Distributed Computing
https://doi.org/10.1007/s00446-021-00410-w

Dynamic scheduling in distributed transactional memory

Costas Busch[1] · **Maurice Herlihy**[2] · **Miroslav Popovic**[3] · **Gokarna Sharma**[4]

Received: 11 January 2021 / Accepted: 18 October 2021

Abstract
We investigate scheduling algorithms for distributed transactional memory systems where transactions residing at nodes of a communication graph operate on shared, mobile objects. A transaction requests the objects it needs, executes once those objects have been assembled, and then sends the objects to other waiting transactions. We study scheduling algorithms with provable performance guarantees. Previously, only the *offline batch scheduling* setting was considered in the literature where transactions are known a priori. Minimizing execution time, even for the offline batch scheduling, is known to be NP-hard for arbitrary communication graphs. In this paper, we analyze for the very first time scheduling algorithms in the *online dynamic scheduling* setting where transactions are not known a priori and the transactions may arrive online over time. We provide efficient and near-optimal execution time schedules for dynamic scheduling in many specialized network architectures. The core of our technique is a method to convert offline schedules to online. We first describe a centralized scheduler which we then adapt to a purely distributed scheduler. To our knowledge, these are the first attempts to obtain provably efficient online execution schedules for distributed transactional memory.

Keywords Transactional memory · Distributed systems · Execution time · Data-flow model · Dynamic scheduling

1 Introduction

Concurrent processes (threads) need to synchronize to avoid introducing inconsistencies in shared data objects. Traditional synchronization mechanisms such as locks and barriers have well-known downsides, including deadlock, priority inversion, reliance on programmer conventions, and vulnerability to failure or delay. *Transactional memory* [18,37] (TM) has emerged as an alternative. Using TM, concurrent code is split into *transactions*, blocks of code that appear to execute atomically with respect to one another. Transactions are executed *speculatively*: synchronization conflicts or failures may cause an executing transaction to *abort* with its effects rolled back and then the transaction is restarted. In the absence of conflicts or failures, a transaction typically *commits*, causing its effects to become visible to all threads.

TM has fundamentally changed the way programming is done in multi-core computers both from theoretical and practical aspect and it is an active area of research in both academia and industry [1]. Several commercial processors provide direct hardware support for TM such as (Intel) Haswell [20], (IBM) Blue Gene/Q [16], (IBM) zEnterprise EC12 [28], and (IBM) Power8 [7]. There are proposals for adapting TM to clusters of GPUs [4,12,21]. TM is predicated to be widely used in distributed systems, going beyond GPUs and clusters.

Here, we consider TM in distributed networked systems which are widely available nowadays and there is a growing interest in implementing TM in them [4,19,26,36,39]. In a distributed TM, there is an underlying network modeled as a weighted graph G. Each transaction resides at a node of G and requires one or more shared objects for

A preliminary version of the paper appears in the Proceedings of the 34th International Parallel and Distributed Processing Symposium (IPDPS 2020), pp. 874–883, New Orleans, Louisiana, May 2020.

✉ Costas Busch
kbusch@augusta.edu

Maurice Herlihy
herlihy@cs.brown.edu

Miroslav Popovic
miroslav.popovic@rt-rk.uns.ac.rs

Gokarna Sharma
sharma@cs.kent.edu

[1] Augusta University, Augusta, GA, USA

[2] Brown University, Providence, RI, USA

[3] University of Novi Sad, Novi Sad, Serbia

[4] Kent State University, Kent, OH, USA

Published online: 20 November 2021

read or write. Particularly, we consider a *data-flow* model of transaction execution [19,36], in which each transaction executes at a node, but data objects are mobile and move to the nodes (transactions) that need them. A transaction initially requests the objects it needs, and executes only after it has assembled them. When the transaction commits, it releases its objects, possibly forwarding them to other waiting transactions.

Execution time is a fundamental metric for any computing system. In a multi-core TM, execute time is primarily dominated by the costs of handling data conflicts. In contrast, in a distributed TM, execution time is dominated by the costs of moving objects from one transaction to another. The goal of a *transaction scheduling algorithm* (sometimes called *contention management*) is to minimize delays caused by data conflicts and data movement.

We consider a synchronous model where time is divided into discrete steps [6]. At any time step, a node may perform three actions: (1) it may receive objects from adjacent nodes, (2) it executes any transaction that has assembled its required objects, and (3) it may forward objects to adjacent nodes. A transaction's execution step models when it commits, i.e., that transaction may have started earlier, but may have been blocked while assembling the objects it needed.

In this paper, we consider for the very first time to analyze the *online dynamic scheduling* setting where transactions are not known a priori. In addition, transactions may arrive online and continuously over time. This departs significantly from the literature which studied and analyzed only the *offline batch scheduling* setting [5] where all transactions were known at beginning of time.

We provide online algorithms to compute conflict-free execution time schedules. Each node issues one transaction at a time requesting to access one or more shared objects. Each object is generated at some node and moves to the transactions that request it. The objective is to minimize the total execution time (makespan) until all transactions complete. The execution schedule determines the time steps when each transaction executes and commits. After a transaction commits, it forwards its objects to the respective next requesting transactions in the execution order that depend on these objects. Typically, an object is sent along a shortest path, implying that the transfer time depends on the distance in G between the sender and receiver nodes. Hence, the execution time depends on both the objects' traversal times and inter-transaction dependencies.

It is known [6], through a reduction from vertex coloring, that for batch problems determining the shortest execution time in arbitrary communication graphs is NP-hard, and even hard to approximate within a sub-linear factor of n, the number of nodes in G. This hardness result also applies to online problems which are more general than batch problems. Therefore, we study online scheduling algorithms from a widely-studied notion of *competitiveness*—the ratio of total execution time produced by a designed algorithm to the smallest execution time achievable by any optimal offline algorithm. The goal is to make the competitive ratio as small as possible (the best possible is 1 which is hard to achieve).

1.1 Contributions

Since the online competitive ratio is hard to approximate in arbitrary networks, we focus on various specific distributed computing architectures: Clique, Hypercube, Butterfly, Grid, Line, Cluster, and Star. All these are popular topologies for a variety of applications in multiprocessors, networks-on-chip, rack-scale or cluster-scale distributed systems [11,27,30].

We consider scheduling problems where each node holds a single transaction at any moment of time, and each transaction requests up to k arbitrary objects. Below we list the competitive ratios resulted from our algorithms assuming that there is a centralized scheduler that decides the execution schedule. These schedules can also be computed by a decentralized scheduler which affects the competitive ratios by a poly-log factor overhead.

- *Clique* in a clique (complete graph) of n nodes we give an online schedule which is $O(k)$ competitive.
- *Hypercube, Butterly, Grids* in a hypercube with n nodes we give an online schedule which is $O(k \log n)$ competitive. The same bound holds also for the butterfly and $\log n$-dimensional grids.
- *Line* in a line graph of n nodes we give an online schedule which is $O(\log^3 n)$ competitive. Note that for the line graph the competitiveness does not depend on k.
- *Cluster* we consider a cluster graph which consists of cliques with β nodes each connected to each other through bridge edges of weight $\gamma \geq \beta$. We show that there is a schedule which is $O(\min(k\beta, \log_c^k m) \cdot \log^3(n\gamma))$-competitive for some constant c.
- *Star* in the star graph topology there is a central node that connects to rays each consisting of β nodes. We obtain a schedule which is $O(\log \beta \cdot \min(k\beta, \log_c^k m) \cdot \log^3 n)$ competitive, for some constant c.

1.2 Techniques

We introduce two techniques to produce the dynamic schedules which are suitable for different network topologies. The first is a direct approach that is suitable to small diameter graphs, while the second is an indirect approach that converts offline batch schedules to online dynamic schedules.

The first technique constructs *online greedy schedules* based on continuously calculating a valid vertex coloring

on the (dynamic) dependency graph of the transactions. Since the optimal chromatic number is hard to approximate in arbitrary graphs, this technique can be only effective in specialized network graphs as for example those with low diameter. Our competitive bounds on the clique, hypercube, butterfly, and $\log n$-dimensional grid are obtained from this technique.

The second technique is more general and it converts arbitrary offline batch schedules to online dynamic schedules. Thus, known batch scheduling results can be adapted to the online setting. The impact to the approximation of the offline schedule is a $O(\log^3(nD))$ factor delay, where D is the graph diameter. This is bigger overhead than the direct method above but the technique can be used to arbitrary graphs including those with larger diameter. With this approach the offline schedules by Busch et al. [5] are converted to online schedules for the line, cluster, and star topologies, which can be graphs with large diameter.

The conversion from offline to online (second technique) is achieved by repeatedly dividing the transactions into *buckets of transactions*, where each bucket is a set of transactions that can be processed using a batch scheduler. When a transaction is generated it is assigned to one of the buckets according to its current dependencies with other existing transactions. The transaction remains in its bucket until it is scheduled to execute.

Each bucket has a level according to the scheduling time for the batch problem of the transactions within. The bucket B_i, at level i, corresponds to a batch problem that takes up to 2^i time steps to execute its transactions. The bucket B_i is processed periodically every 2^i time steps, which gives an opportunity for new dynamically generated transactions to accumulate within the bucket between processing times. When a bucket is processed, the transactions within the bucket are scheduled using a batch scheduler.

There are dependencies between the levels of the buckets, because new transactions in higher level buckets may conflict with existing transactions in lower level buckets. The buckets are processed from lower level to higher level to handle inter-dependencies between various levels. In this way, the periodically accumulated batch problems in the buckets are scheduled sequentially producing a global online schedule.

The main benefit of the buckets is that transactions with small number of conflicts (calibrated to the object distances), which appear in lower level buckets, can make progress faster than transactions with larger number of conflicts in higher levels. In the analysis we show that if a batch scheduling algorithm has approximation ratio $b_{\mathcal{A}}$ then the online schedule is $O(b_{\mathcal{A}} \log^3(nD))$ competitive. The poly-log factor is the price of separating the transactions into buckets, which however makes the online schedule feasible.

When we present the basic bucket scheduling algorithm in Sect. 4 we assume for simplicity of presentation that it is implemented by a central authority that has instant knowledge about all the current transactions and objects. In Sect. 5 we present a decentralized version of the bucket algorithm which is based on a sparse cover decomposition of the graph that gives a $O(b_{\mathcal{A}} \log^9(nD))$ competitive schedule.

1.3 Related work

The most closely related work to ours is due to Busch et al. [5], where they provide efficient execution time schedules for offline batch scheduling on the data-flow model in specialized graphs likely to arise in practice: Clique, Line, Grid, Cluster, Hypercube, Butterfly, and Star. Specifically, the competitive ratios for execution time are $O(k)$ in Clique, $O(1)$ in Line, $O(k \log n)$ in Grid, Butterfly, and Hypercube, $O(\min(k\beta, \log_c^k m))$-competitive in Cluster, and $O(\log \beta \cdot \min(k\beta, \log_c^k m))$-competitive in Star. Although the algorithms are optimal (or near-optimal) for the batch scheduling setting, the techniques do not apply to dynamic scheduling. The algorithms we obtain in this paper for the specialized graphs through the bucket technique work in the online dynamic setting with only poly-log increase in competitive ratios. They also provide a non-trivial lower bound on execution time, improving significantly on the trivial TSP lower bound. In the lower bound, they show that there is a scheduling problem on the grid, with 2 objects per transaction, where every schedule must have execution time $\Omega(n^{1/40}/\log n)$ factor away from the optimal TSP tour length of any object. The same lower bound holds also for trees. These lower bounds also apply to the dynamic setting, since batch problems can be viewed as special execution scenarios of dynamic problems.

In another work, Busch et al. [6] consider the problem of minimizing both the execution time and *communication cost* (the total distance travelled by all the objects in G) simultaneously for transaction scheduling for distributed TMs under the data-flow model. They show that it is impossible to simultaneously minimize execution time and communication cost, that is, minimizing execution time implies high communication cost (and vice-versa), and they provide respective trade-off bounds. In particular, if the execution time (communication cost) is with in a $O(k')$ factor from optimal, then the communication cost (execution time) is a $\Omega(n^{1/4}/k')$ factor sub-optimal, where $1 \leq k' \leq n^{1/4}/16$. They also give efficient algorithms minimizing either execution time or communication cost individually in arbitrary communication graphs, where the result for execution time is sub-optimal due to the known inapproximability of the problem. Specifically, the algorithm minimizing communication cost is $O(\log^4 n/\log\log n)$-competitive and the algorithm

minimizing execution time is $O(\Delta)$-competitive, where Δ is the maximum number of conflicts between transactions. All these results were established in the offline batch scheduling setting. They also sketch an approach for dynamic scheduling based on graph coloring, but that approach does not provide any competitive ratio bounds for the execution time as we do here.

Recently, Poudel et al. [31] studied the problem of scheduling transactions that need to be committed in the given predefined order, which is called the *predefined order scheduling* problem (originally defined in the context of tightly-coupled multi-core systems [13,33]). Poudel et al. considered the data-flow model and established an optimal algorithm in the batch setting and $O(\log^2 n)$-competitive and $O(D)$-competitive algorithms in the online and dynamic settings, respectively. This problem is different and the technique developed cannot be extended to the case of no such predefined order requirement as in this paper.

There are also other previous works [2,19,23,36,38] on the data-flow model of distributed TMs which focus on minimizing only the communication cost for scheduling problem instances with only a single shared object. Herlihy and Sun [19] presented a $O(\log D)$-competitive algorithm for metric-space networks which Sharma and Busch [36] extended to general networks with $O(\log^2 n \cdot \log D)$ competitive ratio. These algorithms are based on a *hierarchical clustering* which is built up on a multi-level distance clustering of the nodes around independent sets.

An alternative approach of *spanning tree based clustering* is used in [2,38]. The competitive ratios are $O(D_T)$, where D_T is the diameter of the spanning/overlay tree used. Kim and Ravindran [23] provide communication cost bounds for special workloads and problem instances with multiple shared objects. The execution time minimization is considered by Zhang et al. [39]. where they use TSP tours for object paths, which for arbitrary graphs can lead to significantly sub-optimal results according to a non-trivial lower bound [5]. Specifically, they showed that an online deterministic scheduling algorithm, that always runs a maximal set of non-conflicting transactions, provides an $\Omega(\max(s, s^2/\overline{D}))$ competitive ratio for s shared objects in a network with normalized diameter $\overline{D}$. They then designed a randomized scheduling algorithm that achieves average-case competitive ratio $O(s \cdot \phi_A \cdot \log^2 m \cdot \log^2 n)$ for n transactions invoked by m nodes, where ϕ_A is an approximation ratio of a TSP algorithm. They further showed for the case of $s = 1$ that finding the optimal execution time is NP-hard because it is equivalent to finding the TSP path.

Several papers [4,9,24,26] present techniques to implement distributed TMs. However, they either use global lock [26], serialization lease [24], or commit-time broadcasting [4,9] which may not scale well with the size of the network. Moreover, several other papers study distributed TMs employing replication and multi-versioning [22,26].

In the *control-flow* model [34], opposite to the data-flow model, objects are immobile and transactions either move to the network nodes where the required objects reside, or invoke remote procedure calls. Hendler et al. [17] studied a lease based hybrid (combining data-flow with control-flow) distributed TM which dynamically determines whether to migrate transactions to the nodes that own the leases, or to demand the acquisition of these leases by the node that originated the transaction. Palmieri et al. [29] present a comparative study of data-flow versus control-flow models for distributed TMs in partially-replicated environments. Others have studied the speculative transaction execution [32] in replicated environments, and transaction scheduling using consistent snapshots [22,26,32] for replicated and multiversioning environments. All these works provide no theoretical analysis of execution time or communication cost.

Both offline batch and dynamic transaction scheduling problems were widely studied in tightly-coupled multi-core systems without involving a communication network. Several scheduling algorithms with provable upper and lower bounds, and impossibility results were given [3,10,14,35]. Additionally, predefined order scheduling has been studied in tightly-coupled multi-core systems [13,33].

1.4 Roadmap

In Sect. 2 we give the model and preliminaries. We give an online greedy algorithm in Sect. 3. We present the bucket algorithm in Sect. 4, and discuss its decentralized version in Sect. 5. We conclude in Sect. 6.

2 Model and preliminaries

Consider a simple graph $G = (V, E, w)$, with nodes V, edges E and an edge weight function $w : E \to \mathcal{Z}^+$. Let D be the diameter of the graph, which is the maximum length of any shortest path between any pair of nodes. We consider a synchronous communication model where all actions occur at discrete time steps. For an edge $e \in E$ it takes $w(e)$ time steps to send a message between its nodes.

Transactions are generated continuously over time. We assume that a transaction T is generated and resides in some node of G and requests a set of objects $O(T)$ for read or write. Transaction T executes once it acquires all the objects in $O(T)$. For simplicity, we assume that the transaction executes instantly at the time step that it gathered all the objects. Thus, all delays in our model are due to communication. A transaction is considered live until the time step it executes at. After that the transaction has completed execution and is not live anymore. We assume that the set of objects $O(T)$

accessed by transaction T are assumed to not change at runtime. However, the values at each object may get updated due to the changes written on it by other transactions that finished execution previously.

We have that at all times an object o_i has a single writable copy which can be modified/updated and possibly multiple readable copies which can only be read.

We assume that an object is created at some time step at some node of G by a transaction. At any time t, an object o_i either has been acquired by some transaction where the object resides at the node of the transaction, or the object o_i is in transit in the graph from one transaction to another. If a transaction T acquired an object and subsequently T finished execution, then the object will remain at the node of T until some other transaction requests it and the scheduler moves the object to that transaction.

At any time t the *latest transaction* of an object o_i, denoted $L_t(o_i)$, is the transaction T that holds the object o_i at time t, or if there is no such T (i.e. the object is in transit), it is the last transaction that acquired or generated o_i before time t. Let $L_t(T) = \bigcup_{o_i \in O(T)} L_t(o_i)$ denote the set of all last transactions of the objects of transaction T at time t. The meaning of $L_t(T)$ is that in order to execute transaction T which is generated at time t, all its objects in $O(T)$ need to be fetched from the previous transactions in $L_t(T)$ to T and then T is ready for execution.

Let $\mathcal{T}_t$ denote the set of all live transactions at time t. Let $L(\mathcal{T}_t) = \bigcup_{T \in \mathcal{T}_t} L_t(T)$ be the set of latest transactions for the objects to be used by the transactions in $\mathcal{T}_t$.

In an online execution schedule S each transaction is executed at some designated time step. Consider a transaction T generated at time t. Suppose that the transaction executes at time $t_T > t$ in schedule S. The *execution duration* of T in schedule S is the time difference $t_T - t$. Let t^* denote the optimal time duration to execute all the transactions in $\mathcal{T}_t$, given the execution times of the transactions in $L(\mathcal{T}_t)$. The competitive ratio for S at time t is $r_S(t) = \max_{T \in \mathcal{T}_t}((t_T - t)/t^*)$. The competitive ratio for S is $r_S = \sup_t r_S(t)$.

Definition 1 (*Algorithm competitive ratio*) For online scheduling algorithm $\mathcal{A}$, the competitive ratio $r_{\mathcal{A}}$ is the maximum competitive ratio over all possible execution schedules $\mathcal{S}$ that it produces, $r_{\mathcal{A}} = \sup_{S \in \mathcal{S}} r_S$. (We also say that $\mathcal{A}$ is $r_{\mathcal{A}}$-competitive.)

A feature of the scheduling algorithms that we describe in the next sections is that the execution times of the new transactions are not affecting the already determined execution times of previously scheduled transactions. With this feature we can still manage to obtain good competitive competitive ratios. Scheduling algorithms with this feature can be appealing in practice because future events are not affecting the currently scheduled transactions.

3 Online greedy schedule

We describe a generic scheduling algorithm which can be applied for arbitrary graphs. The algorithm is near optimal for interesting special cases of small diameter graphs. The basic idea is to give a greedy coloring for a transaction conflict dependency graph H where colors will be translated to execution times. The challenge is that the dependency graph changes over time and the greedy schedule is constrained by the transactions that have already been scheduled. We start with some basic definitions and results on graph coloring.

3.1 Weighted graph coloring

Consider a simple graph $H = (V_H, E_H, w)$ with a weight function $w : E_H \to \mathcal{Z}^+$ on its edges. For each $v \in V_H$ let $N(v) \subseteq V_H$ denote the neighborhood of v which is the set of adjacent nodes of v in H. The *degree* of v is $\Delta(v) = |N(v)|$. The *weighted degree* of v denoted $\Gamma(v) = \sum_{u \in N(v)} w((u, v))$, is the sum of the weights of the edges adjacent to v in H.

A valid coloring $c : V_H \to \mathcal{Z}^*$ of H is an assignment of integer values (colors) to the nodes of H such that for any two adjacent nodes their respective colors differ by at least the weights of their edges. Namely, for any $(u, v) \in E_H$,

$$|c(u) - c(v)| \geq w((u, v)). \tag{1}$$

A *partial coloring* of H is an assignment of colors to a subset of its vertices. A partial coloring is valid as long as Eq. 1 is satisfied for each pair of nodes that have received a color. We continue with a result that assigns a valid color to a node assuming that some other nodes may have already received a color.

Lemma 1 *Given an arbitrary valid partial coloring of a set $V' \subseteq V_H$, any node $v \in V_H \backslash V'$ can be assigned a valid color $c(v) \leq 2\Gamma(v) - \Delta(v)$.*

Proof Let $Q \subseteq N(v) \cap V'$ denote the neighbors of v that have already received a color. We scan for an available color for v sequentially starting from $c(v) = 0$. If there is a node $u \in Q$ such that $|c(u) - c(v)| < w((u, v))$, then $c(v)$ is not a valid color, and we set $c(v) = c(u) + w((u, v))$. We repeat this process until we find an available valid color for v. Note that each time we update $c(v)$ one less node $u \in Q$ will need to be considered later, and the value of $c(v)$ is increased by at most $2w((u, v)) - 1$, in case u had higher color than v. Therefore, this process is repeated at most $|Q|$ steps giving a valid resulting color for v with value at most:

$$\begin{aligned} c(v) &\leq \sum_{u \in Q} (2w((u, v)) - 1) = 2 \sum_{u \in Q} w((u, v)) - |Q| \\ &\leq 2\Gamma(v) - \Delta(v). \end{aligned}$$

□

We can obtain an improved version of Lemma 1 for the case were all edge weights are equal.

Lemma 2 *If all edges have the same weight β, given an arbitrary valid partial coloring of a set $V' \subseteq V_H$ with $c(u) = k_u\beta$ for each $u \in V'$, for some constant $k_u \geq 0$, then each node $v \in V_H \backslash V'$, can be assigned a valid color $c(v)$ such that $c(v) = k_v\beta$, for some constant $k_v \geq 0$, and $c(v) \leq \Gamma(v)$.*

Proof Let $Q \subseteq N(v) \cap V'$ denote the neighbors of v that have already received a color. Each $u \in N(v)$ may use a color $c(u) = k_u\beta$. Since $|N(v)| = \Delta(v)$, by pigeonhole principle, there is a k_v, $0 \leq k_v \leq \Delta(v)$, such that $k_v \neq k_u$, for any $u \in N(v)$. Therefore, we can set $c(v) = k_v\beta$. Note that such a choice of color gives $|c(v) - c(u)| \geq \beta$ for each $u \in N(v)$, and hence $c(v)$ is valid. Moreover, $c(v) \leq \beta\Delta(v) = \Gamma(v)$. □

3.2 Coloring-based schedule

Let $\mathcal{T}_t^g \subseteq \mathcal{T}_t$ denote the newly generated transactions at time t (recall that $\mathcal{T}_t$ are all live transactions at time t). In the greedy schedule, all the newly generated transactions $\mathcal{T}_t^g$ get immediately assigned (at time t) a designated execution time which remains unchanged thereafter. The challenge is to schedule the newly generated transactions $\mathcal{T}_t^g$ based on the existing schedules of the previously generated transactions. Moreover, some objects might be in transit at time t complicating the scheduling task. Below we describe how to resolve these scheduling challenges by creating an appropriate dependency graph for the transactions at time t. The dependency graph will be then colored to provide the resulting execution schedules for the transactions in $\mathcal{T}_t^g$.

3.2.1 Dependency graphs

At time t an object o_i is either in transit or it resides at the node of the latest transaction $L_t(o_i)$ that held o_i. If the object o_i is in transit then assume that at time t it resides at some node $v_t(o_i)$ along a shortest path connecting $L_t(o_i)$ to the next node that requested o_i. In case that at time t the object o_i is in transit along an edge (u, v) (already left u going to v) of the shortest path, then we will assume in the analysis that $v_t(o_i)$ is an artificial node with an edge connecting it to v with weight equal to the time remaining to reach v. Assume also for the sake of simplifying the analysis that $v_t(o_i)$ has a temporary artificial transaction T that uses o_i and executes at time t.

Let $Z_t(o_i)$ denote the current transaction that holds the object at time t, which is either the latest transaction $L_t(o_i)$ or the temporary transaction in $v_t(o_i)$. Denote by $Z_t(T) = \bigcup_{o_i \in O(T)} Z_t(o_i)$ the set of current transactions that hold all the objects of T at time t. Let $Z(\mathcal{T}_t) = \bigcup_{T \in \mathcal{T}_t} Z_t(T)$ denote all the current transactions that hold the objects of the live transactions at time t.

Consider the live transactions $\mathcal{T}_t$ at time t. Two transactions $T_1, T_2 \in \mathcal{T}_t$ *conflict* if $O(T_1) \cap O(T_2) \neq \emptyset$. The *conflict set* $C_t(T)$ of a transaction $T \in \mathcal{T}_t$ is the set of live transactions in $\mathcal{T}_t$ that it conflicts with at time t. The *extended conflict set* $C'_t(T)$ of a transaction T at time t includes $C_t(T)$ and all the current transactions $Z_t(T)$ of the objects in $O(T)$, namely, $C'_t(T) = C_t(T) \cup Z_t(T)$.

The *dependency graph* H_t of transactions at time t, represents the conflicts of the transactions at time t. The set of nodes $V(H_t)$ in H_t correspond to the live transactions, that is, $V(H_t) = \mathcal{T}_t$. The set of edges $E(H_t)$ in H_t correspond to conflicts between transactions, that is, $(T_1, T_2) \in E(H_t)$ if $T_2 \in C_t(T_1)$ (and symmetrically $T_1 \in C_t(T_2)$). The graph H_t is actually weighted, such that the weight of an edge represents the distance between the respective transactions in the original graph G.

Let $\mathcal{T}'_t = \mathcal{T}_t \cup Z(\mathcal{T}_t)$ denote the *extended set* of live transactions at time t. We can define the *extended dependency graph* H'_t with respect to the extended conflict sets of transactions. The set of nodes in H'_t is the extended set of live transactions $\mathcal{T}'_t$ at time t namely, $V(H'_t) = \mathcal{T}'_t$. The set of edges in H'_t corresponds to transaction conflicts in the extended set of live transactions so that $(T_1, T_2) \in E(H'_t)$ if $T_2 \in C'_t(T_1)$ (and symmetrically $T_1 \in C'_t(T_2)$).

3.2.2 Greedy scheduling

A *valid coloring* of a set of transactions $S \subseteq V(H_t)$ (or $S \subseteq V(H'_t)$) assigns a unique positive integer (color) to each transaction in S such that any two adjacent transactions in S receive colors which differ by at least the weight of the incident edge that connects them in H_t (or H'_t). A valid coloring can translate to an execution schedule such that the colors assigned to the transactions correspond to the distinct time steps that transactions execute. Since at time t some objects may be in transit, a coloring of H_t alone may not be adequate to provide a realistic schedule without knowing the current positions of the objects. For this reason, we consider a valid coloring based on the extended dependency graph H'_t which includes the current transactions that hold the objects at time t including the temporary transactions for the objects in transit.

Algorithm 1 has the details of the greedy scheduling algorithm. The main objective is to give a valid coloring to all the newly generated transactions in $\mathcal{T}_t^g$ assuming the existing schedules of previously generated transactions and the current positions of the objects. Since each transaction in $\mathcal{T}'_t \backslash \mathcal{T}_t^g$ has already been scheduled, it is assumed to have a color equal to its scheduled execution time minus t (the remaining time until execution of the transaction). As a con-

Algorithm 1: Online Greedy Schedule

1 **foreach** *time step* t **do**
2 Let $\mathcal{T}_t^g$ be the transactions generated at time t;
3 Let H_t' be the extended conflict graph of the (extended) live transactions $\mathcal{T}_t'$ at time t;
4 Transactions that have already been scheduled ($\mathcal{T}_t' \backslash \mathcal{T}_t^g$) are assumed to have a color in H_t' equal to their assigned execution time minus t;
5 Assign greedily a color $c(T)$ to each transaction $T \in \mathcal{T}_t^g$ by repeatedly applying Lemma 1 to uncolored transactions of H_t';
6 Each transaction $T \in \mathcal{T}_t^g$ is scheduled to execute at time $t + c(T)$;

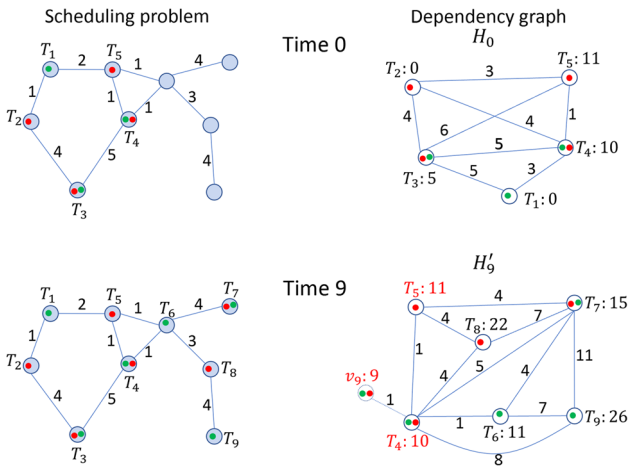


Fig. 1 An example of greedy coloring

sequence, those transactions in $\mathcal{T}_t' \backslash \mathcal{T}_t^g$ that are executing at time t get color 0.

The algorithm applies repeatedly Lemma 1 on the extended dependency graph H_t' to assign a color to each transaction of $\mathcal{T}_t^g$. The color is then translated to an execution time by adding the current time t. This way of coloring assumes that the objects move to the next scheduled transactions that use them by following shortest paths in G.

An example of greedy scheduling is shown in Fig. 1. In this example there is a weighted graph G with nine nodes, where the labels of the edges are their respective weights, as shown on the top left part of Fig. 1. Each node issues a transaction, for a total of nine transactions $T_1, \ldots, T_9$. There are two objects that are used by the transactions shown as green and blue in Fig. 1. The green object is initially at T_1 and it is accessed by transactions $T_1, T_3, T_4, T_6, T_7, T_9$, while the red object is initially at T_2 and it is accessed by transactions $T_2, T_3, T_4, T_5, T_7, T_8$. Transactions $T_1, \ldots, T_5$ are generated at time 0 with respective dependency graph H_0 (identical with extended dependency graph H_0') as shown on the top right of Fig. 1, where an edge between two transactions denotes a conflict of accessing the same object and the weight denotes the shortest path distance between the transactions in G. Algorithm 1 gives a color to each transaction which appears next to the transaction label in Fig. 1 right side. Namely, transactions T_1, T_2, T_3, T_4, T_5 receive respective colors 0, 0, 5, 10, 11, which are also the actual times that the transactions are scheduled to execute. Transactions $T_6, \ldots, T_9$ are generated at time 9 with respected extended dependency graph H_9' as shown at the bottom right of Fig. 1. Graph H_9', includes transactions T_4 and T_5 that are previously scheduled but not executed yet, and also the artificial node v_9 which holds the two objects at time 9 while they are in transit from T_3 to T_4. For v_9 we only depict its conflict with T_4, and we omit the conflicts with the other transactions to avoid clutter. After we apply Algorithm 1, the transactions T_6, T_7, T_8, T_9 receive respective times for execution 11, 15, 22, 26.

We continue with an analysis of Algorithm 1. Let $\Delta_t'(T_i)$ and $\Gamma_t'(T_i)$ denote the respective regular and weighted degree of a transaction T_i in the extended dependency graph H_t'.

Theorem 1 (Greedy online schedule) *There is an execution schedule such that each transaction $T_i \in \mathcal{T}_t^g$ generated at time t executes by time $t + 2\Gamma_t'(T_i) - \Delta_t'(T_i)$.*

Proof Given an arbitrary valid coloring of the already scheduled transactions $T_t' \backslash \mathcal{T}_t^g$ in H_t', by repeatedly applying Lemma 1 to uncolored transactions of H_t', we obtain a valid coloring of the newly generated transactions $\mathcal{T}_t^g$ in H_t' such that each transaction $T_i \in \mathcal{T}_t^g$ can receive a color at most $2\Gamma_t'(T_i) - \Delta_t'(T_i)$.

We can sort the transactions that request any specific object according to their respective $t + c$ where t is the generation time of the transaction and c is its respective color. An object will move from node to node in ascending time order. According to this schedule, each object in $O(T_i)$ will reach T_i no later than time $t + c$, since the valid coloring gives enough time for each object to be transferred to T_i by that time. Therefore, a transaction T_i will execute by time no later than $t + c \leq t + 2\Gamma_t'(T) - \Delta_t'(T)$. □

In Algorithm 1 if we replace Lemma 1 with Lemma 2, we obtain an improved result for the case where all edges in G have the same weight β. Here, we can execute the transactions at time steps which are multiples of β. The proof of the following result is similar to the proof Theorem 1 but with the use of Lemma 2.

Theorem 2 (Schedule for uniform weights) *If all the edges of the graph G have the same weight, then there is an execution schedule such that each transaction T_i generated at time t executes by time $t + \Gamma_t'(T_i)$.*

For the sake of completeness we next show that the sequential time complexity of Algorithm 1 is polynomial with respect to the parameters of the problem. Note that,

according to the execution model we described in Sect. 2, the sequential steps to execute Algorithm 1 are subsumed within a single time step t of the concurrent execution, since the network communication delay is more detrimental to the overall execution time. Thus, we only present a high level overview of its complexity.

For each time step t, the sequential run time complexity of Algorithm 1 is polynomial to the size of the extended dependency graph H'_t at time t. Specifically, it is $O(n' + m' \log n')$ where n' and m' are the respective number of nodes (transactions) and edges of H'_t (note that H'_t may not be connected with possibly having $m' < n'$). The term n' in the asymptotic notation is because the algorithm applies Lemma 1 for each of the n' nodes. Moreover, each node v with $\Delta'(v) = k$ neighbors in H'_t can be processed by first sorting the neighbors according to their current colors (requires $O(k \log k)$ steps), and then checking if the new color choices conflict with each neighbor node (requires $O(k)$ steps). Thus, this part requires $O(k \log k)$ steps in total per node. Considering all n' nodes, we get total steps $\sum_{v \in V(H'_t)} \Delta'(v) \log \Delta'(v)$. Since $\Delta'(v) \leq n'$ and $\sum_{v \in V(H'_t)} \Delta'(v) = 2m'$, we get $O(m' \log n')$ steps for processing all nodes for this part. Therefore, combining with the first term n', the overall number of steps is $O(n' + m' \log n')$.

We continue to apply Algorithm 1 in several special case graphs, such as the complete graph, hypercube and butterfly.

3.3 Complete graph

3.3.1 Scheduling problem

Consider an unweighted complete graph (clique) G with n nodes where every node is connected to every other node with an edge of weight 1. Every node holds one transaction. Each transaction requests an arbitrary set of k objects. Once a transaction completes execution, the node of the transaction issues at the next time step a new transaction requesting an arbitrary set of k objects (possibly different than the previous set). The process repeats.

3.3.2 Algorithm and analysis

We use the greedy schedule of Algorithm 1 (with Lemma 2 instead of Lemma 1). Consider a transaction T_i that is generated at time t. From Theorem 2, T_i can execute by time $t + \Gamma'_t(T_i)$. Suppose that T_i uses objects $O(T_i) = \{o_{i_1}, \ldots, o_{i_k}\}$. Suppose also that each object o_{i_j} is used by l_{i_j} transactions in $\mathcal{T}'_t$. Then, since the edge weights are all $\beta = 1$, the weighted degree in the extended dependency graph is $\Gamma'_t(T_i) \leq \sum_{j=1}^{k} l_{i_j} \leq k l_{\max}$, where $l_{\max} = \max_j l_{i_j}$. Thus, T will execute by time no later than $t + k l_{\max}$.

The transactions scheduled at time t are not affected by the transactions generated at later time. For the transactions generated at time t, the time duration to execute all of them is at least $l_{\max}$, since at least so many transactions at time t request the same object, and that object has to be transferred to all of these transactions. Therefore, we obtain the following result:

Theorem 3 (Complete graph) *In the complete graph, the greedy online schedule of Algorithm 1 has competitive ratio* $O(k)$.

3.4 Hypercube and related graphs

In a hypercube graph [25] with n nodes any pair of nodes is connected with a path of length at most $\log n$ edges (logarithm is base 2). We can represent the hypercube graph as a complete graph with n nodes, where the weight of an edge ranges between 1 and $\log n$ and represents the distance of the respective path between the incident nodes in the hypercube. Assume for simplicity that the weights in the complete graph are all set to the worst case uniform value $\beta = \log n$. Using the analysis for the complete graph, if we apply Algorithm 1 and Theorem 2, we get that any transaction T generated at time t executes by time $t + \Gamma'_t(T)$. Since the (worst-case) edge weights are $\beta = \log n$, for k distinct objects the weighted degree is $\Gamma'_t(T) \leq \beta \sum_{j=1}^{k} l_{i_j} \leq \beta k l_{\max}$, where $l_{\max} = \max_j l_{i_j}$. Thus, T will execute no later than time $t + \beta k l_{\max}$. Since $l_{\max}$ is a lower bound on the execution time, we get that the resulting execution schedule has competitive ratio $O(\beta k) = O(k \log n)$.

The same result applies to other networks where the maximum distance between nodes is bounded by $\beta = O(\log n)$, as for example in butterfly networks [25] and $\log n$-dimensional grids [8].

Note that Theorem 2 is useful for getting upper bounds for the worst case scenario, but Algorithm 1 with Theorem 1 may give better execution schedules when used in practice.

4 Online bucket schedule

The online greedy schedules described in the previous section are obtained assuming a central authority with instant knowledge about the current positions of all the transactions and objects in the system. However, in reality such a centralized authority may not exist since transactions and objects are created in a distributed manner independent of each other.

Since all the graphs considered in the previous section have small diameter, $O(\log n)$, a simple remedy is to have a designated node in G to collect all the information as new transactions are generated and objects move. With this, each actual time step of the execution can be simulated with

$O(\log n)$ time steps which is enough time to collect the information in the designated node and then decide on the actions of the next step of the execution. Thus, all the upper bounds on the execution schedule can be scaled with a $O(\log n)$ factor, proportional to the graph diameter.

Later, in Sect. 5, we will give a more elegant decentralized solution with smaller dependence on the graph diameter and hence that solution is more and suitable for larger diameter graphs. Since that solution will involve a hierarchical decomposition of the graph, its overhead to the overall schedule is a higher order poly-log factor. Thus, it does not benefit significantly the low diameter graphs which were considered in the previous section.

In this section, we discuss an approach that coverts the offline scheduling algorithms to online, given a central authority. These online algorithms will then be modified to operate in a decentralized manner without requiring a central authority. The benefit is that the offline batch algorithms designed for specialized graphs in our previous work [5] can immediately be applied to obtain execution schedules on the online setting. Specifically, consider an arbitrary offline batch scheduling algorithm $\mathcal{A}$ which can schedule any given set of transactions in graph G. We will convert $\mathcal{A}$ to an online algorithm with a technique that uses buckets of transactions.

Before we describe the conversion technique, we need to perform a simple modification to $\mathcal{A}$ to allow it to operate even if some of the batch transactions under consideration have already been scheduled. We can achieve this by computing an execution schedule with algorithm $\mathcal{A}$ for the unscheduled transactions which is appended at the end of the existing schedule of the already scheduled transactions. This approach does not alter the execution times of the already scheduled transactions. In the worst-case, the combined execution time is twice as long with respect to the execution time of a schedule that $\mathcal{A}$ would produce if all combined transactions were unscheduled, since the transactions are processed in two disjoint parts. Thus, the asymptotic performance of $\mathcal{A}$ is unaffected by this modification.

Using the proposed modification, let $F_{\mathcal{A}}(X)$ denote the time to execute a set of transactions X where some transactions in X may have already a determined execution schedule.

4.1 The bucket algorithm

We give the transformation of the offline algorithm $\mathcal{A}$ to an online algorithm. The details appear in Algorithm 2.

We will temporarily store into buckets the newly generated transactions at time t, $\mathcal{T}_t^g \subseteq \mathcal{T}_t$, until their execution schedules are determined. Let $\mathcal{T}_t^s \subseteq \mathcal{T}_t$ denote the set of transactions whose schedules have already been determined (the transactions in $\mathcal{T}_t^s$ may execute at t or later according to their scheduled execution time).

A *bucket* B_i at *level* i, where $i \geq 0$, is a set of unscheduled transactions which are expected to execute in at most 2^i time steps from the moment of their generation. There is one bucket per level and the number of levels is bounded by the diameter and nodes of G. At time t, each new transaction $T \in \mathcal{T}_t^g$ will be inserted into the bucket B_i with smallest level i such that $F_{\mathcal{A}}(\mathcal{T}_t^s \cup B_i \cup \{T\}) \leq 2^i$. That is, T is inserted into the smallest level bucket B_i that does not increase its offline execution time beyond the 2^i limit when we add T to the bucket (i.e. when bucket becomes $B_i \cup \{T\}$), given the fixed execution times of the already scheduled transactions in $\mathcal{T}_t^s$.

Buckets get activated periodically so that the transactions within them get actual execution times. Bucket B_i gets activated every 2^i time steps. Once B_i activates, say at time t', all transactions in B_i get scheduled using algorithm $\mathcal{A}$ on the set $B_i \cup \mathcal{T}_{t'}^s$. The generated schedule does not alter the execution times of the already scheduled transactions in $\mathcal{T}_{t'}^s$. Then, we remove the transactions from B_i (which now becomes an empty bucket), since the transactions in B_i are considered scheduled and become part of $\mathcal{T}_{t'}^s$.

The activation times of different levels are not required to be aligned. If however multiple buckets get activated simultaneously at time t', then we schedule transactions of lower level buckets before higher level buckets. Thus, when higher level buckets are scheduled at t' they assume that the lower level bucket transactions are already scheduled and are members of $\mathcal{T}_{t'}^s$.

Figure 2 shows an example of buckets on the line graph. There are seven transactions, $T_1, \ldots, T_7$, which are located in the seven different nodes. The transactions T_1, T_3, T_5, and T_7 require objects (t-variables) that are in their 2^{i-1} proximity and therefore they are added to the bucket B_{i-1}. Similarly, the transactions T_2 and T_6 require objects that are in their 2^i proximity, and therefore they are added to the bucket B_i. Finally, the transaction T_4 requires objects in its 2^{i+1} proximity, and therefore it is added to the bucket B_{i+1}. To clarify, in this example the transactions that are in the same bucket use different objects. Moreover, each double-arrow line in Fig. 2 represents the span of the objects used by the transaction in the center of the line, so that the span is twice the proximity, i.e. the proximity of T_1 is 2^{i-1} and the span is 2^i.

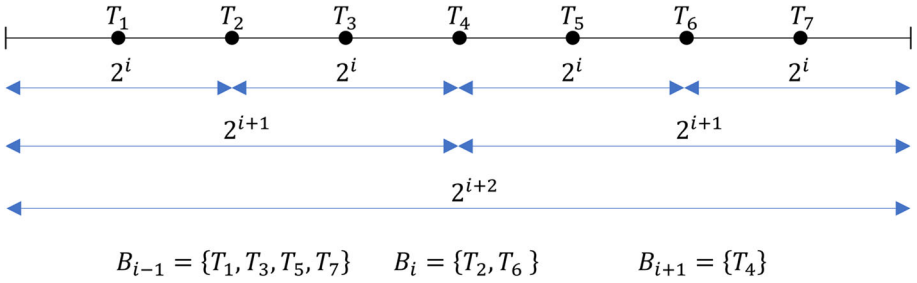
4.2 Analysis of bucket algorithm

We continue with an analysis of Algorithm 2. We first prove two lemmas that lead to the main result in Theorem 4. (Logarithms are base 2.)

Lemma 3 *The maximum level of any bucket is at most* $\log(nD) + 1$.

Proof The worst time execution schedule is when the transactions execute sequentially. At any time step, there can be

Fig. 2 Buckets on the line graph



Algorithm 2: Online Bucket Schedule

```
1 Consider a batch scheduling algorithm A;
2 Assume disjoint buckets B_i at levels i ≥ 0, such that bucket B_i
  activates every 2^i time steps;
3 foreach time step t do
4     Each T ∈ T_t^g will be inserted into the bucket B_i with
      smallest level i such that F_A(T_t^s ∪ B_i ∪ {T}) ≤ 2^i;
5     if B_i activates at time t then
6         All transactions in B_i get scheduled using algorithm A
          with transactions T_t^s ∪ B_i;
7         The resulting schedule does not modify the execution
          times of the already scheduled transactions in T_t^s;
8         B_i becomes empty and all its transactions are inserted
          into T_t^s;
```

no more than n scheduled transactions (one transaction from each node). The maximum distance in G between any pair of transactions is no more than the graph diameter D. Thus, the worst in time schedule will execute the transactions in sequence requiring at most time nD. Hence, the maximum bucket level will not exceed $\lceil \log(nD) \rceil \leq \log(nD) + 1$. □

Lemma 4 *Any transaction $T \in \mathcal{T}_t^g$ (generated at time t) which is inserted into bucket B_i will be executed by time $t + (i+1)2^{i+2}$.*

Proof We prove this by induction on k, the level of bucket B_k. For the basis case the level is $k = 0$. The bucket B_0 gets activated at every time step. Thus, the bucket B_0 gets activated at the same time t (i.e. instantly) that transaction T is generated. Since $T \in B_0$, it was determined that $F_{\mathcal{A}}(\mathcal{T}_t^s \cup B_0 \cup \{T\}) \leq 2^i = 1$, where $i = 0$. Hence, the transaction T will execute by time $t + 1 = t + 2^i \leq t + (i+1)2^{i+2}$ $(i = 0)$.

Suppose that the claim holds for all $k \leq i$. We consider now the case $k = i + 1$. According to the algorithm, the bucket B_{i+1} gets activated at time t', where $t \leq t' \leq t + 2^{i+1}$. Let $B_{i+1,t}$ denote the contents of B_{i+1} at time t (after new transactions were inserted into B_{i+1} at time t). Consider the latest transaction T'' that was inserted into B_{i+1} at some time t'', $t \leq t'' \leq t'$. From the bucket definition, it must be that $F_{\mathcal{A}}(\mathcal{T}_{t''}^s \cup B_{i+1,t''}) \leq 2^{i+1}$.

Suppose for now that no additional transactions had their execution schedule determined between t'' and t', that is, suppose that $\mathcal{T}_{t'}^s \subseteq \mathcal{T}_{t''}^s$. Since $B_{i+1,t'} \subseteq B_{i+1,t''}$, we get $\mathcal{T}_{t'}^s \cup B_{i+1,t'} \subseteq \mathcal{T}_{t''}^s \cup B_{i+1,t''}$, which implies that $F_{\mathcal{A}}(\mathcal{T}_{t'}^s \cup B_{i+1,t'}) \leq F_{\mathcal{A}}(\mathcal{T}_{t''}^s \cup B_{i+1,t''}) \leq 2^{i+1}$. Thus, in this case all the transactions in B_{i+1} can be executed no later than

$$t' + 2^{i+1}. \tag{2}$$

However, between t'' and t', some other buckets may get activated causing a set of new transactions, say transactions A, having their schedules determined. According to the algorithm, these additional transactions must have been from buckets at level i or lower. From induction hypothesis, those buckets are activated the latest at time t' (before B_{i+1} activates), such that the latest execution time of any transaction from those buckets is $t' + (i+1)2^{i+2}$.

To create the schedule for the transactions in B_{i+1}, the schedule for the transactions in A may need to be extended by time

$$2(t' - t'' + (i+1)2^{i+2}), \tag{3}$$

This is enough time to allow the shared objects used by the transactions in A to move to the positions where the transactions execute (according to their scheduled execution times) and then return back to the original positions they were at time t''. After that the objects (if needed) can be used by the transactions in B_{i+1}.

Therefore, combining Eqs. 2 and 3, the transactions in B_{i+1} will execute no later than

$$\begin{aligned} &t' + 2^{i+1} + 2(t' - t'' + (i+1)2^{i+2}) \\ &\leq t + 2^{i+1} + 2^{i+1} + 2(2^{i+1} + (i+1)2^{i+2}) \\ &= t + (i+2)2^{i+3}, \end{aligned}$$

as needed. □

Before we proceed to prove the main result of this section, we need to perform one more modification on the execution schedule of algorithm $\mathcal{A}$ that will help with the analysis. For any execution schedule S with duration τ, a suffix S' is the execution schedule with duration $\tau' \leq \tau$ which consists of the last τ' time steps of S. In the modification, we will ensure that any batch schedule of $\mathcal{A}$ has the following *suffix*

property: for any set of transactions X the corresponding execution schedule S by $\mathcal{A}$ is such that every suffix S' with respective transactions X' executes in time $F_{\mathcal{A}}(X')$.

If a batch execution schedule S that was generated by $\mathcal{A}$ does not satisfy the suffix property, then it can be "repaired" by repeatedly applying algorithm $\mathcal{A}$ to any suffix of S that possibly violates the suffix property. For example, suppose that $S = S_1S_2$ for transactions $X = X_1 \cup X_2$, where S_1 is a prefix and S_2 is a suffix of S with respective transactions X_1 and X_2. Note that while S has execution time $F_{\mathcal{A}}(X)$, the execution time of S_2 may exceed $F_{\mathcal{A}}(X_2)$. When applying the algorithm $\mathcal{A}$ to X_2 we get a schedule S_2' with execution time $F_{\mathcal{A}}(X_2)$.

The suffix repair can first apply to the longest (with largest execution duration) suffix S' of S that violates the suffix property. This is simply accomplished by using algorithm A to schedule the transactions in the problematic suffix S', and by doing so the time to execute the transactions X' of S' becomes immediately $F_{\mathcal{A}}(X')$. After that, if the suffix property is still violated, the repair may then apply to the next longest suffix S (but still shorter than S') that violates the property. We repeat the process until there is no suffix that violates the property anymore.

The purpose of this modification to $\mathcal{A}$ is to compensate for the cases where suffixes of schedules of previously executed transactions may possibly affect the performance of the schedules for the new transactions. The modification allows us to bound the performance of the online algorithm with respect to the known performance of offline algorithm $\mathcal{A}$. Specifically, we are able to show that if a batch scheduling algorithm $\mathcal{A}$ has approximation ratio $b_{\mathcal{A}}$ then the online schedule is $O(b_{\mathcal{A}} \log^3(nD))$ competitive; the $O(\log^3(nD))$ factor is the price of separating the transactions into buckets, which however makes the online schedule feasible.

Theorem 4 (Bucket schedule competitiveness) *The online schedule has competitive ratio $O(b_{\mathcal{A}} \log^3(nD))$, where $b_{\mathcal{A}}$ is the approximation ratio of offline algorithm $\mathcal{A}$.*

Proof Consider some arbitrary time t where the live transactions are $\mathcal{T}_t = \mathcal{T}_t^s \cup \mathcal{T}_t^g$. If the transactions in $\mathcal{T}_t^g$ were scheduled alone by algorithm $\mathcal{A}$ then their execution schedule time would be within a factor $b_{\mathcal{A}}$ from optimal. However, the newly generated transactions $\mathcal{T}_t^g$ are going to be scheduled based upon the restrictions imposed by the already scheduled transactions $\mathcal{T}_t^s$. In the worst case scenario, the newly generated transactions $\mathcal{T}_t^g$ will execute after the transactions in $\mathcal{T}_t^s$. Therefore, we need to estimate how long it will take to execute the last transactions in $\mathcal{T}_t^s$ in order to determine when the transactions in $\mathcal{T}_t^g$ will execute, and hence determine the duration of the whole schedule.

The transactions in $\mathcal{T}_t^s$ have been generated from various levels at previous buckets. Consider a specific level i that added transactions into $\mathcal{T}_t^s$. From Lemma 4, any such transaction from level i was generated no earlier than $t-(i+1)2^{i+2}$. Since the level i bucket activates every 2^i time steps, during the period $[t-(i+1)2^{i+2}, t]$, the number of level i bucket activations are at most $(i+1)2^{i+2}/2^i = 4(i+1)$.

From the definition of B_i, at the moment when the last transaction was inserted in B_i, algorithm $\mathcal{A}$ could execute the transactions at level i within time 2^i which is a $O(b_{\mathcal{A}})$ approximation factor of the optimal. However, the bucket may activate some time later which may stretch this execution time. According to Lemma 4, the execution schedule of B_i has duration at most $(i+1)2^{i+2}$ which is within a factor $O(ib_{\mathcal{A}})$ from the optimal time schedule for the transactions in B_i. Since from Lemma 3 the maximum level is bounded as $i \leq \log(nD)+1$, we have that the execution time of a bucket at any level is within approximation factor

$$\zeta = O(b_{\mathcal{A}} \log(nD))$$

from optimal.

Next, we calculate the overall approximation factor of the schedule which combines the individual approximation factors of all involved buckets. Let ξ be the total number of buckets that contribute transactions to $\mathcal{T}_t^s$. From Lemma 3 and since each level contributes transactions from at most $4(i+1)$ activated buckets in $\mathcal{T}_t^s$ we get:

$$\xi \leq \sum_{i=0}^{\log(nD)+1} 4(i+1) = O(\log^2(nD)).$$

These ξ buckets can be ordered according to when they get activated, from earliest to latest activation time (if the activation times coincide then the lower levels are ordered first).

Now we examine the execution time of the transactions in the ξ buckets with respect to start time t. The first bucket, say B', out of ξ executes the respective transactions in $\mathcal{T}_t^s$ in a schedule which is within factor ζ from optimal. This is due to the way that the schedule is created. Namely, the transactions in B' are placed in a suffix of the original execution schedule when the bucket B' was activated (at time t or later). Since algorithm $\mathcal{A}$ was made to have the suffix property, a suffix that would have B' alone (at t or later) would give an execution time approximation factor $b_{\mathcal{A}}$. As we analyzed in Lemma 4, the schedule of B' may also depend on other buckets that may activate before it (between t and the activation time), which gives a ζ total approximation factor on execution time.

Similarly, for the second of the ξ buckets, say B'', the respective transactions in $\mathcal{T}_t^s$ must execute within time proportional to a factor of $b_{\mathcal{A}}$ from optimal (starting at t or later) if they were executing alone. This implies a ζ approximation factor for B'' if we consider the schedule of B'' together with the other buckets before it. Thus, the combined time of the first two of the ξ buckets execute in time which has

an approximation factor 2ζ from optimal. Generalizing, the approximation factors of the execution times of the ξ buckets in $\mathcal{T}_t^s$ accumulate giving an execution time which is within a factor $\xi\zeta$ from optimal.

Now consider the newly generated transactions $\mathcal{T}_t^g$ at time t. Note that the transactions in $\mathcal{T}_t^g$ may be distributed among different levels, without any specific level having multiple activation times. With an argument similar as above, each bucket in $\mathcal{T}_t^g$ can be executed in time with approximation ratio ζ from optimal, if we only consider the transactions in $\mathcal{T}_t^g$. If we also consider the transactions in $\mathcal{T}_t^s$, each of the buckets of the transactions in T_t^g executes within time which is a factor $\xi\zeta + \zeta$ from optimal, where $\xi\zeta$ is contributed from the previously scheduled buckets in $\mathcal{T}_t^s$, and ζ is the approximation from the newly generated transactions within a bucket in $\mathcal{T}_t^g$.

Thus, combining the analysis for $\mathcal{T}_t^s$ and $\mathcal{T}_t^g$, the total execution time of the transactions in $\mathcal{T}_t = \mathcal{T}_t^s \cup \mathcal{T}_t^g$ is within a factor $O(\xi\zeta) = O(b_{\mathcal{A}} \log^3(nD))$ from optimal. □

4.3 Applications to specialized graphs

We will apply Algorithm 2 to several cases of network topologies to convert batch scheduling algorithms to online scheduling algorithms. Busch et al. [5] give offline scheduling algorithms for a variety of specialized graphs including the line, cluster, and star graphs. For these batch scheduling problems there are w objects where each node generates at most one transaction, and each transaction requests an arbitrary set of k objects out of w. When we convert these offline algorithms to online we obtain the following results.

- *Line* A line graph is a set of n ordered nodes so that each node has an edge of weight 1 connecting it to the next node in order. It is shown [5] that there is an offline algorithm $\mathcal{A}$ which gives a schedule within approximation factor $b_{\mathcal{A}} = O(1)$ from optimal (asymptotically optimal). From Theorem 4, since $D = O(n)$, we obtain an online scheduling algorithm which is $O(b_{\mathcal{A}} \log^3(nD)) = O(\log^3 n)$ competitive.
- *Cluster* A cluster graph consists of α cliques (clusters) with β nodes each; where all edges in the cliques have the same weight 1. Each clique has a designated bridge node. The bridge nodes from different cliques connect to each other with edges of weight $\gamma \geq \beta$ (there is an edge for each pair of bridge nodes). It is shown [5] that there is an offline algorithm $\mathcal{A}$ which gives a schedule with approximation factor $b_{\mathcal{A}} = O(\min(k\beta, \log_c^k m))$ from optimal, for some constant c, where $m = \max(n, w)$. From Theorem 4, since $D = O(n + \gamma)$, we obtain an online scheduling algorithm which is $O(b_{\mathcal{A}} \log^3(nD)) = O(\min(k\beta, \log_c^k m) \cdot \log^3(n\gamma))$ competitive.
- *Star* In the star graph there is a central node that connects to α rays. A ray is a line graph with β nodes, where one end of the ray connects with an edge to the central node. All edges have weight 1. It is shown [5] that there is an offline algorithm $\mathcal{A}$ which gives a schedule with approximation factor $b_{\mathcal{A}} = O(\log \beta \cdot \min(k\beta, \log_c^k m))$ from optimal, for a constant c. From Theorem 4, since $D = O(n)$, we obtain an online scheduling algorithm which is $O(b_{\mathcal{A}} \log^3(nD)) = O(\log \beta \cdot \min(k\beta, \log_c^k m) \cdot \log^3 n)$ competitive.

Note that the cluster and star batch offline scheduling algorithms used above are in fact randomized. In the case that the bad event occurs for a bucket of not getting a schedule with the specified time bound given by the offline schedule (with small probability), then we repeat the offline algorithm for that bucket until we successfully obtain a batch schedule within the required time bound. Thus, the online schedules remain feasible.

The sequential run time complexity of Algorithm 2 in calculating an execution schedule at time step t depends on the sequential complexity of algorithm $\mathcal{A}$ for each bucket level with a small multiplicative logarithmic overhead since there are only $O(\log(nD))$ levels of buckets. The offline batch scheduling algorithms we used above have polynomial time complexity in computing the schedules [5]. Thus, Algorithm 2, has polynomial time complexity in calculating the respective online schedules for the network cases we discussed above. However, according to our execution model these sequential calculations are subsumed within a single time step t of the concurrent execution, since the communication delay is considered the main contributing factor in the concurrent execution time.

5 Distributed bucket approach

The online algorithms we presented earlier use a central authority with knowledge about all current transactions and objects. Here, we discuss a distributed approach which allows the online schedule to be computed in a decentralized manner without requiring a central authority.

The distributed approach is based on the online bucket scheduling algorithm described in Sect. 4 but adapted appropriately to work in the distributed setting. In the adapted version of the bucket algorithm, the buckets of different levels are split among various nodes of the graph G in what we call *partial buckets* and denote by $\overline{B}$. To facilitate the distributed scheduling algorithm, we use a hierarchy of clusters where designated leader nodes at the clusters hold the partial buckets. We continue with describing the details of this approach.

5.1 Cluster decomposition

Divide the graph G into a hierarchy of clusters with $H_1 = \lceil \log D \rceil + 1$ layers, where D is the diameter of G (logarithms are base 2). A cluster is a subset of the nodes, and its diameter is the maximum distance between any two nodes. We use the *weak diameter* of the cluster where distances between nodes in the cluster are measured with respect to G and not within the subgraph induced by the cluster. The diameter of each cluster at layer ℓ, where $0 \leq \ell < H_1$, is no more than $f(\ell)$ for some function f that we specify below. Moreover, each node participates in no more than $g(\ell)$ clusters at layer ℓ, for some function g. An additional property is that for each node u in G there is a cluster at layer ℓ such that the $(2^\ell - 1)$-neighborhood of u is contained in that cluster (the k-neighborhood is the set of nodes which are distance at most k from u; the 0-neighborhood is u itself).

There are cluster constructions [15,36] which give a hierarchy with H_1 layers where $f(\ell) = O(\ell \log n)$, and $g(\ell) = O(\log n)$, known as a *hierarchical sparse cover* of G. These constructions have the *strong diameter* property, where distances are measured within the induced subgraph in the clusters, but these still serve our purpose where we require weak diameter properties.

There is actually a hierarchical sparse cover construction [36] such that in each layer ℓ, a node, say u, belongs to at most $H_2 = O(\log n)$ different clusters. These H_2 clusters can be ordered as sub-layers of clusters 0 to $H_2 - 1$ (in the context of u), where each sub-layer is a cluster of level ℓ of G. Thus, a node u participates in all the H_2 sub-layers of a layer but possibly in a different cluster at each sub-layer. At least one of those H_2 clusters at layer ℓ contains the $(2^\ell - 1)$-neighborhood of u. We use such a sparse cover in our algorithm. In each cluster at layer ℓ a *leader* node is designated such that the leader's $(2^\ell - 1)$-neighborhood is in that cluster.

Since we have the notion of layers and sub-layers, the notion of *height* can be defined as a tuple $h = (h_1, h_2)$ where h_1 denotes layer and h_2 denotes sub-layer. Heights are ordered lexicographically. When the context is clear, we use layer and sub-layer notion interchageably.

We need a notion of *home cluster* for transaction T defined as follows. Let x be the maximum distance from the position of T to one of the objects in the set $O(T)$. Let z be the distance to the furthest conflicting transaction to T from the position of T. The home cluster for T is the lowest level cluster in the hierarchy that includes T and its $\max(x, z)$-neighborhood.

To facilitate the actions of the distributed transaction scheduling, each shared memory object o_i carries with it some information to assist the scheduling task. The object o_i carries the information of all the existing transaction locations (node addresses) that will use it in the execution schedule, and also which of these transactions have already been scheduled or not. In addition, the object o_i has information about the partial bucket locations (node addresses) that have transactions that use it. This information which is carried with the object is updated by the partial buckets when new scheduling decisions are made about the object.

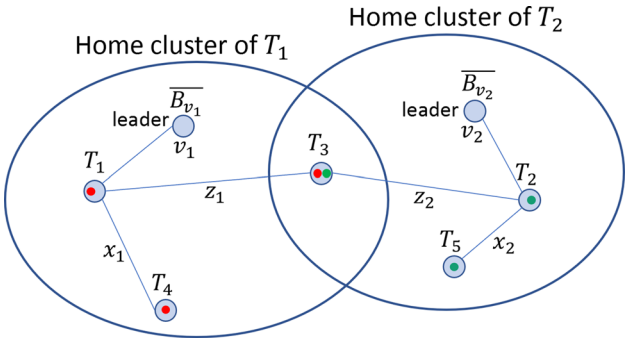


Fig. 3 Home clusters of transactions T_1 and T_2

5.2 Distributed bucket algorithm

Algorithm 3 has the transaction actions of the distributed bucket approach. The bucket B_i is split into possibly multiple *partial i-buckets* $\overline{B}$ which reside at different nodes. The union of the transactions in the partial i-buckets $\overline{B}$ make the whole bucket B_i. The partial i-buckets $\overline{B}$ may appear into multiple layers (or sub-layers) of the hierarchy. In fact, there could be multiple i-buckets $\overline{B}$ (for the same i) in each sub-layer. The partial i-buckets $\overline{B}$ are hosted at leader nodes of the clusters. All of the partial i-buckets get activated simultaneously every 2^i time steps, at a time that corresponds to the activation time of B_i. The scheduling gives priority to lower level buckets first (similar to Algorithm 2), and within the same bucket level priority is given to partial buckets at lower height home clusters.

Consider a transaction T generated at time t at a node v. Transaction T (i.e. the process that executes T in v) looks to find all conflicting transactions with T that have already been scheduled. Transaction T also looks for buckets that have transactions conflicting with it. It collects this information from the objects, as described in Algorithm 3, which helps to determine into which bucket T will be inserted.

Figure 3 depicts an example with the home clusters of two transactions T_1 and T_2. Transaction T_1 uses object o_1 (shown with the red colored circle) and conflicts with transactions T_3 and T_4. Object o_1 resides in the node of T_4 at distance x_1 from T_1. The distance to T_3 is z_1. The home cluster of T_1 includes the $y = \max(x_1, z_1)$-neighborhood of T_1, and T_1 reports to the leader node v_1 of the cluster. The leader node v_1 holds a partial i-bucket $\overline{B_{v_1}}$ which includes T_1, for an appropriate parameter i. Similarly, transaction T_2 uses object o_2 (shown with the green colored circle) and conflicts with transactions

Algorithm 3: Distributed Bucket Schedule

1 **foreach** *new transaction* T **do**
2 Transaction T discovers the current positions of its objects in G; suppose the furthest is x away from T;
3 Each of the objects of T informs the transaction T about other conflicting transactions (both scheduled and unscheduled transactions);
4 Let y be the maximum of x and the distance to the furthest conflicting transaction to T from the position of T;
5 T picks a *home cluster* at the lowest height which includes T and its y-neighborhood;
6 T reports to the leader v of the home cluster; then, v places T into a partial i-bucket in v, where i is determined by the transactions reported to v;
7 When the partial bucket of T gets activated, all the objects of T are informed about the execution schedule;

T_3 and T_5. Transaction T_2 reports to the leader v_2 of its home cluster that holds partial i'-bucket $\overline{B_{v_2}}$, for an appropriate parameter i'. Note that the nodes of transactions may be in multiple clusters, as for example the node of T_3 is in both clusters, but a transaction reports to a single home cluster.

The objects could be moving from node to node while the transactions execute. We can track an object in transit by reaching the last node that the object departed from. While transaction T in v tries to discover the current locations of the objects that it uses, the objects may drift further away from v making it even more difficult to be discovered. For this reason, we require that an object moves in the graph at a slower pace than the discovery request messages travel in the graph. We can accomplish this by slowing down the object travel time so that it takes two time steps to traverse a unit weight link, instead of one, that is, halving the speed that the object moves. In this way, if at time t an object is at distance d from v, then it will be discovered by time $2d$ at most. This is because within $2d$ time steps the object moves at most d additional distance units away from v for a total of at most $2d$ units of distance away from v. At the same time some discovery message emanated from v catches up with the last node that the object departed from, since the discovery messages are transferred with regular speed.

5.3 Analysis of distributed algorithm

We continue with the analysis of Algorithm 3. We first note that the conflicting transactions with T could be split at different buckets at different layers of the cluster hierarchy, which is due to the objects of those transactions possibly being different than those used by T.

Lemma 5 *For any two live conflicting transactions T and T' it cannot be that neither detect each other before they report to their home clusters.*

Proof T and T' both share at least one object. One of the two transactions accesses the object first. Recall that the object maintains the information about the transactions that access it. The second transaction to access the object becomes aware of the first transaction. □

Lemma 6 *If T reports at a home cluster C then there is no other conflicting transaction to T that reports in a different cluster at the same sub-layer as that of C.*

Proof Suppose that there is a conflicting transaction T' that reports to a cluster C' which is at the same sub-layer as that of C. From Lemma 5, T is aware of T' or vice-versa.

If T is aware of T', then the cluster C must contain T', since T picks its home cluster to contain all the transactions that T knows it conflicts with. Hence, T' must have also reported to C, namely, $C = C'$, due to the sub-layer being a partition of G.

Symmetrically, if T' is aware of T, then the cluster that T' has reported to must be the same with the cluster of T. □

From Lemma 6, if there are two transactions T and T' that conflict with each other and both are in i-buckets in the same sub-layer, then T and T' must be in the same i-bucket hosted at the leader of the same home cluster within the sub-layer. Hence, we have the following corollary:

Corollary 1 *In a sub-layer, any two partial i-buckets do not have transactions that conflict with each other.*

We continue with bounding the maximum height that a bucket can appear to in the cluster hierarchy.

Lemma 7 *The maximum height in the cluster hierarchy that a partial i-bucket can appear to is $(i + 1, H_2 - 1)$.*

Proof Consider a partial i-bucket $\overline{B}$ which is hosted at some leader node v of a cluster C at layer ℓ. By the definition of a bucket, the upper bound on the execution time for the transactions in $\overline{B}$ is 2^i if using the offline algorithm $\mathcal{A}$ for scheduling. Thus, the maximum distance x to the objects that any $T \in \overline{B}$ uses is bounded by $x \leq 2^i$. Hence, for $\ell = i+1$, cluster C can contain all up to distance $2^{i+1} - 1 \geq 2^i$ neighbors for each of the transactions in $\overline{B}$. Since the maximum sub-layer is $H_2 - 1$, the height of C is at most $(i + 1, H_2 - 1)$ (heights are ordered lexicographically). □

Define $M = \max(H_1, H_2, L) + 3$, where L is the maximum number of distinct bucket levels, which according to Lemma 3 is $\log(nD) + 1$. Note that $M = O(\log(nD))$. Next are adaptations of Lemma 4 and Theorem 4 in the distributed setting. The next result is an adaptation of Lemma 4.

Lemma 8 *In the distributed setting, any transaction $T \in \mathcal{T}_t^g$ (generated at time t) which is inserted into a partial i-bucket $\overline{B}$ that resides at height (j, k) will be executed by time $t + (iM^2 + jM + k + 3)2^{i+3}$.*

Proof The combination of the level i and height (j,k) of $\overline{B}$ corresponds to a tuple (i,j,k). We prove the claim by induction on the lexicographic order on the tuple (i,j,k), where $i,j,k \geq 0$.

For the basis case, the tuple is $(0,0,0)$. The partial 0-bucket $\overline{B}$ gets activated at every time step. Thus, the bucket $\overline{B}$ gets activated at time t (i.e. instantly). Since $\overline{B}$ is a partial 0-bucket it was determined that $F_{\mathcal{A}}(\mathcal{T}_t^s \cup \overline{B} \cup \{T\}) \leq 2^i = 2^0 = 1$, the transaction T is scheduled to execute by time $t+1 = t+2^i \leq t+3\cdot 2^{i+3}$ $(i=0)$, as needed.

Suppose that the claim holds for all tuples up to $\tau \geq (0,0,0)$. In the induction step, we will prove that the claim holds for tuple τ' which is the immediate next in lexicographic order after τ.

According to the algorithm, $\overline{B}$ gets activated at time t', where $t \leq t' \leq t+2^i$. Let $\overline{B}_t$ denote the contents of $\overline{B}$ at time t, including the new transactions that were inserted into $\overline{B}$ at time t. Consider the latest transaction T'' that was inserted into $\overline{B}$ at some time t'', $t \leq t'' \leq t'$. From the definition of the level of a bucket, it must be that $F_{\mathcal{A}}(\mathcal{T}_{t''}^s \cup \overline{B}_{t''}) \leq 2^i$.

Suppose for now that no additional transactions had their execution schedule determined between t'' and t'. Namely, $\mathcal{T}_{\hat{t}}^s \subseteq \mathcal{T}_{t'}^s$, for $\hat{t}, t'' \leq \hat{t} \leq t'$. Thus, $\mathcal{T}_{t'}^s \cup \overline{B}_{t'} \subseteq \mathcal{T}_{t''}^s \cup \overline{B}_{t''}$. Hence, $F_{\mathcal{A}}(\mathcal{T}_{t'}^s \cup \overline{B}_{t'}) \leq F_{\mathcal{A}}(\mathcal{T}_{t''}^s \cup \overline{B}_{t''}) \leq 2^i$. Since in the distributed setting the objects move at half the speed, this scheduling bound becomes at most 2^{i+1}. Moreover, the objects have to be informed about the schedule and this may take additional time up to 2^i in order for the schedule information to reach the objects, bringing the total time to at most $2^{i+1}+2^i \leq 2^{i+2}$. Thus, in this case all the transactions in $\overline{B}$ can be scheduled to execute no later than:

$$t' + 2^{i+2}. \tag{4}$$

Equation 4 does not include the impact on the schedule from lower lever buckets that are activated between t'' and t'. In order to accommodate these dependencies, we calculate the worst execution schedule that may be caused from these buckets and adjust the time bound of Eq. 4 accordingly. This adjustment provisions that those buckets' schedules may not be communicated on time before t' to $\overline{B}$. We breakdown the analysis to different cases for τ'.

Case (i) $\tau' = (i,0,0)$. In this case, $\overline{B}$ is a partial i-bucket at height $(0,0)$. Note that $\overline{B}$ is at the lowest height for any partial bucket of level i. From Corollary 1, $\overline{B}$ does not conflict with any other partial bucket of height $(0,0)$. Thus, the schedule of $\overline{B}$ may only be affected from lower level buckets $i' < i$.

Between times t'' and t' other partial buckets at level $i' < i$ may get activated resulting to a set of transactions A having their schedules determined. These transactions A may have dependencies with the transactions in $\overline{B}$ and are higher priority than $\overline{B}$ in the scheduling. From Lemma 7, the maximum height of any such bucket is at most $(i'+1, H_2-1)$. In the worst case, those buckets are activated the latest at time t' (before $\overline{B}$). From induction hypothesis, since $i' \leq i-1$ and $M \geq H_2+3$, the latest execution time of any transaction from those buckets is $t'+(i'M^2+(i'+1)M+(H_2-1)+3)2^{i'+3} \leq t'+((i-1)M^2+iM+M-1)2^{i+2}$. To accommodate the transactions in A the schedule of $\overline{B}$ needs to be shifted by time:

$$2(t'-t''+((i-1)M^2+iM+M-1)2^{i+2}). \tag{5}$$

The reason for this adjustment is that it gives enough time to allow the shared objects used by the transactions in A to move to the positions where they execute according to their scheduled execution times and then return back to the original positions they were at time t''.

Therefore, combining Eqs. 4 and 5, and since $t' \leq t+2^i$, $t'-t'' \leq 2^i$ and $iM+M \leq (L+1)M \leq (M-2)M \leq M^2$ the execution time of the transactions in $\overline{B}$ is bounded by

$$\begin{aligned}
&t' + 2^{i+2} + 2(t'-t'' \\
&\quad +((i-1)M^2+iM+M-1)2^{i+2}) \\
&\leq t+2^i+2^{i+2}+2(2^i+((i-1)M^2+M^2-1)2^{i+2}) \\
&\leq t+2^i+2^{i+2}+2(2^i+(iM^2-1)2^{i+2}) \\
&\leq t+2\cdot 2^{i+3}+(iM^2-1)2^{i+3} \\
&\leq t+iM^2 2^{i+3}+2^{i+3} \\
&\leq t+(iM^2+3)2^{i+3},
\end{aligned}$$

as needed (recall that $j=0$ and $k=0$).

Case (ii) $\tau' = (i,j,k)$, $j+k>0$. In this case, the $\overline{B}$ bucket is at height (j,k), where $j>0$ or $k>0$. The schedule of the transactions in $\overline{B}$ depends on the schedule of partial i-buckets at lower heights of the cluster hierarchy, i.e. $(j',k') < (j,k)$ (Corollary 1 excludes dependency within the same height). If no such partial level i bucket exists at lower height than $\overline{B}$, the analysis is similar to case (i) described above. Hence, assume that such a lower height partial bucket exists. Since the largest height below (j,k) is $(j,k-1)$ if $k>0$, and $(j-1,k)$ if $k=0$, the latest execution time of any transaction from those buckets is when the height is $(j,k-1)$ for which the induction hypothesis gives $t'+(iM^2+jM+(k-1)+3)2^{i+3}$.

From the partial buckets definition, all objects needed by every partial bucket at B_i are at distance at most 2^i from the transactions that need them after the first partial bucket of B_i with lowest height activates. Any dependent objects can be passed from one transaction to another in at most 2^i steps. But, considering that the objects traverse at half speed, the objects are transferred in at most 2^{i+1} steps. Moreover, with Eq. 4, we obtain the execution time of the transactions in $\overline{B}$ is bounded by

$$t' + 2^{i+2} + 2^{i+1} + (iM^2+jM+(k-1)+3)2^{i+3}$$

$$\leq t + 2^{i+3} + (iM^2 + jM + (k-1) + 3)2^{i+3}$$
$$\leq t + (iM^2 + jM + k + 3)2^{i+3},$$

as needed. □

Next, is a distributed adaptation of Theorem 4.

Theorem 5 (Distributed bucket competitiveness) *In the distributed setting, the online schedule has competitive ratio* $O(b_{\mathcal{A}} \log^9(nD))$*, where* $b_{\mathcal{A}}$ *is the approximation ratio of offline algorithm* $\mathcal{A}$*.*

Proof Consider some arbitrary time t where the live transactions are $\mathcal{T}_t = \mathcal{T}_t^s \cup \mathcal{T}_t^g$, where $\mathcal{T}_t^s$ are the scheduled transactions and $\mathcal{T}_t^g$ are the new transactions. If the transactions in $\mathcal{T}_t^g$ were scheduled alone by algorithm $\mathcal{A}$ then their execution schedule time would be within $b_{\mathcal{A}}$ from optimal. However, the newly generated transactions $\mathcal{T}_t^g$ are going to be scheduled based on the restrictions imposed by the already scheduled transactions $\mathcal{T}_t^s$. In the worst case scenario, the newly generated transactions $\mathcal{T}_t^g$ will execute after the transactions in $\mathcal{T}_t^s$. Therefore, we need to estimate how long it will take to execute the last transactions in $\mathcal{T}_t^s$ in order to determine when the transactions in $\mathcal{T}_t^g$ will start to execute, and hence determine the duration of the whole schedule.

The schedules of the transactions in $\mathcal{T}_t^s$ have been generated from various levels at previously activated buckets. Consider a specific level i-partial bucket $\overline{B}$ at height (j, k) that contributed transactions in $\mathcal{T}_t^s$. From Lemma 8, any such transaction was generated no earlier than $\hat{t} = t - (iM^2 + jM + k + 3)2^{i+3}$. From Lemma 7, the maximum height of any partial i-bucket is $(i+1, H_2 - 1)$, which implies that $(j, k) \leq (i+1, H_2 - 1)$. Thus, $j \leq i+1$, and since $i+1 \leq H_1$ and $H_1 + 3 \leq M$, we also get $i + 1 \leq M - 3$. Since $H_2 - 1$ is the maximum possible sub-layer, and since $M \geq H_2 + 3$, we also get $k \leq H_2 - 1 \leq M - 4 \leq M - 3$. Hence,

$$\begin{aligned} t &= t - (iM^2 + jM + k + 3)2^{i+3} \\ &\geq t - (iM^2 + (i+1)M + (M-3) + 3)2^{i+3} \\ &\geq t - (iM^2 + (M-3)M + M)2^{i+3} \\ &= t - (iM^2 + M^2 - 2M)2^{i+3} \\ &\geq t - (iM^2 + M^2)2^{i+3} \\ &= t - (i+1)M^2 2^{i+3}. \end{aligned}$$

Therefore, between $\hat{t}$ and t the number of (non-partial) B_i buckets that have been activated is at most $(i+1)M^2 2^{i+3}/2^i = 8(i+1)M^2$, since B_i activates every 2^i time steps. We can treat all the partial i-buckets which are at the same height (j, k) as a single bucket that contributes to $\mathcal{T}_t^s$, since from Corollary 1 these buckets do not interfere with each other. Since from Lemma 7 the maximum height of any partial i-bucket is $(i+1, H_2 - 1)$, the number of distinct involved heights is at most $(i+2)H_2$ (which includes layers 0 to $i+1$ and H_2 sub-layers from each layer). Therefore, the total number of partial i-buckets in B_i that can contribute to $\mathcal{T}_t^s$ is the product of activated buckets (at most $8(i+1)M^2$) and involved heights (at most $(i+2)H_2$), giving at most $8(i+1)(i+2)H_2M^2 \leq 8(i+2)^2M^3$ partial i-buckets.

Let ξ be the total number of distinct buckets that can contribute to $\mathcal{T}_t^s$. From Lemma 3 we have that $L \leq \log(nD) + 1$, and since $H_1 = O(\log n)$ and $H_2 = O(\log n)$, we get $M = O(\log(nD))$. Thus, we get:

$$\xi \leq \sum_{i=0}^{\log(nD)+1} 8(i+2)^2M^3 = O(\log^6(nD)).$$

The ξ buckets can be ordered according to when they get activated. Let's examine the execution time of the transactions in the ξ buckets with respect to starting time t. According to Lemma 8, the execution time of each partial i-bucket (at a specific height) is within a factor $O(iM^2 b_{\mathcal{A}})$ from the optimal time, since 2^i is a $b_{\mathcal{A}}$ approximation of the optimal schedule, by the definition of the bucket level.

Considering the maximum bucket level $i = O(\log nD)$, for the first of the ξ buckets to be activated the respective transactions in $\mathcal{T}_t^s$ must execute within factor of

$$\zeta = O((\log(nD))M^2 b_{\mathcal{A}}) = O(M^3 b_{\mathcal{A}}) = O(b_{\mathcal{A}} \log^3(nD))$$

from optimal. The reason is that their predetermined schedule corresponds to a suffix (starting from t) of the original execution schedule when the bucket was activated. Furthermore, algorithm $\mathcal{A}$ has the suffix property such that any suffix of its schedule has execution time within $b_{\mathcal{A}}$ from optimal, and the schedule of $\mathcal{A}$ is only shifted in time to give ζ total approximation due to other dependencies, as we analyzed in Lemma 8.

Similarly, for the second of the ξ buckets to be activated, the respective transactions in $\mathcal{T}_t^s$ must execute within time proportional to a factor of ζ from optimal starting from t if they were running alone. Thus, the first two of the ξ buckets execute in time 2ζ factor from optimal. Generalizing, the ζ approximation factors accumulate and the total execution time of the ξ buckets is within $\xi\zeta$ from optimal.

With a similar argument, each of the corresponding buckets of T_t^g executes within time which is a factor $\xi\zeta + \zeta$ from optimal, where $\xi\zeta$ is the approximation factor contributed from the previous buckets, while ζ is the approximation from the newly generated transactions. Thus, the total execution time of the transactions in $\mathcal{T}_t$ is within a factor $(\xi + 1)\zeta = O(b_{\mathcal{A}} \log^9(nD))$ from optimal. □

Note that Algorithm 3 also has polynomial time complexity within each of the nodes that hold partial buckets, as it adapts Algorithm 2 which has polynomial time complexity.

6 Concluding remarks

We have presented efficient execution time schedules in the online dynamic scheduling setting on the data-flow model of distributed transactional memory under the synchronous communication model. Only the offline batch scheduling setting was studied formally previously in the literature. Our results are the first known attempts to obtain provably efficient online execution schedules for distributed transactional memory.

There are some open questions. It would be interesting to examine the impact of congestion on the links of the graph, especially if the links have bounded capacity, as it may limit the number of messages that can be propagated concurrently. Furthermore, it would also be interesting to evaluate our algorithm against different application benchmarks in a practical setting.

Acknowledgements M. Popovic is supported by the Ministry of Education, Science and Technology Development of Republic of Serbia Grant 451-03-68/2020-14/200156. G. Sharma is supported by the National Science Foundation Grants CCF-1936450 and CNS-2045597.

References

1. Aguilera, Marcos K., Malkhi, Dahlia., Marzullo, Keith., Panconesi, Alessandro., Pelc, Andrzej., Wattenhofer, Roger.: Announcing the 2012 Edsger W. Dijkstra prize in distributed computing. SIGARCH Computer Architecture News, 40(4):1–2, 2012
2. Attiya, H.: Lower bounds and impossibility results for transactional memory computing. Bull. EATCS 112 (2014)
3. Attiya, H., Epstein, L., Shachnai, H., Tamir, T.: Transactional contention management as a non-clairvoyant scheduling problem. Algorithmica **57**(1), 44–61 (2010)
4. Bocchino, R.L., Adve, V.S., Bradford, L.: Chamberlain. Software transactional memory for large scale clusters. In: PPoPP, pp. 247–258 (2008)
5. Busch, C., Herlihy, M., Popovic, M., Sharma, G.: Fast scheduling in distributed transactional memory. In: SPAA, pp. 173–182 (2017)
6. Busch, C., Herlihy, M., Popovic, M., Sharma, G.: Time-communication impossibility results for distributed transactional memory. Distrib. Comput. **31**(6), 471–487 (2018)
7. Cain, H.W., Michael, M.M., Frey, B., May, C., Williams, D., Le Hung, Q.: Robust architectural support for transactional memory in the POWER architecture. In: ISCA, pp. 225–236 (2013)
8. Chan, M.Y.: Embedding of D-dimensional grids into optimal hypercubes. In: SPAA, pp. 52–57 (1989)
9. Couceiro, M., Romano, P., Carvalho, N., Rodrigues, L.: D2STM: dependable distributed software transactional memory. In: PRDC, pp. 307–313 (2009)
10. Dragojević, A., Guerraoui, R., Singh, A.V., Singh V.: Preventing versus curing: avoiding conflicts in transactional memories. In: PODC, pp. 7–16 (2009)
11. Dubois, M., Annavaram, M., Stenstrm, P.: Parallel Computer Organization and Design. Cambridge University Press, New York (2012)
12. Fung, W.W.L., Singh, I., Brownsword, A., Aamodt, T.M.: Hardware transactional memory for GPU architectures. In: MICRO, pp. 296–307 (2011)
13. Gonzalez-Mesa, M.A., Gutiérrez, E., Zapata, E.L., Plata, O.: Effective transactional memory execution management for improved concurrency. ACM Trans. Archit. Code Optim. **11**(3), 24:1-24:27 (2014)
14. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: PODC, pp. 258–264 (2005)
15. Gupta, A., Hajiaghayi, M.T., Räcke, H.: Oblivious network design. In: SODA, pp. 970–979 (2006)
16. Haring, R., Ohmacht, M., Fox, T., Gschwind, M., Satterfield, D., Sugavanam, K., Coteus, P., Heidelberger, P., Blumrich, M., Wisniewski, R., Gara, A., Chiu, G., Boyle, P., Chist, N., Kim, C.: The IBM blue gene/Q compute chip. IEEE Micro **32**(2), 48–60 (2012)
17. Hendler, D., Naiman, A., Peluso, S., Quaglia, F., Romano, P., Suissa, A.: Exploiting locality in lease-based replicated transactional memory via task migration. In: DISC, pp. 121–133 (2013)
18. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: *ISCA*, pp. 289–300 (1993)
19. Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. Distrib. Comput. **20**(3), 195–208 (2007)
20. Intel. http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell (2012)
21. Irving, S., Chen, S., Peng, L., Busch, C., Herlihy, M., Michael, C.J.: CUDA-DTM: distributed transactional memory for GPU clusters. In: NETYS, pp. 183–199 (2019)
22. Kim, J., Ravindran, B.: Scheduling transactions in replicated distributed software transactional memory. In: CCGrid, pp. 227–234 (2013)
23. Kim, J., Ravindran, B.: On transactional scheduling in distributed transactional memory systems. In: SSS, pp. 347–361 (2010)
24. Kotselidis, C., Ansari, M., Jarvis, K., Lujan, M., Kirkham, C., Watson, I.: DiSTM: a software transactional memory framework for clusters. In: ICPP, pp 51–58 (2008)
25. Leighton, F.T.: Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes. Morgan Kaufmann Publishers Inc., San Francisco (1992)
26. Manassiev, K., Mihailescu, M., Amza, C.: Exploiting distributed version concurrency in a transactional memory cluster. In: PPoPP, pp. 198–208 (2006)
27. Michalewicz, M., Orlowski, L., Deng, Y.: Creating interconnect topologies by algorithmic edge removal: MOD and SMOD graphs. Supercomput. Front. Innov. Int. J. **2**(4), 16–47 (2015)
28. Nakaike, T., Odaira, R., Gaudet, M., Michael, M.M., Tomari, H.: Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In: ISCA, pp. 144–157 (2015)
29. Palmieri, R., Peluso, S., Ravindran, B.: Transaction Execution Models in Partially Replicated Transactional Memory: The Case for Data-Flow and Control-Flow, pp. 341–366. Springer, Cham (2015)
30. Pasricha, S., Dutt, N.: On-Chip Communication Architectures: System on Chip Interconnect. Morgan Kaufmann Publishers Inc., Burlington (2008)
31. Poudel, P., Rai, S., Sharma, G.: Processing distributed transactions in a predefined order. In: ICDCN, pp. 215–224 (2021)
32. Romano, P., Palmieri, R., Quaglia, F., Carvalho, N., Rodrigues, L.: On speculative replication of transactional systems. J. Comput. Syst. Sci. **80**(1), 257–276 (2014)
33. Saad, M.M., Kishi, M.J., Jing, S., Hans, S., Palmieri, R.: Processing transactions in a predefined order. In: PPoPP, pp. 120–132 (2019)

34. Saad, M.M., Ravindran, B..: Snake: control flow distributed software transactional memory. In: SSS, pp. 238–252 (2011)
35. Sharma, G., Busch, C.: Window-based greedy contention management for transactional memory: theory and practice. Distrib. Comput. **25**(3), 225–248 (2012)
36. Sharma, G., Busch, C.: Distributed transactional memory for general networks. Distrib. Comput. **27**(5), 329–362 (2014)
37. Shavit, N., Touitou, D.: Software transactional memory. Distrib. Comput. **10**(2), 99–116 (1997)
38. Zhang, B., Ravindran, B. Relay: a cache-coherence protocol for distributed transactional memory. In: OPODIS, pp. 48–53 (2009)
39. Zhang, B., Ravindran, B., Palmieri, R.: Distributed transactional contention management as the traveling salesman problem. In: SIROCCO, pp. 54–67 (2014)