

On the Efficiency of Dynamic Transaction Scheduling in Blockchain Sharding

Ramesh Adhikari ✉️ 🏠 

School of Computer and Cyber Sciences, Augusta University, GA, USA

Costas Busch ✉️ 

School of Computer and Cyber Sciences, Augusta University, GA, USA

Miroslav Popovic ✉️ 

Faculty of Technical Sciences, University of Novi Sad, Serbia

Abstract

Sharding is a technique to speed up transaction processing in blockchains, where the n processing nodes in the blockchain are divided into s disjoint groups (shards) that can process transactions in parallel. We study dynamic scheduling problems on a shard graph G_s where transactions arrive online over time and are not known in advance. Each transaction may access at most k shards, and we denote by d the worst distance between a transaction and its accessing (destination) shards (the parameter d is unknown to the shards). To handle different values of d , we assume a locality sensitive decomposition of G_s into clusters of shards, where every cluster has a *leader shard* that schedules transactions for the cluster. We first examine the simpler case of the *stateless model*, where leaders are not aware of the current state of the transaction accounts, and we prove a $O(d \log^2 s \cdot \min\{k, \sqrt{s}\})$ competitive ratio for latency. We then consider the *stateful model*, where leader shards gather the current state of accounts, and we prove a $O(\log s \cdot \min\{k, \sqrt{s}\} + \log^2 s)$ competitive ratio for latency. Each leader calculates the schedule in polynomial time for each transaction that it processes. We show that for any $\epsilon > 0$, approximating the optimal schedule within a $(\min\{k, \sqrt{s}\})^{1-\epsilon}$ factor is NP-hard. Hence, our bound for the stateful model is within a poly-log factor from the best possibly achievable. To the best of our knowledge, this is the first work to establish provably efficient dynamic scheduling algorithms for blockchain sharding systems.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms; Theory of computation → Scheduling algorithms

Keywords and phrases Blockchain, Blockchain Sharding, Dynamic Transaction Scheduling

Digital Object Identifier 10.4230/LIPIcs.DISC.2025.2

Related Version *Extended Version*: <https://arxiv.org/abs/2508.07472> [4]

Funding This paper is supported by NSF grant CNS-2131538.

1 Introduction

Blockchains are known for their special features, such as fault tolerance, transparency, non-repudiation, immutability, and security, and have been used in various applications and domains [16]. However, a drawback of blockchains is that the size of the blockchain network may impact the latency and throughput of transaction processing. To append a new block in a blockchain network, the participating nodes reach consensus, which is a time and energy-consuming process [2]. Moreover, each node is required to process and store all transactions, which leads to scalability issues in the blockchain system. *Sharding* protocols have been proposed to address the scalability and performance issues of blockchains [15, 21, 11, 1], which divide the overall blockchain network into smaller groups of nodes called *shards* that allow for processing transactions in parallel. In the sharded blockchain, independent transactions are processed and committed in multiple shards concurrently, which improves the blockchain



© Ramesh Adhikari, Costas Busch, and Miroslav Popovic;
licensed under Creative Commons License CC-BY 4.0

39th International Symposium on Distributed Computing (DISC 2025).

Editor: Dariusz R. Kowalski; Article No. 2; pp. 2:1–2:23



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

system's throughput. However, most of the existing sharding protocols [15, 13, 21, 1] do not provide formal analysis for the scheduling time complexity (i.e. how fast the transactions can be processed).

We consider a blockchain system consisting of n nodes, which are further divided into s shards, where each shard consists of n/s nodes. Shards are connected in a graph network G_s with a diameter D , and each shard holds a subset of the objects (transaction accounts). We assume that transactions are distributed across the shards, and each transaction accesses at most k accounts. A transaction T_i initially is in one of the shards, which is called the *home shard* for T_i . For simplicity, we consider each shard has one transaction at a time, and when that transaction gets processed (either commit or abort), a new transaction will be generated at the home shard. Similar to other sharding systems [11, 1, 2], each transaction T_i is split into subtransactions, where each subtransaction accesses an account. A subtransaction of T_i is sent to the *destination shard* that holds the respective account. We assume that the maximum distance between the home shard of a transaction and the respective destination shards in G_s is at most $d \leq D$. (The parameter d is not known to the system.)

All home shards process transactions concurrently. A problem occurs when *conflicting* transactions try to access the same account simultaneously. In such a case, the conflict prohibits the transactions from being committed concurrently and forces them to serialize [2]. Our proposed *scheduling algorithms* coordinate the home shards and destination shards to process the transactions (and respective subtransactions) in a conflict-free manner in polynomial time. Each destination shard maintains a local blockchain of the subtransactions that are sent to it. The global blockchain can be constructed (if needed) by combining the local blockchains at the shards [1].

We consider online dynamic transaction scheduling problem instances where transactions are not known a priori. Moreover, transactions may arrive online and continuously over time, which are generated by electronic devices or some crypto app that resides on shards. Our proposed schedulers determine the time step for each transaction $T_i \in \mathcal{T}$ to process and commit. The execution of our scheduling algorithm is partially synchronous, where communication delay is upper bounded by a system parameter. The goal of a scheduling algorithm is to efficiently process all transactions while minimizing the total execution time (makespan). Unlike previous sharding approaches [11, 13, 21], our scheduling algorithms are lock-free, namely, they do not require locking mechanisms for concurrency control.

We use the notion of *competitive ratio* [8] to determine the performance of our scheduling algorithms. The competitive ratio typically measures how well a given online algorithm performs compared to the best possible offline algorithm for a specific sequence of operations. However, in our model, the transactions generated in the future depend on the execution history. Hence, we define the competitive ratio to capture the volatile transaction history.

Contributions. To our knowledge, this is the first work to present provably efficient online transaction scheduling algorithms for blockchain sharding systems. We summarize our contributions as follows (also see Table 1):

- **Stateless Scheduling Model:** We first provide transaction scheduling algorithms for the stateless model, where a *leader shard* that is responsible for coordinating transaction execution, does not require knowledge of the current state of the accessed accounts. In this model, we provide two scheduling algorithms:
 - **Single-Leader Scheduler:** In this scheduling algorithm, one of the shards acts as the leader and all other shards send their transaction information to this leader, which determines the global transaction schedule. Our algorithm works in a partially

synchronous communication model, but for the sake of performance analysis purposes, we assume a synchronous model. Let the shard network be represented as a general graph G_s , where each transaction accesses at most k objects (shards). The maximum distance between home shards, accessed shards, and leader is denoted with d . Then, the single-leader scheduler achieves an $O(d \cdot \min\{k, \sqrt{s}\})$ competitive ratio with respect to the optimal scheduler. In the special case where G_s is a clique with unit distances (i.e., $d = 1$), the competitive ratio becomes $O(\min\{k, \sqrt{s}\})$.

- **Multi-Leader Scheduler:** A drawback of the single-leader case is that the distance d involves also the position of the leader. On the other hand, in the multi-leader case, d only involves distances between home and respective destination shards. In this scheduler, multiple leaders process the transactions, which distribute the scheduling load among multiple shards. The multi-leader approach allows for a better adaptation to the value d without requiring knowledge of d and without involving distances to the leaders in the definition of d . This approach uses a hierarchical clustering technique [10] to cluster the shard network, which enables the independent scheduling and commitment of transactions within different clusters. This scheduler achieves a competitive ratio of $O(d \log^2 s \cdot \min\{k, \sqrt{s}\})$.
- **Stateful Scheduling Model:** We next consider a stateful model where the leader shard requires knowledge of the account states. Namely, a leader shard receives the transactions from the home shards (where transactions are initially generated), and then the leader shard first gathers the current state of the accounts from their corresponding account shards before scheduling and pre-committing the transactions. After receiving the state, the leader pre-commits the transactions locally and forwards the pre-committed batch to the destination shards. In this model, the single-leader scheduler achieves a competitive ratio of $O(\min\{k, \sqrt{s}\})$ and the multi-leader scheduler achieves a competitive ratio of $O(\log s \cdot \min\{k, \sqrt{s}\} + \log^2 s)$. Note that these competitive ratios do not depend on d (in contrast to the stateless model), which is the benefit of the stateful approach.
- **Approximation Hardness:** We also show that for any $\epsilon > 0$, obtaining competitive ratio $(\min\{k, \sqrt{s}\})^{1-\epsilon}$ is NP-hard. Hence, our bound for the stateful single-leader scheduler is asymptotically the best we can achieve in polynomial time, and the bound for the stateful multi-leader scheduler is within a poly-log factor of the best achievable.
- **Safety and Liveness Analysis:** We formally analyze the correctness of our proposed schedulers by proving both safety and liveness for the single-leader and multi-leader algorithms.

Paper Organization. The rest of this paper is structured as follows. Section 2 provides related works. Section 3 describes the preliminaries for this study and the sharding model. Section 4 presents a stateless scheduling model with single-leader and multi-leader scheduling algorithms. In Section 5, we provide the stateful single-leader and multi-leader scheduling algorithms. Finally, we give our conclusions in Section 6. Due to space limitations, some proofs are deferred to the appendix, while additional correctness analyses (safety and liveness) of our algorithms are provided in the extended version available on arxiv [4].

2 Related Work

To solve the scalability issue of blockchain, various sharding protocols [15, 21, 11, 17, 20, 6] have been proposed. These protocols have shown promising enhancements in the transaction throughput of blockchain by processing transactions in parallel in multiple shards. However,

■ **Table 1** Comparison of our proposed online transaction scheduling algorithm's competitive ratio with related works [2, 3, 14]. The used notations are as follows: s represents a total number of shards, k denotes the maximum number of shards (objects) accessed by each transaction, d denotes the worst distance between any transaction (home shard) and its accessed objects (destination shard), b denotes the burstiness and c'_1 represents some positive constant. (Note that the bounds in [2] are the actual transaction latencies).

	Proposed Results		Related works	
	Stateless Model	Stateful Model	In [2]	In [3, 14]
Problem	Dynamic Transaction	Dynamic Transaction	Dynamic Transaction	Batch Transaction
Focus	Performance	Performance	Stability	Performance
Single Leader	$O(d \cdot \min\{k, \sqrt{s}\})$	$O(\min\{k, \sqrt{s}\})$	$36bd \cdot \min\{k, \lceil \sqrt{s} \rceil\}$	$O(kd)$
Multi-Leader	$O(d \log^2 s \cdot \min\{k, \sqrt{s}\})$	$O(\log s \cdot \min\{k, \sqrt{s}\} + \log^2 s)$	$2 \cdot c'_1 bd \log^2 s \cdot \min\{k, \lceil \sqrt{s} \rceil\}$	$O(kd \cdot \log d \log s)$
Com. Model	Partial-synchronous	Partial-synchronous	Synchronous	Synchronous

none of these protocols have specifically explored the theoretical analysis of online transaction scheduling problems in a sharding environment. To process transactions in parallel in the sharding model, some research work [13, 11] has used two-phase locking for concurrency control. However, locks are expensive because when one process locks shared data for reading/writing, all other processes attempting to access the same data set are blocked until the lock is released, which lowers system throughput. Moreover, locks, if not handled and released properly, may cause deadlocks. Our scheduling algorithms do not use locks, as concurrency control is managed by scheduling non-conflicting transactions in parallel. In [1] the authors propose lockless blockchain sharding using multi-version concurrency control. However, they lack a performance analysis, and they do not explore the benefits of locality and optimization techniques for transaction scheduling.

In a recent work [2] (see Table 1), the authors provide a stability analysis of blockchain sharding considering adversarial transaction generation. Their focus is on stability, not on performance, where they want to maintain a bounded pending transaction queue size and latency. They consider adversarial transaction generation, where at any time interval of duration t , the number of generated transactions using any object is bounded by $\rho t + b$, where $\rho \leq 1$ is the transaction injection rate per unit time and $b > 0$ is a burstiness injection parameter. They consider stateless scheduling model, and for the single leader scheduler where the shards are connected in the clique graph with unit distance they provide the stable transaction rate $\rho \leq \max\{\frac{1}{18k}, \frac{1}{18\sqrt{s}}\}$, for which they show the number of pending transactions at any round is at most $4bs$ (which is the upper bound on queue size in each shard), and the latency of transactions is bounded by $36b \cdot \min\{k, \lceil \sqrt{s} \rceil\}$. If this single leader scheduler is considered in the general graph where the transaction and its accessing object are d far away, then their latency becomes $36bd \cdot \min\{k, \lceil \sqrt{s} \rceil\}$. Similarly, for a multi-leader scheduler, they provide a stable transaction injection rate $\rho \leq \frac{1}{c'_1 d \log^2 s} \cdot \max\{\frac{1}{k}, \frac{1}{\sqrt{s}}\}$, where c'_1 is some positive constant. For this scheduling algorithm, they show the upper bound on queue size as $4bs$, and transaction latency as $2 \cdot c'_1 bd \log^2 s \cdot \min\{k, \lceil \sqrt{s} \rceil\}$. However, they consider a synchronous communication model, which is not practical in blockchain, and they also do not provide a theoretical analysis of the optimal approximation for the scheduling algorithm, and they only consider a stateless scheduling model. All their latency bounds depend on the burstiness parameter b , which can be arbitrarily large, as it expresses a transaction injection burst of arbitrary size in any given time interval. On the other hand, our system models do not depend on any burstiness parameter, as we adopt a transaction injection model tuned for performance analysis.

In [3, 14] (see Table 1), the authors presented batch scheduling algorithms (for a given set of transactions) while they did not consider dynamic transaction generation. Moreover, their provided bounds are not tight even for batch processing. Furthermore, their algorithms work on a synchronous communication model, which might not be applicable in a real-world distributed blockchain network. The authors in [14] only consider single leader algorithms and have worse performance complexity bounds than [3] by a factor of $\log D$, resulting in a complexity of $O(kd \cdot \log D)$ whereas [3] achieves $O(kd)$ approximation for batch transactions. Here, we provide efficient scheduling algorithms with theoretical analysis for dynamic transaction processing in a blockchain sharding system that works in the partially synchronous communication model.

Several works have been conducted on transaction scheduling in shared memory multi-core systems, distributed systems, and transactional memory [7, 8]. In [5, 18, 19], the authors explored transaction scheduling in distributed transactional memory systems aimed to achieve better performance bounds with low communication costs. In [7] they provide offline scheduling for transactional memory, where each transaction attempts to access an object, and once it obtains the object, it executes the transaction. In another work [8], the authors extended their analysis from offline to online scheduling for the transactional memory in a synchronous communication model. However, these works do not address transaction scheduling problems in the context of blockchain sharding. This is because, in the transactional memory model, the considered system models assume that objects are mobile, and once a transaction obtains the object, it immediately executes the transaction. In contrast, in blockchain sharding, an object is static in a shard, and there is a confirmation scheme to confirm and commit each subtransaction consistently in the respective shard.

3 Technical Preliminaries

3.1 Blockchain Sharding Model

We consider a blockchain sharding model similar to [11, 1, 2, 3], consisting of n nodes which are partitioned into s shards $S_1, S_2, \dots, S_s$ such that $S_i \subseteq \{1, \dots, n\}$, for $i \neq j$, $S_i \cap S_j = \emptyset$, $n = \sum_i |S_i|$, and $n_i = |S_i|$ denotes the number of nodes in shard S_i . Let $G_s = (V, E, w)$ denote a weighed graph of shards, where $V = \{S_1, S_2, \dots, S_s\}$, the edges E correspond to the connections between the shards, and the weight function w represents the distance between the shards. The graph G_s is complete, since each pair of shards can communicate directly, but the weights of the edges may be non-uniform.

Each shard maintains a local blockchain (which is part of the global blockchain) according to its local accounts and the subtransactions it receives and commits. We use f_i to represent the number of Byzantine nodes in shard S_i . To guarantee consensus on the current state of the local blockchain, we assume that every shard executes the PBFT [9] or a similar consensus algorithm. In order to achieve Byzantine fault tolerance, we assume each shard S_i consists of $n_i > 3f_i$ nodes.

We assume that shards communicate with each other via message passing [11], and here, we are not focusing on optimizing the message size. We adopt the cluster-sending protocol described in [12] and Byshard [11], where shards run consensus (e.g., the PBFT [9] consensus algorithm within the shard) before sending a message. For communication between shards S_1 and S_2 , a set $A_1 \subseteq S_1$ of $f_1 + 1$ nodes in S_1 and a set $A_2 \subseteq S_2$ of $f_2 + 1$ nodes in S_2 are chosen (where f_i is the number of faulty nodes in shard S_i). Each node in A_1 is instructed to broadcast the message to all nodes in A_2 . Thus, at least one non-faulty node in S_1 will send

the correct message value to a non-faulty node in S_2 . (Actually, A_1 needs to have size $2f_1 + 1$ to distinguish the correct message.) We consider a partial-synchronous communication model, where sending messages for transactions to their accessing shards has a bounded delay.

Suppose we have a set of shared accounts $\mathcal{O}$ (which we also call *objects*). Similar to previous works in [11, 1, 2], we assume that each shard is responsible for a specific subset of the shared objects (accounts). To be more specific, $\mathcal{O}$ is split into disjoint subsets $\mathcal{O}_1, \dots, \mathcal{O}_s$, where the set of accounts under the control of shard S_i is represented by $\mathcal{O}_i$. Every shard S_i keeps track of local subtransactions that use its corresponding objects in $\mathcal{O}_i$.

3.2 Transactions and Subtransactions

Similar to the works in [11, 2, 3], we consider transactions $\{T_1, T_2, \dots\}$ that are distributed across different shards. Suppose that transaction T_i is generated in a node v_{T_i} within the system, then the *home shard* of T_i is the shard containing v_{T_i} . In this work, we consider transactions that are continuously generated over time. For simplicity and to attain a performance analysis, we assume that each home shard contains at most one transaction at any moment of time, and after the transaction gets processed (either commits or aborts), a new transaction is generated on that home shard.

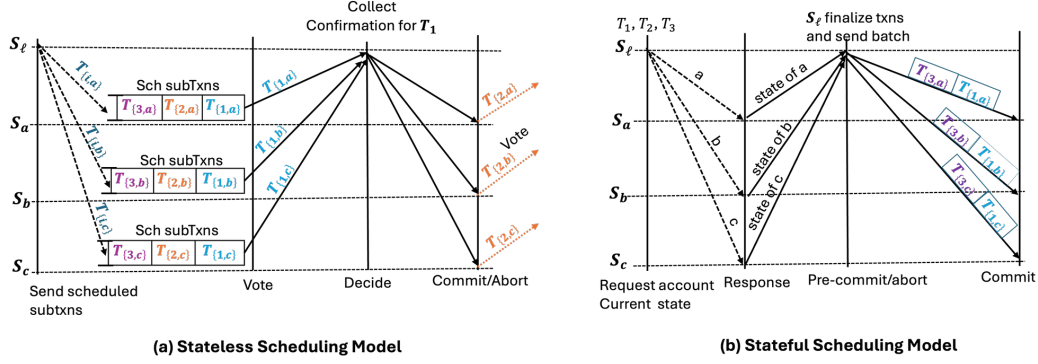
Similar to work in [11, 1, 2], we define a transaction T_i as a group of subtransactions $T_{i,a_1}, \dots, T_{i,a_j}$. Each subtransaction T_{i,a_l} accesses objects only in $\mathcal{O}_{a_l}$ and is associated with shard S_{a_l} . Therefore, each subtransaction T_{i,a_l} has a respective *destination shard* S_{a_l} . The home shard sends the transaction T_i to the leader shard S_ℓ , which is responsible for processing transaction T_i . Then the leader shard of T_i sends subtransaction T_{i,a_l} to shard S_{a_l} for processing, where it is appended to the local blockchain of S_{a_l} . The subtransactions within a transaction T_i are independent, meaning they do not conflict and can be processed concurrently. An example of transactions and subtransactions is deferred to Appendix A.1.

3.3 Stateless and Stateful Scheduling Models

We define two scheduling models to schedule and process the transactions, the stateless and stateful models, which we describe as follows.

Stateless Scheduling Model. Let's suppose there is a designated *leader shard* S_ℓ that coordinates the scheduling and processing of transactions. In this model, the leader shard S_ℓ does not maintain the current state of accounts accessed by the transactions [2, 3, 11]. Upon receiving transactions, S_ℓ constructs (or extends) a transaction conflict graph and colors the graph using an incremental greedy vertex coloring algorithm to determine the commit order for each transaction. Then the leader S_ℓ splits each transaction into subtransactions based on accessed accounts and sends them to the corresponding *destination shards* that hold the relevant account states. Each destination shard maintains the scheduled subtransactions queue sch_{dq} and it picks one color subtransaction from the head of sch_{dq} , validates the sub-transactions (e.g., checking account balances) and sends a *commit* or *abort* vote to the leader. After collecting all votes for a transaction, the leader sends a final decision to each destination shard, which either commits or aborts the subtransactions according to the message received from the leader shard.

For example, suppose S_ℓ receives transactions T_1, T_2, T_3 , each accessing accounts a, b, c , located in shards S_a, S_b, S_c respectively (see Figure 1 (a)). The leader constructs a conflict graph $G_{\mathcal{T}_\ell}$ and applies a greedy vertex coloring algorithm to define a commit order. It then



■ **Figure 1** Illustration of Stateless (a) and Stateful (b) Scheduling Models.

splits transactions into sub-transactions:

$$T_1 \rightarrow \{T_{1,a}, T_{1,b}, T_{1,c}\}, \quad T_2 \rightarrow \{T_{2,a}, T_{2,b}, T_{2,c}\}, \quad T_3 \rightarrow \{T_{3,a}, T_{3,b}, T_{3,c}\}$$

Each destination shard queues the received sub-transactions in a schedule queue sch_{dq} according to the commit order received from S_ℓ , and it processes one color subtransaction at a time. This means S_a picks $T_{1,a}$, S_b picks $T_{1,b}$ and S_c picks $T_{1,c}$ from head of their queues, check the validity and condition of the subtransaction (such as account balance) and send either commit or abort votes to the leader shard. Then the transaction T_i and its subtransactions ($T_{1,a}, T_{1,b}$ and $T_{1,c}$) are committed or aborted based on the final decision received from the leader shard. Next, each destination shard processes the next color subtransactions, for instance $T_{2,a}$, from S_a , $T_{2,b}$ from S_b , and $T_{2,c}$ from S_c (see Figure 1 (a)), and this process repeats.

Stateful Scheduling Model. In the stateful model, the home shard where a transaction is initially generated sends its transaction information to the leader shard S_ℓ . Then the leader shard S_ℓ stores these transactions (i.e. T_1, T_2, T_3) in its pending transaction queue PQ_ℓ . Then, the leader shard identifies accounts accessed by transactions and requests their state from corresponding shards S_a, S_b, S_c . In other words, before processing the transactions, the leader collects the current state of all accessed accounts from the corresponding destination shards. Once the account states are gathered, the leader constructs a conflict graph on which it applies the incremental greedy vertex coloring algorithm. Then the leader shard performs local *pre-commit* for valid transactions (e.g., T_1, T_3) and aborts invalid transactions (e.g., T_2). After that, S_ℓ creates the pre-committed sub-transaction batches: $S_a : \{T_{1,a}, T_{3,a}\}$, $S_b : \{T_{1,b}, T_{3,b}\}$, $S_c : \{T_{1,c}, T_{3,c}\}$ for each destination shard S_a, S_b, S_c . Then these pre-committed batches are sent to the respective destination shards. Since the transactions have already been validated, each destination shard can directly commit and append the received pre-committed order to its local blockchain without further interaction with the leader.

The main difference between the stateless and stateful model is that the stateful model requires the leader to be updated about account states which are at remote shards, while the stateless model does not require leader to be informed about remote accounts.

3.4 Conflicts and Competitive Ratio

Two transactions conflict if they attempt to access the same account, and at least one of the two updates the account. The subtransactions are processed sequentially at each destination shard. For this reason, we extend the notion of conflict to all transactions that access account in the same destination shard.

► **Definition 1 (Conflict).** *Transactions T_i and T_j are said to conflict if they access accounts on the same destination shard S_k and at least one of these transactions writes (updates) the account in S_k .*

Transactions that conflict should be processed in a sequential manner to guarantee atomic object update. In such a case, their respective subtransactions should be serialized in the exact same order in every involved shards. To resolve the conflict between two transactions T_i and T_j while accessing the same destination shard S_k , a scheduler must schedule them one after another in such a way that T_i commits before T_j or vice versa. To perform the schedule, we use a *conflict graph* such that the nodes are transactions, and an edge represents a conflict between two transactions.

We continue with the definition of competitive ratio for our scheduling models. The definition below is an adaptation of the competitive ratio used in dynamic execution in software transactional memory [8]. Since the future transactions depend on the past execution, we define the competitive ratio based on any set of transactions that may appear at any moment of time. Consider a transaction schedule S . Let $\mathcal{T}_t$ denote the set of all pending transactions (that have not committed or aborted) at time t . Let $t' > t$ denote the time such that all transactions in $\mathcal{T}_t$ finalize (commit or abort). Let τ^* denote the optimal time duration to finalize (commit or abort) all the transactions in $\mathcal{T}_t$ if they were the only transactions in the system, processed as a batch. The approximation ratio for S at time t is $r_S(t) = (t' - t)/\tau^*$. The competitive ratio for S is $r_S = \sup_t r_S(t)$.

► **Definition 2 (Algorithm Competitive Ratio).** *For online scheduling algorithm $\mathcal{A}$, the competitive ratio $r_{\mathcal{A}}$ is the maximum competitive ratio over all possible schedules S that it produces, $r_{\mathcal{A}} = \sup_{S \in \mathcal{S}} r_S$. (We also say that $\mathcal{A}$ is $r_{\mathcal{A}}$ -competitive.)*

4 Stateless Scheduler

In this section, we consider the stateless sharding model [1, 2, 11], where the leader shard is responsible for coordinating transaction processing and does not gather the current state of account information (see Section 3.3). We present two transaction scheduling algorithms: the Single-Leader Scheduler and the Multi-Leader Scheduler.

4.1 Stateless Single-Leader Scheduler

In this section, we describe and analyze the *Stateless Single-Leader Scheduler*, which operates under a partially synchronous communication model. We assume a designated leader shard S_ℓ responsible for determining the transaction schedule. All shards send their transactions to the leader shard, which builds a transaction conflict graph and applies an incremental greedy vertex coloring algorithm to determine a schedule.

The algorithm follows an event-driven approach to schedule and process the transactions. When a new transaction T_i is generated at its home shard S_i , then the home shard tags the current timestamp to the transaction T_i and sends the transaction to the leader shard S_ℓ . Upon receiving T_i , the leader adds it to the local transaction set $\mathcal{T}_\ell$ and extends the

Algorithm 1 STATELESS SINGLE LEADER SCHEDULER.

```

1 txn: transaction; txns: transactions; subTxn: subtransaction; subTxns: subtransactions;
2  $T_i$ : txn,  $T_{i,j}$ : subTxn of  $T_i$  for shard  $S_j$ ,  $\mathcal{T}_\ell$ : Set of txns maintained by leader shard  $S_\ell$ ;  $is\_busy$ :
   processing flag (initially false at each shard); Each shard  $S_j$  maintains a lexicographically ordered
   scheduled queue  $sch_{dq}$  for subtransactions;
3 Upon generation of a new txn  $T_i$  at home shard  $S_i$ 
4    $S_i$  tags local timestamp (ts) to  $T_i$  and send it to the leader shard  $S_\ell$ ;
5 Upon receiving new txn  $T_i$  at leader shard  $S_\ell$ 
6    $S_\ell$  adds  $T_i$  to txns set  $\mathcal{T}_\ell$  and extend transaction conflict graph  $G_{\mathcal{T}_\ell}$  with  $T_i$ ;
7   If any colored txn  $T_x$  exists with  $ts(T_x) > ts(T_i)$ , cancel its color, prioritize  $T_i$ , and send cancel
   message for  $T_x$  to corresponding destination shards;
8   Run incremental greedy coloring on  $G_{\mathcal{T}_\ell}$  without altering already scheduled (colored) old txns;
9   Split each newly colored  $T_i$  into subtxns  $T_{i,j}$  and send to respective destination shard  $S_j$ ;
10 Upon receiving subtransaction  $T_{i,j}$  at each destination shard  $S_j$ 
11   Append  $T_{i,j}$  in  $sch_{dq}$  and order (sort)  $sch_{dq}$  lexicographically according to color;
12   if  $is\_busy == false$  then
13     Set  $is\_busy = true$ ;
14     Let  $T_{i,j} \leftarrow$  head of  $sch_{dq}$ ; If  $T_{i,j}$  is valid and local conditions satisfied, it sends commit vote to
     leader shard  $S_\ell$ ; Otherwise, it sends abort vote to  $S_\ell$ ;
15 Upon receiving votes for txn  $T_i$  at leader shard  $S_\ell$ 
16   If any abort vote receive for  $T_i$  then it sends confirmed abort to all corresponding  $S_j$  of  $T_i$ ; else if all
   received votes are commit votes, then it sends confirmed commit to corresponding  $S_j$ ;
17   Remove  $T_i$  from  $\mathcal{T}_\ell$  and  $G_{\mathcal{T}_\ell}$  and send outcome(committed or aborted) to home shard of  $T_i$ ;
18 Upon receiving confirmation for subtxn  $T_{i,j}$  at each destination shard  $S_j$ 
19   If the confirmed commit is received, then it commit  $T_{i,j}$  and append to its local blockchain;
20   Otherwise, if confirmed abort message received then it abort  $T_{i,j}$ ;
21   If  $sch_{dq}$  is not empty, it start to process next subTxn from  $sch_{dq}$ , else it set  $is\_busy = false$ ;
22 Upon receiving cancel message for  $T_{x,j}$  at destination shard  $S_j$ 
23   Remove  $T_{x,j}$  from  $sch_{dq}$ ; a new color will be received later for  $T_{x,j}$  from leader shard  $S_\ell$ ;
24 Upon receiving outcome of  $T_i$  at home shard  $S_i$ 
25   Generate next transaction and repeat process;

```

conflict graph $G_{\mathcal{T}_\ell}$ with this new transaction (T_i). If T_i is older than any already-colored but uncommitted transactions (say T_x), the leader cancels the color of those newer transactions, notifies the relevant shards, and reprocesses them later. This ensures older transactions are prioritized, avoiding starvation. The leader then runs an incremental greedy vertex coloring algorithm [8] to assign colors to all newly received transactions, without modifying the colors of already scheduled old transactions. This ensures that the processing time of already scheduled transactions is not affected by newly generated transactions. Note that a newer transaction might receive a lower color than an older one because the new one does not conflict with any other transaction (except one old transaction), while the old transaction conflicts with others as well. To prevent this and ensure a fair execution order, we assign each new transaction a color no lower than the smallest color among pending old transactions. This approach guarantees progress because at each time step, the lowest possible color will increase over time. After coloring and determining the schedule, each transaction is then split into subtransactions $T_{i,j}$ based on the destination shards it accesses, and these subtransactions are sent to the corresponding shards S_j for processing.

Each destination shard S_j maintains a local scheduled queue sch_{dq} (consisting of subtransactions that have been scheduled but not yet committed) and appends incoming subtransactions into sch_{dq} , which stores subtransactions in the order of their assigned color. To handle partial synchrony, each destination shard S_j uses a busy flag to track whether it is currently processing (in-transit and not committed yet) a subtransaction. If the shard is not busy, it picks one subtransaction from the head of the queue and validates it (e.g., checking conditions like account balance). If the subtransaction is valid, the shard S_j sends a *commit*

vote to the leader S_ℓ ; otherwise, it sends an *abort vote*. Once the leader shard receives votes from all relevant destination shards for a transaction T_i , it decides whether the transaction should be committed or aborted. If all subtransactions vote to commit, the leader sends a *confirmed commit* to each destination shard; otherwise, if any one of the shard send an abort vote, it sends a *confirmed abort*. After the decision, the transaction T_i is removed from the conflict graph $G_{\mathcal{T}_\ell}$ and the transaction set $\mathcal{T}_\ell$, and the outcome (committed or aborted) is sent to the home shard of T_i .

Upon receiving the confirmed decision, each destination shard either commits the subtransaction by appending it to the local blockchain or aborts it. If the scheduled queue is not empty, the shard continues processing the next subtransaction. If the queue becomes empty, the shard marks itself as not busy. Finally, upon receiving the outcome from the leader, the home shard generates a new transaction and repeats the process. This single leader scheduling approach ensures conflict-free execution while preserving consistency and fairness in transaction processing across shards.

Performance Analysis of Single-Leader Scheduler (Algorithm 1). Our proposed scheduling algorithm works on a partial-synchronous communication model; for the sake of performance analysis only, we consider the synchronous communication mode. In the following, we analyze the time units required to process transactions by Algorithm 1. We are focusing on the time period after the leader shard has determined the schedule for the transactions. In the synchronous case, a time unit is the time to send a message along an edge of unit weight. In the single-leader case, d is sensitive to the position of the leader and d denotes the maximum distance between any of the involved shards (home, destination shards, leader shard). In the multi-leader case, the distance to the leaders is not included in the definition of d .

► **Theorem 3.** *[General Graph] In the General graph, where the transactions, their accessing objects, and the leader are at most d distance away from each other, Algorithm 1 has $O(d \cdot \min\{k, \sqrt{s}\})$ competitive ratio.*

Proof. Consider a set of transactions $\mathcal{T}$ generated at or before time t that are still pending (neither committed nor aborted) at time t . Let $G_{\mathcal{T}}$ denote the conflict graph for $\mathcal{T}$, where two transactions conflict if they have a common destination shard. Since we use greedy coloring to color $G_{\mathcal{T}}$, the number of distinct colors assigned to the transactions in $\mathcal{T}$ depends only on the coloring of $G_{\mathcal{T}}$, and not on the colors of the transactions that have been finalized (committed or aborted) before t . (This holds since transactions in $\mathcal{T}$ may use smaller colors of transactions committed before t .)

Let l_i denote the number of transactions in $\mathcal{T}$ that use objects in shard S_i . Let $l = \max l_i$. We have that l is a lower bound on the time that it takes to finalize (commit or abort) the transactions in $\mathcal{T}$, since at least l subtransactions need to serialize in a destination shard.

First, consider the case where $k \leq \sqrt{s}$. We have that each transaction conflicts with at most kl other transactions. Hence $G_{\mathcal{T}}$ can be colored with at most $kl + 1$ colors. The distance between a transaction (home shard) and its accessing objects (destination shards) is at most d away, and to commit subtransactions after being scheduled, Algorithm 1 takes 3 steps of interactions (for each color) between the leader shard and the destination shard. This means each color corresponds to the $3d$ time units. Thus, it takes at most $(kl + 1)3d = O(kld)$ time units to confirm and commit the transactions. Hence, for transactions $\mathcal{T}$, the approximation of their finalization time is $O(kld/l) = O(kd)$.

Next, consider the case $k > \sqrt{s}$. We can write $\mathcal{T}' = A \cup B$, where A are the transactions which access at most $\sqrt{s}$ destination shards, while B are the transactions which access more than $\sqrt{s}$ destination shards. Each transaction in A conflicts with at most $l\sqrt{s}$ other

transactions. Hence, the transactions in A need at most $l\sqrt{s} + 1$ distinct colors. The transactions in B can be serialized, requiring at most $|B|$ distinct colors. Hence, the conflict graph G_T can be colored with at most $l\sqrt{s} + 1 + |B|$ colors, which implies a schedule of length $O(d(l\sqrt{s} + |B|))$ steps to finalize the transactions $\mathcal{T}$. Since each transaction in B accesses more than $\sqrt{s}$ shards, there is a shard accessed by more than $(|B|\sqrt{s})/s = |B|/\sqrt{s}$ transactions. Thus, $l > |B|/\sqrt{s}$. Hence, for transactions $\mathcal{T}$, the approximation of their finalization time is $O(d(l\sqrt{s} + |B|)/l) = O(d\sqrt{s} + d|B|/l) = O(d\sqrt{s} + d\sqrt{s}) = O(d\sqrt{s})$.

Therefore, combining the approximations for the cases $k \leq \sqrt{s}$ and $k > \sqrt{s}$, we have that the combined approximation for the finalization time for $\mathcal{T}$ is $O(d \cdot \min\{k, \sqrt{s}\})$. Since t is chosen arbitrarily, we have that the competitive ratio of Algorithm 1 is $O(d \cdot \min\{k, \sqrt{s}\})$. ◀

Suppose that shards are connected in a clique graph with unit distance, where every shard is connected to every other shard with unit distance. So in this case $d = 1$. Then from Theorem 3, Algorithm 1 has an $O(\min\{k, \sqrt{s}\})$ competitive ratio for a clique graph with unit distance. Thus, we have:

► **Corollary 4** (Unit Distance Clique). *Algorithm 1 has an $O(\min\{k, \sqrt{s}\})$ competitive ratio for a clique graph with unit distance.*

We continue to show that it is an NP-hard problem to approximate the optimal transaction schedule. Thus, the provided bound in Corollary 4, is the best we can do with a polynomial time scheduling algorithm. The result below applies to both the stateful and stateless model.

► **Theorem 5.** *For all $\epsilon > 0$, it is an NP-hard problem to produce a transaction schedule that achieves a competitive ratio $(\min\{k, \sqrt{s}\})^{1-\epsilon}$.*

The proof of Theorem 5 is deferred to Appendix A.3.

4.2 Multi-Leader Scheduler

This section provides the multi-leader scheduler where multiple leaders schedule and process the transactions, distribute the congestion, and load across different leaders. The multi-leader approach allows adaptation to the value d without requiring knowledge of d . Also, here the value d depends only on the maximum distance between the home and destination shards (without involving distances to the leaders). Therefore, the value of d captures better the locality of the transactions, and the resulting schedule allows for shorter messages between home and destination shards. The concepts that we introduce for this algorithm will play a key role for the development of the stateless multi-leader algorithm.

4.2.1 Shard Clustering

In the multi-leader scheduler, shards are distributed across the network, and the distance between the home shard of the transaction and its accessing objects (destination shards) ranges from 1 to D , where D is the diameter of the shard graph. Let us suppose shards graph G_s constructed with s shards, where the weights of edges between shards denote the distances between them. We consider that G_s is known to all the shards. We define z -neighborhood of shard S_i as the set of shards within a distance of at most z from S_i . Moreover, the 0-neighborhood of shard S_i is the S_i itself.

We consider that our multi-leader scheduling algorithm uses a hierarchical decomposition of G_s which is known to all the shards and calculated before the algorithm starts. This shard clustering (graph decomposition) is based on the clustering techniques in [10] and which

were later used in [18, 8, 2]. We divide the shard graph G_s into the hierarchy of clusters with $H_1 = \lceil \log D \rceil + 1$ layers (logarithms are in base 2), and a layer is a set of clusters, and a cluster is a set of shards. Layer q , where $0 \leq q < H_1$, is a sparse cover of G_s such that: (i) Every cluster of layer q has (strong) diameter of at most $O(2^q \log s)$. (ii) Every shard participates in no more than $O(\log s)$ different clusters at layer q . (iii) For each shard S_i there exists a cluster at layer q which contains the $(2^q - 1)$ -neighborhood of S_i within that cluster.

For each layer q , the sparse cover construction in [10] is actually obtained as a collection of $H_2 = O(\log s)$ partitions of G_s . These H_2 partitions are ordered as sub-layers of layer q labeled from 0 to $H_2 - 1$. A shard might participate in all H_2 sub-layers but potentially belongs to a different cluster at each sub-layer. At least one of these H_2 clusters at layer q contains the whole $2^q - 1$ neighborhood of S_i .

In each cluster at layer q , a leader shard S_ℓ is specifically designated such that the leader's $(2^q - 1)$ -neighborhood is in that cluster. As we give an idea of layers and sub-layers, we define the concept of height as a tuple $h = (h_1, h_2)$, where h_1 denotes the layer and h_2 denotes the sub-layer. Similar to [18, 8, 2], heights follow lexicographic order.

The *home cluster* for each transaction T_i is defined as follows: suppose S_i is the home shard of T_i , and z is the maximum distance from S_i to the destination shards that will be accessed by T_i ; the home cluster of T_i is the lowest-layer (and lowest sub-layer) cluster in the hierarchy that contains z -neighborhood of S_i . Each home cluster consists of one dedicated leader shard, which will handle all the transactions that have their home shard in that cluster (i.e., transaction information will be sent from the home shard to the cluster leader shard to determine the schedule). An example of hierarchical clustering is presented in Appendix A.4.

4.2.2 Stateless Multi-Leader Scheduler

We consider a hierarchical clustering of the shard graph G_s , which is assumed to be globally known by all shards. Each cluster C in this hierarchy is characterized by a unique height (q, r) which corresponds to its layer q and sublayer r , and each cluster C has its designated leader shard S_ℓ . The leader shard is responsible for scheduling and coordinating the processing of all transactions whose home cluster is C . Each home shard S_i maintains a local timestamp ts to tag newly generated transactions. Additionally, each destination shard S_j maintains a local scheduling queue sch_{dq} and lexicographically orders for the incoming subtransactions using the tuple $(ts, q, r, color)$, where $color$ is an integer assigned to the transaction by the leader shard S_ℓ through vertex coloring. Algorithm 2 invokes Algorithm 1 in each cluster C to process their transactions.

Algorithm 2 works in a partially synchronous model and follows an event-driven execution by message passing. When a new transaction T_i is generated at its home shard S_i , then the home shard S_i determines the lowest cluster C at height (q, r) that includes both S_i and all of the destination shards accessed by T_i . Moreover, the transaction is tagged with its local timestamp ts , along with the cluster identifiers q and r , and is then sent to the cluster's leader shard S_ℓ .

Upon receiving new transaction(s) T_i , the leader shard S_ℓ of cluster C invokes Algorithm 1 to process their transactions, where leader shard S_ℓ adds T_i to the transaction set $\mathcal{T}_C$ of cluster C and updates the corresponding transaction conflict graph $G_{\mathcal{T}_C}$ to incorporate the new transaction T_i . Then the leader shard uses an incremental greedy vertex coloring algorithm [8] to assign a color only to the newly received transaction without affecting already colored (scheduled) transactions. Once colored, the transaction is split into subtransactions $T_{i,j}$, and sent to the respective destination shard S_j .

Algorithm 2 STATELESS MULTI-LEADER SCHEDULER.

```

1 Assume all shards know a hierarchical cluster decomposition of  $G_s$ ;
2 Each cluster  $C$  is associated with a unique height  $(q, r)$  and has a designated leader shard  $S_\ell$ ;
3 Each shard  $S_j$  maintains a lexicographically ordered queue  $sch_{dq}$  for subtransactions;
4 Upon generation of a new transaction  $T_i$  at home shard  $S_i$ 
5    $S_i$  tags a local timestamp ( $ts$ ) to  $T_i$  and identifies the destination shards accessed by  $T_i$ ;
6    $S_i$  selects the lowest cluster  $C$  with height  $(q, r)$  that contains  $T_i$  and all its destination shards;
7    $S_i$  sends  $T_i$  to the leader shard  $S_\ell$  of cluster  $C$ ;

8 Upon receiving transaction  $T_i$  at the leader shard  $S_\ell$  of cluster  $C$ 
9   The leader shard  $S_\ell$  of each cluster  $C$  invokes Algorithm 1 to schedule and process their transactions.
10  This means each cluster  $C$  invokes Algorithm 1 to process their transactions;
11  // Since multiple clusters process their transactions concurrently, each with its own leader,
12  destination shards may receive subtransactions from different clusters simultaneously.
13  To handle subtransactions from multiple clusters (leaders):
14    Each destination shard  $S_j$  maintains a scheduled subtransactions queue  $sch_{dq}$  ordered
15    lexicographically by the tuple  $(ts, q, r, color)$ . The additional parameters  $(ts, q, r)$  reflect the
16    hierarchical cluster heights (layers and sublayers) in the shard graph  $G_s$ ;
17    Each destination shard  $S_j$  processes their subtransactions from the head of  $sch_{dq}$  following the
18    rules in Algorithm 1, with the modified ordering criteria;

```

Since multiple leader shards process their transactions concurrently by invoking the Algorithm 1, destination shards may receive the subtransactions from different clusters simultaneously. To handle this, we modify the parameters and processing technique of Algorithm 1 as follows: each destination shard S_j maintains a scheduled subtransactions queue sch_{dq} , which is ordered lexicographically by the tuple $(ts, q, r, color)$. The additional parameters (ts, q, r) denote the timestamp ts , and hierarchical cluster heights (layers q and sublayers r) in the shard graph G_s . Moreover, each destination shard S_j processes its subtransactions from the head of sch_{dq} following the steps in Algorithm 1 with the modified ordering criteria.

Additionally, if the destination shard is busy and receives a new subtransaction $T_{i',j}$ such that $ts(T_{i',j}) < ts(T_{i,j})$ in lexicographic order, this means $T_{i',j}$ has a higher priority where $T_{i,j}$ is the currently processed (but not committed) subtransaction, then the shard give priority to $T_{i',j}$ by sending an *ignore* $T_{i,j}$ message to its leader, indicating that a higher-priority transaction (subtransaction $T_{i',j}$) should proceed first. Then, when the leader receives an *ignore* $T_{i,j}$ message for a subtransaction $T_{i,j}$ and the decision for T_i has not yet been made (i.e., not all votes have been received), the leader discards the vote from S_j and replies with an *ignored* $T_{i,j}$ message to the destination shard S_j . If the decision has already been made (i.e., confirm commit or confirm abort) by the leader shard, then no further action is taken for particular subtransaction $T_{i,j}$ at the leader shard S_ℓ . After the destination shard S_j receives an *ignored* message for $T_{i,j}$, then it reinserts $T_{i,j}$ into the scheduled queue, reorders the queue lexicographically, and resumes processing from the head.

Finally, when the home shard S_i receives the final outcome of its transaction T_i , it generates a new transaction and sends it to the corresponding cluster leader shard, and the process repeats. This multi-leader scheduling framework ensures conflict-free and consistent execution by leveraging lexicographic ordering over the tuple $(ts, q, r, color)$, and maintains the fairness and parallelism across shards in the presence of partial synchrony.

Performance Analysis of Stateless Multi-Leader Scheduler. The multi-leader scheduler is the extended version of the single-leader scheduler (Algorithm 3) while introducing an additional overhead cost due to its shard (hierarchical) clustering structure and comes from the layers and sublayers of the clusters.

► **Theorem 6.** *In Multi-leader scheduler (Algorithm 2), where the transactions and their accessing objects are at most d distance away from each other, Algorithm 2 has $O(d \log^2 s \cdot \min\{k, \sqrt{s}\})$ competitive ratio.*

The proof of Theorem 6 is deferred to Appendix A.6.

5 Stateful Scheduler

In this scheduler model, the leader shard gathers all of the transactions and the current states of the accessing accounts and pre-commits the transactions at the leader. After that, the leader creates the pre-committed subtransactions batch and sends that batch to the respective destination shard, where each destination shard reaches a consensus on the received subtransaction order and adds it to their local blockchain. We provide two stateful scheduling algorithms, one with a single leader and the other with multiple leaders.

5.1 Stateful Single-Leader Scheduler

We present and analyze the *stateful single-leader scheduler*, where one of the shards is considered as the leader S_ℓ , which is responsible for scheduling and processing all the transactions.

When a new transaction T_i is generated at its home shard S_i , S_i sends T_i to the leader shard S_ℓ . Upon receipt, S_ℓ appends T_i to its local pending queue PQ_ℓ . Scheduling event is triggered periodically, either every 4λ time units or upon processing transactions associated with λ distinct colors. Here, λ denotes the worst-case communication delay between any two shards, which is at most the diameter of the shard communication graph G_s . The 4λ bound accounts for the communication delays involved in acquiring state information from remote shards and completing the pre-commitment phase and sending the pre-committing batch to the destination shard.

When the scheduling event is triggered, the leader shard moves its pending transactions from PQ_ℓ into the scheduling transaction set $\mathcal{T}_\ell$ and identifies the set of accounts $\mathcal{O}_v$ accessed by transactions which are in $\mathcal{T}_\ell$. If any account state $O_j \in \mathcal{O}_v$ is not locally available at S_ℓ , it determines the responsible destination shard S_j for each such account, and sends batched account state requests to the corresponding shards. If all required states are already available in S_ℓ , an internal STATE-READY (i.e. already available locally) event is triggered immediately.

Upon receiving a state request, each destination shard S_j responds with the current state of the requested accounts (e.g., balances). Then, once all necessary account states are collected at S_ℓ , it extends the conflict graph $G_{\mathcal{T}_\ell}$ by incorporating the new transactions in $\mathcal{T}_\ell$. Then the leader shard S_ℓ runs the incremental greedy vertex coloring algorithm [8] on $G_{\mathcal{T}_\ell}$ and assigns at most ζ colors without altering the coloring of previously scheduled old transactions.

The leader then iteratively processes transactions color by color. For each color group ζ_c , S_ℓ verifies transaction conditions (e.g., sufficient balance) using the up-to-date account state it gathers. Transactions that are valid and conditions are satisfied are *pre-committed*, while invalid ones are aborted. Then S_ℓ splits each pre-committed transaction T_i into subtransactions $T_{i,j}$ based on its accessed shards. These subtransactions are appended to a corresponding pre-commit batch $PrecommitSubTxnBatch(S_j)$ for each destination shard S_j . After processing a transaction, it is removed from $\mathcal{T}_\ell$ and $G_{\mathcal{T}_\ell}$, and the outcome (committed or aborted) is reported back to the transaction's home shard S_i to initiate the next transaction.

Algorithm 3 STETEFUL SINGLE LEADER SCHEDULER.

```

1  $S_\ell$ : Leader shard;  $PQ_\ell$ : Pending txns queue in leader shard;
2  $\mathcal{T}_\ell$ : Set of scheduled txns maintained by leader;  $G_{\mathcal{T}_\ell}$ : Conflict txn graph on  $\mathcal{T}_\ell$ ;
3  $\lambda$ : worst communication delay between any two shards due to partial-synchrony;
4  $PrecommitSubTxnBatch(S_j)$ : Precommitted subtransactions batch for shard  $S_j$ ;
5 Upon generation of a new txn  $T_i$  at home shard  $S_i$ 
6    $S_i$  sends  $T_i$  to the leader shard  $S_\ell$ ;
7 Upon receiving new txn  $T_i$  at leader shard  $S_\ell$ 
8    $S_\ell$  appends  $T_i$  to  $PQ_\ell$ ;
9   if  $S_\ell$  waits for  $4\lambda$  time unit or  $S_\ell$  proceed  $\lambda$  number of scheduled colors then
10     // Trigger scheduling event
11     Move txns from  $PQ_\ell$  to  $\mathcal{T}_\ell$ ; Identify set of accessed accounts  $\mathcal{O}_v$  by txns in  $\mathcal{T}_\ell$ ;
12     if Current state of any account  $O_j \in \mathcal{O}_v$  is not locally available at  $S_\ell$  then
13       For each  $O_j$ , determine the responsible shard  $S_j$  and create request batch for each  $S_j$ ;
14       Send batched account state request to each destination shard  $S_j$ ;
15     else
16        $S_\ell$  has all accounts state, so trigger internal state-ready event (see below);
17 Upon receiving a batched account state request at destination shard  $S_j$ 
18   Respond to leader shard  $S_\ell$  with current states of all requested accounts;
19 Upon receiving account states from each  $S_j$ , or already available locally at  $S_\ell$ 
20    $S_\ell$  extend txn conflict graph  $G_{\mathcal{T}_\ell}$  with new txns in  $\mathcal{T}_\ell$  and runs incremental greedy vertex coloring
    algorithm on  $G_{\mathcal{T}_\ell}$  using  $\zeta$  colors without altering already scheduled old txns;
21   foreach color  $\zeta_c \in \zeta$  do
22     Pre-commit or abort txns  $T_i \in \zeta_c$  by checking txn condition and account state;
23     If  $T_i$  is pre-committed, split  $T_i$  into subTxns and create (append) pre-committed subtxns batch
    order  $PrecommitSubTxnBatch(S_j)$  for each destination shard  $S_j$ ;
24     Remove  $T_i$  from  $\mathcal{T}_\ell$  and  $G_{\mathcal{T}_\ell}$ . Send the outcome(committed/aborted) to home shard of  $T_i$ ;
25     // Track processed color
26     if processed  $\lambda$  number of colors then
27       break;
28    $S_\ell$  sends  $PrecommitSubTxnBatch(S_j)$  to corresponding destination shard  $S_j$  parallelly and start to
    process next batch;
29 Upon receiving precommitted batch  $PrecommitSubTxnBatch(S_j)$  at each  $S_j$ 
30   Reach consensus on  $PrecommitSubTxnBatch(S_j)$  and append batch to the local blockchain;
  
```

The pre-commitment phase terminates once λ colors are processed, after which S_ℓ dispatches all $PrecommitSubTxnBatch(S_j)$ batches to their respective destination shards in parallel. Each destination shard S_j then reaches consensus on the order of subtransactions in the received batch and appends them to its local blockchain. The leader shard S_ℓ , then waits and proceeds to the next scheduling batch.

Performance Analysis of Stateful Single-Leader Scheduler. In the following, we analyze the time unit required to process transactions by Algorithm 3. We focus on the special case where the maximum distance between the transactions, their accessing objects, and the leader is at most d , and at least one transaction accesses objects at a distance $\Omega(d)$. This special case is useful for the analysis of the multi-leader case. We are focusing on the time period after the leader shard has determined the schedule for the transactions. This is because the scheduling and committing steps are executed in parallel.

► **Theorem 7.** [General Graph] *In the General graph, where the transactions, their accessing objects, and the leader are at most d distance away from each other, and at least one transaction is $\Omega(d)$ distance from the accessing shards, Algorithm 3 has $O(\min\{k, \sqrt{s}\})$ competitive ratio.*

The proof of Theorem 7 is deferred to Appendix A.7.

5.2 Stateful Multi-Leader Scheduler

We present a *stateful multi-leader scheduler* in which multiple leader shards are responsible for scheduling and processing transactions. In the single-leader algorithm, the value d includes the distance to the leader, but in the multi-leader, d does not include the relative distance to the leader. This allows the multi-leader algorithm to capture better the locality of transactions, allowing for shorter distance messages between the involved home and destination shards.

The system assumes a hierarchical cluster decomposition [10] of the shard graph G_s , which is globally known to all shards. Each cluster $C(q, r)$ in the hierarchy is associated with a leader shard S_ℓ , a pending transaction queue PQ_ℓ , a scheduled transaction set $\mathcal{T}_\ell$, and a transaction conflict graph $G_{\mathcal{T}_\ell}$. The parameter λ_C denotes the worst-case communication delay between any two shards within the cluster C , which arises from the assumption of a partially synchronous communication model.

In multi-leader scheduling Algorithm 4 (see in Appendix A.9), when a transaction T_i is generated at its home shard S_i , the shard identifies the lowest cluster $C(q, r)$ that contains all the shards accessed by T_i , and then forwards T_i to the leader shard S_ℓ of cluster C . The leader shard S_ℓ appends the received transaction to its pending queue PQ_ℓ . Periodically, the leader checks if either $4\lambda_C$ time units have elapsed since the last scheduling event or if λ_C colors of scheduled transactions have been processed by S_ℓ . If either condition is met and the leader holds the **scheduleControl**, it invokes the single-leader scheduler (Algorithm 3) on its local structures $(PQ_\ell, \mathcal{T}_\ell, G_{\mathcal{T}_\ell}, \lambda_C)$ to process transactions.

The scheduling control, denoted by the boolean flag **scheduleControl**, determines which cluster can perform scheduling operations at a given time unit. The control flows hierarchically between parent and child clusters. A parent cluster of C is any cluster at a higher level in the hierarchy (with height $(q', r') > (q, r)$) that shares at least one shard with C . Similarly, a child cluster of C is a lower-level cluster (with height $(q'', r'') < (q, r)$) that overlaps with C . Clusters may have multiple parents and children. If C is at the bottom-most level (height $(0, 0)$), initially it has **scheduleControl**. Otherwise, it must request control from all its children. Once all children respond the **scheduleControl**, the leader S_ℓ sets **scheduleControl** to true and proceeds with the scheduling.

After executing the single-leader scheduler, if the parent cluster C' requests control, the leader transfers **scheduleControl** to the parent and sets it to false locally. If instead a child cluster C'' has made a request, the control is passed down to the child. If there are no remaining transactions to process, the control is also passed downward to allow lower-level clusters to schedule pending transactions. If the leader does not have **scheduleControl** when scheduling should occur, it sends a control request to the current holder (parent or child). Additionally, if C receives a control request from a parent C' while not holding control, it forwards the request to its children. Once all children respond positively, it passes control up to C' . This hierarchical and event-driven mechanism ensures coordinated and conflict-free scheduling across multiple levels of the cluster hierarchy.

► **Theorem 8.** *In Multi-leader scheduler (Algorithm 4), where the transactions and their accessing objects are at most d distance away from each other, Algorithm 4 has $O(\log s \cdot \min\{k, \sqrt{s}\} + \log^2 s)$ competitive ratio.*

The proof of Theorem 8 is deferred to Appendix A.8.

6 Conclusion

We presented efficient scheduling algorithms for processing dynamic transactions in blockchain sharding systems. Our proposed framework operates under a partially synchronous communication model, which realistically captures the behavior of many real-world blockchain environments. We introduced both stateless and stateful scheduling models, each of which includes single-leader and multi-leader algorithms for transaction scheduling and processing. For these algorithms, we provided competitive ratios relative to an optimal scheduler and established both upper and lower bounds on the scheduling delay.

For future work, we plan to explore efficient inter-shard communication mechanisms, particularly under conditions of network congestion where communication links have bounded capacity. We also aim to conduct extensive simulations and real-world experiments to evaluate the practical performance of our proposed protocols.

References

- 1 Ramesh Adhikari and Costas Busch. Lockless blockchain sharding with multiversion control. In *International Colloquium on Structural Information and Communication Complexity*, pages 112–131. Springer, 2023. doi:10.1007/978-3-031-32733-9_6.
- 2 Ramesh Adhikari, Costas Busch, and Dariusz Kowalski. Stable blockchain sharding under adversarial transaction generation. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, 2024.
- 3 Ramesh Adhikari, Costas Busch, and Miroslav Popovic. Fast transaction scheduling in blockchain sharding. *arXiv preprint arXiv:2405.15015*, 2024. doi:10.48550/arXiv.2405.15015.
- 4 Ramesh Adhikari, Costas Busch, and Miroslav Popovic. On the efficiency of dynamic transaction scheduling in blockchain sharding. *arXiv preprint arXiv:2508.07472*, 2025.
- 5 Hagit Attiya, Vincent Gramoli, and Alessia Milani. Directory protocols for distributed transactional memory. *Transactional Memory. Foundations, Algorithms, Tools, and Applications: COST Action Euro-TM IC1001*, pages 367–391, 2015. doi:10.1007/978-3-319-14720-8_17.
- 6 Zeta Avarikioti and Dimitris Karakostas. Harmony technical report, 2022.
- 7 Costas Busch, Maurice Herlihy, Miroslav Popovic, and Gokarna Sharma. Fast scheduling in distributed transactional memory. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 173–182, 2017. doi:10.1145/3087556.3087565.
- 8 Costas Busch, Maurice Herlihy, Miroslav Popovic, and Gokarna Sharma. Dynamic scheduling in distributed transactional memory. *Distributed Computing*, 35(1):19–36, 2022. doi:10.1007/S00446-021-00410-W.
- 9 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.
- 10 Anupam Gupta, Mohammad T Hajiaghayi, and Harald Räcke. Oblivious network design. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 970–979, 2006.
- 11 Jelle Hellings and Mohammad Sadoghi. Byshard: Sharding in a byzantine environment. *Proceedings of the VLDB Endowment*, 14(11):2230–2243, 2021. doi:10.14778/3476249.3476275.
- 12 Jelle Hellings and Mohammad Sadoghi. The fault-tolerant cluster-sending problem. In *Foundations of Information and Knowledge Systems: 12th International Symposium, FoIKS 2022, Helsinki, Finland, June 20–23, 2022, Proceedings*, pages 168–186. Springer, 2022. doi:10.1007/978-3-031-11321-5_10.
- 13 Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018. doi:10.1109/SP.2018.000–5.

- 14 Ao Liu, Jing Chen, Kun He, Ruiying Du, Jiahua Xu, Cong Wu, Yebo Feng, Teng Li, and Jianfeng Ma. Dynashard: Secure and adaptive blockchain sharding protocol with hybrid consensus and dynamic shard management. *IEEE Internet of Things Journal*, 2024.
- 15 Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 17–30, 2016. doi:10.1145/2976749.2978389.
- 16 Ahmed Afif Monrat, Olov Schelén, and Karl Andersson. A survey of blockchain from the perspectives of applications, challenges, and opportunities. *Ieee Access*, 7:117134–117151, 2019. doi:10.1109/ACCESS.2019.2936094.
- 17 A Secure. The zilliqa project: A secure, scalable blockchain platform, 2018.
- 18 Gokarna Sharma and Costas Busch. Distributed transactional memory for general networks. *Distributed computing*, 27(5):329–362, 2014. doi:10.1007/S00446-014-0214-7.
- 19 Gokarna Sharma and Costas Busch. A load balanced directory for distributed shared memory objects. *Journal of Parallel and Distributed Computing*, 78:6–24, 2015. doi:10.1016/J.JPDC.2015.02.002.
- 20 Alex Skidanov and Illia Polosukhin. Nightshade: Near protocol sharding design. URL: <https://nearprotocol.com/downloads/Nightshade.pdf>, 39, 2019.
- 21 Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, pages 931–948, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3243734.3243853.
- 22 David Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing, STOC '06*, pages 681–690, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1132516.1132612.

A

 Appendix

A.1 Example of Transaction and Subtransactions

Suppose transaction T_1 is: “Transfer 100 coins from account A to account B”. Let us assume that the accounts of A and B reside on different shards S_a and S_b , respectively. T_1 splits into the following subtransactions:

- $T_{1,a}$ in S_a : Condition: Check if account A has at least 100 coins.
Action: Deduct 100 coins from account A.
- $T_{1,b}$ in S_b : Action: Add 100 coins to account B.

A.2 Correctness Analysis of Stateless Single-Leader Scheduler

Our proposed scheduling algorithm works on a partial-synchronous communication model; for the sake of analysis only, we consider the synchronous communication mode.

► **Lemma 9 (Safety).** *If two transactions conflict with each other in Algorithm 1, then they will commit in different time slots, and the local chain produced by Algorithm 1 ensures blockchain serialization.*

Proof. We prove this by induction (analyzing) the execution of Algorithm 1, where each home shard sends its transaction to the leader shard (Line 4), and the leader shard constructs the transaction conflict graph $G_{\mathcal{T}_\ell}$ (Line 6). Then the leader used the incremental greedy vertex coloring algorithm [8] on the conflict graph $G_{\mathcal{T}_\ell}$ (Line 8). As conflicting transactions

share an edge in $G_{\mathcal{T}_\ell}$, they are assigned different colors and are processed in different time slots, which provides the valid commit order. Moreover, each color corresponds to a unique serialization time slot. The leader shard splits the transaction into subtransactions and sends them to the destination shard after coloring (see Line 9), then each destination shard keeps that ordering in the schedule queue (sch_{dq}) and process subtransactions one by one according to the color they get (see Line 11-14), which guarantees the consistent schedule order in each shard. Moreover, the leader shard coordinates to commit the subtransactions in each destination shard, which ensures the consistent commitment (see Line 16-17). As the subtransactions are committed according to the color they receive, and each color corresponds to a globally consistent time slot, this provides global serialization. ◀

► **Lemma 10** (Liveness). *Algorithm 1 guarantees that every generated transaction will eventually be either committed or aborted.*

Proof. We prove liveness by induction, showing that every transaction T_i is either committed or aborted in finite time. Each new transaction T_i is sent to a leader shard S_ℓ (Line 4), which adds it to the set $\mathcal{T}_\ell$ and the conflict graph $G_{\mathcal{T}_\ell}$. If T_i is older than any already colored but not committed transaction T_x , the algorithm cancels the color of T_x and re-colors the graph (Line 7). Coloring is performed incrementally (Line 8) and preserves the colors of previously scheduled transactions. Thus, older transactions are always prioritized, and no transaction is indefinitely prevented from being scheduled due to newer ones. Note that a newer transaction might receive a lower color than an older one because the new one does not conflict with any other transaction (except one old transaction), while the old transaction conflicts with others as well. To prevent this and ensure a fair execution order, we assign each new transaction a color no lower than the smallest color among pending old transactions. This approach guarantees progress because at each time step, the lowest possible color will increase over time.

Moreover, once T_i is colored, its subtransactions are sent to the respective destination shards (Line 9), where they are placed into a queue sch_{dq} sorted by color (Line 11). Each shard processes one color group at a time, controlled by a busy flag. After finishing one subtransaction (commit or abort), the shard proceeds to the next one in the queue. Since every color is eventually dequeued, and subtransactions are processed in order, every scheduled subtransaction is eventually processed. Thus, every transaction is either committed or aborted in a finite time, and this proves the liveness. ◀

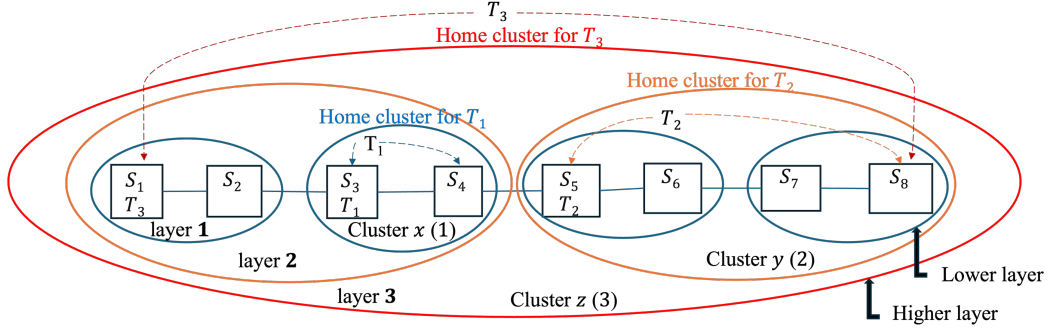
► **Corollary 11.** *From Lemma 9 and Lemma 10, Algorithm 1 ensures the safety and liveness of the transactions.*

A.3 Proof of Theorem 5

Proof. We will use a reduction from vertex coloring. For all $\epsilon > 0$, the problem of approximating the chromatic number of a graph with n nodes within a factor $n^{1-\epsilon}$ is NP-hard [22].

Consider an instance of vertex coloring on a graph $H = (V_H, E_H)$ with n nodes. We can transform the vertex coloring instance H to a scheduling problem instance on a graph shard G_s with $s = |E_H|$ shards, such that G_s is a synchronous clique with unit distances between the shards. Furthermore, each edge of E_H corresponds to a unique node of G_s .

Let $\mathcal{T}$ be a set of n transactions, all generated concurrently at time $t = 0$, such that each node $v_i \in V_H$ is mapped to transaction $T_i \in \mathcal{T}$. For each edge $(v_i, v_j) \in E_H$ we create a conflict between respective transactions T_i and T_j by making the transactions access a common object in the unique shard of G_s that corresponds to edge (v_i, v_j) . Let $G_{\mathcal{T}}$ be the respective conflict graph for the transactions $\mathcal{T}$. The conflict graph $G_{\mathcal{T}}$ is isomorphic to H .



■ **Figure 2** Simple example of cluster decomposition of shard graph G_s .

A correct execution schedule for $\mathcal{T}$ (which gives a valid serialization of the transactions in $\mathcal{T}$) can be represented as a DAG where nodes are transactions and transaction T_i points to T_j if they conflict and T_i executes first in the respective common destination shard with T_j . Then, a layering of the DAG nodes starting from source nodes provides a unique time step for each transaction, so that conflicting transactions receive different time steps. Thus, an execution schedule of the transactions in $\mathcal{T}$ gives a valid vertex coloring of the nodes in $G_{\mathcal{T}}$ which provides a valid coloring for H . The best length of the transaction schedule given from the DAG, is equal to the number of colors that can be assigned to H .

Since $|E_H| \leq n(n-1)/2$, we have that $s = O(n^2)$. Each transaction conflicts with at most $k \leq n-1$ other transactions. Therefore, given k and s , we can create the reduction from graph coloring for $n = \min(k, \sqrt{s})$. Consequently, the NP-hardness of the scheduling problem in G_s follows from the NP-hardness of the reduced graph coloring problem with $n = \min(k, \sqrt{s})$. ◀

A.4 Example of Hierarchical Clustering

Figure 2 shows an example of hierarchical clustering, assuming shards are connected as if they are in a line, where edges in the line have low weights and edges not in the line have large weights. (We omit the sublayers to simplify the example.) Transaction T_1 resides in shard S_3 and has home cluster x at layer 1. The reason for the home cluster x selection is that T_1 accesses an object in S_3 and S_4 , and both of them are in cluster x , and x is the lowest layer cluster including S_3 and S_4 . Similarly, suppose transaction T_2 , which resides in S_5 , has home cluster y at layer 2, because T_2 accesses an object in S_5 and S_8 , and y is the lowest layer cluster that includes both S_5 and S_8 . Similarly, T_3 has home cluster z at layer 3.

A.5 Correctness Analysis of Stateless Multi-Leader Scheduler

► **Lemma 12 (Safety).** *If two transactions conflict with each other in Algorithm 2, then they will commit in different time slots, and the local chain produced by Algorithm 2 ensures blockchain serialization.*

The proof of Lemma 12 is available in arxiv [4]

► **Lemma 13 (Liveness).** *Algorithm 2 guarantees that every generated transaction will eventually be committed or aborted.*

The proof of Lemma 13 is available in arxiv [4].

► **Corollary 14.** *From Lemma 12 and Lemma 13, Algorithm 2 ensures the safety and liveness of the transactions.*

A.6 Proof of Theorem 6

Proof. In the multi-layer scheduler, we need to consider the transactions from all layers and sublayers of the clusters. Suppose q' is the topmost layer accessed by any transaction where the diameter of the cluster on that layer is at most $d_{q'}$.

Consider the destination shard S_j , and we have only subtransactions from one leader shard of cluster layer q where the distance between the transaction and its accessing shard is at most d_q , and it has maximum competitive ratio denoted by $\tau_q = O(d_q \cdot \min\{k, \sqrt{s}\})$ (from Theorem 3) than any other cluster. Therefore the destination shard S_j needs to process subtransactions from all layers $0, \dots, q'$ and from sublayers $0, \dots, H_2 - 1$, and those transactions are processed according to their assigned order.

As discussed in Section 4.2.1, a cluster at layer q has a diameter at most $O(2^q \log s)$. Thus $d_q = O(2^q \log s) = c2^q \log s$, for some positive constant c . This implies $\sum_{q=0}^{q'} d_q \leq 2d_{q'}$. Thus, the competitive ratio of Algorithm 2 considering transactions from all layers and sublayers at destination shard S_j is at most:

$$\tau_{total} \leq \sum_{q=0}^{q'} \sum_{r=0}^{H_2-1} \tau_q \leq \sum_{q=0}^{q'} \sum_{r=0}^{H_2-1} O(d_q \cdot \min\{k, \sqrt{s}\}) \leq O(d_{q'} H_2 \cdot \min\{k, \sqrt{s}\}) . \quad (1)$$

We can replace $H_2 = O(\log s)$ and $d_{q'} = O(d \log s)$ (see Section 4.2.1), then Equation 1 becomes:

$$O(d \log^2 s \cdot \min\{k, \sqrt{s}\}) . \quad \blacktriangleleft$$

The correctness analysis of stateful single leader and multi-leader scheduler is available in arxiv [4].

A.7 Proof of Theorem 7

Proof. This proof follows the same arguments discussed in the proof of Theorem 3. Consider a set of transactions $\mathcal{T}$ generated at or before time t that are still pending (neither committed nor aborted) at time t . Let $G_{\mathcal{T}}$ denote the conflict graph for $\mathcal{T}$, where two transactions conflict if they have a common destination shard. Let l_i denote the number of transactions in $\mathcal{T}$ that use objects in shard S_i . Let $l = \max l_i$. Moreover, from the definition of d , at least one transaction is d distance away from the destination shard or leader. So we have that $\Omega(l + d)$ is a lower bound on the time that it takes to finalize (commit or abort) the transactions in $\mathcal{T}$, since at least l subtransactions need to serialize in a destination shard, and at least one transaction is d distance away.

First, consider the case where $k \leq \sqrt{s}$. We have that each transaction conflicts with at most kl other transactions. Hence $G_{\mathcal{T}}$ can be colored with at most $kl + 1$ colors.

Algorithm 3 schedules and commits transactions in batches. For each batch, the leader shard performs the following steps: first, it gathers the state of accessed accounts, takes at most $2d$ time units (request and receive each takes at most d time units). After pre-committing, the leader sends the pre-commit batch to destination shards, which takes d time units. Additionally, destination shards reach consensus on the received batch within 1 time unit. Hence, the total delay per batch is at most $3d + 1$.

Since the algorithm uses at most $kl + 1$ colors (batches), the total finalization time is at most: $kl + 1 + 3d + 2 = O(kl + d)$.

Next, consider the case $k > \sqrt{s}$. Following the same reasoning above and from Theorem 3, we get $O(l\sqrt{s} + d)$ time to finalize the transactions $\mathcal{T}$.

Overall, Algorithm 3 requires $O(l \cdot \min\{k, \sqrt{s}\} + d)$ time units to finalize the transactions. Since $\Omega(l + d)$ is a lower bound, we have that the approximation factor of the schedule for $\mathcal{T}$ is $O(\min\{k, \sqrt{s}\})$.

Since t is chosen arbitrarily, we have that the competitive ratio of Algorithm 3 is $O(\min\{k, \sqrt{s}\})$. ◀

A.8 Proof of Theorem 8

Proof. Similar to Theorem 6, consider the destination shard S_j , as discussed in the proof of Theorem 7, if we have only subtransactions from one leader shard of cluster layer q where the distance between the transaction and its accessing shard is at most d_q , then the time to process transactions is $O(l \cdot \min\{k, \sqrt{s}\} + d_q)$ or equivalently at most $c_1(l \cdot \min\{k, \sqrt{s}\} + d_q)$ time for some positive constant c_1 . Suppose q' is the maximum layer accessed by any transaction where the diameter of the cluster on that layer is at most $d_{q'}$. Then the destination shard S_j needs to process subtransactions from all layers $0, \dots, q'$ and from sublayers $0, \dots, H_2 - 1$, and those transactions are processed according to their assigned order.

As discussed in Section 4.2.1, a cluster at layer q has a diameter at most $O(2^q \log s)$. Thus $d_q = O(2^q \log s) = c2^q \log s$, for some positive constant c . This implies $\sum_{q=0}^{q'} d_q \leq 2d_{q'}$. Thus, the total time unit required by Algorithm 4 to process all the transactions from all layer and sublayers at destination shard S_j is at most:

$$\tau_{total} \leq \sum_{q=0}^{q'} \sum_{r=0}^{H_2-1} c_1(l \cdot \min\{k, \sqrt{s}\} + d_q) \leq c_1 l H_2 \cdot \min\{k, \sqrt{s}\} + 2c_1 d_{q'} H_2. \quad (2)$$

We can replace $H_2 = c_2 \log s$ as we have $O(\log s)$ sublayers (see Section 4.2.1) and $d_{q'} = c_3 d \log s$, where c_2 and c_3 are some positive constants, then Equation 2 becomes:

$$c_1 l \cdot c_2 \log s \cdot \min\{k, \sqrt{s}\} + 2c_1 \cdot c_3 d \log s \cdot c_2 \log s \Rightarrow O(l \log s \cdot \min\{k, \sqrt{s}\} + d \log^2 s).$$

As discussed in Theorem 7, $\Omega(l + d)$ is a lower bound. Thus, we have that the competitive ratio of Algorithm 4 as $O(\log s \cdot \min\{k, \sqrt{s}\} + \log^2 s)$. ◀

A.9 Pseudocode of Stateful Multi-Leader Scheduler

Algorithm 4 STATEFUL MULTI-LEADER SCHEDULER.

```

1 Each shard knows the hierarchical cluster decomposition of  $G_s$ ;
2 Each cluster  $C(q, r)$  has: leader shard  $S_\ell$ , txn queue  $PQ_\ell$ , scheduled txns  $\mathcal{T}_\ell$ , conflict graph  $G_{\mathcal{T}_\ell}$ ;
3  $\lambda_C$ : worst communication delay between any two shards in cluster  $C$  due to partial-synchrony;
4 scheduleControl: Boolean flag indicating whether the cluster currently holds scheduling control;
5 Upon generation of new txn  $T_i$  at home shard  $S_i$ 
6    $S_i$  determines the lowest cluster  $C(q, r)$  which includes  $T_i$  and its accessing shards. Then  $S_i$  sends  $T_i$ 
   to leader shard  $S_\ell$  of  $C(q, r)$ ;
7 Upon receiving txn(s)  $T_i$  at leader shard  $S_\ell$  of  $C(q, r)$ 
8    $S_\ell$  appends  $T_i$  to its pending transactions queue  $PQ_\ell$ ;
9   if  $S_\ell$  waits for  $4\lambda_C$  time unit or  $S_\ell$  proceed  $\lambda_C$  number of previous scheduled colors then
10    if  $\text{scheduleControl} == \text{True}$  then
11      // Invoke single-leader scheduling logic
12      Run Single-Leader Scheduler (Algorithm 3) with  $(PQ_\ell, \mathcal{T}_\ell, G_{\mathcal{T}_\ell}, \lambda_C)$ ;
13      // If Algorithm 3 break after process  $\lambda_C$  number of scheduled colors then check and do
        following:
14      if parent cluster  $C'$  requests control then
15        Send scheduleControl to the parent and set scheduleControl  $\leftarrow$  False;
16      else if children clusters  $C''$  request control then
17        Send scheduleControl to children and set scheduleControl  $\leftarrow$  False;
18      else if  $C(q, r)$  doesn't have remaining transactions to schedule then
19        Send scheduleControl down to children and set scheduleControl  $\leftarrow$  False;
20    else
21      Send request to current scheduleControl holder (e.g., child or parent cluster);
22 Upon receiving scheduleControl at leader  $S_\ell$  of  $C(q, r)$ 
23   if  $S_\ell$  previously requested scheduleControl to process its txns then
24     Set scheduleControl  $\leftarrow$  True and trigger internal event (see above on line 9-12);
25   else
26     Send scheduleControl to parent or child clusters according to the request it gets;

```
