GraphTM: An Efficient Framework for Supporting Transactional Memory in a Distributed Environment

Pavan Poudel
Kent State University
Kent, Ohio
ppoudel@cs.kent.edu

Gokarna Sharma
Kent State University
Kent, Ohio
sharma@cs.kent.edu

ABSTRACT

In this paper, we present GraphTM, an efficient and scalable framework for processing transactions in a distributed environment. The distributed environment is modeled as a graph where each node of the graph is a processing node that issues transactions. The objects that transactions use to execute are also on the graph nodes (the initial placement may be arbitrary). The transactions execute on the nodes which issue them after collecting all the objects that they need following the data-flow model of computation. This collection is done by issuing the requests for the objects as soon as transaction starts and wait until all required objects for the transaction come to the requesting node. The challenge is on how to schedule the transactions so that two crucial performance metrics, namely (i) total execution time to commit all the transactions, and (ii) total communication cost involved in moving the objects to the requesting nodes, are minimized. We implemented GraphTM in Java and assessed its performance through 3 micro-benchmarks and 5 complex benchmarks from STAMP benchmark suite on 5 different network topologies, namely, clique, line, grid, cluster, and star, that make an underlying communication network for a representative set of distributed systems commonly used in practice. The results show the efficiency and scalability of our approach.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**; **Modeling and simulation**; **Simulation evaluation**; • **Theory of computation** → **Distributed computing models**; **Design and analysis of algorithms**; **Online algorithms**.

KEYWORDS

Distributed system; transactional memory; scheduling; execution time; communication cost; conflicts; waiting time

ACM Reference Format:
Pavan Poudel and Gokarna Sharma. 2020. GraphTM: An Efficient Framework for Supporting Transactional Memory in a Distributed Environment. In *21st International Conference on Distributed Computing and Networking (ICDCN 2020), January 4–7, 2020, Kolkata, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3369740.3369774

ICDCN 2020, January 4–7, 2020, Kolkata, India

ACM ISBN 978-1-4503-7751-5/20/01…$15.00
https://doi.org/10.1145/3369740.3369774

1 INTRODUCTION

Concurrent processes (threads) need to synchronize to avoid introducing inconsistencies while accessing shared data objects. Traditional synchronization mechanisms such as locks and barriers have well-known limitations and pitfalls, including deadlock, priority inversion, reliance on programmer conventions, and vulnerability to failure or delay. *Transactional memory* (TM) [14, 26] has emerged as an attractive alternative. Recently, several commercial processors support TM, for example, Intel's Haswell [17] and IBM's Blue Gene/Q [12], zEnterprise EC12 [23], and Power8 [4].

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.

The TM paradigm has been studied heavily in the past for shared memory multi-core systems where tightly-coupled processing cores share a single shared memory and the latency to perform a memory access is the same for all the processors. Having a single shared memory allows to focus on how to execute and commit transactions so that the total time to commit all the transactions is minimized, not worrying about how to minimize the memory access latency. However, with the recent multi-faceted advances in computing architectures, this tightly-coupled single shared-memory multi-core paradigm is shifting toward many-core to non-uniform memory access (NUMA) to cluster and more general distributed networked architectures, where the latency to perform a memory access varies depending on the processor in which the thread executes and the physical segment of memory that stores the requested memory location. Therefore, researchers shifted their focus recently on how to support TM in these architectures, incorporating latency into analysis and evaluation. Some of the proposals include TM^{2}C for many-core architectures [11], Nemo for NUMA architectures [22], the systems in [1, 20] for clusters [1, 20], the system in [10] for clusters of GPUs, and the Hyflow framework for more general distributed systems [21, 25, 27, 28].

Supporting the TM paradigm in many-core and NUMA architectures seems relatively easier compared to supporting it in cluster and more general distributed systems. There are three main reasons: (i) many-core and NUMA architectures have a relatively simple underlying interconnection network, namely network-on-chip, which interconnects all cores and carries the memory traffic; (ii) a single node typically contains one (or more) multi-core architecture (which is called a NUMA zone) and the number of such nodes is very few and connected to all others so that the latency to perform a memory access to a remote node is within a constant factor of the

latency to perform a memory access in the local node (independent of the network size); and (iii) many-core and NUMA architectures might be equipped with an underlying cache-coherence protocol which can be exploited to support TM.

The aforementioned advantages of many-core and NUMA architectures vanish when looking into more general cluster and distributed systems. For example, the underlying interconnection network might be arbitrary and hence the latency to perform a memory access to a remote node might be significantly high, no high-speed network-on-chip connection, and no underlying cache-coherence support. This necessitates the communication between the nodes through message passing since shared-memory based synchronization is not possible. Furthermore, cache-coherence for objects has to be supported through the implementation itself. Despite many difficulties to deal with, this setting is strong enough to model any complex processing system and it is a natural direction to explore whether TM can be supported efficiently and scalably in these environments.

The goal in this paper is to study whether TM can be supported efficiently and scalably in cluster and distributed systems. Cluster and distributed systems are widely available these-a-days and there is a growing interest in implementing TM on them [1, 10, 20, 21, 25, 27, 28]. However, the previous studies [1, 10, 18, 20, 21, 25, 27, 28, 30] lack on one or more aspects. One major aspect that they lack is efficiency guarantee, in other words, how the total execution time and communication cost to execute transactions compared to the best possible execution time and communication cost, knowing everything about the application workload? The framework we present here provides these efficiency guarantees. Moreover, our framework allows to make better decisions on processing transactions taking into account the knowledge of the topology used by the distributed system considered; previous implementations lack this and hence they may not be able to optimize processing transactions with respect to the underlying communication topology. Furthermore, most of the previous studies, e.g., [1, 18, 20], replicate shared-memory multi-core TM implementations to different nodes and synchronize the execution through global lock, serialization lease, or commit-time broadcasting, which do not scale well with the size of the network, Moreover, they hamper either or both performance metrics.

Contributions. We present a framework, called GraphTM, that supports TM paradigm in cluster and distributed systems efficiently and scalably removing many limitations of the previous studies (as mentioned in the previous paragraph). For achieving efficiency and scalability, GraphTM is designed to be *polymorphic* – the underlying mechanism of transaction processing is switched depending on the underlying network topology used by the distributed system. This is crucial since the recent previous studies [2, 3] showed that no "universal" scheduling strategy works best under different topologies. We design, develop, and implement GraphTM in Java with necessary underlying algorithms to schedule transactions in both offline and online settings. We present evaluation results running our GraphTM implementation against 3 well-known micro-benchmarks, namely *bank*, *linked list*, and *skip list*, and 5 complex benchmarks, namely *bayes*, *genome*, *intruder*, *kmeans*, and *vacation*, from the STAMP benchmark suite, in 5 different communication topologies, clique, line, grid, cluster, and star. The motivation behind choosing these networks for experimentation is that they make an underlying communication network for a representative set of distributed systems commonly used in practice [5, 19]. Currently, GraphTM simulates the execution of distributed transactions in a synchronous setting and presents the results on the maximum time steps for execution time and total number of hops for the communication cost. The framework and results presented here are crucial for design decisions while implementing execution framework in a real cluster environment. In summary, we have the following four contributions in this paper:

- We provide an implementation of efficient and scalable execution framework, called GraphTM, for supporting TM paradigm in a distributed environment.
- We present offline and online algorithms for scheduling transactions using GraphTM. These algorithms follow and extend provably-efficient scheduling studied in [2] for different special graph topologies.
- We extensively evaluate the scheduling algorithms designed for GraphTM using a set of micro-benchmarks and STAMP benchmarks under five different network topologies and report the results on execution time, communication cost, and waiting time under various settings.
- We design an approach to simulate the communication overhead in processing transactions in a distributed setting by running the benchmarks in a multicore environment. We then report the communication overhead results obtained for different benchmarks.

Paper Organization. We discuss related work in detail in Section 2. We discuss model in Section 3 and the design overview of GraphTM in Section 4. The implementation details of GraphTM are provided in Section 5 and evaluated against different benchmarks and topologies in Section 6. We finally conclude in Section 7 with a short discussion. Some figures, algorithms and details are omitted due to space constraints.

2 RELATED WORK

The TM paradigm has been studied for many-core architectures, such as Intel, AMD, and Tilera, in [11]. The authors proposed a system, called TM^{2}C, that exploits shared memory and distributed aspects to provide scalability in TM support. The TM paradigm for NUMA architectures has recently been studied in [22]. The authors proposed a TM system, called Nemo, that focuses on exploiting the properties of the current NUMA designs so that scalable parallelism can be obtained. They observed that increasing parallel execution of threads in different NUMA zones is the key to scalability.

For general distributed systems, quite a few systems have been developed in the literature. Hyflow [25] and Hyflow2 [27] frameworks written in Java support processing transactions in a distributed environment. HyflowCPP [21] extends Hyflow [25] and Hyflow2 [27] frameworks to non-VM-language CPP. Atomic RMI 2 [28] extends the popular java RMI system with support for distributed transactions. Atomic RMI 2 is an extension of Atomic RMI which extends Java RMI system to run transactions in many JVMs located on different network nodes. Zhang and Ravindran [29] provided a distributed dependency-aware (DDA) model for distributed

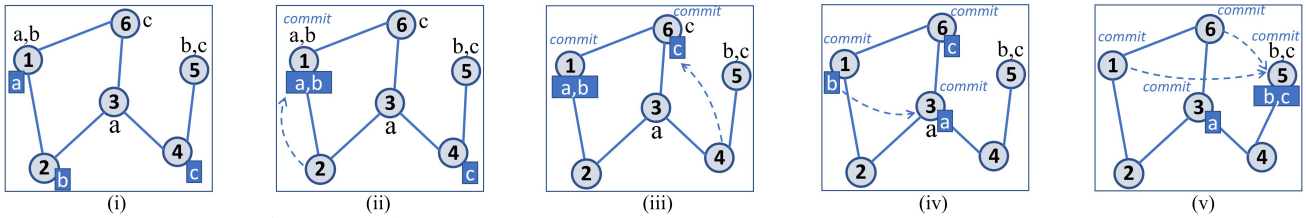

Figure 1: An illustration of (distributed) transaction execution in a distributed system of 6 nodes with three shared objects a, b, c (shown inside box). (i) shows the transactions at different nodes requesting objects a, b, and/or c. (ii)–(v) show how transactions commit and the objects move from one transaction to another that request them.

TM that manages dependencies between conflicting and uncommitted transactions so that they can commit safely. But the model has an inherent tradeoff between the concurrency and communication cost. All these systems lack polymorphism and (provable) efficiency guarantees on time and communication cost that GraphTM has.

Several other distributed TM systems were also proposed, e.g., [1, 6, 16, 18]. However, most of them replicate a non-distributed TM on many nodes and guarantee consistency of replicas. This model is different from the model we use, and has different applications (high-reliability systems rather than for example distributed data stores). Other systems extended non-distributed TMs with a communication layer, e.g., DiSTM [13] extends D2STM [4] with distributed coherence protocols. Moreover, these studies either use global lock, serialization lease, or commit-time broadcasting technique which may not scale well with the size of the network. Moreover, they do not provide formal analysis of either the execution time or the communication cost.

Furthermore, there are distributed TM proposals that employ replication and multi-versioning [20, 24]. In replicated TMs, multiple copies are available for each shared object, whereas multiple versions of each object are available in multi-versioning TMs [20]. This line of work is different than ours.

3 MODEL AND PRELIMINARIES

We consider a distributed system with a set of nodes $\mathcal{N} = \{N_1, N_2, \dots\}$ of a communication graph G that communicate via the message-passing links. We assume that the communication links are of unit weights and take one time step to traverse. Let $\mathcal{T} = \{T_1, T_2, \dots\}$ be the set of transactions and $O = \{O_1, O_2, \dots\}$ be the set of objects accessed by the transactions. We have an offline setting if the transactions in the set $\mathcal{T}$ are known before execution starts. Otherwise, it is an online setting. In both the settings, it is assumed that the objects in O accessed by a transaction are known as soon as transaction arrives, so that they can be requested (if not presently at the local node). We consider the *data-flow* model (see Fig. 1) of distributed execution where transactions are immobile, but objects move from one node to another [15]. Each object O_i is represented with a unique identifier, *obj_id*. Each object has an owner node, denoted by *node_id*. Although, an object may have cached copy at different nodes, it has a single owner node and the owner node can be changed during the movement of the object. A change in ownership occurs upon the successful commit of a transaction which modified the object. Similarly, each transaction has a unique identifier (*tx_id*), read set (*rset*), and write set (*wset*). A transaction contains a sequence of operations, each of which is a read or write operation on an object. An execution of a transaction ends by either a commit (success) or an abort (failure). A transaction can have five possible states: *idle, waiting, running, committed,* and *aborted*. Any aborted transaction is later retried using a new identifier until eventually it commits.

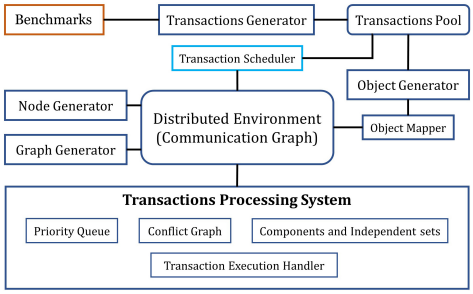

Figure 2: GraphTM architecture

The communication model is assumed to be synchronous where time is divided into discrete steps such that in a time step, a node can receive the messages, process, and send the messages to the adjacent nodes [2, 3]. The execution time is thus measured as the number of time steps taken to execute all the transactions. The communication cost is measured as the total distance (the number of hops) the objects traverse while executing transactions. We also assume that the nodes and links are not faulty and the links deliver messages in FIFO order.

4 GRAPHTM DESIGN OVERVIEW

4.1 GraphTM Architecture

The system architecture of GraphTM is shown in Fig. 2. In GraphTM, we simulate a distributed system environment by using graph networks where a node of a graph represents a networked component (processors) and the edge between two nodes represents the communication path between those networked components. We simulate the transaction processing environment in the distributed system by defining a set of transactions, a set of objects accessed by those transactions, the owner node of each object and the movement of those objects between the nodes of the graph. Each transaction consists of read set and write set of objects residing at different nodes of the graph. The transaction requires those objects to move to the node at which it is issued during the execution phase. We generate nodes, graphs, objects and transactions scalably in GraphTM.

Nodes and Graph Generation. The first task of GraphTM is to generate a graph with a desired number of nodes. We have considered five different types of graph, *clique, line, grid, cluster,* and *star* (GraphTM is general enough to be applied beyond these topologies as well). Each graph is represented by the *graph structure*

(G_{type}), *total nodes* (N_{nodes}), *total edges* (N_{edges}), and *list of nodes* (L_{nodes}). To have a graph, nodes are generated first. Each node has unique *node_id* and is represented by a point (x, y) in 2-dimensional plane. Each node also consists of a list of objects that are contained by it and a list of neighboring nodes. Total number of nodes (N_{nodes}) are created for each type of graph separately and added to the list L_{nodes} of the graph G_{type}. Total number of edges N_{edges} for the graph are calculated thereafter.

Objects Generation. After generating the graph, the framework generates M objects in the system. Each generated object is represented by a unique *obj_id* and the size of the object is defined by *obj_size*. Each object has a unique owner node defined by *obj_node* at which the object resides.

Transaction Generation. Now, the important part of the framework is transaction generation. A transaction is represented by a unique id (*tx_id*), read/write set size (*rws_size*), update rate (*update_rate*), state (*state*), and lists of objects in read set and write set ($rset, wset$). Read/Write set size (*rws_size*) for a transaction defines the total size of objects in read set and write set of a transaction. Considering the size of an object as 1, the *rws_size* for each transaction can be as much as total number of objects, i.e., $0 \leq rws\_size \leq M$. The *update_rate* is used to compute the read set size (*rs_size*) and write set size (*ws_size*) of a transaction T_i where $ws\_size(T_i) = rws\_size(T_i) \times update\_rate(T_i)$ and $rs\_size(T_i) = rws\_size(T_i) - ws\_size(T_i)$. The *update_rate* for a transaction T_i lies between 0 and 1 i.e. $0 \leq update\_rate(T_i) \leq 1$. The read set, $rset(T_i)$ and write set, $wset(T_i)$ of each transaction T_i hold the following two properties:

- $rset(T_i)$ contains exactly $rs\_size(T_i)$ number of objects and $wset(T_i)$ contains exactly $ws\_size(T_i)$ number of objects.
- $rset(T_i)$ and $wset(T_i)$ do not contain any common object (i.e. read set and write set of a transaction do not conflict with each other).

The *state* of a transaction represents the current state among *idle*, *waiting*, *running*, *committed*, and *aborted*. Initially, when a transaction is not assigned to any node, it is in *idle* state. When a transaction is issued to a node, we denote it as in *waiting* state. When a transaction starts executing, it is in *running* state. If the transactions commits successfully, it reaches to the *committed* state, otherwise it aborts and reaches to the *aborted* state. We define the following notions that are required in the transaction execution:

*Definition 4.1 (*Dependecy Graph*). A transaction dependency graph D_G is a directed graph between the set of transactions computed using the objects in rset and wset of each transaction T_i, where the nodes of the graph represent the transactions and the directed edge between the two nodes represent the dependency (at least a write) between the transactions.*

If an object O_i contained in the $rset(T_1)$ or $wset(T_1)$ of a transaction T_1 also lies in the $rset(T_2)$ or $wset(T_2)$ of another transaction T_2, T_1 is dependent on T_2 and vice-versa.

*Definition 4.2 (*Priority Queue*). A priority queue for the transaction execution is a list containing the priority order for the execution of the transactions.*

*Definition 4.3 (*Conflict*). When two transactions T_1 and T_2 are accessing the same object O_i and at least one transaction (T_1 and/or T_2) is performing a write operation on that object, conflict is said to be occurred between the transactions T_1 and T_2.*

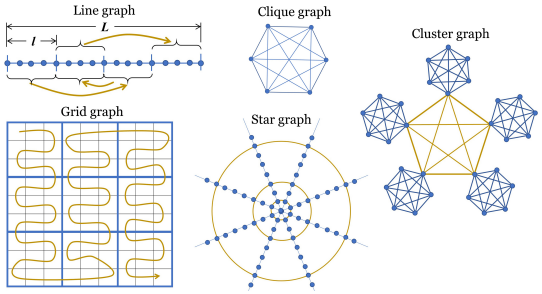

Figure 3: Illustration of division of 5 types of graph into sub-graphs and the priority queue generation.

*Definition 4.4 (*Conflict Graph*). A transaction conflict graph C_G of a set of transactions $\mathcal{T}$ is a graph where each transaction $T_i \in \mathcal{T}$ represents the node and an edge between the two transactions T_i and T_j represents the conflict between T_i and T_j.*

The transaction conflict graph C_G is similar to the transaction dependency graph D_G where the dependency is calculated based on the conflict occurrences between the set of transactions $\mathcal{T}$. The conflict graph together with the priority queue can be represented by a directed acyclic graph (DAG).

*Definition 4.5 (*Execution Time*). The execution time for a set of transactions $\mathcal{T}$ is the total time steps required to finish the execution of all the transactions in $\mathcal{T}$.*

*Definition 4.6 (*Communication Cost*). The communication cost for a transaction T_i is the total distance objects in $rset(T_i)$ and $wset(T_i)$ traverse while bringing them to the executing node of T_i.*

*Definition 4.7 (*Waiting Time*). The waiting time for a transaction T_i is defined as the total time steps spent by the transaction T_i residing at a node before starting its execution.*

As long as there is no *conflict* between the transactions at two different nodes, those transactions can execute in parallel and commit successfully. But when conflict occurs, only one transaction can commit successfully and at least one of them must be aborted when executed in parallel. To decide which transaction to execute and commit successfully in case of conflict occurrence, the *priority queue* of transactions helps; the transaction with higher priority executes and commits first, while the transaction with lower priority waits until the previously running transaction commits.

Priority Queue Generation. The *priority queue* for each type of graph is computed separately. This is useful in the *offline* scheduling algorithms where each node contains a transaction. First, a graph is divided into a number of sub-graphs and transactions are executed in parallel within each sub-graph. Within each sub-graph, the priority order of transaction execution is defined based on the position of each node. Fig. 3 illustrates the division of each type of graph into sub-graphs to generate the priority queue. The priority queue computation for five different graphs (line, clique, grid, cluster, and star) that we incorporate in GraphTM follows the schedule proposed for each type of graph in [2]. The provably-efficient guarantee of GraphTM also implies from [2].

4.2 High Level Overview of Algorithm

In this paper, we propose two variants of algorithms for processing transactions in a distributed environment based on the initial set of known parameters. One is the *offline algorithm* in which the *graph structure* (G_{type}), all the available *transactions* ($\mathcal{T}$), total number of *nodes* ($\mathcal{N}$), and total *objects* ($\mathcal{O}$) are known before the transactions start executing. The other one is *online algorithm* in which the *graph structure* (G_{type}) is known beforehand and other parameters are known only after transactions arrive.

Offline Algorithm. In the offline algorithm, a centralized scheduler schedules transactions in all the nodes of a distributed system; the scheduler knows total available transactions, objects, nodes and graph structure. The scheduler also computes non-conflicting transactions among them to execute concurrently. In case of conflict occurrence between two transactions at two different nodes, transaction assigned to the node with higher priority is executed first and the other transaction waits until the first transaction commits. We have two different versions of the *offline* algorithm: (i) OfflineBatch and (ii) OfflineStream.

OfflineBatch: In the OfflineBatch algorithm, transactions are executed in batch in such a way that one transaction is executed at each node in every round. When a transaction $T_{i,1}$ running at a node N_i commits in round j, next available transaction $T_{i,2}$ reaches N_i to execute in round $j+1$. $T_{i,2}$ waits to start its execution until all the transactions assigned to different nodes for current round j finish their execution and commit. In this algorithm, each node runs equal (± 1) number of transactions when execution is completed. This technique provides better performance when the transactions are of constant size and not conflicting with each other. In this case, transactions run concurrently at all the nodes in each round. But in the case where transactions have varying read/write set sizes and are conflicting to one another, the batch technique penalizes the performance. This is because transactions may need to wait for a longer period due to conflicts and non-uniform read/write set sizes.

OfflineStream: In the OfflineStream algorithm, the transactions at different nodes do not require to be executed in batch. Any new transaction $T_{i,j}$ is taken into account to create the transaction conflict graph C_G as soon as it reaches node N_i. Then, after generating the components and independent sets of C_G, all the non-conflicting transactions at separate independent sets start their executions concurrently. The new non-conflicting transaction does not need to wait for other transactions to commit. Still, the transactions at the node(s) having low priority order may wait for a long time when transaction conflict rate is high.

Online Algorithm. In the online algorithm (denoted as Online), each node is known only about the graph structure and the transaction(s) arrived on it. There is no synchronicity on when the transaction(s) reach to the free nodes. That means a node may remain idle (free) for an undefined period of time. For each object, a data structure containing the information about accessing transactions is maintained which is updated after each transaction commit. Therefore, a node can detect the case if the objects in read set ($rset(T_i)$) and write set ($wset(T_i)$) of a transaction T_i assigned to it are concurrently accessed by any other transaction(s). If no conflict is detected for the transaction T_i, it starts running. After a transaction T_i at a node commits, next transaction T_j will reach to the node anytime later. But, as soon as the transaction T_j reaches the node, the conflict detection is performed and the non-conflicting transactions are executed in parallel.

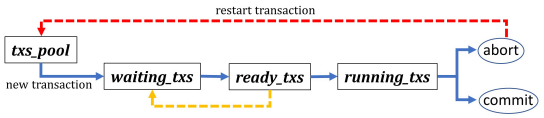

Figure 4: Illustration of flow of transactions in GraphTM.

5 IMPLEMENTATION DETAILS

In this section, we first present the data structures used by GraphTM to realize scheduling through offline and online algorithms and then describe the execution of transactions in GraphTM followed by the implementation details. Finally, we discuss the correctness of algorithms adapted in GraphTM.

Data Structures. GraphTM uses the following data structures to efficiently issue a transaction to a node for execution:

- *txs_pool*: The set of total available transactions. GraphTM stores all the initially generated transactions in *txs_pool*.
- *waiting_txs*: A set of transactions arrived at nodes but waiting for execution turn.
- *ready_txs*: A set of waiting transactions at nodes that are ready to competing for execution turn. This is a subset of *waiting_txs* i.e. $ready\_txs \subseteq waiting\_txs$.
- *running_txs*: A set of in-flight transactions at different nodes.
- *txs_access_list*: A list consisting of transactions arrived at different nodes and are going to write on an object O_i. This data structure is used in online algorithm to find the concurrently executing conflicting transactions.
- *committed_txs*: A set of committed transactions.
- *aborted_txs*: A set of aborted transactions.

Transactions Execution. The execution of a transaction T_i begins after it reaches to a node N_i. Each transaction in *txs_pool* is in *idle* state. Transaction T_i changes its state from *idle* to *waiting* when it reaches to a node and is added to *waiting_txs*. The waiting transaction is then added to the *ready_txs* denoting that it is ready to compete for an execution turn. If T_i does not detect another concurrent and conflicting running transaction T_j, it starts its execution and changes its state from *waiting* to *running*. The *waiting_txs* and the *running_txs* lists are also updated accordingly by removing the transaction T_i from *waiting_txs* and adding it to the *running_txs*. Since GraphTM has implemented the data-flow model, each transaction needs to collect the required objects before performing read and write operations on them. The accessed objects of a transaction T_i in read set ($rset(T_i)$) and write set ($wset(T_i)$) are collected to the executing node N_i where T_i is issued by following the shortest path. The objects can move in parallel to reach the destination node. When all the required objects are collected to the executing node N_i, T_i starts write operation on them. As soon as T_i finishes writing successfully to the objects, it commits and changes its state to *committed* from *running*. If T_i fails due to any reason, it aborts and changes its state to *aborted* and restarted later adding it to the *txs_pool*. The total communication cost, total execution time and waiting time of each transaction is updated when the transaction commits. The node N_i is marked $free$ after T_i commits. The overall flow of the execution of transactions is depicted in Fig. 4.

Algorithm 1: OfflineStream()

Input : txs_pool and graph G with N nodes;
Output: *total exec time, total comm cost* and $committed\_txs$;

$total\_txs \leftarrow txs\_pool \cdot size()$;
$ready\_txs \leftarrow \phi$; $running\_txs \leftarrow \phi$; $waiting\_txs \leftarrow \phi$;
while $committed\_txs \cdot size() < total\_txs$ **do**
  for each node $nd \in N$ **do**
    if $nd \cdot state = free \wedge txs\_pool \neq \phi$ **then**
      $t \leftarrow txs\_pool \cdot get(0)$;
      $t.node \leftarrow nd$; $t.state \leftarrow waiting$; $nd.state \leftarrow busy$;
      $ready\_txs \cdot add(t)$; $waiting\_txs \cdot add(t)$;
      $txs\_pool \cdot remove(t)$;
  $C \leftarrow$ list of components generated using $ready\_txs$;
  for each component $c_i \in C$ **do**
    // sort based on priority queue
    $c_s^i \leftarrow$ sorted c_i containing txs with priority order;
    $I \leftarrow$ list of non-conflicting txs (independent set);
    for each transaction $tx \in I$ **do**
      if $tx \notin running\_txs$ **then**
        $running\_txs \cdot add(tx)$;
        $ready\_txs \cdot remove(t)$; $waiting\_txs \cdot remove(t)$;
        $tx \cdot begin()$; $tx \cdot execute()$;
  for each transaction $tx \in running\_txs$ **do**
    update *total exec time* and *comm cost*;
    if $tx \cdot commit()$ **then**
      $tx \cdot state \leftarrow committed$; $committed\_txs \cdot add(tx)$;
      for each $obj \in wset(tx)$ **do**
        $obj \cdot node \leftarrow tx \cdot node$;
    else if $tx \cdot abort()$ **then**
      $tx \cdot state \leftarrow aborted$; $aborted\_txs \cdot add(tx)$;
    $running\_txs \cdot remove(tx)$; $(tx \cdot node) \cdot state \leftarrow free$;
  for each transaction $tx \in aborted\_txs$ **do**
    $tx \cdot state = idle$; $txs\_pool \cdot add(tx)$;

Implementation of Offline Algorithm. The pseudocode for OfflineStream is provided in Algorithm 1. The pseudocode for OfflineBatch is removed due to space constraints. In the offline algorithm, the centralized scheduler is known about the complete list of transaction execution parameters for all the nodes beforehand. All the transactions are buffered in a pool, txs_pool, before the scheduler issues them to a node. The scheduler issues a transaction to a free node following the *offline* algorithm (OfflineBatch or OfflineStream) and marks the node as *busy*. The issued transaction is added to the list $waiting\_txs$ denoting that the transaction is waiting for its turn to execute. In OfflineStream algorithm, all the waiting transactions in $waiting\_txs$ are immediately added to the list $ready\_txs$ as well. Whereas in OfflineBatch algorithm, the transactions in $waiting\_txs$ are added to $ready\_txs$ only after the completion of current round. Here, *round* denotes the total period during which all the nodes finish executing one transaction each. We use a counter ($exec\_count$) for each node to keep record of executed transaction at that node. The priority order of transaction execution for each type of graph is computed as described in Section 4. Instead of sequentially executing the transactions at every node following the priority order, the algorithm finds the non-conflicting concurrent transactions for execution. For this, a transaction conflict graph (Definition 4.4) is generated first from the transactions in $ready\_txs$. To construct the transaction conflict graph, each transaction is denoted as a node and the conflict between two transactions is donated as an edge between the nodes representing the respective transactions. The conflict between the two transactions T_1 and T_2 is found by checking the read sets and write sets of both the transactions. After this, it computes the *components* (Definition 5.1) of the conflict graph.

*Definition 5.1 (*Component*). A component of a transaction conflict graph is a sub-graph of transactions where any two transactions inside the same component conflict with each other but the transactions from two different components do not conflict with each other.*

Note here that two different components of a conflict graph consist of disjoint set of transactions such that there is no common object in read sets and write sets of two transactions from the two different components. Moreover, no object in read set and write set of a transaction issued to a node from one component resides at a node to which another transaction is issued in the next component. Inside each component, the transactions are sorted based on the order computed in priority queue. From each component, at least one transaction can run concurrently without conflict. To increase the parallelism, we further compute *independent sets* (Definition 5.2) inside each component by picking up the available transactions with higher priority order for execution.

*Definition 5.2 (*Independent Set*). An independent set I of a component C having n number of transactions is a list consisting of a group of transactions $T_{IS} = \{T_1, T_2, \ldots, T_k\}, 0 < k \leq n$, where there is no edge (conflict) between any two transactions $T_i, T_j \in T_{IS}$ in C.*

The independent set I of a component consists of a set of transactions having no directly connected edge between them. That means I represents the set of non-conflicting transactions within a component and those transactions can execute in parallel. The scheduler selects the transaction with higher priority order first to add in the independent set. The computation of the components and the independent sets is performed when a new transaction is added to $ready\_txs$. Once the transaction starts running, it is removed from both the lists $ready\_txs$ and $waiting\_txs$ and added to the list $running\_txs$ containing current set of in-flight transactions. When a transaction $T_i \in running\_txs$ running at a node commits, T_i is removed from $running\_txs$ and is added to the list of committed transactions $committed\_txs$. When T_i aborts, it is added to the $aborted\_txs$ list and is restarted later after collecting it to the txs_pool. The execution time and communication cost for the transaction is noted down at the time of commit (or abort). Also, the node at which the transaction was executed is marked $free$ and next available transaction T_j from txs_pool is assigned to the free node changing it to busy. In the OfflineStream algorithm, the new transaction T_j is added to the $ready\_txs$ immediately in the next time step and which will participate to find the non-conflicting concurrent transactions. But, in the OfflineBatch algorithm, although the transaction T_j is available at the node, it is not added to the $ready\_txs$ immediately, rather it waits for next round.

Algorithm 2: Online()

Input : txs_pool and graph G with N nodes;
Output: *total exec time, total comm cost* and $committed\_txs$;

$total\_txs \leftarrow txs\_pool \cdot size()$;
$ready\_txs \leftarrow \phi$; $running\_txs \leftarrow \phi$; $waiting\_txs \leftarrow \phi$;
while $committed\_txs \cdot size() < total\_txs$ **do**
  for each node $nd \in N$ **do**
    if $nd \cdot state = free \wedge txs\_pool \neq \phi \wedge Random()$ **then**
      $t \leftarrow txs\_pool \cdot get(0)$;
      $t.node \leftarrow nd$; $t.state \leftarrow waiting$; $nd.state \leftarrow busy$;
      $ready\_txs \cdot add(t)$; $waiting\_txs \cdot add(t)$;
      $txs\_pool \cdot remove(t)$;
  for each transaction $tx \in ready\_txs$ **do**
    $conflict \leftarrow false$;
    for each object $obj \in wset(tx)$ **do**
      $T \leftarrow txs\_access\_list(obj)$;
      for each transaction $t \in T$ **do**
        if $t \in running\_txs$ **then** $conflict \leftarrow true$;
    if $conflict = false$ **then**
      $running\_txs \cdot add(tx)$; $tx \cdot begin()$;
      $tx \cdot execute()$;
  for each transaction $tx \in running\_txs$ **do**
    update *total execution time* and *communication cost*;
    if $tx \cdot commit()$ **then**
      $tx \cdot state \leftarrow committed$; $committed\_txs \cdot add(tx)$;
      for each $obj \in wset(tx)$ **do**
        $obj \cdot node \leftarrow tx \cdot node$;
    else if $tx \cdot abort()$ **then**
      $tx \cdot state \leftarrow aborted$; $aborted\_txs \cdot add(tx)$;
    $running\_txs \cdot remove(tx)$; $(tx \cdot node) \cdot state \leftarrow free$;
  for each transaction $tx \in aborted\_txs$ **do**
    $tx \cdot state = idle$; $txs\_pool \cdot add(tx)$;

Implementation of Online Algorithm. The pseudocode for the Online algorithm is given in Algorithm 2. In the online case, the distributed system is aware only of the graph structure (e.g., clique, grid, star, etc.) but it does not know how many nodes on the graph. Moreover, the total number of transactions in the system is also not known. Therefore, each node only has the information of transaction issued to it. We implement the *visible* write and *invisible* read operations. That means, each transaction T_i can detect other concurrent and conflicting transaction T_j writing on the memory location(s) of any object in its write set, $wset(T_i)$. But it does not know about the concurrent transaction reading the memory location of any object in its read set ($rset(T_i)$) and write set ($wset(T_i)$). The information of each transaction writing to the memory location of an object is maintained by using the txs_access_list data structure for each object O_i denoted as $txs\_access\_list(O_i)$.

When a transaction T_i arrives at a free node, it is added to the txs_access_list of each object in its write set, $wset(T_i)$ and the node is marked *busy*. The transaction T_i is also added to the $ready\_txs$ and $waiting\_txs$ lists. The arrival of a new transaction at a node is implemented by using a random function. For each transaction $T_i \in ready\_txs$, the online algorithm computes the *conflict* by checking concurrently executing transactions in $txs\_access\_list(O_i), \forall O_i \in wset(T_i)$. When a transaction T_i does not conflict with any other transaction T_j, it is added to the $running\_txs$ and started executing concurrently. When the transaction T_i running at a node commits (aborts), the node is marked *free* and T_i changes its state to *committed* (*aborted*). The transaction T_i is removed from the $running\_txs$ and added to the $committed\_txs$ ($aborted\_txs$). New transaction may arrive to the free node anytime later. When a new transaction arrives at the node, the above procedure is repeated. If a transaction T_i conflicts with previously running transaction T_j, T_i has to wait until the T_j finishes execution.

6 EVALUATION

The experiment is performed on an Intel Core i7-7700K processor with 32 GB RAM by simulating the distributed environment in GraphTM. We conducted the experiment by considering five different communication graphs as the distributed environment and varied the total number of nodes from 100 to 1,000 for each graph. The total number of objects, total number of transactions and the transaction sizes vary on different benchmarks. Also, each transaction is executed with 20 percent update rate. Each test case is run 10 times and the results presented are the average of these 10 runs. We used a set of micro-benchmarks and complex benchmarks from STAMP in the empirical evaluation.

We measure three execution parameters namely *execution time, communication cost* and *waiting time* for all three algorithms OfflineBatch, OfflineStream and Online. We compute the optimal execution time considering an optimal algorithm (denoted as Optimal). In Optimal, transactions are scheduled in such a way that every time, a non-conflicting transaction reaches to a free node and it can start executing immediately. The respective communication cost for the Optimal algorithm is also computed. Since no transaction is required to wait for execution turn after reaching a node, the waiting time for Optimal becomes zero.

We also compare the communication overhead for processing transactions in distributed environment by running the set of micro- and complex benchmarks in a multi-core system using TinySTM [8, 9]. TinySTM is a word-based software transactional memory (STM) implementation for tightly-coupled multi-core systems sharing a single shared memory. Similar to GraphTM, we measure the time steps and number of hops in TinySTM. In TinySTM, multiple threads can be run in parallel to execute transactions concurrently. The total number of parallel threads in multi-core system are analogous to the total number of nodes in distributed system. But, since every thread is accessing the same shared memory in a single processor, we count every memory access as one time step. A transaction commits upon successful read/write operation without conflict. Otherwise, transaction aborts and changed memory locations by the transaction are rolled back to the previous consistent state. This suggests that every commit takes one time step to reflect the changes on the memory locations and every abort takes two time steps; first step, to make changes on a memory location and second step, to roll back the changes to the previous consistent states. Thus, in the multi-core system, using TinySTM, we measure the communication cost as $total\_communication\_cost = t_C + 2 \times t_A$ where t_C and t_A

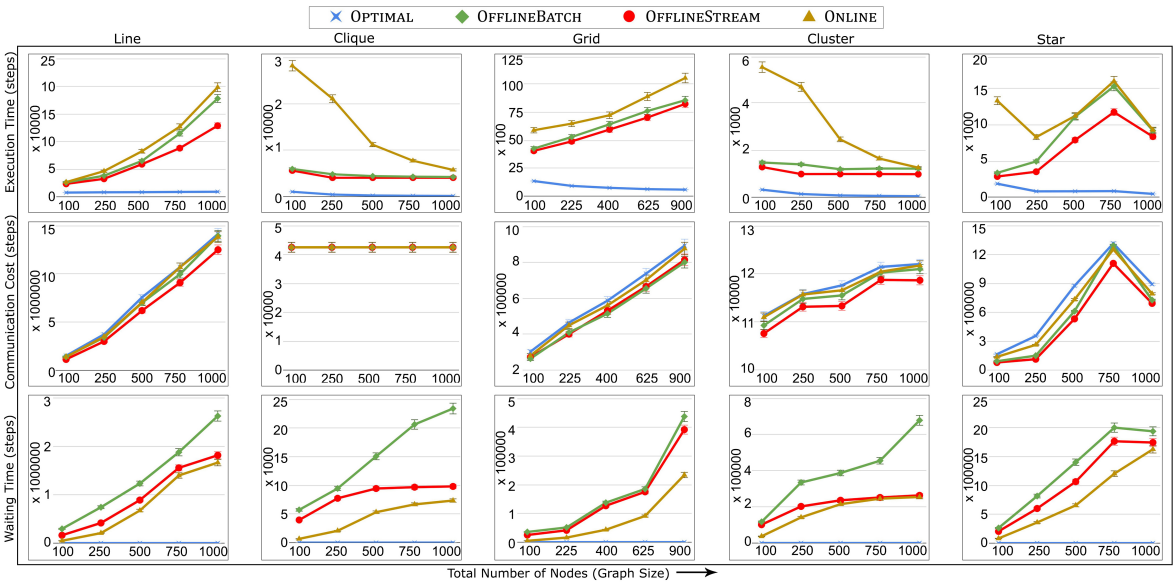

Figure 5: Illustration of execution results in five different graphs for bank micro-benchmark varying the graph size.

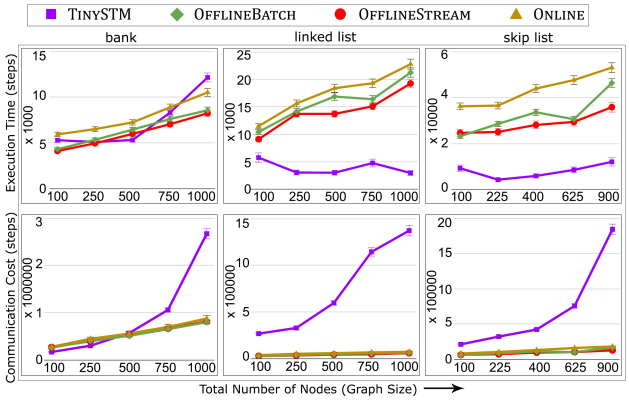

Figure 6: Illustration of communication overhead by running micro-benchmarks on grid graph.

are the total number of commits and total number of aborts in all the parallel threads, respectively. If t_C^i and t_A^i be the total number of commits and total number of aborts in a thread $i, 0 \leq i \leq n-1$, the execution time is measured as $exec\_time = max(t_C^i + 2 \times t_A^i)$.

Porting the Benchmarks to GraphTM**.** The micro and STAMP benchmarks have been extensively used for evaluation of TM implementation in multi-core systems processors [7, 13, 16]. Bank, linked list, and vacation have also been used for evaluation of TM in previous distributed TM implementations [11, 21, 22, 27]. Since the micro and STAMP benchmarks were designed for multi-core systems, the benchmarks need modifications to execute in a distributed scenario. For this purpose of porting them to GraphTM, we use TinySTM API. Each benchmark is first executed with TinySTM generating a set of transactions $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$. Each memory location accessed (read/written) by a transaction is mapped to a unique object defined by *obj_id*. Total objects for GraphTM are then computed by finding the union of all those objects (memory locations) accessed by the transactions. Since objects are located at different nodes in a distributed environment, each object is mapped to a (owner) node of the graph defined by *obj_node*. Note that the GraphTM uses a single writable copy of each object and hence one object has no more than one owner node. Now, $\mathcal{T}$ consists of a set of transactions having objects in read set and write set distributed over the nodes of a graph. These transactions are stored in pool *txs_pool*. During runtime, one available transaction from the pool is assigned to a free node at a time. Moreover, in Online algorithm, a transaction may not reach to a node as soon as it becomes free. To simulate this online scenario, we use a random function to assign the available transaction from the pool to a free node.

Results on Micro-benchmarks. We executed 10,000 transactions for each micro-benchmark and measured the execution time, communication cost and waiting time for each algorithm. Fig. 5 illustrates the execution result for bank micro-benchmark for all 5-different graphs (distributed environments) with varying sizes from 100 to 1,000 nodes. The plots for linked list and skip list are omitted due to space constraint.

The plots show that Optimal has minimal execution time compared to OfflineBatch, OfflineStream and Online but the communication cost of Optimal is greater than that of others. We observed up to 1.54×, 4× and 2× greater communication cost in Optimal compared to OfflineBatch, OfflineStream and Online, respectively for the micro-benchmarks. Fig. 5 also shows that OfflineBatch and OfflineStream have less execution time and communication cost than the Online. This is because in offline case, as soon as a node becomes free, a new transaction can be assigned to it, while in online case, a node may remain idle for a random period of time. Also, in offline algorithms, transaction on each node is scheduled earlier based on the priority order which decreases the total execution time and total communication cost.

On clique, cluster and star graph, the execution time of Online is very high compared to OfflineBatch and OfflineStream when executed with less number of nodes. This is because in clique, cluster or star graph, there is high contention between transactions with less number of nodes and hence less number of concurrently executing transactions. We observed up to 9.6× greater execution time (in clique) in Online compared to the OfflineStream when executed with 100 nodes. We see that the execution time in Online

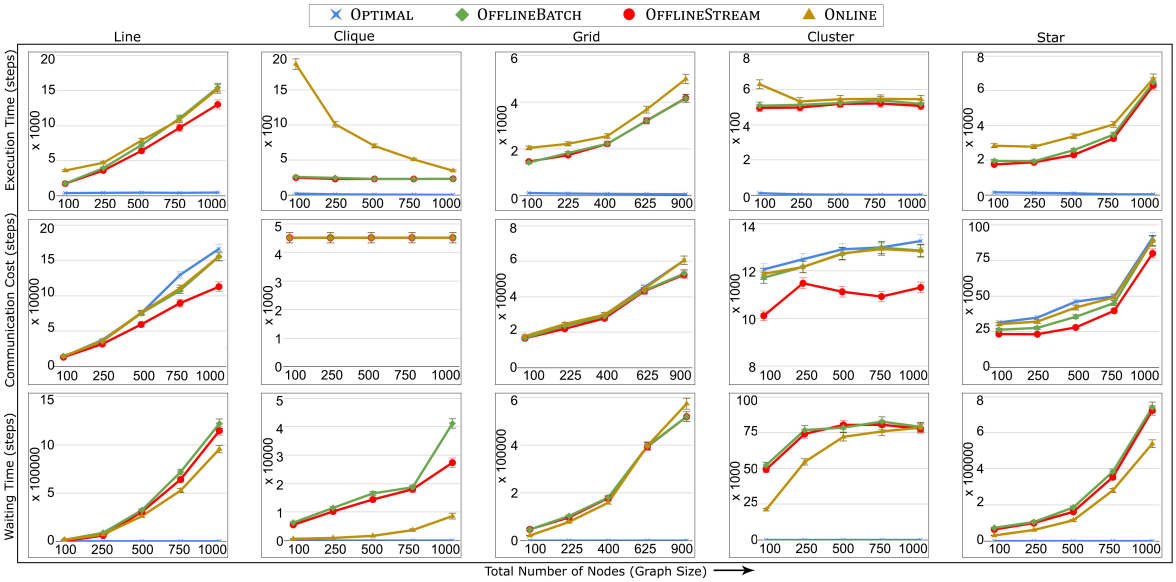

Figure 7: Illustration of execution results in five different graphs for bayes benchmark from STAMP varying the graph size.

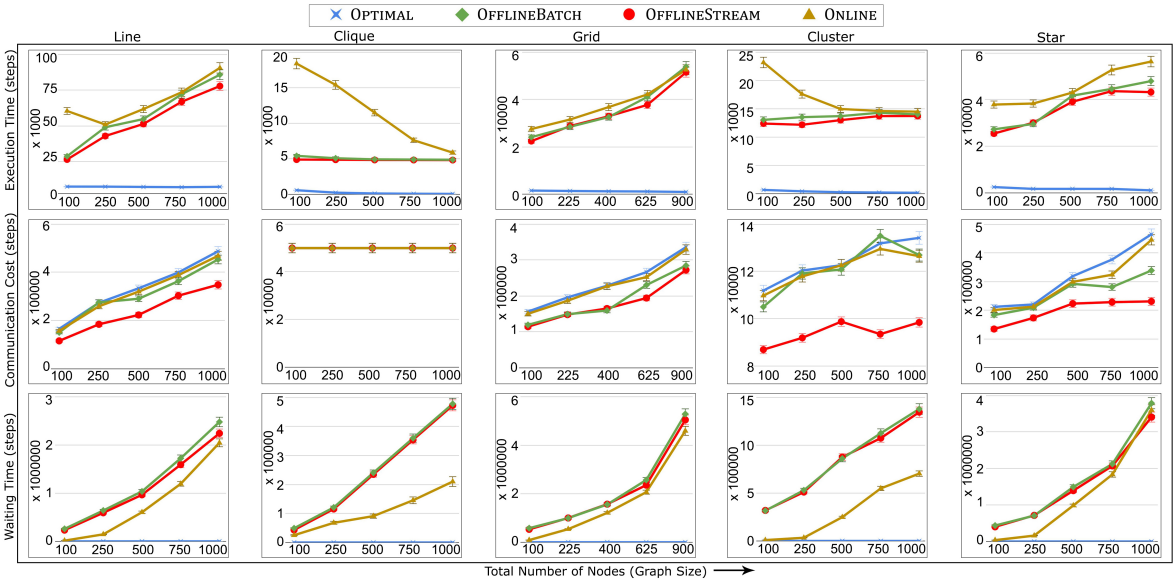

Figure 8: Illustration of execution results in five different graphs for genome benchmark from STAMP varying the graph size.

gradually decreases with more number of nodes. On the other hand, the waiting time for Offline is greater than that for the Online. This is relatable from the fact that transactions arrive later in the online case requiring to wait at a node for less amount of time before execution. The execution time in offline algorithms increases with the increase in waiting time.

Fig. 6 illustrates the communication overhead of transactions execution in grid using micro-benchmarks. The figure shows that the communication cost in TinySTM (multi-core system) increases largely with the increase in total number of parallel threads (analogous to nodes in the grid). Compared to the increase in communication cost, the increase in execution time is less in TinySTM. But, in GraphTM, execution time increases largely even with the small increase in communication cost compared to the TinySTM. This proves our claim that the communication overhead largely affects the runtime of transactions in distributed environment.

Results on STAMP benchmarks. Fig. 7 and Fig. 8 present the experimental results for *bayes* and *genome* from STAMP, respectively. The plots for *intruder*, *kmeans* and *vacation* are omitted due to space constraint. Results for STAMP benchmarks follow similar to the results for micro-benchmarks. OfflineBatch and OfflineStream showed better execution time and communication cost than Online. OfflineStream performed better than OfflineBatch in terms of both execution time and communication cost.

The plots show that clique and cluster graphs have almost constant execution time and communication cost for any graph size. In Online, the execution time is high with less number of nodes and becomes almost constant with large number of nodes. This is due to the property of complete graph where each node is connected with all others and this controls the concurrently executing transactions. All the plots show that waiting time for transactions increases continuously with more number of nodes.

	Execution time gain		
	Batch/Stream	Online/Stream	Stream/Optimal
line	3.1 (vacation)	5.4 (kmeans)	38 (linked list)
clique	1.2 (kmeans)	9.6 (kmeans)	84 (genome)
grid	1.3 (skip list)	3.5 (vacation)	55 (genome)
cluster	2.2 (vacation)	9.3 (kmeans)	51 (intruder)
star	2.0 (linked list)	4.5 (bank)	49 (vacation)

Table 1: Summary of comparison of execution time.

	Communication cost gain		
	Batch/Stream	Online/Stream	Optimal/Stream
line	4.0 (vacation)	4.5 (vacation)	5.6 (vacation)
clique	1.0	1.0	1.0
grid	1.3 (skip list)	2.2 (vacation)	2.7 (vacation)
cluster	2.0 (vacation)	2.0 (vacation)	2.2 (vacation)
star	2.1 (linked list)	2.6 (vacation)	2.7 (vacation)

Table 2: Summary of comparison of communication cost.

Table 1 shows the execution time gain in OfflineStream compared to OfflineBatch and Online. It also shows the execution time gain of Optimal compared to OfflineStream. We noticed that OfflineStream achieved up to 9.6× better performance compared to OfflineBatch and Online. Table 2 shows the communication cost gain in OfflineStream compared to OfflineBatch, Online and Optimal algorithms. We observed up to 5.6× communication gain in OfflineStream compared to Optimal.

In summary, OfflineStream performs better in terms of both execution time and communication cost among OfflineBatch, OfflineStream and Online algorithms. OfflineStream has also less communication cost compared to the Optimal for execution time. Also, the execution time for processing transactions in distributed system is largely affected by the communication cost overhead.

7 CONCLUDING REMARKS

In this paper, we have presented an efficient and scalable simulation framework GraphTM for supporting transactional memory in a distributed environment. GraphTM avoids the limitations of previous studies using provably-efficient approach of scheduling transactions so that the two crucial performance metrics (execution time and communication cost) are minimized. The evaluation results show the efficiency of GraphTM against both micro and complex benchmarks. For future work, it will be interesting to deploy our framework in real distributed system(s) and obtain the wall clock execution time, communication costs, and other related parameters. Moreover, it will be interesting to extend GraphTM to other topologies such as hypercube and butterfly. Furthermore, it will be interesting to extend STAMP and other benchmarks to execute them in a distributed setting.

ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation grant CCF-1936450.

REFERENCES

[1] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. 2008. Software transactional memory for large scale clusters. In *PPoPP*. 247–258.
[2] Costas Busch, Maurice Herlihy, Miroslav Popovic, and Gokarna Sharma. 2017. Fast Scheduling in Distributed Transactional Memory. In *SPAA*. ACM, 173–182.
[3] Costas Busch, Maurice Herlihy, Miroslav Popovic, and Gokarna Sharma. 2018. Time-communication impossibility results for distributed transactional memory. *Distributed Computing* 31, 6 (2018), 471–487.
[4] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Q. Le. 2013. Robust architectural support for transactional memory in the power architecture. In *ISCA*. 225–236.
[5] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. 2015. R2C2: A Network Stack for Rack-scale Computers. In *SIGCOMM*. 551–564.
[6] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís Rodrigues. 2009. D2STM: Dependable Distributed Software Transactional Memory. In *PRDC*. 307–313.
[7] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. 2010. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPOPP*. 67–78. https://doi.org/10.1145/1693453.1693464
[8] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. 2010. Time-Based Software Transactional Memory. *IEEE Trans. Parallel Distrib. Syst.* 21, 12 (2010), 1793–1807.
[9] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *PPOPP*. 237–246.
[10] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. 2011. Hardware transactional memory for GPU architectures. In *MICRO*. 296–307.
[11] Vincent Gramoli, Rachid Guerraoui, and Vasileios Trigonakis. 2018. TM^{2}C: a software transactional memory for many-cores. *Distributed Computing* 31, 5 (2018), 367–388.
[12] Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, Alan Gara, George Chiu, Peter Boyle, Norman Chist, and Changhoan Kim. 2012. The IBM Blue Gene/Q Compute Chip. *IEEE Micro* 32, 2 (2012), 48–60.
[13] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. 2003. Software Transactional Memory for Dynamic-sized Data Structures. In *PODC*. 92–101.
[14] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-free Data Structures. In *ISCA*. 289–300.
[15] Maurice Herlihy and Ye Sun. 2007. Distributed transactional memory for metric-space networks. *Distributed Computing* 20, 3 (2007), 195–208.
[16] Sachin Hirve, Roberto Palmieri, and Binoy Ravindran. 2014. Archie: a speculative replicated transactional system. In *Middleware*. 265–276.
[17] Intel. 2012. http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell. (2012).
[18] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Lujan, Chris Kirkham, and Ian Watson. 2008. DiSTM: A software transactional memory framework for clusters. In *ICPP*. 51–58.
[19] Dawei Li, Jie Wu, Zhiyong Liu, and Fa Zhang. 2017. Towards the Tradeoffs in Designing Data Center Network Architectures. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (Jan. 2017), 260–273.
[20] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. 2006. Exploiting Distributed Version Concurrency in a Transactional Memory Cluster. In *PPoPP*. 198–208.
[21] Sudhanshu Mishra, Alexandru Turcu, Roberto Palmieri, and Binoy Ravindran. 2013. HyflowCPP: A Distributed Transactional Memory Framework for C++. In *NCA*. 219–226.
[22] Mohamed Mohamedin, Sebastiano Peluso, Masoomeh Javidi Kishi, Ahmed Hassan, and Roberto Palmieri. 2018. Nemo: NUMA-aware Concurrency Control for Scalable Transactional Memory. In *ICPP*. 38:1–38:10.
[23] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. 2015. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *ISCA*. 144–157.
[24] Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luís Rodrigues. 2012. When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication. In *ICDCS*. 455–465.
[25] Mohamed M. Saad and Binoy Ravindran. 2011. HyFlow: A High Performance Distributed Software Transactional Memory Framework. In *HPDC*. 265–266.
[26] Nir Shavit and Dan Touitou. 1997. Software Transactional Memory. *Distrib. Comput.* 10, 2 (1997), 99–116.
[27] Alexandru Turcu, Binoy Ravindran, and Roberto Palmieri. 2013. Hyflow2: A High Performance Distributed Transactional Memory Framework in Scala. In *PPPJ*. 79–88.
[28] Paweł T. Wojciechowski and Konrad Siek. 2016. Atomic RMI 2: Distributed Transactions for Java. In *AGERE*. 61–69.
[29] Bo Zhang and Binoy Ravindran. 2010. Brief announcement: on enhancing concurrency in distributed transactional memory. In *PODC*. 73–74.
[30] Bo Zhang, Binoy Ravindran, and Roberto Palmieri. 2014. Distributed Transactional Contention Management as the Traveling Salesman Problem. In *SIROCCO*. 54–67.