Ordered Scheduling in Control-flow Distributed Transactional Memory[☆]

Pavan Poudel[a,*], Shishir Rai[b], Swapnil Guragain[b]

[a]*ATGWORK, Norcross, Georgia, USA*
[b]*Kent State University, Kent, Ohio, USA*

Abstract

Consider the *control-flow* model of transaction execution in a distributed system modeled as a communication graph where shared objects positioned at nodes of the graph are immobile but the transactions accessing the objects send requests to the nodes where objects are located to read/write those objects. The control-flow model offers benefits to applications in which the movement of shared objects is costly due to their sizes and security purposes. In this paper, we study the *ordered scheduling* problem of committing *dependent* transactions according to their predefined priorities in this model. The considered problem naturally arises in areas, such as loop parallelization and state-machine-based computing, where producing executions equivalent to a priority order is needed to satisfy certain properties. Specifically, we study ordered scheduling considering two performance metrics fundamental to any distributed system: (i) *execution time* - total time to commit all the transactions and (ii) *communication cost* - the total distance traversed in accessing required shared objects. We design scheduling algorithms that are individually or simultaneously efficient for both the metrics and rigorously evaluate them through several benchmarks on random and grid graphs, validating their efficiency. To our best knowledge, this is the first study of ordered scheduling in the control-flow model of distributed transaction execution.

Keywords: Distributed system; transactional memory; scheduling; control-flow model; predefined order; execution time; communication cost; competitive ratio

[☆]A preliminary version of this paper appears in the Proceedings of ICDCIT'23 [1].

[*]Corresponding author. Tel.: +1 800 551 7943

Email addresses: poudelpavan@gmail.com (Pavan Poudel), srai@kent.edu (Shishir Rai), sguragai@kent.edu (Swapnil Guragain)

1. Introduction

Concurrent processes (threads) need to synchronize to avoid introducing inconsistencies while accessing shared data objects. Traditional mechanisms of locks and barriers have well-known downsides, including deadlock, priority inversion, reliance on programmer conventions, and vulnerability to failure or delay. *Transactional memory* (TM) [2, 3] has emerged as an attractive alternative. Using TM, program code is split into *transactions*, blocks of code that appear to execute atomically. Transactions are executed *speculatively*: synchronization conflicts (or failures) may cause an executing transaction to *abort*: its effects are rolled back and the transaction is restarted. In the absence of conflicts (or failures), a transaction typically *commits*, causing its effects to become visible to all threads. Several commercial processors support TM, e.g., Intel's Haswell [4] and IBM's Blue Gene/Q [5], zEnterprise EC12 [6], and Power8 [7].

TM has been studied extensively for *multiprocessors*, where processors operate on a single shared memory and the latency to access (read/write) shared memory is the same (and negligible) for each processor. However, recently, the computing trend is shifting toward *distributed multiprocessors*, where the memory access latency varies depending on the processor in which the thread executes and the physical segment of memory that stores the requested memory location. Therefore, the recent research focus is on how to support TM in distributed multiprocessors. Some proposals in this direction include TM^{2}C [8], Nemo [9], cluster-TM [10, 11], GPU-TM [12], and Hyflow [13].

TM is beneficial in distributed systems where data is spread across multiple nodes. For example, distributed data centers can use TM to simplify the burden of distributed synchronization and provide more reliable and efficient program execution while accessing data from remote nodes. *Distributed TM* (DTM) designed for such systems need to execute transactions effectively by taking into consideration the system's infrastructure. The network structure can play a crucial role in the DTM performance, since the data transactions access has to be reached across the network in a timely manner.

In this paper, we study ordered scheduling (denoted as OrdS) problem in distributed multiprocessors. We model distributed multiprocessors as an n-node connected, undirected, and weighted graph G, where each node denotes a processor and each edge denotes a communication link between two processors. A set of w shared objects $\mathcal{S} := \{S_1, S_2, \ldots, S_w\}$ reside on the (possibly different) nodes of G. We consider the *control-flow* model [14], where objects are immobile but transactions send access requests to the nodes the required objects are located. Consider a set $\mathcal{T} := \{T(v_1, age_1), T(v_2, age_2), \ldots\}$ of transactions mapped (arbitrarily) to the nodes of G with each $T(v_i, age_i)$ accessing an arbitrary subset of the shared objects

$\mathcal{S}(T(v_i, age_i)) \subseteq \mathcal{S}$, where age is an externally provided parameter that is unique for each transaction providing a priority order. We can assume the existence of a module that can assign such unique priorities (ages) to transactions; a detailed study on this module is beyond the scope of this paper.

We say transaction $T(v_i, age_i)$ is *dependent* on $T(v_j, age_j), age_j < age_i$, if at least an object read/write by $T(v_i, age_i)$ is being written by $T(v_j, age_j)$. The OrdS problem is to commit the dependent transactions in the age order. For example, transaction $T(v_i, age_i)$ that depends on $T(v_j, age_j), age_j < age_i$, commits only after $T(v_j, age_j)$ has been committed. Non-dependent transactions can execute and commit in parallel, irrespective of their priority order. For example, among the two transactions $T(v_i, age_i)$ and $T(v_j, age_j), age_j < age_i$, let $T(v_j, age_j)$ is not *dependent* on any other transaction $T(v', age'), age' < age_j$. Also, let $T(v_i, age_i)$ is not *dependent* on any other transaction $T(v'', age''), age'' < age_i$ including $T(v_j, age_j)$. Then, $T(v_i, age_i)$ and $T(v_j, age_j)$ are called *non-dependent* transactions and can be executed in parallel.

OrdS naturally arises in applications where producing (dependent) executions equivalent to a priority order is needed to satisfy/guarantee certain properties. Example applications include speculative loop parallelization and distributed computation using state machine approach [15]. In loop parallelization [16], loops designed to run sequentially are parallelized by executing their operations concurrently using TM. Providing an order matching the sequential one is fundamental to enforce equivalent semantics for both the parallel and sequential code. Regarding state machine approach [17], many distributed systems order tasks before executing them to guarantee that a single state machine abstraction always evolves consistently on distinct nodes, e.g., Paxos [18]. Some more instances of OrdS problem are Network Function Virtualization systems [19, 20], Osmotic Computing [21], and Cyber-physical Systems [22]. In a Network Function Virtualization (NFV) system, network services are specified as service chains, obtained by the concatenation of network functions which are dependent computational tasks to be executed in the NFV infrastructure such as servers distributed over the network. In Osmotic Computing system, an application is divided into microservices and are deployed over an edge/cloud server infrastructure. When users send requests, an orchestrator handles those requests processing in several of the microservices taking into account the dependencies between the microservices. Cyber-physical systems also involve performing dependent coordinated tasks in multiple subsystems.

OrdS has been studied heavily in multiprocessors [23, 15] where execution time is the only metric of interest. However, those studies focused on empirical studies and they do not extend to distributed multiprocessors as they do not consider latency. Poudel *et al.* [24] studied for the first time the OrdS problem in a distributed multiprocessor. However, they considered the *data-flow* model where transactions

are immobile but the objects are mobile. Since the data-flow model is direct opposite of the control-flow model, the contributions in [24] do not apply to the control-flow model.

Contributions. In this paper, we design a set of scheduling algorithms to solve the OrdS problem in the control-flow model and establish complementary results compared to [24]. We consider *synchronous* communication model [25, 26] where time is divided into discrete steps. Notice that the algorithms we present correctly schedule transactions even when there is no synchronous communication, but the synchronous model helps to establish bounds on performance. We optimize two performance metrics: (i) *execution time* – the total time to execute and commit all the transactions, and (ii) *communication cost* – the total distance messages travel to access shared objects. A transaction's execution finishes as soon as it commits. The presented algorithms determine the time step when each transaction executes and commits. We measure the efficiency using a widely-studied notion of *competitiveness* – the ratio of total time (communication cost) for a designed algorithm to the minimum time (communication cost) achievable by an optimal scheduling algorithm.

Specifically, we have the following six contributions:

1. We provide an impossibility result showing that the optimal execution time and optimal communication cost can not be achieved simultaneously.
2. For the offline version, we provide two algorithms, one with optimal execution time and another with 2-competitive on communication cost.
3. For the partial dynamic version with the knowledge of transactions and their priorities (meaning that all transactions arrive in the beginning) but not the shared objects, we provide an $O(\log^2 n)$–competitive algorithm for both execution time and communication cost.
4. For the fully dynamic version with transactions arriving over time (and hence a transaction may not know how many other transactions are currently in the system and their priorties), we provide an $O(D)$–competitive algorithm for both execution time and communication cost, where D is the diameter of the graph G.
5. We implement and rigorously evaluate the designed algorithms through micro-benchmarks and complex STAMP benchmarks on random, grid and small-world graphs, which validate the efficiency of the designed algorithms.
6. We compare the results of designed algorithms for the control-flow model against the algorithms for data-flow model [24] and analyze them.

Techniques. For the offline version with complete knowledge of transactions, their priorities, and the shared objects they access, we provide two algorithms, one is optimal in terms of execution time and the other algorithm is 2-competitive in terms

of communication cost. The optimal time algorithm uses the shortest path to access required objects. Each transaction sends access requests to all the required objects in parallel following the shortest paths in G. The 2-competitive communication cost algorithm uses that minimum Steiner tree to access the required objects. Each transaction sends (combined) access requests through a minimum Steiner tree that connects the graph nodes containing the required objects.

In the partial dynamic version (with the knowledge of transactions and their priorities but not the shared objects), the proposed algorithm exploits the concept of *distributed directory protocols* [27, 28]. Particularly, the directory protocol technique based on the hierarchical partitioning of the graph into clusters is used. This technique guarantees that the object access cost for a transaction is within an $O(\log^2 n)$ factor from the cost of minimum Steiner tree for that transaction. The directory protocol technique is then extended to the dynamic version guaranteeing $O(D)$-competitiveness without knowing transactions and their priorities a priori. This bound is interesting since the hierarchical partitioning technique used in the partial dynamic version is shown to only provide $O(D \log^2 n)$-competitive bound for the fully dynamic version. The observation is that a transactions knows about the existence of another dependent transaction only after its request reaches the root node of the directory. Therefore, the dynamic algorithm uses the directory protocol running on a spanning tree.

Related Work. Gonzalez-Mesa *et al.* [23] introduced the OrdS problem for multiprocessors and Saad *et al.* [15] presented three improved algorithms and evaluated them through empirical studies. Transaction scheduling with no predefined ordering is widely-studied in multiprocessors providing provable upper and lower bounds, and impossibility results [29, 30, 31, 32, 33], besides several other scheduling algorithms that were only evaluated experimentally [34]. The multiprocessor ideas are not suitable for distributed multiprocessors as they do not deal with a crucial metric, communication cost. The mostly closely related work to ours is Poudel *et al.* [24] where they studied the OrdS problem in the data-flow model (transactions are immobile but objects move to the nodes where transactions are executing). We consider the control-flow model [14] which is the direct opposite of the data-flow model (objects are immobile and transactions send requests to objects) and the solutions in the data-flow model do not apply to the control-flow model.

Many previous studies on transaction scheduling in distributed multiprocessors, e.g., [35, 36, 37, 25, 26, 28, 38], considered the data-flow model. The papers [27, 28, 39] focused on the data-flow model with the objective of minimizing communication cost. Kim and Ravindran [40] provided communication cost bounds for special workloads and problem instances with multiple shared objects. Execution time minimization is considered by Zhang *et al.* [39]. Busch *et al.* [36]

considered minimizing both execution time and communication cost. Busch *et al.* [37] considered special topologies (e.g., grid, line, clique, star, hypercube, butterfly, and cluster) and provided offline algorithms minimizing execution time and communication cost. Poudel and Sharma [41] provided an evaluation framework for executing transactions in distributed multiprocessors. Later, Busch *et al.* [26] provided dynamic (online) algorithms for minimizing execution time and communication cost. However, all these works have no predefined ordering requirement.

Some papers considered the hybrid model that combines data-flow with control-flow. Hendler *et al.* [42] studied a lease based hybrid DTM 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.* [43] presented a comparative study of data-flow versus control-flow models. Recently, Busch *et al.* [44] provided a set of offline algorithms for transactional memory on trees using hybrid model known as the *dual-flow* model, which combines the data-flow and control-flow models by sometimes moving transactions to object nodes and sometimes moving objects to transaction nodes. They provided optimal algorithm when considering a single shared object and $k-$factor away when considering multiple objects where k is the maximum number of objects accessed by a transaction.

Some other papers considered transaction scheduling with no priority ordering requirement in distributed multiprocessors through replication and multi-versioning [45, 11, 46]. We do not consider replication and multi-versioning. In fact, we assume that there is only one copy of the object for both read and write.

Paper Organization. We describe model and preliminaries in Section 2. In Section 3, we establish the impossibility result. In Sections 4, 5 and 6, we provide our algorithms for offline, partial dynamic, and dynamic versions of the OrdS problem, respectively. Experimental results are discussed in Section 7 in which we also compare our results with the data-flow model and analyze them. Finally, we conclude in Section 8 with a short discussion.

2. Model and Preliminaries

Graph. We consider a distributed network $G = (V, E, \mathfrak{w})$ of n nodes (representing processing nodes) $V = \{v_1, v_2, \ldots, v_n\}$, edges (representing communication links between nodes) $E \subseteq V \times V$, and edge weight function $\mathfrak{w} : E \to \mathbb{Z}^+$. A *path* p in G is a sequence of nodes (with respective edges between adjacent nodes) with $\mathsf{length}(p) = \sum_{e \in p} \mathfrak{w}(e)$. We assume that G is connected and $\mathsf{dist}(u, v)$ denotes the shortest path length between two nodes $u, v \in G$. The *diameter* $D := \max_{u,v \in G} \mathsf{dist}(u, v)$, the maximum shortest path distance between two nodes $u, v \in G$. The communication links are *bidirectional* – messages can be sent in both directions. Both the nodes

and links are non-faulty and the links deliver messages in FIFO order. There is no bandwidth restriction on the edges, i.e., the messages can be of any size and any number of messages can traverse an edge at any time. The *k-neighborhood* of a node $u \in G$ is the set of nodes which are at distance $\leq k$ from u.

Transactions. Let $\mathcal{S} = \{S_1, S_2, \ldots, S_w\}$ denote the w shared objects residing on nodes of G. Each object has some value which can be read/written. The node of G where an object S_i is currently positioned is called the *owner* of S_i, denoted as $owner(S_i)$. A transaction $T(v_i, age_i)$ is an atomic block of code mapped at node v_i which requires a set of objects $\mathcal{S}(T(v_i, age_i)) \subseteq \mathcal{S}$ and has priority age_i. To simplify the analysis, we assume that each object has a single copy (for both read/write). We assume that each node runs a single thread and issues transactions sequentially.

Communication Model. We consider the synchronous communication model where time is divided into discrete steps such that at each time step a node receives messages, performs a local computation, and then transmits messages to adjacent nodes [37, 25, 26]. For an edge $e = (u, v) \in E$, it takes $\mathfrak{w}(e)$ time steps to transfer a message msg from u to v (and vice-versa); the *communication cost* contributed by msg is $\mathfrak{w}(e)$. Moreover, we assume that a node can merge or unmerge the messages that are directed to or originated from multiple nodes. If a transaction $T(v_i, age_i)$ contains multiple objects in $\mathcal{S}(T(v_i, age_i))$, then $T(v_i, age_i)$ merges multiple access requests to (all or part of) those objects and sends it along the common path. The intermediate node at which the common path branches to reach the destination node, it unmerges the message and sends individual object access requests to the respective nodes. Inversely, the reply messages are merged at those intermediate nodes and sent to the origin node.

Control-flow Model. In the *control-flow* model, objects are static and transactions move from one node to another to access the objects. Control-flow allows transactions to send *control requests*, in a manner similar to remote procedure calls (RPCs), to the nodes where the required objects are located [14, 47].

Let S_1 be an object required by a transaction $T(v_i, age_i)$. Let $owner(S_1)$ be the owner node of S_1. We define $objAccess(S_1)$ as the control request sent by $T(v_i, age_i)$ to $owner(S_1)$ for accessing object S_1. If there is no conflict on the request, then $T(v_i, age_i)$ performs access operation (read or write) on S_1 and $owner(S_1)$ replies a *grant* message back to v_i. Here, the *grant* message refers that the request for accessing S_1 has been successful. If there is a conflict on the request for accessing S_1, and $T(v_i, age_i)$ cannot perform the requested access operation, then $owner(S_1)$ replies a *deny* message back to v_i. The *deny* message refers that the request for accessing S_1 has not been successful. If v_i receives *grant* message for each of the control requests sent for $T(v_i, age_i)$, then $T(v_i, age_i)$ commits. Other-

wise, $T(v_i, age_i)$ aborts.

In our algorithms, we model control requests in two ways:

i. **Solo Control Request:** In *solo control request*, transaction $T(v_i, age_i)$ sends individual control request $objAccess(S_j)$ for each object $S_j \in S(T(v_i, age_i))$. There will be total $|S(T(v_i, age_i))|$ number of solo control requests per transaction $T(v_i, age_i)$. For each $objAccess(S_j)$, v_i receives either a *grant* or *deny* message. Transaction $T(v_i, age_i)$ commits only if it receives *grant* messages for all the $objAccess(S_j)$ requests where $S_j \in S(T(v_i, age_i))$. We use solo control requests for accessing the objects in parallel.

ii. **Group Control Request:** If a transaction sends a single control request for accessing two or more than two objects, then we define it as a *group control request*. In a group control request, if the requested objects are at different nodes, those are accessed in a recursive order. Let $S(T(v_i, age_i)) := \{S_1, S_2, \ldots, S_k\}$ be the set of required objects for T_i and $\{owner(S_1), owner(S_2), \ldots, owner(S_k)\}$ be the order of nodes for accessing the objects. Then $T(v_i, age_i)$ first sends the group control request $objAccess(S(T(v_i, age_i)))$ to $owner(S_1)$ where it accesses object S_1. If the access operation on S_1 is successful, $owner(S_1)$ forwards the control request to $owner(S_2)$, otherwise it replies a *deny* message back to v_i. The process continues similarly in the recursive order of the nodes until either any node replies a *deny* message or the $objAccess(S(T(v_i, age_i)))$ reaches $owner(S_k)$. Finally, at $owner(S_k)$, when $T(v_i, age_i)$ successfully accesses S_k, it replies back the *grant* message to v_i. As soon as v_i receives the *grant* message for $objAccess(S(T(v_i, age_i)))$, $T(v_i, age_i)$ commits. Otherwise if v_i receives any *deny* for $objAccess(S(T(v_i, age_i)))$, $T(v_i, age_i)$ aborts. We use group control request for accessing the required objects recursively.

Throughout this paper, we use the terms *control request* and *access request* interchangeably.

Transaction Execution and Conflicts. Let a transaction $T(v_i, age_i)$ be located at node v_i and $\mathcal{S}(T(v_i, age_i)) \subseteq \mathcal{S}$ be the set of objects that $T(v_i, age_i)$ is going to read or write. Node v_i then sends object access request (READ or WRITE) on behalf of $T(v_i, age_i)$ to the owner node $owner(S_j)$ of each object $S_j \in \mathcal{S}(T(v_i, age_i))$. For an access request received for S_j from $T(v_i, age_i)$, $owner(S_j)$ handles that request by allowing $T(v_i, age_i)$ to read or write (update) S_j and replies a *grant* message back to v_i. If $owner(S_j)$ receives two access requests for object S_j at the same time and at least one of them is a write request, *conflict* is said to be occurred between transactions accessing S_j. $owner(S_j)$ handles such type of simultaneous access requests by denying at least one request. In case $owner(S_j)$ denies the access request, it replies a *deny* message back to node v_i.

Performance Metrics. We consider two performance metrics fundamental to any distributed system, namely execution time and communication cost. Let $\mathcal{E}$ be an execution schedule following an algorithm $\mathcal{A}$.

Definition 1 (*Execution Time*). *For a set of transactions $\mathcal{T}$, the total time for $\mathcal{E}$ is the time elapsed until the last transaction finishes its execution in $\mathcal{E}$. The execution time of algorithm $\mathcal{A}$ is the maximum time over all possible executions for $\mathcal{T}$.*

Definition 2 (*Communication Cost*). *For a set of transactions $\mathcal{T}$, the communication cost of $\mathcal{E}$ is the sum of the distances messages travel during $\mathcal{E}$. The communication cost of $\mathcal{A}$ is the maximum cost over all possible executions for $\mathcal{T}$.*

The OrdS Problem. Each transaction $T(v_i, age_i)$ is assigned age, age_i, before it is activated, and the age signifies the transaction commit order under dependencies. Following [23, 24, 15], parameter *age* is (i) *unique* – no two transactions can have the same age, (ii) *non-modifiable* – it never changes once assigned, and (iii) *externally determined* – it does not depend on transaction execution. We assume the existence of a module that handles how to determine and assign unique age for each transaction.

For a transaction $T(v_i, age_i)$, let $\mathcal{S}(T(v_i, age_i)) := read(\mathcal{S}(T(v_i, age_i))) \cup write(\mathcal{S}(T(v_i, age_i)))$ where $read(\mathcal{S}(T(v_i, age_i)))$ and $write(\mathcal{S}(T(v_i, age_i)))$ represent the set of objects that need to be read and written by $T(v_i, age_i)$, respectively. We say $T(v_i, age_i)$ is *dependent* on $T(v_j, age_j)$, $age_j < age_i$, if $(write(\mathcal{S}(T(v_i, age_i))) \cap read(\mathcal{S}(T(v_j, age_j))) \neq \emptyset) \vee (read(\mathcal{S}(T(v_i, age_i))) \cap write(\mathcal{S}(T(v_j, age_j))) \neq \emptyset)$. That means, at least an object read/write by $T(v_i, age_i)$ is being written by $T(v_j, age_j)$. If $T(v_i, age_i)$ is dependent on $T(v_j, age_j)$, then $T(v_i, age_i)$ can commit only after $T(v_j, age_j)$ commits. Formally, the OrdS problem is defined as follows:

Definition 3 (*The OrdS problem*). *Given a set of transactions $\mathcal{T} := \{T(v_1, age_1), T(v_2, age_2), \ldots\}$, possibly arriving over time, mapped (arbitrarily) to the nodes of G, commit dependent transactions in $\mathcal{T}$ in the increasing order of age in the control-flow model.*

3. Impossibility Result

We show that it is impossible to simultaneously minimize execution time and communication cost in the control-flow model of distributed transactional memory. Minimizing the execution time results an increase in the communication cost and vice-versa. Busch *et al.* [25] have shown that it is impossible to simultaneously

minimize execution time and communication cost in the data-flow model of distributed transactional memory. In this paper, we show that this result is also true for the control-flow distributed transactional memory.

Let us start with an example. Consider a star graph G as shown in Figure 1 with eight rays going out from the center node. Let there be three nodes on each ray (except the center node). Additionally, let the end nodes of consecutive rays are connected. Suppose there are six objects a, b, c, d, e, and f positioned on six consecutive end nodes, and a transaction T is mapped at the center node and it requests all six objects. All edges have unit weight.

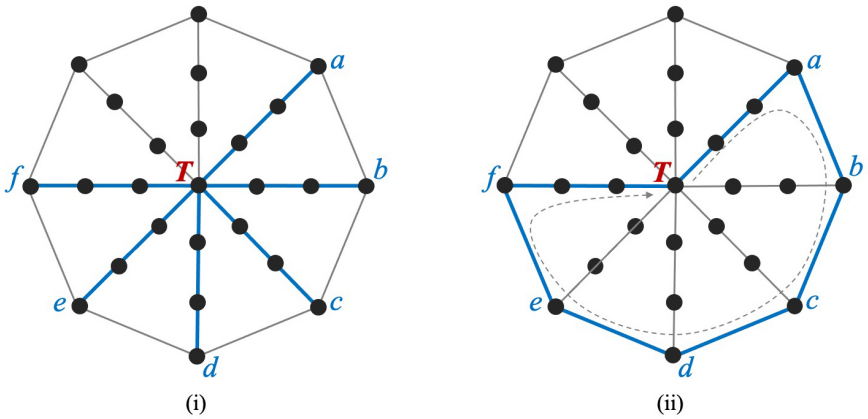

Figure 1: **(i)** Transaction T accessing objects in parallel through blue colored paths, **(ii)** T accessing objects sequentially again through blue colored paths.

Each object is 3 units away from the node where transaction T is located. Thus, in the control-flow model, when T sends solo control requests for accessing the objects in parallel, they can be reached in 3 steps. In next 3 steps, T gets reply (grant) messages from all the object nodes, and in one additional step, it can execute and commit. This gives optimal execution time of 7 steps. However, total communication cost becomes 36 (3 units to reach request to each object and 3 units to receive reply back from each object).

Alternatively, let T accesses all the objects in a sequential order of a, b, c, d, e and f. That means, T sends a group control request for accessing the objects first to a and then to b, c, d, e, f in order. Note here that while processing the group control request at the corresponding node, if the requested object can not be accessed, it replies a *deny* message back to T, otherwise forwards the control request to the next node in order after accessing the object. Finally, when the request reaches f, the node replies a *grant* message back to the node containing T after successfully accessing f. Following the shortest path, the reply (*grant*) message traverses the ray connecting f and T. Then, the total communication cost becomes 11, which is

optimal. But, on the other hand, it takes total 11 time steps to access and receive the reply messages from all the objects. Thus, the total execution time becomes $11 + 1 = 12$, which is sub-optimal.

Figure 1 (i) illustrates the scenario of minimum execution time with an increase in the total communication cost and Figure 1 (ii) illustrates the scenario of minimum communication cost with an increase in the total execution time.

We prove the following theorem:

Theorem 1. *There are transaction scheduling instances for which execution time and communication cost cannot be minimized simultaneously in the control-flow model.*

Proof. Let $\mathcal{T} = \{T(v_1, age_1), T(v_2, age_2), \ldots, T(v_n, age_n)\}$ be the set of transactions and $\mathcal{S} = \{S_1, S_2, \ldots, S_w\}$ be the set of shared objects accessed by the transactions. Both objects and transactions are arbitrarily positioned at the nodes $V = \{v_1, v_2, \ldots, v_n\}$ of graph G. To finish the execution of all the transactions in $\mathcal{T}$ in the minimum possible time, let all of them start at time $t = 0$ and access the required objects in parallel following the shortest path in G. Then, the minimum execution time becomes,

$$Exec_{min} = \max_{T(v_i, age_i) \in \mathcal{T}} \left\{ \max_{S_j \in \mathcal{S}(T(v_1, age_1))} 2 \cdot \mathsf{dist}(v_i, owner(S_j)) + 1 \right\}.$$

And the total communication cost becomes,

$$Comm = \sum_{T(v_i, age_i) \in \mathcal{T}} \left\{ \sum_{S_j \in \mathcal{S}(T(v_1, age_1))} 2 \cdot \mathsf{dist}(v_i, owner(S_j)) \right\}.$$

However, this communication cost is not the minimum possible. Let us see an instance of the schedule that provides minimum communication cost. For each transaction in $T(v_i, age_i) \in \mathcal{T}$, let the required objects are accessed in a sequence of the shortest route to visit all of them. This becomes equivalent to the *traveling salesman problem (TSP)* [48] where the transaction node represents the initial position of the salesman and the nodes of required objects represent the cities where the salesman needs to visit with the minimum cost and finally return to the initial position. Using the minimum cost approximation algorithms [49] for TSP, for example Christofides' algorithm [50] that provides $3/2-$approximation, all the transactions can be executed with the minimum communication cost. Nevertheless, the execution time of each transaction in this case becomes the total length of the route to visit the nodes of the required objects. That means, if $Exec$ is the total execution

time for executing all the transactions in $\mathcal{T}$, then,

$$Exec \geq \max_{T(v_i,age_i)\in\mathcal{T}} \left\{ \max_{S_j\in\mathcal{S}(T(v_1,age_1))} 2 \cdot \mathsf{dist}(v_i, owner(S_j)) + 1 \right\} \geq Exec_{min}.$$

Hence, minimizing the execution time, the communication cost increases and vice-versa. The theorem follows. □

4. Offline Algorithms

In this section, we study the offline version of the OrdS problem. Note that in the offline version, the system has complete knowledge of transactions, their priorities, and the shared objects they access a priori. We present two algorithms, one called OffExec that achieves optimal execution time and another called OffComm that is 2-competitive in communication cost.

4.1. Execution Time Algorithm: OffExec

OffExec accesses required objects for each transaction in parallel. All transactions in $\mathcal{T}$ are initiated at time step $t = 0$. Therefore, at $t = 0$, all the transactions in $\mathcal{T}$ send solo control requests to access the required objects to the respective owner nodes following the shortest paths. Each owner node then replies *grant* message for every request (after performing the read/write operation) respecting the age order and dependency of the transactions at corresponding owner node.

For transaction $T(v_i, age_i)$ at node v_i, let $\mathcal{S}(T(v_i, age_i)) \subseteq \mathcal{S}$ be the set of objects it needs. $T(v_i, age_i)$ sends corresponding access requests to $owner(S_j)$ of each object $S_j \in \mathcal{S}(T(v_i, age_i))$ following the shortest path from v_i to $owner(S_j)$. After the access request reaches $owner(S_j)$, $owner(S_j)$ sends *grant* message back to v_i as soon as $T(v_i, age_i)$ is able to read/write that object respecting the age order. Specifically, there can be two cases: (i) There is no $T(v_k, age_k), age_k < age_i$, in $\mathcal{T}$ which also wants to access S_j, then $owner(S_j)$ immediately sends *grant* message back to v_i (ii) There is another transaction $T(v_k, age_k), age_k < age_i$, in $\mathcal{T}$ that conflicts with $T(v_i, age_i)$ while accessing S_j, then $owner(S_j)$ sends *grant* message to v_k first and to v_i in the next time step. When v_i receives *grant* messages from all $owner(S_j)$, $T(v_i, age_i)$ finishes its execution and commits.

Let $t_i^{S_j}$ be the time step at which $owner(S_j)$ of object $S_j \in S(T(v_i, age_i))$ replies *grant* message back to node v_i corresponding to the request sent by $T(v_i, age_i)$. Then,

$$t_i^{S_j} = \max\{t^{S_j}_{prev(T(v_i,age_i))} + 1, \mathsf{dist}(v_i, owner(S_j))\},$$

where $t^{S_j}_{prev(T(v_i,age_i))}$ is the time step at which $owner(S_j)$ replies to the dependent transaction of $T(v_i, age_i)$ that is immediately previous to $T(v_i, age_i)$ in the age order.

For the lowest aged transaction $T(v_1, age_1)$,

$$t_1^{S_j} = \mathsf{dist}(v_1, owner(S_j)).$$

Let CT_i be the time step at which transaction $T(v_i, age_i) \in \mathcal{T}$ commits. Then,

$$CT_i = \begin{cases} CT_{prev(T(v_i,age_i))} + 1, & \text{if } t'_i < CT_{prev(T(v_i,age_i))} \\ t'_i + 1, & \text{otherwise.} \end{cases}$$

where $CT_{prev(T(v_i,age_i))}$ is the time at which the transaction dependent to $T(v_i, age_i)$ that is immediately previous to $T(v_i, age_i)$ in the age order commits and

$$t'_i = \max_{S_j \in \mathcal{S}(T(v_i,age_i))} (t_i^{S_j} + \mathsf{dist}(v_i, owner(S_j))).$$

For the lowest aged transaction $T(v_1, age_1)$,

$$CT_1 = \max_{S_j \in \mathcal{S}(T(v_1,age_1))} 2 \cdot \mathsf{dist}(v_1, owner(S_j)) + 1.$$

Theorem 2. OffExec *achieves optimal execution time.*

Proof. The execution time depends on two factors. First, how long does a transaction take to access required objects and second, when does each transaction commit? In OffExec, each transaction accesses required objects using the shortest path in G which is thus optimal. Now, we need to show that each transaction commits at the earliest possible time. First, let there is no conflict between any transactions in $\mathcal{T}$. Then all the transactions can access required objects in parallel and as soon as each transaction receives *grant* messages from the owner nodes of each required object, it can commit. The total execution time becomes

$$\max_{T(v_i,age_i) \in \mathcal{T}} \left\{ \max_{S_j \in \mathcal{S}(T(v_1,age_1))} 2 \cdot \mathsf{dist}(v_i, owner(S_j)) + 1 \right\}$$

which is optimal. Now, let there are conflicts between transactions while accessing objects. Let $\mathcal{T} = \{T(v_1, age_1), T(v_2, age_2), \ldots, T(v_n, age_n)\}$ be the set of transactions. Let a dependency graph $H = (V_H, E_H)$ holds the dependency between the conflicting transactions where the nodes V_H represent transactions in $\mathcal{T}$ and the directed edges E_H represent dependencies between the transactions. The edge $(T(v_i, age_i), T(v_j, age_j)) \in E_H$, where $age_i < age_j$, represents a dependency between $T(v_i, age_i)$ and $T(v_j, age_j)$ such that $T(v_j, age_j)$ can commit only after $T(v_i, age_i)$ commits. The OrdS problem requires the dependent transactions to commit in their age order. The diameter D_H of H provides the longest chain of

dependent transactions and the total execution time of any optimal algorithm will be the time required by all the transactions that belong to D_H to commit. During the execution of OffExec, for each transaction $T(v_i, age_i)$, if there is no any dependent transaction in H or all the dependent transactions in H have already been committed, then $T(v_i, age_i)$ can commit as soon as it receives *grant* messages from the owner nodes of all required objects. Note that, both object access requests and *grant* messages are sent through the shortest paths in G. When the highest age transaction that belongs to D_H of H commits, OffExec finishes. Hence, the total execution time is optimal. □

Theorem 3. OffExec *is k-competitive in communication cost, where k is the maximum number of shared objects accessed by a transaction in* $\mathcal{T}$.

Proof. Let G be a graph with $n > k$ nodes and $\mathcal{T} = \{T(v_1, age_1), T(v_2, age_2), \ldots, T(v_n, age_n)\}$ be the set of transactions, each accessing at most k shared objects. Then, using OffExec, communication cost incurred by a transaction $T(v_i, age_i)$ is

$$Comm_i = 2 \cdot \sum_{S_j \in \mathcal{S}(T(v_i, age_i))} \mathsf{dist}(v_i, owner(S_j)) \leq 2 \cdot k \cdot n = O(k \cdot n).$$

On the other hand, using any optimal communication cost algorithm in the control-flow model, the transaction can access the required objects in a sequential order. Then, the optimal communication cost becomes

$$Comm_{opt} = 2 \cdot c \cdot \max_{S_j \in \mathcal{S}(T(v_i, age_i))} \mathsf{dist}(v_i, owner(S_j)) = O(n),$$

where $c \leq k$ is a constant. Hence, $\frac{Comm_i}{Comm_{opt}} = \frac{O(k \cdot n)}{O(n)} = O(k)$. □

4.2. Communication Cost Algorithm: OffComm

In OffComm, we convert the execution of each transaction to a Minimum Steiner Tree (MST) [51, 52]. Steiner trees have been extensively studied in the context of weighted graphs [53]. Given a graph $G = (V, E)$ and a subset $P \subseteq V$, a *Steiner tree* spans through P. The Steiner tree problem in our case is to find a Steiner tree that connects all the vertices of P with the minimum possible total weight. Computing MST is known to be NP-Hard. We follow the algorithm of Takahashi and Matsuyama [54] which provides $2(1 - 1/|P|)$–approximation for MST. The algorithm of [54] constructs a Steiner tree as follows:

- Start from a participant node in P.
- Find the next participant that is closest to the current tree.
- Join the closest participant to the closest node of the tree.

- Repeat until all nodes in P are connected.

Now, we discuss how MST is constructed for each transaction in $\mathcal{T}$. Let $\mathcal{S}(T(v_i, age_i)) \subseteq \mathcal{S}$ be the set of objects required by a transaction $T(v_i, age_i) \in \mathcal{T}$. Let $P_i \subseteq V$ contains node v_i and the owner node of each object $S_j \in \mathcal{S}(T(v_i, age_i))$ (i.e., $P_i := (\forall_{S_j \in \mathcal{S}(T(v_i, age_i))} owner(S_j)) \cup v_i$). Now, the problem is to find a MST that connects the nodes in P_i which is constructed by following the algorithm of [54] and is denoted as MST_i. Then, $T(v_i, age_i)$ sends object access requests in MST_i. The total message cost incurred by transaction $T(v_i, age_i)$ is $2.|MST_i|$. That means, messages visit each edge of MST_i exactly twice, one for sending access request and the other for receiving reply (*grant* or *deny*) message from each owner node.

Instead of sending requests individually to access the objects in $S(T(v_i, age_i))$, $T(v_i, age_i)$ sends them collectively in MST_i. Each neighboring node recursively sends the request to the next neighbor in MST_i until the request reaches all the owner nodes of the required objects. To be specific, if $v_p, v_q \in MST_i$ be any two owner nodes of objects which share a common path from v_i up to some intermediate node v_s, then the requests to v_p and v_q from v_i are sent collectively up to v_s as a single message. The request is then divided into two at v_s and they are forwarded separately towards v_p and v_q. When all the access requests reach respective owner nodes, the reply messages are collected in the opposite direction. Here, each intermediate node which had initially sent access requests to the neighboring nodes later collects the reply messages from those neighboring nodes and returns them collectively to the ancestor node. When v_i receives reply messages from all the neighboring nodes in MST_i, $T(v_i, age_i)$ commits (provided that all the reply messages are *grant* messages).

The OffComm algorithm works as follows. It produces a conflict-free execution schedule. At time step $t = 0$, each transaction $T(v_i, age_i)$ sends access requests to required objects following its corresponding MST_i. When the access request reaches $owner(S_j)$, $owner(S_j)$ sends *grant* message back to v_i as soon as $T(v_i, age_i)$ is able to read/write that object respecting the age order of the dependent transactions. Let $\mathsf{dist}_{MST_i}(v_i, v_j)$ represents the distance between nodes v_i and v_j following the shortest path in MST_i. Then, for each $T(v_i, age_i) \in \mathcal{T}$, $owner(S_j)$ of each $S_j \in S(T(v_i, age_i))$ replies *grant* message to v_i at time step:

$$t_i^{S_j} = \max\{t^{S_j}_{prev(T(v_i, age_i))} + 1, \mathsf{dist}_{MST_i}(v_i, owner(S_j))\},$$

where $t^{S_j}_{prev(T(v_i, age_i))}$ is the time step at which $owner(S_j)$ replies to the dependent transaction of $T(v_i, age_i)$ that is immediately previous to $T(v_i, age_i)$ in the age order.

The commit time step CT_i for each $T(v_i, age_i)$ is:

$$CT_i = \begin{cases} CT_{prev(T(v_i,age_i))} + 1, & \text{if } t'_i < CT_{prev(T(v_i,age_i))} \\ t'_i + 1, & \text{otherwise.} \end{cases}$$

where $CT_{prev(T(v_i,age_i))}$ is the time at which the transaction dependent to $T(v_i, age_i)$ that is immediately previous to $T(v_i, age_i)$ in the age order commits and

$$t'_i = \max_{S_j \in \mathcal{S}(T(v_i,age_i))} (t_i^{S_j} + \mathsf{dist}_{MST_i}(v_i, owner(S_j))).$$

Theorem 4. OFFCOMM *is 2-competitive in communication cost.*

Proof. Let MST_i be the minimum cost Steiner tree constructed for transaction $T(v_i, age_i)$ in OFFCOMM. Let $\mathsf{dist}_{MST_i}(v_x, v_y)$ be the shortest path distance between v_x and v_y in MST_i. If $\mathsf{dist}(v_x, v_y)$ be the shortest path distance in G, then we have: $\mathsf{dist}_{MST_i}(v_x, v_y) \leq 2 \cdot \mathsf{dist}(v_x, v_y)$. Since OFFCOMM follows the shortest paths in respective MSTs for accessing required objects, the communication cost $C_{T(v_i,age_i)}$ of executing each transaction $T(v_i, age_i) \in \mathcal{T}$ is:

$$C_{T(v_i,age_i)} = 2 \cdot C_{opt}^{T(v_i,age_i)},$$

where $C_{opt}^{T(v_i,age_i)}$ is the cost of any optimal communication algorithm for executing $T(v_i, age_i)$ that accesses required objects following the shortest paths in G. If C_{total} and C_{opt} be the total communication costs of OFFCOMM and any optimal algorithm, respectively, such that $C_{opt} = \sum_{T(v_i,age_i)\in\mathcal{T}} C_{opt}^{T(v_i,age_i)}$, then,

$$C_{total} = \sum_{T(v_i,age_i)\in\mathcal{T}} C_{T(v_i,age_i)} = \sum_{T(v_i,age_i)\in\mathcal{T}} 2 \cdot C_{opt}^{T(v_i,age_i)} = 2 \cdot C_{opt}.$$

□

Theorem 5. OFFCOMM *is r-competitive in execution time, where r is the maximum stretch of MST computed for each transaction in* $\mathcal{T}$ *which is given by:*

$$r = \max_{T(v_i,age_i)\in\mathcal{T}} \left\{ \max_{S_j \in \mathcal{S}(T(v_i,age_i))} \frac{\mathsf{dist}_{MST}(v_i, owner(S_j))}{\mathsf{dist}(v_i, owner(S_j))} \right\}.$$

Proof. OFFEXEC provides optimal execution time by accessing the required object by a transaction $T(v_i, age_i)$ through the shortest path in G. In OFFCOMM, objects are accessed by $T(v_i, age_i)$ using the shortest path in MST_i built on G for $T(v_i, age_i)$. If $\mathsf{dist}(v_i, v_j)$ and $\mathsf{dist}_{MST_i}(v_i, v_j)$ are the shortest path distances between two nodes

v_i, v_j in G and MST_i, respectively, then the stretch of MST_i (i.e., $r = \frac{\mathsf{dist}_{MST_i}(v_i,v_j)}{\mathsf{dist}(v_i,v_j)}$) provides the competitiveness for $T(v_i, age_i)$ for the time required to access any object at v_j. While executing all the transactions,

$$\max_{T(v_i,age_i)\in\mathcal{T}} \left\{ \max_{S_j\in\mathcal{S}(T(v_i,age_i))} \frac{\mathsf{dist}_{MST}(v_i, owner(S_j))}{\mathsf{dist}(v_i, owner(S_j))} \right\}$$

provides the necessary competitiveness. □

5. Partial Dynamic Algorithm

Here we study the partial dynamic version of the OrdS problem, where a priori knowledge on transactions and their priorities is available, but not the shared objects they access and their locations. All transactions arrive at time $t = 0$. Thus, the following two tasks are additional to the offline version:

i. Find the owner nodes of all the shared objects that a transaction requests.
ii. Find the node where the next transaction in the commit order is located and the path to reach that node.

We present an efficient algorithm PartDyn using the well-studied distributed directory protocol technique [35, 55, 27, 56, 28]. We compute two distributed queues, the first helps transactions accessing required objects and the second helps sending commit messages to the next dependent transaction in age order. The first is called *distributed object queue* where *object access tours* are constructed for each transaction. The second is called *distributed transaction queue* that satisfies the commit order of transactions. Each transaction sends commit message to the next transaction in order following the path in its respective transaction tour in the distributed transaction queue. We use the hierarchy-of-clusters-based overlay tree ($\mathcal{OT}$) (discussed next) for the computation of both queues.

Overlay Tree $\mathcal{OT}$ Construction. The well-known approaches for $\mathcal{OT}$ construction are based on either a *spanning tree* or a *hierarchy of clusters* on G. The spanning tree was used in directory protocols [55, 35] and the hierarchy of clusters was used in directory protocols [27, 28, 38].

Both approaches work, however, hierarchy-of-clusters-based overlay trees are more suitable to control communication costs (and hence the execution time) compared to the spanning-tree-based overlay trees. Therefore, in the following, we discuss the construction of hierarchy-of-clusters-based overlay tree $\mathcal{OT}$. In a high level, divide the graph G into a hierarchy of clusters with $H_1 = \lceil \log D \rceil + 1$ layers such that the clusters sizes grow exponentially (i.e., $2^\ell, 0 \le \ell \le H_1$). A *cluster* is a subset of nodes, and its diameter is the maximum distance between any two nodes.

The diameter of each cluster at layer ℓ, where $0 \leq \ell < H_1$, is no more than $f(\ell)$, for some function f, and each node participates in no more than $g(\ell)$ clusters at layer ℓ, for some other function g. Moreover, 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.

There are known algorithms, such as a *hierarchical sparse cover* of G, that give a cluster hierarchy $\mathcal{Z}$ of H_1 layers with $f(\ell) = O(\ell \log n)$ and $g(\ell) = O(\log n)$. This construction was used in the directory protocol, Spiral, by Sharma *et al.* [28], where additionally, each layer ℓ is decomposed into $H_2 = O(\log n)$ sub-layers of clusters, such that a node participates in all the sub-layers of a layer but in a different cluster within each sub-layer, i.e., at each layer ℓ a node u participates in $g(\ell) = O(\log n)$ clusters. Suppose a node in each cluster is designated as the *leader* of the cluster. Connecting the leaders of the clusters in the subsequent levels gives $\mathcal{OT}$.

An *upward path* $p(u)$ for each node $u \in G$ is built by visiting leader nodes in all the clusters that u belongs to starting from layer 0 (the bottom layer in $\mathcal{Z}$) up to layer H_1 (the top layer in $\mathcal{Z}$). Within each layer, H_2 sub-layers are visited by $p(u)$ according to the order of their sub-layer labels. The upward path $p(u)$ visits two subsequent leaders using shortest paths in G between them. Lets say two paths *intersect* if they have a common node. Using this definition, two upward paths intersect at layer i if they visit the same leader at layer i. The lemmas below are satisfied in the construction of [28].

Lemma 1. *The upward paths $p(u)$ and $p(v)$ of any two nodes $u, v \in G$ intersect at layer* $\min\{H_1, \lceil \log(\mathsf{dist}(u, v)) \rceil + 1\}$.

Lemma 2. *For any upward path $p(u)$ for any node $u \in G$ from the bottom layer upto layer ℓ (and any sub-layer in layer ℓ),* $\mathsf{length}(p(u)) \leq O(2^\ell \log^2 n)$.

Computing Distributed Transaction Queue. We denote the distributed transaction queue by $DTQueue(\mathcal{T})$. To construct $DTQueue(\mathcal{T})$, each transaction $T(v_i, age_i)$ sends a $findT(T(v_i, age_i))$ message in its upward path $p(v_i)$ in $\mathcal{OT}$. The $findT(T(v_i, age_i))$ message contains information about the required objects by $T(v_i, age_i)$ and moves upward until it meets the similar messages sent by it's previous and next conflicting transactions in age order. When two messages $findT(T(v_i, age_i))$ and $findT(T(v_j, age_j))$ meet at some node v_k, it can easily be found that whether $T(v_i, age_i)$ and $T(v_j, age_j)$ conflict with each other or not by looking at the information of required objects for each of them. When such meetings happen for all $findT(prev(T(v_i, age_i))$, $findT(T(v_i, age_i))$, and $findT(next(T(v_i, age_i)))$, $1 \leq i \leq n$, the computation of $DTQueue(\mathcal{T})$ is completed.

The upward paths $p(v_i)$ and $p(v_j)$ for the two consecutive dependent transactions $T(v_i, age_i)$ and $T(v_j, age_j)$ intersect at some node v_k at some layer $l > 0$. Transaction

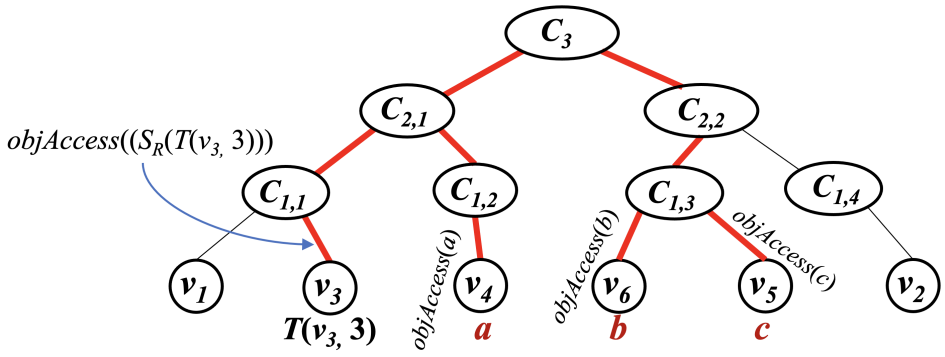

Figure 2: Illustration of computation of distributed object queue for transaction $T(v_3, 3)$ requiring objects (a, b, c). $T(v_3, 3)$ sends $objAccess(S_R(T(v_3, 3)))$ message in its upward path to cluster $C_{1,1}$ which recursively sends it to $C_{2,1}$. $C_{2,1}$ contains the owner node of object a (i.e., v_4), thus sends $objAccess(a)$ message to v_4. Then, after removing a from $S_R(T(v_3, 3))$, $C_{2,1}$ sends $objAccess(S_R(T(v_3, 3)))$ message to cluster C_3. C_3 sends the message downward until the requests reach nodes v_5 and v_6. Later, all three nodes v_4, v_5, and v_6 reply *grant* messages which are combined at clusters $C_{1,3}$ and $C_{2,1}$, and finally reach node v_3. Then $T(v_3, 3)$ commits. The edges traversed by the messages are highlighted in red.

$T(v_i, age_i)$ sends a commit message to $T(v_j, age_j)$ by first sending it upward in $p(v_i)$ up to v_k and then sending the message downward in $p(v_j)$ from v_k up to node v_j. The following theorem follows from the hierarchy of clusters based $\mathcal{OT}$.

Theorem 6. *If d is the shortest path distance between nodes $v_i, v_j \in G$, then the distance between v_i, v_j following the upward paths $p(v_i)$ and $p(v_j)$ in $\mathcal{OT}$ is $O(d \cdot \log^2 n)$.*

Computing Distributed Object Queues. Distributed object queue for each transaction $T(v_i, age_i) \in \mathcal{T}$ is denoted as $DOQueue(T(v_i, age_i))$. $DOQueue(T(v_i, age_i))$ contains object tour(s) to access the object(s) requested by $T(v_i, age_i)$.

$DOQueue(T(v_i, age_i))$ is constructed as follows. Let $\mathcal{S}_R(T(v_i, age_i)) \subseteq \mathcal{S}(T(v_i, age_i))$ be the set of objects required by $T(v_i, age_i)$ that are not present on v_i. $T(v_i, age_i)$ sends $objAccess(S_R(T(v_i, age_i)))$ message in its upward path $p(v_i)$. Let at some level $l > 0$, $objAccess(\mathcal{S}_R(T(v_i, age_i)))$ reaches a cluster with node v_j that contains an object $S_j \in \mathcal{S}_R(T(v_i, age_i))$. Then the leader of the cluster (say v_l) forwards $objAccess(S_j)$ to the node v_j downward in the path $p(v_j)$. The leader also removes object S_j from $\mathcal{S}_R(T(v_i, age_i))$ and forwards $objAccess(\mathcal{S}_R(T(v_i, age_i)))$ message upward in the path $p(v_i)$ if $\mathcal{S}_R(T(v_i, age_i))$ is not empty. This process continues until $\mathcal{S}_R(T(v_i, age_i))$ becomes empty and by that time, the computation of $DOQueue(T(v_i, age_i))$ is completed.

Later, during the execution of $T(v_i, age_i)$, when the object access request $objAccess(S_j)$ reaches the owner node of S_j, $owner(S_j)$, $T(v_i, age_i)$ performs read

or write operation on S_j. After the read or write operation is completed, v_j replies a *grant* message back following the previous path in the opposite direction (i.e., upward from v_j to the leader node v_l in $p(v_j)$). Each leader node when receives reply messages from the owner nodes of objects, combines them into a single message and sends it back downward in the path $p(v_i)$ to node v_i. The leader node waits to combine the reply message until it receives reply messages from all the paths that it has sent previously the access requests. Figure 2 illustrates this idea.

Algorithm PartDyn**.** PartDyn starts with computing distributed object queues $DOQueue(T(v_i, age_i))$ for each transaction $T(v_i, age_i) \in \mathcal{T}$ and distributed transaction queue $DTQueue(\mathcal{T})$. $DOQueue(T(v_i, age_i))$ contains object tours to access all the required objects in $\mathcal{S}(T(v_i, age_i))$.

All the transactions that do not depend on any lower aged transactions start execution at time $t = 0$. $T(v_1, age_1)$ starts at $t = 0$ and sends object access requests recursively following object tours in $DOQueue(T(v_1, age_1))$. Then, for each object $S_j \in \mathcal{S}(T(v_1, age_1))$, $objAccess(S_j)$ reaches the owner node $owner(S_j)$. $T(v_1, age_1)$ performs read or write operation on all S_j and a *grant* message from each $owner(S_j)$ is replied back following the object tours in the backward direction. $T(v_1, age_1)$ commits after it receives *grant* messages from all the owner nodes of required objects (possibly in combined form). Let $T(v_1, age_1)$ commits at time step $t_1 > 0$. $T(v_1, age_1)$ sends commit message $commit(T(v_1, age_1))$ to the next conflicting transaction in age order $next(T(v_1, age_1)) = T(v_k, age_k), age_k > age_i$, by following upward paths in $DTQueue(\mathcal{T})$. When $T(v_k, age_k)$ receives commit messages from all the dependent transactions, $T(v_k, age_k)$ executes and commits at time step $t_k > t_1$ and sends $commit(T(v_k, age_k))$ message to $next(T(v_k, age_k))$. The process continues until the highest aged transaction $T(v_h, age_h)$ commits at some time step t_h.

Theorem 7. PartDyn *is* $O(\log^2 n)$*-competitive in both execution time and communication cost.*

Proof. The OrdS problem requires all the dependent transactions to commit in their age orders. So, if $ST \subseteq \mathcal{T}$ be the set of transactions containing the longest chain of dependent transactions in $\mathcal{T}$, the optimal total execution time for executing all the transactions in $\mathcal{T}$ is

$$t_{opt} = \sum_{T(v_i, age_i) \in ST} \left(\max_{S_j \in \mathcal{S}(T(v_i, age_i))} \mathsf{dist}\left(v_i, owner(S_j)\right) \right).$$

In PartDyn, each transaction $T(v_i, age_i)$ sends commit message to the next conflicting transaction in the age order $next(T(v_i, age_i)$ to maintain the predefined commit order. We have from Theorem 6 that the distance between any two nodes in

the hierarchy of clusters based $\mathcal{OT}$ increases by $O(\log^2 n)$ factor. In PartDyn, each transaction $T(v_i, age_i)$ uses $DOQueue(T(v_i, age_i))$ constructed using the hierarchy of clusters based $\mathcal{OT}$ for accessing required objects. Then, the distance from v_i to the owner node $owner(S_j)$ of each object $S_j \in \mathcal{S}(T(v_i, age_i))$ also increases by $O(\log^2 n)$ factor. Moreover, when $T(v_i, age_i)$ successfully accesses all the required objects, it commits and sends the commit message to the next conflicting transaction in the age order $T(v_j, age_j)$. To send the commit message, $T(v_i, age_i)$ uses $DTQueue(\mathcal{T})$ and hence, again from Theorem 6, the distance between v_i and v_j increases by $O(\log^2 n)$ factor. Let $\mathsf{dist}_{PART}(v_i, v_j)$ represents the distance between any two nodes v_i and v_j in PartDyn. Then,

$$\mathsf{dist}_{PART}(v_i, v_j) = \mathsf{dist}(v_i, v_j) \cdot O(\log^2 n)$$

Let t_{PART} be the total execution time in PartDyn, which becomes

$$\begin{aligned}
t_{PART} &= \sum_{T(v_i, age_i) \in ST} \Big(\max_{S_j \in \mathcal{S}(T(v_i, age_i))} \mathsf{dist}_{PART}(v_i, owner(S_j)) \\
&\quad + \mathsf{dist}_{PART}(v_i, owner(next(T(v_i, age_i))))\Big) \\
&= \sum_{T(v_i, age_i) \in ST} \Big(\max_{S_j \in \mathcal{S}(T(v_i, age_i))} \mathsf{dist}(v_i, owner(S_j)) \\
&\quad + \mathsf{dist}(v_i, owner(next(T(v_i, age_i)))) \cdot O(\log^2 n) \\
&\leq \sum_{T(v_i, age_i) \in ST} \Big(\max_{S_j \in \mathcal{S}(T(v_i, age_i))} \mathsf{dist}(v_i, owner(S_j))\Big) \\
&\quad \cdot k \cdot O(\log^2 n) \\
&\leq t_{opt}.O(\log^2 n).
\end{aligned}$$

Any optimal communication cost algorithm communicates between nodes by using the shortest paths in G. Since PartDyn uses hierarchy of cluster based $\mathcal{OT}$, the distance between any two nodes increases by $O(\log^2 n)$ factor. Thus, the cost of sending object requests and receiving *grant* messages increases by $O(\log^2 n)$. In addition to this, each $T(v_i, age_i)$ sends $commit(T(v_i, age_i))$ to the next conflicting transaction in the age order. Which increases the communication cost by $O(k.\log^2 n) = O(\log^2 n)$. That means, in total, communication increases by the factor of $O(\log^2 n)$. □

6. Fully Dynamic Algorithm

Here, we study OrdS with no a priori knowledge on transactions, their priorities, and the locations of the shared objects they access. Additionally, transactions arrive

at different nodes of G arbitrarily over time. Once a transaction arrives at some node v_i, it knows the priority (i.e., age) of that transaction and the objects needed by it. Note that the age parameter of a transaction is unique in the dynamic case as well and is defined when the transaction is assigned to a node. We can assume the existence of a module that provides a unique age for each transaction issued. For an illustration, suppose a distributed system containing a centralized server which accepts job requests from end users continuously over time. The server assigns the received job requests to different processing nodes by defining the priority of the job. We present an algorithm Dyn that achieves $O(D)$ competitive ratio in both execution time and communication cost. Algorithm Dyn works on top of a spanning-tree-based overlay tree, denoted as OT_{ST}. The tree-based overlay is used to control the costs since the requests may reach to the root of the overlay in the worst-case and the cluster based overlays incur more cost than the tree based overlays/. Let v_{root} be the root node of $\mathcal{OT}_{ST}$. For any node v, the upward path $p(v)$ in $\mathcal{OT}_{ST}$ is the path obtained by connecting the parent nodes in ST from node v up to the root v_{root}. Dyn executes in two phases:

- **Phase 1 – Object Advertisement** in which each node of graph G is advertised with the locations of all the objects.
- **Phase 2 – Transaction Execution** in which transactions are executed and committed according to age order.

Phase 1 – Object Advertisement. The object advertisement phase makes each node of G know the locations of all the shared objects. Later, when a transaction at node v_i needs some object S_j, v_i can forward object access request to the owner node of that object. The ownership of each object is advertised in the form of a hash map where each key-value pair represents $(objID, nodeID)$, where $objID$ is the ID of an object located at node $v \in V$ and $nodeID$ is the ID of v.

Execution starts from leaf nodes of $\mathcal{OT}_{ST}$. Each leaf node v_l sends a hash map $(objID, nodeID)$. If v_l contains no object, v_l sends an empty hash map. Also, if v_l contains more than one object, it sends a hash map with multiple key-value pairs. When a parent node v_{p1} receives hash maps from all its child nodes, v_{p1} merges those into a single hash map and appends new key-val pair(s) if it contains any object(s). The updated hash map is then sent upward to the next parent node v_{p2}. v_{p2} again merges all hash maps into a single one after receiving from all the child nodes. This process is repeated until the current node is the root v_{root}. When v_{root} receives hash maps from all of its child nodes, it merges them into a single hash map and replies back the updated hash map to all the child nodes recursively. This phase ends when all the leaf nodes receive updated hash map containing all $(objID, nodeID)$ pairs.

Lemma 3. Phase 1 finishes in $O(D)$ time steps.

Proof. Since $\mathcal{OT}_{ST}$ is based on minimum spanning tree, the height of $\mathcal{OT}_{ST}$ is $O(D)$ where D is the diameter of graph G. The execution of Phase 1 starts from leaf nodes of $\mathcal{OT}_{ST}$. Each leaf node sends corresponding hash maps to their parent nodes with the information of objects located at it. Parent nodes merge the hash maps received from all the child nodes and send them upward to the respective parent nodes. Following this process, the root node v_{root} of $\mathcal{OT}_{ST}$ receives all the hash maps sent from leaf nodes in $O(D)$ time steps. Now, v_{root} merges all the hash maps received from all its child nodes and sends the merged hash map downward up to the leaf nodes, which also takes $O(D)$ time steps. Phase 1 ends after all the leaf nodes receive the merged hash map from v_{root}. Thus, in total, Phase 1 finishes in $2 \cdot O(D) = O(D)$ time steps. □

Lemma 4. The communication cost in Phase 1 is $O(n)$.

Proof. There are total n nodes and $n-1$ edges in $\mathcal{OT}_{ST}$. Each node sends one message (i.e., hash map) to the parent node in upward direction (except v_{root}) and one message to the child nodes in downward direction (except leaf nodes). That means, there are exactly two messages that traverse each edge of $\mathcal{OT}_{ST}$. Thus, the total communication cost becomes $2(n-1)$, i.e., $O(n)$. □

Phase 2 – Transaction Execution. Let H be the height of $\mathcal{OT}_{ST}, H \leq D$. As soon as transaction $T(v_i, age_i)$ is initiated, it sends an arrival message $T_{arrival}(T(v_i, age_i), t_i)$ to v_{root} following the upward path $p(v_i)$, where t_i is the time step at which $T(v_i, age_i)$ arrives at node v_i and age_i is unique to the transaction $T(v_i, age_i)$ which shows the priority order of the $T(v_i, age_i)$. Let $\mathcal{T}_t(v_{root})$ be a list maintained by v_{root} which contains the information of pending transactions at time step t sorted by arrival time. The arrival message $T_{arrival}(T(v_i, age_i), t_i)$ sent from node v_i reaches v_{root} in $\leq H$ time steps. Thus, when v_{root} receives a transaction arrival message $T_{arrival}(T(v_i, age_i), t_i)$ at some time step $t_r \geq t_i$, it includes $T(v_i, age_i)$ in $\mathcal{T}_t(v_{root})$ at time step $t'_i = t_i + H$.

Let $T(v_x, age_x) \in \mathcal{T}_t(v_{root})$ be the lowest age transaction in $\mathcal{T}_t(v_{root})$ at time t. v_{root} sends $startExec(T(v_x, age_x))$ message to node v_x to execute $T(v_x, age_x)$. $T(v_x, age_x)$ sends object access requests to the owner nodes of $\mathcal{S}(T(v_x, age_x))$. When $T(v_x, age_x)$ successfully accesses all the required objects in $\mathcal{S}(T(v_x, age_x))$, it commits and sends a commit message to v_{root}. Then, v_{root} removes $T(v_x, age_x)$ from $\mathcal{T}_t(v_{root})$ and schedules next conflicting transaction in the age order to execute. Note that, v_{root} can schedule multiple transactions together which are not dependent on any lower aged transactions or receive commit messages from all the dependent transactions during the execution. Phase 2 finishes when all the transactions in $\mathcal{T}$ commit.

Lemma 5. In Phase 2, each transaction finishes its execution in $O(D)$ time steps.

Proof. Let transaction $T(v_i, age_i) \in \mathcal{T}$ starts execution at time step t_i. $T(v_i, age_i)$ sends access requests to the owner nodes of the objects in $\mathcal{S}(T(v_i, age_i))$ which takes at most D time steps to reach them. When the request reaches the respective owner node of the object, $T(v_i, age_i)$ accesses that object and the owner node replies back a *grant* message to the node v_i. The *grant* message takes at most another D time steps to reach node v_i. So, at time step $t'_i \leq t_i + 2D$, $T(v_i, age_i)$ successfully accesses all the required objects and receives *grant* messages from each owner node of the required objects. Then, $T(v_i, age_i)$ commits at time step $t''_i \leq t_i + 2D + 1$ and sends commit message $commit(T(v_i, age_i)$ to v_{root}. $commit(T(v_i, age_i)$ reaches v_{root} at time step $t'''_i \leq t_i + 3D + 1$. v_{root} now removes $T(v_i, age_i)$ from $\mathcal{T}_t(v_{root})$ and schedules next transaction which was dependent on $T(v_i, age_i)$ to execute. Therefore, each transaction in Phase 2 finishes its execution in $O(3D + 1) = O(D)$ time steps. □

Lemma 6. The communication cost for each transaction in Phase 2 is $O(D)$-competitive.

Proof. The communication cost for executing a transaction $T(v_i, age_i)$ in Phase 2 of Dyn consists of the traversal of four types of messages: $T_{arrival}(T(v_i, age_i), t_i)$ (from node v_i to v_{root}), $startExec(T(v_i, age_i))$ (from v_{root} to v_i), $objAccess(*)$ (request and response messages to and from the owner nodes of all the required objects of $T(v_i, age_i)$), and $commit(T(v_i, age_i))$ (from v_i to v_{root}). The communication cost incurred due to $T_{arrival}(T(v_i, age_i), t_i)$, $startExec(T(v_i, age_i))$ and $commit(T(v_i, age_i))$ is at most $3D$. Moreover, since the distance between any two nodes in a minimum spanning tree may increase by at most $O(D)$ factor compared to the shortest path distance between them, the communication cost due to $objAccess(*)$ message traversal may also increase by at most $O(D)$ factor compared to that in optimal case. That means, if c be the communication cost due to $objAccess(*)$ message traversal in optimal algorithm, then the total communication cost for transaction $T(v_i, age_i)$ becomes at most $(c \cdot D + 3D)$. Let $C_{i,OPT}$ and $C_{i,ALG}$ be the communication costs for executing transaction $T(v_i, age_i)$ in optimal and Dyn, respectively, then,

$$C_{i,ALG} = C_{i,OPT} \cdot O(D).$$

□

Combining Lemmas 3–6, we have,

Theorem 8. Dyn *is $O(D)$-competitive in both execution time and communication cost.*

Proof. Dyn executes in two phases, Phase 1 and Phase 2, sequentially. Phase 1 finishes in $O(D)$ time steps. In Phase 2, each transaction in $\mathcal{T}$ spends $O(D)$ time

steps to execute and commit. So, for all n transactions in $\mathcal{T}$, it takes $O(n \cdot D)$ time steps to execute and commit. In total, both Phase 1 and Phase 2 of Dyn end in $O(D) + O(n \cdot D) = O(n \cdot D)$ time steps. Since, transactions need to follow the age order to commit, any optimal algorithm requires at least $O(n)$ time steps to execute and commit. Hence, Dyn is $O(D)$-competitive in execution time. The same analysis works to show $O(D)$-competitive in communication cost. □

7. Evaluation

We have implemented all four proposed algorithms (OffExec, OffComm, Part-Dyn, and Dyn) and evaluated them using a set of micro and complex STAMP benchmarks. We compared the results against that of the data-flow model in [24]. The experiments were performed on an Intel Core i7-7700K processor with 32 GB RAM. We wrote our own discrete-event simulator representing a distributed network. Both micro and complex (STAMP) benchmarks donot have already defined transaction priorities (i.e., ages). We defined our own parameter setting in the implementation to assign the age of the transactions. To be specific, we assigned a random unique number between 0 to the total number of transactions for each transaction in each benchmark and the transaction with lower age number has higher priority. The experiments were run preserving the dependency order of the transactions. That means, the conflicting (i.e., dependent) transactions were executed in the increasing order of age whereas non-conflicting (i.e., non-dependent) transactions were executed in parallel without waiting for lower aged transactions to commit. We simulated three different communication graphs, namely *random*, *small-world*, and *grid*.

To build a random graph, we used the Erdős-Rényi model [57] and generated random graphs of different sizes. Particularly, we used the $G(n, \rho)$ variant of the Erdős-Rényi model [57] where a graph G is constructed connecting nodes randomly such that each edge is included in G with probability $0 < \rho < 1$, independent from every other edge. The graphs we used in the experiments were generated by setting $\rho = 0.01$. In case where the model generates a disconnected graph, we made it connected by adding a disconnected node to the longest connected component and attached smaller connected components to longer ones.

For small-world graphs, we used Watts–Strogatz model presented in [58] to build graphs where most nodes are not connected by an edge but they can still be reached in a few hops through other neighboring nodes. We used $p = 0.03$ as the probability to rewire an existing edge. We followed the requirement $n \gg k \gg \ln(n) \gg 1$ to select k edges for n nodes. Specifically, we used 15 edges for 64 nodes, 17 for 128, 20 for 256, and 28 for 512, respectively. In case the generated graph was disconnected, we ran the model until a connected graph is built.

Graph type	Number of nodes	Diameter (Weighted)	Diameter (Unweighted)
random	16	22	3
	32	21	4
	64	55	9
	128	109	20
	256	69	12
	512	38	8
small-world	16	7	2
	32	9	3
	64	11	4
	128	14	4
	256	16	5
	512	13	5
grid	16	27	6
	36	42	10
	64	49	14
	144	79	22
	256	107	30
	529	148	44

Table 1: Graph sizes and their diameters for ordered scheduling experimentation

Grid graphs are two-dimensional grids where corner nodes are connected to two neighbors each, the remaining boundary nodes are connected to three neighbors each, and inner nodes are connected to four neighbors each.

The graphs and their respective diameters are shown in Table 1. For random and small-world graphs, total number of nodes varied from 16 to 512, and for grid graphs, from 16 to 529. The total number of shared objects, transactions, and the transaction sizes vary based on specific applications in each benchmark. The results presented are the average of 10 runs. Figures 3-8 contain results for three different graphs we considered. Top row in each figure is for random graphs, middle row is for small-world graphs, and finally the third row is for grid graphs.

In the experiments, execution time is measured as the number of time steps. Communication cost is measured as the total distance traversed by the transactions to access objects and send commit notifications in the respective communication graphs. Latency is defined as the distance between two nodes in the network. That means, if two nodes v_1 and v_2 are directly connected by an edge between them, we assume that it takes $\mathsf{dist}(v_1, v_2)$ time step to reach from v_1 to v_2 (or vice-versa) and the communication cost to send a message between them is also $\mathsf{dist}(v_1, v_2)$.

Results on micro-benchmarks: We experimented the algorithms against three micro-benchmarks bank, linked list, and skip list. Figures 3–5 show their results

in random, small-world and grid graphs, respectively.

Results on STAMP benchmarks: We experimented the algorithms against *intruder*, *genome*, and *vacation* from STAMP [59] benchmarks. Figures 6–8 show their results in random, small-world and grid graphs, respectively.

7.1. Results Discussion

For all graph topologies, OffExec has the lowest execution time (optimal) in all benchmarks. The execution time for OffComm is higher than OffExec. Similarly, in all the benchmarks, OffComm has the minimum communication cost, which is within a factor of 2 from optimal. The experimental results also show that the execution time of PartDyn is always within $O(\log^2 n)$ factor compared to OffExec. Moreover, the execution time in Dyn is always within $O(D)$ factor. The communication cost results follow the same pattern and are substantially better than the theoretical bounds of PartDyn and Dyn. We can also see that Dyn has less execution time and less communication cost than PartDyn in all the benchmarks. This is because of $D < \log^2 n$ in the experiment.

7.2. Comparison between Data-flow and Control-flow Models

We compared the results obtained for OrdS in the data-flow model in [24] with the results for the control-flow model obtained here. In the data-flow model, for the offline setting with complete knowledge of transactions, their priorities, and the shared objects they need, [24] presents an offline algorithm (Off-Opt) that is optimal in terms of both execution time and communication cost. However, for the same setting in the control-flow model, we have two different algorithms, OffExec and OffComm, the first is optimal in execution time and the second is optimal in communication cost. We have shown that, in the control-flow model, it is impossible to have an algorithm that achieves simultaneously optimal execution time and communication cost (Theorem 1).

In the offline setting, with partial knowledge (i.e., transactions and their priorities are known beforehand, but the shared objects and their locations are not known until runtime), we have presented $O(\log^2 n)$-competitive algorithms for both execution time and communication cost. In the dynamic setting where transactions arrive arbitrarily over time, we have presented $O(D)$-competitive algorithms for both metrics.

In spite of having the same competitive bounds, the execution time (and communication cost) incurred in the data-flow [24] and control-flow models are different. The execution time achieved by a transaction in Off-Opt while running in the data-flow model is different than the execution time for the same transaction in OffExec while running in the control-flow model. This applies to both execution time and

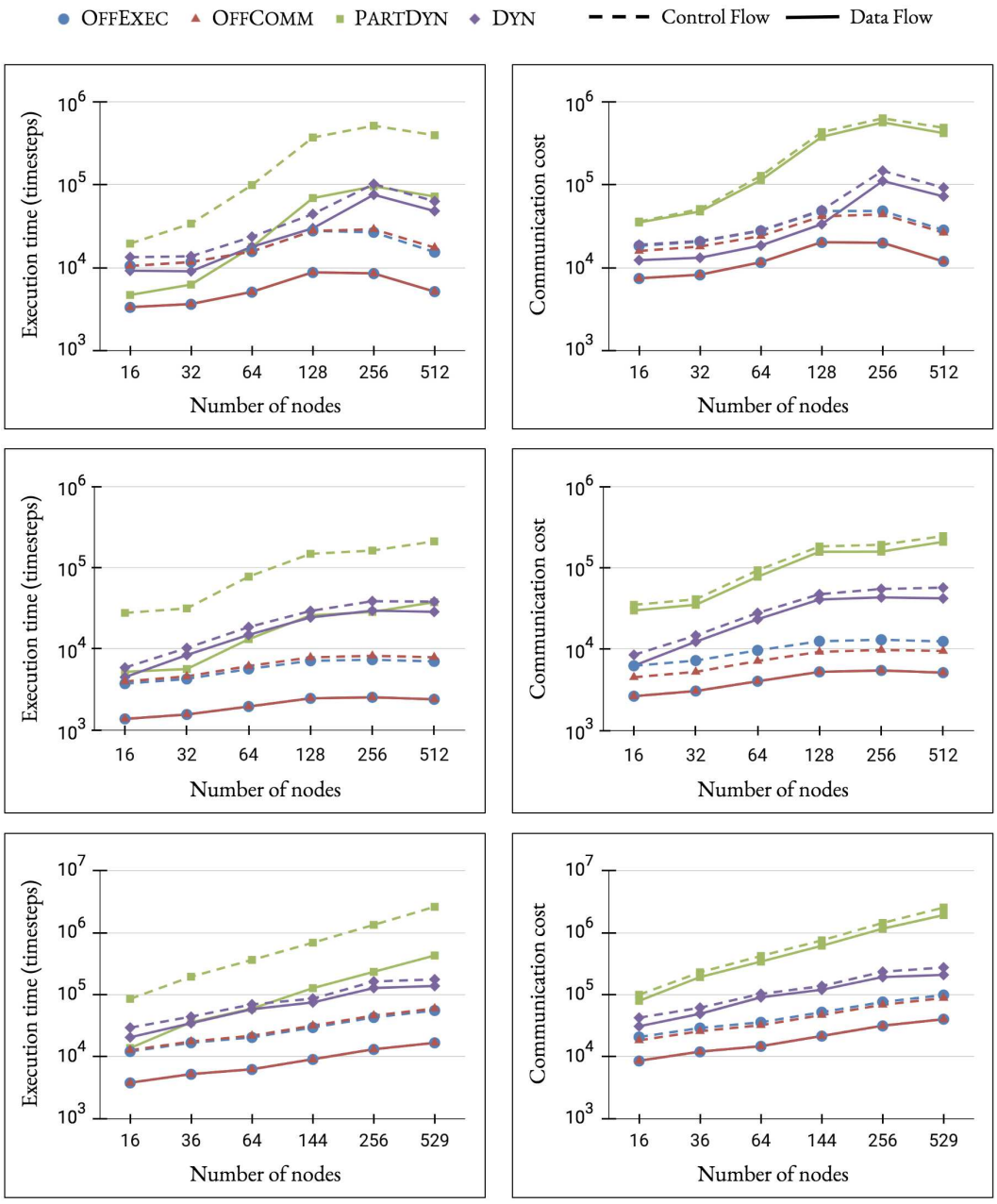

Figure 3: Execution time and communication cost (log scale) in bank micro-benchmark on random, the top row, small-world, the middle row, and grid graphs, the bottom row, respectively.

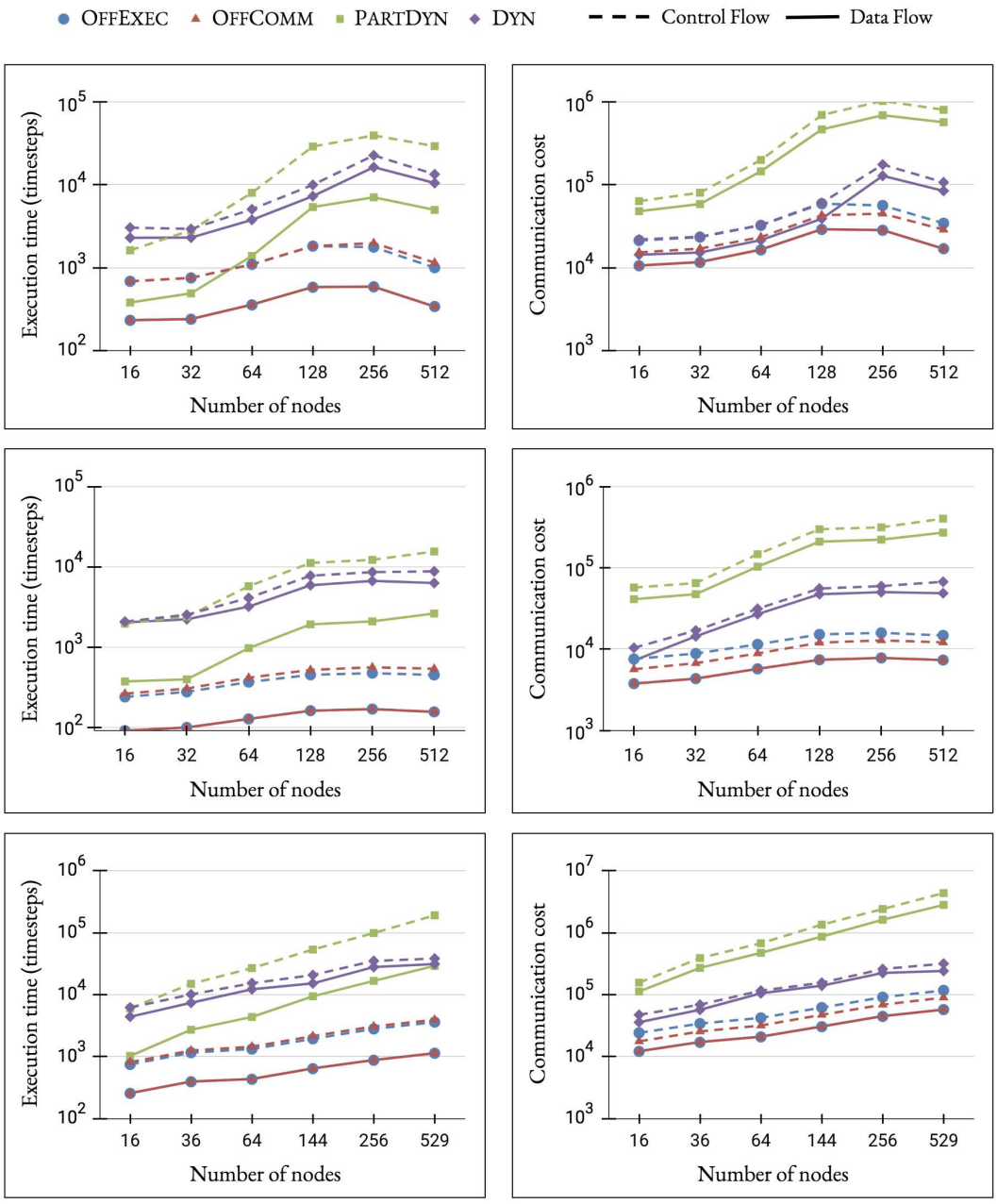

Figure 4: Execution time and communication cost (log scale) in linked-list micro-benchmark on random, the top row, small-world, the middle row, and grid graphs, the bottom row, respectively.

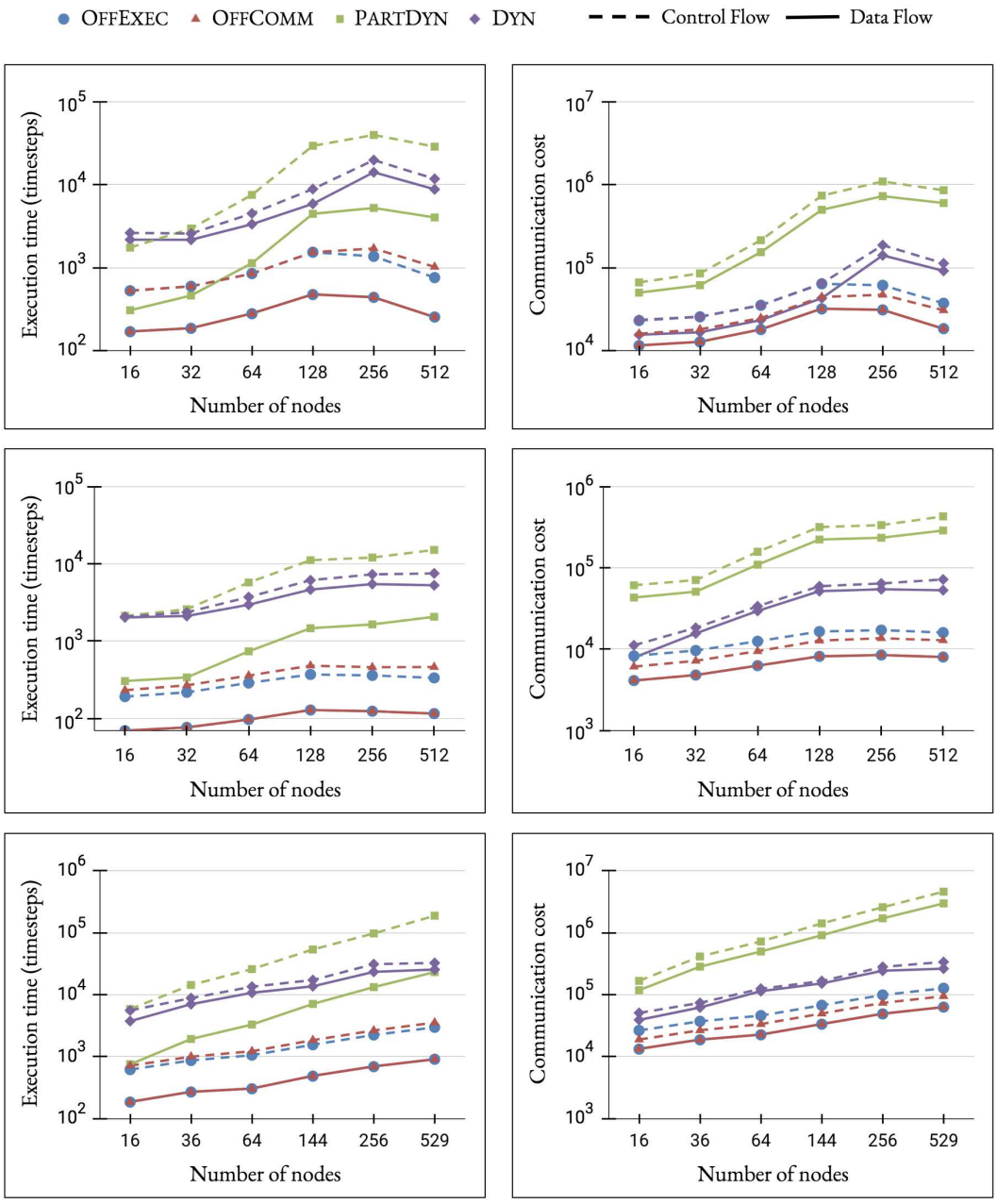

Figure 5: Execution time and communication cost (log scale) in skip-list micro-benchmark on random, the top row, small-world, the middle row, and grid graphs, the bottom row, respectively.

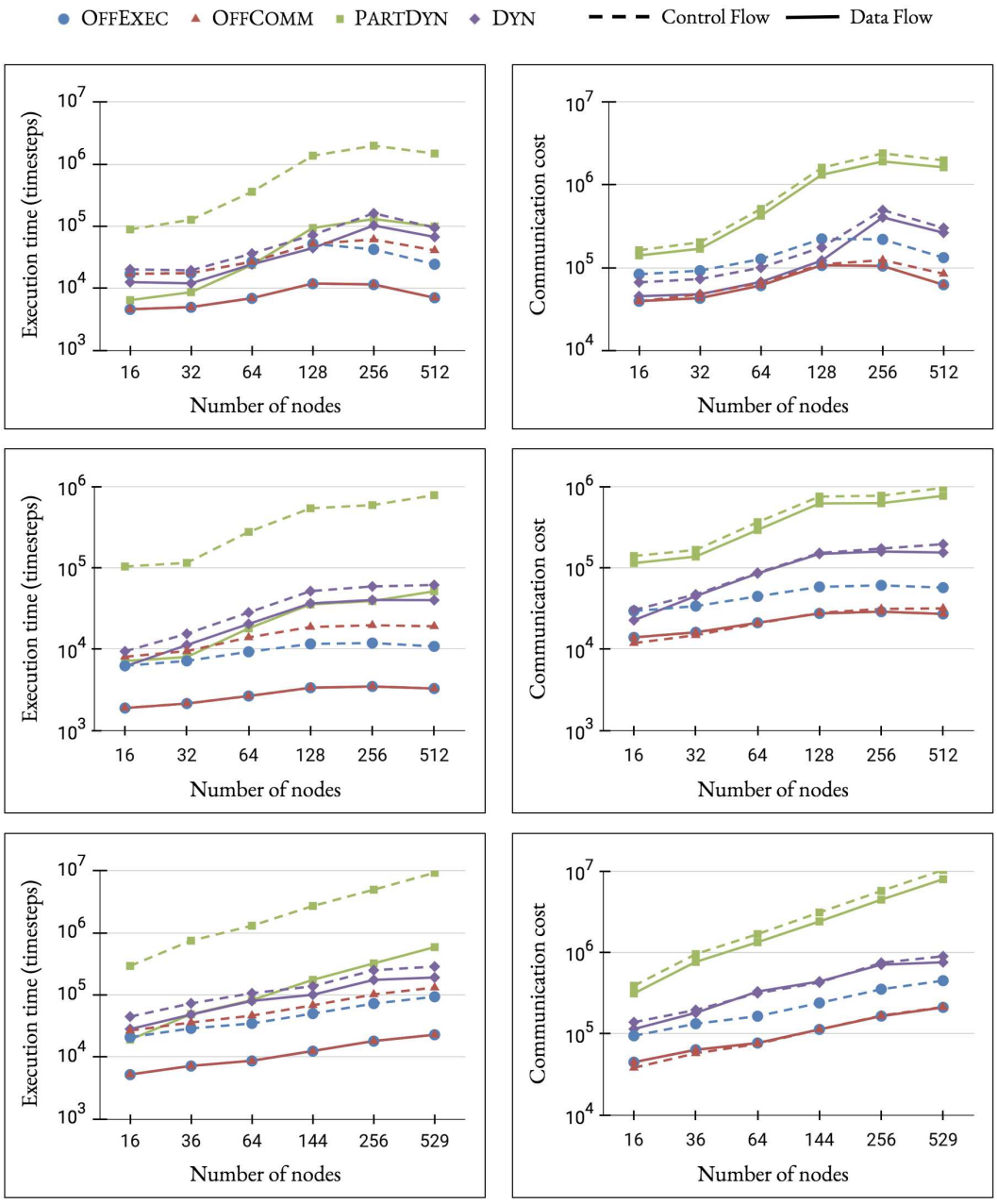

Figure 6: Execution time and communication cost (log scale) in genome benchmark on random, the top row, small-world, the middle row, and grid graphs, the bottom row, respectively.

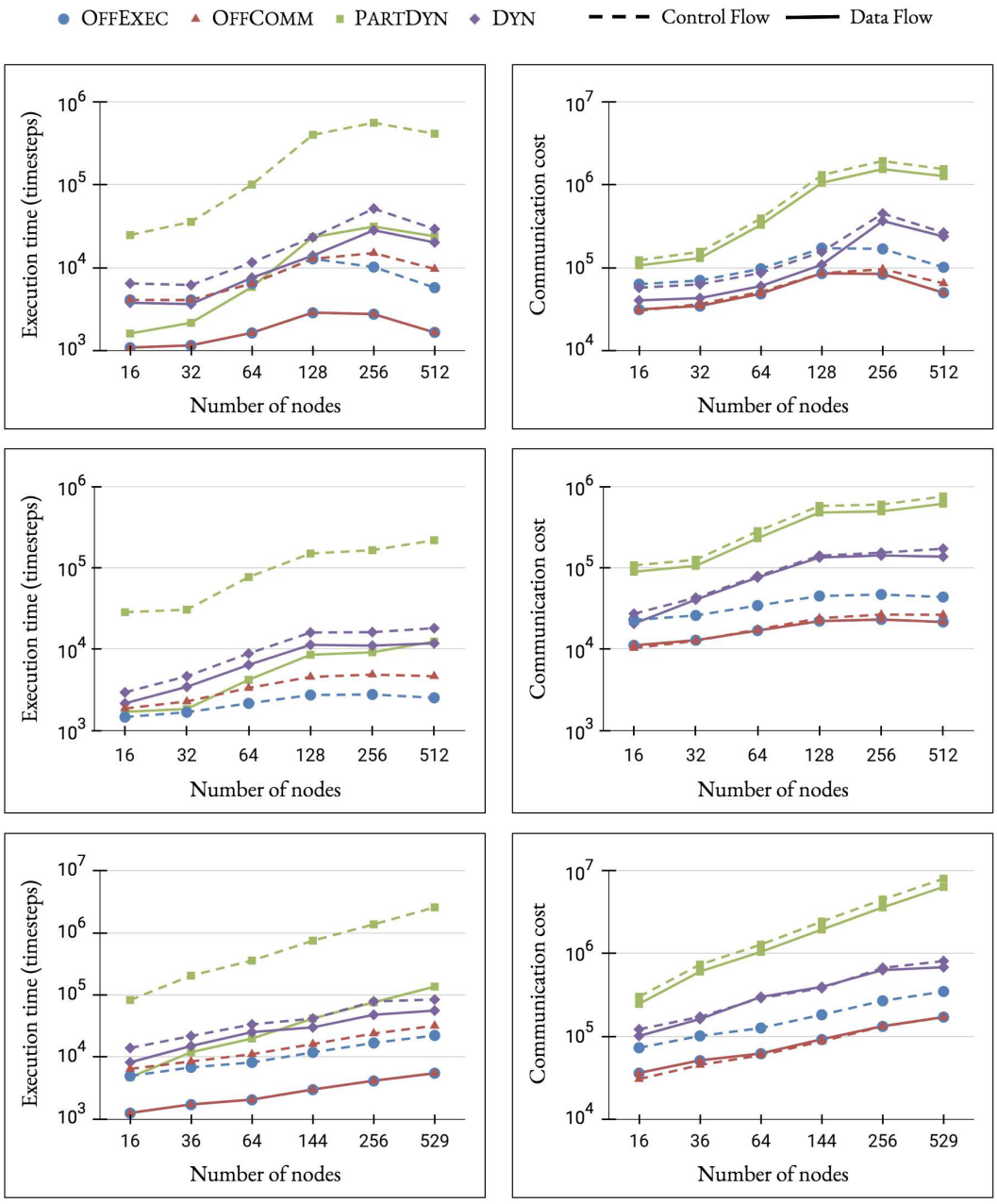

Figure 7: Execution time and communication cost (log scale) in intruder benchmark on random, the top row, small-world, the middle row, and grid graphs, the bottom row, respectively.

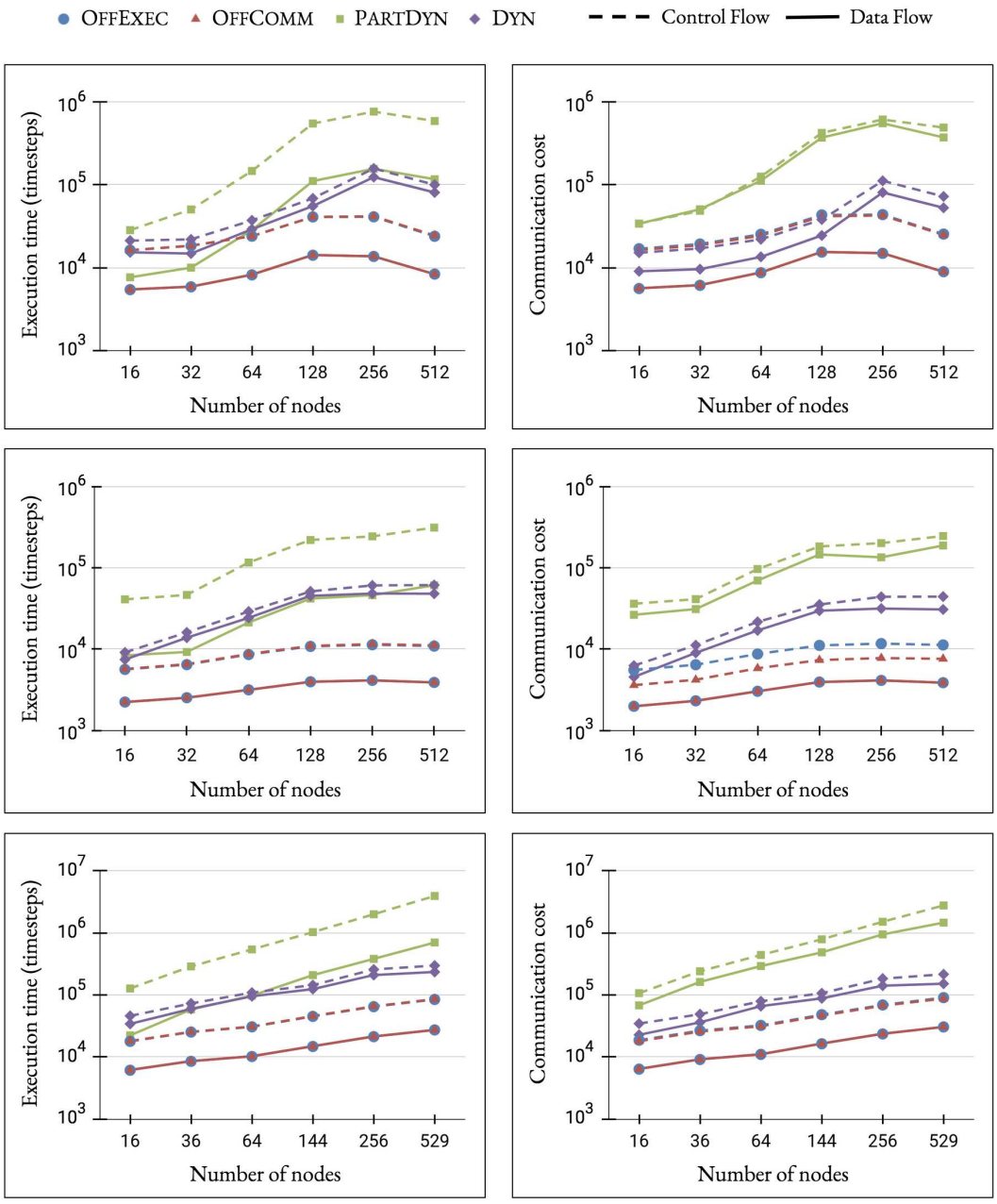

Figure 8: Execution time and communication cost (log scale) in vacation benchmark on random, the top row, small-world, the middle row, and grid graphs, the bottom row, respectively.

communication cost in all three settings. Thus, it is interesting to see how the two metrics differ in the two models with the experimental results.

Figures 3–8 show the comparison of the data-flow and control-flow models in terms of execution time and communication cost in random, small-world, and grid graphs, respectively. They provide the comparison results for offline, dynamic, and partial dynamic algorithms in both models. The results show that the control-flow model incurs more execution time as well as communication cost than the data-flow model. To be specific, the control-flow model has as much as 2× more execution time and communication cost than the data-flow model. In the data-flow model, when a transaction needs to access an object, the object directly moves to the node where the transaction is located. But, in the control-flow model, the transaction sends an object access request to the owner node of the object and when the access request is successful, a *grant* message is sent to the transaction. That means, in the data-flow model, there is a one-way traversal to access an object whereas, in the control-flow model, there is a two-way traversal to access each object. Thus, the control-flow model has as much as 2× more execution time and communication cost than the data-flow model.

In general, control-flow achieves worse performance than data-flow. But this is not always the case. For scheduling lightweight transactions (with minimum read-write sets), data-flow model seems less expensive. But, if the application is data-intensive then the movements of data become costly, so the control-flow model will be more efficient for achieving lower communication cost. This can be seen in the results for the benchmarks *genome* and *intruder* (Figures 6 and 7) where we can see data-flow incurs higher communication cost. This is due to transactions in these benchmarks requiring a lot of read-write sets. Hence, the selection of the execution model depends on the requirement and the nature of the actual application.

8. Concluding Remarks

In this paper, we have studied the ordered scheduling problem of committing transactions according to their predefined priorities in the control-flow distributed transactional memory, minimizing execution time and communication cost. The control-flow model is important because in many applications, the movement of data is costly due to its size and security purposes. We have provided a range of algorithms considering this problem in the offline and dynamic settings. Our results are (i) optimal in the offline setting with complete knowledge to (ii) poly-log competitive in the offline setting with partial knowledge to (iii) diameter competitive in the dynamic online setting. We also presented the comparative results for the algorithms in the data-flow and control-flow models. As a future work, it will be

interesting to deploy the algorithms in real distributed system(s) and measure the wall clock results.

References

[1] P. Poudel, S. Rai, S. Guragain, G. Sharma, Ordered scheduling in control-flow distributed transactional memory, in: A. R. Molla, G. Sharma, P. Kumar, S. Rawat (Eds.), Distributed Computing and Intelligent Technology - 19th International Conference, ICDCIT 2023, Bhubaneswar, India, January 18-22, 2023, Proceedings, Vol. 13776 of Lecture Notes in Computer Science, Springer, 2023, pp. 67–83.

[2] M. Herlihy, J. E. B. Moss, Transactional memory: Architectural support for lock-free data structures, in: A. J. Smith (Ed.), Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, USA, May 1993, ACM, 1993, pp. 289–300.

[3] N. Shavit, D. Touitou, Software transactional memory, Distributed Comput. 10 (2) (1997) 99–116.

[4] Intel, http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell (2012).

[5] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G. Chiu, P. Boyle, N. Chist, C. Kim, The ibm blue gene/q compute chip, IEEE Micro 32 (2) (2012) 48–60.

[6] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, H. Tomari, Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and POWER8, in: ISCA, 2015, pp. 144–157.

[7] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, H. Q. Le, Robust architectural support for transactional memory in the power architecture, in: A. Mendelson (Ed.), The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013, ACM, 2013, pp. 225–236.

[8] V. Gramoli, R. Guerraoui, V. Trigonakis, Tm^2c: a software transactional memory for many-cores, Distributed Comput. 31 (5) (2018) 367–388.

[9] M. Mohamedin, S. Peluso, M. J. Kishi, A. Hassan, R. Palmieri, Nemo: Numa-aware concurrency control for scalable transactional memory, in: Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018, ACM, 2018, pp. 38:1–38:10.

[10] R. L. B. Jr., V. S. Adve, B. L. Chamberlain, Software transactional memory for large scale clusters, in: S. Chatterjee, M. L. Scott (Eds.), Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008, ACM, 2008, pp. 247–258.

[11] K. Manassiev, M. Mihailescu, C. Amza, Exploiting distributed version concurrency in a transactional memory cluster, in: J. Torrellas, S. Chatterjee (Eds.), Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006, ACM, 2006, pp. 198–208.

[12] W. W. L. Fung, I. Singh, A. Brownsword, T. M. Aamodt, Hardware transactional memory for GPU architectures, in: C. Galuzzi, L. Carro, A. Moshovos, M. Prvulovic (Eds.), 44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011, ACM, 2011, pp. 296–307.

[13] A. Turcu, B. Ravindran, R. Palmieri, Hyflow2: a high performance distributed transactional memory framework in scala, in: M. Plümicke, W. Binder (Eds.), Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013, ACM, 2013, pp. 79–88.

[14] M. M. Saad, B. Ravindran, Snake: Control flow distributed software transactional memory, in: X. Défago, F. Petit, V. Villain (Eds.), Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings, Vol. 6976 of Lecture Notes in Computer Science, Springer, 2011, pp. 238–252.

[15] M. M. Saad, M. J. Kishi, S. Jing, S. Hans, R. Palmieri, Processing transactions in a predefined order, in: J. K. Hollingsworth, I. Keidar (Eds.), Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019, ACM, 2019, pp. 120–132.

[16] M. M. Saad, R. Palmieri, B. Ravindran, Lerna: Parallelizing dependent loops using speculation, in: Proceedings of the 11th ACM International Systems and Storage Conference, SYSTOR 2018, HAIFA, Israel, June 04-07, 2018, ACM, 2018, pp. 37–48.

[17] S. Hirve, R. Palmieri, B. Ravindran, Archie: a speculative replicated transactional system, in: L. Réveillère, L. Cherkasova, F. Taïani (Eds.), Proceedings of the 15th International Middleware Conference, Bordeaux, France, December 8-12, 2014, ACM, 2014, pp. 265–276.

[18] L. Lamport, The part-time parliament, ACM Trans. Comput. Syst. 16 (2) (1998) 133–169.

[19] J. Gil Herrera, J. F. Botero, Resource allocation in nfv: A comprehensive survey, IEEE Trans. on Netw. and Serv. Manag. 13 (3) (2016) 518–532. doi:10.1109/TNSM.2016.2598420.
URL https://doi.org/10.1109/TNSM.2016.2598420

[20] A. M. Alwakeel, A. K. Alnaim, E. B. Fernandez, A survey of network function virtualization security, in: SoutheastCon 2018, 2018, pp. 1–8. doi:10.1109/SECON.2018.8479121.

[21] M. Villari, M. Fazio, S. Dustdar, O. Rana, R. Ranjan, Osmotic computing: A new paradigm for edge/cloud integration, IEEE Cloud Computing 3 (6) (2016) 76–83. doi:10.1109/MCC.2016.124.

[22] S. K. Roy, R. Devaraj, A. Sarkar, D. Senapati, Slaqa: Quality-level aware scheduling of task graphs on heterogeneous distributed systems, ACM Trans. Embed. Comput. Syst. 20 (5). doi:10.1145/3462776.
URL https://doi.org/10.1145/3462776

[23] M. A. Gonzalez-Mesa, E. Gutiérrez, E. L. Zapata, O. G. Plata, Effective transactional memory execution management for improved concurrency, ACM Trans. Archit. Code Optim. 11 (3) (2014) 24:1–24:27.

[24] P. Poudel, S. Rai, G. Sharma, Processing distributed transactions in a predefined order, in: ICDCN '21: International Conference on Distributed Computing and Networking, Virtual Event, Nara, Japan, January 5-8, 2021, ACM, 2021, pp. 215–224.

[25] C. Busch, M. Herlihy, M. Popovic, G. Sharma, Time-communication impossibility results for distributed transactional memory, Distributed Comput. 31 (6) (2018) 471–487.

[26] C. Busch, M. Herlihy, M. Popovic, G. Sharma, Dynamic scheduling in distributed transactional memory, in: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020, IEEE, 2020, pp. 874–883.

[27] M. Herlihy, Y. Sun, Distributed transactional memory for metric-space networks, Distributed Comput. 20 (3) (2007) 195–208.

[28] G. Sharma, C. Busch, Distributed transactional memory for general networks, Distributed Comput. 27 (5) (2014) 329–362.

[29] H. Attiya, L. Epstein, H. Shachnai, T. Tamir, Transactional contention management as a non-clairvoyant scheduling problem, Algorithmica 57 (1) (2010) 44–61.

[30] A. Dragojevic, R. Guerraoui, A. V. Singh, V. Singh, Preventing versus curing: avoiding conflicts in transactional memories, in: S. Tirthapura, L. Alvisi (Eds.), Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009, ACM, 2009, pp. 7–16.

[31] R. Guerraoui, M. Herlihy, B. Pochon, Toward a theory of transactional contention managers, in: M. K. Aguilera, J. Aspnes (Eds.), Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, PODC 2005, Las Vegas, NV, USA, July 17-20, 2005, ACM, 2005, pp. 258–264.

[32] G. Sharma, C. Busch, A competitive analysis for balanced transactional memory workloads, Algorithmica 63 (1-2) (2012) 296–322.

[33] G. Sharma, C. Busch, Window-based greedy contention management for transactional memory: theory and practice, Distributed Comput. 25 (3) (2012) 225–248.

[34] R. M. Yoo, H. S. Lee, Adaptive transaction scheduling for transactional memory systems, in: F. M. auf der Heide, N. Shavit (Eds.), SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008, ACM, 2008, pp. 169–178.

[35] H. Attiya, V. Gramoli, A. Milani, Directory protocols for distributed transactional memory, in: R. Guerraoui, P. Romano (Eds.), Transactional Memory. Foundations, Algorithms, Tools, and Applications - COST Action Euro-TM IC1001, Vol. 8913 of Lecture Notes in Computer Science, Springer, 2015, pp. 367–391.

[36] C. Busch, M. Herlihy, M. Popovic, G. Sharma, Impossibility results for distributed transactional memory, in: C. Georgiou, P. G. Spirakis (Eds.), Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015, ACM, 2015, pp. 207–215.

[37] C. Busch, M. Herlihy, M. Popovic, G. Sharma, Fast scheduling in distributed transactional memory, in: C. Scheideler, M. T. Hajiaghayi (Eds.), Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2017, Washington DC, USA, July 24-26, 2017, ACM, 2017, pp. 173–182.

[38] G. Sharma, C. Busch, A load balanced directory for distributed shared memory objects, J. Parallel Distributed Comput. 78 (2015) 6–24.

[39] B. Zhang, B. Ravindran, R. Palmieri, Distributed transactional contention management as the traveling salesman problem, in: M. M. Halldórsson (Ed.), Structural Information and Communication Complexity - 21st International Colloquium, SIROCCO 2014, Takayama, Japan, July 23-25, 2014. Proceedings, Vol. 8576 of Lecture Notes in Computer Science, Springer, 2014, pp. 54–67.

[40] J. Kim, B. Ravindran, On transactional scheduling in distributed transactional memory systems, in: S. Dolev, J. A. Cobb, M. J. Fischer, M. Yung (Eds.), Stabilization, Safety, and Security of Distributed Systems - 12th International Symposium, SSS 2010, New York, NY, USA, September 20-22, 2010. Proceedings, Vol. 6366 of Lecture Notes in Computer Science, Springer, 2010, pp. 347–361.

[41] P. Poudel, G. Sharma, Graphtm: An efficient framework for supporting transactional memory in a distributed environment, in: N. Mukherjee, S. V. Pemmaraju (Eds.), ICDCN 2020: 21st International Conference on Distributed Computing and Networking, Kolkata, India, January 4-7, 2020, ACM, 2020, pp. 11:1–11:10.

[42] D. Hendler, A. Naiman, S. Peluso, F. Quaglia, P. Romano, A. Suissa, Exploiting locality in lease-based replicated transactional memory via task migration, in: Y. Afek (Ed.), Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings, Vol. 8205 of Lecture Notes in Computer Science, Springer, 2013, pp. 121–133.

[43] R. Palmieri, S. Peluso, B. Ravindran, Transaction execution models in partially replicated transactional memory: The case for data-flow and control-flow, in: R. Guerraoui, P. Romano (Eds.), Transactional Memory. Foundations, Algorithms, Tools, and Applications - COST Action Euro-TM IC1001, Vol. 8913 of Lecture Notes in Computer Science, Springer, 2015, pp. 341–366.

[44] C. Busch, B. S. Chlebus, M. Herlihy, M. Popovic, P. Poudel, G. Sharma, Flexible scheduling of transactional memory on trees, in: S. Devismes, F. Petit, K. Altisen, G. A. D. Luna, A. F. Anta (Eds.), Stabilization, Safety, and Security of Distributed Systems - 24th International Symposium, SSS 2022, Clermont-Ferrand, France, November 15-17, 2022, Proceedings, Vol. 13751 of Lecture Notes in Computer Science, Springer, 2022, pp. 146–163.

[45] J. Kim, B. Ravindran, Scheduling transactions in replicated distributed software transactional memory, in: 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013, Delft, Netherlands, May 13-16, 2013, IEEE Computer Society, 2013, pp. 227–234.

[46] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, L. E. T. Rodrigues, When scalability meets consistency: Genuine multiversion update-serializable partial data replication, in: 2012 IEEE 32nd International Conference on Distributed Computing Systems, Macau, China, June 18-21, 2012, IEEE Computer Society, 2012, pp. 455–465.

[47] K. Arnold, R. Scheifler, J. Waldo, B. O'Sullivan, A. Wollrath, Jini Specification, Addison-Wesley Longman Publishing, 1999.

[48] G. Z. Gutin, A. P. Punnen, The traveling salesman problem, Discret. Optim. 3 (1) (2006) 1.

[49] D. P. Williamson, D. B. Shmoys, The Design of Approximation Algorithms, Cambridge University Press, 2011.

[50] N. Christofides, Worst-case analysis of a new heuristic for the travelling salesman problem, Oper. Res. Forum 3 (1).

[51] F. K. Hwang, On steiner minimal trees with rectilinear distance, SIAM Journal on Applied Mathematics 30 (1) (1976) 104–114.

[52] F. Hwang, D. Richards, P. Winter, The Steiner Tree Problem, ISSN, Elsevier Science, 1992.

[53] L. E. N. Gouveia, T. L. Magnanti, Network flow models for designing diameter-constrained minimum-spanning and steiner trees, Networks 41 (3) (2003) 159–173.

[54] T. Hiromitsu, M. Akira, An approximate solution for the steiner problem in graphs, MATH. JAP.; JPN; DA. 1980; VOL. 24; NO 6; PP. 573-577; BIBL. 9 REF.

[55] M. J. Demmer, M. Herlihy, The arrow distributed directory protocol, in: S. Kutten (Ed.), Distributed Computing, 12th International Symposium, DISC '98, Andros, Greece, September 24-26, 1998, Proceedings, Vol. 1499 of Lecture Notes in Computer Science, Springer, 1998, pp. 119–133.

[56] S. Rai, G. Sharma, C. Busch, M. Herlihy, Load balanced distributed directories, Inf. Comput. 285 (Part) (2022) 104700.

[57] P. Erdös, A. Rényi, On random graphs i, Publicationes Mathematicae Debrecen 6 (1959) 290.

[58] D. J. Watts, S. H. Strogatz, Collective dynamics of 'small-world' networks, Nature 393 (6684) (1998) 440–442.

[59] C. C. Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: stanford transactional applications for multi-processing, in: D. Christie, A. Lee, O. Mutlu, B. G. Zorn (Eds.), 4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008, IEEE Computer Society, 2008, pp. 35–46.