asynchronous system and define the *value-oblivious scheduler*. At each step this scheduler determines the next processor to operate based on the full history of the *dynamics* of the execution; the scheduler is oblivious to the intermediate *values* the processors manipulate. We argue that the value-oblivious scheduler captures the possible sources of asynchrony in real systems.

Assuming the value oblivious adversary, we study the asynchronous consensus problem in the shared-memory setting with atomic reads and writes. We present a probabilistic algorithm that obtains consensus in $O(n \log^2 n)$ *total work*. Here, total work is defined as the total number of steps performed by *all* processors collectively. Thus, the amortized work per processor is $O(\log^2 n)$.

1 Introduction

The Problem. The consensus problem on n processors is defined as follows. Initially, each processor P_i has a private input value $v_{P_i} \in \{0, 1\}$. The goal is to obtain one *consensus value* v, agreed upon by all processors, such that $v \in \{v_{P_i}\}_{i=1}^{n}$. If all processors operate synchronously, then consensus is easily attainable. In reality, however, processors are unreliable and may fail or become exceedingly slow. Even in a fault-free situation processors may encounter delays as a result of page faults, exhausting scheduling quotas, interrupts and network communication delays. Thus, we seek a protocol that obtains consensus using whatever time slices it is awarded by the participating processors.

Efficiency. The first requirement from a consensus protocol is that it be *wait-free*: each processor must obtain the consensus value after a bounded number, T, of its *own* steps, regardless of the speeds of the other processors. Most previous work focuses on minimizing the value of T. This measure, however, only considers the worst case bound *per processor* and does not address the question of the total efficiency of the protocol. In particular, there is no advantage if more processors

⋆ This work was done while the author was at Harvard University, supported in part by ONR contract ONR-N00014-91-J-1981.

⋆⋆ The author was supported by NSF contract CCR-9313775.

cooperate in the protocol. Thus, Attiya, Lynch, and Shavit [6] observe that many wait-free protocols are slow in “normal” executions. They suggest also measuring the number of *rounds* in executions having no or relatively few faulty processors.

In this paper, we employ a strong, unified complexity measure, previously used in the context of A-PRAM (*Asynchronous* PRAM) computations (see [16]). This measure directly gauges the efficiency with which the protocol uses whatever cycles are provided by the processors. In the We count the *total number of instruction cycles* used by the protocol. We present a protocol for n asynchronous processors to obtain consensus in $O(n \log^2 n)$ total number of instruction cycles. After this many instructions, consensus is obtained regardless of the distribution of the instructions among the processors and regardless of the processors' relative speeds and interleaving activity. In particular, if the system is near synchronous, then each processor performs only $O(\log^2 n)$ steps; if only one processor is fast, then this processor reaches a decision after $O(n \log^2 n)$ steps; if processors alternate arbitrarily between running slowly and quickly, then the protocol still makes optimal use of all instruction cycles. The algorithm is randomized and the behavior is guaranteed in both the Las Vegas and Monte Carlo senses.

The protocol runs in the shared-memory setting, and assumes a *value-oblivious adversary scheduler*. This adversary model, similar to that described by Rabin in the context of mutual exclusion [18], is somewhat weaker than the adversary usually considered for consensus algorithms. We describe the model in detail.

The Model. The consensus problem can be viewed as a game between the processors, which seek to attain consensus, and an adversary scheduler, which tries to prevent them from doing so. At each step, the scheduler decides which processor operates next. How much information should be made available to the adversary for making this decision? Most previous papers assume a dynamic adversary scheduler (e.g. [12, 10, 11, 1, 3, 14, 4]) having full knowledge of the entire state of the system at all previous steps.

In the present work, we make a distinction between two separate components of the system's history: the *control history* and the *content history*. The *control history* includes all information regarding the *dynamics* of the computation. In particular, the control history specifies the active processors in previous steps, the operations performed by these processors (read, write, add etc.), and the addresses/locations of the operands of the operations. We define the *value-oblivious scheduler*, which has full knowledge of control history, but is not provided with the actual *values* stored in the memory locations (the *content history*). The scheduler can gain information regarding the value stored in a memory location only insofar as this value affects the dynamics of the computation. In addition, in order to account for pipeline effects and delays due to page faults, we assume that the scheduler has the *operation lookahead* capability: it knows the next operation to be performed by each of the processors. We also assume that the scheduler has full knowledge of the processors' input values and their programs. In fact, we allow the adversary to have full knowledge of the values stored in all memory locations other than those used to store intermediate values of our algorithm.

We argue that the power given to the value-oblivious scheduler faithfully captures the possible sources of asynchronous behavior of real systems. Common causes of asynchrony, such as hardware failures, clock skews, operating system interrupts, cache misses, page faults, and network delays, all depend on the dynamics of the execution. However, on modern computers, most integer ALU operations run in the same number of cycles regardless of the values of the operands. (Our algorithms do not use division, where an interrupt may depend on the computed value.) Similarly, the time to load and store data from memory does not depend on the data. Thus, the value-oblivious adversary captures all the real-world sources of asynchrony.

Previous Related Work. Fischer, Lynch, and Paterson [12] prove the impossibility of deterministic asynchronous consensus in the message-passing model, even if only one processor fails. Chor, Israeli, and Li [10] and Loui and Abu-Amara [15] show that the same holds in the shared-memory model (see also [11]). Randomized solutions for the standard, full knowledge adversary scheduler, are presented in [10, 1, 3, 8, 4]. All of these solutions require $\tilde{O}(n)$ operations per-processor. Our model and results differ in several ways. On the one hand, we consider a weaker adversary scheduler. In addition, we allow multiple-writer registers, which are not used in the above. On the other hand, our work bounds are much stricter. Even for the best of the previous results, the total number of instructions required for completion may be as high as $O(n^2 \log n)$. If most processors are active, then our protocol completes in $O(\log n)$ steps, versus $O(n \log n)$ from the above. The methods used in [10, 1, 3, 8, 4] focus on the number of *rounds* and are ill-suited for our *work* bounds. Thus, we introduce new techniques that allow the protocol to make optimal usage of *instruction cycles*, regardless of the particular processor executing the cycle.

Saks, Shavit, and Woll [19] consider the problem of obtaining 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. The [19] algorithm provides a good solution for the case that most processors are synchronous and only a small number are faulty. However, it does not provide the flexibility that our work bounds provide for the general asynchronous case. We allow all processors, faulty and nonfaulty, to behave in an arbitrary asynchronous fashion. We emphasis, however, that the [19] result assumes the stronger full-knowledge adversary model and single writer registers, while ours assumes the value-oblivious scheduler and multi writer registers.

Recently, Chandra obtained improved work bounds for consensus in an adversary model similar to ours. In [9], Chandra presents a consensus protocol with expected $O(\log^2 n)$ steps per processor. Thus, the total work is identical to ours, but his result also provides a bound on the work *per processor*. We note however, that the expected contention on the multi-writer registers in [9] can be as high as $O(n)$. In our algorithm, the expected contention is $O(1)$, and with high probability never do more than $O(\log n)$ processors try to access the same register concurrently. Thus, our results also hold in the model that processors queue for shared registers, or the model that all concurrent accesses to the same register fail.

Definitions and Notations. The system consists of n *processors* $\{P_i\}_{i=1}^n$ sharing a common memory space. Reading and writing shared memory cells are atomic operations. W.l.o.g. we assume the computation proceeds in discrete *time steps*, and that at each time step exactly one processor performs one atomic operation. A *schedule* S, is a function $S : \mathbb{N} \rightarrow \{1, \ldots, n\}$ that associates one active processor $P_{S(t)}$ with each time step t. The schedule is determined by a dynamic *scheduler* $\mathcal{S}$; the scheduler must determine $S(t)$ before time step t.

We define the information available to the scheduler for making this decision. For each time step $t' < t$, let $A(t')$ be a the triplet $(S(t'), O(t'), L(t'))$, where $P_{S(t')}$ is the processor active at t', $O(t')$ the operation performed by $P_{S(t')}$ at time t', and $L(t')$ is the set of memory locations/registers accessed in this operation. The sequence $(A(1), \ldots, A(t-1))$ is the *control history* for time t. In addition, for processor P_i and time t let $N(t, i)$ be the action P_i will perform as soon as it is active. (In particular, $N(t, i) = A(t)$ if the scheduler sets $S(t) = i$.) We call the set $\{N(t, i)\}_{i=1}^{n}$ the *operation lookahead set* for time t. We assume that the scheduler is provided with the following: (i) the entire content of the memory before starting the execution of the algorithm, (ii) the control history for time t, and (iii) the operation lookahead for time t. Efficiency is measured by the total number of steps performed in the system, summed up over all processors. This total may be arbitrarily distributed among the processors. We say that an event E occurs *with high probability (w.h.p.)* if for any $c > 0$ there exists a proper choice of constants such that $\Pr[E] \geq 1 - n^{-c}$.

The Phase Clock. Many of our algorithmic constructs operate in *phases*, each requiring of $\Theta(n)$ or $\Theta(n \log n)$ work. Measuring work in the asynchronous setting is nontrivial. Aumann and Rabin [7] present a general *Phase Clock* structure which provides the necessary tool. We briefly review the main features of the [7] "clock" structure. At any point in time the Phase Clock has a *value* which is either an integer, or "undefined". The integral values are non-decreasing in time. The value of the clock is initialized to 0. The `Clock-Update` procedure, which takes $O(1)$ operations, allows the processors to participate in advancing the clock. For any n and α there exists a Phase Clock such that: (i) after every $O(n)$ invocations of `Clock-Update`, the value of the clock advances from one integral value to the next, (ii) the clock remains in each integral value for at least αn invocations. The `Read-Clock` procedure, which takes $O(\log n)$ operations, allows processors to get the current value of the clock, if it exists. Regardless, `Read-Clock` never returns a value greater than that in the clock. As we shall see later, by interleaving clock updates among the rest of the system's work, we get a measure of the total work performed in the system.

Outline. The main consensus procedure is presented in Section 3. The procedure necessitates $O(\log n)$ common *weak random coins* (defined later). In Section 2 we show how processors collectively generate these weak random coins. Combining the procedures of Sections 2 and 3 yields a randomized consensus algorithm in the Monte-Carlo sense (i.e. with high probability it is successful, but may also fail). In Section 4 we show how to convert the algorithm to one with a Las Vegas guarantee. The conversion is based on a novel construct we introduce, which we call the *Write-Once Register (WOR)*.

2 Weak Random Coins

A weak random coin has at least a constant probability of falling on heads and at least a constant probability of falling on tails, but it can also have an undefined value. In this section we show how the processors collectively flip $O(\log n)$ coins, half of which are guaranteed to be weak random coins.

Set $d = \lceil \log(n/\log n) \rceil$ and $N = 2^d$. We employ the following data structure. With each weak random coin r_i, we associate a fixed $N \times (d+1)$ array B_i. We view each array B_i as a butterfly structure of $d+1$ 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 linked to 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. We call the cells of level 0 the *inputs* and those of level d the *outputs*.

A pseudocode description of the algorithm is given hereunder. All cells $B_i[k][j]$ are assumed to be initialized to 0. The cells are (over)written one level at a time. A cell that is overwritten is said to be *filled*. First, the processors fill the inputs of all butterflies randomly with either 0 or 1, with $\Pr[\text{writing } 1] = 1/N$. Once most inputs are filled, subsequent levels of all butterflies are filled in sequence. To fill level k, active processors repeatedly choose at random a cell of the level and write to it the logical OR of its two predecessors. The Phase Clock of [7] is used to keep track of the level currently being filled. At the first level, p Processors read the clock after each write in the first level, and every $\log n$ writes, at subsequent levels. **ClockUpdate** is invoked following each round.

```
Weak-Random-Coin (for processor P_i)
   k ← Read-Clock                                          { O(log n) operations }
   if k > d then abort                                     { algorithm completed }
   if k = 0 then                                           { phase 0, fill inputs }
       choose i, j at random              { random butterfly B_i and cell B_i[0][j] }
       choose r ∈ {0, 1} at random with Pr[r = 1] = 1/N
       B_i[0][j] ← r
   else                                                    { phase k > 0 }
       for t = 1 to log n do
           choose i, j at random          { random butterfly B_i and cell B_i[k][j] }
           B_i[k][j] ← B_i[k − 1][j] OR B_i[k − 1][j^(k−1)]
   Clock-Update                                            { O(1) operations }
   goto line 1
```

Analysis. We denote by *phase* k, $0 \leq k \leq d$, the time interval when the clock has value k. A processor should fill level k in phase k.

Lemma 1. *W.h.p. the weak random coin algorithm completes in* $O(n \log^2 n)$ *total work.*

Proof. There are $O(\log n)$ levels and $O(n \log n)$ writes per level. Each write requires $O(1)$ work. □

Recall that there are $c \log n$ butterflies and that at the clock remains in each integral value for at least αn clock updates.

Lemma 2. *For any c there exists an α such that w.h.p. 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. During phase 0 the processors update the clock at least αn times. Since line 11 is reached at least αn times, line 4 is reached at least αn times. Level 0 contains cn cells, and each time a processor reaches line 4 it chooses a cell at random. Thus, by a version of the Chernoff bounds, for a sufficiently large α, w.h.p. $(c-1)n$ cells are chosen. After a processor chooses a cell, the adversary can stop the processor from advancing to line 5. However, there are only n processors. Thus, the adversary can only stop n chosen cells from being filled. Hence the total number of filled cells in the level is at least $(c-2)n$. The proof for other levels is similar, except that w.h.p. each cell is chosen $O(\log n)$ times. □

Consider an input-output pair of cells $(B_i[0][j], B_i[d][j'])$. Exactly one directed path $c_0, c_1, \ldots, c_{d-1}, c_d$ extends from input $B_i[0][j]$ to output $B_i[k][j']$.

Definition 3. Let $c_0, c_1, \ldots, c_d$ be an input-output path in B_i. We say that c_0 *successfully affects* c_d if c_0 is filled in phase 0 and each cell c_k is filled during phase k. We say an input cell c_0 is *strongly affecting* if it successfully affects at least 3/4 of the outputs of B_i. We say that B_i is *strongly input dependent* if at least half of its inputs are strongly affecting and there are at most $N/4$ writes to its inputs after the completion of phase 0.

The following lemma follows from the structure of the butterfly using a counting argument. The proof is omitted.

Lemma 4. *There exists a constant c such that w.h.p. at least $c \log n/2$ of the B_i's are strongly input dependent.*

Lemma 5. *Consider the state the system after the procedure completes. There is a constant p such for any $r \in \{0,1\}$ and any strongly input dependent B_i, $\Pr[3/4$ of the outputs B_i are $r] \geq p$.*

Proof. Consider the set of values stored in the inputs of B_i when phase 0 completes and the set of values of the late writes to the inputs of B_i. These are the only values that affect the outputs. Since B_i is strongly input dependent, these two sets contain at most $5n/4 \log n$ values. Each single value is 1 with probability $\log n/n$. Thus, there is a constant probability that none are 1. In this case all outputs of B_i are 0's. On the other hand, consider the set of strongly affecting inputs of B_i that are not written after phase 0 ends. There are at least $n/4 \log n$ such inputs. There is a probability p bounded from 0 by a constant, that the last value written to at least one of these cells is a 1. In this case at least 3/4 of the outputs get the value 1. Note that the dynamics of the execution are independent of the values appearing in the cells. Thus, to thwart the coin flip, the adversary must determine which cells are not filled independent of the input values. Thus, whether a cell is strongly affecting is independent of the values of these cells. □

Reading the Coin Flip Value. Processor P_i obtains the value of the coin flip r_i by sampling $O(\log n)$ of the output cells of B_i (level d). The majority of the sample is the coin flip value. By Lemma 5, if B_i is strongly input dependent then w.h.p. all samples of B_i give the same value, and this value constitutes a weak random coin flip.

3 The Consensus Procedure

We now explain how to obtain consensus using the weak random coin procedure of the previous section. Let C be an array of size γn, where γ is a constant to be determined later. All cells of C are assumed to be initialized to Λ. The pseudocode of the algorithm appears hereunder. The algorithm runs in *phases*, each consisting of $\Theta(n)$ write cycles. The phases are determined using the clock structure of [7]. During the first phase (phase 0), each processor P_i repeatedly picks a cell of C at random and copies its private input v_{P_i} into this cell. By the end of the phase most cells of C contain values. Now, consider phase $k > 0$. First, processor P_i samples $O(\log n)$ locations of C. Let β be the fraction of 1's in the sample. Then, from β and r_k (the value of the k-th weak random coin), processor P_i determines the *value*, s, of the sample as follows:

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

Processor P_1 then writes s to a random location in C. This procedure is repeated for $c \log n = O(\log n)$ phases.

Consensus-Procedure (for processor P_i)
1 $k \leftarrow$ `Read-Clock`
2 **if** $k > c \log n$ **then** abort { algorithm completed }
3 **if** $k = 0$ **then** { first phase }
4 $s \leftarrow$ local input v_{P_i} { copy local input }
5 **else** { subsequent phases }
6 sample $b \log n$ locations of C. Let β be the fraction of 1's in the sample.
7 obtain value of weak random coin r_k
8 $s \leftarrow \left(\beta > \frac{1+r_k}{3}\right)$ { threshold decision }
9 choose $j \in [1, \ldots, \gamma n]$ at random
10 $C[j] \leftarrow s$ { write value in random cell of current level }
11 `Clock-Update`
12 **goto** line 1

Analysis.

Lemma 6. *W.h.p. the consensus procedure completes in $O(n \log^2 n)$ steps.*

Proof. The are $O(\log n)$ phases. Each contains $O(n)$ clock updates. There is $O(\log n)$ work between consecutive updates. □

Recall that the array is of size γn and the clock advances every αn updates.

Lemma 7. *For any γ there exists an α such that w.h.p. the following holds: for any k at least 0.9 of the cells of C are written during phase k.*

Proof. Line 11 is reached at least αn times during the phase. There are n processors. Thus, lines 3-10 are executed at least $(\alpha - 2)n$ times. □

For $1 \leq k \leq c \log n$, let M_k be the number of cells in C that store the value 1 when phase k starts. Set $m_k = M_k / \gamma n$.

Lemma 8. *There exist constants b and 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 w.h.p. all but at most n of the writes to C during phase k are with the value 1 (res. 0).*

Proof. Suppose $m_k \geq 1/2$ and $r_k = 0$. Let t_k be the starting time of phase k. At most n writes from phase k have samples (line 6) before t_k. We call these *late writes*, and say that they may produce any value. We prove that all writes having their entire sample within phase k output the value 1. The proof is by induction. Consider a single round of the protocol (i.e. execution of lines 1-10) that has its write (line 10) at time t. Assume by induction that, apart from the late writes, all writes in phase k occurring prior to t output the value 1. Then, at all times $t_k \leq t' < t$, the number of 1's in C is at least $M_k - n$. Hence, with γ sufficiently large, if $m_k \geq 1/2$, then *throughout* the interval $[t_k, t)$ the fraction of 1's in C is at least 0.4. Thus, w.h.p. 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 9. *W.h.p. there exists a value $v \in \{0, 1\}$ such that any time after the last phase completes, at least 0.8 of the cells in C store the value v.*

Proof. Consider phase k. Suppose $m_k \geq 1/2$. By Lemmas 5 and 4 there is a constant probability that $r_k = 0$. If $r_k = 0$ then by Lemma 8 w.h.p. all but n writes during the phase write the value 1. By Lemma 7 at least 0.9 of the cells of C are rewritten during the phase. Thus, for γ sufficiently large, the fraction of ones in C at the beginning of phase $k+1$ is at least 0.8. Hence, by induction, in all subsequent phases, w.h.p. all samples find the fraction of 1's to be at least 0.7 and thus output a 1 (regardless of the coin flip). Thus, the values in 0.8 of the cells of C remains 1. Similarly, if $m_k < 1/2$, there is a constant probability that $r_k = 1$. Note that since the adversary does not know the value r_k, m_k is independent of r_k. Thus, after $O(\log n)$ phases, w.h.p. only one value appears in 0.8 of the filled cells. □

Lemma 10. *Any value other than Λ appearing in C is one of the processors' original input value.*

Proof. In the first phase C is filled with input values. From then on, values are only copied. □

To obtain the consensus value, the processors either sample C, or else use the error-free protocol of the next section.

4 The Las Vegas Protocol

The algorithm described in the previous section has a (polynomially-small) probability of failure. In this section we show how to transform the algorithm into a Las Vegas algorithm having $O(n \log^2 n)$ expected work.

The Write-Once Register

The Las Vegas algorithm relies on a basic building block we call the *Write-Once Register (WOR)*. The WOR is a deterministic data structure that can only store a single value. If overwritten, the WOR obtains a "garbage" value. We show how to construct a WOR from three elementary read/write registers, assuming 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 a WOR register it attaches its own processor identification number (P-*id*) to the value it writes. The registers are assumed to be initialed to Λ. To write to a WOR, a processor writes the three registers in sequence, checking before each write that no other processor has attempted writing the same WOR. The WOR is said to *hold a value* iff all three registers constituting the WOR are identical. If the registers are not identical, the value of the WOR is defined to be "garbage".

```
Write-WOR(v, P_id)
 1   for i = 1 to 3
 2       for j = 1 to i − 1
 3           if w_j ≠ (v, id) then abort
 4       for j = i to i
 5           if w_j ≠ Λ then abort
 6       w_i ← (v, id)

Read-WOR()
 7   read w_1, w_2 and w_3 in order
 8   if w_1 = w_2 = w_3 then
 9       return w_1.value
10   else
11       return "garbage"
```

Lemma 11. *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. W.l.o.g. suppose processor P_1 wrote v_1, and P_2 wrote v_2. Also, w.l.o.g. assume that P_1 wrote 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 3, with $i = 3$, then $w_1 = (v_1, P_1)$. Therefore, P_2 must write value v_2 to w_1 after P_1 reaches line 3 with $i = 3$. Thus, when P_2 reaches line 4, with $i = 2$ P_1 has already written w_2. The test in line 5 must therefore fail. □

The Certified Agreement Procedure

We now present a procedure by which processors are either convinced that consensus is achieved or else are warned of a failure and can rerun the protocol. In Section 3 we showed that w.h.p. a consensus value appears in 80% of the γn cells of the array C. To obtain a *proof* that consensus is achieved, the processors collectively estimate the number of 1's and 0's in C. The procedure employs a butterfly structure W with WOR registers at the first and last levels. Intermediate levels are composed of regular registers. Each level of W has γn nodes and there are $\log \gamma n + 1$ levels. Processors fill the levels of W in sequence. For the first level, the input level, processors copy the values from C, substituting a (-1) for 0 and a 0 for Λ. 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. We employ the [7] clock structure to keep track of the level currently being filled, where we tune the clock so that at least $\alpha n \log n$ clock updates are required to advance the clock by one. A description of the algorithm appears hereunder. All reads and writes to the first and last levels are WOR procedures.

```
Estimate-Counting (for processor P_i)
1    k ← Read-Clock
2    if k > log γn then
3        abort                                   { algorithm completed }
4    choose j ∈ [1, ..., γn] at random
5    if k = 0 then                               { fill inputs }
6        if C[j] = Λ then                        { cell not filled }
7            W[0][j] ← 0                         { substitute 0 for Λ }
8        else                                    { C[j] has a value }
9            W[0][j] ← 2C[c log n][j] − 1        { substitute a −1 for 0 }
10   else                                        { subsequent levels }
11       W[k][j] ← W[k − 1][j] + W[k − 1][j^(k)] { sum of predecessors }
12   Clock-Update
13   goto line 1
```

Once the last level, the output level, is filled, the processors obtain the consensus value by searching for an output with an absolute value greater than $\frac{1}{2}\gamma n$. We call such an output a *strongly-dominated output.* The sign of the strongly-dominated output determines the consensus value. If the processor cannot find a strongly-dominated output it decides that a failure has occurred and reruns the algorithm (line 15). A psuedocode of the process appears in the next page. Note that it is possible that one processor detects a failure, while another exists with a value, v. In this case we guarantee that all processors rerunning the protocol enter with v as their private input value. The array V, initialed to Λ, and lines 5-6, 9-14, provide this guarantee.

Analysis

Due to lack of space, most proofs in this section are omitted.

Lemma 12. *W.h.p. at most* $1/6$*-th of the inputs and* $1/6$*-th of the outputs get the value "garbage".*

Lemma 13. *There exist constants* γ, α *such that w.h.p. 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 phase* k*.*

The proof is similar to the proof of Lemma 2. A *strongly-dominated output* is an output that has absolute value greater than $\gamma n/2$.

```
Obtaining-the-Consensus-Value (for processor P_i)
1   repeat                                                      { scan outputs }
2       choose j ∈ [1, ..., γn] at random.
3       w ← W[log γn][j]
4       if |w| > (1/2)γn then                          { strongly-dominated output }
5           V[i] ← 1/2 + w/(2|w|)          { write sign of w to candidate value array }
6           if w = W[log γn][j] then                    { W[log γn][j] unchanged }
7               exit with consensus value v = 1/2 + w/(2|w|)
8   until all outputs are sampled
9   for j = 1 to γn do               { this line is reached only in case of a failure }
10      overwrite W[log γn][j] with "garbage"[3]                { erase all outputs }
11  s ← v_{P_i}                                                { use private input }
12  for j = 1 to n do                                           { scan the array V }
13      if V[j] ≠ Λ then                                      { a candidate exists }
14          s ← V[j]                                         { use candidate value }
15  Start a new agreement protocol using s as the private input value
```

Lemma 14. *There exist constants γ (of the array size) and α (of advancing the clock), such that if* 80% *of the cells of C have the value* 1 *(res.* 0*) then w.h.p. at least half of the outputs are strongly dominated and hold a positive (res. negative) value.*

The next lemma follows from Lemma 10.

Lemma 15. *If all private inputs to the consensus procedure are* 1 *(res.* 0*) then any strongly-dominated output is positive (res. negative).*

Corollary 16. *All processes successfully terminating at line 7 exit the same consensus v. This value is one of the input values.*

Corollary 16 completes the proof that w.h.p. in $O(n \log^2 n)$ steps consensus is achieved. We now consider the low-probability event of failure and prove that consensus is always obtained, and in expected $O(n \log^2 n)$ steps. We call the algorithm executed in line 15, the *alternate consensus execution.*

Lemma 17. *If any processor exits the procedure with a consensus value v in line 7, then any processor participating in the alternate execution enters with the value v.*

Proof. Suppose P_1 exits with the value 1 and P_2 participates in the alternate execution. Let $w = W[\log \gamma n][j]$ be the strongly-dominated output found by P_1. In lines 9–10 processor P_2 erases all outputs. Thus, P_1 must have read w for the second time before P_2 reached line 10. By this time, P_1 has already written the value 1 in $V[1]$. By Lemma 15 no other value can be written to any other entry in V. Thus, at line 13, P_2 finds the value 1 in V, and uses it as its input value for the alternate execution (line 15).

The following lemma follows from Lemma 15 and line 14.

Lemma 18. *Any value used as an input value to the alternate execution is one of the original input value.*

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

We thus obtain:

Theorem 19. *Using the above algorithm w.h.p. consensus is obtained in $O(n\log^2 n)$ total number of steps. The algorithm never fails, and $O(n\log^2 n)$ steps is also the expected number of steps.*

References

1. K. Abrahamson. On achieving consensus using shared memory. In *Proceedings of the 7th Annual ACM Symposium on the Principles of Distributed Computing*, pages 291–302, 1988.
2. J. Aspnes. Time- and space-efficient randomized consensus. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 325–331, 1990.
3. J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, September 1990.
4. J. Aspnes and O. Waarts. Randomized consensus in expected $O(n\log^2 n)$ operations per processor. In *Proceedings of the 33rd Annual Symposium on the Foundations of Computer Science*, pages 137–146, 1992.
5. H. Attiya, D. Dolev, and N. Shavit. Bounded polynomial randomized consensus. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, pages 281–294, 1989.
6. H. Attiya, N. Lynch, and N. Shavit. Are Wait-Free Algorithms Fast? In *31st FOCS*, pages55–64, 1990.
7. Y. Aumann and M.O. Rabin. Clock construction in fully asynchronous parallel systems and pram simulation. *Theoretical Computer Science*, 128:3–30, 1994.
8. 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.* Springer-Verlag, 1991.
9. T.D. Chandra. Polylog Randomized Wait-Free Consensus. To appear in *Proceedings of the 15th PODC*, 1996.
10. B. Chor, A. Israeli, and L. Ming. On processor coordination using asynchronous hardware. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.
11. 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.
12. M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of ACM*, 32(2):374–382, April 1985.
13. M. Herlihy. Impossibility results for asynchronous PRAM. In *Proceedings of the 3rd ACM Symposium on the Parallel Architectures and Algorithms*, pages 327–336, 1991.
14. M. Herlihy. Wait–free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
15. M. G. Loui and H. Abu-Amara. Memory Requirements for Agreement Among Unreliable Asynchronous Processors. *Advances in Computing Research*, vol. 4, pages 163–183, 1987.
16. C. Martel, R. Subramonian, and A. Park. Asynchronous PRAMs are (almost) as Good as Synchronous PRAMs. In Proceeding of 31st FOCS, pages 590–599, 1990.
17. S. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, pages 159–176, 1989.
18. M.O. Rabin. N-Process mutual exclusion with bounded waiting by $4\log_2 N$-valued shared variable. *Jour. Comp. Sys. Sc.*, 25:66–75, 1982.
19. M. Saks, N. Shavit, and H. Woll. Optimal Time Randomized Consensus - Making Resilient Algorithms Fast in Practice. In Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, pages 351–362, 1991.