Efficient Execution of Nondeterministic Parallel Programs on Asynchronous Systems*

Yonatan Aumann† Michael A. Bender‡ Lisa Zhang§

Abstract

We consider the problem of asynchronous execution of parallel programs. We assume that the original program is designed for a synchronous system, whereas the actual system may be asynchronous. We seek an *automatic execution scheme*, which allows the asynchronous system to execute the synchronous program. Previous execution schemes provide solutions only for the case where the original program is deterministic. Here, we provide the first solution for the more general case where the original program can be nondeterministic (e.g. randomized). Our scheme is based on a novel agreement protocol for the asynchronous parallel setting. Our protocol allows n asynchronous processors to agree on n *word-sized* values in $O(n \log n \log\log n)$ *total work*. Total work is defined to be the summation of the number of steps performed by *all* processors (including steps from busy waiting).

1 Introduction

Motivation. Parallel programs are frequently designed assuming tightly-coupled processors, operating in synchrony. A typical example of such a programming model is the PRAM model, in which all processors are assumed to operate in lock-step on the individual instruction level [16]. In less extreme models synchronization is often not assumed at every step, but it is still an indispensable ingredient of the overall structure (e.g. the BSP model [26]). Synchronization assumptions are convenient, because they free the programmer from the need to consider actual processor and network timings and let the programmer focus on the major task of parallelizing the program. However, these assumptions do not correspond to the way many actual parallel systems operate. Typically, in a multitasking parallel system –

*An initial version of this paper was presented in SPAA 1996.

†Department of Mathematics and Computer Science, Bar-Ilan University, Ramat-Gan, Israel. Email: aumann@bimacs.cs.biu.ac.il. This work was done while the author was at MIT Laboratory of Computer Science, supported in part by a Wolfson postdoctoral fellowship, and Darpa contract N00014-92-J-1799.

‡Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138. Supported by NSF contract CCR-9313775. Email: bender@das.harvard.edu.

§Department of Mathematics and Laboratory for Computer Science, MIT, Cambridge, MA 02139. Supported by an NSF graduate fellowship, ARMY grant DAAH04-95-1-0607 and ARPA contract N00014-95-1-1246. Email: ylz@math.mit.edu.

and especially in a network environment – processors operate on separate parts of the same application asynchronously and at considerably different speeds. For example, a heavily-loaded processor may dedicate considerably less CPU time to a given application than a lightly-loaded processor. In this case, the *application* running on the system would experience asynchronous behavior of the processors, even if the clocks *are* synchronized. Other sources for asynchrony include interrupts, context switches, network congestion, and page faults.

Execution Schemes. Faced with a gap between the idealized synchronous models, which facilitate program design, and reality, which dictates asynchrony, an *execution scheme* [24, 20] is the necessary bridge. (An execution scheme is also referred to as a method for *automatic program transformation* [20].) The execution scheme allows the asynchronous system to run programs written for the idealized synchronous model. Roughly speaking, the asynchronous system *emulates* the operation of the synchronous system. Thus, the programmer can write programs assuming the idealized synchronous model and run the programs on the asynchronous system. Designing efficient execution schemes is the focus of much previous work [24, 20, 8, 7]. However, all previous schemes are restricted to the deterministic case. If the original program is nondeterministic, e.g. randomized, the execution scheme fails. In this paper we provide the first efficient solution for the execution of *nondeterministic* parallel programs in the asynchronous setting. The solution provides a scheme that works regardless of the source of nondeterminism (e.g. randomization, local operating system decisions, etc.). Our solution is based on a novel agreement protocol for the asynchronous parallel setting. The agreement protocol allows the n asynchronous processors to agree on n *word-sized* values in a total of only $O(n \log n \log\log n)$ word operations. Previous asynchronous consensus protocols generally require $\tilde{\Omega}(n^2)$ operations per consensus on a single bit, and therefore are too slow to be useful in this context.

The Model. For concreteness and simplicity we describe a solution for fine-grained parallel programs. The same techniques also provide a solution for other cases (e.g. large-grained programs [7]). We consider an n-thread EREW PRAM program $\mathcal{P}$ written assuming a synchronous PRAM machine. The asynchronous *host system* H consists of n processors having a shared memory space and individual sources of randomness. We postulate a global *word size* for the system, which we denote by ω. Each processor has a small (constant) number of internal registers. Each processor of the host system has a set of *atomic operations* which it may execute. We assume that each atomic operation is executed to completion without interruption. The atomic operations include,

1. reading a word from shared memory,
2. writing a word to shared memory,
3. executing a fixed set of *basic computations* (e.g. add, multiply, etc.) on words in local registers. We assume that the set of basic computations allows any processor to perform any single computing instruction of the PRAM program $\mathcal{P}$ in a single step.

(I.e. if the program $\mathcal{P}$ includes the instruction $x \leftarrow f(y,z)$, then computing f is a basic computation.)

We assume that in a single atomic operation the host system can read or write a full word of the PRAM program together with an appropriate timestamp. (Timestamps are of size $O(\log n)$.) We emphasize that no atomic operations can both read from *and* write to shared memory. Thus, no compound operation such as *test & set* or *compare & swap* is atomic.

The processors act asynchronously. Formally, with each processor P_i we associate a *schedule function* $S_i : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{\infty\}$, which reflects the actual time instances in which the steps of P_i are performed. The expression $S_i(k) = t$ means that the k-th operation of processor P_i is executed at actual time t. Thus, S_i is monotone. An ∞ value indicates a faulty processor. The *total schedule* is the n-tuple $S = (S_1, \ldots, S_n)$. For simplicity we assume that all simultaneous reads succeed and that among the simultaneous writes to the same location an arbitrary one succeeds. Our results also hold for the model where all simultaneous accesses to the same location are rejected (see [26]). We postulate an *oblivious adversary scheduler*, which determines the schedule independently of the random choices of the processors. The adversary knows the program $\mathcal{P}$, the input values to the program, and the execution scheme. The adversary is not provided, however, with the dynamic random choices made by the processors during the execution. Complexity is measured by the total number of steps performed in the system, summed over all processors. Formally, an actual-time interval I is said to contain w *work units* if $w = \sum_i |\{k : S_i(k) \in I\}|$. A computation starting at t_0 and ending at t_1 is said to take w work if the interval $[t_0, t_1]$ contains w work units. Note that this formulation accounts for busy waiting and idling as well as effective work.

Our execution scheme is randomized and is successful with high probability. 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 $\Pr[E] \geq 1 - n^{-c}$.

Related Work. The problem of efficient parallel computation using unreliable and/or asynchronous processors is the focus of much previous work. Kanellakis and Shvartman [18, 19] introduce the *fail-stop* PRAM model and describe solutions to specific algorithmic problems in this model. Kedem, Palem, and Spirakis [22], and together with Raghunathan [21], show how to execute any deterministic PRAM program on a fail-stop PRAM (see also [19]). Gibbons [17] and Cole and Zajicek [13] introduce the Asynchronous PRAM (A-PRAM). Nishimura [25] shows how to execute specific computations using this model. Martel, Park, and Subramonian [24] provide a general execution scheme that allows any deterministic PRAM program to be executed on an A-PRAM, assuming *loose read&write atomicity*. In this scheme, executing a T-step PRAM program requires $O(T \log n)$ steps. Thus, the *overhead* for using the asynchronous system is $O(\log n)$. Kedem, Palem, Rabin, and Raghunathan [20] provide the first general solution for the case where only reads and writes are atomic. Their scheme entails an $O(\log^3 n)$ work overhead. Aumann and Rabin [8, 9] provide a solution with an $O(\log^2 n)$ overhead. Aumann, Kedem, Palem, and Rabin [7] consider large-grained programs, as opposed to the PRAM programs discussed in the previous works. For these programs they provide a solution with an $O(\log^* n)$ overhead. All of the above schemes are restricted to the execution of deterministic programs and fail if the original program is nondeterministic. The techniques we provide here allow any of the above schemes to operate in the nondeterministic case and have an $O(\log n \log\log n)$ *additive* overhead.

At the heart of our execution scheme is a new asynchronous agreement protocol designed for the A-PRAM model. The problem is closely related to, but distinct from, the classical asynchronous consensus problem. The classical problem assumes a dynamic adversary scheduler, which at any time determines the next processor to operate based on the entire history of the computation. The A-PRAM model, in contrast, assumes an oblivious adversary scheduler, where the entire schedule must be determined by the adversary in advance. However, the time bounds required for efficient program execution are much stricter than those provided by classical consensus protocols. In particular, the best protocols for the classical problem complete in $\tilde{O}(n)$ steps *per processor per bit*, yielding a total of $\tilde{O}(n^2)$ per bit. If employed in an execution scheme, these protocols would result in the unacceptable $\tilde{O}(n)$ overhead. Thus, a new scheme designed for the A-PRAM setting is needed.

We briefly describe the results for asynchronous consensus where the dynamic adversary scheduler is assumed. Fischer, Lynch, and Paterson [15] prove the impossibility of deterministic asynchronous consensus in the message-passing model, even if only one processor fails. Chor, Israeli, and Li [12] and Loui and Abu-Amara [23] show that the same result holds in the shared-memory model (also see [14]). A randomized polynomial-time solution is first presented in [12]; the result assumes a full-knowledge scheduler and an atomic *coin-flip&write* operation. Abrahamson [1] provides the first (exponential) solution for the standard model, which assumes a full-knowledge scheduler and atomic reads and writes. Aspnes and Herlihy [3] introduce the notion of the *weak random coin*. They give the first polynomial solution ($O(n^4)$), which uses unbounded registers. Attiya, Dolev, and Shavit [5] provide a polynomial, bounded-register solution. Aspnes [2] gives an $O(n^2(p^2+n))$ bounded-register solution, where p is the number of active processors. Bracha and Rachman [10] give an $O(n^2 \log n)$ unbounded-register solution. Finally, Aspnes and Waarts [4] provide a solution of $O(n \log n)$ steps *per processor*, which uses unbounded registers.

Recently, Aumann and Bender [6] and Chandra [11] defined intermediate-adversary models. These adversaries are less powerful than the dynamic adversary considered in the classic consensus work but more powerful than the oblivious adversary of the A-PRAM. Aumann and Bender [6] provide a consensus procedure that completes in $O(n \log^2 n)$ work per bit, assuming an intermediate adversary. Chandra [11] provides a consensus procedure that completes in expected $O(\log^2 n)$ work per processor per bit, also assuming an intermediate adversary (slightly different than that of [6]).

2 The Execution Scheme

In this section we present the overall structure of the execution scheme. The techniques provided in this paper allow any of the previously-known schemes to be converted from the deterministic to the nondeterministic setting. For concreteness we describe the result using the scheme presented in [9]. Extensions for other schemes are analogous.

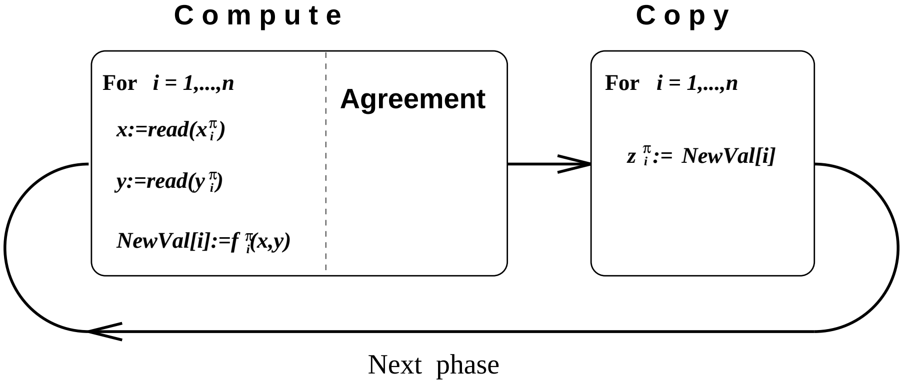

Figure 1: The execution scheme.

2.1 The Overall Structure

Consider an n-thread EREW PRAM program $\mathcal{P}$. The program is a sequence of *steps*. At each step π each thread T_i performs one *instruction* $z_i^{(\pi)} \leftarrow f_i^{(\pi)}(x_i^{(\pi)}, y_i^{(\pi)})$, where $x_i^{(\pi)}, y_i^{(\pi)}$, and $z_i^{(\pi)}$ are shared-memory variables, and $f_i^{(\pi)}$ is one of the program's basic operations (e.g. add, multiply). On the ideal PRAM all instructions of a single step are assumed to be performed simultaneously. In reality, the program $\mathcal{P}$ is executed on an n-processor asynchronous system. The execution is conducted in a sequence of *phases*. Each phase corresponds to a single step. A phase is composed of two subphases, COMPUTE and COPY. In the COMPUTE subphase, for each $i = 1, \ldots, n$, the values of $x_i^{(\pi)}$ and $y_i^{(\pi)}$ are read from the shared memory and the function $f_i^{(\pi)}$ is computed. Then the resulting value is stored in a temporary location NewVal[i] in shared memory. In the COPY subphase the value in NewVal[i] is copied to $z_i^{(\pi)}$. A schematic picture of the operation appears in Figure 1. (This split-execution procedure, introduced in [22], insures idempotence of operations.)

At any subphase there are n tasks to be performed. Processors of the asynchronous system repeatedly choose from these tasks at random, and execute a chosen task to completion. The [9] scheme guarantees that with high probability the system does not advance to the next subphase until all tasks of the current subphase are completed.

The Phase Clock. In the asynchronous system processors may go to sleep in one subphase and wake up much later. Thus, it is necessary to establish a method for these *tardy* processors to determine the current subphase. The *Phase-Clock* construction from [9] provides such a method. 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. `Read-Clock` takes $\Theta(\log n)$ operations, and `Update-Clock` takes $O(1)$ operations. The value of the clock is initialized to 0. Every $\Theta(n)$ invocations of `Update-Clock` the value of the clock advances from one integral value to the next. Specifically, for any $\alpha_1 > 0$ there exists an $\alpha_2 \geq \alpha_1$ such that the following holds. For all n there is a Phase Clock such that at least $\alpha_1 n$ invocations of `Update-Clock` are necessary and $\alpha_2 n$ are sufficient to advance the clock from one integral value to the next (regardless of which processors invoke the procedure).

The Phase Clock serves a double function. Firstly, it assists processors in identifying the current phase and subphase. Secondly, the Phase Clock guarantees that all tasks of the subphase are completed before the computation advances to the next subphase. This is achieved by interleaving clock updates with task execution. With a proper choice of the constants α_1 and α_2, it is guaranteed that during each subphase $\Theta(n \log n)$ tasks are chosen at random. The total number of distinct tasks in a subphase is n. Thus, with high probability, every task is performed at least once. For details see [9].

2.2 The Agreement Problem

Consider again a single COMPUTE subphase in the [9] scheme. The scheme guarantees that each task is executed at least once. However, tasks may also be performed more than once. In fact, in an asynchronous system this redundancy is unavoidable. If the program $\mathcal{P}$ is deterministic, multiple executions of the same task pose no problem; performing the same deterministic operation several times yields the same result. However, if the functions $f_i^{(\pi)}$ are nondeterministic, then the values written to NewVal[i] by different processors may be different. Thus, the processors need to reach an agreement on all values NewVal[1], ..., NewVal[n] before continuing to the COPY subphase. For this purpose we augment the COMPUTE subphase with an agreement protocol that establishes agreement on all new values before the COPY subphase begins. (See Figure 1.) Note that processors need to agree on n different values, each of which is a machine *word*. Agreement must be reached within the time bound of a single subphase, which is $\tilde{O}(n)$ operations. Such an agreement protocol is the main topic of this paper.

We formulate the problem in the following abstract terms. There are n asynchronous processors $P_1, \ldots, P_n$ performing a sequence of nonoverlapping *phases* separated by gaps. The phases correspond to the COMPUTE subphases of the execution scheme, and the gaps correspond to the COPY subphases. Associated with each phase π there are n nondeterministic functions $f_1^{(\pi)}, \ldots, f_n^{(\pi)}$. We seek a protocol that provides the following. For each π, upon completion of phase π and during the entire gap between phase π and phase $\pi+1$, there is a set of n values NewVal[1], ..., NewVal[n], agreed upon by and available to all processors; the values are such that for all i, NewVal[i] $\in f_i^{(\pi)}$. The Phase Clock determines the duration of the phases and the gaps.

3 The Agreement Protocol

The protocol employs a data structure that we call a *bin array.* The structure consists of an array of n *bins* corresponding to the n consensus values to be agreed upon. Each bin consists of $\beta \log n$ *cells* (β to be determined later). We denote the i-th bin by Bin_i and the j-th cell of this bin by $\mathrm{Bin}_i[j]$. The same bin array is used repeatedly in all phases of the execution scheme. Thus, since the processors are asynchronous, it is possible for a slow processor from one phase to overwrite values of a later phase. In order to distinguish between current and obsolete values, each write is *time stamped* with the current phase number. When a processor reads from a bin it only considers the values with the current time stamp. We call locations with current time stamps *filled* and locations with previous time stamps *empty*.

The protocol operates in *cycles*, which processors execute repeatedly. The cycles for all processors are identical. We show that after $O(n \log n)$ cycles, w.h.p. all n values are computed and agreed upon, regardless of the asynchronous schedule of the processors and of the identity of the processors performing the cycles. Thus, after $O(n \log n)$ cycles the phase is completed. Each processor reads the Phase Clock every $\log n$ cycles. The clock indicates the current phase and signals if the processor is working on an "old" phase.

```
Agreement Protocol, one cycle, phase π                                { n processors }
    choose i ∈ [1, ..., n] at random                                  { pick random Bin_i }
    perform binary search to find j such that
        Bin_i[j] is empty and Bin_i[j − 1] is filled, or j = 1
    if the search fails then          { implying all but clobbered cells in Bin_i are filled }
        abort cycle
    if Bin_i[j] is filled then            { by the end of the search, cell j may be filled }
        abort cycle
    else
        if j = 1 then                                                 { Bin_i is empty }
            Bin_i[1] ← f_i^(π)
        else
            Bin_i[j] ← Bin_i[j − 1]                                   { copy cell j − 1 to cell j }
```

Figure 2: One cycle of the agreement procedure.

Pseudocode for one cycle of the agreement procedure appears in Figure 2. The cycle starts with the processor P choosing a bin Bin_i at random. Throughout this entire cycle, processor P works on Bin_i only. Cells of the bin are written in increasing order. (An exception is when a tardy processor writes to a cell.) Processor P first uses binary search to find the first empty cell $\mathrm{Bin}_i[j]$ of the bin. If this is the first cell of the bin (i.e. $j = 1$) then P evaluates $f_i^{(\pi)}$ and writes the resulting value in $\mathrm{Bin}_i[1]$. Otherwise, P copies to the first empty cell the value appearing in the previous cell.

Work Per Cycle. Each cycle requires $O(\log \log n)$ steps. For the correctness of the protocol it is necessary that all cycles execute the exact same number of steps regardless of the random choices made by the processors. Thus, let $\omega = \alpha \log \log n$ be the maximum number of steps necessary to complete any one cycle. All processors spend ω steps on each cycle, executing no-ops if necessary.

Obtaining the Agreement Values. A processor obtains the i-th agreement value $\mathrm{NewVal}[i]$ by reading the cells in Bin_i between $\mathrm{Bin}_i[\beta \log n/2]$ and $\mathrm{Bin}_i[\beta \log n]$. Any value appearing in a filled cell in this range is a valid value for $\mathrm{NewVal}[i]$. In the next section we prove the following theorem.

Theorem 1 *For a sufficiently large β and for any given phase π, after $O(n \log n \log\log n)$ work units w.h.p. the following holds for each i:*

1. *Uniqueness: there exists a single value v_i, such that for all $j \geq (\beta \log n)/2$, if cell $\mathrm{Bin}_i[j]$ is filled then it stores the value v_i;*
2. *Stability: the value of v_i does not change (until the next phase begins);*
3. *Accessibility: half of the cells $\mathrm{Bin}_i[j]$, for $j \geq (\beta \log n)/2$, are filled;*
4. *Correctness: $v_i \in f_i^{(\pi)}$.*

4 Analysis

We now prove Theorem 1. First we bound the destructive effect tardy processors have on the current phase. For a given phase π, we say that a cell $\mathrm{Bin}_i[j]$ is *clobbered* if it is overwritten by a cycle associated with a previous phase.

Lemma 1 *For any given phase π w.h.p. there are at most $O(\log n)$ clobbers in each bin.*

Proof: Processors read the Phase Clock every $\log n$ cycles. A processor executing a cycle writes at most one cell (lines 9 and 11). Thus, any processor clobbers at most $\log n$ cells before it reads the clock. Therefore, during any given phase, each processor can clobber at most $\log n$ cells. Each clobber is to a randomly-chosen bin (line 1). Thus, in the given phase, there are a total of $O(n \log n)$ random clobbers on n bins. W.h.p. there are at most $O(\log n)$ clobbers per bin. ∎

4.1 Accessibility

Consider one cycle execution C. We denote the starting time of C by $S[C]$ and the finishing time of C by $F[C]$. (I.e. $S[C]$ is the time when C starts executing line 1, and $F[C]$ is the time when it finishes executing line 9 or 11.) Let $D[C]$ be the time when the cycle reaches line 5, i.e. after the binary search and before the write.

Consider a given phase π. Let $\omega = \Theta(\log\log n)$ be the amount of work units necessary to complete one cycle. Recall that ω is fixed for all cycles. We divide each phase into *stages* as follows. Let t_0 be the starting time of the phase. Assume t_k is defined. Then t_{k+1} is the first time after t_k such that the interval $[t_k, t_{k+1}]$ contains $3\omega n$ work units. *Stage k*, denoted Π_k, is the interval $[t_{k-1}, t_k]$. Let $S(\Pi_k)$ and $F(\Pi_k)$ denote the starting and finishing times of stage Π_k, respectively. We say that a cycle C is a *complete cycle in* Π_k if the entire execution of the cycle is performed within the stage. (By an abuse of language, we use the shorthand term "cycle" to mean "cycle execution".)

Lemma 2 *Each stage contains at least n and at most $3n$ complete cycles.*

Proof: Since there are n processors, at most n cycles can overlap the beginning of a stage, and at most n can overlap the end of a stage. ∎

Definition 1 *We say that stage* Π_k *is* effective *in* Bin_i *if during the stage*

1. *there exists a complete cycle in* Π_k *that operates on* Bin_i*,*
2. *there is no clobber in* Bin_i *during the stage.*

Define the *frontier* of Bin_i at any given time t to be the lowest indexed cell never written in the current phase. A cell j is a *hole* if it is empty but has an index smaller than the frontier. Since cells are written in increasing order, holes can only be formed by clobbers (i.e. writes by tardy processors operating for a previous phase). Note, however, that holes may prevent the binary search (line 2) from finding the true frontier of the bin.

Lemma 3 *After* $O(\log n)$ *effective stages in* Bin_i *all cells of the bin have been written in the current phase.*

Proof: Let f_k, ℓ_k, and h_k denote, respectively, the index of the frontier of Bin_i, the total number of clobbers to Bin_i, and the number of holes in Bin_i, at the finishing time of the k-th effective stage. Let f'_k, ℓ'_k, and h'_k be defined analogously for the starting time of the k-th effective stage. Then,

$$\begin{aligned} f_k &\leq f'_{k+1} \leq f_{k+1}, && (1)\\ h_{k+1} &\leq h'_{k+1} \leq h_k + (\ell_{k+1} - \ell_k). && (2) \end{aligned}$$

Inequality (1) is immediate. The left side of (2) follows from the definition of an effective phase. The right side holds since the $\ell_{k+1} - \ell_k$ new clobbers between the k-th and $(k+1)$-st effective stages can create at most $\ell_{k+1} - \ell_k$ holes.

We prove by induction that

$$\forall k,\ f_k \geq \min\{k - (\ell_k - h_k), \beta \log n\}. \quad (3)$$

If $f'_{k+1} = \beta \log n$, then

$$\begin{aligned} f_{k+1} &\geq f'_{k+1} = \beta \log n \\ &\geq \min\{(k+1) - (\ell_{k+1} - h_{k+1}), \beta \log n\}. \end{aligned}$$

Thus, we may assume $f'_{k+1} < \beta \log n$. There are two cases to consider:

1. One of the writes in the $(k+1)$-st effective stage pushes forward the frontier. Hence, from Inequality (1) and the inductive hypothesis

$$\begin{aligned} f_{k+1} &\geq f'_{k+1} + 1 \geq f_k + 1 \\ &\geq \min\{k - (\ell_k - h_k), \beta \log n\} + 1. \end{aligned}$$

However, $k - (\ell_k - h_k) \le f_k \le f'_{k+1} < \beta \log n$. Thus,

$$\begin{aligned} f_{k+1} &\ge k - (\ell_k - h_k) + 1 \\ &= k - (\ell_{k+1} - h_{k+1}) + (\ell_{k+1} - \ell_k) + (h_k - h_{k+1}) + 1. \end{aligned}$$

From Inequality (2) we conclude that

$$f_{k+1} \ge (k+1) - (\ell_{k+1} - h_{k+1}).$$

2. None of the writes in the $(k+1)$-st effective stage push forward the frontier. However, since the stage is effective, there must be at least one write to the bin during the stage. Thus, the write during the stage must be to a hole, therefore "filling" it. Hence, $h_{k+1} + 1 \le h'_{k+1}$. From Inequality (1) and the inductive hypothesis, and since $f'_{k+1} < \beta \log n$,

$$\begin{aligned} f_{k+1} &= f'_{k+1} \ge f_k \\ &\ge \min\{k - (\ell_k - h_k), \beta \log n\} \\ &= k - (\ell_k - h_k). \end{aligned}$$

By Inequality (2) we have

$$h_{k+1} + 1 \le h'_{k+1} \le h_k + (\ell_{k+1} - \ell_k).$$

Hence, we conclude

$$f_{k+1} \ge (k+1) - (\ell_{k+1} - h_{k+1}).$$

This completes the proof of Inequality (3).

From Lemma 1, w.h.p. there are at most $O(\log n)$ clobbers to Bin_i. Thus, $\ell_k \le O(\log n)$. Therefore, by Inequality (3), after $O(\log n) + \beta \log n = O(\log n)$ effective stages, the frontier is at $\mathrm{Bin}_i[\beta \log n]$, the last cell in the bin. Since cells are written in increasing order, the lemma follows. ∎

Lemma 4 *W.h.p. after $O(n \log n \log\log n)$ work units and for each i, half of the cells $\mathrm{Bin}_i[j]$ with $j \ge (\beta \log n)/2$ are filled.*

Proof: Each cycle requires $\Theta(\log\log n)$ work units. Thus, $O(n \log n \log\log n)$ work units constitute $c_1 \log n = O(\log n)$ stages. In each stage there are at least n complete cycles. Each cycle randomly chooses the bin on which it operates. Thus, for any given stage Π_k and Bin_i,

$$p_1 = \Pr\left[\text{There is a complete cycle in } \Pi_k \text{ in } \mathrm{Bin}_i\right] \ge 1 - \left(1 - \frac{1}{n}\right)^n \ge 1 - e^{-1}.$$

By the Chernoff bound, among $c_1 \log n$ stages w.h.p. at least $c_2 \log n$ stages have complete cycles in Bin_i, where $c_2 \ge p_1 c_1/2$. By Lemma 1 w.h.p. there are at most $c_3 \log n = O(\log n)$ clobbers to Bin_i (with c_3 independent of c_1 and c_2). Thus, w.h.p., at least $(c_2 - c_3)\log n$ stages are effective in Bin_i. By Lemma 3 for $c_4 = (c_2 - c_3)$ sufficiently large, after $c_4 \log n$ stages *all* cells of the bin are written in the current phase.

At most $c_3 \log n$ cells of the bin are clobbered. Thus, choosing $\beta > 4c_3$, at least half of the cells $\mathrm{Bin}_i[j]$, with $j \ge (\beta \log n)/2$, store the new value. ∎

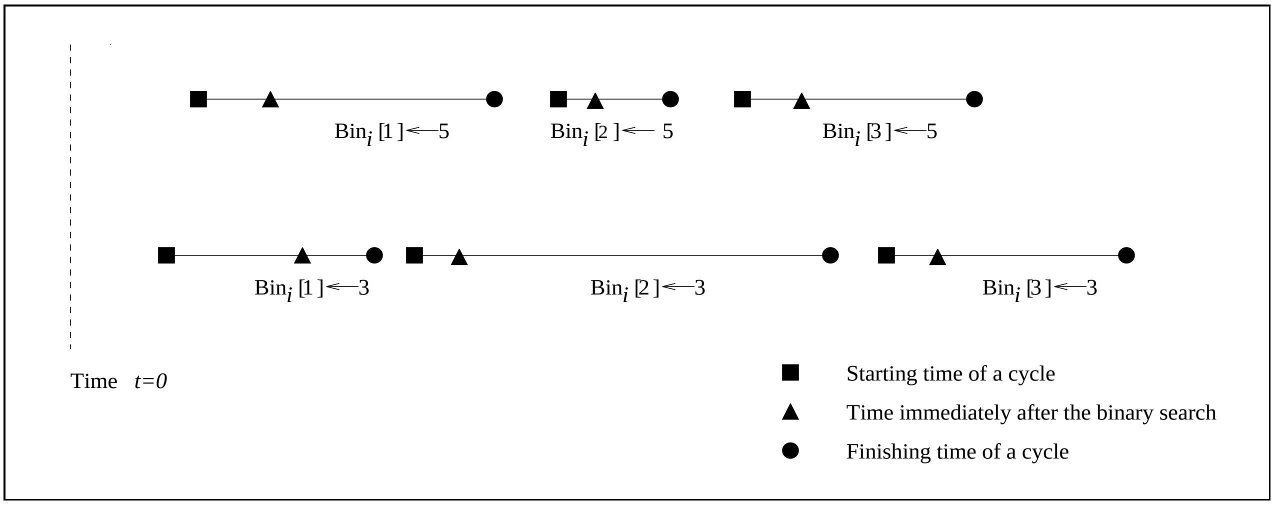

Figure 3: An arrangement of cycles in Bin_i that prevents Bin_i from converging. The values in Bin_i oscillates between 3 and 5. If this low-probability situation continues then Bin_i never converges.

4.2 Uniqueness and Stability

We say that a cell $\mathrm{Bin}_i[j]$ is *stable* (in a given phase) if whenever it is filled, it stores the same value. We say that Bin_i *reaches stability at cell* j if all cells $\mathrm{Bin}_i[j']$, $j' \geq j$ are stable. Figure 3 gives a low-probability situation where a bin does not stabilize. We show that w.h.p. all bins reach stability by the middle cell.

Definition 2 *A* stabilizing structure *in* Bin_i *is a pair of consecutive stages* (Π_{2k-1}, Π_{2k}) *such that*

1. *in each of the stages* Π_{2k-1} *and* Π_{2k} *there is exactly one complete cycle in* Bin_i*;*
2. *for all cycles* C *in* Bin_i*, if* $D[C] \in \Pi_{2k-1}$ *then* $F[C] \in \Pi_{2k-1}$*, and if* $D[C] \in \Pi_{2k}$ *then* $F[C] \in \Pi_{2k}$*.*

Figure 4 depicts a stabilizing structure.

Lemma 5 *Consider a stabilizing structure* (Π_{2k-1}, Π_{2k}) *on* Bin_i*. Let* $\mathrm{Bin}_i[j]$ *be the frontier at* $F[\Pi_{2k}]$*. Suppose that*

1. *there are no clobbers to* Bin_i *during* Π_{2k-1} *and* Π_{2k}*;*
2. *the complete cycles in stages* Π_{2k-1} *and* Π_{2k} *do not write to holes;*
3. $\mathrm{Bin}_i[j-1]$ *is not clobbered after* $F[\Pi_{2k}]$*.*

Then Bin_i *reaches stability at* $\mathrm{Bin}_i[j]$*,*

Proof: Since both Π_{2k-1} and Π_{2k} contain cycles that do not write to holes, each of these cycles advances the frontier. Thus, at $S[\Pi_{2k}]$ the frontier must be at most at $\mathrm{Bin}_i[j-1]$,

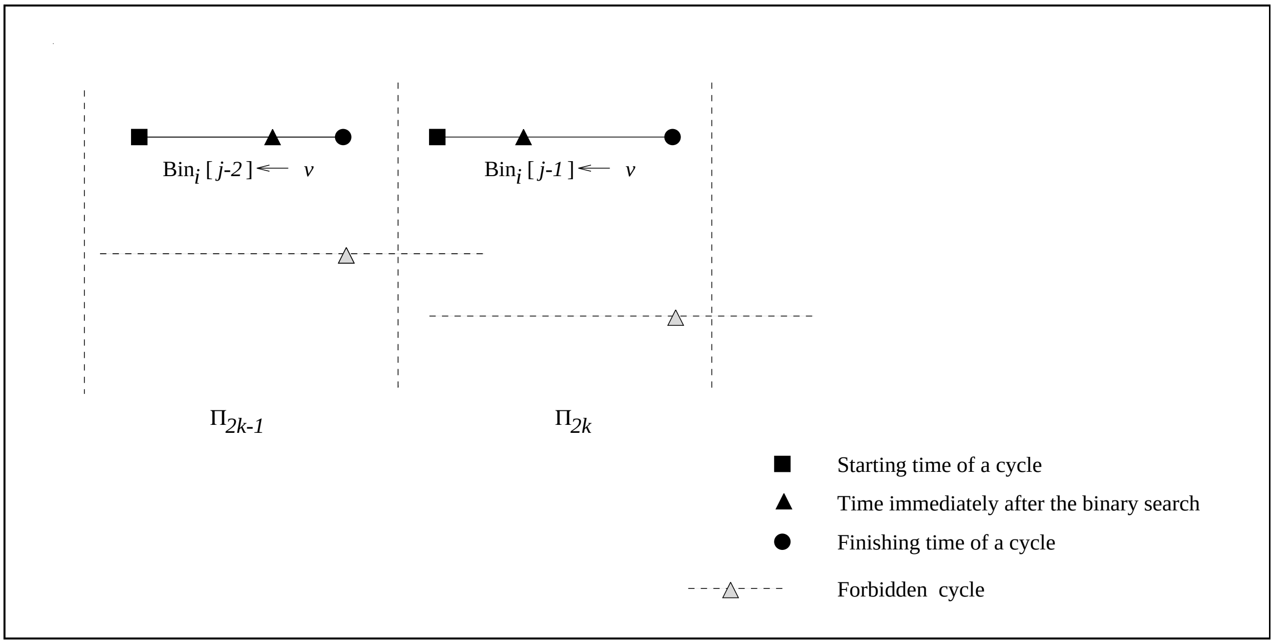

Figure 4: A stabilizing structure in Bin_i. Notice that in each of the phases there is exactly one complete cycle in Bin_i. Besides these two complete intervals there are no other intervals C such that $D[C] \in (\Pi_{2k-1}, \Pi_{2k})$. If the conditions of Lemma 5 hold, then Bin_i reaches stability by $\mathrm{Bin}_i[j]$.

and at $S[\Pi_{2k-1}]$ the frontier must be at most at $\mathrm{Bin}_i[j-2]$. Let v be the value in $\mathrm{Bin}_i[j-1]$ at $F[\Pi_{2k}]$. We show that following $F[\Pi_{2k}]$, v is the only value stored in filled cells $\mathrm{Bin}_i[j']$ for $j' \geq j-1$. Initially, at $F[\Pi_{2k}]$, all cells $j' > j-1$ are empty, and thus the claim holds. Consider a cycle writing after $F[\Pi_{2k}]$. By the definition of the stabilizing structure there are two possibilities for C.

1. $D[C] < S[\Pi_{2k-1}]$. In this case, during the entire binary search of C the frontier never moves beyond $\mathrm{Bin}_i[j-2]$. Hence, C writes to cell $\mathrm{Bin}_i[j'']$ with $j'' \leq j-2 < j'$.

2. $D[C] > F[\Pi_{2k}]$. We prove by induction on $j' \geq j-1$, that all (nonclobber) writes to cell j' are with the value v. For $j' = j-1$, $\mathrm{Bin}_i[j-1]$ is never clobbered after it is written in Π_{2k-1} and hence never written after $F[\Pi_{2k}]$. Thus, the value v remains in $\mathrm{Bin}_i[j-1]$. Assume by induction that the claim is true for all j'', $j-1 \leq j'' < j'$, and that C writes to cell $\mathrm{Bin}_i[j']$. Then by the inductive hypothesis, v is the value of $\mathrm{Bin}_i[j'-1]$, and is the value copied to $\mathrm{Bin}_i[j']$.

∎

Lemma 6 *There exists a constant p such that for any k and i, the probability that (Π_{2k-1}, Π_k) constitutes a stabilizing structure on Bin_i is at least p, independent of all other k and i.*

Proof: If stage Π_{2k} contains m complete cycles, then the probability that exactly one cycle is operating on Bin_i is

$$m\left(\frac{1}{n}\right)\left(1-\frac{1}{n}\right)^{m-1}.$$

By Lemma 2, the probability that condition 1 of Definition 2 holds is at least

$$\left[n\left(\frac{1}{n}\right)\left(1-\frac{1}{n}\right)^{3n-1}\right]^2 \approx e^{-6}.$$

Since there are n processors, at most n cycles C do not satisfy condition 2 of Definition 2, i.e. $D[C] \in \Pi_{2k-1}$ and $F[C] \notin \Pi_{2k-1}$, or $D[C] \in \Pi_{2k}$ and $F[C] \notin \Pi_{2k}$. The probability that none of these cycles are in Bin_i is at least $\left(1-\frac{1}{n}\right)^{2n} \approx e^{-2}$. Thus $p > e^{-8}$. ∎

Lemma 7 *For sufficiently large β w.h.p. all bins reach stability by cell $(\beta \log n)/2$.*

Proof: Recall that ω is the amount of work per cycle and that each stage consists of $3n\omega$ work units. Consider Bin_i. For any β there exists a $\beta_1 \geq \beta$, such that after $\omega\beta_1 n \log n$ work units, w.h.p. there are at most $(\beta \log n)/2$ writes to Bin_i. Thus, after $(\beta_1 \log n)/3$ stages the frontier of Bin_i has not reached cell $(\beta \log n)/2$.

Let $\mathcal{S}$ be the set of stabilizing structures in the first $(\beta_1 \log n)/3$ stages. By Lemma 6 and Chernoff Bounds, there exists a β_2, increasing with β and independent of n, such that w.h.p. $|\mathcal{S}| \geq \beta_2 \log n$. We show that conditions 1-3 of Lemma 5 hold for at least one of the stabilizing structures in $\mathcal{S}$. From this, by Lemma 5, we get that Bin_i reaches stability before $\mathrm{Bin}_i[(\beta \log n)/2]$.

By Lemma 1, w.h.p. Bin_i contains at most $c \log n = O(\log n)$ clobbers, where c is independent of β. Thus,

1. At most $c \log n$ of the stabilizing structures in $\mathcal{S}$ contain a clobber.
2. At most $c \log n$ of the stabilizing structures in $\mathcal{S}$ contain complete cycles that write to holes. (Each clobber produces at most one hole, and once a cell is filled in a stage, no complete cycles from future stages write to it unless it is clobbered.)
3. For at most $c \log n$ of the stabilizing structures in $\mathcal{S}$ that do not write to holes, the frontier at the end of the structure is subsequently clobbered. This is because two structures not writing holes cannot have the same frontier.

For a sufficiently large β, we have $\beta_2 > 3c$. Thus, all three conditions of Lemma 5 hold. ∎

4.3 Correctness

For all i a value written to $\mathrm{Bin}_i[1]$ is $f_i^{(\pi)}$ as computed by some processor. Cell $\mathrm{Bin}_i[j+1]$ is copied from $\mathrm{Bin}_i[j]$. Thus, by induction any value v_i appearing in any cell is a valid value, i.e. $v_i \in f_i^{(\pi)}$.

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] Y. Aumann and M. A. Bender. Efficient asynchronous consensus with the value-oblivious adversary scheduler. In *Proceedings of the 23rd International Colloquium on Automata, Languages, and Programming*, 1996.

[7] Y. Aumann, K. Palem, Z. Kedem, and M. O. Rabin. Highly efficient asynchronous execution of large grained parallel programs. In *Proceedings of the 34th Annual Symposium on the Foundations of Computer Science*, pages 271–280, November 1993.

[8] Y. Aumann and M. O. Rabin. Clock construction in fully asynchronous parallel systems and PRAM simulation. In *Proceedings of the 33rd Annual Symposium on the Foundations of Computer Science*, pages 147–156, 1992.

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

[10] 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.

[11] T. D. Chandra. Polylog randomized wait-free consenus. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, 1996.

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

[13] R. Cole and O. Zajicek. The expected advantage of asynchrony. In *Proc. of the ACM Symposium on Parallel Architectures and Algorithms*, pages 85–94, 1989.

[14] 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.

[15] 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.

[16] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[17] P. B. Gibbons. A more practical PRAM model. In *Proc. of the 1st ACM Symposium on Parallel Architectures and Algorithms*, pages 158–168, June 1989.

[18] P. Kanellakis and A. Shvartsman. Efficient parallel algorithms can be made robust. In *Proceedings of the 8th Annual ACM Symposium on the Principles of Distributed Computing*, pages 211–221, 1989.

[19] P. Kanellakis and A. Shvartsman. Effecient parallel algorothms on restartable fail-stop processors. In *Proceedings of the 10th Annual ACM Symposium on the Principles of Distributed Computing*, pages 23–36, 1991.

[20] Z. M. Kedem, K. V. Palem, M. O. Rabin, and A. Raghunathan. Efficient program transformation for resilient parallel computation via randomization. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, May 1992.

[21] Z. M. Kedem, K. V. Palem, A. Raghunathan, and P. G. Spirakis. Combining tentative and definite executions for very fast dependable parallel computing. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 381–390, May 1991.

[22] Z. M. Kedem, K. V. Palem, and P. G. Spirakis. Efficient robust parallel computations. In *Proceedings of the 22rd Annual ACM Symposium on Theory of Computing*, pages 138–148, May 1990.

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

[24] 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*, pages 590–599, 1990.

[25] N. Nishimura. Asynchronous shared memory parallel computation. In *Proc. of the 2nd ACM Symposium on Parallel Architectures and Algorithms*, pages 76–84, 1990.

[26] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.