Lower Bounds on the Amortized Time Complexity of Shared Objects

Hagit Attiya[*1] **and Arie Fouren**[†2]

1 **Department of Computer Science, Technion, Haifa 32000, Israel**
`hagit@cs.technion.ac.il`

2 **Faculty of Business Administration, Ono Academic College, Kiryat Ono, 5545173, Israel**
`aporan@ono.ac.il`

Abstract

The *amortized* step complexity of an implementation measures its performance as a whole, rather than the performance of individual operations. Specifically, the amortized step complexity of an implementation is the average number of steps performed by invoked operations, in the worst case, taken over all possible executions. The amortized step complexity of a wide range of known lock-free implementations for shared data structures, like stacks, queues, linked lists, doubly-linked lists and binary trees, includes an additive factor linear in the *point contention*—the number of processes simultaneously active in the execution.

This paper shows that an additive factor, linear in the point contention, is inherent in the amortized step complexity for lock-free implementations of many distributed data structures, including stacks, queues, heaps, linked lists and search trees.

1998 ACM Subject Classification C.1.4 Parallel Architectures, D.4.1 Process Management

Keywords and phrases monotone objects, stacks and queues, trees, step complexity, remote memory references

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.16

1 Introduction

Evaluating the complexity of lock-free implementations, in which an operation may never terminate, is best done through their *amortized* step complexity, defined as the average number of steps performed by invoked operations, in the worst case taken over all possible executions [12]. Amortized step complexity measures the performance of the system as a whole, rather than the performance of individual operations.

Ruppert [12] defined this complexity measure and observed that upper bounds on the amortized step complexity of a wide range of lock-free implementations of shared data structures has an additive factor of $\dot{c}(op)$; $\dot{c}(op)$ is the *point contention* during an operation *op*, namely, the number of processes simultaneously active during the execution interval of *op*. These objects include stacks and queues [14], linked lists [6], doubly-linked lists [13] and binary search trees [5]. Ruppert asks whether the additive factor of $\dot{c}(op)$ in the expression for amortized step complexity for lock-free distributed data structures is inherent.

[*] This work is supported by the Israel Science Foundation (grant number 1749/14)
[†] This work is supported by the Ono Academic College Research Fund

This paper answers this question in the affirmative, for a wide range of shared data structures, in particular, stacks, queues, heaps, linked lists and search trees. We prove two classes of lower bounds.

The first, bounds the amortized number of *Remote Memory References* (*RMR*s) and it is done by reduction to a variant of the set object, called sack. A sack supports two operations: one operation adds an element, while another atomically removes an element and returns it (returning $\perp$ if the sack is empty). We prove that the amortized RMR complexity of any implementation of a sack using reads, writes and conditional primitives (such as CAS), is at least $\Omega(\dot{c})$. The proof is by reduction to the lower bound for *mutual exclusion* [10], which we extend to hold for amortized step complexity. We show that several shared objects (i.e., stacks, queues and heaps) can easily be used to implement a sack with an $O(1)$ additional cost. This proves that the amortized RMR complexity of these shared objects is also at least $\Omega(\dot{c})$, when the point contention is in $O(\sqrt{\log\log n})$.

We prove another set of lower bounds on the amortized step complexity of *monotone* objects that do not support an atomic remove operation. The proof holds for data structures that are implemented by a connected graph of nodes, like linked lists, skip lists or search trees. The proof is more self-contained, but it bounds only the *step* complexity—a measure that is larger than the RMR complexity. The lower bound of $\Omega(\dot{c})$ for the amortized step complexity holds for implementations using 1-*revealing primitives*, a class including reads, writes, LL/SC, test&set and CAS [3].[1] The lower bound holds when the point contention is in $O(\log\log n)$.

Ruppert [12] provides analysis showing that the known lock-free implementations of stacks and queues [14] have $O(\dot{c}(op))$ amortized step complexity. In addition, he also observes that:

- The search, put and delete operations of the linked-list implementation presented in [6], have $O(n(op) + \dot{c}(op))$ amortized step complexity, where $n(op)$ is the number of the elements in the list when operation op is invoked.
- The amortized step complexity of the search, put and delete on a non-blocking binary tree [5] is $O(h(op) + \dot{c}(op))$, where $h(op)$ is the height of the tree at the beginning of the operation op.
- The amortized step complexity of put and delete in a doubly-linked list [13] is $O(\dot{c}(op))$.
- The wait-free union-find implementation [2] has $O(\alpha(n) + \dot{c}(op))$ amortized step complexity, where n is the number of elements in the sets and $\alpha(n)$ is the inverse of Ackermann function.

Our lower bounds show that the $\dot{c}(op)$ component in the amortized step complexity of these implementations is inherent, except perhaps for the last one, the wait-free union-find, which remains an interesting open problem.

2 The Computation Model

In the *asynchronous shared-memory* model [10], n *processes*, $p_0, \ldots, p_{n-1}$ communicate by applying *primitive operations* (in short, *primitives*) to shared memory registers. Initially, all shared registers hold the value $\perp$. A process is described as a state machine, with a set of (possibly infinitely many) *states*, one of which is a designated *initial state*, and a state transition function.

[1] Using primitives with more revealing power one can get a more efficient implementation of monotone objects. For example, using swap primitive that is $m/2$-revealing [3], it is possible to implement an add operation for a monotone linked list with $O(1)$ step complexity.

The executions of the system are sequences of events. In each event, based on its current state, a process applies a primitive to a shared memory register and then changes its state, according to the state transition function. An *event* ϕ in which a process p applies a primitive op to register R is denoted by a triple $\langle p, R, op\rangle$. An *execution* α is a (finite or infinite) sequence of events $\phi_0, \phi_1, \phi_2, \ldots$. There are no constraints on the interleaving of events by different processes, reflecting the assumption that processes are asynchronous. The value of variable v after α is denoted $val(v, \alpha)$. Without loss of generality, we assume that the value of a shared register is never set to $\perp$ during an execution.

For an execution α and a set of processes P, $\alpha|_P$ is the sequence of all events in α by processes in P; $\alpha|_{\overline{P}}$ is the sequence of all events in α that are *not* by processes in P. If $P = \{p\}$, we write $\alpha|_p$ instead of $\alpha|_{\{p\}}$ and $\alpha|_{\overline{p}}$ instead of $\alpha|_{\overline{\{p\}}}$. An execution α is P*-only* if $\alpha = \alpha|_P$, and it is P*-free* execution if $\alpha = \alpha|_{\overline{P}}$.

The basic primitives are read and write: A read(R) primitive returns the current value of R and does not change its value. A write(v, R) operation sets the value of R to v, and does not return a value. Every process can read from or write to every register, i.e., registers are *multi-writer multi-reader*. A compare&swap (or CAS for short) primitive works as follows. If the register R holds the value v, then after CAS (R, v, u) the state of R is changed to u and true is returned (the CAS *succeeds*). Otherwise, the state of R remains unchanged and false is returned (the CAS *fails*).

An *implementation* of a high-level object provides algorithms for each high-level operation supported by the object. Some transitions are *requests*, invoking a high-level operation, or *responses* to a high-level operation. When a high-level operation is invoked, the process executes the algorithm associated with the operation, applying primitives to the shared registers, until a response is returned.

Let α' be a finite prefix of an execution α. Process p_i performing a high-level operation op is *active* at the end of α', if α' includes an invocation of op without a return from op. The set of the processes active at the end of α' is denoted $active(\alpha')$. The *point contention* at the end of α', denoted $\dot{c}(\alpha')$, is $|active(\alpha')|$.

Consider an execution α of an algorithm A implementing a high-level operation op. For process p_i executing operation op_i, $step(A, \alpha, op_i)$ is the number events by p_i, when executing op_i in α. The step complexity of A in α, denoted $step(A, \alpha)$, is the maximum of $step(A, \alpha, op_i)$ over all operations op_i of all processes p_i.

The *amortized* step complexity of A in an execution α is the total number of shared-memory events performed by all the operations initiated in α (denoted $initiated(\alpha)$) divided by the number of invoked operations:

$$amortizedStep(A, \alpha) = \frac{\sum_{op_i \in initiated(\alpha)} step(A, \alpha, op_i)}{|initiated(\alpha)|}$$

The amortized step complexity of A is the maximum of $amortizedStep(A, \alpha)$ over all possible executions α of A:

$$amortizedStep(A) = \max_{\alpha}\{amortizedStep(A, \alpha)\}$$

Consider a bounded function $\mathrm{S} : \mathcal{N} \mapsto \mathcal{N}$. The step complexity of an algorithm implementing operation op is S-*adaptive to point contention* if for every execution α and every operation op_i with interval β_i, $step(A, \alpha, op_i) \leq \mathrm{S}(\dot{c}(\beta_i))$. That is, the step complexity of an operation op_i with interval β_i is bounded by a function of the point contention during β_i.

Similarly, the amortized step complexity of an algorithm A is S-*adaptive to point contention* if for every execution α, $amortizedStep(A, \alpha) \leq \mathrm{S}(\dot{c}(\alpha))$. That is, the amortized step complexity of algorithm A in an execution α is bounded by a function of the point contention during α.

In this paper, we consider only the *cache-coherent* (CC) model. In the CC model, each process has a local cache in addition to shared memory. A shared variable accessed by a process is loaded to its local cache and remains locally accessible to the process until it is modified by another process. Such a modification results with an update or an invalidation of the cached variable, according to some *cache-consistency* protocol.

In many mutual-exclusion algorithms, a process busy-waits by repeatedly testing one or more local "spin variables". For such algorithms the number of shared memory operations may be unbounded. Instead of counting the shared-memory operations, we count the number of *remote-memory-references* (RMR) generated by the algorithm [1]. The RMR complexity counts only events that cause process-memory interconnection traffic, and ignores events in busy-wait loops on unchanged local variables.

Consider an execution prefix $\alpha'\phi$ of α, where $\phi_i = \langle p_i, v, op_i \rangle$. Following [10], ϕ_i is an *RMR event* if ϕ_i is the first event in α in which p_i accesses v (that is, p_i does not access v in α') or ϕ_i is the first event in α in which p_i accesses v after v was modified by another process (that is, the last event in α' that modifies v is performed by a process $p_j \neq p_i$).

The definition of *RMR complexity* is the same as the definition of step complexity, except that we count only RMR events. Similarly, the definition of *amortized RMR complexity* is the same as the definition of amortized step complexity, counting only RMR events.

3 Lower Bound for Sack Implementation and Related Objects

In this section we define a variation of set object, called sack, and use it to implement mutual exclusion with additional $O(1)$ cost (Algorithm 1). Then we show that sack can be implemented from queues, stacks and heaps with additional $O(1)$ cost (Lemma 7). An $\Omega(\dot{c})$ lower bound for all these data structures then follows from the $\Omega(\dot{c})$ lower bound for mutual exclusion [10], which we extend to amortized RMR complexity (Appendix A).

A sack object supports the put and draw operations, with the following specification:

S.put(v) adds element v to the sack S

S.draw() removes an arbitrary element from the sack S, and returns the removed element. Note that we do not specify what element should be removed. If before the invocation of the operation the sack is empty, the operation returns $\perp$.

The mutual exclusion algorithm shown in this section uses a sack object combined with a few additional CAS and R/W variables. A process can enter the critical section either through a *fast lane*, winning the mutual exclusion associated with a CAS variable *fastLane*, or through a *waiting room*, implemented using a sack object *waitingRoom*. After passing the fast lane or the waiting room, a process has to win a 2-process mutual exclusion implemented with a CAS variable *gate*, in order to enter the actual critical section (see Figure 1).

In more detail (see Algorithm 1 in the appendix), to enter the critical section, process p_i sets $wants[id_i]$ to TRUE, and adds itself to the set of the waiting processes (Line 1.3). Then it tries to win the fast lane, performing a CAS on the *fastLane* variable (Line 1.4). If the CAS succeeds and p_i successfully writes $\langle \text{OCCUPIED}, id_i \rangle$ to *fastLane*, then p_i accesses the *gate* mutual exclusion from the fast lane side (Line 1.9). If CAS on *fastLane* fails (Line 1.4), then p_i busy waits until $privateGate[id_i]$ becomes OPEN (Line 1.7), and then it accesses the *gate* mutual exclusion from the waiting room side (Line 1.9).

In the exit section, p_i sets its variable $wants[id_i]$ to FALSE (Line 1.12) to indicate to other processes that it is not interested to enter the critical section and thus does need help. To complete clean-up, p_i sets $privateGate[id_i]$ to CLOSED (Line 1.13). If p_i previously entered *gate* mutual exclusion through the fast line (in this case $fastLane = \langle \text{OCCUPIED}, id \rangle$), then it resets *fastLane* to OPEN performing a CAS (Line 1.14).

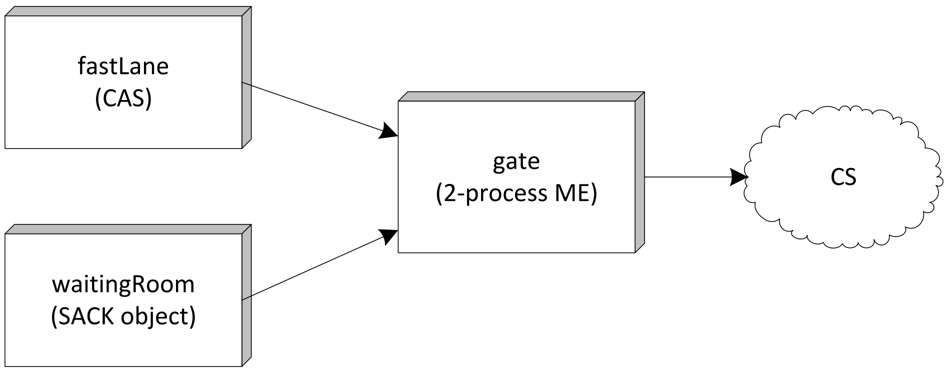

Figure 1 Mutual exclusion using a sack object and CAS presented in Algorithm 1.

Then p_i tries to promote one of the processes from *waitingRoom* to enter the critical section. The R/W variable *promoted* is used to keep track of the promoted process. If in the exit section p_i finds *promoted* equal to its own id_i, then p_i resets *promoted* to $\bot$ (Line 1.16). If p_i finds no promoted process ($promoted = \bot$, Line 1.18), then p_i repeatedly removes a process from *waitingRoom*, until it finds a process *next* that is still interested to enter the critical section, or until *waitingRoom* is empty (Line 1.20). If the removed process *next* is still interested to enter the critical section ($wants[next] =$ TRUE), then p_i promotes this process by setting $promoted = next$ (Line 1.22), and opens the *next*'s private gate by setting $privateGate[next] =$ OPEN (Line 1.23). Finally, p_i releases the critical section associated with *gate* by setting *gate* to TRUE (Line 1.26).

Below we prove the correctness and bound the RMR complexity of Algorithm 1.

The *mutual exclusion* property of the algorithm follows from the mutual exclusion property of the gate block (Figure 1). A process p_i enters critical section only after successful CAS that sets *gate* to CLOSED (Line 1.9) and resets *gate* to OPEN, when it releases the critical section (Line 1.26). Therefore, no two processes may be simultaneously inside the critical section.

The following definition of a *promoted* process is used in the proof of deadlock freedom and in the analysis of the RMR complexity of the algorithm.

A process p_i is *promoted* if $privateGate[id_i] =$ OPEN. The next lemma shows that at most one process is promoted at any point.

▶ **Lemma 1.** *At any point of execution, if $privateGate[id_i] =$ OPEN, then $promoted = id_i$.*

Proof. Initially, for every process p_j, $privateGate[id_j] =$ CLOSED and the lemma trivially holds.

Suppose that the lemma holds for an execution prefix α', and let p_i be the first process that modifies *privateGate* after α'. Before p_i sets $privateGate[next] =$ OPEN in its exit section (Line 1.23), p_i sets $promoted = next$ (Line 1.22). By the mutual exclusion property of the algorithm, these steps are performed by p_i in exclusion, and the lemma holds. ◀

The following technical lemma is used to prove that the algorithm has no deadlocks.

▶ **Lemma 2.** *Let $exit_i$ be the execution interval corresponding to the exit section of a process p_i. Then one of the following holds:*

(a) *at the end of $exit_i$ there is a promoted process p_j, or*
(b) *p_i's invocation of $waitingRoom.\mathsf{draw}()$ returns $\perp$ (Line 1.19).*

Proof. If p_i reads $promoted = id_j \neq \perp$ (Line 1.18), then by Lemma 1 and the mutual exclusion property, $privateGate[id_j] = \mathsf{OPEN}$ at this point, and p_j remains to be promoted until the end of $exit_i$. In this case, condition (a) holds.

If p_i reads $promoted = \perp$ (Line 1.18), and succeeds to set $privateGate[next] = \mathsf{OPEN}$ (Line 1.23) then process $next$ is promoted at this point, and it remains promoted until the end of $exit_i$, and condition (a) holds.

If p_i reads $promoted = \perp$ (Line 1.18) but it does not perform $privateGate[next] = \mathsf{OPEN}$ (Line 1.23), then $waitingRoom.\mathsf{draw}()$ returns $\perp$ (Line 1.19), implying condition (b). ◀

The next lemma proves the deadlock-freedom property of the algorithm. For simplicity, we assume that each process enters the critical section at most once. Since the lower bound presented in [10] holds for one-shot mutual exclusion, this suffices for our proof.

▶ **Lemma 3.** *Algoritm 1 is deadlock-free.*

Proof. Suppose that there is an execution α with prefix α', such that after α' there is a process p_i in its entry section, and from that point on no process ever enters the critical section in α.

If p_i's CAS on $fastLane$ succeeds (Line 1.4), then p_i waits on the $gate$ variable (Line 1.9). This is a contradiction, since no process remains in the critical section associated with $gate$ forever.

Therefore, p_i loses the $fastLane$ in Line 1.4 and remains in $waitingRoom$ forever, waiting for $privateGate[id]$ to be OPEN (Line 1.7).

Since p_i's CAS on $fastLane$ fails, another process p_j wins $fastLane$ in its enter section (Line 1.4) before p_i's CAS, and releases $fastLane$ in its exit section (Line 1.14) after p_i's CAS. Process p_j performs $waitingRoom.\mathsf{draw}()$ (Line 1.19) after it releases $fastLane$ (Line 1.14). By assumption, p_i remains in $waitingRoom$ forever, and therefore, the $waitingRoom.\mathsf{draw}()$ invocation of p_j (Line 1.19) does not return $\perp$.

Therefore, statement (a) of Lemma 2 holds, implying that some process p_k is promoted at the end of p_j's exit interval. Eventually, p_k accesses the $gate$ mutual exclusion after it reads $privateGate[k] = \mathsf{OPEN}$ (Line 1.23) or wins the $fastLane$ (Line 1.4). Since $gate$ mutual exclusion has no deadlocks, eventually some process enters the critical section after α', which is a contradiction. ◀

Now we show that at any point, at most one process can access the $gate$ mutual exclusion from the fast lane, and at most one process from the waiting room side (Figure 1). This implies that the $gate$ mutual exclusion has constant RMR complexity (in addition to the RMR complexity of the sack implementation for $waitingRoom$).

▶ **Definition 4.** A process p_i is on *fast lane* if and only if $fastLane = \langle \mathsf{OCCUPIED}, id_i \rangle$.[2]

By the semantics of CAS and induction on the execution order, we have the following lemma:

▶ **Lemma 5.** *The execution intervals in which different processes p_i and p_j are on fast lane are disjoint.*

[2] That is, p_i is on fast lane after it successfully sets $fastLane$ to $\langle \mathsf{OCCUPIED}, id_i \rangle$ (Line 1.4) and before it successfully resets $fastLane$ to EMPTY (Line 1.14) in the exit section.

Lemma 5 implies that at any point, at most one process accesses *gate* mutual exclusion from the fast lane, and Lemma 1 implies that at most one process accesses the *gate* mutual exclusion from the waiting room. We have that at most two processes simultaneously access the *gate* mutual exclusion and therefore, waiting for *gate* (Line 1.9) has a constant amortized RMR complexity (each event that generates RMR can be charged to a successful entry to the *gate* critical section). Each process is added and removed from the waiting room sack at most once, and the rest of the operations in the entry and exit sections have a constant RMR complexity. This implies:

▶ **Lemma 6.** *Given an implementation of* sack *with $O(f(\dot{c}))$ amortized RMR complexity. Then there is a mutual exclusion algorithm with $O(f(\dot{c}))$ amortized RMR complexity, using in addition,* R/W *and* CAS.

The following lemma shows that sack can be implemented from several shared objects, with a constant additional cost:

▶ **Lemma 7.** *The following data structures: queues, stacks and heaps, can be used to implement the* sack *object operations with $O(1)$ additional steps.*

Proof. (a) queue operations enqueue and dequeue trivially implement the put and draw operations of sack; (b) stack operations push and pop trivially implement the put and draw operations of sack; (c) heap operations add and removeMax trivially implement the put and draw operations of sack. ◀

Theorem 20 (presented in Appendix A), Lemma 6 and Lemma 7 imply the next theorem:

▶ **Theorem 8.** *Any implementation of queues, stacks and heaps, from reads, writes and conditional primitives has at least $\Omega(\dot{c})$ amortized RMR complexity.*

4 Lower Bounds for Graph-Based Set Implementations

This section defines *graph-based* implementations of the set object (denoted graph-based-set), and proves that any implementation of graph-based-set from 1-revealing primitives has an $\Omega(\dot{c})$ amortized step complexity. This is a generalization of data structures that can be represented by a connected graph of nodes, like linked lists, skip lists or search trees. The lower bound on the amortized step complexity of graph-based-set holds also for implementation of these data structures.

To emphasize the nature of the data structures based on a connected set of nodes, this section assumes the following memory model. Each shared variable contains one data structure node, which has a data field and a constant number d of pointers connecting it to other nodes (Figure 2). A process can read or modify all the components of a node in a single computational step.[3]

The state of the shared memory after an execution prefix α can be represented by a *memory graph* $G(\alpha) = (V, E)$, where V is set of the shared variables in the system, and there is an edge $v_i \to v_j \in E$ if and only if after α, the shared variable corresponding to $v_i \in V$ contains a pointer to the shared variable corresponding to $v_j \in V$.

[3] Many known implementations of linked data structures use a separate variable for each node component, and a fixed memory offset to access different components of the node. These implementations can be converted to graph-based-set preserving the asymptotic step complexity, by replacing each data structure node with a linked list of graph-based-set nodes, having a separate node for each component.

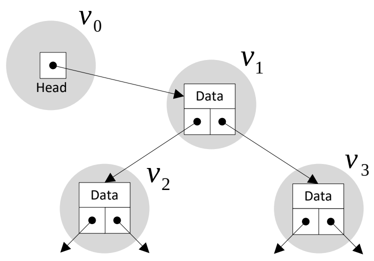

Figure 2 A graph-based representation of a set. Each node of the graph is stored in a separate shared variable v_i and contains a data field and a constant number of pointers. The pointers between the nodes form the edges of the graph. An element v_i is in the set iff it can be traced following the pointers starting from the *head* node.

A graph-based-set implementation specifies a special node, *head*, and supports one operation, add(e), which adds an element e to the set. It should satisfy the following invariant:

- an element e is in the graph-based-set S after an execution prefix α if and only if there is a directed path from $S.head$ to a node v with $v.data = e$, in the memory graph G_α.

Many linked-list and tree-based set implementations [4, 7–9, 11, 16] satisfy this invariant. In contrast to the RMR lower bound of Section 3, the lower bound for graph-based-set only requires an add operation, and does not rely on an atomic remove operation.

We construct an execution of $3k$ processes, in which each process p_i invokes add(id_i) to add its id to the set. We show that these $3k$ processes collectively perform $\Omega(k^2)$ steps, implying that any implementation of graph-based-set has at least $\Omega(k) = \Omega(\dot{c})$ amortized step complexity. The proof uses a technique similar to the $\Omega(\dot{c})$ lower bound on the step complexity of adaptive collect [3].

We construct the execution in rounds, maintaining two disjoint sets of processes, the *invisible* set P, initially containing all processes, and the *visible* set W, initially empty. Intuitively, invisible processes are not aware of each other and do not detect each other's operations, while visible processes are possibly aware of other processes or vice versa.

In round r, each process that is still invisible after the previous round performs its next event. These steps are scheduled so that after each round, at most 2 invisible processes become visible. These processes cannot be erased from the execution. Instead, they are stopped and take no steps in the later rounds. Some of the other invisible processes are retroactively erased from the execution in order to keep the remaining processes invisible after round r.

In each round, at most one process succeeds to add its id to the graph-based-set, and at least k processes are invisible after k rounds. Since in each round each invisible process takes one step, in k rounds the last k invisible processes collectively perform $\Omega(k^2)$ steps, implying an $\Omega(\dot{c})$ lower bound on the amortized step complexity.

To guarantee that a process p_i remains invisible during the execution, we need to show that no other invisible process reads information stored by p_i, and that p_i does not modify any variable previously modified by another invisible process. Otherwise, p_i cannot be erased from the execution without affecting other processes. To formalize the notion of variables affected by a process, we define a set of *evidence* variables. Intuitively, the evidence variables of process p_i are the variables that may contain "traces" of p_i's events, so that the values of these variables may change if p_i's events are deleted from the execution.

▶ **Definition 9** (Evidence Set of Variables)**.** The *evidence set* of process p after execution α, denoted $evidence(p, \alpha)$, is the set of the variables v whose values at the end of α would change if the events of p are deleted from α, that is,

$$evidence(p, \alpha) = \{v \mid val(v, \alpha) \neq val(v, \alpha|_{\overline{p}})\}$$

For a set of processes P, $evidence(P, \alpha) = \cup_{p \in P}\ evidence(p, \alpha)$.

Now we define the notion of a set of processes that are invisible to each other. Intuitively, erasing any subset of an invisible set of processes is undetectable by the remaining invisible processes. To facilitate the inductive construction, we require some additional properties.

▶ **Definition 10** (Invisible Set of Processes)**.** A set P of processes is *invisible in an execution* α if the following hold:
(a) for any subset $Q \subset P$, processes $P \setminus Q$ get the same responses in the executions α and $\alpha|_{\overline{Q}}$;
(b) for any subset $Q \subset P$, and for any process $p \in P \setminus Q$, $evidence(p, \alpha) = evidence(p, \alpha|_{\overline{Q}})$;
(c) for any pair of processes $p, q \in P$, $evidence(p, \alpha) \cap evidence(q, \alpha) = \emptyset$.
The corresponding set of *visible* processes are the active processes in α that are not in P, i.e., $W = active(\alpha) \setminus P$.

Definition 10(a) means that erasing any subset Q of the invisible processes P is undetectable to the remaining invisible processes in $P \setminus Q$. Property (b) implies that erasing any subset Q of invisible processes P does not affect the evidence sets of the remaining invisible processes $P \setminus Q$; it prevents an invisible process $p \in P$ from modifying a variable previously modified by another invisible process $q \in P$, and ensures that erasing p will not make q visible. Property (c) states that the evidence sets of invisible processes are disjoint, so each variable belongs to evidence set of at most one process.

The next lemma (proved in [3]) states that after erasing any subset of an invisible set from an execution, the remaining processes still form an invisible set.

▶ **Lemma 11.** *If a set of processes P is invisible in an execution α, then for every subset $Q \subset P$, the subset $(P \setminus Q)$ is invisible in $\alpha|_{\overline{Q}}$.*

The following definitions of accessible variables and accessible processes are used to bound the size of the underlying graph of graph-based-set.

▶ **Definition 12** (Accessible Variables)**.** A shared variable v is an *accessible variable of a process p_i after an execution prefix α*, if there is a variable $v_0 \in evidence(p_i, \alpha)$, and there is a directed path from v_0 to v in the graph G_α (Figure 3(a)).
The set of the accessible variables of p_i after α is denoted $accessibleVariables(p_i, \alpha)$.

▶ **Definition 13** (Accessible Variables of a graph-based-set)**.** The *accessible variables of a* graph-based-set *S after execution prefix α*, denoted $accessibleVariables(S, \alpha)$, is the set of variables v such that after α there is a directed path from $S.head$ to v in the memory graph G_α (Figure 3(b)).

▶ **Definition 14** (Accessible Processes of a graph-based-set)**.** The *accessible processes of a* graph-based-set *S after execution prefix α*, denoted $accessibleProcesses(S, \alpha)$, are those whose evidence variables are in the set of the accessible variables of S :

$$accessibleProcesses(S, \alpha) = \{p_i : evidence(p_i, \alpha) \cap accessibleVariables(S, \alpha) \neq \emptyset\}$$

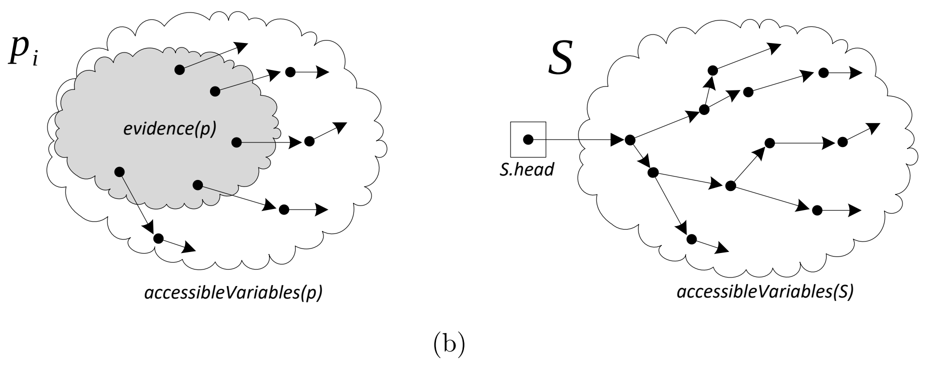

Figure 3 (a) The accessible variables of process p_i are the shared variables that can be accessed through a directed path in the memory graph G_α, starting from a variable in $\mathit{evidence}(p_i, \alpha)$. (b) The accessible variables for a graph-based-set S are the shared variables that can be accessed through a directed path in the graph G_α, starting from $S.\mathit{head}$.

Assume, without loss of generality, that a constant number of primitive types Op_0, $Op_1, \ldots, Op_{t-1}$ is used, and each process applies primitives cyclically in this order during its execution. That is, in its i-th step the process performs a primitive of type $Op_{i \bmod t}$. Any algorithm may be modified to follow this rule, by introducing dummy steps of the required type. This increases the step complexity of the algorithm by a constant factor t, since the algorithm uses a constant number of primitive types, and does not affect the asymptotic step complexity.

Intuitively, the next definition implies that the events of invisible processes accessing the same variable v with the same primitive Op can be ordered in such a way that at most m of these processes are revealed, while the rest of the processes remain invisible.

▶ **Definition 15** (m-Revealing Primitives [3]). Suppose that a set of processes P is invisible in an execution α and consider a variable $v \notin \mathit{evidence}(P, \alpha)$. Suppose that there is a subset $P_k \subseteq P$ of k processes whose next events $\phi_1, \ldots, \phi_k$ apply the same primitive Op to the variable v:

$$\forall p \in P_k : \ \mathit{next\_event}(p, \alpha) = \langle p, v, Op \rangle$$

We say that the primitive Op is *m-revealing*, for $m \geq 0$, if there is a permutation π of the next events $\phi_1, \ldots, \phi_k$ and a subset $P_m \subseteq P_k$ of size $\leq m$, such that $(P \setminus P_m)$ is invisible in $\alpha\pi$.

The next lemma provides the induction step for the lower bound proof, leading to a new invisible set P_{r+1}, a corresponding visible set W_{r+1} and a new state of the graph-based-set S. Our goal is to keep the invisible set P_{r+1} as large as possible, and the graph-based-set S as small as possible, in order to carry out the induction for as long as possible.

▶ **Lemma 16.** *Suppose that there is an execution α_r such that:*

(a) *there is an invisible set P_r of size $|P_r| = m_r \geq 2$ and a corresponding visible set W_r of size $|W_r| = w_r$;*

(b) *each process $p_i \in P_r$ performs exactly r steps in α_r;*

(c) *for every pair of processes $p_i, p_j \in P_r$,*
$\mathit{accessibleVariables}(p_i, \alpha_r) \cap \mathit{accessibleVariables}(p_j, \alpha_r) = \emptyset$;

(d) *for each process $p_i \in P_r$, $|evidence(p_i, \alpha_r)| \le r$ and $|accessibleVariables(p_i, \alpha_r)| \le r(d+1)$ (recall that d is the number of pointers that can be stored in a shared variable);*
(e) *for the* graph-based-set *S, $|accessibleProcesses(S, \alpha_r)| \le r$ and $|accessibleVariables(S, \alpha_r)| = \frac{r(r+1)(d+1)}{2}$.*

Then there is an execution α_{r+1} such that after α_{r+1}:

(a1) *there is an invisible set P_{r+1} of size $m_{r+1} \ge \sqrt{\frac{m_r}{2d+3}}$ and a corresponding visible set W_{r+1} of size $w_{r+1} \le w_r + 2$;*
(b1) *each process $p_i \in P_{r+1}$ performs exactly $r+1$ steps in α_{r+1};*
(c1) *for every pair of processes $p_i, p_j \in P_r$, $accessibleVariables(p_i, \alpha_{r+1}) \cap accessibleVariables(p_j, \alpha_{r+1}) = \emptyset$.*
(d1) *for every process $p_i \in P_{r+1}$, $|evidence(p_i, \alpha_{r+1})| \le r+1$ and $|accessibleVariables(p_i, \alpha_{r+1})| \le (d+1)(r+1)$;*
(e1) *for the* graph-based-set *S, $|accessibleProcesses(S, \alpha_{r+1})| \le r+1$ and $|accessibleVariables(S, \alpha_{r+1})| = \frac{(r+1)(r+2)(d+1)}{2}$.*

Proof. We show how to extend α_r with one more round to obtain an execution α_{r+1}, and a set P_{r+1} invisible in α_{r+1}, such that each process in P_{r+1} performs $(r+1)$ steps in α_{r+1} and properties (a1)—(e1) hold.

By the induction hypothesis (e), $|accessibleProcesses(S, \alpha_r)| \le r$. Hence, at most r processes complete their add(id_i) operations in α_r. Since the rest of the invisible processes $p_i \in (P_r \setminus accessibleProcesses(S, \alpha_r))$ do not complete their add(id_i) operations in α_r, they are poised to execute their next event $\phi_i = next\_event(p_i, \alpha_r)$ in round $r+1$. By assumption and property (b), all the events $\phi_1, \phi_2, \ldots$ of the processes P_r in round $r+1$ apply the same 1-revealing primitive.

Four issues must be addressed in order to keep many processes invisible and S small:

(1) Avoid conflicts between the events of round $r+1$ and the events performed in the previous rounds that may violate the properties of invisible set. Otherwise, if process p_i in round $r+1$ accesses a variable in the evidence set of another process p_j, then p_j cannot be later erased from the execution without affecting the execution of p_i. To simplify the proofs, we require not only the evidence sets of the invisible processes to be disjoint, but also their accessible sets.
(2) Ensure that the *accessibleVariables* of the invisible processes are disjoint (c1). The last two kinds of conflicts are eliminated using inductive hypothesis (c) and applying Turán's Theorem.
(3) Ensure there are no conflicts between events in round $r+1$. These are eliminated by using the properties of 1-revealing primitives.
(4) Keep the size of *accessibleVariables*(p_i) small, for every invisible processes p_i, in order to keep the size of *accessibleVariables*(S) and *accessibleProcesses*(S) small.

(1) Eliminating conflicts with the previous rounds. Consider a *visibility* graph $G(V, E)$, with vertices V corresponding to the processes in P_r. There are two kinds of edges in G (see Figure 4): A *solid* edge $p_i \rightarrow p_j \in E$ exists if process p_i accesses a variable $v_j \in evidence(p_j, \alpha_r)$, $p_i \ne p_j$, in round $r+1$. A *dashed* edge $p_i \dashrightarrow p_j \in E$ exists if a process p_i writes to a variable v_i a *pointer* to a variable $v_j \in accessibleVariables(p_j, \alpha_r)$, $p_i \ne p_j$, in round $r+1$.

We prove that for each process p_i, there is at most one outgoing solid edge in the graph $G_{\alpha_{r+1}}$. By Definition 10(c) of the invisible set P_r, the evidence sets of the processes P_r are disjoint. Therefore variable v_j belongs to evidence set of at most one process $p_j \in P_r$. In

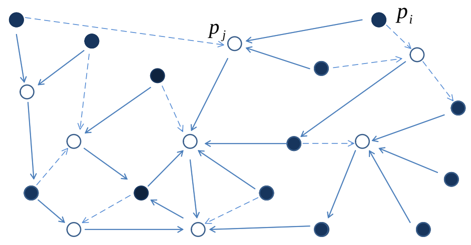

Figure 4 Example of the visibility graph $G(V, E)$ used in the proof of Lemma 16.

round $r+1$, process p_i accesses at most one variable v_j. Therefore, for each process $p_i \in V$ there is at most one outgoing solid edge $p_i \to p_j \in E$.

We also prove that for each process p_i, there are at most d outgoing dashed edges in $G_{\alpha_{r+1}}$. If in round r, process p_i writes to a variable v_i a pointer to a variable $v_j \in \mathit{accessibleVariables}(p_j, \alpha_r)$, for some process $p_j \in P_r$. By inductive hypothesis (c), the *accessibleVariables* of invisible processes are disjoint after α_r. Since each variable contains at most d pointers, there are at most d such processes p_j. This implies that for each process p_i, the graph $G_{\alpha_{r+1}}$ contains at most d outgoing dashed edges $p_i \dashrightarrow p_j \in E$.

Therefore, $|E| \leq (d+1)|V|$ and the average degree of $G_{\alpha_{r+1}}$ (as an undirected graph) is

$$2|E|/|V| \leq 2(d+1)|V|/|V| \leq 2(d+1).$$

Next, we apply the Turán's theorem [15]:

▶ **Theorem 17** (Turán). *Let $G(V, E)$ be an undirected graph, where V is the set of vertices and E is the set of edges. If the average degree of G is r, then $G(V, E)$ has an independent set with at least $\left\lceil \frac{|V|}{r+1} \right\rceil$ vertices.*

It follows that G has an independent set $V' \subseteq V$ with at least $\left\lceil \frac{|V|}{(2d+3)} \right\rceil$ vertices (represented by shaded circles on Figure 4). We leave the processes corresponding to V' in the execution (along with the visible processes W_r that cannot be erased), and erase all the other invisible processes $V \setminus V'$. That is, we define $\alpha'_r = \alpha_r|_{V' \cup W_r}$. By the construction of V', if a process $p \in V'$ is about to access a variable v (in round $r+1$ after α'_r), then this variable is not in the evidence set of any other process in V', and p_i does not write to v a pointer to a variable in the evidence set of another process in V'. Therefore, there are no conflicts between the primitives of round $r+1$ and the primitives of rounds $1, \ldots, r$.

It is still possible that two processes $p, q \in V'$ are about to access the same variable v (not in the evidence set of any one of them) in round $r+1$, violating Properties (a) and (b) of an invisible set (Definition 10). Thus, in order to keep the set of the processes invisible, we should eliminate conflicts between events in round $r+1$. We do this separately for processes that access the variables from $\mathit{accessibleVariables}(S, \alpha_r)$, and the processes that do not access graph-based-set S.

(2) Eliminating conflicts between events in the same round. We next order the events of round $r+1$ for the processes that do not access S. Let $M(V')$ be the set of the variables accessed by the processes in V' in round $r+1$. Note that by the construction of V', if distinct processes $p, q \in V'$ access a variable $v \in M(V')$, then v is not in the evidence set of

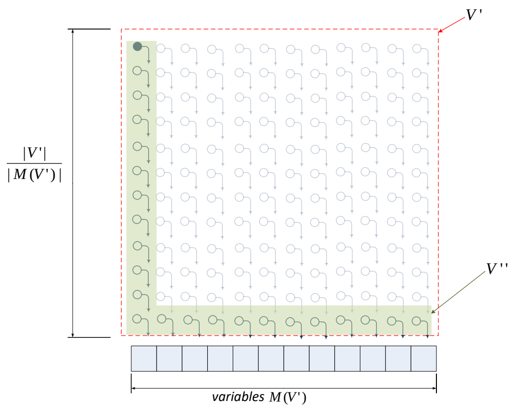

Figure 5 Induction step in the proof of Lemma 16.

any one of them. We use the properties of 1-revealing primitives to keep as many processes in V' invisible as possible, revealing only a small fraction of them. Suppose that after α'_r all processes in V' are about to perform a 1-revealing primitive of type $Op_{(r+1) \bmod t}$.

Let $v_1 \in M(V')$ be the variable that is accessed by the largest number of processes. In round $r+1$, we keep all the processes accessing the variable v_1, and exactly one process for each variable in $M(V') \setminus \{v_1\}$. Let V'' be the set of these processes (Figure 5). We define $\alpha''_r = \alpha'_r|_{V'' \cup W_r}$.

Now we will define the sequence of the next events of processes V'' for round $r+1$. Since $V'' \subseteq P_r$ is invisible in α_r, Lemma 11 implies that V'' is invisible in α''_r. Let $p_{1,1}, p_{1,2}, \ldots, p_{1,l} \in V''$ be the processes that are about to access variable v_1 by performing events $\phi_{1,1}, \phi_{1,2}, \ldots, \phi_{1,l}$. Note that all these events apply the same 1-revealing primitive to v_1. By Definition 15 of a 1-revealing primitive, there is a permutation π_1 of the events $\phi_{1,1}, \phi_{1,2}, \ldots, \phi_{1,l}$ such that after $\alpha''_r\pi_1$, at most one of the processes $p_{1,1}, p_{1,2}, \ldots, p_{1,l}$, say $p(v_1)$, becomes visible while the others remain invisible. By Definition 15, all other processes that are invisible in α''_r are also invisible in $\alpha''_r\pi_1$.

Finally, we schedule the events $\phi_{j_1}, \ldots, \phi_{j_m}$ of the processes that access the variables $M(V') \setminus \{v_1\}$, obtaining the execution $\alpha_{r+1} = \alpha''_r\pi_1\phi_{j_1}, \ldots, \phi_{j_m}$, in which each process in V'' performs exactly $r+1$ steps.

In the execution segment π_1 defined above, only one process, $p(v_1)$, becomes visible, while all other processes remain invisible. Consider the events $\phi_{j_1}, \ldots, \phi_{j_m}$ scheduled at the end of round $r+1$. By the construction, there is exactly one process accessing each of the variables $M(V') \setminus \{v_1\}$, and after $\alpha''_r\pi_1$, none of these variables belongs to evidence set of the processes in V''. Therefore, after $\alpha_{r+1} = \alpha''_r\pi_1\phi_{j_1}, \ldots, \phi_{j_m}$, processes $V'' \setminus \{p(v_1)\}$ form an invisible set, denoted P_{r+1}.

(3) Bounding the size of the invisible set P_{r+1} from below. Since $|V'|$ processes access $|M(V')|$ different variables, on average, $\frac{|V'|}{|M(V')|}$ processes access the same variable. Note that the number of processes that access the variable v_1 is more than the average $\frac{|V'|}{|M(V')|}$. As shown above, all the processes accessing v_1 remain invisible except one process $p(v_i)$.

Therefore, the number of invisible processes after the execution α_{r+1} is

$$|P_{r+1}| = |V''| - 1 \geq \frac{|V'|}{|M(V')|} + |M(V')| - 2 \; .$$

For simplicity of notation, denote $|M(V')| = x$. Using this notation, the number of processes that are invisible after round $r+1$ is $m_{r+1} \geq \frac{|V'|}{x} + x - 2$.

Differentiating by x and equating to 0, we get $m'_{r+1}(x) = -\frac{|V'|}{x^2} + 1 = 0$, implying $x = \sqrt{|V'|}$. Therefore, m_{r+1} is minimized with

$$\frac{|V'|}{\sqrt{|V'|}} + \sqrt{|V'|} - 2 = 2\sqrt{|V'|} - 2 \; .$$

Taking $|V'| = m_r/(2d+3)$ implies $m_{r+1} \geq 2\sqrt{\frac{m_r}{2d+3}} - 2 \geq \sqrt{\frac{m_r}{2d+3}}$, showing property (a1).

(4) Bounding the size of the evidence set and the accessible nodes set. In round $r+1$, each invisible process p_i performs one step at which it accesses at most one variable $v \notin \mathit{evidence}(p_i, \alpha_r)$. Therefore, in round $r+1$, at most one new variable is added to the evidence set of p_i, and the size of the evidence set grows at most by 1, showing that $\mathit{evidence}(p_i, \alpha_{r+1})| \leq r+1$, as required by the first part of property (d1).

By inductive hypothesis (c), no variable $v_i \in \mathit{accessibleVariables}(p_i, \alpha_r) \setminus \mathit{evidence}(p_i, \alpha_r)$ is in the evidence set of another process p_j. Therefore, no such variable v_i is modified by any process in α_r, it remains in its initial state $\perp$ and does not contain a pointer to another variable. This implies that for every variable $v \in \mathit{evidence}(p_i, \alpha_r)$, there are at most d variables $v_i \in (\mathit{accessibleVariables}(p_i, \alpha_r) \setminus \mathit{evidence}(p_i, \alpha_r))$ accessible from v. This implies

$$|\mathit{accessibleVariables}(p_i, \alpha_r) \setminus \mathit{evidence}(p_i, \alpha_r)| \leq d \cdot |\mathit{evidence}(p_i, \alpha_r)|$$

that is, $|\mathit{accessibleVariables}(p_i, \alpha_r)| \leq (d+1) \cdot |\mathit{evidence}(p_i, \alpha_r)| \leq (d+1)(r+1)$, implying the second part of property (d1).

Finally, we show that the set $\mathit{accessibleVariables}(S)$ can be kept relatively small, while keeping enough processes invisible. Consider the processes $P_{S,r} \subseteq P_r$, whose computational events in round $r+1$ access a variable in $\mathit{accessibleVariables}(S, \alpha_r)$.

Let v be the node of graph-based-set S accessed by the largest number of processes in $P_{S,r}$ in round $r+1$. We keep the processes accessing v (denoting them $P_{S,r+1}$), and erase the rest of the processes $(P_r \setminus P_{S,r+1})$ from the execution. Since all the processes $P_{S,r}$ perform the same 1-revealing primitive in round $r+1$, by Definition 15, there is a permutation of the computation events such that only one of these processes, p_i, becomes visible, and the rest remain invisible.

By induction hypothesis (d), $|\mathit{accessibleVariables}(p_i, \alpha_r)| \leq r(d+1)$. Therefore, in round $r+1$, the size of $\mathit{accessibleProcesses}(S, \alpha_r)$ grows by at most 1, and the size of $\mathit{accessibleVariables}(S, \alpha_r)$ grows by at most $r(d+1)$, implying property (e1) (Figure 6). By inductive hypothesis (e),

$$|\mathit{accessibleVariables}(S, \alpha_r)| \leq \frac{r(r+1)(d+1)}{2}$$

Therefore,

$$|P_{S,r+1}| \geq \frac{|P_S|}{|\mathit{accessibleVariables}(S, \alpha_r)|} \geq \frac{2|P_S|}{r(r+1)(d+1)}$$

The number of processes accessing S decreases more slowly than the number of other invisible processes. Property (a1) bounds from below the number of invisible processes after α_{r+1}. ◀

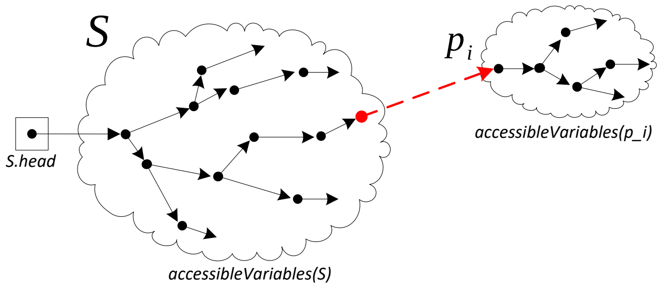

Figure 6 Bounding the size of graph-based-set in Lemma 16.

By Lemma 16, the number of invisible processes after round $r+1$ is $m_{r+1} \geq \sqrt{\frac{m_r}{2d+3}}$.

To prove the lower bound on the amortized step complexity, we start with the empty execution α_0 that satisfies the inductive hypothesis of Lemma 16 with the invisible set P_0 containing all the n processes in the system. Then, we inductively construct execution α_r containing r rounds, using Lemma 16. We require that after α_r there are $|P_r| = r$ invisible processes, i.e., $m_r = r$. Solving this recurrence, we get $m_0 = (2d+3)^{2^r-1} r^{2^r} = \frac{((2d+3)r)^{2^r}}{2d+3}$. Since d is a constant, the total number of the processes in the system should be $n \geq m_0 = \Omega(((2d+3)r)^{2^r})$, or $r = O(\log\log n)$.

By Lemma 16(a1), in each of r rounds of the execution α_r, at most 2 process becomes visible and stopped, and r processes remain invisible after α_r. The rest of the processes are erased from the execution. Therefore, $\dot{c}(\alpha_r) = |P_r| + |W_r| \leq 3r = O(\log\log n)$.

By Lemma 16(e1), at most one process $p_i \in P_r$ completes its operation add(id_i) in round r, and each invisible process takes one step in each round, until it becomes visible. Therefore, the total number of steps performed by the processes P_r in α_r is at least $r|P_r| = r^2 = \Omega(\dot{c}^2)$.

We have constructed an execution of $\dot{c}$ processes that collectively take $\Omega(\dot{c}^2)$ steps. Therefore, the amortized step complexity of an operation add is in $\Omega(\dot{c}^2/\dot{c}) = \Omega(\dot{c})$, provided the contention of the execution is in $O(\log\log n)$. This implies the next theorem:

▶ **Theorem 18.** *Any implementation of* graph-based-set *using any combination of* 1*-revealing primitives has an execution of* $\dot{c}$ *processes, each performing one* add *operation, such that the processes collectively take* $\Omega(\dot{c}^2)$ *steps, implying that the amortized (and hence, worst-case) step complexity of the* add *operation is* $\Omega(\dot{c})$*, provided* $\dot{c} \in O(\log\log n)$*.*

A data structure using a connected set of nodes is a graph-based-set, and therefore, Theorem 18 implies the lower bound also for these data structures.

▶ **Theorem 19.** *Any implementation of linked lists, skip lists, search trees and other data structures based on* graph-based-set*, using any constant set of* 1*-revealing primitives, has an execution of* $\dot{c}$ *processes, each performing one* add *operation, such that the processes collectively take* $\Omega(\dot{c}^2)$ *steps. Therefore, the amortized (and hence, worst-case) step complexity of the implementation is* $\Omega(\dot{c})$*, provided* $\dot{c} \in O(\log\log n)$*.*

Acknowledgements. We would like to thank the referees for many helpful comments.

References

1 James H Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 16(2):75–110, 2003.

2 Richard J. Anderson and Heather Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 370–380, New York, NY, USA, 1991. ACM. `doi:10.1145/103418.103458`.

3 Hagit Attiya and Arie Fouren. Poly-logarithmic adaptive algorithms require revealing primitives. *Journal of Parallel and Distributed Computing*, 109:102–116, 2017. `doi:10.1016/j.jpdc.2017.05.010`.

4 Rudolf Bayer and Mario Schkolnick. Concurrency of operations on b-trees. *Acta informatica*, 9(1):1–21, 1977.

5 Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 332–340, New York, NY, USA, 2014. ACM. `doi:10.1145/2611462.2611486`.

6 Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 50–59, New York, NY, USA, 2004. ACM. `doi:10.1145/1011767.1011776`.

7 Timothy Harris. A pragmatic implementation of non-blocking linked-lists. *Distributed Computing*, pages 300–314, 2001.

8 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th international conference on Principles of Distributed Systems*, pages 3–16. Springer-Verlag, 2005.

9 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2011.

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

11 Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

12 Eric Ruppert. Analysing the average time complexity of lock-free data structures. In *Workshop on Complexity and Analysis of Distributed Algorithms*, 2016. [`http://www.birs.ca/events/2016/5-day-workshops/16w5152/videos/watch/201612011630-Ruppert.html`; accessed 22-Mar-2017].

13 Niloufar Shafiei. *Non-Blocking Data Structures Handling Multiple Changes Atomically.* PhD thesis, York University, 2015.

14 R. Kent Treiber. *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

15 P. Turan. On an extremal problem in graph theory (in Hungarian). *Mat. Fiz. Lapok*, 48:436–452, 1941.

16 John D Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM, 1995.

A Extending the Kim-Anderson Lower Bound

The $\Omega(\dot{c})$ lower bound proof on the worst-case step complexity for adaptive mutual exclusion [10], can be extended to yield an $\Omega(\dot{c})$ lower bound on the *amortized* step complexity.

Algorithm 1 Adaptive mutual exclusion using a set object and CAS

```
$waitingRoom$: a sack object, initially empty
$promoted$ : a RW register, initially $\perp$
$fastLane$: CAS register, initially EMPTY
$gate$: CAS register, initially OPEN
$privateGate[0, \dots n-1]$: initially CLOSED
$wants[0, \dots, n-1]$ : initially FALSE

procedure enterCS($id$)
    $wants[id]$ = TRUE                        ▷ announce that the process needs to enter CS
    $waitingRoom$.put($id$)                   ▷ add itself to the set of the waiting processes
    if CAS ($fastLane$, EMPTY, ⟨OCCUPIED, $id$⟩) then          ▷ try to occupy the fast lane
        continue   ▷ $p_i$ successfully passed the fast lane, continue to the $gate$ mutual exclusion
    else
        wait until $privateGate[id]$ == OPEN                   ▷ wait until another process
                                                              ▷ will open $p_i$'s private gate
    end if
    wait until CAS ($gate$, OPEN, CLOSED)           ▷ try to win 2-process mutual exclusion
end procedure

procedure exitCS($id$)
    $wants[id]$ = FALSE
    $privateGate[id]$ = CLOSED                                              ▷ clean-up
    CAS ($fastLane$, ⟨OCCUPIED, $id$⟩, EMPTY)      ▷ if the fast lane occupied by $p_i$, release it
    if $promoted = id$ then                               ▷ if $p_i$ was the promoted process
        $promoted = \perp$                                  ▷ the reset $promoted$ to $\perp$
    end if
    if $promoted = \perp$ then                   ▷ if there is no promoted process, promote one
        do $next$ = $waitingRoom$.draw()                 ▷ remove some process from the sack
        until ($wants[next]$==TRUE or $next == \perp$)      ▷ until the removed process is interested
                                                ▷ to enter CS, or until $waitingRoom$ is empty
        if $wants[next]$==TRUE then                     ▷ process $next$ is interested to enter CS
            $promoted = next$                                                ▷ promote $next$
            $privateGate[next]$ = OPEN               ▷ signal to $next$ by opening its private gate
        end if
    end if
    $gate$ = OPEN                                   ▷ release the 2-process mutual exclusion
end procedure
```

In each round of the proof of [10, Lemma 7], at most two processes are added to the set of *finished* processes $Fin(H)$. Before a process p is added to $Fin(H)$ in round j, it performs j steps. The execution H has k rounds, and there is at least one process that is not in $Fin(H)$ at the end of round k; this process performs at least k steps. Let x be the number of the processes in $Fin(H)$ at the end of the execution: $x = |Fin(H)|$. The processes $Fin(H)$ are simultaneously active in H, therefore $\dot{c}(H) \geq x$. The total number of steps performed by all processes in H is equal to the number of steps performed by the processes $Fin(H)$, which is at least $\sum_{j=1}^{x/2} 2j \geq x^2/4$, plus k steps performed by the last process. Thus, the total number of steps is $x^2/4+k$, and the amortized step complexity is $t(x) = (x^2/4+k)/(x+1) = \Omega(x) = \Omega(\dot{c}(H))$. Differentiating the last expression by x and equating to 0, we get that the amortized step complexity is minimized at $\min t(x) = \frac{1}{2}\left(\sqrt{4k+1}-1\right)$ for $x = \sqrt{4k+1}-1$. That is, the construction guarantees amortized step complexity $t(x) \geq x/2 = \Omega(\dot{c}(H))$.

The maximal number of processes in the system required for the construction is $N(k) = (2k+4)^{2(2^k-1)}$, implying $k = O(\log\log N)$. Since $x = O(\sqrt{k})$, we have that $x = O(\sqrt{\log\log N})$.

This gives the next observation:

▶ **Theorem 20.** *For any given x, there is an execution H, with point contention $\dot{c}(H) \geq x$ and RMR amortized step complexity $amortizedStep(H) = \Omega(\dot{c}(H))$, provided that $x = O(\sqrt{\log \log N})$, where N is the total number of the processes in the system.*