Lower Bounds for Shared-Memory Leader Election Under Bounded Write Contention

Dan Alistarh
IST Austria, Klosterneuburg, Austria

Rati Gelashvili
Novi Research, Menlo Park, CA, USA

Giorgi Nadiradze
IST Austria, Klosterneuburg, Austria

Abstract

This paper gives tight logarithmic lower bounds on the solo step complexity of leader election in an asynchronous shared-memory model with single-writer multi-reader (SWMR) registers, for both deterministic and randomized obstruction-free algorithms. The approach extends to lower bounds for deterministic and randomized obstruction-free algorithms using multi-writer registers under bounded write concurrency, showing a trade-off between the solo step complexity of a leader election algorithm, and the worst-case number of stalls incurred by a processor in an execution.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Lower Bounds, Leader Election, Shared-Memory

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.4

Funding *Dan Alistarh*: Supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 805223 ScaleML).
Giorgi Nadiradze: Supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 805223 ScaleML).

Acknowledgements The authors would like to thank the DISC anonymous reviewers for their useful feedback and comments.

1 Introduction

Leader election is a classic distributed coordination problem, in which a set of n processors must cooperate to decide on the choice of a single "leader" processor. Each processor must output either a *win* or *lose* decision, with the property that, in any execution, a single processor may return *win*, while all other processors have to return *lose*. Moreover, any processor returns *win* in *solo* executions, in which it does not observe any other processor.

Due to its fundamental nature, the time and space complexity of variants of this problem in the classic asynchronous shared-memory model has been the subject of significant research interest. Leader election and its linearizable variant called *test-and-set* are weaker than consensus, as processors can decide without knowing the leader's identifier. Test-and-set differs from leader election in that no processor may return *lose* before the eventual winner has joined the computation, and has consensus number two. It therefore cannot be implemented deterministically wait-free [19]. Tromp and Vitányi gave the first *randomized* algorithm for *two-processor* leader election [29], and Afek, Gafni, Tromp and Vitányi [1] generalized this approach to n processors, using the tournament tree idea of Peterson and Fischer [27].

Their algorithm builds a complete binary tree with n leaves; each processor starts at a leaf, and proceeds to compete in two-processor leader-election objects located at nodes, returning *lose* whenever it loses at such an object. The winner at the root returns *win*. Since

each two-processor object can be resolved in expected constant time, their algorithm has expected step complexity $O(\log n)$ against an adaptive adversary. Moreover, their algorithm only uses *single-reader multiple-writer (SWMR)* registers: throughout any execution, any register may only be written by a single processor, although it may be read by any processor.

Follow-up work on time upper bounds has extended these results to the adaptive setting, showing logarithmic expected step complexity in the number of participating processors k [4, 16]. Further, Giakkoupis and Woelfel [16] showed that, if the adversary is oblivious to the randomness used by the algorithm, $O(\log^\star k)$ step complexity is achievable, improving upon a previous sub-logarithmic upper bound by Alistarh and Aspnes [2]. Another related line of work has focused on the *space complexity* of this problem, which is now resolved. Specifically, it is known that $\Omega(\log n)$ distinct registers are *necessary* [28, 16], and a breakthrough result by Giakkoupis, Helmi, Higham, and Woelfel [15] provided the first asymptotically matching upper bound of $O(\log n)$, improving upon an $O(\sqrt{n})$ algorithm by the same authors [14].

The clear gap in the complexity landscape for this problem concerns time complexity lower bounds. Specifically, in the standard case of an adaptive adversary, the best known upper bound is the venerable tournament-tree algorithm we described above [1], which has $O(\log n)$ expected time complexity and uses SWMR registers. It is not known whether one can perform leader election in classic asynchronous shared-memory faster than a tournament.[1] Due to the simplicity of the problem, none of the classic lower bound approaches, e.g. [23, 24, 22], apply, and resolving the time complexity of shared-memory leader election is known to be a challenging open problem [2, 16]. Moreover, given that the step complexities of shared-memory consensus [8] and renaming [3] have been resolved, leader election remains one of the last basic objects for which no tight complexity bounds are known.

We show tight logarithmic lower bounds on the step complexity of leader election in asynchronous shared-memory with SWMR registers. Our motivating result is a natural potential argument showing that any *deterministic* obstruction-free algorithm for leader election – in which processors must return if they execute enough solo steps – must have worst-case step complexity $\Omega(\log n)$ *in solo executions*, that is, even if processors execute in the absence of concurrency, as long as registers are SWMR.

Our main contribution is a new and non-trivial technique showing that a similar statement holds for *randomized* algorithms: in the same model, any *obstruction-free* algorithm for leader election has worst-case expected cost $\Omega(\log n)$. In this case as well, the lower bound holds in terms of expected step complexity *in solo executions*. The lower bound technique is based on characterizing the expected length of solo executions by analyzing the number of reads and writes over distinct registers required by a correct algorithm.

These are the first non-trivial lower bounds on the time complexity of classic shared-memory leader election, although they assume restrictions on the algorithms. They are both matched asymptotically by the tournament-tree approach, as the algorithm of [1] can be modified to be deterministic obstruction-free, by using two-processor obstruction-free leader election objects. This essentially shows that the tournament strategy is optimal for SWMR registers. The results will also apply to the case where algorithms may employ stronger two-processor read-modify-write primitives, such as two-processor test-and-set operations instead of reads and exclusive writes. Interestingly, the result holds for a *weak* version of leader election, in which all processors may return *lose* if they are in a *contended* execution.

[1] Sub-logarithmic step complexity is achievable in other models, e.g. distributed and cache-coherent shared-memory [17] or message-passing [5].

The main limitation of the approach concerns the SWMR restriction on the registers used by the algorithm. We investigate relaxations of this, and show that, for deterministic algorithms, if κ is the maximum number of processors which might be poised to write to a register in any given execution, then any algorithm will have worst-case solo step complexity $\Omega((\log n)/\kappa)$. Conversely, assuming κ is super-constant with respect to n, any algorithm with $O((\log n)/\kappa)$ solo step complexity will have an execution in which $\Omega(\kappa)$ distinct processors are poised to write concurrently to the same register. Since this latter quantity is an asymptotic lower bound on the worst-case *stall complexity* at a processor [13],[2] this yields a logarithmic trade-off between the worst-case slow-down due to steps in a solo execution, and the worst-case slow-down at a processor due to high register contention, measured in stalls, for any deterministic algorithm.

We generalize this argument to the *randomized* case as well, to show that, for $\kappa \geq 2$, any algorithm ensuring that at most $\kappa - 1$ worst-case stalls at a processor must have expected step complexity $\Omega((\log n)/\kappa^2)$. In practical terms, our results show that any gain made due to decreased steps on the solo fast-path is paid for by an increase in the worst-case stall complexity at a processor incurred by any obstruction-free leader election algorithm.

Additional Related Work. The previous section already covered known time and space complexity results for the classic leader election problem in the standard asynchronous shared-memory model. This fundamental problem has also been considered in under related models and complexity metrics. Specifically, Golab, Hendler and Woelfel [17] have shown that leader election can be solved using *constant* remote memory references (RMRs) in the cache-coherent (CC) and distributed shared-memory (DSM) models. Their result circumvents our lower bounds due to differences in the model and in the complexity metrics. In the same model, Eghbali and Woelfel [12] have shown that *abortable* leader election requires $\Omega(\log n/\log\log n)$ time in the worst case. The abortability constraint imposes stronger semantics, and they consider a different notion of complexity cost, but multi-writer registers.

In addition, our results are also related to early work by Anderson and Yang [30], who assume bounds on the write contention at each register, and prove $\Omega(\log n)$ lower bounds for a weak version of mutual exclusion, assuming bounded write contention per register. Upon careful consideration, one can obtain that their approach can be used to prove a similar logarithmic lower bound for *obstruction-free leader election* in the read-write model with contention constraints. However, we emphasize that their argument works only for *deterministic algorithms*.

Specifically, relative to this paper, our contribution is the randomized lower bound. The argument of Anderson and Yang [30] does not generalize to randomized algorithms, for the same reason that the simple deterministic argument we provide as motivation does not generalize in the randomized case. Even focusing on the deterministic case, our approach is slightly different than the one of Anderson and Kim: we use covering plus a potential argument, while they use a different covering argument based on eliminating contending processors by leveraging Turan's theorem. However, their approach can provide a better dependency on contention in the bound: $\Omega(\log n/\log\kappa)$ versus $\Omega(\log n/\kappa)$ in our case.

We note that similar trade-offs between contention and step complexity have been studied by Dwork, Herlihy and Waarts [11], and by Hendler and Shavit [18], although in the context of different objects, and for slightly different notions of cost. We believe this paper is the first to approach such questions for randomized algorithms, and for leader election.

[2] If $\kappa \geq 2$ processors are poised to write concurrently to a register, then the last processor to write will incur $\kappa - 1$ stalls.

From the technical perspective, the simple deterministic argument we propose can be viewed as a *covering argument* [23, 10, 9, 24, 7], customized for the leader-election problem, and leveraging the SWMR property. The new observation is the potential argument showing that some processor must incur $\Omega(\log n)$ distinct steps in a solo execution. To our knowledge, the lower bound approach for randomized algorithms is new. The generalized argument for bounded concurrent-write contention implies bounds in terms of the *stall metric* of Ellen, Hendler and Shavit [13], which has also been employed by other work on lower bounds, e.g. [7]. These prior approaches do not apply to leader election.

2 Model, Preliminaries, and Problem Statement

We assume the asynchronous shared-memory model, in which n processors may participate in an execution, $t < n$ of which may fail by crashing. Processors are equipped with unique identifiers, which they may use during the computation. For simplicity, we will directly use the corresponding indices, e.g. i, j, to identify processors in the following, and denote the set of all processors by $\mathcal{P}$. Unless otherwise stated, we assume that processors communicate via atomic read and write operations applied to a finite set of registers. The scheduling of processor steps is controlled by a strong (adaptive) adversary, which can observe the structure of the algorithm and the full state of processors, including their random coin flips, before deciding on the scheduling.

As stated, our approach assumes that the number of processors which may be poised to write to any register during the execution is deterministically bounded. Specifically, for an integer parameter $\kappa \geq 1$, we assume algorithms ensure κ-*concurrent write contention*: in any execution of the algorithm, at most κ processors may be concurrently poised to write to any given register. We note that, equivalently, we could assume that the worst-case write-stall complexity of the algorithms is $\kappa - 1$, as having κ processors concurrently poised to write to a given register necessarily implies that the "last" processor scheduled to write incurs $\kappa - 1$ stalls, one for each of the other writes.

Notice that this assumption implies a (possibly random) mapping between each register and the set of processors which write to it in every execution. For $\kappa = 1$, we obtain a variant of the SWMR model, in which a single processor may write to a given register in an execution. Specifically, we emphasize that we allow this mapping between registers and writers to *change* between executions: different processors may write to the same register, but in different executions. This is a generalization of the classic SWMR property, which usually assumes that the processor-to-registers mapping is fixed across all executions.

Without loss of generality, we will assume that algorithms follow a fixed pattern, consisting of repetitions of the following sequence: 1) a shared read operation, possibly followed by local computation, including random coin flips, and 2) a shared write operation, again possibly followed by local computation and coin flips. Note that any algorithm can be re-written following this pattern, without changing its asymptotic step complexity: if necessary, one can insert dummy read and write operations to dedicated NULL registers.

We measure complexity in terms of processor steps: each shared-memory operation is counted as a step. Total step complexity will count the total number of processor steps in an execution, while individual step complexity, which is our focus, is the number of steps that any single processor may perform during any execution.

We now introduce some basics regarding terminology and notation for the analysis, following the approach of Attiya and Ellen [9]. We view the algorithm as specifying the set of possible states for each processor. At any point in time, for any processor, there exists a single

next step that the processor is poised to take, which may be either a shared-memory read or write step. Following the step, the processor changes state, based on its previous state, the response received from the shared step (e.g., the results of a read), and its local computation or coin flips. *Deterministic* protocols have the property that the processor state following a step is exclusively determined by the previous state and the result of the shared step, e.g. the value read. *Randomized* protocols have the property that the processor has multiple possible next steps, based on the results of local coin flips following the shared-memory step. Each of these possible next steps has a certain non-zero probability. As standard, we assume that the randomness provided to the algorithm is *finite-precision*, and so, the number of possible next steps at each point is *countable*.[3]

A *configuration* C of the algorithm is completely determined by the state of each processor, and by the contents of each register. We assume that initially all registers have some predetermined value, and thus the *initial* configuration is only determined by the input state (or value) of each processor. Two configurations C and C' are said to be *indistinguishable* to processor p if p has the same state in C and C', and all registers have the same contents in both configurations.

A processor p is said to be *poised* to perform step e, which could be a read or a write, in configuration C if e is the next step that p will perform given C. Given a valid configuration C and a valid next step e by p, we denote the configuration after e is performed by p as Ce. An *execution* E is simply a sequence of such valid steps by processors, starting at the initial configuration. Thus, a configuration is *reachable* if there exists an execution E resulting in C. In the following, we will pay particular attention to *solo* processor executions, that is, executions E in which only a single processor p takes steps.

Our progress requirement for algorithms will be *obstruction-freedom* [20], also known as *solo-termination* [23]. Specifically, an algorithm satisfies this condition if, from any reachable configuration C, any processor p must eventually return a decision in every p-solo extension of C, i.e. in every extension $C\alpha_p$ such that α_p only consists of steps by p.

In the following, we will prove lower bounds for the following simplified variant of the leader election problem.

▶ **Definition 1** (Weak Leader Election)**.** *In the Weak Leader Election problem, each participating processor starts with its own identifier as input, and must return either* win *or* lose*. The following must hold:*

1. *(Leader Uniqueness) In any execution, at most a single processor can return* win*.*
2. *(Solo Output) Any processor must return* win *in any execution in which it executes solo.*

We note that this variant does not fully specify return values in contended executions – in particular, under this definition, all processors may technically return *lose* if they are certain that they are not in a solo execution – and does not require linearizability [21], so it is weaker than test-and-set. Our results will apply to this weaker problem variant.

3 Lower Bound for Deterministic Algorithms

As a warm-up result, we provide a simple logarithmic lower bound for the solo step complexity of leader election with SWMR registers. Specifically, the rest of this section is dedicated to proving the following statement:

▶ **Theorem 2.** *Any deterministic leader election protocol in asynchronous shared-memory with SWMR registers has* $\Omega(\log n)$ *worst-case solo step complexity.*

[3] Our analysis would also work in the absence of this requirement. However, it appears to be standard, and it will simplify the presentation: it will allow us to sum, rather than integrate, over possible executions.

3.1 Adversarial Strategy

We will specify the lower bound algorithmically, as an iterative procedure that the adversary can follow to create a worst-case execution. More precisely, the adversarial strategy will proceed in steps $t \in \{0, 1, \ldots, n-1\}$ and will maintain two sets of processors at each step, the *available set* $\mathcal{V}_t$ and the *frozen set* $\mathcal{F}_t$. In addition, we maintain a prefix of the worst-case execution, which we denote by E_t.

Initially, all processors are in initial state, and placed in the pool of *available* processors $\mathcal{V}_0$, while the set of frozen processors $\mathcal{F}_0$ is empty, and the worst-case execution E_0 is empty as well. In addition, we will associate a *blame counter* $\beta_t(i)$ to each available processor i, initially 0. Intuitively, this represents the number of processors that were placed in the frozen set because of i.

In each step $t \geq 0$, we first identify the processor j whose blame count $\beta_t(j)$ is *minimal* among processors from the available set $\mathcal{V}_t$, breaking ties arbitrarily. We then execute the sequence of solo steps α_j of processor j, until we first encounter a *write step* w_j of j to some register r_j which is *read by some available processor* $k \in \mathcal{V}_t$ *in its solo execution*. Note that the step w_j itself is *not* added to the execution prefix E_t. Below, in Lemma 4, we will show that such a write step by j must necessarily exist: otherwise, we could run j until it returns *win*, without this fact being visible to any other processor in the available set.

Having identified this first write step w_j by j, we "freeze" processor j exactly before w_j, and place it in the frozen set at the next step, $\mathcal{F}_{t+1}$, removing it from $\mathcal{V}_{t+1}$. We then update the worst-case execution prefix to $E_{t+1} = E_t\alpha_j$. Finally, we increment the blame count by 1 for every processor $k \in \mathcal{V}_{t+1}$ with the property that k reads from r_j in its solo execution. At this point, step t is complete, and we can move on to step $t+1$. The process stops when there are no more available processors.

3.2 Analysis

We begin by noting the following invariants, maintained by the adversarial strategy:

▶ **Lemma 3.** *At the beginning of each step t, the adversarial strategy enforces the following:*
1. *All available processors $i \in \mathcal{V}_t$ are in their initial state;*
2. *The contents of all registers read by processors in $\mathcal{V}_t$ during their respective solo executions are the same as in the initial configuration.*

Proof. Both claims follow by the structure of the construction. The first claim follows since the only processor which executes in any step $t \geq 0$ is eliminated from $\mathcal{V}_t$. The second claim follows since, at every step $t \geq 0$, we freeze the corresponding processor j *before* it writes to any register read by any of the remaining processors in $\mathcal{V}_{t+1}$. ◀

Notice that this result practically ensures that the execution prefix generated up to every step t is indistinguishable from the initial configuration for processors in the available set $\mathcal{V}_t$. Next, we show that the strategy is well-defined, in the sense that the step processor w_j specified above must exist at each iteration of the strategy.

▶ **Lemma 4.** *Fix a step t and let j be the chosen processor of minimal blame count $\beta_t(j)$. Then there must exist a step w_j in the solo execution of j which writes to some register r_j which is read by some available processor $k \in \mathcal{V}_t$.*

Proof. We will begin by proving a slightly stronger statement, that is, for *any* processor $k \in \mathcal{V}_t$, there must exist a register r_j^k which is written by j in its solo execution and read by k in its solo execution. We will then choose r_j to be the *first* such register written to by j in its solo execution, and w_j to be the corresponding write step.

Assume for contradiction that there exists a processor $k \in \mathcal{V}_t$, $k \neq j$ which does not read from any registers written to by j in its solo execution. By Lemma 3, the current execution is indistinguishable from a solo execution for j. Thus, if j runs solo from the prefix E_t until completion, j must return *win*. However, if k runs solo after j returns, k also must return *win*, since it does not read from any register which j wrote to, and therefore, by Lemma 3, it observes a solo execution as well. This contradicts the *leader uniqueness* property in the resulting execution.

We have therefore established that *every* other processor $k \in \mathcal{V}_t$ must eventually read from a register r_j^k written to by j in its solo execution. (Notice that these registers need not be distinct with respect to processors.) Let w_j^k be the step where j first writes to r_j^k during its solo execution. To ensure the requirements of the adversarial strategy, it suffices to pick w_j to be the *first* such step w_j^k, in temporal order, in j's solo execution. ◀

We now return to the proof, and focus on the blame counts of available processors at any fixed step $t \geq 0$, $(\beta_t(i))_{i \in \mathcal{V}_t}$. Define the potential Γ_t to be $\sum_{i \in \mathcal{V}_t} 2^{\beta_t(i)}$ at time t.

Since $i \in \mathcal{V}_t$ and $\beta_0(i) = 0$, for all processors i, we have that $\Gamma_0 = n$. Next, we show that, due to the way in which we choose the next processor to be executed, we can always lower bound this potential by n.

▶ **Lemma 5.** *For any step $t \geq 0$, we have $\Gamma_t \geq n$.*

Proof. We will proceed by induction. The base step is outlined above. Fix therefore a step $0 \leq t < n - 1$ such that $\Gamma_t \geq n$.

Again, let $j \in \mathcal{V}_t$ be the processor we freeze at step t. For each $i \in \mathcal{V}_t \setminus \{j\}$, let g_i be the weight by which we incremented the blame count of processor i in this step. By Lemma 4. we have that there exist $i \in \mathcal{V}_t \setminus \{j\}$ such that $g_i = 1$. Further, since we chose to execute the processor j with minimal blame count, we have that $\beta_t(j) \leq \beta_t(i)$. Let us now analyze the difference

$$\Gamma_{t+1} - \Gamma_t = 2^{\beta_t(i)+g_i} - 2^{\beta_t(i)} - 2^{\beta_t(j)} = 2^{\beta_t(i)} - 2^{\beta_t(j)} \geq 0.$$

Hence, $\Gamma_{t+1} \geq \Gamma_t \geq n$, as required. ◀

To complete the proof of Theorem 2, let ℓ be the last remaining non-frozen processor before the process completes, i.e. $\mathcal{V}_{n-1} = \{\ell\}$. By Lemma 5, we have that $\Gamma_{n-1} = 2^{\beta_{n-1}(\ell)} \geq n$, which implies that $\beta_{n-1}(\ell) \geq \log_2 n$. Further, notice that processor ℓ must have performed at least $\beta_{n-1}(\ell)$ *distinct* read operations: for every increment of $\beta_{n-1}(\ell)$, there must exist a unique processor i which wrote to some register r_t^i from which ℓ reads in its solo execution. Since we are assuming SWMR registers, the reads performed by ℓ must be also *unique*. Hence, processor ℓ performs $\log_2 n$ steps in a solo execution, implying an $\Omega(\log n)$ solo step complexity lower bound for the algorithm.
This strongly suggests that the tournament-tree approach is optimal for SWMR registers.

3.3 Discussion

Bounded Concurrent-Write Contention and Stalls. Second, it is interesting to observe what happens to the above argument in the case of *multi-writer* registers. Let $\kappa \geq 1$ be the bound on the concurrent-write contention over any single register, in any execution, that is, on the maximum number of processors which may be concurrently poised to write to a register. Notice that the overall construction and the blaming mechanism would still work. Therefore, the potential lower bound still holds, but in the proof of the last step, the different steps taken by the last processor ℓ do not necessarily need to be distinct. Specifically, we

note that a single read step by ℓ may be counted at most κ times, once for each different processor which may be frozen upon its write to the corresponding register. The lower bound is therefore weakened linearly in κ.

▶ **Corollary 6.** *Any deterministic leader election protocol in asynchronous shared-memory where at most $\kappa \geq 1$ may be poised to write to a register concurrently has worst-case solo step complexity $(\log n)/\kappa$. Moreover, if the lower bound construction above implies $(\log n)/\kappa$ worst-case step complexity for a processor, then there must exist an execution in which the concurrent-write contention on some register is κ.*

Recall that, when interpreted in the stall model of [13], having $\kappa \geq 2$ processors poised to write to a register at the same time implies *(write-)stall complexity* $\kappa - 1$ for one of the processors. Thus, this last result implies a logarithmic multiplicative trade-off between the worst-case step complexity of a protocol and its worst-case stall complexity.

Stronger Primitives. Third, we note that this approach can also be extended to deterministic algorithms employing SWMR registers supporting *read*, and *write*, and additionally 2-processor *test-and-set* objects. We can then apply the same freezing strategy, and note that an access to a *test-and-set* object can only lead to freezing a processor and incrementing the blame counter of another processor just once (otherwise there is a combined execution with more than 2 processors accessing it). Hence, we still obtain a lower bound of $\log n$ solo step complexity, i.e. that the tournament tree is the optimal strategy.

4 Lower Bound for Randomized Algorithms

We now shift gears and present our main result, which is a logarithmic expected-time lower bound for randomized obstruction-free algorithms. Our approach in this case will be different, as we are unable to build an explicit worst-case adversarial strategy. Instead, we will argue about the expected length of executions by bounding the expected number of reads over distinct registers required for algorithms to be correct. In turn, this will require a careful analysis of the probability distribution over solo executions of a specific well-chosen structure.

We first focus on the SWMR case, and cover it exclusively in Sections 4.1 to 4.3. We will then provide a generalization to MWMR registers under bounded concurrency in Section 4.4.

4.1 Preliminaries

Fix a processor $p \in \mathcal{P}$. For each p, we define the set $S(p)$ as the set of all possible solo executions of p, and will focus on understanding the probability distribution over reads and writes for executions in $S(p)$. By the *solo output* property of the algorithm (Definition 1), all these executions have to be finite in length. For any possible solo execution e of processor $p \in \mathcal{P}$, $\Pr[e]$ will be used to denote the probability that if we let run p run solo, it will execute e and return. In particular, $\sum_{e \in S(p)} Pr[e] = 1$.

Let $\mathcal{R}$ denote the set of all registers which could be used by the algorithm over all *solo* executions by some processor. Since the randomness provided to the algorithm is finite-precision, the number of possible next steps in every configuration is countable and by the spiral argument, $\mathcal{R}$ must be countable as well[4]. Fix a register $r \in \mathcal{R}$; by definition, r is read or written by some processor during some solo execution. Let $\mathcal{A}(r)$ be a set of all solo executions which read from a register r:

[4] Our argument works even when R isn't countable, but this simplifies notation, e.g. discrete sums.

$$\mathcal{A}(r) = \Big\{e \big| \exists p \in \mathcal{P}[e \in S(p) \wedge \text{read}(r) \in e]\Big\}.$$

We define the *read potential* $\rho(r)$ to roughly count the sum of probabilities that a register is read from during solo executions by any processor. Formally,

$$\rho(r) = \sum_{e \in \mathcal{A}(r)} \Pr[e] = \sum_{p \in \mathcal{P}} \sum_{e_p \in S(p) \cap \mathcal{A}(r)} \Pr[e_p].$$

Analogously, let $\mathcal{B}(r)$ be as a set of all solo executions which write to a register r:

$$\mathcal{B}(r) = \Big\{e \big| \exists p \in \mathcal{P}[e \in S(p) \wedge \text{write}(r) \in e]\Big\}.$$

We define the *write potential* of a register r as

$$\gamma(r) = \sum_{e \in \mathcal{B}(r)} \Pr[e] = \sum_{p \in \mathcal{P}} \sum_{e_p \in S(p) \cap \mathcal{B}(r)} \Pr[e_p].$$

For the simplicity we assume that for any $r \in \mathcal{R}$, $\gamma(r) > 0$ (or alternatively $\mathcal{B}(r) \neq \emptyset$). Otherwise, the reads from r do not change the outcome of the solo executions, and we can assume that they do not use r.

Further, for any given solo execution $e \in S(p)$ of processor p, we define the *trace* of e, $\mathcal{TR}(e)$, as the sequence of registers written by p during e, in the order in which they were written, but omitting duplicate registers. For instance, if in execution e processor p wrote to u_1, then u_2 followed by u_1 again and finally u_3, the trace would be u_1, u_2, u_3 (notice that registers are sorted by the order they are written to for the first time in e). Also, for each register $r \in \mathcal{R}$ and solo execution $e \in \mathcal{B}(r)$, let $\xi_r(e)$ be the index of register r in the trace of e. That is, if $\mathcal{TR}(e) = u^e_1, u^e_2, \ldots, u^e_{|\mathcal{TR}(e)|}$, then $r = u^e_{\xi_r(e)}$.

Our lower bound relies heavily on double-counting techniques. To familiarize the reader with notation and provide some intuition, we isolate and prove the following simple properties of traces. We fix an execution e, and the corresponding notation, as defined above.

▶ **Lemma 7.** *Given the above notation, we have that* $\sum_{r \in \mathcal{TR}(e)} \rho(r) \geq n - 1$.

Proof. Fix a processor $p \in \mathcal{P}$. Recall that every processor $q \in P \setminus \{p\}$ has to read from some register which p writes to in its solo execution e. Otherwise, there is an interleaving of p's solo execution e, followed by q's execution, which neither p nor q can distinguish from their respective solo executions. Therefore, in this interleaved execution, p and q will both return *win*, which leads to a contradiction.

This means that, for every solo execution $e_q \in S(q)$, there exists a register $r \in \mathcal{TR}(e)$, such that $read(r) \in e_q$ and hence:

$$\sum_{r \in \mathcal{TR}(e)} \rho(r) \geq \sum_{r \in \mathcal{TR}(e)} \sum_{q \in P \setminus \{p\}} \sum_{e_q \in S(q) \cap \mathcal{A}(r)} \Pr[e_q] \geq \sum_{q \in P \setminus \{p\}} \sum_{e_q \in S(q)} \Pr[e_q] = n - 1. \qquad \blacktriangleleft$$

Before proving the next lemma, we provide an intuition from the deterministic setting. For each processor p, assume that there exists $e_p \in S(p)$ such that $\Pr[e_p] = 1$ and consider the sum $\sum_{p \in \mathcal{P}} \sum_{r \in \mathcal{TR}(e_p)} \frac{\rho(r)}{\gamma(r)}$. For each register r, we know that $\frac{\rho(r)}{\gamma(r)}$ appears $\Big|\{p | r \in \mathcal{TR}(e_p)\}\Big| = \Big|\{p | \text{write}(r) \in e_p\}\Big| = \gamma(r)$ times in this summation. Hence, $\sum_{p \in \mathcal{P}} \sum_{r \in \mathcal{TR}(e_p)} \frac{\rho(r)}{\gamma(r)} = \sum_{r \in \mathcal{R}} \frac{\rho(r)}{\gamma(r)} \gamma(r) = \sum_{r \in \mathcal{R}} \rho(r)$.

▶ **Lemma 8.**

$$\sum_{p\in\mathcal{P}}\sum_{e\in S(p)}\Pr[e]\sum_{r\in\mathcal{TR}(e)}\frac{\rho(r)}{\gamma(r)}=\rho(r). \tag{1}$$

Proof. We have that

$$\begin{aligned}\sum_{p\in\mathcal{P}}\sum_{e\in S(p)}\Pr[e]\sum_{r\in\mathcal{TR}(e)}\frac{\rho(r)}{\gamma(r)}&=\sum_{p\in\mathcal{P}}\sum_{e\in S(p)}\Pr[e]\sum_{r\in\mathcal{R}}\frac{\rho(r)}{\gamma(r)}\mathbb{1}_{r\in\mathcal{TR}(e)}\\&=\sum_{r\in\mathcal{R}}\frac{\rho(r)}{\gamma(r)}\sum_{p\in\mathcal{P}}\sum_{e\in S(p)}\Pr[e]\mathbb{1}_{r\in\mathcal{TR}(e)}=\sum_{r\in\mathcal{R}}\frac{\rho(r)}{\gamma(r)}\sum_{p\in\mathcal{P}}\sum_{e\in S(p)}\Pr[e]\mathbb{1}_{\mathrm{write}(r)\in e}\\&=\sum_{r\in\mathcal{R}}\frac{\rho(r)}{\gamma(r)}\sum_{e\in\mathcal{B}(r)}\Pr[e]=\sum_{r\in\mathcal{R}}\frac{\rho(r)}{\gamma(r)}\gamma(r)=\sum_{r\in\mathcal{R}}\rho(r).\end{aligned}$$

Where in the second equality we simply rearranged the terms. ◀

Finally, we will need the following useful property.

▶ **Lemma 9.** *For any sequence of positive real numbers $x_1, x_2, ...x_m$, we have that*

$$\sum_{i=1}^{m}\frac{x_i}{1+\sum_{j=1}^{i-1}x_j}\geq\ln\Big(1+\sum_{i=1}^{m}x_i\Big).$$

Proof. Notice that for any $i\geq 1$:

$$\frac{x_i}{1+\sum_{j=1}^{i-1}x_j}=\int_{1+\sum_{j=1}^{i-1}x_j}^{1+\sum_{j=1}^{i}x_j}\frac{1}{1+\sum_{j=1}^{i-1}x_j}\,dx\geq\int_{1+\sum_{j=1}^{i-1}x_j}^{1+\sum_{j=1}^{i}x_j}\frac{1}{x}\,dx.$$

Hence:

$$\begin{aligned}\sum_{i=1}^{m}\frac{x_i}{1+\sum_{j=1}^{i-1}x_j}&\geq\sum_{i=1}^{m}\int_{1+\sum_{j=1}^{i-1}x_j}^{1+\sum_{j=1}^{i}x_j}\frac{1}{x}\,dx=\int_{1}^{1+\sum_{j=1}^{m}x_j}\frac{1}{x}\,dx\\&=\ln\Big(1+\sum_{j=1}^{m}x_j\Big).\end{aligned}$$

◀

4.2 The "Carefully-Normalized" Read Potential Lemma

Our lower bound is based on the following key lemma, which intuitively provides a lower bound over the sum of the read potentials of registers written to in a solo execution e by processor p. Importantly, the read potentials are carefully normalized by, roughly, the probability that these registers are written to by other processors in some other executions.

▶ **Lemma 10.** *Let $e\in S(p)$ be a solo execution of processor p, and $\mathcal{TR}(e)$ be its trace. Then, $\sum_{r\in\mathcal{TR}(e)}\frac{\rho(r)}{1+\sum_{q\in P\setminus\{p\}}\sum_{e_q\in S(q)\cap\mathcal{B}(r)}\Pr[e_q]}\geq\ln n$.*

The rest of this sub-section will be dedicated to proving this lemma. Specifically, we prove the following two claims in the context of the theorem, i.e. for a fixed solo execution e of processor p. We expand $\mathcal{TR}(e)$ as $u_1^e, u_2^e, \ldots, u_{|\mathcal{TR}(e)|}^e$.

▷ Claim 11. $\sum_{i=1}^{|\mathcal{TR}(e)|}\frac{\rho(u_i^e)}{1+\sum_{j=1}^{i-1}\rho(u_j^e)}\geq\ln n$.

Proof. Applying Lemma 9 to positive real numbers $\rho(u_1^e), \rho(u_2^e), \ldots, \rho(u_{|\mathcal{TR}(e)|}^e)$, we get:

$$\sum_{i=1}^{|\mathcal{TR}(e)|} \frac{\rho(u_i^e)}{1+\sum_{j=1}^{i-1}\rho(u_j^e)} \geq \ln\left(1+\sum_{j=1}^{|\mathcal{TR}(e)|}\rho(u_j^e)\right).$$

By Lemma 7, $\sum_{j=1}^{|\mathcal{TR}(e)|}\rho(u_j^e) \geq n-1$, completing the proof. ◁

Next, we prove the following extension:

▷ Claim 12. Under the above notation, we have

$$\sum_{j=1}^{i-1}\rho(u_j^e) \geq \sum_{q\in P\setminus\{p\}}\sum_{e_q\in S(q)\cap\mathcal{B}(u_i^e)}\Pr[e_q].$$

Proof. Let us substitute the definition of ρ. We want to prove that:

$$\sum_{j=1}^{i-1}\sum_{q\in\mathcal{P}}\sum_{e_q\in S(q)\cap\mathcal{A}(u_j^e)}\Pr[e_q] \geq \sum_{q\in P\setminus\{p\}}\sum_{e_q\in S(q)\cap\mathcal{B}(u_i^e)}\Pr[e_q].$$

Both the left and right-hand sides of this expression contain the sum of probabilities of certain solo executions. On the right hand side, (the probability of) any execution e_q of processor $q \in P \setminus \{p\}$ can appear at most once. This is not necessarily true for the left hand side due to the outer summation. Therefore, we only need to show that for any e_q whose probability $\Pr[e_q]$ is included in the summation on the right hand side, $\Pr[e_q]$ is also included in the summation on the left hand side – in other words, there exists $1 \leq j \leq i-1$, such that $\text{read}(u_j^e) \in e_q$.

We prove this fact by contradiction. Suppose processor $q \in P \setminus \{p\}$ has a solo execution $e_q \in S(q)$ such that register u_i^e is written to during e_q, but no register among $u_1^e, \ldots u_{i-1}^e$ are read (which are all registers written prior to u_i^e in p's solo execution e). Now consider a combined execution of p and q, which consists of p running as in e until it becomes poised to write register u_i^e – crucially, please note that so far p has actually executed solo. From this point, we consider processor q executing identically as it runs solo in execution e_q. This is possible because the only registers written to so far the system are $u_1^e, \ldots, u_{i-1}^e$, which q does not read in e_q. As a result, q will write to register u_i^e, after which we can immediately allow p to also write to u_i^e. This implies that two processors write to the same register during the same execution, and contradicting the SWMR property. ◁

Then, Lemma 10 follows by combining Claim 11, Claim 12 and the definition of trace.

4.3 Completing the Lower Bound Proof

We now finally proceed to proving the following theorem:

▶ **Theorem 13.** *Any randomized leader election protocol in asynchronous shared-memory with SWMR registers has* $\ln n$ *worst-case expected solo step complexity.*

Proof. We start by summing up inequalities given by Lemma 10 for all processors, and their solo executions:

$$n \ln n \leq \sum_{p \in \mathcal{P}} \sum_{e \in S(p)} \Pr[e] \left(\sum_{r \in \mathcal{TR}(e)} \frac{\rho(r)}{1 + \sum_{q \in P \setminus \{p\}} \sum_{e_q \in S(q) \cap \mathcal{B}(r)} \Pr[e_q])} \right)$$
$$\leq \sum_{p \in \mathcal{P}} \sum_{e \in S(p)} \Pr[e] \left(\sum_{r \in \mathcal{TR}(e)} \frac{\rho(r)}{\sum_{q} \sum_{e_q \in S(q) \cap \mathcal{B}(r)} \Pr[e_q])} \right), \tag{2}$$

where in the last step we used that $\sum_{e_p \in S(p) \cap \mathcal{B}(r)} \Pr[e_p] \leq \sum_{e_p \in S(p)} \Pr[e_p] = 1$. Hence, by using $\sum_q \sum_{e_q \in S(q) \cap \mathcal{B}(r)} \Pr[e_q] = \gamma(r)$ and Lemma 8 we get that

$$n \ln n \leq \sum_{p \in \mathcal{P}} \sum_{e \in S(p)} \Pr[e] \left(\sum_{r \in \mathcal{TR}(e)} \frac{\rho(r)}{\gamma(r)} \right) = \sum_{r \in \mathcal{R}} \rho(r).$$

Note that $\rho(r)$ is the lower bound on the expected number of total reads from register r. Hence, since the expected number of total reads is at least $n \ln n$, there must exist a processor which performs at least $\ln n$ reads in expectation. ◀

4.4 Extension for Bounded Concurrent-Write Contention

We now extend our result to the case where the maximum number of processors which may be poised to write concurrently to a register, which we defined as the concurrent-write contention, is bounded. Specifically, suppose that, in any execution, at most $\kappa \geq 1$ different processors may be poised to write to the same register. We preserve the notation from the previous section. Upon close examination, notice that Lemma 7 and Lemma 8 still hold in this MWMR model, as well as Claim 11, since they do not employ the SWMR property. (By contrast, Claim 12 no longer holds for $\kappa > 1$.) We therefore continue to use only the above results.

We will prove a $\frac{\ln n}{\kappa^2}$ lower bound on the expected solo step complexity, under the above assumptions on κ. As before, let $\mathcal{TR}(e) = u_1^e, u_2^e, \ldots, u_{|\mathcal{TR}(e)|}^e$ be the trace of execution e.

▶ **Lemma 14.** *We have that* $n \ln n \leq \sum_{r \in \mathcal{R}} \rho(r) \sum_{p \in \mathcal{P}} \sum_{e \in S(p) \cap \mathcal{B}(r)} \frac{Pr[e]}{1 + \sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e)}$.

Proof. The proof is similar to the proof of Theorem 13, but using Claim 11 directly instead of Lemma 10. Specifically, we start by summing up the inequalities resulting from Claim 11 for all processors and solo executions:

$$n \ln n \leq \sum_{p \in \mathcal{P}} \sum_{e \in S(p)} \Pr[e] \left(\sum_{i=1}^{|\mathcal{TR}(e)|} \frac{\rho(u_i^e)}{1 + \sum_{j=1}^{i-1} \rho(u_j^e)} \right)$$
$$= \sum_{r \in \mathcal{R}} \rho(r) \sum_{p} \sum_{e \in S(p) \cap \mathcal{B}(r)} \frac{Pr[e]}{1 + \sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e)}.$$

The last equality follows by re-arranging terms to be grouped by register instead of by processor. Note that this is similar to the proof of Lemma 8. However, in this case the resulting equation cannot be simplified further, since, unlike $\gamma(u_i^e)$, the denominator term $1 + \sum_{j=1}^{i-1} \rho(u_j^e)$ also depends on the execution e. ◀

For any register r, we call the set of processors G *a poise set* for r if:

- G contains $g := |G|$ solo executions of different processors, i.e. $G = \{e_1, e_2, \ldots, e_g\}$, such that $r \in \mathcal{TR}(e_i)$, $e_i \in S(p_i)$ and $p_i \neq p_j$ for $i \neq j$.
- Let e'_i be the prefix of e_i up to and including p_i's first write step to the register r. There exists *a combined execution* e by processors $p_1, \ldots p_g$, such that at the end of e all processors p_i have written to r. Moreover, e is indistinguishable from e'_i to p_i (i.e. p_i takes steps as in e_i and does not read anything written by $p_{j\neq i}$ until it writes to r).

As g processors can be poised to write to r in the combined execution, no poise set can have size $> \kappa$.

▶ **Lemma 15.** *Let $E \subseteq \mathcal{B}(r)$ be a set of solo executions. Let k be the maximum size of a poise set for register r among executions in E. Then, there exists a subset of executions $H(E) \subseteq E$, such that:*

- $\sum_{e\in H(E)} \frac{Pr[e]}{1+\sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e)} \leq k$;
- *Every poise set for register r among executions in $E \setminus H(E)$ has size at most $k-1$.*

Proof. Let $\beta = \min_{p_1,\ldots,p_k} \sum_{q\notin\{p_1,\ldots p_k\}} \sum_{e\in E\cap S(q)} Pr[e]$, i.e. the sum of probabilities of executions in E, excluding executions by k processors. We define the set H as follows:

$$H(E) = \Big\{ e \mid \sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e) \geq \frac{\beta}{k} \Big\}.$$

So, an execution e is included in $H(E)$ if the sum of read potentials of registers written prior to r in e is lower bounded by a term that depends on k. Notice that for $k = 1$ this is analogous to the condition in Claim 12.

The parameter β satisfies the following useful property:

$$\begin{aligned} k + \beta &= k + \min_{p_1,\ldots,p_k} \sum_{q\notin\{p_1,\ldots p_k\}} \sum_{e\in E\cap S(q)} Pr[e] = \min_{p_1,\ldots,p_k} \Big(k + \sum_{q\notin\{p_1,\ldots p_k\}} \sum_{e\in E\cap S(q)} Pr[e]\Big) \\ &\geq \min_{p_1,\ldots,p_k} \Big(\sum_{q\in\{p_1,\ldots p_k\}} \sum_{e\in E\cap S(q)} Pr[e] + \sum_{q\notin\{p_1,\ldots p_k\}} \sum_{e\in E\cap S(q)} Pr[e]\Big) \qquad (3) \\ &= \min_{p_1,\ldots,p_k} \Big(\sum_{q} \sum_{e\in E\cap S(q)} Pr[e]\Big) = \sum_{q} \sum_{e\in E\cap S(q)} Pr[e] = \sum_{e\in E} Pr[e]. \end{aligned}$$

where in (3) we have used that, from the definition of S, $\sum_{e\in S(p_i)} \Pr[e] = 1$.

Using this property, we get:

$$\begin{aligned} \sum_{e\in H(E)} \frac{Pr[e]}{1+\sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e)} &\leq \sum_{e\in H(E)} \frac{Pr[e]}{1+\frac{\beta}{k}} = k \sum_{e\in h(E)} \frac{Pr[e]}{k+\beta} \\ &\leq k \frac{\sum_{e\in h(E)} Pr[e]}{\sum_{e\in E} Pr[e]} \leq k. \end{aligned}$$

This proves the first part of the lemma. We prove the second part of the lemma by contradiction. Suppose there is a poise set $G = \{e_1, e_2, \ldots e_k\}$ for register r among executions in $E \setminus H(E)$, where e_i is an execution of processor p_i.

Consider any execution $e' \in E \cap S(q)$ for $q \notin \{p_1, \ldots, p_k\}$. Execution e' must read one of the registers written during some time step before the point when r is written in e_i. Otherwise, e', and more precisely, the prefix of e' up to the write to r, can be added at the end of G's interleaved execution, implying that $G \cup \{e'\}$ would be a poise set of size $k+1$ among executions in E, which does not exist by definition. Hence:

$$\sum_{e \in G} \Big(\sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e) \Big) \geq \sum_{q \notin \{p_1, \ldots p_k\}} \sum_{e \in E \cap S(q)} Pr[e] \geq \beta.$$

Notice how this generalizes Claim 12: we can now apply the pigeonhole principle to $|G| = k$ terms on the left hand side. We get that for some i, $e_i \in H(E)$, giving the desired contradiction, specifically, that G consists of executions from $E \setminus H(E)$ only. This completes the proof of the Lemma. ◀

We are now ready to prove the main result of this section.

▶ **Theorem 16.** *Any randomized leader election protocol in asynchronous shared-memory has $\frac{\ln n}{\kappa^2}$ worst-case expected solo step complexity, when κ is the maximum number of processors which may be poised to write concurrently to the same register.*

Proof. Fix a register r. We start by applying Lemma 15 to the set $\mathcal{B}(r)$ and the maximum poise set size of κ. Let E_1 be the resulting subset of executions $H(\mathcal{B}(r))$, and $k_1 \leq \kappa$ be the maximum size of a poise set among executions in $\mathcal{B}(r) \setminus E_1$. Next, we apply Lemma 15 again to $\mathcal{B}(r) \setminus E_1$, and define $E_2 = H(\mathcal{B}(r) \setminus E_1)$, and $k_2 < k_1$ as the maximum size of a poise set among executions in $\mathcal{B}(r) \setminus (E_1 \cup E_2)$. The next application of Lemma 15 will be to $\mathcal{B}(r) \setminus (E_1 \cup E_2)$, defining $E_3 = H(\mathcal{B}(r) \setminus (E_1 \cup E_2))$ and $k_3 < k_2$. We repeat the process until some k_ℓ becomes 0, implying that the set of remaining executions $\mathcal{B}(r) \setminus (\cup_{t=1}^{\ell-1} E_t)$ is empty. Since $0 = k_\ell < k_{\ell-1} < \ldots \leq \kappa$, Lemma 15 will be applied at most κ times. Therefore we have obtained that:

$$\begin{aligned} \sum_p \sum_{e \in S(p) \cap \mathcal{B}(r)} \frac{Pr[e]}{1 + \sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e)} &= \sum_{e \in \mathcal{B}(r)} \frac{Pr[e]}{1 + \sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e)} \\ &= \sum_{t=1}^{\ell-1} \sum_{e \in E_t} \frac{Pr[e]}{1 + \sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e)} \leq \kappa^2, \end{aligned}$$

as there are at most κ terms, each of which is upper bounded by κ, by Lemma 15.

Combined with Lemma 14, this gives $\sum_r \rho(r) \geq \frac{n \ln n}{\kappa^2}$. By the pigeonhole principle, some processor must perform at least $\frac{\ln n}{\kappa^2}$ reads in expectation, over its solo executions. ◀

5 A Complementary Upper Bound for Weak Leader Election

It is interesting to consider whether the lower bound approach can be further improved to address the MWMR model under n-concurrent write contention. This is not the case for the specific definition of the weak leader election problem we consider (Definition 1), and to which the lower bound applies. To establish this, it suffices to notice that the classic *splitter* construction of Lamport [25] solves weak leader election for n processes, *in constant time*, by leveraging MWMR registers with maximal (concurrent) write contention n.

Please recall that this construction, restated for convenience in Figure 1, uses two MWMR registers. Given a splitter, we can simply map the *stop* output to *win*, and the *left* and *right* outputs to *lose*. In this case, it is immediate to show that the splitter ensures the following:

1. a processor will always return *win* in a solo execution, and
2. no two processes may return *win* in the same execution.

```
   shared data:
1  atomic register race, big enough to hold an id, initially ⊥
2  atomic register door, big enough to hold a bit, initially open
3  procedure splitter(id)
4      race ← id
5      if door = closed then
6          return right
7      door ← closed
8      if race = id then
9          return stop
10     else
11         return down
```

Figure 1 The classic Lamport splitter [25], restated following [26, 6].

This matches the requirements of the *weak leader election* problem, but not of *test-and-set* objects generally, as this algorithm has contended executions in which all processors return *lose*, which is also impractical.

One may further generalize this approach by defining κ-splitter objects for $\kappa \geq 2$, each of which is restricted to κ participating processors (and thus also κ-concurrent write contention), and then arranging them in a complete κ-ary tree. We can then proceed similarly to tournament tree, to implement a weak leader election object. The resulting construction has $O(\log n / \log \kappa)$ step complexity in solo executions, suggesting that the dependency on κ provided by our argument can be further improved.

This observation suggests that the trade-off between step complexity and concurrent-write contention/worst-case stalls outlined by our lower bound may be the best one can prove for *weak leader election*, as this problem can be solved in constant time with MWMR registers, at the cost of linear worst-case stalls. At the same time, it shows that lower bound arguments wishing to approach the general version of the problem have to specifically leverage the fact that, even in contended executions, not all processors may return *lose*.

6 Conclusion

Overview. We gave the first tight logarithmic lower bounds on the solo step complexity of leader election in an asynchronous shared-memory model with single-writer multi-reader (SWMR) registers, for both deterministic and randomized algorithms. We then extended these results to registers with bounded concurrent-write contention $\kappa \geq 1$, showing a trade-off between the step solo complexity of algorithms, and their worst-case stall complexity. The approach admits additional extensions, and is tight in the SWMR case. The impossibility result is quite strong, in the sense that logarithmic time is required *over solo executions* of processors, and for a weak variant of leader election, which is not linearizable and allows processors to all return lose in in contended executions.

Future Work. The key question left open is whether sub-logarithmic upper bounds for strong leader election / test-and-set may exist, specifically by leveraging multi-writer registers, or whether the lower bounds can be further strengthened. Another interesting question is whether our approach can be extended to handle different cost metrics, such as remote memory references (RMRs).

References

1 Yehuda Afek, Eli Gafni, John Tromp, and Paul M. B. Vitányi. Wait-free test-and-set (extended abstract). In *WDAG '92: Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 85–94, 1992.

2 Dan Alistarh and James Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *International Symposium on Distributed Computing*, pages 97–109. Springer, 2011.

3 Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Rachid Guerraoui. Tight bounds for asynchronous renaming. *Journal of the ACM (JACM)*, 61(3):1–51, 2014.

4 Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *Proceedings of the 24th international conference on Distributed computing*, DISC'10, pages 94–108, Berlin, Heidelberg, 2010. Springer-Verlag. URL: `http://portal.acm.org/citation.cfm?id=1888781.1888794`.

5 Dan Alistarh, Rati Gelashvili, and Adrian Vladu. How to elect a leader faster than a tournament. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 365–374, 2015.

6 James Aspnes. Notes on theory of distributed systems. *arXiv preprint*, 2020. `arXiv:2001.04235`.

7 James Aspnes, Keren Censor-Hillel, Hagit Attiya, and Danny Hendler. Lower bounds for restricted-use objects. *SIAM Journal on Computing*, 45(3):734–762, 2016.

8 Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *J. ACM*, 55(5):1–26, 2008. `doi:10.1145/1411509.1411510`.

9 Hagit Attiya and Faith Ellen. Impossibility results for distributed computing. *Synthesis Lectures on Distributed Computing Theory*, 5(1):1–162, 2014.

10 James E Burns and Nancy A Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993.

11 Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM (JACM)*, 44(6):779–805, 1997.

12 Aryaz Eghbali and Philipp Woelfel. An almost tight rmr lower bound for abortable test-and-set. *arXiv preprint*, 2018. `arXiv:1805.04840`.

13 Faith Ellen, Danny Hendler, and Nir Shavit. On the inherent sequentiality of concurrent objects. *SIAM Journal on Computing*, 41(3):519–536, 2012.

14 George Giakkoupis, Maryam Helmi, Lisa Higham, and Philipp Woelfel. An $O(\sqrt{n})$ space bound for obstruction-free leader election. In *International Symposium on Distributed Computing*, pages 46–60. Springer, 2013.

15 George Giakkoupis, Maryam Helmi, Lisa Higham, and Philipp Woelfel. Test-and-set in optimal space. In *Proceedings of the forty-seventh annual ACM symposium on Theory of Computing*, pages 615–623, 2015.

16 George Giakkoupis and Philipp Woelfel. Efficient randomized test-and-set implementations. *Distributed Computing*, 32(6):565–586, 2019.

17 Wojciech Golab, Danny Hendler, and Philipp Woelfel. An o(1) rmrs leader election algorithm. *SIAM Journal on Computing*, 39(7):2726–2760, 2010.

18 Danny Hendler and Nir Shavit. Operation-valency and the cost of coordination. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 84–91, 2003.

19 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, 1991.

20 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pages 522–529. IEEE, 2003.

21 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

22 Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 201–210, 1998.

23 Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.

24 Yong-Jik Kim and James H Anderson. A time complexity lower bound for adaptive mutual exclusion. *Distributed Computing*, 24(6):271–297, 2012.

25 Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)*, 5(1):1–11, 1987.

26 Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25:1–39, 1995.

27 Gary L. Peterson and Michael J. Fischer. Economical solutions for the critical section problem in a distributed system (extended abstract). In *Proceedings of the ninth annual ACM symposium on Theory of computing*, STOC '77, pages 91–97, New York, NY, USA, 1977. ACM. doi:10.1145/800105.803398.

28 Eugene Styer and Gary L Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 177–191, 1989.

29 John Tromp and Paul Vitányi. Randomized two-process wait-free test-and-set. *Distrib. Comput.*, 15(3):127–135, 2002. doi:10.1007/s004460200071.

30 Jae-Heon Yang and James H Anderson. Time bounds for mutual exclusion and related problems. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of Computing*, pages 224–233, 1994.