2014

Scheduling in Transactional Memory Systems: Models, Algorithms, and Evaluations

Gokarna Sharma
Louisiana State University and Agricultural and Mechanical College

SCHEDULING IN TRANSACTIONAL MEMORY SYSTEMS: MODELS, ALGORITHMS, AND EVALUATIONS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Electrical Engineering and Computer Science

by
Gokarna Sharma
B.E., Tribhuvan University, 2004
M.S., Vienna University of Technology/Free University of Bolzano, 2008
August 2014

Acknowledgements

First, I would like to express my sincere gratitude to my advisor Prof. Costas Busch for his consistent support and guidance throughout my Ph.D. study. This long journey would not have been possible and this dissertation would not have been in this present form without his patience, motivation, and encouragement. His guidance helped me in diverse ways starting from doing research to doing good research. I cannot imagine having a better advisor and mentor. It has truly been an honor and pleasure to work with him.

I would also like to express my sincere gratitude to my advisory committee members: Prof. Bijaya B. Karki, Prof. Rajgopal Kannan, Prof. T. Warren Liao, and Prof. Rahul Shah, for being in my committee, their encouragement, and their insightful comments and suggestions which improved the quality of this dissertation in many ways. Moreover, I would like to thank Dr. Jong-Hoon Kim for his support and stimulating discussions during early years of my Ph.D. study.

I would also like to thank professors/researchers who I have had the opportunity to collaborate with in research related/unrelated to this dissertation, and the fellow Ph.D. students and friends who I have had the opportunity to get to know personally.

Finally, I would like to thank my family for believing in me and supporting me throughout my life. Especially, I am grateful to my wife, Sukirti Nepal, for everything. She was on my side all the time; she helped me whenever I needed help, supported me whenever I needed support, and encouraged me whenever I needed encouragement.

Table of Contents

List of Tables

List of Figures

Abstract

Transactional memory provides an alternative synchronization mechanism that removes many limitations of traditional lock-based synchronization so that concurrent program writing is easier than lock-based code in modern multicore architectures. The fundamental module in a transactional memory system is the transaction which represents a sequence of read and write operations that are performed atomically to a set of shared resources; transactions may conflict if they access the same shared resources. A transaction scheduling algorithm is used to handle these transaction conflicts and schedule appropriately the transactions.

In this dissertation, we study transaction scheduling problem in several systems that differ through the variation of the intra-core communication cost in accessing shared resources. Symmetric communication costs imply tightly-coupled systems, asymmetric communication costs imply large-scale distributed systems, and partially asymmetric communication costs imply non-uniform memory access systems. We made several theoretical contributions providing tight, near-tight, and/or impossibility results on three different performance evaluation metrics: execution time, communication cost, and load, for any transaction scheduling algorithm. We then complement these theoretical results by experimental evaluations, whenever possible, showing their benefits in practical scenarios. To the best of our knowledge, the contributions of this dissertation are either the first of their kind or significant improvements over the best previously known results.

Chapter 1
Introduction

1.1 Transactional Memory

To take the full advantage of the gains allowed by Moore's law, recent progress in multicore architectures has led to mainstream processor manufacturers adapting multicore designs. Modern multicore architectures enable the concurrent execution of an unprecedented number of threads. The benefit depends on using multiple threads efficiently within applications. This gives rise to the opportunity for extreme performance and the complex challenge of synchronization. The opportunity is that threads will be available to an unprecedented degree, and the challenge is that more programmers will be exposed to concurrency related synchronization problems that until now were of concern only to a selected few. Writing concurrent programs is a non-trivial task because of the complexity of ensuring proper synchronization. Conventional lock-based synchronization has several drawbacks which limits the parallelism offered by multicore architectures. Coarse-grained locks do not scale. Fine-grained locks are difficult to program correctly because locks are generally not composable. Transactional memory (TM) [76, 131] provides an alternative synchronization mechanism that is non-blocking, composable, and easier to write than lock-based code [112]. TM-based synchronization has recently been included in IBM's Blue Gene/Q [66, 138] and Intel's Haswell processors [35]. Previously, it was included in the .NET framework, Intel's STM C++ Compiler, and the UltraSPARC ROCK processor. TM is predicted to be widely used in future processors, possibly even GPUs [53, 140]. In the research community, several TM implementations (hardware, software, and hybrid) have been proposed and studied, e.g., [29, 39, 40, 44, 45, 51, 52, 65, 67, 69, 72, 73, 75, 99, 100, 106, 131, 135, 141]. The TM

book by Harris *et al.* [68] provides an excellent overview of the design and implementation of TM systems up to early spring 2010.

TM operates in a way similar to database transactions, and aggregates a sequence of shared resource accesses (reads or writes) that should be executed atomically (by a single thread) in a fundamental module called *transaction*. A transaction is a piece of code that executes a series of reads and writes to shared memory. As for example see Fig. 1.1 where transaction T_2 aggregates a sequence of shared resource accesses starting from $y = 2$ to $x = 3$, where x and y are shared resources. These reads and writes logically occur as a transaction at a single instance in time; intermediate states are not visible to other (successful) transactions. TM increases parallelism as no threads need to wait for access to a shared resource and different threads can simultaneously modify disjoint parts of a data structure that would normally be protected under the same lock. A transaction ends either by committing, in which case all of the updates take effect, or by aborting, in which case no update is effective. Each program thread generates a sequence of transactions. Transactions of the same thread execute sequentially by following the program execution flow. However, transactions of different threads may *conflict* when they attempt to access the same shared memory resources. The two transactions T_1 and T_2 of Fig. 1.1 conflict while executing concurrently as they try to access same shared resource x. The advantage of TM is that if there are no conflicts between transactions then the threads continue execution without delays that would have been caused unnecessarily if locking mechanisms were used. Thus, TM can be viewed as an *optimistic* synchronization mechanism [75]. The aborted transactions waste computing resources, energy, and reduce the overall performance of the TM system, sometimes drastically. Ideal execution of concurrent transactions should order the transactions to execute in such a way that it would minimize the number of aborts, but such an ordering may be difficult to obtain because transactions usually act on dynamic data and the conflicts are produced dynamically with no a priori knowledge not even of the data items to be accessed.

Transaction conflicts are detected using conflict detection mechanisms [133]. If a transaction T_1 discovers that it conflicts with another transaction T_2 (because they access a shared resource),

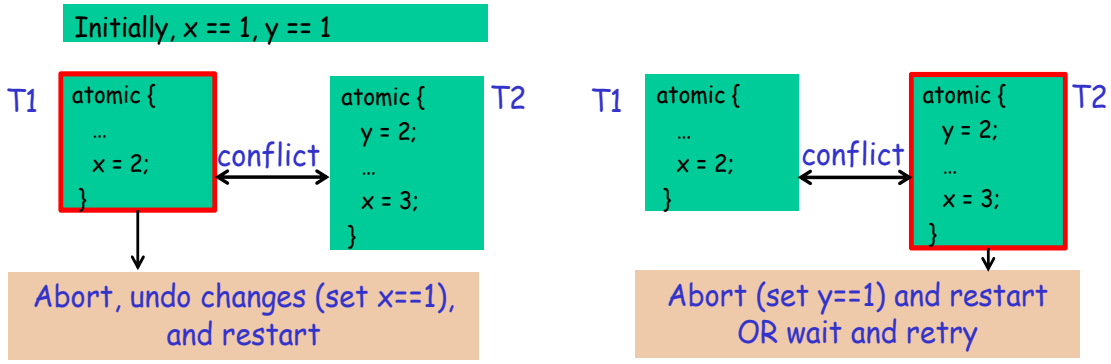


Figure 1.1: Resolving conflicts using a contention manager.

then T_1 has the following three choices: (i) it can give T_2 a chance to finish and commit by T_1 aborting itself; (ii) it can proceed and commit by forcing T_2 to abort; the aborted transaction T_2 then retries immediately again until it eventually commits; or (iii) it can wait (or back off) for a short period of time and retry the conflicting access again. In other words, a *conflict handling mechanism* decides which transactions should continue and which transactions should abort and try again until they eventually commit. If the conflict handling mechanism decides in favor of T_2, then T_1 will abort, undo its changes (i.e. sets $x == 1$ as the value 2 that was written in x while it was executing is not successful), and restarts its execution (see the left of Fig. 1.1). If the conflict handling mechanism decides in favor of T_1, then T_2 will abort, undo its changes setting $y == 1$, and restarts its execution or it waits and try to commit after backing off for a while (see the right of Fig. 1.1). This decision process leads us to the *transaction scheduling* problem. Typically, this transaction scheduling problem is *online* in the sense that transaction conflicts are not known a priori and they generally evolve over time.

Algorithms for this transaction scheduling problem are appealing as they need to make sure that the resulting execution of transactions give a serializable schedule. In other words, transaction can run concurrently but the results should follow some sequential execution. One such example is given in Fig. 1.2. Assuming that $x == 1$ and $y == 2$ initially, executing T_2 after t_1 gives $r_1 == 2$ and $r_2 == 3$ which is a serializable schedule; similarly, executing T_1 after T_2 gives $r_1 == 1$ and $r_2 == 2$ which is again serializable (see the left of Fig. 1.2). However, the execution scenario shown in the right of Fig. 1.2 gives $r_1 == 1$ and $r_2 == 3$ which is not a serializable schedule as

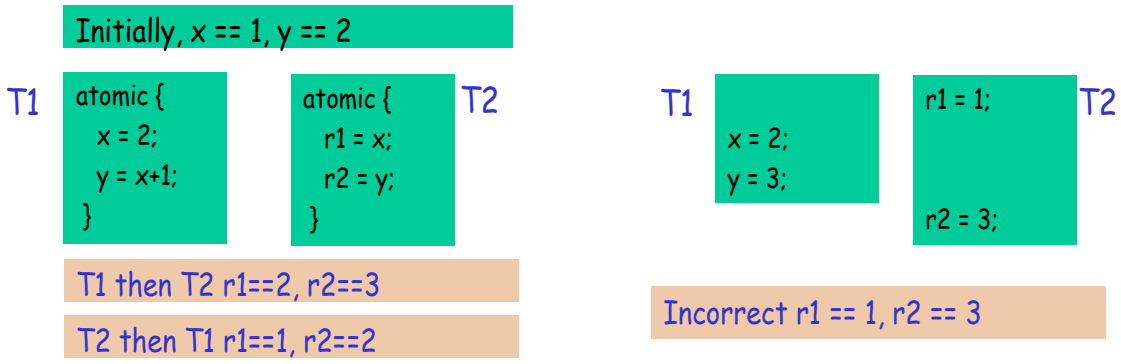


Figure 1.2: Serializability of transactions.

the execution of the sequences of the shared memory accesses of T_1 and T_2 *interleave* with each other.

To solve the transaction scheduling problem efficiently, each transaction consults with the *contention manager* module of the TM system for which choices to make. Contention manager modules help any transaction scheduling problem by detecting the conflicts which eventually determine whether the shared memory accesses of two or more transactions interleave. DSTM [75] is the first software TM (STM) implementation that uses a contention manager as an independent module to resolve conflicts between transactions and ensure *progress* – some useful work in done in each time step of the execution. A major challenge in guaranteeing progress through contention managers is to devise a scheduling algorithm which ensures that all transactions commit in the shortest possible time. Given a set of transactions, a central optimization metric in the literature, e.g. [9, 11, 57, 59, 117, 119], is to minimize the *makespan* which is defined as the duration from the start of the execution schedule, i.e., the time when the first transaction is issued, until all transactions commit. In a dynamic scenario where transactions are issued continuously, the makespan translates to the *throughput*, measured as the number of committed transactions per unit of time. The makespan of a transaction scheduling algorithm, which has minimal knowledge of the input transactions, can be compared to the makespan of an optimal off-line scheduling algorithm, which has complete knowledge of the resource requests, to provide a *competitive ratio*.

Since it is projected that a processor chip will have a large number of cores, it is important to design TM systems which scale gracefully with the variability of the system sizes and com-

plexities. To achieve this goal, it is desirable to devise scheduling algorithms which have both good theoretical asymptotic behavior and also exhibit good practical performance. Provable formal properties help to better understand worst-case and average-case scenarios and determine the scalability potential of the system. It is also equally important to design scheduling algorithms with good performance for various reasonable practical execution scenarios. This dissertation studies TM implementations in several system models and propose several transaction scheduling algorithms that exhibit both good theoretical and practical performance.

1.1.1 Chapter Organization

The rest of the chapter is organized as follows. We proceed by models and metrics that capture the performance evaluation of scalable TM systems in Sections 1.2 and 1.3, respectively. We then discuss the transaction scheduling problem in these models in Sections 1.4 and 1.5. We then discuss the motivation behind this study and present the objective in Section 1.6. In Section 1.7, we outline contributions of our study in different TM models. We conclude this chapter in Section 1.8 with an outline of the dissertation.

1.2 Transactional Memory Models

TM has been studied mainly in three system models that we describe below. The main distinction between these models is the variation of the intra-core communication cost in accessing shared memory locations. The communication cost can be symmetric, asymmetric, or partially symmetric. These types of communication cost models are appropriate to cover tightly-coupled systems, large-scale distributed systems, and their combinations.

1.2.1 Tightly-coupled Systems (Symmetric Communication)

This model represents the most common scenario where multiple cores reside in the same chip and they are connected to a single shared memory (see Fig. 1.3). A shared memory

refers to a (typically) large block of random access memory that can be accessed by several different processors in a multiple-processor computer system. A shared memory system is relatively easy to program since all processors share a single view of data and the communication between processors can be as fast as memory accesses to a same location. That is, processors operate on a same shared memory and the shared memory access cost is symmetric (uniform) across different processors (for example, the recent multicore processors such as Intel Xeon, AMD Opteron, Sun UntraSPARC, etc.). The shared memory access mechanism is implemented in tightly-coupled systems through a multi-level cache coherence algorithm (see left of Fig. 1.5). Transactional memory designs in tightly-coupled systems extend the built-in cache-coherence protocols already supported by modern architectures to provide multi-level cache coherence, so the focus is mainly on how to schedule the transactions such that conflicts among transactions are minimized.

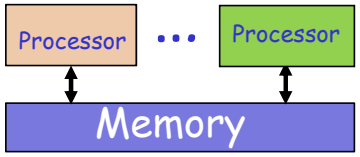


Figure 1.3: Illustration of a tightly-coupled system.

1.2.2 Distributed Networked Systems (Asymmetric Communication)

This model represents the scenario of completely decentralized distributed shared memory where processors are connected through a large-scale message passing system (1.4); the transactions that are running at different nodes operate on a common shared memory space that is split among processors. Hence, the shared memory is decentralized. Typically, the network is represented as a graph where the processors are nodes and links are weighted edges (see right of Fig. 1.5). The distance between processor nodes plays a significant role in the communication cost which is typically asymmetric among different network nodes. Note that this model is general enough to also include the uniform

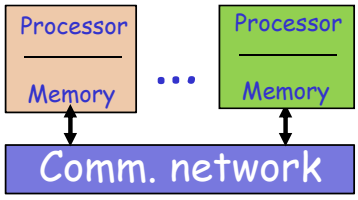


Figure 1.4: Illustration of a large-scale distributed system.

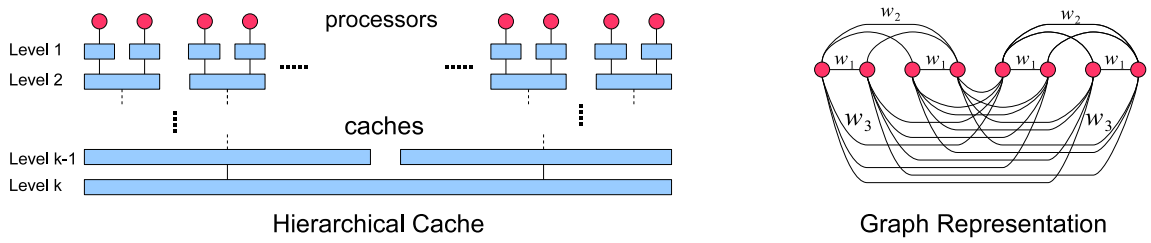


Figure 1.5: Left: a hierarchical multilevel cache; Right: a processor communication graph.

case of the tightly-coupled systems. This model is also suitable to model transaction scheduling scenarios that arise in cloud computing systems and heterogenous architectures.

1.2.3 Non-uniform Memory Access Systems (NUMA, Partially Symmetric Communication)

This model is a bridge between tightly-coupled systems and distributed systems. It represents a set of multiprocessors communicating through a small scale interconnection network. The interconnection network has a regular structure such as a grid (mesh), hypercube, butterfly, etc.; see Fig. 1.6 for a 3-dimensional multicore processor grid. Such network topologies have been extensively studied in the literature [92] and have predictable performance guarantees in terms of communication efficiency. There are two levels of communication: local (symmetric) communication within cores of the same processor and larger-scale (asymmetric) communication between different processors in different areas of the network topology. High performance multiprocessors are typically organized with such an architecture, e.g. [2, 34, 81], and their efficiency is vital for scientific applications.

1.3 Performance Evaluation Metrics

We focus on the following metrics that are used for evaluating the formal and experimental performance of transaction scheduling algorithms in the aforementioned TM models.

- *Makespan:* It measures the commit duration for the last transaction in a given input set of transactions. This is a typical performance metric in transaction scheduling in all TM

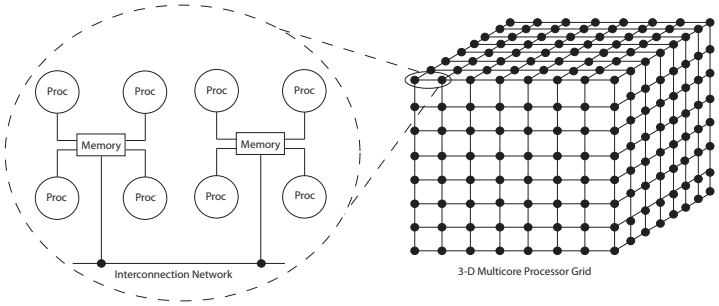


Figure 1.6: Illustration of a multi-processor system with high speed interconnect (i.e., Intel QPI [36]).

models. In a dynamic setting, the makespan translates to throughput. A primary goal for a transaction scheduling algorithm (i.e., a contention manager) is to minimize the makespan.

- *Communication cost:* It concerns distributed network TM models, and measures the number of messages sent on network links for scheduling the transactions. This metric relates to the total utilization of the distributed system resources, and it translates to the time and energy performance of the distributed transaction scheduling.

- *Load balancing:* This is particularly relevant for distributed and NUMA TM models, and it concerns the load of the network edges and nodes that is involved in fulfilling requests for the shared objects. Load balancing is important when energy and resource utilization needs to be minimized.

1.4 Transaction Scheduling in Tightly-Coupled Systems

Consider a set of $M \geq 1$ transactions $\mathcal{T} = \{T_1, T_2, \ldots, T_M\}$, one transaction each in M different threads $\mathcal{P} = \{P_1, \ldots, P_M\}$, and a set of $s \geq 1$ shared resources $\mathcal{R} = \{R_1, R_2, \ldots, R_s\}$. Since there is only one transaction in each thread, we call this problem the *one-shot* transaction scheduling problem. Each transaction is a sequence of actions (or operations) that is either a read or a write to some shared resource $R_i, 1 \leq i \leq s$. A resource can be read in parallel by arbitrarily many transactions. After a transaction is issued it either commits or aborts. The sequence of actions in a transaction must be atomic: all actions of a transaction are guaranteed to either completely occur

or have no effects at all. A transaction after it is issued and starts execution, it completes either with a *commit* or an *abort*. A transaction is *pending* after its first action until its last action; it takes no further actions after a commit or an abort. A pending transaction can restart multiple times until it eventually commits. The first action of a transaction must be a read or a write and its last action is either a commit or an abort.

Concurrent write-write actions or read-write actions to same shared resources by two or more transactions cause conflicts between transactions. If a transaction conflicts then it either aborts or it may commit and force all other conflicting transactions to abort. In *eager conflict management* TM systems, conflicts are resolved as soon as they are detected, whereas in *lazy conflict management* TM systems, conflict detection and resolution process is deferred to the end of a transaction. An execution schedule is called *greedy* if a transaction aborts due to conflicts it then immediately restarts its execution and attempts to commit again.

It is assumed that the execution time advances synchronously for all threads. Each transaction $T_i \in \mathcal{T}$ has execution time duration $\tau_i > 0$. The execution time is the total number of discrete time steps that the transaction requires to commit uninterrupted from the moment it starts. Let $\tau_{\max} := \max_i \tau_i$ be the execution time of the longest transaction, and $\tau_{\min} := \min_i \tau_i$ be the execution time of the shortest transaction. A resource can be read in parallel by arbitrarily many transactions. A transaction is called *read-only* if it only reads the shared resources, otherwise it is a *writing* transaction.

The *makespan* of a schedule for the transactions is defined as the duration from the start of the schedule, i.e., the time when the first transaction is issued, until all transactions have committed. The makespan of the transaction scheduling algorithm $\mathcal{A}$, denoted $makespan_{\mathcal{A}}$, for a given instance can be compared to the makespan of an optimal off-line scheduling algorithm, denoted $makespan_{\text{opt}}$, to provide a competitive ratio. Note that the makespan and the competitive ratio primarily depend on the *workload* $-$ the set of transactions, along with their arrival times, execution time duration, and resources they read and modify [11]. Therefore, the one-shot model described above is general enough to extend to different variations introducing some restrictions.

We now formally define pending commit property and makespan and competitive ratio.

Definition 1 (pending commit property [59]) *A transaction scheduling algorithm obeys the* pending commit property *if, whenever there are pending transactions, some running transaction T will execute uninterrupted until it commits.*

Definition 2 (makespan and competitive ratio) *Given a transaction scheduling algorithm $\mathcal{A}$ and a workload $\mathcal{T}$, $makespan_{\mathcal{A}}(\mathcal{T})$ is the total time $\mathcal{A}$ needs to commit all the transactions in $\mathcal{T}$. The* competitive ratio *of $\mathcal{A}$ on $\mathcal{T}$ is $CR_{\mathcal{A}}(\mathcal{T}) = \frac{makespan_{\mathcal{A}}(\mathcal{T})}{makespan_{opt}(\mathcal{T})}$, where opt is the optimal off-line scheduler. The competitive ratio of $\mathcal{A}$ independent of $\mathcal{T}$ is $CR_{\mathcal{A}} = \max_{\mathcal{T}} CR_{\mathcal{A}}(\mathcal{T})$ which is the maximum over all workloads $\mathcal{T}$.*

1.4.1 Conflict Graph

Consider a set of k transactions $\mathcal{T} := \{T_1, \ldots, T_k\}$. Let $\mathcal{R}(T_i)$ denote the set of resources used by transaction T_i. We can write $\mathcal{R}(T_i) = \mathcal{R}_w(T_i) \cup \mathcal{R}_r(T_i)$, where $\mathcal{R}_w(T_i)$ are the resources which are to be written by T_i and $\mathcal{R}_r(T_i)$ are the resources to be read by T_i.

Definition 3 (transaction conflict) *Two transactions T_i and T_j* conflict *if at least one of them writes on a common resource, that is, there is a resource R such that $R \in (\mathcal{R}_w(T_i) \cap \mathcal{R}(T_j)) \cup (\mathcal{R}(T_i) \cap \mathcal{R}_w(T_j))$ (we also say that R causes the conflict).*

From the definition of transaction conflicts we can define the conflict graph for a set of transactions. In the conflict graph, each node corresponds to a transaction and each edge represents a conflict between the adjacent transactions.

Definition 4 (conflict graph) *For a set of transactions $\mathcal{T}$, the* conflict graph $\mathcal{G}(\mathcal{T}) = (V, E)$ *is an undirected graph, which has as nodes the transactions, $V = \mathcal{T}$, and $(T_i, T_j) \in E$ for any two transactions T_i, T_j that conflict.*

Let $\delta(T_i)$ denote the degree of node T_i in G. We denote $C := \max_i \delta(T_i)$. Let $\gamma(R_j)$ denote the number of transactions that write to resource R_j, and let $\gamma_{\max} := \max_j \gamma(R_j)$ be the maximum

number among $\gamma(R_j), 1 \leq j \leq s$. Let $\lambda(T_i) = |\{R : R \in \mathcal{R}(T_i) \wedge (\gamma(R) \geq 1)\}|$ denote the number of resources that can be the cause of conflicts to transaction T_i, and let $\lambda_{\max} := \max_i \lambda(T_i)$ be the maximum number of resources that cause conflicts to any transaction in $\mathcal{T}$. Note that, in the conflict graph $\mathcal{G}(\mathcal{T})$, $C \leq \lambda_{\max} \cdot \gamma_{\max}$ and $C \geq \gamma_{\max} - 1$.

1.4.2 Problem Complexity

The transaction scheduling problem that we discussed above is a **NP-Hard** problem. The NP hardness result can be proven by reducing a well-known vertex coloring problem to the transaction scheduling problem. We provide here a short description of how that reduction works. Consider a vertex coloring problem instance that asks whether a given graph G is k-colorable [55]. A valid k-coloring is an assignment of integers $\{1, 2, \ldots, k\}$ (the colors) to the vertices of G so that neighbors receives different colors. The chromatic number, $\chi(G)$, is the smallest k such that G has a valid k-coloring. It is also shown in [50] that unless $\mathsf{NP} \subseteq \mathsf{ZPP}$, there does not exist a polynomial time algorithm to approximate $\chi(G)$ with approximation ratio smaller than $O(n^{1-\epsilon})$ for any constant $\epsilon > 0$, where n denotes the number of vertices in graph G. A transaction scheduling problem instance P asks whether a set of transaction $\mathcal{T}$ with a set of resource $\mathcal{R}$ has makespan k time steps assuming that each transaction has the execution time of length 1 time step.

The vertex coloring problem can be reduced to the transaction scheduling problem in polynomial time. Consider an input graph $G = (V, E)$ of the vertex coloring problem, where $|V| = n$ and $|E| = s$. We can construct a set of transactions $\mathcal{T}$ such that for each $v \in V$ there is a respective transaction $T_v \in \mathcal{T}$; clearly, $|\mathcal{T}| = |V| = n$. We also use a set of resources $\mathcal{R}$ such that for each edge $e \in E$ there is a respective resource $R_e \in \mathcal{R}$; clearly, $|\mathcal{R}| = |E| = s$. If $e = (u, v) \in E$, then both the respective transactions T_u and T_v use the resource R_e for write. Let $\mathcal{G}(P)$ be the conflict graph for the transactions $\mathcal{T}$. Note that $\mathcal{G}(P)$ is isomorphic to G. Node colors in G correspond to time steps in which transactions in $\mathcal{G}(P)$ are issued. Suppose that G has a valid k-coloring. If a node $v \in G$ has a color x, then the respective transaction $T_v \in \mathcal{G}(P)$ can be issued and commit at time step x, since no conflicting transaction (neighbor in $\mathcal{G}(P)$) has the same time assignment

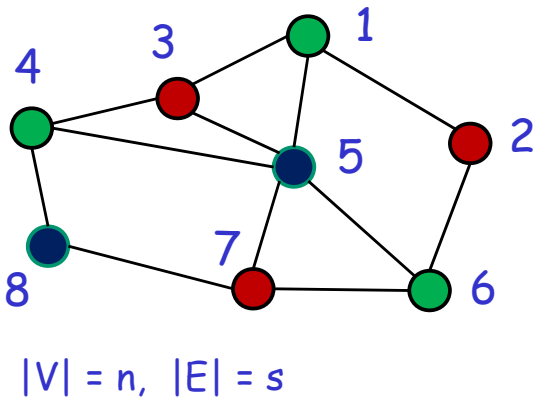
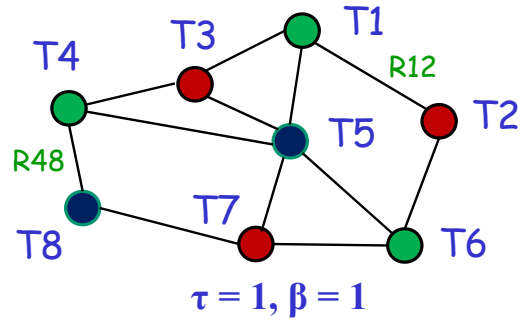


Figure 1.7: A vertex coloring problem.



Time	Step 1	Step 2	Step 3
Run and commit	T1, T4, T6	T2, T3, T7	T5, T8

Figure 1.8: A transaction scheduling problem, where $\tau = 1$ denotes that all transaction have execution time length of one time step and $\beta = 1$ denotes that all these transaction access shared resource for writing only.

(color) as T_v. Thus, a valid k-coloring in G implies a schedule with makespan k for the transactions in $\mathcal{T}$. Symmetrically, a schedule with makespan k for $\mathcal{T}$ implies a valid k-coloring in G.

Therefore, the transaction scheduling problem is in NP and from the reduction of the vertex coloring problem, we also obtain that the transaction scheduling problem is NP-complete. Fig. 1.8 shows the equivalent transaction scheduling problem after reducing the vertex coloring problem given in Fig. 1.7. As the transaction scheduling problem is in NP, the exact computation of the shortest makespan takes exponential time and therefore we try to schedule transactions such that the makepsan is not far from the shortest makespan.

1.5 Transaction Scheduling in Distributed and NUMA Systems

In distributed networked and NUMA systems, transactions in $\mathcal{T}$ are in the network nodes or processors (one transaction each in M different processors). It is assumed that there is a shared memory which is split (possibly equally) among the processors. Each processor has its own cache, where copies of *objects* (individual entries at the shared resources) reside. When a transaction running at a processor node issues a read or write operation for a shared memory location, the data object at that location is loaded into the processor-local cache. Some of the shared objects needed by a transaction may be in the shared memory of the node which is executing that transaction and some of the shared objects may be in the shared memory of other nodes. To be able to execute the transaction, either the shared objects in other nodes need to be moved to the node where the transaction is currently executing or the transaction needs to be moved to the node where the shared object needed by that transaction currently resides. This decision depends on the implementation technique used. In a *data-flow* implementation [77], transactions are immobile and objects are moved to nodes that need them. In a *control-flow* implementation [114], objects are immobile and transactions are moved to the nodes when objects reside. In a hybrid implementation, what to move, transactions or objects, is determined using some criteria minimizing some performance metric. We consider in this dissertation the data-flow implementation only.

In addition to makespan (Definition 2), any transaction scheduling algorithm for TM in distributed systems needs to minimize the communication cost and network load incurred in moving objects to the nodes that need them. Let $\mathcal{E} = \{r_0, r_1, \cdots, r_l\}$ be the $l+1$ shared object movements (or operations) from source nodes s_i to destination nodes t_i. The destination node t_i for each operation r_i is not known beforehand and the scheduling algorithm should find out the destination node online while in execution. The goal is to find a path p_i from s_i to t_i, for every object movement r_i, while minimizing both the maximum congestion along any edge e (any node v) in the network and the communication cost (the number of edges e that p_i uses).

Definition 5 (stretch) *Let $A(\mathcal{E}) = \sum_{i=1}^{l} |p_i|$ be the total communication cost of any consistency algorithm $\mathcal{A}$ while executing all the shared object operations in $\mathcal{E}$, where $|p_i|$ is the number of edges that the path p_i of the request r_i uses (in edge-weighted networks, $|p_i|$ translates to the total weight of the edges in the path p_i). The* stretch *of $\mathcal{A}$ on $\mathcal{E}$ is $stretch_{\mathcal{A}}(\mathcal{E}) = \frac{A(\mathcal{E})}{A^*(\mathcal{E})}$, where $A^*(\mathcal{E})$ is the communication cost of an optimal consistency algorithm that has complete knowledge about all the requests in $\mathcal{E}$. The stretch of $\mathcal{A}$ independent of $\mathcal{E}$ is $stretch_{\mathcal{A}} = \max_{\mathcal{E}} stretch_{\mathcal{A}}(\mathcal{E})$, which is the maximum over all possible sets of shared object operations $\mathcal{E}$.*

Definition 6 (congestion approximation) *Let $C = \max_e |\{i : e \in p_i\}|$ be the total edge congestion of any consistency algorithm $\mathcal{A}$ on an edge e and $C_n = \max_v |\{i : v \in p_i\}|$ be the total node congestion of any consistency algorithm $\mathcal{A}$ on a node v while executing the shared object operations in $\mathcal{E}$, where p_i is the path that is used by the request $r_i \in \mathcal{E}$. The* congestion approximation (CA) *of $\mathcal{A}$ on the edge e while executing the set $\mathcal{E}$ of shared object operations is $CA_{\mathcal{A}}(e) = \frac{C}{C^*}$ ($CA_{\mathcal{A}}(v) = \frac{C_n}{C_n^*}$), where C^* (C_n^*) is the optimal congestion on the edge e (the node v) that is attainable by any consistency algorithm to provide an approximation ratio on congestion for any edge e (any node v).*

Congestion on network edges and nodes can adversely affect the overall performance of the algorithm, especially in systems with limited bandwidth and/or in systems with limited computation power. For example, in sensor networks congestion can lead to random dropping of data and dramatic increase in energy consumption [90]. Congestion minimization is very important because it allows to evenly utilize available network resources (edges and nodes), avoiding the chance of the system being bottleneck due to some "hotspot" resources. This is done by reducing the communication/computation load on network edges and nodes through load distribution optimization.

In NUMA systems, the cost of accessing shared resources is asymmetric across different processors (symmetric communication within the cores of the same processor and asymmetric communication between different processors), in contrast to tightly-coupled systems where the cost is assumed to be symmetric.

1.5.1 Problem Complexity

A naive approach to minimize stretch is to find the required objects by flooding the object requests to the whole network. All nodes which have objects will reply and the node which issued the request can choose the object of its interest. Clearly, this approach is inefficient. Alternatively, if all object information is stored at a specific node (e.g., the sink), no flooding is needed. But, whenever an object is moved from one node to another node, the information at sink needs to be updated, which might be a major bottleneck. Moreover, when objects move frequently, abundant messages will be generated for the information update at the sink. Therefore, the approach should be such that flooding and sink update at all times in not required. Moreover, flooding approach should not be used as flooding makes congestion in each edge proportional to the number of object requests.

1.6 Motivation and Objective

The efficiency of the TM systems relies on the good performance of the transaction scheduling algorithms [59, 74, 75, 117] and it is of great importance to design transaction scheduling algorithms which scale gracefully with the size and complexity of the system (i.e. when the number of cores in a multiprocessor chip increases). The objective of this dissertation is to design scalable algorithms for transactional scheduling in tightly-coupled, distributed, and NUMA systems. We focus primarily on theoretical foundations and present experimental evaluations as well, when deemed necessary.

1.6.1 Tightly-Coupled Systems

In the tightly-coupled TM model where performance is analyzed in terms of the number of shared resources, Attiya et al. [9] provided the best known general formal competitive ratio bound of $\mathcal{O}(s)$, where s is the number of shared resources. In this particular model, they also proved a matching lower bound of $\Omega(s)$ in the competitive ratio. When the number of resources s increases,

the performance degrades linearly. A difficulty in obtaining better competitive ratios is that the algorithms studied in the literature [9, 11, 45, 46, 57, 59, 70, 119] apply to the *one-shot scheduling problem*, where each thread issues a single transaction. One-shot problems are directly related with vertex coloring (Section 1.4.2), where the problem of determining the chromatic number of a graph is reduced to finding an optimal time schedule for the one-shot problem. Since it is known that computing an optimal coloring given complete knowledge of the graph is a very hard problem to approximate, the one-shot problem is very hard to approximate too [84].

On the one hand, if we consider scenarios where each thread issues many transactions in sequence over time (i.e., the *multi-shot scheduling problem*), the competitive ratio degrades by a factor of the maximum number of transactions among sequences, in the worst-case, when applying the one-shot scheduling algorithms. A natural question which we address in this dissertation is whether there are alternative models for multi-shot scheduling problems which have the potential to improve the trivial competitive bounds obtained using the one-shot scheduling algorithms. As we show in Chapter 3, it is indeed possible to obtain new and alternative performance bounds (within a poly-log factor of $\mathcal{O}(s)$) for multi-shot scheduling problems.

On the other hand, we are interested to address the question "is it possible to obtain better than $\mathcal{O}(s)$ competitive ratio for the one-shot scheduling problem?" Note that in the non-clairvoyant job scheduling model used by Attiya et al. [9] there are matching upper and lower bounds of $\mathcal{O}(s)$ and $\Omega(s)$, respectively. We answer the aforementioned question affirmatively in Chapter 4 that it is indeed possible to obtain better than $\mathcal{O}(s)$ (i.e. sub-linear) competitive ratios for the one-shot scheduling problem of Section 1.4 by just introducing two additional fairly minimalistic assumptions.

1.6.2 Large-Scale Distributed Systems

In distributed networked systems processors that are placed in the nodes of a network communicate through a message passing environment, in contrast to tightly-coupled architectures where communication latency is not considered. Some distributed *cache-coherence* (DTM) mechanism should

ensure that shared objects remain *consistent* while executing transactions in distributed TM implementations, i.e., writing to an object automatically *locates* and *invalidates* other cached copies of that object. A DTM protocol typically supports three kinds of operations: (i) *publish* operation which allows a node which created an object in its memory space to publish it so that other nodes in the network can find it; (ii) *lookup* operation, the protocol should locate the current copy of the object and move it to the requesting node's cache (*shared* access), without modifying the old copy; (iii) *move* operation, where a transaction attempts to access an object to update explicitly the DTM protocol should locate the current cached copy of the object and move it to the requesting node's cache invalidating the old copy.

The distributed networked systems typically do not come with built-in protocols that can be extended to provide required cache-coherence, so TM implementations in large-scale distributed systems require building something equivalent [77]. The conflicts between transactions running at different nodes can be handled using the scheduling algorithms designed for tightly-coupled systems. Therefore, one of our goal will be to design scalable and efficient cache-coherence algorithms for TM implementations in distributed systems. We will focus after that on whether makespan and cost to provide cache-coherence can be minimized simultaneously.

Previous cache-coherence approaches, Arrow [43], Relay [143], Combine [10], and Ballistic [77], were only for either specific network topologies or they do not scale well in arbitrary network topologies. The objective is to design a DTM protocol that is suitable for arbitrary network topologies. The goal is to devise a consistency protocol which ensures that the shared object requests by the transactions (running on some particular nodes of the network) are served with minimum overhead in any arbitrary network. We answer this question in Chapters 5 and 6 by presenting and analyzing a DTM protocol that is suitable for arbitrary (general) network topologies.

1.6.3 NUMA Systems

For NUMA systems, we are interested in minimizing the communication cost, makespan, and also the network load while executing transactions. In this direction, we present and analyze a

distributed consistency algorithm in Chapter 7 that minimizes simultaneously the communication cost and the network load in accessing the memory locations of the shared objects. After that we provide a trade-off between makespan and cost to provide cache-coherence; in particular, we show in Chapter 8 that both metrics can not be minimized simultaneously.

1.7 Dissertation Contributions

We now discuss the contributions of this dissertation in detail. Some of these contributions are published in journals and conferences [121–130].

1.7.1 Tightly-Coupled Systems

We made the following two contributions for transaction scheduling in tightly-coupled systems.

- We propose *execution window model* of transactions with M threads and N transactions per thread by extending the original one-shot model of M transactions with one transaction per thread. We then present, formally analyze, and experimentally evaluate three scheduling algorithms that are suitable for execution window model. The first algorithm Offline-Greedy produces a schedule of length $\mathcal{O}(\tau \cdot (C + N \cdot \log(MN)))$ with high probability, where τ denotes execution time duration of each transaction and C denotes the number of transactions inside the window a transaction conflicts with. The second algorithm Online-Greedy produces a schedule of length that is only a $\mathcal{O}(\log(NM))$ factor worse than Offline-Greedy. The third algorithm Adaptive-Greedy is the adaptive version of the previous algorithms which produces a schedule of length asymptotically the same as with online algorithm by adaptively guessing the value of C. All of the algorithms exhibit competitive ratio very close to $\mathcal{O}(s)$, where s is the number of shared resources, and at the same time, our algorithms provide new non-trivial tradeoffs for greedy transaction scheduling that parameterize window sizes and transaction conflicts within the execution window. We evaluate these window-based algorithms experimentally using the sorted link list, red-black tree, skip list,

and vacation benchmarks. The evaluation results confirm their benefits in practical performance throughput and other metrics such as aborts per commit ratio and execution time overhead, along with the non-trivial provable properties of the algorithms.

- We propose *balanced workload model* by again extending the original one-shot model such that if a transaction is writing, the number of write operations it performs is a constant fraction of its total reads and writes. We then present and analyze two new polynomial time scheduling algorithms that achieve sub-linear competitive bounds. In particular, the first algorithm Clairvoyant is $\mathcal{O}(\sqrt{s})$-competitive and the second algorithm Non-Clairvoyant is $\mathcal{O}(\sqrt{s} \cdot \log n)$-competitive, with high probability We also prove that the performance of Clairvoyant is close to optimal, since there is no polynomial time contention management algorithm for the balanced transaction scheduling problem that is better than $\mathcal{O}((\sqrt{s})^{1-\epsilon})$-competitive for any constant $\epsilon > 0$, unless NP $\subseteq$ ZPP.

1.7.2 Large-Scale Distributed and NUMA Systems

We made the following two contributions for TM implementation in large-scale distributed systems.

- We present and analyze Spiral, a novel DTM algorithm for transaction scheduling which guarantees an $\mathcal{O}(\log^2 n \cdot \min\{\log n, \log D\})$ stretch for object requests in general networks, where n is the number of nodes and D is the diameter of the network. It also guarantees poly-log approximation for lookup requests. To the best of our knowledge, this is the first consistency protocol for distributed transactional memory that achieves poly-log approximation in general networks.

- We present a framework to analyze DTM algorithms when object requests are generated at arbitrary moments of time and give stretch bounds for several DTM algorithms, including Spiral.

The bandwidth of the network is usually the major bottleneck, especially in NUMA systems. In the context of DTM algorithms, previous approaches including Spiral [10, 43, 77, 129, 143] were only for different network topologies with stretch bounds (Table 2.2) and they do not control the congestion. Therefore, we made the following contribution for TM implementation in NUMA systems.

- We present and analyze Multibend, a novel DTM algorithm suitable for NUMA systems in the sense that it minimizes congestion as well as stretch for d-dimensional mesh topologies. Recall that mesh topologies are widely used in high performance parallel and distributed computing. Particularly, for any set of object operations, Multibend achieves congestion approximation of $\mathcal{O}(d^2 \cdot \log n)$ and stretch of $\mathcal{O}(d \cdot \log n)$, where n is the number of nodes of the mesh; the congestion approximation is optimal with in a constant factor and stretch is optimal with a $\mathcal{O}(\log \log n)$ factor for constant d.

We then consider transaction scheduling in both distributed and NUMA systems. We focus on whether makespan and communication cost can be simultaneously minimized. This minimization is very important to schedule transactions with multiple objects with scalable performance. We made the following contribution in this aspect for transaction scheduling in distributed and NUMA systems.

- We show that there are transaction scheduling problem instances in distributed and NUMA systems such that makespan and communication cost can not be simultaneously minimized. This result justifies our study of DTM protocols in the sense that for the transaction scheduling in these systems these two optimization problems should be independently minimized. We then present and analyze algorithms that independently minimize either the makespan or the communication cost and achieve near-optimal bounds.

1.8 Dissertation Organization

This rest of the dissertation is organized as follows: In Chapter 2, we discuss the related work on transaction scheduling in tightly-coupled, large-scale distributed , and NUMA systems. We then discuss results related to transaction scheduling for tightly-coupled systems in Chapters 3 and 4. Chapter 3 is dedicated to the execution window model, the algorithms, and evaluation results. In Chapter 4, we introduce the balanced workload model and provide two algorithms that achieve sub-linear bounds.

We then discuss results related to transaction scheduling in distributed networked systems and NUMA systems in Chapters 5, 6, 7, and 8. Recall that, in distributed networked and NUMA systems, transactions are scheduled and conflicts are resolved in each network node using a globally-consistent scheduling algorithm similar to the one that is designed for TM implementation in tightly-coupled systems. Therefore, the focus of our transaction scheduling work for distributed networked and NUMA systems will be on how to find the shared objects needed by a transaction efficiently from the remote nodes and provide consistency of the objects after transactions commit and abort.

Chapter 5 is dedicated to the general network model and results on that model. We present an framework to analyze consistency algorithms for any arbitrary execution of requests that arrive in arbitrary moments of time in Chapter 6. In Chapter 7, we present a load balanced consistency algorithm that is suitable for NUMA systems. In Chapter 8, we show that there are transaction scheduling problem instances in distributed and NUMA systems such that makespan and communication cost can not be simultaneously minimized. This result justifies our study of consistency algorithms that only minimize the communication cost (not the execution time). Finally, we conclude this dissertation with a short discussion and an outline of possible future research directions in Chapter 9.

Chapter 2
Literature Review

In this chapter, we provide an overview of the literature on transaction scheduling in tightly- coupled, distributed, and NUMA systems. We start with describing the related work in the literature, and then present high level details of our work on obtaining new efficient scheduling algorithms and their experimental evaluations.

2.1 Tightly-Coupled Systems

Tightly-coupled systems represent the typical scenario of a multicore chip with multilevel cache organization, where the lower level caches are distinct to each processor, while the highest level cache is common to all the cores in the chip (see left of Fig. 1.5). Communication costs between the processors are symmetric. In 2003, Herlihy, Luchangco, Moir, and Scherer III [75] proposed Dynamic STM (DSTM) for dynamic-sized data structures. They give experimental results in DSTM using Polite, Aggressive, and Simple Locking contention management mechanisms on IntSetSimple, IntSetRelease, and red-black tree benchmarks, and conclude that choice of a contention management algorithm can significantly affect the transaction throughput and some contention manager that exhibit good performance at some benchmarks may not achieve the same performance result at other benchmarks. Scherer III and Scott [117] propose and analyze different contention management policies considering visible and invisible versions of read accesses, and different benchmarks that vary in complexity, level of contention, and mix of reads and writes. Their analysis of the throughput results reveals that choice of a contention manager is crucial for

the performance throughput in different benchmarks. They conclude that Polka generally gives good overall performance in most of the benchmarks even though it has no provable properties.

Most of the algorithms proposed in the literature [6, 45, 58, 75, 107, 117, 118, 142] for the transaction scheduling problem (Section 1.4) have been assessed only experimentally by using specific benchmarks. Guerraoui *et al.* [59] were the first to develop a scheduling algorithm which exhibits non-trivial provable worst-case guarantees along with good practical performance. Their Greedy scheduling algorithm decides in favor of old transactions using timestamps and achieves $\mathcal{O}(s^2)$ competitive ratio in comparison to the *optimal* off-line scheduling algorithm for n concurrent transactions that share s resources, and at the same time has good empirical performance. They argue that this bound holds for any algorithm which ensures the *pending commit* property (see Definition 1). They experimented with Greedy in DSTM [75] using the list and red-black tree benchmarks and concluded that it achieves performance comparable to other scheduling algorithms like Polka [117] and Aggressive [118] along with its provable worst-case guarantees. the model used in Guerraoui *et al.* [59] is based on the model suggested by Garey and Graham [54] for multiprocessor scheduling under resource constraints. Later, Guerraoui *et al.* [57] studied the impact of transaction failures on transaction scheduling. They presented the algorithm FTGreedy and proved an $\mathcal{O}(k \cdot s^2)$ competitive ratio when some running transaction may *fail* at most k times and then eventually commits. A transaction is called failed when it encounters an illegal instruction producing a segmentation fault or experiences a page fault resulting to wait for a long time for the page to be available [57].

Several other algorithms have also been proposed for the efficient transaction scheduling and the performance of some of them has been analyzed formally [9, 11, 46, 119]. The detailed comparison of the results and their properties are listed in Table 2.1. Attiya *et al.* [9] improved the competitive ratio of Greedy to $\mathcal{O}(s)$ and of FTGreedy to $\mathcal{O}(k \cdot s)$, and proved a matching lower bound of $\Omega(s)$ ($\Omega(k \cdot s)$ when transactions may fail) for any deterministic *work-conserving* algorithm which schedules as many transactions as possible (by choosing a maximal independent set of transactions at each time step). The model used in Attiya *et al.* [9] is the *non-clairvoyant* job

Table 2.1: Comparison of transaction scheduling algorithms, where C denotes the number of conflicts and it can be as much as the number of shared resources s, k denotes the number of times a transaction can fail, M denotes the number of different threads (or cores), and N denotes the number of transactions in each thread. The assumptions of the algorithms for the failure-free case are applied also to their versions for transaction failures.

Algorithm	**Model**	**Competitive ratio**	**Deterministic/ Randomized**	**Assumptions**
Serializer [45], ATS [142]	One-shot	$\Theta(\min\{s, M\})$ [9, 46]	Deterministic	-
Polka [117], Size-Matters [107]	One-shot	$\Omega(\min\{s, M\})$ [9, 119]	Deterministic	-
Restart [46], SoA [6]	One-shot	$\Theta(\min\{s, M\})$ [9, 11]	Deterministic	-
Greedy [59]	One-shot	$\mathcal{O}(s^2)$ [59]	Deterministic	Unit length transactions
FTGreedy [57]	One-shot	$\mathcal{O}(k \cdot s^2)$ [57]	Deterministic	Transactions can fail
Greedy [59]	One-shot	$\Theta(s)$ [9]	Deterministic	-
FTGreedy [57]	One-shot	$\Theta(k \cdot s)$ [9]	Deterministic	Transactions can fail
Phases [9]	One-shot	$\mathcal{O}(\max\{s, k \log k\})$ [9]	Randomized	Unit length transactions
RandomizedRounds [119]	One-shot	$\mathcal{O}(C \cdot \log M)$ [119]	Randomized	Equal length transactions
CommitRounds [119]	One-shot	$\mathcal{O}(\min\{s, M\})$ [9, 119]	Deterministic	Equal length transactions
Bimodal [11]	One-shot	$\Theta(s)$ [11]	Deterministic	Bimodal workloads
Offline-Greedy (Ch. 3)	Window	$\mathcal{O}(s + \log(MN))$ [125]	Randomized	Equal length transactions; conflict graph is known
Online-Greedy (Ch. 3)	Window	$\mathcal{O}(s \cdot \log(MN) + \log^2(MN))$ [125]	Randomized	Equal length transactions; conflict graph is not known
Offline-Greedy (Ch. 3)	Window	$\mathcal{O}(k \cdot (s + \log(MN)))$ [125]	Randomized	Transactions can fail
Online-Greedy (Ch. 3)	Window	$\mathcal{O}(k \cdot (s \cdot \log(MN) + \log^2(MN)))$ [125]	Randomized	Transactions can fail
Clairvoyant (Ch. 4)	One-shot	$\mathcal{O}(\sqrt{s})$ [123]	Deterministic	Balanced workloads
Non-Clairvoyant (Ch. 4)	One-shot	$\mathcal{O}(\sqrt{s} \cdot \log M)$ [123]	Randomized	Balanced workloads
Clairvoyant (Ch. 4)	One-shot	$\mathcal{O}(k \cdot \sqrt{s})$ [123]	Deterministic	Transactions can fail
Non-Clairvoyant (Ch. 4)	One-shot	$\mathcal{O}(k \cdot \sqrt{s} \cdot \log M)$ [123]	Randomized	Transactions can fail

scheduling model, suggested by Motwani et al. [101], in the sense that it requires no prior knowledge about the transactions while they are executed. They also gave a randomized scheduling algorithm Phases that achieves $\mathcal{O}(\max\{s, k \log k\})$ competitive ratio for the special case of unit length transactions in which a transaction may fail at most k times before it eventually commits. Schneider and Wattenhofer [119] proposed an algorithm, called RandomizedRounds, which produces a $\mathcal{O}(C \cdot \log M)$-competitive schedule with high probability, for the transaction scheduling problem with C conflicts (assuming unit delays for transactions). They also gave a deterministic algorithm CommitRounds with $\mathcal{O}(\min\{s, M\})$ competitive ratio. The model used in Schneider and Wattenhofer [119] is based on the degree of a transaction (i.e., neighborhood size) in the conflict graph of transactions.

While previous studies, e.g. [117], showed that contention managers Polka [117] and SizeMatters [107] exhibit good overall performance in variety of benchmarks, Schneider and Wattenhofer's work [119] showed that they may perform exponentially worse than their RandomizedRounds algorithm from the worst-case perspective. Another recent proposal for the contention management is Serializer [45], which resolves a conflict by removing a conflicting transaction T from the processor core where it was running, and scheduling it on the processor core of the other transaction to which it conflicted with.

Later, Attiya *et al.* [11] proposed a $\Theta(s)$-competitive algorithm for the one-shot scheduling problem in bimodal workloads. A workload is called *bimodal* if it contains only early-write and read-only transactions; a transaction is called *early-write* if the time from its first write access until its completion is at least half of its duration [11]. The model in [11] is also *non-clairvoyant* in the sense that it requires no prior knowledge about the transactions while they are executed. Hasenfratz *et al.* [70] studied different schedulers to adapt the load in STM systems based on contention. Hasenfratz et al. [70] also showed the performance improvement of these strategies by comparing their throughput with the existing contention management policies (e.g., Karma, Timestamp, Polka) which can not perform load adaption. Schneider and Wattenhofer [119] proved that the scheduling algorithms Polka [117] and SizeMatters [107] are $\Omega(M)$-competitive.

Attiya and Milani [11] showed that Steal-on-Abort (SoA) [6] and Serializer [45] algorithms are $\Omega(M)$-competitive. In SoA, the aborted transaction is given to the opponent transaction and queued behind it, preventing the two transactions from conflicting again. Moreover, Dragojević *et al.* [46] proved that Serializer [45] and adaptive transaction scheduling (ATS) [142] algorithms are $\mathcal{O}(M)$-competitive. ATS is the transaction scheduler which measures adaptively the contention intensity of a thread, when the contention intensity increases beyond a threshold, it serializes the transactions. Shrink and Restart due to Dragojević *et al.* [46] are the scheduling algorithms which predict the future accesses of a transaction based on the past accesses, and dynamically serializes transactions based on the prediction to prevent conflicts. They are also shown to be $\mathcal{O}(M)$-competitive. Attiya *et al.* [9] proved that every deterministic scheduling algorithm is $\Omega(s)$-competitive. Combining all these results, we obtain the bounds listed in Table 2.1 for these algorithms.

The transaction scheduling problem is also studied in several other papers, e.g. [17, 23–25, 97, 116, 132], for TM implementations in both hardware and software. However, they do not provide the formal analysis and the performance of their techniques is evaluated through benchmarks only.

We provided novel techniques and bounds in [121–123, 125] for the formal performance analysis of transaction scheduling algorithms with respect to the makespan in tightly-coupled systems. At the same time we have evaluated the performance of the scheduling algorithms experimentally for other performance metrics, such as performance throughout, as well. We provide two main scheduling models, the *balanced workload* model and the *window-based execution* model. Both of these models aim at improving the previous formal bounds relating the makespan performance of TM to the number of resources. In the balanced workload model, we give sub-linear bounds with respect to the number of resources for a simple restricted version of the one-shot scheduling problem of Section 1.4. In the window-based model, we actually give an alternative bound based on the metric of conflict number C which may be smaller than the number of resources for an extended version of the one-shot problem of Section 1.4. In this proposal, we use the analysis modeling and techniques based on the degree estimation of a transaction in the conflict graph similar to [119] for

transactional contention management in tightly-coupled shared memory architectures (particularly, Chapters 3 and 4).

2.2 Large-Scale Distributed Systems

Distributed networked systems represent the scenario of completely decentralized distributed shared memory where processors are placed in a network which communicate through a message passing environment. Here, the network is represented with an arbitrary weighted graph $G = (V, E, w)$, where V is the set of nodes (machines), E is the set edges (interconnection links between machines), and w is a weight function in E which reflects physical distances and delays. This model is more abstract than the hierarchical multilevel cache, because the network could be any arbitrary topology not restricted to any specific multiprocessor architecture. Thus, it models distributed networks over large areas. To solve the transaction scheduling problem in distributed systems, nodes need to use a transaction scheduling algorithm to resolve conflicts that arise while executing transactions. To support transaction scheduling satisfying atomicity, each node is enriched with a *transactional memory proxy* module that interacts with the local node and also with the proxies at other nodes [77]. The proxy module is asked to open the shared object when it is needed for reading or writing by a transaction. The proxy module checks whether the object is at the local cache, otherwise it calls an appropriate algorithm to get that object from the node that has it. At the commit time of a transaction, proxy checks whether any object that is read and written by that transaction was not invalidated by other transactions that are committed from other nodes. If that is the case, the proxy asks the transaction to abort, otherwise it allows the transaction to commit. The aborted transactions restart their execution and try to commit again.

When the proxy module of a node receives a request (from a remote node) for the shared object that is at the local node, it checks whether a local pending transaction using it. If the object is in use, the proxy can give the object to the requester aborting the local transaction or delay the response for a while so that local transaction can commit. This decision is done through the scheduling algorithm used in the nodes.

Several researchers [26, 38, 87, 98] presented techniques to implement TM in distributed networked systems. Manassiev *et al.* [98] presented the lazy conflict detection and handling algorithm based on global lock. Kotselidis *et al.* [87] presented the serialization/multiple lease based algorithm. Bocchino *et al.* [26] and Couceiro *et al.* [26, 38] presented the commit-time broadcasting based algorithm. Control-flow based distributed TM implementation is studied by Saad and Ravindran [114]. Romano *et al.* [111] discussed the use of the TM programming model in the context of the cloud computing paradigm and posed several open problems. Kim and Ravindran [85] studied transaction scheduling in replicated *data-flow* based distributed TM systems. Saad and Ravindran [114] provided a Java framework implementation, called HyFlow, for distributed TM systems. Recently, Hendler *et al.* [71] studied a lease based hybrid distributed software TM implementation 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.

As transactions are scheduled and conflicts are resolved using a scheduling algorithm in each network node, the focus in the TM implementation in distributed networked systems is on how to find the shared objects needed by transactions efficiently from the remote nodes and provide the *consistency* of the objects after transactions commit and abort. These previous algorithms [26, 38, 87, 98, 111] essentially try to provide consistency of the shared objects. However, they either use global lock, serialization lease, or commit-time broadcasting technique which do not scale well with the size of the network [10]. Moreover, they do not provide the formal analysis of the cost incurred by their algorithms to support distributed transaction scheduling and the performance of these techniques are evaluated through experiments only. Thus, it is of great importance to design consistency algorithms that scale well with the size, complexity, and network kind of the distributed systems, and also provide reasonable theoretical and empirical performance.

We now provide an overview of the work on designing scalable consistency algorithms for supporting TM in distributed networked systems. Herlihy and Sun [77] proposed Ballistic consistency algorithm. This algorithm is *hierarchial*: network nodes are organized as clusters at different levels. They evaluated the formal performance of Ballistic by its *stretch* (i.e., the competitive ratio on

Table 2.2: Comparison of consistency algorithms, where $S_{ST} = \mathcal{O}(D)$ is the stretch of the spanning tree, $S_{OT} = \mathcal{O}(D)$ is the stretch of the overlay tree, $l \leq n$ is the number of *move* operations in one-shot executions, n is the number of nodes, and D is the diameter of the network.

Algorithm	StretchStretch			**Network**	**Runs on**
	sequential	one-shot	dynamic		
Arrow [43]	$\mathcal{O}(S_{ST})$ [43]	$\mathcal{O}(S_{ST} \cdot \log l)$ [78]	$\mathcal{O}(S_{ST} \cdot \log D)$ [89]	General	Spanning tree
Relay [143]	$\mathcal{O}(S_{ST})$ [143]	$\mathcal{O}(S_{ST} \cdot \log l)$ [78]	$\mathcal{O}(S_{ST} \cdot \log D)$ [144]	General	Spanning tree
Combine [10]	$\mathcal{O}(S_{OT})$ [10]	$\mathcal{O}(S_{OT} \cdot \log l)$ [78]	$\mathcal{O}(S_{OT} \cdot \log D)$ [89]	General	Overlay tree
Combine [10]	$\mathcal{O}(\log D)$ [126]	$\mathcal{O}(\log D)$ [126]	$\mathcal{O}(\log D)$ [126]	Constant doubling	Hierarchical directory (independent sets)
Ballistic [77]	$\mathcal{O}(\log D)$ [77]	$\mathcal{O}(\log D)$ [77]	$\mathcal{O}(\log D)$ [126]	Constant doubling	Hierarchical directory (independent sets)
Spiral (Ch. 5 & 6)	$\mathcal{O}(\log^2 n \cdot \min\{\log n, \log D\})$ [129]	$\mathcal{O}(\log^2 n \cdot \min\{\log n, \log D\})$ [129]	$\mathcal{O}(\log^2 n \cdot \min\{\log n, \log D\})$ [126]	General	Hierarchical directory (sparse covers)
MultiBend (Ch. 7)	$\mathcal{O}(d \log n)$ [124]	$\mathcal{O}(d \log n)$ [124]	$\mathcal{O}(d \log n)$ [126]	d-D mesh	Hierarchical decomposition of the mesh

distances): each time a node issues a request for a remote shared object, compute the ratio of the algorithm's communication cost for that request to the optimal communication cost for that request. The optimal communication cost is computed based on the shortest path distances between the requesting node and the node in which the request finds that object. In *constant doubling* networks, their algorithm achieves amortized $\mathcal{O}(\log D)$ stretch, where D is the diameter of the constant doubling network for non-overlapping (i.e., sequential) requests to locate and move a cached copy of an object from one node to another. In this algorithm, concurrent requests are synchronized by *path reversal*: when two requests meet at some intermediate node, the second request is diverted behind the first request.

The Arrow algorithm [43] originally designed for the distributed queuing problem can also be used as the consistency algorithm for TM in distributed systems [43, 79, 137]. It maintains a distributed queue using path reversal [102]. Zhang and Ravindran [143] proposed the Relay consistency algorithm. Both Arrow and Relay run on a *spanning tree*. In Relay, the pointers lead to the node that is currently holding the object and the pointers are changed only after the object moves from one node to another, like the tree-based mutual exclusion algorithm of Raymond [109]. Relay has stretch $\mathcal{O}(S_{ST})$ in sequential executions, where S_{ST} is the stretch of the pre-selected spanning tree ST. They also showed that Relay efficiently reduces the worst-case number of total abortions of transactions to $\mathcal{O}(M)$ in comparison to using Arrow [43, 109], which has an $\mathcal{O}(M^2)$ for M transactions requesting the same object. Recently, Attiya *et al.* [10] proposed Combine, which runs on an *overlay tree*, whose leaves are the computing nodes of the system. They claimed that Combine avoids *race conditions* (missing one concurrent request by another) of Ballistic and Relay by combining requests that overtake each other as they pass through the same node. Combine exhibits the stretch $\mathcal{O}(S_{OT})$ in sequential executions, where S_{OT} is the stretch of the embedded overlay tree OT. The stretch of Arrow, Relay and Combine may be as much as the diameter of the network. Kim and Ravindran [86] proposed a technique that improves the stretch of Relay to $\mathcal{O}(\log n)$ in bimodal workloads in the worst-case and $\Theta(\log(n-m))$ in the average-case, for n nodes and m reading transactions. Table 2.2 summarizes the properties of the consistency algorithms in all possible (sequential, one-shot, and dynamic) execution scenarios. In *sequential* executions, object requests do not overlap with each other, whereas object requests are issued at the same time in *one-shot* executions and no further requests occur. Object requests are issued in arbitrary moments of time in *dynamic* executions.

There have been endeavors analyzing the dynamic performance of distributed protocols that are based on pre-selected spanning trees. An analysis of the Arrow protocol [43] given in [72, 89] for an arbitrary set of (online) ordering requests generated over a period of time shows that Arrow is $\mathcal{O}(s \cdot \log D)$-competitive, where s and D, respectively, are the stretch and the diameter of the spanning tree on which Arrow operates. Note that s can be as large as D, as for example, in ring

networks, giving a competitive ratio $\mathcal{O}(D \cdot \log D)$, which is significantly larger than ours. The Arrow protocol, originally developed for distributed mutual exclusion [109], is one of the simplest distributed directory protocol based on spanning trees. Along the lines of Arrow, an analysis of the Relay protocol [143] is presented in [145], for dynamic (online) requests in the context of distributed transactional memory, and shown that Relay is $\mathcal{O}(s \cdot \log D)$-competitive, for a set of transactions that request the same object.

We present Spiral, a novel consistency algorithm, in Chapter 5. To the best of our knowledge, Spiral is the first consistency algorithm for TM in distributed systems that achieves poly-log approximation for stretch in general networks. Previous approaches, Arrow [43], Relay [143], Combine [10], and Ballistic [77], were only for either specific network topologies or they do not scale well in arbitrary network topologies. For example, Ballistic is only suitable for doubling-dimension metrics, which is not general enough to cover other network topologies; further, the spanning tree approach of Relay [143] does not perform well on trees that do not preserve the shortest path metric, as for example, in ring networks. Moreover, we present a framework for analyzing distributed consistency algorithms in Chapter 6 and provide stretch bounds for several algorithms in dynamic execution of shared object operations.

2.3 NUMA Systems

Multicore processor architectures provide interfaces that enable multicore chips to connect with each other through high speed interconnect communication links, in order to form larger size multiprocessor systems. An example is the Intel QuickPath Interconnect (QPI) [36] which is implemented in the Intel Pentium i7 Nehalem multicore architecture [37]. Fig. 1.6 illustrates an example organization of an interconnect multiprocessor system in a 3-dimensional grid. Such large scale architectures are suitable for high performance distributed and parallel computing. In IBM Blue Gene/L 65,000 nodes are interconnected as a $64 \times 32 \times 32$ 3-dimensional mesh or torus [2]. Recently, IBM Blue Gene/Q integrated a 5-dimensional torus [34]. Moreover, Cray XT5 [81] is also based on a similar multiprocessor organization. These configurations are known as Non-Uniform

Memory Access (NUMA) systems where the shared memory is distributed among various processors. There are various ways to ensure that the caches of the cores are coherent, such as snoopy bus algorithm, or a distributed directory organization. An important characteristic is the *NUMA factor* which is related to the difference in latency for accessing data from a local memory location as opposed to a non-local one.

Wang *et al.* [139] evaluated several STM implementations on a big SMP machine that uses cache coherent NUMA (ccNUMA) architecture. They concluded that latencies due to remote memory accesses is the key factor that influences STM performance. Lu *et al.* [94] proposed a latency-based scheduling algorithm with a forecasting-based conflict prevention method to improve the TM performance in NUMA systems. Kotselidis *et al.* [87] studied how to exploit STM on clusters. They concluded that the performance depends on network congestion. Blagodurov *et al.* [22] provided a case for NUMA-aware scheduling on multicore systems. However, they did not consider implementing transactions. Calciu *et al.* [30] designed a family of reader-writer lock algorithms tailored to NUMA architectures, extending the existing lock algorithms designed for UMA architectures.

For NUMA systems, we are interested in minimizing the communication cost, makespan, and also the network load while executing transactions. In this direction, we give a distributed consistency algorithm, Multibend in Chapter 7 that minimizes simultaneously the communication cost and the network load in accessing the memory locations of the shared objects. For achieving simultaneously low communication cost and low congestion (i.e., load balancing), we applied techniques from *oblivious routing* [28] on d-dimensional grid network topologies, with near optimal congestion while maintaining small stretch (competitive ratio on distances). In particular, we combined an oblivious routing algorithm approach with the Spiral algorithm of Chapter 5 to obtain the desired algorithm with poly-log approximation in stretch and poly-log approximation in congestion (with respect to optimal edge congestion). In small (constant) degree graphs, low edge congestion implies also low node congestion. The algorithm MultiBend presented in Chapter 7 demonstrates that such a construction with dual optimization in grids is feasible.

In Chapter 8, we show that for the transaction scheduling in distributed and NUMA systems, the execution time and communication cost can not be minimized simultaneously and one has to rely on algorithms which minimize either execution time or communication cost.

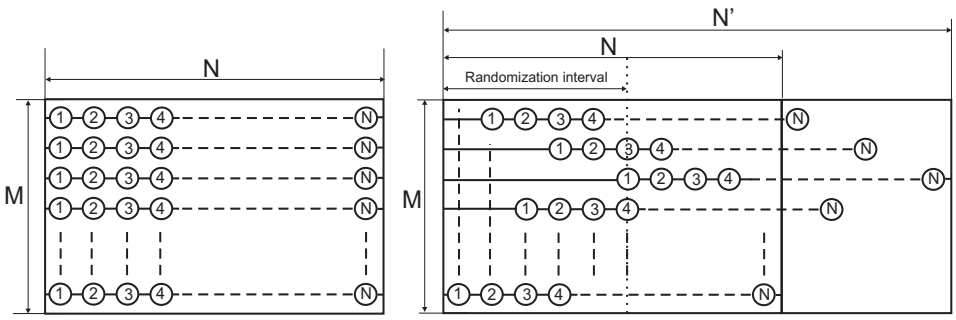
Chapter 3
Tightly-Coupled Systems: Execution Window Model

3.1 Introduction

In this chapter[1], in order to obtain non-trivial provable properties along with promising empirical performance, we consider the performance of program executions in *windows of transactions* (see Fig. 3.1a), which has the potential to overcome some of the limitations of the coloring reduction in certain circumstances. An $M \times N$ *window* W consists of M threads with an execution sequence of N different transactions per thread. The execution window W can be viewed as a collection of N one-shot transaction sets with M concurrent transactions in each set.

We show that we can obtain new and improved performance bounds for the multi-shot scheduling problem using window-based execution of transactions. We present and evaluate a family of window-based randomized greedy contention management algorithms where transactions are assigned priorities values, such that for some random initial interval in the beginning of the window W each transaction is in low priority mode and then after the random period expires the transactions switch to high priority mode. In high priority mode the transaction can only be aborted by other high priority transactions. The random initial delays have the property that the conflicting transactions are shifted inside their window and their execution times may not coincide (see

[1]This chapter published in:
Gokarna Sharma and Costas Busch. Window-Based Greedy Contention Management for Transactional Memory: Theory and Practice. *Distrib. Comput.* 25(3):225–248, 2012. http://link.springer.com/article/10.1007/s00446-012-0159-7



(a) Before execution (b) After execution

Figure 3.1: Execution window model for transactional memory.

Fig. 3.1b). The benefit is that conflicting transactions can execute at different time slots and potentially many conflicts are avoided. The benefits become more apparent in scenarios where the conflicts are more frequent inside the same column (i.e., simultaneously executed) transactions and less frequent between different column transactions. The experimental evaluation results on different benchmarks confirm the benefits of using window-based execution of transactions as an efficient contention management strategies in transactional memories.

The execution window model we consider here is useful in many real-world execution scenarios. The one prominent example is the scenario in which each thread needs to execute a job comprises of many transactions over time, i.e., a thread running on some processor creates $N \geq 1$ transactions $T_1, T_2, \cdots, T_N$ one after another and all of them are executed sequentially on the same processor core, i.e., T_i is executed as soon as T_{i-1} has finished execution and committed. In this multi-shot transaction scheduling scenario, the execution performance analysis based on the window model improves significantly over the trivial approach of using one-shot analysis.

3.1.1 Theoretical Contributions

We propose the contention measure C within the window to allow more precise statements about the worst-case complexity bound of any contention management algorithm, where C denotes the maximum number of conflicting transactions for any transaction in the window. As there are at most MN transactions in the window W, $C \leq MN$ when considering all the transactions. If we assume that all transactions have the same duration τ, then a straightforward upper bound for the

makespan of the window is $\tau \cdot \min(CN, MN)$, since $\tau \cdot CN$ follows from the observation that each transaction in a thread may be delayed at most C times by its conflicting transactions, and $\tau \cdot MN$ follows from the serialization of the transactions. The competitive ratio of the makespan using the one-shot analysis results is bounded by $\mathcal{O}(s \cdot N)$. This is because of the need of applying $\mathcal{O}(s)$-competitive algorithm of [9] N times, in the worst case, for the transactions in W. Similarly, using the one-shot Algorithm RandomizedRounds provided in [119] N times, the completion time is in the worst case $\mathcal{O}(\tau \cdot CN \cdot \log M)$.

We give three window-based randomized greedy algorithms for the contention management in any execution window W that perform significantly better than the trivial bounds mentioned above. For simplicity, we assume that each transaction has the same duration τ (this assumption can be removed; see Section 3.7). The first algorithm, Offline-Greedy, is tailored for environments where the conflict relations and the contention measure C on the shared resources are known in advance, while the second algorithm, Online-Greedy, is best suited to online scheduling environment where it is difficult to predict conflict relations. The third algorithm, Adaptive-Greedy, is the adaptive version of previous algorithms which assumes no knowledge of conflict relations and not even the conflict measure C.

Our first algorithm Offline-Greedy gives a schedule of length $\mathcal{O}(\tau \cdot (C + N \cdot \log(MN)))$ with high probability. An advantage of this schedule is that if the conflicts inside the window are bounded by $C \leq N \cdot \log(MN)$ then the schedule length is within a logarithmic factor from optimal, since $\tau \cdot N$ is a trivial lower bound in total execution time. This is a reasonable improvement over the trivial approach of using N one-shot executions from the worst-case perspective. We also show that this algorithm is $\mathcal{O}(s + \log(MN))$-competitive (for any choice of C). The algorithm is offline in the sense that it uses explicitly the conflict graph of the transactions (the global view of the system) at each time step of execution to resolve the conflicts. Moreover, as the analysis of this algorithm depends on transactions to be *deterministic* (i.e., if a transaction T conflicts with another transaction T', it will always conflict if they execute concurrently), it will not be able to handle *non-deterministic* transactions (i.e., transactions that change their execution and conflict

dependencies according to some value they read). This is because non-deterministic transactions define a conflict graph that may change over time.

Algorithm Offline-Greedy is appropriate for the broad class of *scheduling with conflicts* environments which generally arise in resource-constrained scheduling [54]. In such scheduling, a subset of transactions conflict if their cumulative demand for a resource exceeds the supply of that resource. Conflicts between transactions are modeled by a conflict graph [48], where nodes correspond to transactions and edges represent conflicts between transactions. A scheduling algorithm for these environments should know the set of transactions that conflict with each other at each time step to resolve conflicts. There are many applications of this type of scheduling environment which generate predictable conflict patterns with known conflict graphs, such as balancing parallel computation load, traffic intersection control, session management in local area networks, frequency assignment is cellular networks, and dining philosophers problem [14, 18, 27, 64, 82]. Conflict measure C is generally known in these applications because all transactions only need a constant amount of resources exclusively and each resource is required by a constant number of transactions [119]. We can take as an example the classical dining philosophers problem with n unit length transactions sharing s shared resources such that the transaction T_i demands only two resource R_i and $R_{(i+1) \bmod s}$ exclusively at any time.

Our second algorithm Online-Greedy produces a schedule of length $\mathcal{O}(\tau \cdot (C \cdot \log(MN) + N \cdot \log^2(MN)))$ with high probability. This is only a factor of $\mathcal{O}(\log(MN))$ worse schedule in comparison to Offline-Greedy. We also prove that this algorithm is $\mathcal{O}(s \cdot \log(MN) + \log^2(MN))$-competitive (for any choice of C). The benefit of the online algorithm is that it does not need to know the conflict graph of the transactions to resolve the conflicts. It takes decisions based on the local view of the system. Conflicts between transactions are resolved by randomized priorities. The algorithm uses as a subroutine a variation of algorithm RandomizedRounds [119]. Moreover, in contrast to Offline-Greedy, this algorithm will be able to handle non-deterministic transactions and its competitive ratio still holds if the execution of non-deterministic transactions keeps the maximum degree constantly at C despite committing transactions. However, if the execution of

non-deterministic transactions increases the maximum degree C by a factor of η the schedule length of this algorithm also increases by the same factor. Algorithm Online-Greedy is suitable for scheduling environments where conflicts are not known in advance and cannot be predicted ahead of time, and it is randomized. Transactional memory contention management is usually related to online scheduling, where the conflicts between two transactions are discovered on the fly when they access the same shared resource at any step of the execution. It is difficult to reliably predict conflicts in this scenario because of their changing behavior over time. The algorithms for online scheduling should resolve such dynamic conflicts without assuming conflict knowledge of transactions. The conflict measure C is generally bounded by the number of transactions for online scheduling problems in the worst-case.

The assumption about the known value of C in the previous algorithms is limited in the sense that their performance depends on the right choice of C. Our third algorithm, Adaptive-Greedy, is the adaptive version of the online algorithm which achieves similar worst-case performance even without the knowledge of contention measure C. It adaptively guesses the value of C starting from $C = 1$, and similar to Online-Greedy, this algorithm handles also the non-deterministic transactions.

We analyze the window-based algorithms assuming that N is uniform over all threads. For the transaction execution in the realistic scenarios, the assumption that N is uniform over all threads can be (somehow) limited. That is because different threads can have different number of transactions (not necessarily N). We note that our assumption of uniform N for all threads is for the analysis purpose only. As long as threads have at most N different transactions in sequence, the performance bounds of our algorithms hold without any changes. The technique we use for the analysis of these algorithms is similar to the one used by Leighton et al. [91] to analyze an *online packet scheduling* problem.

3.1.2 Practical Contributions

We implement the aforementioned window-based contention management algorithms and some of their variants. We used DSTM2 [74], which is an eager conflict management STM implementation, that has been modified to employ the random initial delays and frame based approach to execute transactions. The window-based algorithms are evaluated with four widely used benchmarks for transactional memories: sorted link list [75], red-black tree [75], skip list [104], and vacation from STAMP suite [31].

The evaluation results show that our window-based scheduling algorithms have a very reasonable performance throughput in different TM benchmarks, comparing to other scheduling algorithms used in practice. The performance comparison is with five widely known scheduling algorithms available in the literature: (i) Polka [117], the overall best performing contention manager, among the scheduling algorithms proposed in the literature, in most of the TM workloads (although it has no provable properties); (ii) Greedy [59], the first contention manager with provable theoretical and practical performance properties for one-shot scheduling problem; (iii) Priority [117], a simple static priority based contention manager; (iv) Serializer [45], a contention manager that is generally suitable for high contention scenarios; and (v) RandomizedRounds [119], a contention manager similar to Priority where priority of a transaction changes at every start and restart.

The conclusion from the evaluation results is that window-based scheduling algorithms achieve comparable performance with Polka, and outperform Greedy, Priority, Serializer, and RandomizedRounds in most of the benchmarks used in the experiments, sometimes by significant margins. The evaluation results confirm the benefits of our window-based scheduling algorithms in practical performance throughput and other transactional metrics such as aborts per commit ratio and execution time overhead. Moreover, we study the relation among the choice of the conflict measure C, the time step τ, and the size of the frames on the performance of window algorithms in different benchmarks in different amounts of contention. The results show that the impact of these parameters can be minimized using a novel technique of dynamic contraction and expansion of frames in the execution window model.

To summarize, our algorithms have comparable experimental performance to Polka, and at the same time, have provable theoretical performance guarantees. Therefore, our algorithms combine good characteristics from theory and practice. This is a very significant step toward designing scalable transactional memory schedulers that cope with the increased number of cores and system complexity in multi-core architectures.

3.1.3 Chapter Organization

The rest of the chapter is organized as follows: We present the execution window model for transactional memory in Section 3.2. We present and formally analyze three different randomized transaction scheduling algorithms in Sections 3.3, 3.4, and 3.5. We present the brief description of the algorithm variants and the benchmarks used in the experiments, and the evaluation results in Section 7.6. Section 3.7 concludes the chapter with some discussions.

3.2 Model and Preliminaries

Consider a set of at most $M \geq 1$ threads $\mathcal{P} := \{P_1, \cdots, P_M\}$. We consider a model that is based on an $M \times N$ execution window W consisting of a set of transactions $\mathcal{T}(W) := \{(T_{11}, \cdots, T_{1N}), (T_{21}, \cdots, T_{2N}), \ldots, (T_{M1}, \cdots, T_{MN})\}$, where each thread P_i issues N transactions $T_{i1}, \cdots, T_{iN}$ in sequence, so that T_{ij} is issued as soon as $T_{i(j-1)}$ has committed. This departs from the one-shot model given in Section 1.4 where $N = 1$ (one transaction per thread). However, similar to the one-shot model, transactions share a set of $s \geq 1$ shared resources $\mathcal{R}$, i.e. $\mathcal{R}(T_i)$ denotes the resources read or written by T_i. For the purpose of analysis, we assume that all transactions have the same execution time duration $\tau = \tau_{ij}$ and this time does not change over time.

Assuming that there is one transaction per thread, the conflict graph $\mathcal{G}(\mathcal{T}(W))$ can be used to obtain a simple greedy schedule of the transactions as follows. Compute a $C + 1$ vertex coloring of the conflict graph. All transactions of same color can commit simultaneously. The transactions can be scheduled in a greedy manner by giving a different priority to each transaction color. This produces a greedy schedule of length $makespan = \tau \cdot (C + 1)$. Since $C \leq \lambda_{\max} \cdot \gamma_{\max}$, we have

that $makespan \leq \tau \cdot (\lambda_{\max} \cdot \gamma_{\max} + 1)$. Further, since $C \geq \gamma_{\max} - 1$, $makespan_{\text{opt}} \geq \tau \cdot \gamma_{\max}$. Since $\lambda_{\max} \leq s$, the competitive ratio of the schedule is $\lambda_{\max} + 1 = \mathcal{O}(s)$.

3.3 Offline Algorithm

We present and analyze Algorithm Offline-Greedy (Algorithm 1), which is an offline greedy contention resolution algorithm in the sense that it uses the conflict graph explicitly to resolve conflicts of transactions. In addition to M and N, we assume that each thread P_i knows C_i, which denotes the maximum number of transactions that any transaction in P_i conflicts with; namely, using the conflict graph $G(\mathcal{T}(W))$, $C_i := \max_j \delta(T_{ij})$. Note that $C := \max_i C_i$.

Time is measured in discrete time steps, where each time step represents the duration τ of the transactions. We divide time into *frames* (see Fig. 3.2), which are time periods of duration $\Theta(\tau \cdot \ln(MN))$ (namely, each frame consists of $\Phi = \Theta(\ln(MN))$ time steps for the Offline-Greedy algorithm)[2]. Then, each thread P_i is assigned an initial random time period consisting of q_i frames as shown in Fig. 3.2, where q_i is chosen randomly, independently and uniformly from the range $[0, \alpha_i - 1]$, where $\alpha_i = C_i / \ln(MN)$. Moreover, each transaction $T_{kl}, 1 \leq l \leq N$, of each thread $P_k, 1 \leq k \leq M$, is assigned to frame $F_{kl} = q_k + (l - 1)$, which we call the *assigned frame* for T_{kl}. Each transaction has two priorities either of: low or $high$. Transaction T_{ij} is initially in low priority. Transaction T_{ij} switches to high priority in the first time step of frame $F_{ij} = q_i + (j - 1)$ (this is the assigned frame for T_{ij}) and remains in high priority thereafter until it commits. For example, the assigned frame for T_{23}, the third transaction of thread 2, is F_{23} as given in Fig. 3.2, which is the third frame after the random delay q_2 for thread 2. In the analysis, we show that with high probability each transaction commits in its assigned frame.

The priorities are used to resolve conflicts. A high priority transaction may only be aborted by another high priority transaction. A low priority transaction is always aborted if it conflicts

[2]Note that for the non-integral values of Φ, Φ', α_i, etc., we perform the rounding up to the smallest following integer, that is, $\Phi = \lceil \Theta(\tau \cdot \ln(MN)) \rceil$, $\Phi' = \lceil \Theta(\tau \cdot \ln^2(MN)) \rceil$, $\alpha_i = \lceil C_i / \ln(MN) \rceil$, etc. For reasons of clarity and simplicity, we do not explicitly show this rounding of non-integral values in algorithms description and analysis, as it does not affect their performance bounds.

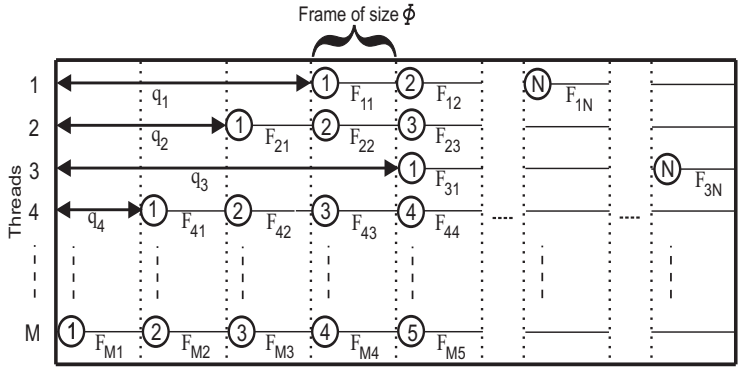


Figure 3.2: Illustration of initial random delays and frame based execution in window model.

Algorithm 1: Offline-Greedy

Input: An $M \times N$ window W of transactions with M threads, each with N transactions; Each thread P_i knows C_i, the maximum number of transactions in W that any transaction in P_i conflicts with; Each transaction has the same duration τ;

Output: A greedy execution schedule for the window of transactions W;

Divide time into frames consisting of $\Phi = 1 + (e^2 + 2) \cdot \ln(MN)$ time steps;
Each thread P_i chooses a random number $q_i \in [0, \alpha_i - 1]$ for $\alpha_i = C_i / \ln(MN)$;
Each transaction T_{ij} is assigned to frame $F_{ij} = q_i + (j - 1)$;
foreach *time step* $t = 0,\ 1 \cdot \tau,\ 2 \cdot \tau,\ 3 \cdot \tau, \ldots$ **do**
    Phase 1: Priority Assignment
    foreach *transaction* T_{ij} **do**
        if $t < F_{ij} \cdot \tau \cdot \Phi$ **then** $Priority(T_{ij}) \leftarrow 1\ (low)$; **else** $Priority(T_{ij}) \leftarrow 0\ (high)$;
    Phase 2: Conflict Resolution
    begin
        Let G_t be the conflict graph at time t;
        Compute G_t^H and G_t^L, the subgraphs of G_t induced by high and low priority nodes, respectively;
        Compute $I_H \leftarrow I(G_t^H)$, maximal independent set of nodes in graph G_t^H;
        $Q \leftarrow$ low priority nodes adjacent to nodes in I_H;
        Compute $I_L = I(G_t^L \setminus Q)$, maximal independent set of nodes in graph G_t^L after removing Q nodes;
        Commit $I_H \cup I_L$;

with a high priority transaction. Let G_t denote the conflict graph of transactions at time step t which evolves while the execution of the transactions progresses. Note that the maximum degree of G_t is bounded by C, but the effective degree between high priority transactions is lower. At each time step t we select to commit a maximal independent set of transactions in G_t. We first

select a maximal independent set I_H of high priority transactions, then remove this set and its neighbors from G_t, and then select a maximal independent set I_L of low priority transactions from the remaining conflict graph. The transactions that commit are $I_H \cup I_L$. As the maximal independent set at each time step can be computed in polynomial time by a simple distributed algorithm, e.g. Luby [95], the algorithm estimates the schedule in polynomial time.

The intuition behind the algorithm is as follows: consider a thread i and its first transaction in the window T_{i1}. According to the algorithm, T_{i1} becomes high priority in the beginning of frame F_{i1}. Because q_i is chosen at random among $C_i/\ln(MN)$ positions it is expected that T_{i1} will conflict with at most $\mathcal{O}(\ln(MN))$ transactions in its assigned frame F_{i1} which become simultaneously high priority in F_{i1}. Since a time frame contains $\Phi = \Theta(\ln(MN))$ time steps, transaction T_{i1} and all its high priority conflicting transactions will be able to commit by the end of time frame F_{i1}, using the conflict resolution graph. The initial randomization period of $q_i \cdot \Phi$ time steps will have the same effect to the remaining transactions of the thread i, which will also commit within their assigned frames.

3.3.1 Analysis of Offline Algorithm

We study the makespan and the competitive ratio of Algorithm Offline-Greedy. According to the algorithm, when a transaction T_{ij} is issued, it will be in low priority until the respective frame F_{ij} starts. As soon as F_{ij} starts, the transaction T_{ij} will begin executing in high priority (if it didn't commit already). Let A denote the set of conflicting transactions with T_{ij} in the conflict graph $G(\mathcal{T}(W))$. Let $A' \subseteq A$ denote the subset of conflicting transactions with T_{ij} which become high priority during frame F_{ij} (simultaneously with T_{ij}).

Lemma 3.3.1 *If $|A'| \leq \Phi - 1$ then transaction T_{ij} will commit in frame F_{ij}.*

Proof. Due to the use of the high priority independent sets in the conflict graph G_t, if in time t during frame F_{ij} transaction T_{ij} does not commit, then some conflicting transaction in A' must commit. Since there are at most $\Phi - 1$ high priority conflicting transactions, and the length of the frame F_{ij} is exactly equal to Φ time steps, T_{ij} will commit by the end of frame F_{ij}. □

Note that Lemma 3.3.1 holds even if we include in A' also transactions that become high priority before F_{ij}, but were still active in this frame. However, we do not consider these transactions in A' because this scenario occurs with very low probability (at most $(MN)^{-2}$) as we show below in Lemma 3.3.3 and all these scenarios are considered in Lemma 3.3.4. We show next that it is unlikely that $|A'| > \Phi - 1$. We use the following Chernoff bound:

Lemma 3.3.2 (Chernoff Bound 1) *Let $X_1, X_2, \ldots, X_n$ be independent Poisson trials such that, for $1 \leq i \leq n$, $\mathbf{Pr}(X_i = 1) = pr_i$, where $0 < pr_i < 1$. Then, for $X = \sum_{i=1}^{n} X_i$, $\mu = \mathbf{E}[X] = \sum_{i=1}^{n} pr_i$, and any $\delta > e^2$, $\mathbf{Pr}(X > \delta \cdot \mu) < e^{-\delta \cdot \mu}$.*

Lemma 3.3.3 *$|A'| > \Phi - 1$ with probability at most $(1/MN)^2$.*

Proof. Let $A_k \subseteq A$, where $1 \leq k \leq M$, denote the set of transactions of thread P_k that conflict with transaction T_{ij}. We partition the threads $P_1, \ldots, P_M$ into 3 classes Q_0, Q_1, and Q_2, such that:

- Q_0 contains every thread P_k which either $|A_k| = 0$, or $|A_k| > 0$ but the positions of the transactions in A_k are such that it is impossible to overlap with F_{ij} for any random intervals q_i and q_k.
- Q_1 contains every thread P_k with $0 < |A_k| < \alpha_i$, and at least one of the transactions in A_k is positioned so that it is possible to overlap with frame F_{ij} for some choices of random intervals q_i and q_k.
- Q_2 contains every thread P_k with $\alpha_i \leq |A_k|$. Note that $|Q_2| \leq C_i/\alpha_i = \ln(MN)$.

Let Y_k be a random binary variable, such that $Y_k = 1$ if in thread P_k any of the transactions in A_k becomes high priority in F_{ij} (same frame with T_{ij}), and $Y_k = 0$ otherwise. Let $Y = \sum_{k=1}^{M} Y_k$. Note that $|A'| = Y$. Denote $pr_k = \mathbf{Pr}(Y_k = 1)$. We can write $Y = Z_0 + Z_1 + Z_2$, where $Z_\ell = \sum_{P_k \in Q_\ell} Y_k$, for $0 \leq \ell \leq 2$. Clearly, $Z_0 = 0$. and $Z_2 \leq |Q_2| \leq \ln(MN)$.

Recall that for each thread P_k there is a random initial interval with q_k frames, where q_k is chosen uniformly at random in $[0, \alpha_k - 1]$. Given the random choice of P_k, $0 < pr_k \leq |A_k|/\alpha_i < 1$,

since there are $|A_k| < \alpha_i$ conflicting transactions in A_i and there are at least α_i random choices for the relative position of transaction T_{ij}. Consequently,

$$\mu = \mathbf{E}[Z_1] \;\; = \;\; \sum_{P_k \in Z_1} pr_k \leq \sum_{P_k \in Z_1} \frac{|A_k|}{\alpha_i} = \frac{1}{\alpha_i} \cdot \sum_{P_k \in Z_1} |A_k| \leq \frac{C_i}{\alpha_i} \leq \ln(MN).$$

By applying the Chernoff bound of Lemma 3.3.2 we obtain that

$$\mathbf{Pr}(Z_1 > (e^2+1) \cdot \mu) < e^{-(e^2+1)\cdot\mu} < e^{-2\cdot\ln(MN)} = (MN)^{-2}.$$

Since $Y = Z_0 + Z_1 + Z_2$, and $Z_2 \leq \ln(MN)$, we obtain

$$\mathbf{Pr}((|A'| = Y) > ((e^2+2) \cdot \mu = \Phi - 1)) < (MN)^{-2},$$

as needed. □

Lemma 3.3.4 *All transactions commit by the end of their assigned frames with probability at least* $1 - (MN)^{-1}$.

Proof. From Lemmas 3.3.1 and 3.3.3, Φ time steps do not suffice to commit transaction T_{ij} within its assigned frame F_{ij} with probability at most $(NM)^{-2}$ (we call this a *bad event*). Considering all the MN transactions in the window a bad event for any of them occurs with probability at most $MN \cdot (MN)^{-2} = (MN)^{-1}$. Thus, with probability at least $1 - (MN)^{-1}$, all transactions will commit within their assigned frames. □

Since $C := \max_i C_i$, the makespan bound of the algorithm follows immediately from Lemma 3.3.4.

Theorem 3.3.5 (Makespan of Offline-Greedy**)** *Algorithm* Offline-Greedy *produces a schedule of length* $\mathcal{O}(\tau \cdot (C + N \cdot \log(MN)))$ *with probability at least* $1 - (MN)^{-1}$.

Since in the conflict graph $G(\mathcal{T}(W))$, $C \leq \lambda_{\max} \cdot \gamma_{\max}$, we have that $makespan = \mathcal{O}(\tau \cdot (\lambda_{\max} \cdot \gamma_{\max} + N \cdot \log(MN)))$. Further, since $C \geq \gamma_{\max} - 1$ and $\tau \cdot N$ is a lower bound on the schedule

length, $makespan_{\text{opt}} \geq \tau \cdot \max(\gamma_{\max}, N)$. Therefore, the competitive ratio of the schedule is $\mathcal{O}(\lambda_{\max} + \log(MN)) = \mathcal{O}(s + \log(MN))$.

Corollary 3.3.6 (Competitive Ratio of Offline-Greedy**)** *The makespan of the schedule produced by Algorithm* Offline-Greedy *has competitive ratio* $\mathcal{O}(s + \log(MN))$ *with probability at least* $1 - (MN)^{-1}$.

3.4 Online Algorithm

A limitation of Algorithm 1 is that the conflict graph of the transactions is assumed to be known at each time step. We present and analyze Algorithm Online-Greedy (Algorithm 2) which removes this limitation. This algorithm is called online in the sense that it does not depend on knowing the dependency graph to resolve conflicts. In addition to M and N, we assume that each thread P_i knows C_i. This algorithm is similar to Algorithm 1 with the difference that in the conflict resolution phase we use as a subroutine a variation of Algorithm RandomizedRounds proposed by Schneider and Wattenhofer [119]. The makespan of the online algorithm is slightly worse than the offline algorithm, since the duration of the frame (the frame size), as shown in Fig. 3.2, is now $\Phi' = \mathcal{O}(\tau \cdot \ln^2(MN))$.

There are two different priorities associated with each transaction under this algorithm. The pair of priorities for a transaction T_{ij} is given as a vector $\langle \pi_{ij}^{(1)}, \pi_{ij}^{(2)} \rangle$, where $\pi_{ij}^{(1)}$ represents the Boolean priority value either of low or $high$ (with respective values 1 and 0) as described in Algorithm 1, and $\pi_{ij}^{(2)} \in [1, M]$ represents the random priorities used in Algorithm RandomizedRounds [119]. The conflicts are resolved in lexicographic order based on the priority vectors, so that vectors with lower lexicographic order have higher priority.

Conflicts are resolved as follows. When a transaction T_{ij} is issued, it starts to execute immediately in low priority ($\pi_{ij}^{(1)} = 1$) until the respective randomly chosen time frame F_{ij} starts where it switches to high priority ($\pi_{ij}^{(1)} = 0$). Once in high priority, the field $\pi_{ij}^{(2)}$ will be used to resolve conflicts with other high priority transactions. A transaction chooses a discrete number $\pi_{ij}^{(2)}$ uniformly at random in the interval $[1, M]$ on start of the frame F_{ij}, and after every abort. In case of a

Algorithm 2: Online-Greedy

Input: An $M \times N$ window W of transactions with M threads, each with N transactions; Each thread P_i knows C_i, the maximum number of transactions in W that any transaction in P_i conflicts with; Each transaction has the same duration τ;

Output: A greedy execution schedule for the window of transactions W;

Divide time into frames of $\Phi' = 16 \cdot e \cdot \Phi \cdot \ln(MN)$ time steps, where $\Phi = 1 + (e^2 + 2) \cdot \ln(MN)$;
Each thread P_i chooses a random number $q_i \in [0, \alpha_i - 1]$ for $\alpha_i = C_i / \ln(NM)$;
Each transaction T_{ij} is assigned to frame $F_{ij} = q_i + (j - 1)$;
Associate pair of priorities $\langle \pi_{ij}^{(1)}, \pi_{ij}^{(2)} \rangle$ to each transaction T_{ij};
foreach *time step* $t = 0,\ 1 \cdot \tau,\ 2 \cdot \tau,\ 3 \cdot \tau, \ldots$ **do**
    Phase 1: Priority Assignment
    foreach *transaction* T_{ij} **do**
        if $t < F_{ij} \cdot \tau \cdot \Phi'$ **then** Priority $\pi_{ij}^{(1)} \leftarrow 1\ (low)$; **else** Priority $\pi_{ij}^{(1)} \leftarrow 0\ (high)$;
    Phase 2: Conflict Resolution
    if $\pi_{ij}^{(1)} == 0$ *(T_{ij} has high priority)* **then**
        On (re)start of transaction T_{ij};
        $\pi_{ij}^{(2)} \leftarrow$ random integer in $[1, M]$;
    On conflict of transaction T_{ij} with transaction T_{kl};
    if $\pi_{ij}^{(1)} < \pi_{kl}^{(1)}$ **then** $abort(T_{ij}, T_{kl})$;
    else if $\pi_{ij}^{(1)} > \pi_{kl}^{(1)}$ **then** $abort(T_{kl}, T_{ij})$;
    else if $\pi_{ij}^{(2)} < \pi_{kl}^{(2)}$ **then** $abort(T_{ij}, T_{kl})$;
    else $abort(T_{kl}, T_{ij})$;
    // In case a transaction T_{ij} aborts T_{kl} because $\pi_{ij}^{(2)} < \pi_{kl}^{(2)}$, then when T_{kl} restarts it cannot abort T_{ij} until T_{ij} commits or aborts

conflict of a transaction T_{ij} with another transaction T_{kl}, if the Boolean priority value $\pi_{ij}^{(1)} < \pi_{kl}^{(1)}$, then T_{ij} aborts T_{kl}. If $\pi_{ij}^{(1)} > \pi_{kl}^{(1)}$, then T_{kl} aborts T_{ij}. If the Boolean priority value for both T_{ij} and T_{kl} is the same (this happens only when they are high priority at the same frame), then we use the random priority number of T_{ij} and T_{kl} to resolve conflict. If $\pi_{ij}^{(2)} < \pi_{kl}^{(2)}$, then the transaction T_{ij} proceeds and T_{kl} aborts; otherwise (in the case where $\pi_{ij}^{(2)} \not< \pi_{kl}^{(2)}$), the transaction T_{kl} proceeds and T_{ij} aborts. (The procedure $abort(T_{ij}, T_{kl})$ in Algorithm 2 aborts transaction T_{kl}.) Note also that when the aborted transaction T_{kl} restarts, it cannot abort T_{ij} until T_{ij} has been committed or aborted.

3.4.1 Analysis of Online Algorithm

In the analysis given below, we study the makespan, the response time, and the competitive ratio of Algorithm Online-Greedy. The analysis is based on the following adaptation of the response time analysis of a one-shot transaction problem with algorithm RandomizedRounds [119]. It uses the following Chernoff bound:

Lemma 3.4.1 (Chernoff Bound 2) *Let $X_1, X_2, \ldots, X_n$ be independent Poisson trials such that, for $1 \leq i \leq n$, $\mathbf{Pr}(X_i = 1) = pr_i$, where $0 < pr_i < 1$. Then, for $X = \sum_{i=1}^{n} X_i$, $\mu = \mathbf{E}[X] = \sum_{i=1}^{n} pr_i$, and any $0 < \delta \leq 1$, $\mathbf{Pr}(X < (1-\delta) \cdot \mu) < e^{-\delta^2 \cdot \mu/2}$.*

Lemma 3.4.2 (Adaptation from Schneider and Wattenhofer [119]) *Given a one-shot transaction scheduling problem with U transactions, the time span a transaction T needs from the moment it is issued until commit is $16 \cdot e \cdot (d_T + 1) \cdot \log U$ with probability at least $1 - \frac{1}{U^2}$, where d_T is the number of transactions conflicting with T.*

Proof. Consider the respective conflict graph G of the one-shot problem. Let N_T denote the set of conflicting transactions for T (these are the neighbors of T in G). Let $d_T = |N_T| \leq U$. Let y_T denote the random priority number choice of T in range $[1, U]$. The probability that for transaction T no transaction $K \in N_T$ has the same random number is:

$$\mathbf{Pr}(\nexists K \in N_T | y_T = y_K) = \left(1 - \frac{1}{U}\right)^{d_T} \geq \left(1 - \frac{1}{U}\right)^{U} \geq \frac{1}{e}.$$

The probability that y_T is at least as small as y_K for any transaction $K \in N_T$ is $\frac{1}{d_T+1}$. Thus, the chance that y_T is smallest and different among all its neighbors in N_T is at least $\frac{1}{e \cdot (d_T+1)}$. If we conduct $16 \cdot e \cdot (d_T + 1) \cdot \ln U$ trials, each having success probability $\frac{1}{e \cdot (d_T+1)}$, then the probability that the number of successes Z is less than $8 \cdot \ln U$ becomes:

$$\mathbf{Pr}(Z < 8 \cdot \ln U) < e^{-2 \cdot \ln U} = \frac{1}{U^2},$$

using the Chernoff bound of Lemma 3.4.1. □

Lemma 3.4.3 *In Algorithm* Online-Greedy *all transactions commit by the end of their assigned frames with probability at least* $1 - 2 \cdot (MN)^{-1}$.

Proof. According to the algorithm, a transaction T_{ij} becomes high priority ($\pi_{ij}^{(1)} = 0$) in frame F_{ij}. When this occurs the transaction will start to compete with other transactions having high priority. Lemma 3.3.3 from the analysis of Algorithm 1 implies that the effective degree of T_{ij} with respect to high priority transactions is $d_T > \Phi - 1$ with probability at most $(MN)^{-2}$ (we call this *bad event-1*). From Lemma 3.4.2, if $d_T \leq \Phi - 1$, the transaction will not commit within $16 \cdot e \cdot (d_T + 1) \cdot \log(MN) \leq \Phi'$ time slots with probability at most $(MN)^{-2}$ (we call this *bad event-2*). Therefore, T_{ij} does not commit in F_{ij} when either bad event-1 or bad event-2 occurs, which happens with probability at most $(MN)^{-2} + (MN)^{-2} = 2 \cdot (MN)^{-2}$. Considering now all the MN transactions, the probability of failure is at most $2 \cdot (MN)^{-1}$. Thus, with probability at least $1 - 2 \cdot (MN)^{-1}$, every transaction T_{ij} commits during the F_{ij} frame. □

The makespan and the competitive ratio of the algorithm follow immediately from Lemma 3.4.3.

Theorem 3.4.4 (Makespan of Online-Greedy**)** *Algorithm* Online-Greedy *produces a schedule of length* $\mathcal{O}(\tau \cdot (C \cdot \log(MN) + N \cdot \log^2(MN)))$ *with probability at least* $1 - 2 \cdot (MN)^{-1}$.

Corollary 3.4.5 (Competitive Ratio of Online-Greedy**)** *The makespan of the schedule produced by Algorithm* Online-Greedy *has competitive ratio* $\mathcal{O}(s \cdot \log(MN) + \log^2(MN))$ *with probability at least* $1 - 2 \cdot (MN)^{-1}$.

In the analysis above, we assumed that the effective degree d_T of a transaction T_{ij} (which becomes high priority in the beginning of frame F_{ij}) with respect to other high priority transactions in F_{ij} is known but it does not have the knowledge whether d_T is constant. In some special cases, the performance bounds of Algorithm Online-Greedy can be improved. Let us consider the classical dining philosophers problem [14] where d_T is constant (at most 2) for all transactions $T_{ij} \in \mathcal{T}(W)$

Algorithm 3: Adaptive-Greedy

Input: An $M \times N$ execution window W with M threads each with N transactions, where C is unknown;

Output: A greedy execution schedule for the window of transactions;

```
1 Code for thread P_i;
2 begin
3     Initial contention estimate C_i ← 1;
4     repeat
5         Online-Greedy(C_i, W);
6         if bad event then
7             C_i ← 2 · C_i ;
8     until all transactions are committed;
```

(irrespective of the value of C). This is because each shared resource is only required by a constant number of transactions and all transactions only need a constant amount of shared resource accesses exclusively. In such executions, the frame size of $\Phi' = \mathcal{O}(\tau \cdot \ln(MN))$ is sufficient for all the high priority transactions in F_{ij} to commit by the end of it, with high probability, and the Online-Greedy algorithm achieves the total makespan and the competitive ratio the same as Offline-Greedy.

3.5 Adaptive Algorithm

A limitation of Algorithms 1 and 2 is that the values C_i need to be known in advance for each thread P_i. We present the Algorithm Adaptive-Greedy (Algorithm 3) in which each thread can guess the individual values of C_i. The algorithm works based on the exponential back-off strategy used by many scheduling algorithms developed in the literature such as Polka [117].

Each thread P_i starts with assuming $C_i = 1$. Based on the current estimate C_i, the thread attempts to execute Algorithm 2, for each of its transactions assuming the window size $M \times N$. Now, if the choice of C_i is correct then each transaction of the thread P_i in the window W should commit by the end of the assigned frame in which it becomes high priority. Thus, all transactions of thread P_i should commit within the time estimate of Algorithm 2 which is $L_i = \mathcal{O}(\tau \cdot (C_i \cdot \log(MN) + N \cdot \log^2(MN)))$. However, if during L_i thread P_i is unable to commit one of its

transactions within its assigned frame (we call this a *bad event*), then thread P_i will assume that the choice of C_i is incorrect, and will start over again with the remaining transactions assuming $C'_i = 2 \cdot C_i$. Eventually thread P_i will guess the value of C'_i for the window W, such that the actual value C_i of the thread P_i is $C'_i/2 < C_i \leq C'_i$, and all its transactions will commit within their respective time frames. It is easy to see that the correct choice of $\lceil C_i \rceil$ will be reached by a thread P_i within $\log \lceil C_i \rceil$ iterations. The total makespan and the competitive ratio are asymptotically the same as with Algorithm 2.

3.6 Experimental Evaluation

The experimental evaluation aims to investigate the performance benefits of the window-based contention management algorithms by executing several benchmarks using different contention configurations (ranging from low contention to high contention). The platform used to execute benchmarks is a 2 x quad-core Intel Xeon Processor 2.4 GHz system with 6GB RAM and hyper-threading on (total 16 cores), running Ubuntu 10.04, and using Java 1.6.0_27. We perform our experiments in DSTM2 [74], an eager conflict management STM implementation, using the default shadow factory and visible reads. Experiments are executed with $M = 1$, 2, 4, 8, and 16 threads, and $N = 50$ transactions in sequence for a execution window, unless otherwise stated. We limit our experiments to maximum 16 concurrent threads, because, in practice, the platform we used for the experiments can not execute more than 16 threads concurrently without context switching. We run the experiments for 10 seconds and the data plotted are the average of 6 experiments.

DSTM2, like other STMs (e.g., TL2 [44], RSTM [99], TinySTM [52]), creates a number of threads that concurrently execute transactions. We extend this into a thread pool model by adding a thread-safe work queue java.util.concurrent.LinkedBlockingDeque to each thread. We use multiple work queues (one work queue per thread) to overcome significant serialization overhead when fetching the transactions using some locking mechanism from a single work queue only. The transactions submitted for execution are first distributed to work queues in a round robin manner.

Threads then acquire transactions from the head of their own queue when their current transaction commits.

The benchmarks used to evaluate our window-based algorithms are three simple benchmarks sorted link list [75], red-black tree [75], skip list [104], and a complex benchmark vacation from the STAMP suite [31]. Hereafter, we refer them as List, RBTree, SkipList, and Vacation, respectively for clarity and conciseness. The benchmarks are configured to generate different amounts of transactional conflicts (i.e., low contention to high contention scenarios) that facilitate us to evaluate the algorithms we proposed in this chapter. Particularly, we measure the experimental results using three different contention scenarios (i.e., amount of contention): (i) *Low contention* − each transaction needs to perform only 20% update operations; (ii) *Medium contention* − each transaction needs to perform 60% update operations, hence medium amount of contention; and (iii) *High contention* − each transaction needs to do 100% update operations, hence high contention. That is, increasing percentage of update operations increase significantly the contention probability among transactions.

We proceed with briefly describing each benchmark used in the experiments. The List benchmark transactionally inserts and removes random numbers into a sorted linked list. Similarly, the RBTree benchmark transactionally inserts and removes random numbers into a tree. The SkipList is a benchmark that stores a sorted list of items, using a hierarchy of linked lists that connect increasingly sparse subsequences of the items. The insertion and removal of an item in the SkipList is also done transactionally. List, RBTree, and SkipList are configured to perform randomly selected insertion and deletion of transactions with equal probability. Vacation is a benchmark from the STAMP suite which simulates a travel booking database with three tables to hold bookings for flights, hotels, and cars. Each transaction simulates a customer making several bookings, and thus several modifications to the database. High contention scenario is achieved by configuring Vacation to execute many transactions which perform large number of modifications to the travel booking database.

3.6.1 Algorithm Variants Used in Experiments

We now briefly describe the window-based algorithm variants used in the experimental evaluation (see Fig. 3.3 for their performance throughput in high contention scenarios). We did not use Offline-Greedy algorithm of Section 3.3 in the evaluation because it resolves conflicts based on the conflict graph, which requires global knowledge.

- Online: is the same algorithm described in Section 3.4.
- Online-Dynamic: is the improved version of Online algorithm where frames are dynamically contracted or expanded based on the amount of contention inside the frame (see Sections 3.6.2 and 3.6.5 for details).
- Adaptive: is same as the one described in Section 3.5.
- Adaptive-Improved: is the variant of Adaptive algorithm where the new contention measure value C_i' is calculated based on the contention intensity (CI) calculation similar to Yoo and Lee [142].
- Adaptive-Improved-Dynamic: is the variant of Adaptive-Improved where frames are dynamically contracted or expanded similar to Online-Dynamic.

Online and Online-Dynamic algorithms require to know the contention measure C_i (in addition to M and N) for each thread P_i to choose random initial delay of q_i frames. For the simplicity in the evaluation, we assume that $C = MN$ for each $P_i, 1 \leq i \leq M$, in any contention scenarios (see Section 3.6.5 for the study on the effect of this choice of C in Online and Online-Dynamic algorithms in medium and low contention scenarios) and P_i is assigned an initial random period consisting of q_i frames chosen randomly from the range $[0, \alpha - 1]$, where $\alpha = C/\ln(MN)$. For example, as we assumed $N = 50$, for the number of threads $M = 2$, $C = 100$ and $\alpha = \lceil 100/\ln(100) \rceil = 22$. That is, each thread P_i chooses randomly a number between 0 to 21 which gives the number of frames as initial random period for P_i. Moreover, we fix a timestamp (i.e., a time step) of size $\tau = 100$ microseconds for List and RBTree, and $\tau = 20$

microseconds for SkipList and Vacation, empirically, by running each benchmark sequentially for 100 seconds in a single thread and finding the longest execution time duration of a single transaction among the committed transactions. The execution time of transactions is significantly longer in List and RBTree due to the long chain of nodes that must be traversed and the time needed to rebalance the tree, respectively. In contrast, SkipList and Vacation have moderate length transactions, mainly due to the layer structure (with less number of layers) and the moderate read and write set sizes, respectively. We implement window-based algorithms in DSTM2 in such a way that if a transaction aborts before the time step τ expires (because its execution time duration is less than τ), the transaction will be restarted in the beginning of the new time step.

Recall that window-based scheduling algorithms use the randomized time period at the beginning of the window and the frames of predefined time steps for the execution of transactions having high priority in the beginning of each frame. Due to the randomized interval, the probability of conflict among transactions that are in high priority at particular frame is very low. As a result, they may finish execution and commit sufficiently before the end of the frame. In this situation, we use a simple *busy-waiting* (i.e., spinning) mechanism in a while loop to make each thread wait until the new frame starts. As the threads do not need to wait for the very long time for the current frame to finish, the busy-waiting mechanism that we use wastes very little CPU time.

The performance of our window-based algorithm variants is also compared through experiments with the following scheduling algorithms (see Figs. 3.4 and 3.5 for throughput comparison in high and medium contention scenarios). We briefly describe them here (the detailed description can be found in [45, 59, 117, 119]):

- Polka [117]: combines Karma [117] and Backoff [117] by giving the enemy transaction exponentially increasing amounts of time to commit, for a number of iterations equal to the difference in the transactions' priorities, before aborting the enemy transaction. This is the overall best performing contention manager, among the scheduling algorithms proposed in the literature, in most of the TM workloads.

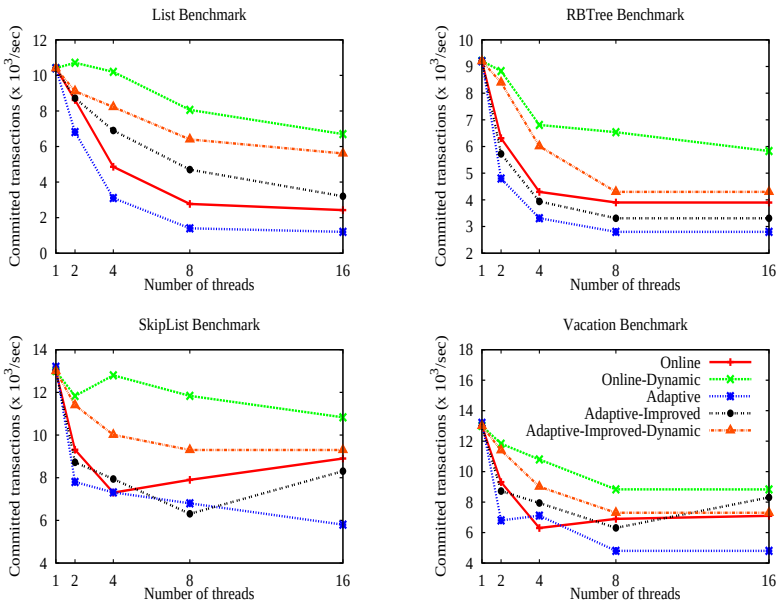


Figure 3.3: Performance throughput results of window-based algorithm variants in high contention. Higher is better.

- Greedy [59]: aborts the younger transaction between the two conflicting transactions based on static timestamps, unless the older transaction is suspended or waiting. This is the first contention manager which has non-trivial theoretical provable properties along with promising empirical performance.

- Priority [117]: is a static priority-based manager, where the priority of a transaction is its start time, that aborts lower priority transactions during conflicts. This is a very simple contention manager available in the literature.

- Serializer [45]: is a contention manager, which upon detecting a conflict between two concurrently executing transactions, aborts one transaction and moves it to the (per-core) transactions work queue of the other. This serializes transactions so that they will not conflict again. It is generally suitable for high contention scenarios.

- RandomizedRounds [119]: is a contention manager which resolves conflicts based on discrete random priorities assigned to transactions at every start and restart. This is a variation of Priority in the sense that priority of a transaction is not static.

3.6.2 Throughput Results

The throughput results of different window-based algorithm variants in List, RBTree, SkipList, and Vacation benchmarks are given in Fig. 3.3 for high contention scenarios. The dynamic variants Online-Dynamic and Adaptive-Improved-Dynamic improve the throughput compared to their static variants Online and Adaptive-Improved in all the benchmarks. In comparison to Online, the throughput improvement by Online-Dynamic is generally 1.1–5 fold in List, 1.1–2 fold in RBTree, 1.1–1.7 fold in SkipList, and 1.1–1.8 fold in Vacation. Similarly, the throughput improvement by Adaptive-Improved-Dynamic is generally 1–2 fold in List, 1.1–1.7 in RBTree, 1–2 in SkipList, and 1–1.3 in Vacation than Adaptive-Improved. The results of Adaptive also compare similarly as of Adaptive-Improved compares to Adaptive-Improved-Dynamic. Moreover, the performance variance is generally minimal between the two best performing window-based algorithm variants Online-Dynamic and Adaptive-Improved-Dynamic. Adaptive-Improved-Dynamic performs little worse than Online-Dynamic due to the time needed by it to adapt to the contention measure C_i for each thread P_i, which however, is not needed in latter one as it assumes a fixed value of C_i for each P_i and executes transactions accordingly. However, the trade-off is, if the assumed value of C_i is incorrect (generally smaller than the actual value of conflict measure for each P_i), Online-Dynamic may perform worse than Adaptive-Improved-Dynamic, and generate also the large number of bad events. The reason is that due to the incorrect choice of C, the randomization period might not be sufficient to shift the conflicting transactions to different time slots so that many of the conflicts are avoided. We do not list the throughput results of window algorithms for medium and low contention scenarios as they show patterns similar to high contention scenarios.

The remaining time between the last transaction in the frame commits and the end of the frame is wasted in Online, Adaptive, and Adaptive-Improved algorithms. This is because transactions that are in high priority at that frame may have very short execution time duration in comparison to τ (the execution time of the longest transaction) we considered, and also they may induce a very few number of conflicts. The Online-Dynamic and Adaptive-Improved-Dynamic algorithms

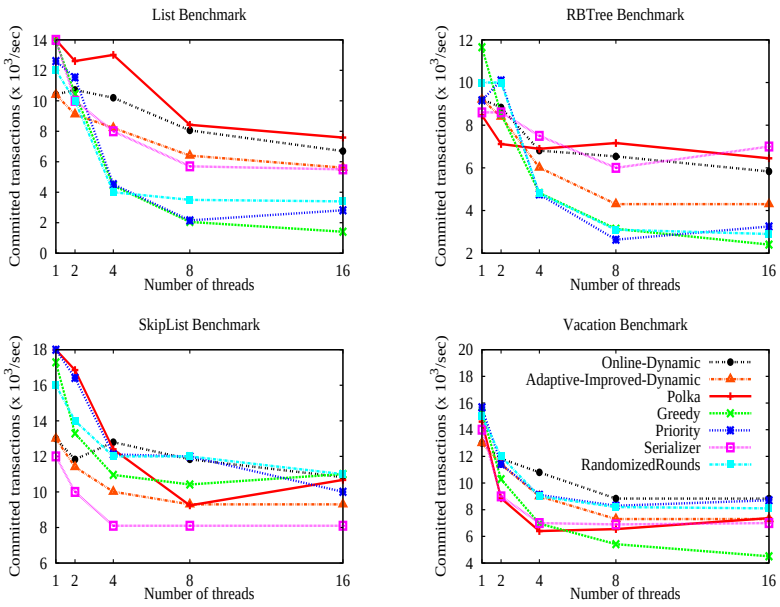


Figure 3.4: Comparison of performance throughput results in high contention. Higher is better.

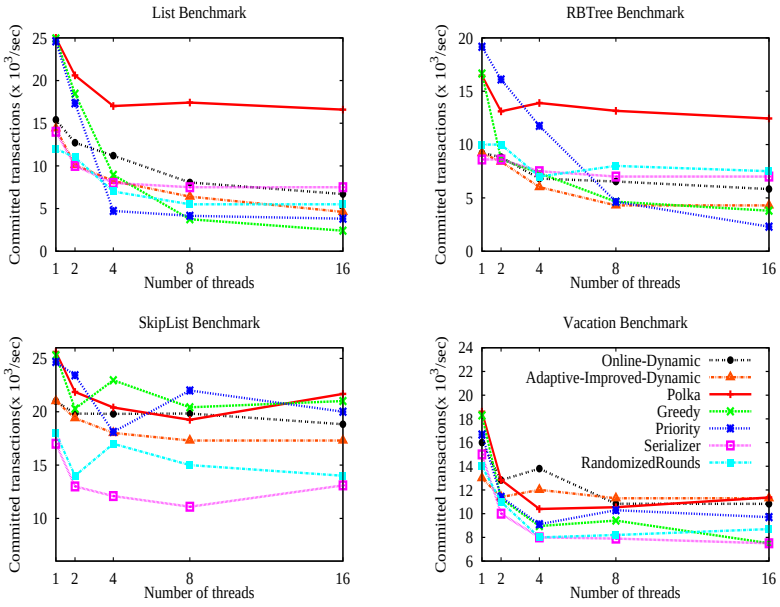


Figure 3.5: Comparison of performance throughput results in medium contention. Higher is better.

work based on *dynamic contraction* of the frames to utilize the remaining time in frames, i.e., as soon as last transaction inside a particular frame finishes, the new frame is started. This helps in reducing the overhead imposed by random delay in the beginning of the window and the size of the frames. It also helps in minimizing the busy-waiting time of the threads waiting for the current

frame to finish. That is why, as shown in Fig. 3.3, the performance throughput of dynamic variants is always better in comparison to their static variants Online, Adaptive, and Adaptive-Improved in all the benchmarks. Therefore, in the rest of the chapter, we only focus on the comparison of the best performing window variants with other scheduling algorithms in the literature. Moreover, the value of $C = MN$ we assumed for Online and Online-dynamic algorithms for high contention scenarios may not be suitable for them in medium and low contention scenarios. We analyze, in detail, the effect of the choice of C on the performance of Online and Online-Dynamic algorithms in medium and low contention scenarios in Section 3.6.5.

The throughput comparison of our algorithms with Polka, Greedy, Priority, Serializer, and RandomizedRounds in List, RBTree, SkipList, and Vacation benchmarks is given in Figs. 3.4 and 3.5 for high and medium contention scenarios (we omit the throughput results of low contention as they show similar patterns), respectively. We compare the performance of window-based algorithms with Polka because it is the overall best performing contention manager among the scheduling algorithms proposed in the literature, for most of the TM benchmarks (although it has no provable theoretical bounds). Similarly, we compare with Greedy because it is the first contention manager that exhibits non-trivial provable worst-case guarantees along with good empirical performance. We are specially interested to the comparison results of window-based algorithm variants with Greedy because of its both theoretical and practical performances. Priority is the simplest contention manager for comparison which decides to abort the transaction based on priority comparison. Moreover, we compare with Serializer because it is claimed to be suitable for high contention scenarios, and RandomizedRounds because it may give better performance using discrete randomized priorities.

The conclusion from the performance throughput results is that our window-based scheduling algorithms always improve throughput over Greedy in List, RBTree, and Vacation in high contention scenarios (see Fig. 3.4), sometimes by significant margins. This is from large transaction delays in Greedy incurred due to a transaction waiting for another transaction, which is not needed in window algorithms. The performance improvement is generally 3–6 fold in List, 3–4

fold in RBTree, and 3 fold in Vacation than Greedy, in high contention scenarios. In SkipList, the throughput of our algorithms is comparable to Greedy due to generally low conflict probability of SkipList benchmark. The throughput results are also comparable to Polka in all the benchmarks, except Vacation where window-based algorithm variants outperform by 1.3–2$\times$ (see Fig. 3.4). Polka performs well due to its careful combination of transaction priorities and exponential waiting mechanisms to resolve conflicts.

Similarly, the window-based scheduling algorithms outperform Serializer in List and SkipList by 2–4$\times$ and 1.5–2$\times$, respectively (see Fig. 3.4). This is due to the serialization overhead of the basic serializing contention manager without any proactive scheme we used in the experiments, which lowers the throughput of Serializer compared to window algorithms. In RBTree and Vacation, the throughput results of window algorithms are comparable to Serializer. Moreover, the window variants outperform RandomizeRounds in both List and RBTree by 2–3$\times$ (see Fig. 3.4). Window algorithms throughput is also comparable to RandomizedRounds in both SkipList and Vacation. The throughput comparison of our algorithms with Priority is also similar as their comparison with RandomizedRounds in high contention scenarios. This is because of the similarity of the two algorithms (Priority and RandomizedRounds) in assigning priorities to transactions; the only difference is that one maintains static priority even after the transaction restarts while another assigns new random priority after every (re)start. Moreover, even our algorithms use a variation of RandomizedRounds for conflict resolution, they perform better because the number of transactions conflicting with some transaction T inside a frame is very low (at most Φ) compared to RandomizedRounds, where it may be as much as M in each time step of execution.

The throughput comparison of our window-based scheduling algorithms in medium contention scenarios is given in Fig. 3.5. Because of the less number of conflicts, the throughput of all algorithms is generally high in medium contention scenarios in comparison to their throughput in high contention scenarios. Our algorithms outperform Greedy in List and Vacation, whereas the results are comparable in RBTree and SkipList. Similarly, the throughput results are comparable to Polka in all the benchmarks; Polka outperforms our algorithms in List and RBTree by the

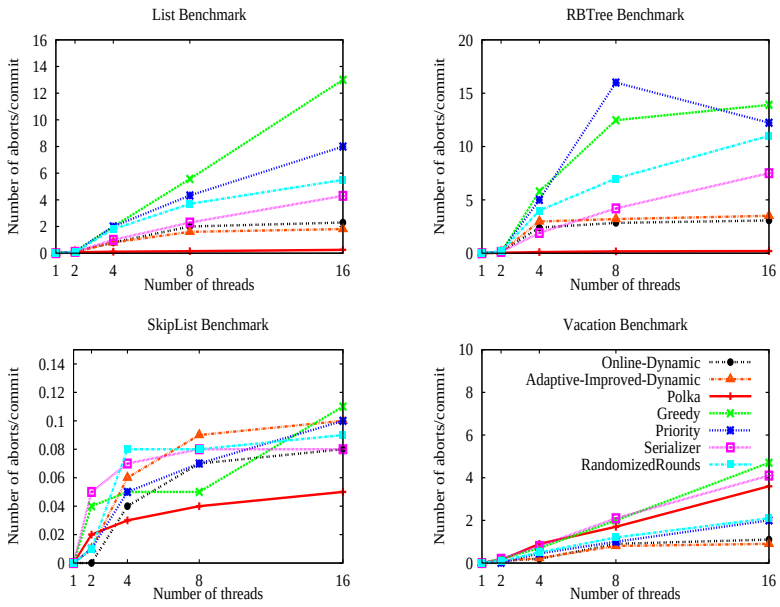


Figure 3.6: Comparison of aborts per commit ratio results in high contention. Lower is better.

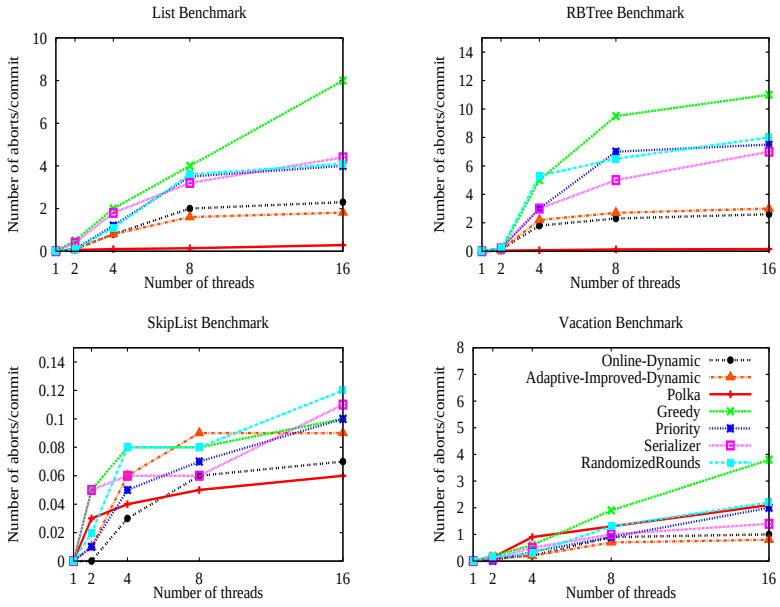


Figure 3.7: Comparison of aborts per commit ratio results in medium contention. Lower is better.

factor of 2 only. Our algorithms outperform Serializer in SkipList and Vacation by the factor of 1.3–2, whereas the results are comparable in List and RBTree. Moreover, our algorithms outperform RandomizedRounds in List, SkipList, and Vacation by the factor of 1.1–1.8, whereas the throughput is comparable in RBTree. In comparison to Priority, the throughput of our algo-

rithms is 1.2–1.8× better in List and Vacation; in RBTree and SkipList, the throughput results are comparable.

The results given in Figs. 3.4 and 3.5 also show that throughput results scale better with the increasing number of threads when the amount of contention decreases. The reason is that, as all threads modify the data structure in very high contention scenarios, the scalability is affected by the number of conflicts increases in proportional to the increasing number of concurrent threads. In contrast, in medium and low contention scenarios, the number of conflicts does not increase that significantly with the increasing number of threads compared to the number of conflicts in very high contention scenarios, hence it helps in achieving scalable throughput.

3.6.3 Aborts per Commit Ratio Results

Aborts per commit is the ratio of number of aborts to the number of commits of transactions. It is another metric used to measure the efficiency of the contention manager in utilizing the computing resources. The higher aborts per commit ratio signifies the waste of computing resources due to the aborted transactions. Figs. 3.6 and 3.7 show aborts per commit ratio results in high and medium contention scenarios (the results in low contention show similar patterns; hence omitted). The results indicate that best performing window-based algorithm variants reduce the number of aborts per commit ratio in List, RBTree, and Vacation significantly in comparison to Greedy, Priority, Serializer, and RandomizedRounds (1.5–7× less). This is because window algorithms keep conflict degree low in each time step in comparison to Greedy, Priority and RandomizedRounds, where it may be as much as the number of concurrent threads. Moreover, window algorithms also minimize aborts through randomization which helps conflicting transactions execute at different time slots so that many conflicts are avoided. The number of aborts in Serializer is due to the scheme where transactions may conflict again after the serialization.

Similarly, the number of aborts per commit of window algorithms are comparable to Polka (only 1.1–3× more) in all benchmarks except Vacation, where window-based algorithm variants outperform by 1.5–4× (see Figs. 3.6 and 3.7). This is because Polka does not immediately abort

the enemy transaction after conflict; it gives the enemy transaction exponentially increasing time to commit, which significantly minimizes number of aborts. The aborts per commit ratio results are comparable for all strategies in SkipList, due to the low conflict probability of transactions in it, in comparison to other benchmarks.

Moreover, similar to the throughput results of Section 3.6.2, the number of aborts per commit also decreases with the increasing number of threads when the amount of contention decreases. That is, the number of aborts per commit ratio in medium contention scenarios (see Fig. 3.7) is generally lower compared to the number of aborts per commit ratio in high contention scenarios (see Fig. 3.6) in all the benchmarks. This is because, as all threads modify the data structure in very high contention scenarios, transactions usually experience repeated number of conflicts before commit with the number of concurrent threads increases. In contrast, in medium and low contention scenarios, the number of repeated conflicts does not increase that significantly with the increasing number of threads, hence it helps in lowering the number of aborts. The serializing schemes, such as Serializer and Steal-On-Abort, generally give the lower number of aborts per commit ratio in all contention scenarios, but due to the use of serialization and/or transaction reordering to avoid repeat conflicts, their throughput does not scale proportionally with the increasing number of threads, when conflicts are more frequent only inside the same column transactions.

3.6.4 Execution Window Overhead Results

We measure the overhead of execution window model by allowing the window-based algorithm variants to execute 20000 randomly generated transactions in each benchmark and take into account the total time needed to commit all of them. Figs. 3.8 and 3.9 show the results for the total time needed for different scheduling algorithms (window-based algorithms and others) to commit 20000 randomly generated transactions on List, RBTree, SkipList, and Vacation benchmarks under different amounts of contention, using 16 and 4 threads, respectively.

Our best performing algorithms (Online-Dynamic and Adaptive-Improved-Dynamic) always need less time than Greedy, Priority, and RandomizedRounds in List and RBTree (see

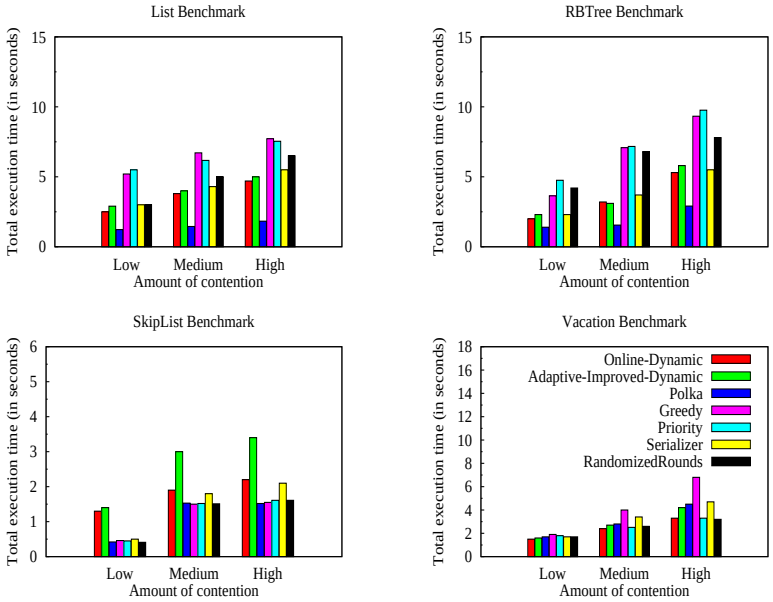


Figure 3.8: Comparison of total time needed to commit 20000 transactions using 16 threads. Lower is better.

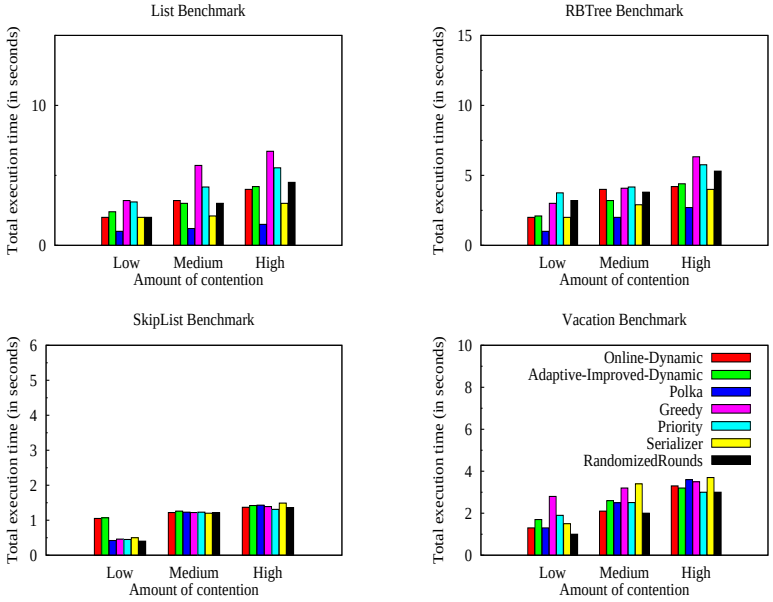


Figure 3.9: Comparison of total time needed to commit 20000 transactions using 4 threads. Lower is better.

Fig. 3.8), using 16 threads, in all contention scenarios. This is due to the large transaction delays in Greedy and due to the high degree of conflict among transactions in Priority and RandomizedRounds. The time needed in List and RBTree by Online-Dynamic is 1.5–2.8× less than

Greedy, 1.4–2.7× less than Priority, and 1.2–1.8× less than RandomizedRounds; the time needed is only 1.2–2.5× more than Polka. The time needed by Polka is lower due to the combination of good behaviors of Karma and Backoff, which give an enemy transaction sufficient time to commit before aborting it. Similarly, the time needed by Adaptive-Improved-Dynamic in List and RBTree is 1.3–2.7× less than Greedy, 1.25–2.6× less than Priority, and 1.1–1.7× less than RandomizedRounds; it only needs time the factor of 1.3–2.6 more than Polka. Similar results can be seen in List and RBTree, using 4 threads, for both window algorithms (see Fig. 3.9).

Moreover, Online-Dynamic performs 1.1× better than Serializer in medium and low contention scenarios, while its performance is comparable in high contention scenarios, using 16 threads. Using 4 threads, the performance of Online-Dynamic is either comparable or little worse than Serializer in all contention scenarios. The worse performance of Serializer in high contention scenarios is due to the serialization overhead of transaction reordering. It signifies that repeat conflicts among different column transactions are usually low in the benchmarks we used for evaluation. The time performance of Adaptive-Improved-Dynamic also compares similarly as of Online-Dynamic compares to Serializer in List and RBTree.

In the SkipList, the overhead is high (1.2–2× worse) in our algorithms in all contention scenarios, using 16 threads, due to initial randomization period and time needed for adaptive guessing of contention (not from the time needed to execute transactions), which is not generally needed in other scheduling algorithms (see Fig. 3.8). Moreover, our algorithms achieve comparable time performance in medium and high contention scenarios as of other scheduling algorithms in SkipList, while using only 4 threads in execution (see Fig. 3.9). However, the performance of our algorithms in SkipList is worse in low contention scenarios by at most a factor of 2. As overhead of our algorithms in SkipList is generally high compared to other benchmarks, we also evaluate window algorithms without random initial delay to see whether it helps in minimizing overhead in SkipList. The conclusion from such experiments is that it helps in reducing the overhead by the factor of at most 1.8 in high contention scenarios in SkipList, but it creates also the significantly large number (upto 27% in some of the execution windows) of bad events (the transactions that could not

commit within the particular frame where they switched to high priority). This is because, without the initial random delay, there will be at most M concurrent transactions released by M different threads inside a frame (similar to one-shot scheduling problem), such that all the transactions could not commit by the end of that frame. One solution to avoid these bad events is to expand the frame till all the transactions inside that frame commit (see Section 3.6.5 for details on dynamic expansion of frames), which again ends up giving the time performance that is comparable to SkipList's performance with initial random delay as shown in Figs. 3.8 and 3.9.

In Vacation, window-based variants outperform Polka, Greedy, and Serializer, but give comparable performance to Priority and RandomizedRounds (see Fig. 3.8), while using 16 threads for execution. The reason behind it is similar to the reasons we give in Section 3.6.2 for throughput results because maximizing throughput automatically helps in minimizing the total execution time. The time needed in Vacation by Online-Dynamic is 1.1–1.2$\times$ less than Polka, 1.1–1.7$\times$ less than Greedy, and 1.1–1.25$\times$ less than Seriaizer; the time needed is only 1.2$\times$ more than Priority and RandomizedRounds, in high and medium contention scenarios, using 16 threads. In low contention, window algorithms perform similar to Priority and RandomizedRounds. In contrast, RandomizedRounds appears to be the best performing contention manager for execution trials in Vacation, using 4 threads (see Fig. 3.9). This is because of the RandomizedRounds algorithm's low maximum degree of conflict among transactions in Vacation. Our window algorithms still outperform Greedy and Serializer using 4 threads, whereas they exhibit similar performance as of Polka and Priority. In summary, in low and medium contention scenarios, the overhead can be visible like in SkipList, but in high contention scenarios, the overhead due to randomization is negligible like in List, RBTree, and Vacation. Therefore, the benefits we achieve from window-based scheduling algorithms are more significant in high contention scenarios than the benefits we achieve in low contention scenarios. We can also conclude from the execution patterns that the overhead lowers with the number of threads decreases, and also with the decreasing amount of contention in most of the benchmarks.

3.6.5 Relation Among the Choice of C, τ, and the Dynamic Contraction/Expansion of Frames

The performance of window-based scheduling algorithms directly depends on the right choice of the contention measure C and the time step τ, in addition to M and N. As we can fix N and the number of threads M is generally known, we focus in this section how to choose C and τ for the better performance of the window algorithms. The choice of C impacts on the initial random period and the choice of τ impacts on the frame size. For Online and Online-Dynamic algorithms, their performance depends on both C and τ, but as adaptive variants (Adaptive, Adaptive-Improved, and Adaptive-Improved-Dynamic) adaptively guess C, their performance depends mainly on the choice of the time step τ. In high contention scenarios, it is reasonable to assume all transactions conflict with each other, i.e. $C = MN$, for Online and Online-Dynamic algorithms, however for medium and low contention scenarios, this value of C may not be suitable. Moreover, it is generally difficult to come up with the right value of C that works for medium and low contention scenarios, without applying some guessing techniques.

We argue in this section that the use of dynamic contraction/expansion of the frames helps in lowering the impact of the choice of C and τ in the performance of window algorithms. Particularly, we compare the performance of the Online and Adaptive-Improved algorithms with their dynamic variants Online-Dynamic and Adaptive-Improved-Dynamic, respectively, for the total time needed to commit 20000 randomly generated transactions in low, medium, and high contention scenarios. For the purpose of experimentation, we manually calculate the right value of C and the frame sizes for every execution window for Online in each contention scenario and execute the transactions inside that window accordingly. In Online-Dynamic, we assume $C = MN$ for each contention scenario and execute the transactions inside every window using dynamic contraction of the frames (i.e., we start the new frame as soon as all transactions inside a particular frame commit). Similarly, we compare Adaptive-Improved with Adaptive-Improved-Dynamic for frame sizes. For comparison, we manually determine the frame sizes for each execution window for Adaptive-Improved, whereas the dynamic contraction of frames is used in Adaptive-

Improved-Dynamic. The τ we used in static variants is the maximum among the execution times of transactions inside every execution window.

The comparison of total time needed by the algorithms to commit 20000 transactions using 16 threads in different contention scenarios is given in Table 3.1. The experimental results show that the dynamic variant Online-Dynamic can achieve the similar performance as of Online without the right choice of both the value of C and the frame sizes. However, the variance in total time depends on the workload where the algorithms are executed and also on the amount of contention. In SkipList and Vacation, Online-Dynamic achieves very similar time performance (with low variance) in each contention scenario because of generally shorter transactions in them compared to List and RBTree. Similarly, Adaptive-Improved-Dynamic achieves the performance comparable to Adaptive-Improved without the right choice of frame sizes (see Table 3.1). The variance in time performance is generally minimal between two adaptive variants in all the benchmarks in all contention scenarios. This is because as they guess C, the only impact in total time is due to τ which is usually very low compared to random delay incurred from higher values of C. Moreover, results also show that the difference in total time by both static and their dynamic variants decreases with the increasing amount of contention. These aforementioned benefits are due to the dynamic contraction of the frames which helps in reducing the influence of the choice of the contention measure C and also the time wasted in frames in the performance of dynamic variants.

Moreover, in some cases, due to the incorrect choice of time step τ and/or the contention measure C (generally smaller than the their actual values) for initial random period, all the transactions that are in high priority inside a particular frame may not commit until the end of the frame. In such situations we can expand the frame till all the transactions commit, which we call *dynamic expansion* of frames. The basic expansion of the frame can be obtained by adding an extra frame. As window-based scheduling algorithms obey pending commit property, even if all the transactions conflicts with each other and they need to be serialized, all transactions, with very high probability, finish by the end of the frame, if the choice of C and τ are correct. Thus, dynamic expansion of frames is generally not needed.

Table 3.1: The comparison of total time needed to commit 20000 transactions by four different window algorithms in different contention scenarios, using 16 threads. In Online, the right value of C and frame sizes are calculated manually for each contention scenario, however in Online-Dynamic, we assume $C = MN$ for all contention scenarios and perform dynamic contraction of frames. As adaptive variants guess C, we compare them for frame sizes only.

Algorithm	Contention	List	RBTree	SkipList	Vacation
Online	Low	2.1	1.8	1.2	1.4
	Medium	3.1	2.9	1.8	2.3
	High	4.2	4.0	2.1	2.8
Online-Dynamic	Low	2.4	2.2	1.5	1.7
	Medium	3.7	3.3	2.2	2.7
	High	4.8	4.6	2.2	3.2
Adaptive-Improved	Low	2.5	2.1	1.3	1.6
	Medium	3.4	3.1	2.2	2.4
	High	4.1	4.8	2.6	3.2
Adaptive-Improved-Dynamic	Low	2.6	2.3	1.5	1.6
	Medium	4.1	3.4	2.4	2.5
	High	5.3	5.1	2.7	3.4

Table 3.2: The ratio of average frame size of the dynamic variants of window algorithms compared to their static variants to commit 20000 transactions in different contention scenarios, using 16 threads. We assume $C = MN$ for both Online and Online-Dynamic algorithms in all contention scenarios.

Algorithm	Contention	List	RBTree	SkipList	Vacation
Online-Dynamic	Low	0.53	0.49	0.35	0.41
	Medium	0.68	0.57	0.45	0.52
	High	0.87	0.80	0.57	0.73
Adaptive-Improved-Dynamic	Low	0.44	0.45	0.31	0.39
	Medium	0.66	0.55	0.46	0.50
	High	0.80	0.76	0.56	0.69

We now compare the frame sizes of the static and dynamic variants of window algorithms to measure the time that was wasted in the frames by the static variants. We assume for this comparison that the frame sizes are same for both static and dynamic variants before the execution and they are also sufficient for all the transactions that are in high priority inside every frame of the

window to commit before the frame expires, even if serialization among transactions is needed. In execution, Online and Adaptive-Improved do not change the frame size, i.e., frames are fixed, but Online-Dynamic and Adaptive-Improved-Dynamic dynamically contract and expand the frames according to contention inside that particular frame. In this setting we compute the average frame size for dynamic variants and compare with the frame size of static variants. The ratio of average frame size of the dynamic variants of window algorithms in comparison to their static variants to commit 20000 transactions using 16 threads in different contention scenarios is given in Table 3.2. Results show that dynamic variants always perform better and minimize the overhead due to frame sizes in the performance of window algorithms. In SkipList and Vacation, dynamic variants observe very small frame sizes compared to their static variants in all contention scenarios. This is due to shorter transactions in SkipList and Vacation, and also due to the low conflict degree among them. Dynamic variants also perform better than their static variants in List and RBTree in all contention scenarios, but the frame sizes are not reduced drastically due to relatively longer transactions with high conflict degree among them.

3.7 Summary and Discussions

We considered greedy scheduling algorithms for transactional memory for $M \times N$ windows of transactions with M threads and N transactions per thread. We presented algorithms with new formal bounds and experimentally evaluated their variants using List, RBTree, SkipList, and Vacation benchmarks on DSTM2. These algorithms are efficient, adaptive, and improve on the worst-case performance of previous results which were based on one-shot scheduling problem. The evaluation results confirm the benefits of window-based algorithms in practical performance throughput and other transactional metrics such as aborts per commit ratio, execution time overhead, etc., along with their non-trivial provable properties. These algorithms present new trade-offs in the design and analysis of scheduling algorithms, which is certainly a step forward in the quest to design scalable scheduling algorithms for software transactional memory implementations. Moreover, the comparable performance achieved by our algorithms with respect to Polka suggests the existence

of strategies that may outperform Polka and also have both theoretical and practical performance guarantees.

The execution window model we studied in this chapter is (somehow) restrictive in assuming a fixed set of threads, all of which are ready for execution at the beginning of the window. Nevertheless, window-based algorithms operate correctly even if threads have different release times, and new threads arrive during the execution window. As long as the total number of concurrent threads in the system after the arrival of new threads does not exceed M and the value of the conflict measure C remains the same, the execution window model guarantees the same performance bounds proved in Sections 3.3, 3.4, and 3.5 for window algorithms. When new threads arrive, they can choose independently and uniformly the random initial delay consisting of q_i frames from the range $[0, (C/\ln(MN)) - 1]$, and as soon as the delay expires, start executing transactions.

However, when the total number of concurrent threads exceeds M and/or the value of C changes (resp. the conflict graph) due to the arrival of new threads, the window algorithms given in Sections 3.3, 3.4, and 3.5 may not guarantee that all the transactions that switched to high priority at the starting of some particular frame F_{ij} finish execution and commit, with high probability, before the frame expires. This is because due to the change in M and/or C after the arrival of new threads, the original value of q_i may not provide sufficient random delay in the beginning of the window and the original frame F_{ij} of size $\mathcal{O}(\ln(MN))$ time steps may not be sufficient to commit all the transactions having high priority inside it before it expires, as q_i and F_{ij} change with the new values of M and C. However, the correctness of the algorithms is still not affected. Similar to Algorithm 3 for guessing C, an adaptive algorithm can be designed to guess the right value of M for the window model where threads arrive and leave frequently. As Algorithm 3 guesses the value of C, it works perfectly even if both C and the conflict graph change due to the arrival of new threads, as long as total number of concurrent threads does not exceed M.

When we consider variable time durations for the transactions, in the makespan bounds expressions in Theorems 3.3.5 and 3.4.4 of our algorithms we can replace the parameter τ with $\tau^{\max}$, which is the maximum duration of any transaction in the window. The impact is that in the com-

petitive ratio in Corollaries 3.3.6 and 3.4.5 there will appear an additional factor $\tau^{\max}/\tau^{\min}$, where $\tau^{\min}$ is the minimum duration of any transaction in the window. In the algorithms, the basic time step duration is changed from τ to $\tau^{\max}$. Note that with variable time delays the transactions are not perfectly aligned when they enter a frame. In Offline-Greedy, this doesn't cause a problem when we compute the independent sets. On the other hand, we need to modify Online-Greedy so that when a high-priority transaction aborts, it always gives the right of way to the transaction that aborted it.

With this work, we are left with two main issues for future work. First, in the theoretical performance analysis, we plan to explore alternative algorithms where the randomization does not occur at the beginning of each window but rather during the execution of the algorithm by inserting random periods of low priority between the subsequent transactions in each thread. We will also consider the theoretical analysis of the dynamic expansion and contraction of the execution window to preserve the contention measure C. This will result in more practical algorithms with good performance guarantees.

Second, in the empirical performance analysis, as window-based algorithms exhibit encouraging performance in different benchmarks, we plan to evaluate them for other performance measures such as *wasted work*, *repeat conflicts*, *average committed transactions duration*, *average response time*, etc. Wasted work metric is the ratio which measures the proportion of execution time spent in executing aborted transactions and it is useful in measuring the cost of aborted transactions in terms of computing resources. Similarly, repeat conflicts measures the amount of time spent in executing aborted transactions. Aborts per commit ratio, wasted work, and repeat conflicts are related and minimizing the one metric automatically improve the performance on other metric. In this sense, they complement each other. However, aborts per commit ratio and repeat conflicts ignore the execution durations of the aborted and committed transactions. Since window model reduced the number of aborts using randomization, which in turn should have reduced the average committed transactions duration and repeat conflicts. At last, the average response time bounds the time spent by individual transaction in the system. We also plan to continue our evaluation

in other more complex benchmarks from the STAMP suite [31] (such as kmeans, bayes, genome, etc.) and also from STMBench7 [60] benchmark. Moreover, due to some of the inherent overheads associated with DSTM2 implementation, we plan to evaluate our algorithms using other STM implementations such as TinySTM [52] and TL2 [44] to judge accurately the benefits of the window-based contention manager variants.

Chapter 4
Tightly-Coupled Systems: Balanced Workload Model

4.1 Introduction

As we discussed in Chapter 1, in the model where performance is analyzed in terms of the number of shared resources, Attiya et al. [9] provided the best known general formal competitive ratio bound of $\mathcal{O}(s)$, where s is the number of shared resources. Moreover, Attiya et al. [9] provided a matching lower bound of $\Theta(s)$. When the number of resources s increases, the performance degrades linearly. A difficulty in obtaining better competitive ratios is that the scheduling problem of n concurrent transactions is directly related to the vertex coloring problem which is a hard problem to approximate [84]. A natural question which we address in this chapter[1] is whether it is possible to obtain better competitive ratios for the one-shot scheduling problem. As we show below, it is indeed possible to obtain sub-linear competitive ratios for the balanced transaction scheduling problem. Moreover, we provide a new harness result for any one-shot transaction scheduling problem by reducing the well-known graph coloring problem to the transaction scheduling problem. Note that we use the analysis modeling and techniques based on the neighborhood degree estimation of a transaction in the conflict graph similar to Chapter 3.

[1]This chapter published in:
Gokarna Sharma and Costas Busch. A Competitive Analysis for Balanced Transactional Memory Workloads. *Algorithmica* 63(1–2):296–322, 2012. http://link.springer.com/article/10.1007/s00453-011-9532-3

4.1.1 Contributions

In this chapter, we study contention management in the context of *balanced workloads* which have better performance potential for transactional memory. A balanced workload consists of a set of transactions in which each transaction has the following property: if the transaction performs write operations, then the number of writes it performs is a constant fraction of the total number of operations (read and writes) of the transaction. We define the *balancing ratio* β in Section 7.2 which expresses the ratio of write operations of a transaction to the overall operations of the transaction. The balancing ratio is bounded as $\frac{1}{s} \leq \beta \leq 1$, since a writing transaction writes to at least one resource. In fact, the ratio β bounds the maximum and the minimum number of writes out of the overall reads and writes of a transaction, and $\beta = \Theta(1)$ for all the writing transactions in balanced workloads.

As advocated in [11, 60], transactional memory workloads are *read-dominated*: transactions do not need write access to resources most of their duration. This includes *read-only* transactions, where transactions only observe data and do not modify it, and *late-write* transactions, where transactions first search for the data and perform insertion or deletion only after they locate it. Balanced workloads include read-only transactions, and also late-write transactions in which the number of writes are at some fraction of the total reads and writes. A similar argument holds for *early-write* transactions that write most of their duration [11]. Balanced workloads naturally include read-only transactions, but we assume that there is at least one transaction that performs writes, since otherwise the scheduling problem is trivial (no conflicts).

Balanced transactional memory workloads represent interesting and practical transaction memory scheduling problems. For example, balanced workloads represent the case where we have small sized transactions each accessing a small (constant) number of resources, where trivially $\beta = \Theta(1)$, such as *mini-transactions* $-$ simple atomic operations on a small number of locations [8]. Other interesting scenarios are transactional memory workloads which are *write intensive*, where transactions perform many writes, as for example in scientific computing applications where transactions have to update large arrays.

We present two new contention management algorithms which are especially tailored for balanced workloads and analyze their theoretical performance boundaries from the worst-case perspective. The first algorithm, Clairvoyant, is tailored for environments where the conflict relations on shared resources are known in advance, while the second algorithm, Non-Clairvoyant, is best suited to online scheduling where it is difficult to predict conflict relations. Both algorithms are greedy and able to resolve conflicts in polynomial time.

Our first algorithm, Clairvoyant, is appropriate for the broad class of *scheduling with conflicts* environments which generally arise in resource-constrained scheduling [54]. In such scheduling, a subset of transactions conflict if their cumulative demand for a resource exceeds the supply of that resource. Conflicts between transactions are modeled by a conflict graph [48], where nodes correspond to transactions and edges represent conflicts between transactions. There are many applications of this type of scheduling environment which generate predictable conflict patterns with known conflict graphs, such as balancing parallel computation load, traffic intersection control, session management in local area networks, frequency assignment is cellular networks, and dining philosophers problem [14, 18, 27, 64, 82]. Properties of balanced workloads hold in these applications due to the specific pattern of accesses on resource locations, as for example in the classical dining philosophers problem with s shared resources [14], where a transaction T_i demands resource R_i and $R_{(i+1) \bmod s}$ exclusively at any time.

Algorithm Clairvoyant is $\mathcal{O}\left(\ell \cdot \sqrt{\frac{s}{\beta}}\right)$-competitive, where s is the number of shared resources, and ℓ expresses the logarithm ratio of the longest to shortest execution times of the transactions (the transaction execution time is the time it needs to commit uninterrupted from the moment it starts). For balanced transactional memory workloads where $\beta = \Theta(1)$, and when transaction execution times are close to each other, i.e. $\ell = \mathcal{O}(1)$, Algorithm Clairvoyant is $\mathcal{O}(\sqrt{s})$-competitive. This algorithm is greedy and has the *pending commit* property (where at least one transaction executes uninterrupted each time). However, it depends on assigning priorities to the transactions based on the explicit knowledge of the transaction conflict graph at each time step of execution. That is, the algorithm should know the set of transactions that conflict with each other (which can

be represented in the form of conflict graph) to resolve conflicts. In other words, the Algorithm Clairvoyant takes decision based on the complete set of transactions (the global view of the system) at each time step of the execution. The conflict graph is highly dynamic and evolves while the execution of the transactions progresses. It also assumes that each transaction knows how long is its execution time and how many resources it accesses.

Our second algorithm, Non-Clairvoyant, is suitable for scheduling environments where conflicts are not known in advance and cannot be predicted ahead of time. Transactional memory contention management is usually related to online scheduling, where the conflicts between two transactions are discovered on the fly when they access the same shared resource at any step of the execution (i.e., conflicts between transactions are not known in advance). It is difficult to reliably predict conflicts in this scenario because of their changing behavior over time. The scheduling algorithms for online scheduling should resolve such dynamic conflicts without assuming conflict knowledge of transactions. Algorithm Non-Clairvoyant, is suitable for such online scheduling and it is randomized.

Algorithm Non-Clairvoyant achieves $\mathcal{O}\left(\ell \cdot \sqrt{\frac{s}{\beta}} \cdot \log n\right)$ competitive ratio, with high probability, at least $1 - \frac{1}{n}$, where n is the number of transactions concurrently executing in n threads. For balanced transactional memory workloads, where $\beta = \Theta(1)$, and when transaction execution times are close to each other, i.e. $\ell = \mathcal{O}(1)$, Algorithm Non-Clairvoyant is $\mathcal{O}(\sqrt{s} \cdot \log n)$-competitive. Its competitive ratio is only a $\mathcal{O}(\log n)$ factor worse in comparison with Clairvoyant, but does not require explicit knowledge of the conflict graph. That is, the algorithm does not need to know the set of transactions that conflict with each other to resolve conflicts. In other words, the Algorithm Non-Clairvoyant takes decision based on the the local view of the system at each time step of the execution. The local knowledge of the set of transactions in the system is provided by the randomized priorities (as discrete numbers) that are assigned to each transaction uniformly at random from some interval on startup and after every abort. In case of a conflict the transaction with the smallest priority number proceeds and the other aborts. This algorithm is also greedy. This algorithm uses as a subroutine a variation of the RandomizedRounds scheduling algorithm by Schneider and

Wattenhofer [119] which uses randomized priorities as described above to resolve conflicts from the local knowledge of the system and doesn't require knowledge of the conflict graph.

The $\mathcal{O}(\sqrt{s})$ bound of Algorithm Clairvoyant that appears in Section 4.3 is actually close to optimal. Through a reduction from the graph coloring problem, we show that it is impossible to approximate in polynomial time any transaction scheduling problem with $\beta = 1$ and $\ell = 1$ with a competitive ratio smaller than $\mathcal{O}((\sqrt{s})^{1-\epsilon})$ for any constant $\epsilon > 0$, unless NP $\subseteq$ ZPP.

When transactions may fail (not as a result of a conflict), we show in Section 4.6 that a simple adaption of our algorithms has a competitive ratio of at most $\mathcal{O}\left(k \cdot \ell \cdot \sqrt{\frac{s}{\beta}}\right)$ for Clairvoyant and at most $\mathcal{O}\left(k \cdot \ell \cdot \sqrt{\frac{s}{\beta}} \cdot \log n\right)$ for Non-Clairvoyant, with high probability, assuming that a transaction may fail at most k times before it eventually commits, for some $k \geq 1$. For balanced transactional memory workloads, where $\beta = \Theta(1)$, and when transaction execution times are close to each other, i.e. $\ell = \mathcal{O}(1)$, the adaption of Algorithm Clairvoyant is $\mathcal{O}(k \cdot \sqrt{s})$-competitive and the adaption of Non-Clairvoyant is $\mathcal{O}(k \cdot \sqrt{s} \cdot \log n)$-competitive.

To our knowledge, these results are significant improvements over the best previously known bound of $\mathcal{O}(s)$ ($\mathcal{O}(k \cdot s)$ when transactions may fail) for transactional memory contention managers. For general workloads (including non-balanced workloads), where transactions are equi-length ($\ell = \mathcal{O}(1)$), our analysis gives $\mathcal{O}(s)$ competitive worst case bound, since $\beta \geq 1/s$. This bound matches the best previously known bound of $\mathcal{O}(s)$ for general workloads. The parametrization of β that we provide gives more tradeoffs and flexibility for better scheduling performance, as depicted by the performance of our algorithms in balanced workloads.

4.1.2 Chapter Organization

The rest of the chapter is organized as follows. We present our TM model and definitions in Section 7.2. We present and formally analyze two new randomized algorithms, Clairvoyant and Non-Clairvoyant, in Sections 4.3 and 4.4, respectively. The hardness result of balanced transaction scheduling is presented in Section 4.5. Section 4.6 concludes the chapter with a short discussion.

4.2 Model and Preliminaries

Consider a system of $M \geq 1$ threads $\mathcal{Q} := \{Q_1, \ldots, Q_M\}$ with a finite set of s shared resources $\mathcal{R} := \{R_1, \ldots, R_s\}$. We consider batch execution problems, where the system issues a set of M transactions $\mathcal{T} := \{T_1, \ldots, T_M\}$ (*transactional memory workload*), one transaction T_i per thread Q_i.

For any transaction T_i we define the *balancing ratio* $\beta(T_i) = \frac{\lambda_w(T_i)}{\lambda(T_i)}$ as the ratio of number of writes versus the total number of resources it accesses. For a read-only transaction $\beta(T_i) = 0$. For a writing transaction it holds $\frac{1}{s} \leq \beta(T_i) \leq 1$, since there will be at least one write performed by T_i to one of the s resources. We define the *global balancing ratio* as the minimum of the individual writing transaction balancing ratios: $\beta := \min_{(T_i \in \mathcal{T}) \wedge (\lambda_w(T_i) > 0)} \beta(T_i)$. We define *balanced transactional memory workloads* as follows (recall that we consider workloads with at least one writing transaction):

Definition 7 (Balanced Workloads) *We say that a workload (set of transactions) $\mathcal{T}$ is* balanced *if* $\beta = \Theta(1)$.

In other words, in balanced transactional memory workloads the number of writes that each writing transaction performs is a constant fraction of the total number of resource accesses (for read or write) that the transaction performs. In fact, β bounds the maximum and the minimum number of writes out of the total resource accesses.

4.3 Clairvoyant Algorithm

We describe and analyze Algorithm Clairvoyant (see Algorithm 4), which depends on the prior knowledge of the conflict graph. We start with a high level overview of the algorithm. We divide the transactions into groups according to execution time duration, and further into subgroups according to the number of resources they access. We then assign an order among groups and subgroups, where lower order subgroups (groups) have always higher priority than higher order subgroups (groups). The higher priority transactions abort lower priority transactions at the time

of conflicts. The priorities within the same subgroup are determined by computing a maximal independent set in the conflict graph of pending transactions. We obtain tight competitive ratio bounds by separating the analysis of the different groups and subgroups, which is feasible due to their ordering. In particular, in a group the ratio of execution time durations is at most 2, and in a subgroup the ratio between number of shared resources accessed is bounded by 2. These constants simplify the competitive ratio analysis and makes it easier to obtain an aggregate bound for all transactions when we combine the respective results from all groups and subgroups. The balancing ratio β appears as a lower bound in the makespan analysis of a subgroup, and hence it is one of the factors in the competitive ratio analysis. Then, parameter β is important when we combine the bounds from the various groups and subgroups and it appears in the final bound expression.

Now we proceed with the details of Algorithm Clairvoyant. The writing transactions are divided into ℓ groups $A_0, A_1, \ldots, A_{\ell-1}$, where $\ell = \left\lceil \log \left(\frac{\tau_{\max}}{\tau_{\min}} \right) \right\rceil + 1$, in such a way that A_i contains transactions with execution time duration in range $[2^i \cdot \tau_{\min}, (2^{i+1} \cdot \tau_{\min} - 1)]$, for $0 \leq i \leq \ell - 1$ (Line 1 of Algorithm 4). Each group of transactions A_i is then again divided into κ subgroups $A_i^0, A_i^1, \ldots, A_i^{\kappa-1}$, where $\kappa = \lceil \log s \rceil + 1$, such that each transaction $T \in A_i^j$ accesses (for read and write) a number of resources in range $\lambda(T) \in [2^j, 2^{j+1} - 1]$, for $0 \leq j \leq \kappa - 1$ (Line 2 of Algorithm 4). We assign an order to the subgroups in such a way that $A_i^j < A_k^l$ if $i < k$ or $i = k \wedge j < l$ (Line 3 of Algorithm 4). Note that some of the subgroups may be empty. The read-only transactions are placed into a special group B which has the highest order (Lines 1, 3 of Algorithm 4).

The intuition behind the algorithm is as follows: at any time t the pending transactions are assigned a priority level which determines which transactions commit or abort. A transaction is assigned a priority which is either: *high* or *low*. Let Π_t^h and Π_t^l denote the set of transactions which will be assigned *high* and *low* priority, respectively, at time t. In conflicts, high priority transactions abort low priority transactions. Conflicts between transactions of the same priority level are resolved arbitrarily. Suppose that $\widehat{A}_t$ is the lowest order subgroup that contains pending transactions at time t. Only transactions from $\widehat{A}_t$ can be given high priority, that is $\Pi_t^h \subseteq \widehat{A}_t$.

Algorithm 4: Clairvoyant

Input: A set $\mathcal{T}$ of n transactions with global balancing ratio β;
Output: A greedy execution schedule;

1 Divide writing transactions into $\ell = \lceil \log(\frac{\tau_{\max}}{\tau_{\min}}) \rceil + 1$ groups $A_0, A_1, \cdots, A_{\ell-1}$ in such a way that A_i contains transactions with execution time duration in range $[2^i \cdot \tau_{\min}, (2^{i+1} \cdot \tau_{\min} - 1)]$; Read-only transactions are placed in special group B;
2 Divide A_i again into $\kappa = \lceil \log s \rceil + 1$ subgroups $A_i^0, A_i^1, \cdots, A_i^{\kappa-1}$ in a way that each subgroup A_i^j contains transactions that access a number of resource in the range $[2^j, 2^{j+1} - 1]$;
3 Order the groups and subgroups such that $A_i^j < A_k^l$ if $i < k$ or $i = k \wedge j < l$; special group B has highest order;
4 **foreach** *time step* $t = 0,\ 1,\ 2,\ 3, \ldots$ **do**
5 **Set Definitions:**
6 $\mathcal{T}_t$: set of transactions that are pending; // $\mathcal{T}_0 \leftarrow \mathcal{T}$
7 $\widehat{A}_t$: lowest order group that contains pending transactions;
8 $\widehat{\mathcal{T}}_t$: set of transactions in $\widehat{A}_t$ which are pending; // $\widehat{\mathcal{T}}_0 \leftarrow \widehat{A}_0$
9 $\widehat{S}_t$: set of transactions in $\widehat{\mathcal{T}}_t$ which were started before t;
10 $\widehat{S}'_t$: set of conflicting transactions in $\mathcal{T}_t$ which conflict with $\widehat{S}_t$;
11 $\widehat{I}_t$: maximal independent set in the conflict graph $G(\widehat{\mathcal{T}}_t \setminus \widehat{S}'_t)$;
12 **Priority Assignment:**
13 High priority transactions: $\Pi_t^h \leftarrow \widehat{I}_t \cup \widehat{S}_t$;
14 Low priority transactions: $\Pi_t^l \leftarrow \mathcal{T}_t \setminus \Pi_t^h$;
15 **Conflict Resolution:**
16 Execute all pending transactions;
17 **On conflict** of transaction T_u with transaction T_v:
18 **if** $(T_u \in \Pi_t^h) \ \wedge \ (T_v \in \Pi_t^l)$ **then** $abort(T_u, T_v)$;
19 **else** $abort(T_v, T_u)$;
 // $abort(T_u, T_v)$ `aborts transaction` T_v

We now give the details on how the priorities of transactions (i.e., *high* Π_t^h, and *low* Π_t^l priority sets) are determined and conflicts are resolved. The priorities are determined according to the conflict graph for the transactions. Lets divide the transactions in different sets (Lines 6–11 of Algorithm 4) according to their start time, ordering among groups and subgroups, and conflicts with other transactions at time t. Let $\mathcal{T}_t$ denote the set of all transactions which are pending at time t ($\mathcal{T}_t$ includes all transactions which have been started executing at or before time t but not aborted or committed yet). (Initially, $\mathcal{T}_0 \leftarrow \mathcal{T}$.) Let $\widehat{\mathcal{T}}_t$ denote the pending transactions of $\widehat{A}_t$ at time t.

(Initially, $\widehat{\mathcal{T}}_0 \leftarrow \widehat{A}_0$.) Let $\widehat{S}_t$ denote the set of transactions in $\widehat{\mathcal{T}}_t$ which are pending and have started executing before t but have not yet committed or aborted. Let $\widehat{S}'_t$ denote the set of transactions in $\mathcal{T}_t$ which conflict with $\widehat{S}_t$. Let $\widehat{I}_t$ be a maximal independent set in the conflict graph $G(\widehat{\mathcal{T}}_t \setminus \widehat{S}'_t)$. Then, the set of high priority transactions at time t is to be $\Pi^h_t \leftarrow \widehat{I}_t \cup \widehat{S}_t$ (Line 13 of Algorithm 4). The remaining transactions are given low priority, that is, $\Pi^l_t \leftarrow \mathcal{T}_t \setminus \Pi^h_t$ (Line 14 of Algorithm 4). Note that the transactions in Π^h_t do not conflict with each other. The transactions Π^h_t will remain in high priority in subsequent time steps $t' > t$ until they commit, since the transactions in $\widehat{S}_{t'}$ are included in $\Pi^h_{t'}$. In case of conflict between T_u and T_v, if $T_u \in \Pi^h_t$ and $T_v \in \Pi^l_t$ then T_u aborts T_v; otherwise T_v aborts T_u (Lines 17, 19 of Algorithm 4). (The routine $abort$(transaction T_u, T_v) aborts transaction T_v.) The aborted transaction immediately restarts and tries to commit again.

This algorithm is clairvoyant in the sense that it requires explicit knowledge of the various conflict relations at each time t. That is, the algorithm should know the set of transactions that conflict with each other at each time step to resolve conflicts. In other words, the Algorithm Clairvoyant takes decision based on the complete set of transactions (the global view of the system) at each time step of the execution. The conflict graph is highly dynamic and evolves while the execution of the transactions progresses. The algorithm is greedy, since at each time step each pending transaction is not idle. The algorithm also satisfies the pending commit property since at any time step t at least one transaction from $\widehat{A}_t$ will execute uninterrupted until it commits. Clearly, the algorithm computes the schedule in polynomial time.

4.3.1 Analysis of Clairvoyant Algorithm

We now give a competitive analysis of Algorithm Clairvoyant. In the next results we will first focus on a subgroup A^j_i and we will assume that there are no other transactions in the system. We give two independent bounds for the competitive ratio for A^j_i. Then, we give the competitive ratio bound for a group A_i of transactions by combining the competitive bounds of κ subgroups. At last, we give the overall performance bound for all the transactions in $\mathcal{T}$ by combining the competitive ratio bounds of ℓ groups of writing transactions and a special group B of read-only transactions.

Table 4.1: Summary of notations used in the algorithms and analysis of Sections 4.3 and 4.4.

$\tau_{\min}, \tau_{\max}$	:	$\min_i \tau_i, \max_i \tau_i$ (execution time of the shortest and the longest transaction in $\mathcal{T}$, respectively)
A_i, A_i^j, B	:	A group, a subgroup, and a special group of read-only transactions, respectively
ℓ, κ	:	$\left\lceil \log\left(\frac{\tau_{\max}}{\tau_{\min}}\right)\right\rceil + 1, \lceil \log s \rceil + 1$ (number of groups A_i and subgroups A_i^j, respectively)
$\tau_{\min}^j, \tau_{\max}^j$	:	$2^i \cdot \tau_{\min}, 2^{i+1} \cdot \tau_{\min} - 1$ (execution time of the shortest and the longest transaction in A_i^j, respectively)
$\lambda_r(T), \lambda_w(T), \lambda(T)$	:	Number of resources which are being accessed by transaction T for read, write, and either read or write, respectively
$\lambda_{\min}^j, \lambda_{\max}^j$	:	$2^j, 2^{j+1}-1$ (minimum and maximum number of resources that reads or writes a transaction in A_i^j, respectively)
γ'	:	$\max_{v\in[1,s]} \gamma_i^j(R_v)$, where $\gamma_i^j(R_v)$ is the number of transactions in A_i^j that write resource $R_v, 1 \leq v \leq s$

Before analyzing the bounds, we give here a brief description of the notations we use throughout the analysis. As writing transactions are divided into groups according to execution time duration, the duration of each transaction $T \in A_i^j$ will be in range $[\tau_{\min}^j, \tau_{\max}^j]$, where $\tau_{\min}^j = 2^i \cdot \tau_{\min}$ and $\tau_{\max}^j = (2^{i+1} \cdot \tau_{\min} - 1)$ are the execution time of the shortest and the longest transaction, respectively. As a particular group A_i is divided again into subgroups according to resources and irrespective of execution time duration $\tau_{\max}^j \leq 2 \cdot \tau_{\min}^j$ for each subgroup A_i^j. Similarly, according to the division of writing transactions in subgroups, note that for each transaction $T \in A_i^j$, the number of resources used by T is in range $\lambda(T) \in [\lambda_{\min}^j, \lambda_{\max}^j]$, where $\lambda_{\min}^j = 2^j$ and $\lambda_{\max}^j = 2^{j+1} - 1$ are the minimum and the maximum number of resources needed by T for either read or write, respectively. Also for each subgroup, $\lambda_{\max}^j \leq 2 \cdot \lambda_{\min}^j$. We summarize some of the notations used throughout the analysis of the algorithms in Table 4.1 for clarity.

We now prove the first independent bound which is deduced from the analysis of lower and upper bounds based on the degree of a transaction (the neighborhood) in the conflict graph of the transactions in A_i^j.

Lemma 4.3.1 *If we only consider transactions in subgroup A_i^j, then the competitive ratio is bounded by $CR_{Clairvoyant}(A_i^j) \leq 2 \cdot \lambda_{\max}^j + 2$.*

Proof. Let $\gamma_i^j(R_v)$ denote the number of transactions in a subgroup A_i^j that write resource $R_v, 1 \leq v \leq s$. Let $\gamma' := \max_{v \in [1,s]} \gamma_i^j(R_v)$, the maximum number among $\gamma_i^j(R_v), 1 \leq v \leq s$. Since there is only one subgroup, $\widehat{A}_t = A_i^j$. A transaction $T \in A_i^j$ conflicts with at most $\lambda_{\max}^j \cdot \gamma'$ other transactions in the same subgroup. If transaction T is in low priority it is only because some other conflicting transaction in A_i^j is in high priority. If no conflicting transaction is in high priority then T becomes high priority immediately. Since a high priority transaction executes uninterrupted until it commits, it will take at most $\lambda_{\max}^j \cdot \gamma'$ time steps until all conflicting transactions with T have committed. Thus, it is guaranteed that in at most $\lambda_{\max}^j \cdot \gamma' \cdot \tau_{\max}^j$ time steps T becomes high priority. Therefore, T commits by time $(\lambda_{\max}^j \cdot \gamma' + 1) \cdot \tau_{\max}^j$. Since T is an arbitrary transaction in A_i^j, the makespan of the algorithm is bounded by:

$$makespan_{Clairvoyant}(A_i^j) \leq (\lambda_{\max}^j \cdot \gamma' + 1) \cdot \tau_{\max}^j.$$

There is a resource that is accessed by at least γ' transactions of A_i^j for write, i.e., there are γ' nodes which degree is at least $\gamma' - 1$. All these transactions have to be serialized because they all conflict with each other in accessing the common resource. Therefore, the optimal makespan is bounded by:

$$makespan_{opt}(A_i^j) \geq \gamma' \cdot \tau_{\min}^j.$$

When we combine the upper and lower bounds we obtain a bound on the competitive ratio of the algorithm:

$$CR_{Clairvoyant}(A_i^j) = \frac{makespan_{Clairvoyant}(A_i^j)}{makespan_{opt}(A_i^j)} \leq \frac{(\lambda_{\max}^j \cdot \gamma' + 1) \cdot \tau_{\max}^j}{\gamma' \cdot \tau_{\min}^j} \leq 2 \cdot \lambda_{\max}^j + 2.$$

□

We now give the second independent bound which is deduced from the analysis of upper and lower bounds based on the pending commit and the balancing ratio properties for the transactions in A_i^j. From the pending commit property, the makespan of Algorithm Clairvoyant, in the worst-case, is bounded by the serialization of all transactions in A_i^j. But, the optimal algorithm can uniformly distribute the write accesses of transactions in A_i^j among the available shared resources so that the number of transactions conflict in accessing a shared resource R be minimized. The balancing ratio β gives the minimum number of transactions that write a particular resource $R \in \mathcal{R}$ such that serialization among the transactions accessing R is needed because of conflicts. Thus, β will appear in the lower bound of makespan, and also in the competitive ratio of the algorithm.

Lemma 4.3.2 *If we only consider transactions in subgroup A_i^j, then the competitive ratio is bounded by $CR_{Clairvoyant}(A_i^j) \leq 4 \cdot \frac{s/\beta}{\lambda_{\max}^j}$.*

Proof. Since the algorithm satisfies the pending commit property (Definition 1), if a transaction $T \in A_i^j$ does not commit, then some conflicting transaction $T' \in A_i^j$ must commit. Therefore, the makespan of the algorithm is bounded by:

$$makespan_{Clairvoyant}(A_i^j) \leq |A_i^j| \cdot \tau_{\max}^j.$$

Recall the definition of balancing ratio that for any transaction T_i, $\beta(T_i) = \lambda_w(T_i)/\lambda(T_i)$. Each transaction $T \in A_i^j$ accesses at least $\lambda_w(T)$ resources for write out of total $\lambda(T) \in [\lambda_{\min}^j, \lambda_{\max}^j]$ resources. Since we only consider transactions in A_i^j, according to the definition of β, $\lambda_w(T) \geq \beta \cdot \lambda_{\min}^j \geq \beta \cdot \frac{\lambda_{\max}^j}{2}$ for the transaction T. Consequently, by the pigeonhole principle, there is a resource $R \in \mathcal{R}$ which is accessed by at least

$$\sum_{T \in A_i^j} \frac{\lambda_w(T)}{s} \geq \frac{|A_i^j| \cdot \beta \cdot \lambda_{\max}^j}{(2 \cdot s)}$$

transactions for write. That is, when $|A_i^j|$ transactions in the subgroup A_i^j access the shared resources, the minimum number of transactions that access a particular resource $R \in \mathcal{R}$ is at least

the ratio of the sum of $\lambda_w(T)$ among all $T \in A_i^j$ (i.e., total number of writes of all transactions) to the total number of resources s in the system. Similar to Lemma 4.3.1, all these transactions accessing R have to be serialized because they conflict with each other. Therefore, the optimal makespan is bounded by:

$$makespan_{opt}(A_i^j) \geq \frac{|A_i^j| \cdot \beta \cdot \lambda_{\max}^j}{2 \cdot s} \cdot \tau_{\min}^j.$$

When we combine the above bounds of the makespan we obtain the following bound on the competitive ratio of the algorithm:

$$CR_{Clairvoyant}(A_i^j) \quad = \quad \frac{makespan_{Clairvoyant}(A_i^j)}{makespan_{opt}(A_i^j)} \leq \frac{|A_i^j| \cdot \tau_{\max}^j}{\frac{|A_i^j| \cdot \beta \cdot \lambda_{\max}^j}{2 \cdot s} \cdot \tau_{\min}^j} \leq 4 \cdot \frac{s/\beta}{\lambda_{\max}^j}.$$

□

From Lemmas 4.3.1 and 4.3.2, we obtain:

Corollary 4.3.3 *If we only consider transactions in subgroup A_i^j, then the competitive ratio of the algorithm is bounded by $CR_{Clairvoyant}(A_i^j) \leq 4 \cdot \min\left\{\lambda_{\max}^j, \frac{s/\beta}{\lambda_{\max}^j}\right\}$.*

Recall that lower order subgroups of A_i have always higher priority than higher order subgroups and the transactions requiring few resources reside in the lower order groups. Corollary 4.3.3 exhibits that one can choose from two independent bounds to work on the one which gives the minimum competitive ratio for the balanced transaction scheduling problem. We now continue to provide a bound for the performance of each individual group A_i (Lemma 4.3.4), where execution time of transactions does not appear in competitive bound, since the ratio of execution time duration of the longest and the shortest transaction in each subgroup A_i^j, $0 \leq j \leq \kappa$, is at most a factor of 2. This will help to provide bounds for all the transactions $\mathcal{T}$ (Theorem 4.3.5) by basically combining the competitive ratios of ℓ such groups, which in the worst-case perspective appears from the execution commit ordering starting from the lowest order group to the highest order group.

Lemma 4.3.4 *If we only consider transactions in group A_i, then the competitive ratio of the algorithm is bounded by $CR_{Clairvoyant}(A_i) \leq 32 \cdot \sqrt{\frac{s}{\beta}}$.*

Proof. Since the maximum number of resource accesses by a transaction $T \in A_i^j$ $\lambda_{\max}^j = (2^{j+1} - 1)$, Corollary 4.3.3 gives for each subgroup A_i^j competitive ratio

$$CR_{Clairvoyant}(A_i^j) \leq 4 \cdot \min\left\{2^{j+1} - 1, \frac{s/\beta}{2^{j+1} - 1}\right\} \leq 8 \cdot \min\left\{2^j, \frac{s/\beta}{2^j}\right\}.$$

When we consider all the κ subgroups of a group A_i, the competitive ratio of each subgroup A_i^j forms a bitonic sequence with single maximum (i.e., peak) at the subgroup $\frac{\log(s/\beta)}{2}$. Let $\psi = \frac{\log(s/\beta)}{2}$. Note that

$$\min\left\{2^j, \frac{s/\beta}{2^j}\right\} \leq 2^j, \forall j \in [0, \lfloor\psi\rfloor]; \text{and}$$

$$\min\left\{2^j, \frac{s/\beta}{2^j}\right\} \leq \frac{s/\beta}{2^j} = 2^{2\cdot\psi - j}, \forall j \in [\lfloor\psi\rfloor + 1, \kappa - 1].$$

Group A_i contains κ subgroups of transactions and the subgroups are ordered based on the resources where higher priority is given to transactions requiring few resources. In the worst case, Algorithm Clairvoyant will commit the transactions in each subgroup according to their order starting from the lowest order subgroup and ending at the highest order subgroup, since that's the order that the transactions are assigned a high priority. Therefore,

$$\begin{aligned}
CR_{Clairvoyant}(A_i) &\leq \sum_{j=0}^{\kappa-1} CR_{Clairvoyant}(A_i^j) \\
&= \sum_{j=0}^{\lfloor\psi\rfloor} CR_{Clairvoyant}(A_i^j) + \sum_{j=\lfloor\psi\rfloor+1}^{\kappa-1} CR_{Clairvoyant}(A_i^j) \\
&\leq 8 \cdot \left(\sum_{j=0}^{\lfloor\psi\rfloor} 2^j + \sum_{j=\lfloor\psi\rfloor+1}^{k-1} 2^{2\cdot\psi - j}\right) \\
&\leq 8 \cdot \left(2 \cdot 2^{\psi} + 2 \cdot 2^{\psi}\right) = 32 \cdot \sqrt{\frac{s}{\beta}}.
\end{aligned}$$

□

Theorem 4.3.5 (Competitive Ratio of Clairvoyant**)** *For set of transactions $\mathcal{T}$, Algorithm* Clairvoyant *has competitive ratio $CR_{Clairvoyant}(\mathcal{T}) = \mathcal{O}\left(\ell \cdot \sqrt{\frac{s}{\beta}}\right)$.*

Proof. In the algorithm, groups are ordered based on the execution time of transactions where higher priority is given to short transactions. As there are ℓ groups of transactions A_i, and one group B, in the worst case, Algorithm Clairvoyant will commit the transactions in each group according to their order starting from the lowest order group and ending at the highest order group. Clearly, the algorithm will execute the read-only transactions in group B in optimal time. Therefore, using Lemma 4.3.4, we obtain:

$$\begin{aligned} CR_{Clairvoyant}(\mathcal{T}) &\leq \sum_{i=0}^{\ell-1} CR_{Clairvoyant}(A_i) + CR_{Clairvoyant}(B) \\ &\leq \sum_{i=0}^{\ell-1} 32 \cdot \sqrt{\frac{s}{\beta}} + 1 = 32 \cdot \ell \cdot \sqrt{\frac{s}{\beta}} + 1. \end{aligned}$$

□

The corollary below follows immediately from Theorem 4.3.5.

Corollary 4.3.6 (Balanced Workload) *For any balanced workload with $\beta = \Theta(1)$ and when $\ell = \mathcal{O}(1)$, Algorithm* Clairvoyant *has competitive ratio $CR_{Clairvoyant} = \mathcal{O}(\sqrt{s})$.*

Through a reduction from vertex coloring, we prove in Theorem 4.5.1 (Section 4.5), that there does not exist a polynomial time algorithm for every input instance with $\beta = 1$ and $\ell = 1$ of the transaction scheduling problem such that the algorithm achieves competitive ratio smaller than $\mathcal{O}((\sqrt{s})^{1-\epsilon})$ for any constant $\epsilon > 0$. This implies that the $\mathcal{O}(\sqrt{s})$ bound of Algorithm Clairvoyant given above in Corollary 4.3.6 is arbitrarily close to optimal as ϵ approaches 0.

4.4 Non-Clairvoyant Algorithm

A limitation of Algorithm 4 is that the conflict graph of transactions to be known at each time step to resolve conflicts. We present and analyze Algorithm Non-Clairvoyant (see Algorithm 5)

Algorithm 5: Non-Clairvoyant

Input: A set $\mathcal{T}$ of n transactions with global balancing ratio β;
Output: A greedy execution schedule;

1 Divide transactions into $\ell = \lceil \log(\frac{\tau_{\max}}{\tau_{\min}}) \rceil + 1$ groups $A_0, A_1, \cdots, A_{\ell-1}$ in such a way that A_i contains transactions with execution time duration in range $[2^i \cdot \tau_{\min}, (2^{i+1} \cdot \tau_{\min} - 1)]$; Read-only transactions are placed in special group B;
2 Divide A_i again into $\kappa = \lceil \log s \rceil + 1$ subgroups $A_i^0, A_i^1, \cdots, A_i^{\kappa-1}$ in a way that each subgroup A_i^j contains transactions that access a number of resource in the range $[2^j, 2^{j+1} - 1]$;
3 Order the groups and subgroups such that $A_i^j < A_k^l$ if $i < k$ or $i = k \wedge j < l$; special group B has highest order;
4 **foreach** *time step* $t = 0,\ 1,\ 2,\ 3, \ldots$ **do**
5 Execute all pending transactions; `// at` $t = 0$ `issue all transactions`
6 **On (re)start** of transaction T:
7 $r(T) \leftarrow$ random integer in $[1, n]$;
8 **On conflict** of transaction $T_u \in A_i^j$ with transaction $T_v \in A_k^l$:
9 **if** $A_i^j < A_k^l$ **then** $abort(T_u, T_v)$;
  `// Compare order of subgroups`
10 **else if** $A_i^j > A_k^l$ **then** $abort(T_v, T_u)$;
11 **else if** $r(T_u) < r(T_v)$ **then** $abort(T_u, T_v)$;
  `// The case` $A_i^j = A_k^l$
12 **else** $abort(T_v, T_u)$;
  `// In case a transaction` T_u `aborts` T_v `because` $r(T_u) < r(T_v)$`, then when` T_v `restarts it cannot abort` T_u `until` T_u `commits or aborts`

which removes this limitation. This algorithm is similar to Clairvoyant given at Section 4.3 with the difference that the conflicts are resolved without explicitly knowing the conflict graph.

The intuition behind the algorithm is as follows: similar to Algorithm Clairvoyant, the transactions are organized in groups and subgroups (Lines 1, 2 of Algorithm 5) and lower order subgroups (groups) have always higher priority than higher order subgroups (groups) (Line 3 of Algorithm 5). At each time step t, let $\widehat{A}_t$ denote the lowest order subgroup. Clearly, the transactions in $\widehat{A}_t$ have higher priority than the transactions in all other subgroups, and in case of conflicts only the transactions in $\widehat{A}_t$ win. When transactions in the same subgroup conflict, the conflicts are resolved according to discrete random priority numbers. A transaction T, as soon as it starts ex-

ecution, chooses a discrete priority number $r(T)$ uniformly at random in the interval $[1, n]$, i.e., $r(T) \in [1, n]$. The transaction with small priority number wins at the time of conflict.

We now give the details on how the algorithm resolves conflicts. In case of a conflict of transaction $T_u \in A_i^j$ with another transaction $T_v \in A_k^l$ (Line 8 of Algorithm 5), the order of the subgroups A_i^j and A_k^l is compared first. If A_i^j is lower order subgroup than A_k^l, then T_u aborts T_v (Line 9 of Algorithm 5). If A_i^j is higher order subgroup than A_k^l, then T_v aborts T_u (Line 10 of Algorithm 5). If A_i^j and A_k^l are basically the same subgroup, we use the random priority number of T_u and T_v to resolve conflict (Lines 6, 6 of Algorithm 5). If $r(T_u) < r(T_v)$, then T_u aborts T_v (Line 11 of Algorithm 5); otherwise (in the case where $r(T_u) \not< r(T_v)$), T_v aborts T_u (Line 12 of Algorithm 5). When the aborted transaction T_v restarts, it cannot abort T_u until T_u has been committed or aborted. After every abort, the newly started transaction chooses again a new discrete priority number uniformly at random in the interval $[1, n]$ (Lines 6, 6 of Algorithm 5). This is a different technique than the timestamp approach of Greedy [59], where transactions retain the timestamp even after abort. The idea of randomized priorities has been introduced originally by Schneider and Wattenhofer [119] in their Algorithm RandomizedRounds.

This algorithm is non-clairvoyant in the sense that it does not depend on knowing explicitly the conflict graph to resolve conflicts. That is, Algorithm Non-Clairvoyant takes decision based on the the local view of the system at each time step of the execution. The local knowledge of the set of transactions in the system is provided by the randomized priorities that are assigned to each transaction uniformly at random from the interval $[1, n]$ on startup and after every abort as described in aforementioned paragraph. The algorithm is greedy but does have the pending commit property. The groups and subgroups can be implemented in the algorithm since we assume that each transaction knows its execution time and the number of resources that it accesses. Clearly, the algorithm computes the schedule in polynomial time.

4.4.1 Analysis of Non-Clairvoyant Algorithm

In the analysis given below, we study the properties of Algorithm Non-Clairvoyant and give its competitive ratios. We now give two independent competitive bounds for some subgroup A_i^j and later extend the results to all the transactions in $\mathcal{T}$. The proofs are similar as in the analysis of Algorithm Clairvoyant and given here for the sake of completeness.

Lemma 4.4.1 *If we only consider transactions in subgroup A_i^j, then the competitive ratio is bounded by $CR_{Non-Clairvoyant}(A_i^j) \leq 64 \cdot e \cdot \lambda_{\max}^j \cdot \ln n$ with probability at least $1 - \frac{|A_i^j|}{n^2}$.*

Proof. Recall the notion defined in Lemma 4.3.1 that $\gamma' = \max_{v \in [1,s]} \gamma_i^j(R_v)$, where $\gamma_i^j(R_v)$ denote the number of transactions in a subgroup A_i^j that write $R_v, 1 \leq v \leq s$. Since there is only one subgroup, a transaction $T \in A_i^j$ conflicts with at most $d_T \leq \lambda_{\max}^j \cdot \gamma'$ other transactions in the same subgroup. From Lemma 3.4.2, it will take at most

$$x = 16 \cdot e \cdot (\lambda_{\max}^j \cdot \gamma' + 1) \cdot \tau_{\max}^j \cdot \ln n$$

time steps until T commits, with probability at least $1 - \frac{1}{n^2}$. Considering now all the transactions in A_i^j, and taking the union bound of individual event probabilities, we have that all the transactions in A_i^j commit within time x with probability at least $1 - \frac{|A_i^j|}{n^2}$. Therefore, with probability at least $1 - \frac{|A_i^j|}{n^2}$, the makespan is bounded by:

$$makespan_{Non-Clairvoyant}(A_i^j) \leq 16 \cdot e \cdot (\lambda_{\max}^j \cdot \gamma' + 1) \cdot \tau_{\max}^j \cdot \ln n.$$

Similar to Lemma 4.3.1, there is a resource that is accessed by at least γ' transactions of A_i^j for write so that all these transactions have to be serialized because of the conflicts. Therefore, the optimal makespan is bounded by

$$makespan_{opt}(A_i^j) \geq \gamma' \cdot \tau_{\min}^j.$$

By combining the upper and lower bounds, we obtain a bound on the competitive ratio:

$$\begin{aligned} CR_{Non-Clairvoyant}(A_i^j) &= \frac{makespan_{Non-Clairvoyant}(A_i^j)}{makespan_{opt}(A_i^j)} \\ &\leq \frac{16 \cdot e \cdot (\lambda_{\max}^j \cdot \gamma' + 1) \cdot \tau_{\max}^j \cdot \ln n}{\gamma' \cdot \tau_{\min}^j} \\ &\leq 32 \cdot e \cdot (\lambda_{\max}^j + 1) \cdot \ln n \\ &\leq 64 \cdot e \cdot \lambda_{\max}^j \cdot \ln n, \end{aligned}$$

with probability at least $1 - \frac{|A_i^j|}{n^2}$. □

Lemma 4.4.2 *If we only consider transactions in subgroup A_i^j, then the competitive ratio is bounded by $CR_{Non-Clairvoyant}(A_i^j) \leq 64 \cdot e \cdot \frac{s/\beta}{\lambda_{\max}^j} \cdot \ln n$ with probability at least $1 - \frac{|A_i^j|}{n^2}$.*

Proof. Since for any transaction $T \in A_i^j$, $d_T \leq |N_T| \leq |A_i^j| - 1$, similar to the proof of Lemma 4.4.1, with probability at least $1 - \frac{|A_i^j|}{n^2}$, the makespan is bounded by:

$$makespan_{Non-Clairvoyant}(A_i^j) \leq 16 \cdot e \cdot |A_i^j| \cdot \tau_{\max}^j \cdot \ln n.$$

Similar to Lemma 4.3.2, the optimal makespan is bounded by:

$$makespan_{opt}(A_i^j) \geq \frac{|A_i^j| \cdot \beta \cdot \lambda_{\max}^j}{2 \cdot s} \cdot \tau_{\min}^j.$$

When we combine the above bounds of the makespan we obtain a bound on the competitive ratio:

$$\begin{aligned} CR_{Non-Clairvoyant}(A_i^j) &= \frac{makespan_{Non-Clairvoyant}(A_i^j)}{makespan_{opt}(A_i^j)} \\ &\leq \frac{16 \cdot e \cdot |A_i^j| \cdot \tau_{\max}^j \cdot \ln n}{\frac{|A_i^j| \cdot \beta \cdot \lambda_{\max}^j}{2 \cdot s} \cdot \tau_{\min}^j} \leq 64 \cdot e \cdot \frac{s/\beta}{\lambda_{\max}^j} \cdot \ln n, \end{aligned}$$

with probability at least $1 - \frac{|A_i^j|}{n^2}$. □

From Lemmas 4.4.1 and 4.4.2, we obtain:

Corollary 4.4.3 *If we only consider transactions in subgroup A_i^j, then the competitive ratio of the algorithm is bounded by $CR_{Non-Clairvoyant}(A_i^j) \leq 64 \cdot e \cdot \min\left\{\lambda_{\max}^j, \frac{s/\beta}{\lambda_{\max}^j}\right\} \cdot \ln n$ with probability at least $1 - \frac{|A_i^j|}{n^2}$.*

Similar to the analysis of Algorithm 4, we now provide a bound for the performance of individual groups in Algorithm 5 which will help to provide bounds for all the transactions.

Lemma 4.4.4 *If we only consider transactions in group A_i, then the competitive ratio of the algorithm is bounded by $CR_{Non-Clairvoyant}(A_i) \leq 512 \cdot e \cdot \sqrt{\frac{s}{\beta}} \cdot \ln n$ with probability at least $1 - \frac{|A_i|}{n^2}$.*

Proof. Since $\lambda_{\max}^j = (2^{j+1} - 1)$, Corollary 4.4.3 gives for each subgroup A_i^j competitive ratio

$$\begin{aligned} CR_{Non-Clairvoyant}(A_i^j) &\leq 64 \cdot e \cdot \min\left\{2^{j+1} - 1, \frac{s/\beta}{2^{j+1} - 1}\right\} \cdot \ln n \\ &\leq 128 \cdot e \cdot \min\left\{2^j, \frac{s/\beta}{2^j}\right\} \cdot \ln n, \end{aligned}$$

with probability at least $1 - \frac{|A_i^j|}{n^2}$. Following the proof steps as in Lemma 4.3.4, we obtain:

$$CR_{Non-Clairvoyant}(A_i) \leq 512 \cdot e \cdot \sqrt{\frac{s}{\beta}} \cdot \ln n.$$

This bound holds with with probability at least $1 - \frac{\sum_{j=0}^{\kappa-1} |A_i^j|}{n^2} = 1 - \frac{|A_i|}{n^2}$, since $\sum_{j=0}^{\kappa-1} |A_i^j| = |A_i|$.

□

Theorem 4.4.5 (Competitive Ratio of Non-Clairvoyant**)** *For a set of transactions $\mathcal{T}$, Algorithm* Non-Clairvoyant *has competitive ratio $CR_{Non-Clairvoyant}(\mathcal{T}) = \mathcal{O}\left(\ell \cdot \sqrt{\frac{s}{\beta}} \cdot \log n\right)$ with probability at least $1 - \frac{1}{n}$.*

Proof. Similar to the proof of Theorem 4.3.5, as there are ℓ groups of transactions A_i, and one group B, in the worst case, Algorithm Non-Clairvoyant will commit the transactions in each

group according to their order starting from the lowest order group and ending at the highest order group. Clearly, the algorithm will execute the read-only transactions in group B in optimal time. Therefore, using Lemma 4.4.4 we obtain:

$$\begin{aligned} CR_{Non-Clairvoyant}(\mathcal{T}) &\leq \sum_{i=1}^{\ell} CR_{Non-Clairvoyant}(A_i) + CR_{Non-Clairvoyant}(B) \\ &\leq \sum_{i=0}^{\ell-1} 512 \cdot e \cdot \sqrt{\frac{s}{\beta}} \cdot \ln n + 1 \\ &= 512 \cdot e \cdot \ell \cdot \sqrt{\frac{s}{\beta}} \cdot \ln n + 1, \end{aligned}$$

with probability at least $1 - \frac{\sum_{i=0}^{\ell-1} |A_i|}{n^2} = 1 - n^{-1}$, since $\sum_{i=0}^{\ell-1} |A_i| = |\mathcal{T}| = n$. □

The corollary below follows immediately from Theorem 4.4.5.

Corollary 4.4.6 (Balanced Workload) *For any balanced workload with $\beta = \Theta(1)$ and when $\ell = \mathcal{O}(1)$, Algorithm* Non-Clairvoyant *has competitive ratio $CR_{Non-Clairvoyant} = \mathcal{O}(\sqrt{s} \cdot \log n)$ with probability at least $1 - \frac{1}{n}$.*

4.5 Hardness of Balanced Transaction Scheduling

In this section, we show that the performance of Clairvoyant is close to optimal by reducing the graph coloring problem to the transaction scheduling problem. Similar reductions from vertex coloring to conflict graphs appear in several previous work (e.g., [12]). However, we provide here the reduction details for the sake of completeness.

A VERTEX COLORING problem instance asks whether a given graph G is k-colorable [55]. A valid k-coloring is an assignment of integers $\{1, 2, \cdots, k\}$ (the colors) to the vertices of G so that neighbors receive different integers. The chromatic number, $\chi(G)$ is the smallest k such that G has a valid k-coloring. We say that an algorithm approximates $\chi(G)$ with approximation ratio $q(G)$ if it outputs $u(G)$ such that $\chi(G) \leq u(G)$ and $u(G)/\chi(G) \leq q(G)$. Typically, $q(G)$ is expressed only as a function of n, the number of vertices in G. It is well known that VERTEX COLORING

problem is NP-complete. It is also shown in [50] that unless NP $\subseteq$ ZPP, there does not exist a polynomial time algorithm to approximate $\chi(G)$ with approximation ratio smaller than $\mathcal{O}(n^{1-\epsilon})$ for any constant $\epsilon > 0$, where n denotes the number of vertices in graph G.

A TRANSACTION SCHEDULING problem instance P asks whether a set of transactions $\mathcal{T}$ with a set of resources $\mathcal{R}$ has makespan k time steps. We give a polynomial time reduction of the VERTEX COLORING problem to the TRANSACTION SCHEDULING problem P. Consider an input graph $G = (V, E)$ of the VERTEX COLORING problem, where $|V| = n$ and $|E| = s$. We construct a set of transactions $\mathcal{T}$ such that for each $v \in V$ there is a respective transaction $T_v \in \mathcal{T}$; clearly, $|\mathcal{T}| = |V| = n$. We also use a set of resources $\mathcal{R}$ such that for each edge $e \in E$ there is a respective resource $R_e \in \mathcal{R}$; clearly, $|\mathcal{R}| = |E| = s$. If $e = (u, v) \in E$, then both the respective transactions T_u and T_v use the resource R_e for write. Since all transaction operations are writes, we have that $\beta = 1$. We take all the transactions to have the same execution length equal to one time step, that is, $\tau_{\max} = \tau_{\min} = 1$, and $\ell = 1$.

Let $G(P)$ be the conflict graph for the transactions $\mathcal{T}$. Note that $G(P)$ is isomorphic to G. Node colors in G correspond to time steps in which transactions in $G(P)$ are issued. Suppose that G has a valid k-coloring. If a node $v \in G$ has a color x, then the respective transaction $T_v \in G(P)$ can be issued and commit at time step x, since no conflicting transaction (neighbor in $G(P)$) has the same time assignment (color) as T_v. Thus, a valid k-coloring in G implies a schedule with makespan k for the transactions in $\mathcal{T}$. Symmetrically, a schedule with makespan k for $\mathcal{T}$ implies a valid k-coloring in G.

It is easy to see that the TRANSACTION SCHEDULING problem is in NP. From the reduction of the VERTEX COLORING problem, we also obtain that TRANSACTION SCHEDULING problem is NP-complete. Further, we see that the reduction given above is gap preserving with gap preserving parameter $\rho = 1$ [80].

From the above reduction, we have that an approximation ratio $q(G)$ of the VERTEX COLORING problem implies the existence of a scheduling algorithm $\mathcal{A}$ with competitive ratio $CR_{\mathcal{A}}(\mathcal{T}) = q(G)$ of the respective TRANSACTION SCHEDULING problem instance P, and vice-versa. Since

$s = |\mathcal{R}| = |E| \leq (n^2 - n)/2$, an $(\sqrt{s})^{1-\epsilon}$ competitive ratio of $\mathcal{A}$ implies at most an $n^{1-\epsilon}$ approximation ratio of VERTEX COLORING. Since, we know that unless NP $\subseteq$ ZPP, there does not exist a polynomial time algorithm to approximate $\chi(G)$ with approximation ratio smaller than $\mathcal{O}(n^{1-\epsilon})$ for any constant $\epsilon > 0$, we obtain a symmetric result for the TRANSACTION SCHEDULING problem P:

Theorem 4.5.1 (Approximation Hardness of TRANSACTION SCHEDULING) *Unless* NP $\subseteq$ ZPP*, we cannot obtain a polynomial time transaction scheduling algorithm such that for every instance with $\beta = 1$ and $\ell = 1$ of the* TRANSACTION SCHEDULING *problem the algorithm achieves competitive ratio smaller than $\mathcal{O}((\sqrt{s})^{1-\epsilon})$ for any constant $\epsilon > 0$.*

Theorem 4.5.1 implies that the $\mathcal{O}(\sqrt{s})$ bound of Algorithm Clairvoyant, given in Corollary 4.3.6 (Section 4.3) for $\beta = \Theta(1)$ and $\ell = \mathcal{O}(1)$, is arbitrarily close to optimal as ϵ approaches 0.

We observe that some instances of the TRANSACTION SCHEDULING problem can be transformed into other instances with smaller number of resources and isomorphic conflict graphs. For example, the problem instance with n transactions and $(n^2 - n)/2$ resources (accessed pairwise by transactions) forms a clique of size n, while there is a problem with only one resource (accessed by all n transactions) which also forms a clique of size n. As an interesting consequence of Theorem 4.5.1, there are non-trivial problem instances with $s \geq n - 1$ shared resources such that it is not always possible to find in polynomial time (unless NP $\subseteq$ ZPP) an instance with smaller number of resources and isomorphic graphs. If every instance P with $s \geq n - 1$ resources can be replaced in polynomial time with an instance P' with at most $f(s) < s$ resources, we could obtain a polynomial time algorithm for the VERTEX COLORING problem with $\mathcal{O}(n^{1-\epsilon})$ approximation for some constant $\epsilon > 0$. The argument behind it is that starting with some arbitrary connected graph $G = (V, E)$ we obtain through the reduction described above in polynomial time a scheduling problem P and then in polynomial time a scheduling problem P', such that G is isomorphic to $G(P)$ and $G(P')$, where P has $|V| = n$ transactions and $|E| = s \geq n - 1$ shared resources, and P' has also n transactions and at most $f(s) < s$ shared resources. Algorithm 4 gives $\mathcal{O}(\sqrt{f(s)})$ competitive ratio for the schedule of P' which gives $\mathcal{O}(\sqrt{f(s)})$ competitive ratio for P

and $\mathcal{O}(\sqrt{f(|E|)})$ approximation for $\chi(G)$. Taking $f(x) = x^{1-\epsilon}$ for any chosen $0 < \epsilon \leq 1$, since $|E| = \mathcal{O}(n^2)$, we obtain an $\mathcal{O}(n^{1-\epsilon})$ approximation for the chromatic number of G in polynomial time, which is a contradiction using the known result of [50] (unless NP $\subseteq$ ZPP).

4.6 Summary and Discussions

We have studied the competitive ratios achieved by transactional contention managers on balanced transactional memory workloads. The contention management algorithms presented in this chapter achieve close to optimal competitive bounds on balanced workloads. We also establish hardness results on the competitive ratios in our balanced workload model by reducing the well known NP-complete vertex coloring problem to the transaction scheduling problem. These are the first such results that show competitive ratio bounds smaller than best previously known $\mathcal{O}(s)$ competitive ratio bound can be achieved using reasonable assumptions for the contention management policies.

When we consider a system in which transactions are faulty; if a transaction T_i running at time t fails (not as a result of a conflict), the execution of T_i needs to be restarted subsequently by the contention manager. Following Guerraoui et al. [57] we also assume that a transaction may fail at most k times, for some $k \geq 1$, before it eventually commits. The transaction is immediately restarted after each failure. Definitely, for any transaction T_i, our algorithms may run T_i almost to completion at most k times due to at most k subsequent restarts in its execution due to a failure before it eventually commits in the $(k+1)$-th round. This gives the upper bound in processing time of T_i to $(k+1)\tau_i$. This implies if each transaction fails at most k times then in the competitive ratio bound expressions of a simple adaption of our algorithms there will appear an additional factor of k, i.e., the adaption of Clairvoyant is $\mathcal{O}\left(k \cdot \ell \cdot \sqrt{\frac{s}{\beta}}\right)$-competitive and the adaption of Non-Clairvoyant is $\mathcal{O}\left(k \cdot \ell \cdot \sqrt{\frac{s}{\beta}} \cdot \log n\right)$-competitive, with high probability. For balanced workloads, where $\beta = \Theta(1)$, and when transaction execution times are close to each other, i.e. $\ell = \mathcal{O}(1)$, the adaption of Algorithm Clairvoyant is $\mathcal{O}(k \cdot \sqrt{s})$-competitive and the adaption of Non-Clairvoyant is $\mathcal{O}(k \cdot \sqrt{s} \cdot \log n)$-competitive.

There are several interesting directions for future work. As advocated in [75], our algorithms are *conservative* − abort at least one transaction involved in a conflict − as it reduces the cost to track conflicts and dependencies. It is interesting to look whether the other schedulers which are less conservative can give improved competitive ratios by reducing the overall makespan. First, our study can be complemented by studying other performance measures, such as the *average response or waiting time* or the *average punishment time* of transactions under balanced workloads. Second, while we have theoretically analyzed the behavior of balanced workloads, it is interesting to see how our contention managers compare experimentally with prior transactional contention managers, e.g., [7, 45, 46, 59, 70, 107, 130, 142].

Chapter 5
Distributed Systems: General Network Model

5.1 Introduction

In this chapter[1], we present a data-flow distributed implementation of transactional memory (DTM) protocol that is suitable for arbitrary network topologies. Previous approaches, Arrow [43], Relay [143], Combine [10], and Ballistic [77], were only for either specific network topologies or they do not scale well in arbitrary network topologies (see Table 2.2 in Chapter 2 for the comparison of results). Our protocol ensures that all three operations for shared objects are served with minimum overhead in any arbitrary network. In order to analyze the DTM protocol, we model the network as a weighted graph, where graph nodes correspond to processors and graph edges correspond to communication links between processors. In the analysis, similar to previous proposals [10, 43, 77, 143] for DTMs, we consider transactions with only one shared object. This protocol can be generalized to accommodate transactions with multiple objects by appropriately replicating the protocol for each shared object.

5.1.1 Theoretical Contributions

We propose a novel DTM protocol called Spiral which is suitable for general graphs. Spiral is a directory-based protocol implemented on a hierarchy of clusters. There are $h + 1 = \mathcal{O}(\log D)$ cluster levels such that cluster diameters increase exponentially. In each cluster one node is chosen

[1]This chapter accepted for publication in:
Gokarna Sharma and Costas Busch. Distributed Transactional Memory for General Networks. *Distrib. Comput.*, Preprint, 2014. http://link.springer.com/article/10.1007/s00446-014-0214-7

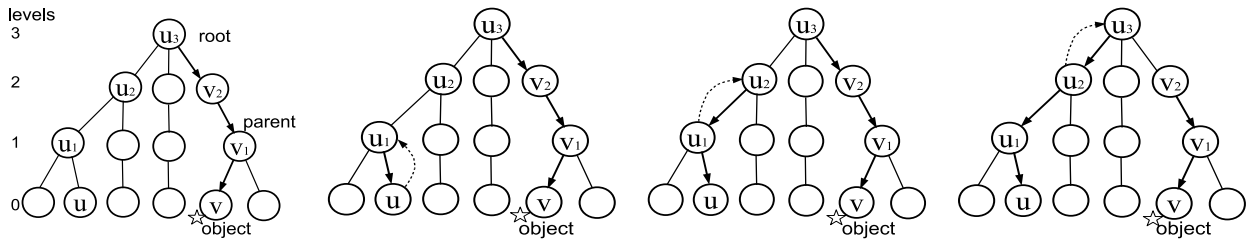
to act as a leader which is used to communicate with different level clusters. Clusters may overlap and the same node may act as a leader in multiple levels. At the bottom level (level 0) each cluster consists of individual nodes, while at the top level (level h) there is a single cluster for the whole graph with a special leader node called *root*. Only the bottom level nodes can issue requests for the shared object, while the nodes in higher levels are used to propagate the requests in the graph.

The protocol maintains a *directory path* which is directed from the root to the bottom-level node that owns the shared object. The directory path is updated whenever the object moves from one node to another. In order to get access to the object, each bottom level node uses a *spiral path* to intersect the directory path and then reach the object. The spiral path is built by visiting upward the leader nodes in all the clusters that the node belongs to starting from the bottom level up to the top level. The name of the protocol is inspired by the spiral path form which slowly unwinds outwards while it visits cluster leaders of higher levels.

The directory path is built upon the spiral paths of nodes initiating operations in the graph. As soon as the object is created by some bottom level node, it publishes the object by following its spiral path towards the root, making each parent pointing to its child and hence forming the initial directory path. Fig. 5.1a shows the leaders in the cluster hierarchy after the successful *publish* operation of v with directory path from the root u_3 to v. When some node u issues a *move* request, the request goes upward following u's spiral path until it intersects the directory path to v (Figs. 5.1b−5.1d). While going up, the *move* request also sets downward links toward u. The *move* request resets the directory path it follows while descending towards the owner v (Figs. 5.1e−5.1g); the directory path now points to u. As soon as the *move* request reaches v, the object is forwarded to u along some shortest path (Fig. 5.1h). A *lookup* operation is served similar to *move* without modifying the directory path.

Spiral guarantees:

- $\mathcal{O}(\log^4 n)$ stretch for any *lookup* operation, and
- $\mathcal{O}(\log^2 n \cdot \log D)$ stretch for *move* operations.



(a) Initially, the owner node v publishes the object

(b) Node u issues a move request

(c) The request continues up phase

(d) Directory path found, new owner node u

(e) The request goes down, discarding old pointers

(f) The request continues down phase

(g) The request reaches the owner node v

(h) Object is moved directly from v to u

Figure 5.1: Illustration of Spiral protocol for a $move$ request. The *move* request is issued by node u for the object at node v and the nodes shown in the figure from level 1 to 3 are leader nodes of the respective clusters.

The *publish* cost is proportional to the diameter of the network and it is a fixed initial cost which is only considered once and compensated by the costs of the *lookup* and *move* operations which are issued thereafter. For *move* operations, the above stretch is obtained for two kinds of execution scenarios after the object is being published: (i) *sequential* executions which consist of a non-overlapping sequence of *move* operations, and (ii) *concurrent* executions where a set of *move* operations are issued simultaneously. The reason for considering sets of *move* operations is because they provide small amortized cost compared to considering individual operations. Note that *lookup* operations have always small cost even when considered individually. To the best of our knowledge, this is the first DTM protocol that achieves poly-log stretch in general graphs.

The small stretch of the Spiral protocol is achieved due to the novel use of a (σ, χ)-*labeled cover hierarchy*. Each level i of clusters is a sparse cover with locality $\gamma_i = \Theta(2^i)$, such that for each node there is at least one cluster that contains the whole γ_i neighborhood of that node, the diameter of any cluster does not exceed $\sigma\gamma_i$, and each node belongs to at most χ clusters. The novelty is that in every level we assign a *label* to each cluster that corresponds to number between

1 and χ. The spiral path of a node v visits all the leaders of all the clusters that v belongs to starting from level 0 up to level h. The combination [level,label] specifies the order that clusters are to be visited when forming the spiral path. Since v may belong up to χ clusters in a level, the labels define the order that the clusters are to visited inside the level. We provide a $(\mathcal{O}(\log n), \mathcal{O}(\log n))$-labeled cover hierarchy which is derived from the hierarchical graph partition in [62]. This sparse cover hierarchy enables to bound the lengths of the spiral paths, and also bound the distance of the intersections of the spiral paths with respect to the the origin node distances, which further give the aforementioned bounds on *lookup* and *move* operations. It also helps to avoid race conditions that may occurs in concurrent executions. In other words, during concurrent execution of move operations, Ballistic [77] may need to lock simultaneously multiple parent nodes in the same level and probe them sequentially which may introduce *blocking* (a request may need to wait at some immediate level before probing its parents), while in Spiral only one node needs to be locked at a time in the spiral path which avoids blocking (details are in Section 5.7).

The concept of Spiral (and also Ballistic [77]) is similar to the location-aware DHTs to locate nearest neighbors, tracking mobile users, compact routing, and related problems (e.g., [15, 20, 88, 103, 105, 136]). However, these approaches provide efficient techniques only to locate copies and when the objects move autonomously (without being requested). The techniques for DTM provide mechanisms that can make moving, looking up, and republishing of objects efficient and also avoid race conditions that might occur while synchronizing concurrent requests.

The lower bounds known for some immediately related problems including *mobile user* [15] and *universal TSP tour* [56, 63, 83], also apply in our case. The mobile user problem lower bound of $\Omega(\log n/\log\log n)$ by Alon *et al.* [4] for various networks including the hypercube, and any highly expanding graph, applies immediately for our sequential execution scenario of *move* operations, since each *move* operation corresponds to a relocation of a mobile object from one node to another. The universal TSP tour lower bounds, such as $\Omega(\sqrt{\log n/\log\log n})$ by Jia *et al.* [83] for Euclidean metrics, $\Omega(\sqrt[6]{\log n/\log\log n})$ by Hajiyaghayi *et al.* [63] for $n \times n$ grid, and $\Omega(\log n)$ by Gorodezky *et al.* [56] for Ramanujan graphs, apply to our concurrent execution

scenario of *move* operations, since the solution provided by our protocol can be easily converted to give a universal TSP tour on an arbitrary graph. Our results thus imply that Spiral is not far from being optimal, showing that hierarchical labeled covers yield near-optimal bounds for DTMs.

5.1.2 Practical Contributions

For performance guarantees of Spiral is real world scenarios, we implemented and experimented it for its performance in random networks of different sizes that are generated using the Erdős-Rényi model [47]. We compared also the performance of Spiral with the performance of Arrow [43] in those networks. This comparison is also extended to ring networks of different sizes for the performance tradeoffs in the worst-case. These experimental results confirmed our theoretical findings and showed the benefits of using the hierarchy of clusters approach.

5.1.3 Chapter Organization

The rest of the chapter is organized as follows: We present the network model in Section 5.2. In Section 5.3, we give our novel construction of sparse cover hierarchy. We present the details of Spiral in Section 5.4. We then formally analyze the Spiral protocol in Section 5.5. We present experimental evaluation results in Section 5.6. Section 5.7 concludes the chapter with some discussions.

5.2 Model and Preliminaries

We start with some necessary definitions. We represent a distributed network as a weighted graph $G = (V, E, \mathfrak{w})$, with nodes (network machines) V, where $|V| = n$, edges (interconnection links between machines) $E \subseteq V \times V$ and edge weight function $\mathfrak{w} : E \rightarrow \mathbb{R}^+$. We assume that $\mathfrak{w}(u, u) = 0$ for any $u \in V$. A path p in G is a sequence of nodes, with respective sequence of edges connecting the nodes, such that $\mathrm{length}(p) = \sum_{e \in p} \mathfrak{w}(e)$. For convenience, we will treat paths as walks which may consist of a single node or the same node may be repeated. A *sub-path*

of p is any path obtained by a subsequence of consecutive nodes in p; we may also refer to a sub-path as a *fragment* of p. For simplicity assume that G is connected, that is, there is a path in G between any pair of nodes. Let $\mathrm{dist}(u, v)$ denote the shortest path length (distance) between nodes u and v. The k-neighborhood of a node v is the set of nodes which are within distance at most k from v (including v). The *diameter* D is the maximum shortest path distance over all pairs of nodes in G.

We assume that G represents a network in which nodes do not crash, it implements FIFO communication between nodes (i.e. no overtaking of messages occurs), and messages are not lost. As can be observed in Table 2.2, most of the DTM protocols (with the exception of Combine) have also the FIFO assumption. We also assume that, upon receiving a message, a node is able to perform a local computation and send a message in a single atomic step. Moreover, we treat each node in the graph G as a process. Each process pc_i has *local variables*. Processes also have *states* (including initial states and possibly also final states), while variables take on *values*. We define a *system* as a collection of processes, where processes communicate via message passing between links. A *configuration* is a complete description of the system at some point in time, i.e., the state of each process and the state of each local variable. In other words, a configuration captures the current snapshot of the entire system. There is a (unique) *initial* configuration in which every process is in its initial state.

An *event* is a *step* in which a process pc_i either (i) executes some local computation (*computation* event) or (ii) delivers a message to some other process pc_j (*delivery* event), which provides a change to the process's state. An *execution interval* is a finite or infinite alternating sequence $C_0, \phi_0, C_1, \phi_1, C_2, \cdots$, where C_k is a configuration, ϕ_k is an event, and the application of ϕ_k to C_k results in C_{k+1}, for every $k = 0, 1, \cdots$. An *execution* constitutes an execution interval in which C_0 is the initial configuration. Moreover, we say that a process state is *quiescent* if there is no sequence of events from that state in which a message is sent (i.e., process will not send another message until it receives a message). A configuration is said to be quiescent if no messages are in transit and every process is in a quiescent state.

5.3 Hierarchical Clustering

We describe how to represent the network as a hierarchy of clusters. Based on those clusters we will define paths that are used by our DTM algorithm.

5.3.1 Labeled Cover

A *node cluster* is any set of nodes $X \subseteq V$. The diameter of a cluster X is the maximum distance between any of its nodes, namely, $\mathrm{diam}(X) = \max_{u,v \in X} \mathrm{dist}(u, v)$, where distances are with respect to G.

A cover is any set of clusters $Z = \{X_1, X_2, \ldots, X_k\}$ such that each node in $u \in V$ belongs to at least one cluster in Z. Let $Z(u)$ denote the set of clusters that u belongs to in Z. The diameter of cover Z is the maximum diameter of its clusters: $\mathrm{diam}(Z) = \max_{X \in Z} \mathrm{diam}(X)$. We say that Z has *locality* γ, if for a node u, there is some cluster $X \in Z$ such that it contains the γ-neighborhood of $u \in X$.

A χ-*labeling* of Z, for some positive integer χ, is an assignment of integer labels to its clusters, $\lambda(X_i) \in \{1, 2, \ldots, \chi\}$. A χ-labeling is *valid* if for each node $u \in V$ every cluster that contains u has a different label, that is, if $X_i, X_j \in Z(u)$, $i \neq j$, then $\lambda(X_i) \neq \lambda(X_j)$.

Definition 8 (labeled cover) *Z is a* (σ, χ, γ)-labeled cover *when it satisfies the following properties: Z is a cover with locality γ, $\mathrm{diam}(Z) \leq \sigma\gamma$, and accepts a valid χ-labeling.*

5.3.2 Cover Hierarchy

We now give the definition of a hierarchy of labeled covers with exponentially increasing locality:

Definition 9 (labeled cover hierarchy) *$\mathcal{Z} = \{Z_0, Z_1, \ldots, Z_h\}$ is a* (σ, χ)-labeled cover hierarchy *when each Z_i, $1 \leq i \leq \kappa$, is a (σ, χ, γ_i)-labeled cover with locality $\gamma_i = 2^{i-1}$, where $Z_0 = V$ (each node in V is a cluster by itself) and $h = \lceil \log D \rceil + 1$. We say that $Z_i \in \mathcal{Z}$ is the* level i cover*, and any cluster $X \in Z_i$ is a* level i cluster.

We present a $(\mathcal{O}(\log n), \mathcal{O}(\log n))$-labeled cover hierarchy. The structure is based on well-known ideas for clustering the graph to approximate graph distance metrics by distributions over tree metrics [19, 49]. Specifically, we borrow the clustering technique used by Gupta, Hajiaghayi, and Räcke [62] (which in turn is an extension of the scheme proposed by Fakcharoenphol, Rao, and Talwar [49]). We present the results in the context of labeled covers. Note that some other clustering methods including the early work of Awerbuch and Peleg [13] can also be used in our construction.

The construction in [62] is based on laminar partition hierarchies which they define as follows. A *partition* of G is a cover consisting of disjoint clusters of nodes. A *laminar partition hierarchy* $\mathcal{P} = \{P_0, P_1, \ldots, P_{h'}\}$, where $h' = \lceil \log D \rceil$, has the following properties: (i) $P_{h'}$ is a single cluster that consists of all nodes in V; (ii) each P_i is a partition with diameter at most 2^i; (iii) each cluster in P_i is completely contained in some cluster in P_{i+1}, for $0 \leq i \leq h' - 1$. They also define the notion of α-*padded* node $v \in V$ with respect to $\mathcal{P}$ to be a node whose $\alpha 2^i$-neighborhood is included in a cluster of P_i in every level i, where $0 \leq i \leq h'$. They prove the existence of a family of $l = \mathcal{O}(\log n)$ laminar partition hierarchies $\mathcal{F} = \{\mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^l\}$, such that every node $v \in V$ is $\Omega(1/\log n)$-padded in at least one of the partition hierarchies in $\mathcal{F}$. The construction in [62] is randomized, and its correctness (padding property) holds with high probability. The family of hierarchical partitions can be computed in polynomial time. The authors also mention that the construction can be de-randomized with standard techniques.

Lemma 5.3.1 *There is a $(\mathcal{O}(\log n), \mathcal{O}(\log n))$-labeled cover hierarchy $\mathcal{Z}$, which can be constructed in deterministic polynomial time.*

Proof. We transform the family of laminar partition hierarchies $\mathcal{F}$ to an appropriate (σ, χ)-labeled cover hierarchy $\mathcal{Z}$. Let $a = 1/(c \log n)$ be the padding of $\mathcal{F}$ for some constant c. The cover $Z_i \in \mathcal{Z}$ is obtained from the union of all the level $j_i = i + \lfloor \log(c \log n) \rfloor$ partitions, namely, $Z_i = \bigcup_{\mathcal{P} \in \mathcal{F}} P_{j_i}$, for $1 \leq i \leq h$; in case $j_i > h'$, then, we use $Z_i = \bigcup_{\mathcal{P} \in \mathcal{F}} P_{h'}$. We set $Z_0 = V$, namely every node in level 0 is a cluster.

The locality of Z_i is $\gamma_i \geq \alpha 2^{j_i} \geq 2^{i-1} \cdot c \log n / c \log n = 2^{i-1}$, since a-padding implies that there is a cluster C in partition level j_i that includes a node u and its $\alpha 2^{j_i}$ neighborhood, and this cluster C appears in level i of $\mathcal{Z}$. Note that according to the definition of the partition, $\mathrm{diam}(Z_i) \leq 2^{j_i} \leq 2^i \cdot c \log n \leq 2c\gamma_i \log n$. Therefore, we can set $\sigma = 2c \log n$.

We can get a χ-labeling of each cover Z_i as follows. If a cluster $X \in Z_i$ came from partition hierarchy $\mathcal{P}^k$ then it obtains label $\lambda(X) = k$. This implies that we will have $\chi = l = \mathcal{O}(\log n)$ labels. The resulting labeling is valid, since for each level $Z_i \in \mathcal{Z}$, each cluster is obtained from a different partition hierarchy in $\mathcal{F}$, and thus we can not have any two clusters in $Z_i(u)$ with the same label. □

We normalize the (σ, χ)-labeled cover hierarchy $\mathcal{Z}$ obtained from Lemma 5.3.1 to satisfy the following properties which will be useful in our algorithms:

i. At level 0 each node in V belongs to exactly one cluster which consists only of the node itself.

ii. Cover Z_h (highest level) contains χ copies of the cluster that contains all nodes V, where $\chi = \mathcal{O}(\log n)$, each copy obtained from a different partition hierarchy in $\mathcal{F}$. We will keep only one copy and remove the rest so that there is only one cluster at level h of the hierarchy.[2]

iii. In any level i, $1 \leq i \leq h-1$, of $\mathcal{Z}$ each node $u \in V$ belongs to exactly χ clusters (one cluster from each partition hierarchy of $\mathcal{F}$); that is, $|Z_i(u)| = \chi$. Repeated clusters will be treated as different and will be assigned a different label.

5.3.3 Spiral Paths

Let $\mathcal{Z} = \{Z_0, Z_1, \ldots, Z_h\}$ be a (σ, χ)-labeled cover hierarchy as obtained from Lemma 5.3.1. Based on $\mathcal{Z}$, we define a path $p(u)$ for each node $u \in V$ which will refer to as the "spiral" path of u. The path $p(u)$ is built by visiting designated leader nodes in all the clusters that u belongs to starting from level 0 up to h. In each level, the clusters are visited according to the order of their

[2]To be more precise $\Theta(\log \log n)$ of the highest covers are equal to Z_h but in those (except Z_h) we will not remove the replicated clusters.

labels. From an abstract point of view, the path forms a spiral which slowly unwinds outwards while it visits cluster leaders of higher levels which are possibly further away from u.

Let $X_{i,j}(u) \in Z_i(u)$ denote the cluster at level i, $1 \leq i \leq h-1$, that u belongs to and has label j. We will refer to level i, label j, as the *sub-level* (i,j). Note that level i consists of χ sub-levels $(i,1),(i,2),\ldots,(i,\chi)$, for $1 \leq i \leq h-1$. Levels 0 and h are special cases which consist of a single sub-level each which for convenience we denote as $(0,\chi)$ and $(h,1)$, respectively. We can order the sub-levels lexicographically so that $(i,j) < (i',j')$ if $i < i'$, or $i = i'$ and $j < j'$. We define the function $\mathrm{next}(i,j)$ (resp. $\mathrm{prev}(i,j)$) to return the sub-level immediately higher (resp. lower) than (i,j).

In every cluster X we choose a designated *leader* node chosen arbitrarily which we denote as $\ell(X)$. Denote the leader of cluster $X_{i,j}(u)$ as $\ell_{i,j}(u) = \ell(X_{i,j}(u))$. Since Z_h consists of a single sub-level it has a unique leader which we denote $\ell_{h,1}(u) = r$. Trivially, every node $u \in V$ is a leader of its own cluster at level 0, $\ell_{0,\chi}(u) = u$.

For any pair of nodes $u, v \in V$, let $s(u,v)$ denote a shortest path from u to v. For any set of nodes $u_1, u_2, \ldots, u_k \in V$, let $s(u_1,u_2,\ldots,u_k)$ denote the concatenation of shortest paths $s(u_1,u_2),\ s(u_2,u_3),\ldots,\ s(u_{k-1},u_k)$. The spiral path $p(u)$ is formed by taking the concatenation of the shortest paths that connect the ascending sequence of leaders starting from node u (sub-level $(0,\chi)$) up to node r (sub-level $(h,1)$). Formally, we define the spiral path as follows:

Definition 10 (spiral path) *The spiral path of node u is:*

$$p(u) \;=\; s(u, \underbrace{\ell_{1,1}(u),\ldots,\ell_{1,\chi}(u)}_{\textit{level } 1}, \underbrace{\ell_{2,1}(u),\ldots,\ell_{2,\chi}(u)}_{\textit{level } 2}, \ldots, \underbrace{\ell_{h-1,1}(u),\ldots,\ell_{h-1,\chi}(u)}_{\textit{level } h-1}, r).$$

We say that two paths *intersect* if they have a common node. We also say that two spiral paths intersect at level i if they visit the same leader node at level i.

Lemma 5.3.2 *For any two nodes $u, v \in V$, their spiral paths $p(u)$ and $p(v)$ intersect at level* $\min\{h, \lceil \log(\mathrm{dist}(u,v)) \rceil + 1\}$.

Proof. It is trivial to see that $p(u)$ and $p(v)$ intersect in level h at node r. Suppose $\iota = \lceil \log(\mathrm{dist}(u,v)) \rceil + 1 \leq h$. From the definition of $\mathcal{Z}$ from Section 5.3.1, the clusters at level ι have locality $\gamma_\iota = 2^{\iota-1} \geq \mathrm{dist}(u,v)$. Thus, some cluster $X \in Z_\iota(u)$ will contain v. Therefore, the paths $p(u)$ and $p(v)$ intersect in leader node $\ell(X)$. □

5.3.4 Canonical Paths

In the analysis of the distributed directory-based consistency algorithm, we will examine paths which are obtained from fragments of spiral paths. These paths start at level 0 and are concatenations of shortest paths connecting leaders at successive sub-levels, where pairs of successive leaders are from clusters of the same node. One such example is shown in Fig. 5.2 on the right which depicts a canonical path q with down-pointing arrows between node w and node v_6 obtained from the fragments of the spiral paths of nodes u and v. For w the sub-path of q between v_4 to v_6 is the fragment of the spiral path of v, the sub-path between u_2 to v_4 is the fragment of the spiral path of u, and the sub-path between w to u_2 is the fragment of its own spiral path. We will refer to such paths as *canonical*. Formally, we define the canonical paths as follows:

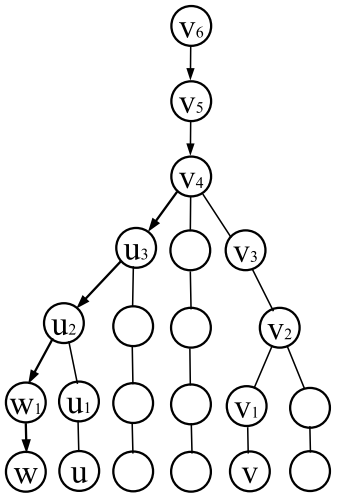


Figure 5.2: Illustration of a canonical path.

Definition 11 (canonical path) *A canonical path q up to sub-level $(k,\iota) \leq (h,1)$ is of the form*

$$q = s(x_{0,\chi}, \underbrace{x_{1,1}, \ldots, x_{1,\chi}}_{\textit{level } 1}, \underbrace{x_{2,1}, \ldots, x_{2,\chi}}_{\textit{level } 2}, \ldots, \underbrace{x_{k,1}, \ldots, x_{k,\iota}}_{\textit{level } k}),$$

such that for any two consecutive nodes $x_{i,j}$ and $x_{\mathrm{next}(i,j)}$, where $(0,\chi) \leq (i,j) < (k,\iota)$, there is a node $y \in V$ with $x_{i,j} = \ell_{i,j}(y)$ and $x_{\mathrm{next}(i,j)} = \ell_{\mathrm{next}(i,j)}(y)$.

We will refer to $x_{0,\chi}$ and $x_{k,\iota}$ as the *bottom* and *top* nodes of q, respectively. The bottom node is always at level 0. A canonical path can be either *partial* when the top node is below level h

(below the level of the root), or *full* when the top node is the root r. A spiral path $p(u)$ is a full canonical path which is derived from Definition 11 by taking y to be equal to u at every sub-level up to $(h, 1)$. Any prefix of a spiral path is a partial canonical path. We continue to bound the length of a canonical path when we use the (σ, χ)-labeled cluster hierarchy from Lemma 5.3.1.

Lemma 5.3.3 *For any canonical path q up to level k (and any sub-level (k, ι)), $\text{length}(q) \leq c_3 2^{k+2} \log^2 n$, for some constant c_3.*

Proof. Consider two consecutive nodes $x_{i,j}, x_{\text{next}(i,j)} \in q$, where $(0, \chi) < (i, j) < (k, \iota)$. From the definition of canonical paths, there is a node $y \in V$ with $x_{i,j} = \ell_{i,j}(y)$ and $x_{\text{next}(i,j)} = \ell_{\text{next}(i,j)}(y)$. Therefore,

$$\begin{aligned} \text{dist}(x_{i,j}, x_{\text{next}(i,j)}) &= \text{dist}(\ell_{i,j}(y), \ell_{\text{next}(i,j)}(y)) \\ &\leq \text{dist}(y, \ell_{i,j}(y)) + \text{dist}(y, \ell_{\text{next}(i,j)}(y)) \\ &\leq \text{diam}(X_{i,j}(y)) + \text{diam}(X_{\text{next}(i,j)}(y)) \end{aligned}$$

We explore two cases:

i. $\text{next}(i, j) = (i, j + 1)$: clusters $X_{i,j}(y)$ and $X_{\text{next}(i,j)}(y)$ are at the same level i. We have $\text{diam}(X_{i,j}(y)) \leq \sigma\gamma_i$ and $\text{diam}(X_{\text{next}(i,j)}(y)) \leq \sigma\gamma_i$. Since from Lemma 5.3.1 $\sigma = \mathcal{O}(\log n)$, and $\gamma_i = 2^{i-1}$, we get $\sigma\gamma_i \leq c_1 2^{i-1} \log n$, for some constant c_1. Thus, $\text{dist}(x_{i,j}, x_{\text{next}(i,j)}) \leq c_1 2^i \log n$.

ii. $\text{next}(i, j) = (i+1, 1)$: clusters $X_{i,j}(y)$ and $X_{\text{next}(i,j)}(y)$ are at levels i and $i+1$, respectively. We have that $\text{diam}(X_{i,j}(y)) \leq \sigma\gamma_i \leq c_1 2^{i-1} \log n$ and $\text{diam}(X_{\text{next}(i,j)}(y)) \leq \sigma\gamma_{i+1} \leq c_1 2^i \log n$. Which gives $\text{dist}(x_{i,j}, x_{\text{next}(i,j)}) \leq c_1(2^{i-1} + 2^i) \log n \leq c_1 2^{i+1} \log n$.

We define the following subpaths of q:

$$\begin{aligned} q_i &= s(x_{i-1,\chi}, x_{i,1}, x_{i,2}, \ldots, x_{i,\chi}), \text{ for } 1 \leq i < k \\ q_k &= s(x_{k-1,\chi}, x_{k,1}, x_{k,2}, \ldots, x_{k,\iota}) \end{aligned}$$

When we apply case ii for the first two nodes in q_i and case i for the remaining pairs of nodes, we obtain $\text{length}(q_i) \leq \chi c_1 2^i \log n$. Since from Lemma 5.3.1 $\chi = \mathcal{O}(\log n)$, we have that $\chi \leq c_2 \log n$, for some constant c_2. Therefore, $\text{length}(q_i) \leq c_1 c_2 2^{i+1} \log^2 n$. Similarly, $\text{length}(q_k) \leq c_1 c_2 2^{k+1} \log^2 n$. Finally, we obtain:

$$\begin{aligned} \text{length}(q) &= \sum_{i=1}^{k} q_i \leq c_1 c_2 (\log^2 n) \sum_{i=1}^{k} 2^{i+1} \\ &\leq c_1 c_2 2^{k+2} \log^2 n \leq c_3 2^{k+2} \log^2 n, \end{aligned}$$

for some constant $c_3 = c_1 c_2$. □

5.4 The Spiral Protocol

We present our protocol (Algorithms 6–8) which implements a DTM for shared objects over a graph G.

5.4.1 Protocol Overview

Consider some shared object ξ. The protocol guarantees that any moment of time only one node holds the shared object ξ which is the *owner* of the object. The owner is the only node who can modify the object (write the object); the other nodes can only access the object for read. Our protocol provides three basic operations to access ξ:

- *publish*(ξ): this operation is issued by the creator of ξ when the object is introduced into the network so that other nodes can find it. Node u becomes the first owner of ξ. Note that the *publish* operation is applied only once for ξ when the object is created.

- *lookup*(ξ): a node issues this operation to obtain a read-only copy of the object from the owner (without changing the owner).

- *move*(ξ): a node issues this operation to become the owner of the object in order to be able to modify it. The object moves from the previous owner to the new, by putting an object

copy to the new owner and invalidating the copy from the previous owner. If necessary, a *move* operation invalidates also the read-only copies of the previous owner.

The three operations of the protocol are implemented upon a (σ, χ)-labeled cover hierarchy $\mathcal{Z}$ provided by Lemma 5.3.1. Let r be the root node of G as specified by $\mathcal{Z}$. The basic idea is to maintain a *directory path* which is a directed path from the root node r to the bottom-level node that currently owns the shared object ξ. Initially, the directory path is formed from the spiral path $p(u)$ of the creator node u when it issues the *publish*(ξ) operation by assigning pointers along the edges of $p(u)$ directed toward u.

A *lookup*(ξ) operation issued by a node v uses its own spiral path $p(v)$ which intersects with the directory path and then leads to the object ξ. Let u be the owner of the object. Let x be the intersecting node of spiral path $p(v)$ and the directory path to u. The intersecting node x is guaranteed to exist because in the worst case scenario the paths intersect at the root r. The *lookup* is implemented in two phases: (i) in the *up phase*, a request message is sent from v upward in the hierarchy $\mathcal{Z}$ along the spiral path $p(v)$ towards the root r until the request intersects at a node (i.e. node x) with the directory path; (ii) in the *down phase*, the request message follows the directory path from node x to the object owner; then the owner sends a copy of ξ to v (along some shortest path).

A *move*(ξ) operation issued by v is similar to the *lookup* operation, with the only difference that it simultaneously modifies the directory path to point toward v. In the up phase, the *move* operation sets the directions of the edges in the fragment of $p(v)$ between v and x to point toward v. In the down phase it deletes the downward pointers (or links) in the fragment of the directory path from x to u. Now the new directory path points toward v. When the down phase reaches u, v obtains a copy of the object and invalidates u's copy. This process has resulted to a canonical directory path that consists of two spiral path fragments, a fragment of u's spiral path between r and x and a fragment of v's spiral path between x and v. Subsequent *move* operations may result into further fragmentation of the directory path into multiple (more than two) spiral path fragments.

A *downward path* is any canonical path whose edges are directed to point down. The directory path is the only *full* downward path in the network which has its top node at root r. *Move* operations result to the creation of *partial* downward paths whose top nodes are at levels below r. Partial downward paths coexist temporarily with the directory path during the up phase or the down phase of *move* requests. In the example above, before v's *move* request reaches the intersection x there is a partial downward path to v that coexists with the old directory path to u. In the down phase after the *move* request reaches the intersection x there is a partial downward path to u that coexists with the new directory path to v. These partial downward paths are temporary and last up to the end of the respective phase, after which only the directory path will exist.

Concurrent *lookup* and *move* requests may be served through partial downward paths instead of the directory path. These requests are *queued* while the new directory path is being formed. For example, consider the scenario where a *lookup* operation is issued by a node w concurrently with the *move* operation of v. Suppose also that the *lookup* and *move* requests intersect in their up phase paths before their requests reach the directory path to u. Then the *lookup* request will descend down to v through a partial downward path while the *move* request ascends to x. The *lookup* will request the read-only copy of the object ξ from v. However, v may not have the copy of ξ yet. In this case, w's request is queued in v and it will be served when v receives ξ.

In the scenario where w's operation was a *move*, then two partial downward paths would coexist at the same time with the directory path until the up phases of u and v intersect. After that again two partial paths can coexist until the down phase of w reaches v and before the up phase of v reaches x. The result is that the *move* request from w will be queued after v. Similarly, multiple concurrent *move* operations temporarily lead to the formation of multiple partial downward paths to the origins of the requests. The *move* operations get queued in the origin nodes forming a distributed queue of *move* operations. Eventually, every *move* operation will be served by passing the object from the current owner at the head of the queue to the next node in the queue.

Last, we want to note that after several move operations the directory path may become highly fragmented. In such cases a *lookup* request may not find immediately the directory path to the

Algorithm 6: Publish request handling

```
// y = ℓ_{i,j}(u) is the level (i,j) leader node, x(= ℓ_{prev(i,j)}(u)) is its
   child, and parent_{p(u)}(y) is its parent, all in the spiral path p(u)
   of the leaf node u.
1 When y receives m = ⟨u, up, publish⟩ from x:
2 begin
3     y.link = x;                                        // set downward pointer
4     if y is not a root node then
5         send m to parent_{p(u)}(y) ;                   // continue up phase
```

shared object ξ, even if the *lookup* originates near ξ. In order to avoid this situation and guarantee efficient *lookup* we introduce the notion of a *special parent* node, such that whenever a downward link is formed at a node z the special parent of z is also informed about z holding a downward pointer. The special parent is selected in such a way that any nearby *lookup*, close to z will either reach z or its special parent. The details appear below in the special description of the algorithm.

5.4.2 Detailed Description

We describe Algorithms 6–8 that implement *publish*, *lookup*, and *move*, respectively. We denote a message m (in the system) by a triple $\langle id, phase, type\rangle$, where $id \in \mathbb{N}$ is the identifier of the request, $phase \in \{$*up*, *down*$\}$ are the phases, and $type \in \{$*publish*, *lookup*, *move*$\}$ are the type of requests.

When a transaction at some node v needs the shared object ξ for read (resp. write) it consults with the transaction interface of the proxy module first. If the shared object is in the local cache of that node the proxy immediately provides that object to the transaction. If not, the transaction interface of the proxy module issues a *lookup* message $m = \langle v,$ *up*, *lookup*$\rangle$ (resp. *move* message $m = \langle v,$ *up*, *move*$\rangle$) to the network interface. After this, the network interface handles the *lookup* (resp. *move*) request using Algorithm 7 (resp. Algorithm 8) to fetch the shared object, based on the type of request it receives from the transaction interface. To publish the created object, if any, by a node u, the transaction interface simply issues a *publish* message $m = \langle u,$ *up*, *publish*$\rangle$ to the network interface which is then handled by Algorithm 6. The network interface is also responsible

Algorithm 7: Lookup request handling

```
// y = ℓ_{i,j}(v) is the level (i,j) leader node and parent_{p(v)}(y) is its
   parent, both in the spiral path p(v) of the leaf node v.
   Moreover, x(= ℓ_{prev(i,j)}(v)) is y's child in the lookup up phase and
   x(= ℓ_{next(i,j)}(v)) is y's parent in the lookup down phase.
1  When y receives m = ⟨v, phase, lookup⟩ from x:
2  begin
3      if m = ⟨v, up, lookup⟩ then                                  // lookup up phase
4          if y.link = ⊥ then                                              // link null
5              if y.slink.isEmpty() = True then                // y's slink list is empty
6                  send m to parent_{p(v)}(y);
7              else send ⟨v, down, lookup⟩ to y.slink.sendFirst() ;
8          else send ⟨v, down, lookup⟩ to y.link;
9      if m = ⟨v, down, lookup⟩ then                              // lookup down phase
10         if y is a leaf node then
11             send the read-only copy of ξ to v and remember v;
12         else send m to y.link;
```

for handling *transient* messages that it will receive from other nodes, besides providing interface to the local transactions.

Before giving details of Algorithms 6–8, we define notations of parent and special-parent, that we use in the protocol description. We denote a *parent* node y of a node x in the spiral path $p(u)$ of a bottom-level node u as $y = \text{parent}_{p(u)}(x)$, where if $y = \ell_{i,j}(u)$ is the level (i, j) leader of u in $p(u)$ then $x = \ell_{\text{prev}(i,j)}(u)$. Note that $\ell_{0,\chi}(u) = u$. The special-parent node is defined as follows:

Definition 12 (special-parent) *We denote a* special-parent *node of y at level (i, j) in the spiral path $p(u)$ of u as $\text{sparent}_{p(u)}(y)$, such that $\text{sparent}_{p(u)}(y)$ is the leader node of one of the clusters $X \in Z_k(u)$ at level k, where $k = i + 4 + 2\log\log n + \log c_3$, that contains the γ_k-neighborhood of $y \in X$. That is, $\text{sparent}_{p(u)}(y)$ is some* ancestor *node of y at level k in the spiral path $p(u)$.*

Each node knows its parent and special-parent node in the hierarchy, except the root node, whose parent and special-parent are both $\perp$ (null). A node might have (i) a $link$ towards one of its children (otherwise it is $\perp$); the link at the root is not $\perp$, and (ii) a $slink$ towards *special-child* node y from its special-parent node $\text{sparent}_{p(u)}(y)$ (otherwise it is $\perp$). Note that, as a leader node

Algorithm 8: Move request handling

```
// y = ℓ_{i,j}(v) is the level (i,j) leader node, parent_{p(v)}(y) is its parent,
   and sparent_{p(v)}(y) is its special-parent, all in the spiral path p(v)
   of the leaf node v. Moreover, x(= ℓ_{prev(i,j)}(v)) is y's child in the
   lookup up phase and x(= ℓ_{next(i,j)}(v)) is y's parent in the lookup down
   phase.
1  When y receives m = ⟨v, phase, move⟩ from x:
2  begin
3      if m = ⟨v, up, move⟩ then                                   // move up phase
4          oldlink ← y.link;                                       // remember link
5          y.link = x;                                        // set downward pointer
6          sparent_{p(v)}(y).slink.add(y);        // Add pointer to y in slink of y's
           special parent
7          if oldlink = ⊥ then
8              send m to parent_{p(v)}(y);                        // continue up phase
9          else send ⟨v, down, move⟩ to oldlink;
10     if m = ⟨v, down, move⟩ then                                 // move down phase
11         if sparent_{p(v)}(y).slink.has(y) = True then       // y is in the slink list
12             sparent_{p(v)}(y).slink.remove(y);   // erase pointer of special parent
13         if y is not a leaf node then                       // is owner not found yet?
14             oldlink ← y.link;
15             y.link ← ⊥;
16             send m to oldlink; // continue down phase
17         else send the writable copy of ξ to v;
18         invalidate(ξ) from the owner node and the read-only copies from other nodes ;
```

w_k of a particular level k cluster might participate as a special-parent node for several level (i, j) cluster leader nodes, we maintain a list of special-children for $slink$ at each leader node w_k.

Publish request handling. A shared object created at a leaf node u is published by setting each $y.link = x$, a single directory path from the root node r to u (Algorithm 6), by visiting each level clusters in the hierarchy along the spiral path $p(u)$. This operation is served in the up phase only. For example, Fig. 5.1a shows the object published by v using a *publish* operation such that there is a directory path from the root u_3 to the leaf node v along $p(v)$.

Lookup request handling. A *lookup* request can be started by a leaf node v by creating a message m of type *lookup* (Lines 4,11 of Algorithm 7). It is served in two phases (Lines 4−12 of

Algorithm 7). In the first phase (Lines 4−8), called up phase, the parent nodes (i.e., y) at increasing levels are probed until a non-null directory path is found along the spiral path of the requesting node v (Lines 4−6). At each level (i, j), x initiates a probe to its parent node y. If the probe finds no directory path in parent node (i.e., link of y is null and slink list is empty) at that level (i, j), it repeats the process at the next higher level immediately above level (i, j). If the probe discovers a directory path, then the second phase, called down phase, starts (Lines 11−12); directory path pointers are followed to reach the leaf node that either holds the object or will hold the object soon (Line 12). Note that if the directory path is discovered in y's slink list, then the path formed by the first non-null link node is followed (Line 7). As soon as ξ becomes available, a read-only copy of ξ is sent directly to v and v is included in the owner node's list of nodes which has the read-only copy of ξ (Line 13).

Move request handling. Similar to *lookup*, a *move* request can be started by the leaf node u by creating a message m of type *move* (Lines 16,22 of Algorithm 8) The *move* operation also operates on two phases (Lines 16−18 of Algorithm 8). In the up phase (Lines 16−9), the request probes the parent nodes at increasing levels on the spiral path $p(v)$ until non-null directory path is found (Lines 7−9). While probing upward, the pointer from the leader node at level (i, j) is set to point the leader node at level immediately lower than level (i, j) (Lines 4,5). This information is also forwarded to special-parent node $\mathrm{sparent}_{p(v)}(y)$ for the use of it in *lookup* operation (Line 6). For the down phase (Lines 22−18), when the request finds a downward path at level (i, j), it first erases the information stored at $\mathrm{sparent}_{p(v)}(y)$, redirects its link to the leader node x, and descends to the child pointed by the new link (Lines 12−13). The request then follows the chain of downward pointers, setting each one to null, until it arrives at owner node (Line 17). This owner node either has the object ξ, or is waiting for it. When the object ξ is available, it is sent directly to the requested node v (Line 17), invalidating the local copy at the owner node and the read-only copies from other nodes (Line 18). Figs. 5.1b−5.1h show one complete *move* operation issued by node v for the object that is currently owned by node v.

Note that at any time a request locks at most one node along the spiral path or a downward path. The special parent node doesn't need to be locked because only one specific $slink$ needs to be updated without the need to lock the whole list of special links. We observe that it may be the case that the node y is participating as a cluster leader on many sub-level clusters. That is, sometimes y may be the leader node of level (i, j) and level $\mathrm{next}(i, j)$ clusters in $p(v)$. In such cases, to provide consistent data structures for different level operations, we add a duplicate node of y for each level it is participating as a cluster leader and create a virtual link between the duplicate and y itself in subsequent clusters.

5.5 Analysis of Spiral Protocol

We continue with the correctness proof of Spiral (Section 5.5.1) and give the performance analysis of *publish* and *lookup* requests (Section 5.5.2). We then focus on the performance analysis of *move* requests in sequential executions (Section 5.5.3). We then give the performance analysis of *move* requests in concurrent executions (Section 5.5.4).

5.5.1 Correctness

We first show that our protocol guarantees that every request will be served within finite time (no starvation) and the queue of successor requests for the shared object form a list (with no cycle). We then focus on proving the existence of a directory path from the root to a leaf node at any configuration. We extend the correctness results of Ballistic and Combine from [10, 77] and use them to prove the correctness of Spiral.

In the Spiral protocol, a request from a node v will be served as soon as v receives a copy of the object, which is read-only copy in case of a *lookup* request. A DTM protocol should be *responsive* so that every request issued by any node at any time is eventually served [77]. Overtaking can happen in Spiral when satisfying concurrent moves: a node v may issue a move operation at a later time than a node w, yet v's *move* request may be ordered before w's *move* request if v is

closer to the object than w. Nevertheless, we show that such overtaking can occur only during a bounded window of time, implying that every *move* request eventually completes.

As we assume FIFO communication between nodes, no multiple *move* requests can arrive at the owner node simultaneously because a *move* follows a path of pointers that it immediately removes. This holds when no new *move* request is issued from a node that is waiting for the object or that owns the object. In this setting, we can also observe that: (i) a move request can not be passed by another *move* request; and (ii) a *move* request generated by some node v does not visit the same node twice.

Lets denote by T_F the time for a *move* request to go upward from a leaf node v (requesting node) to the lowest common ancestor of v and a owner node u in the hierarchy $\mathcal{Z}$ following v's spiral path, and then down to u (or vice versa). Similarly, lets denote by T_M the time needed for an object to reach its requesting node from the owner node. These parameters are network-specific but finite, because requests are never blocked to reach from the requesting node to the owner node and vice versa. We assume that T_M also includes the time needed to invalidate existing read-only copy before moving a writable copy and the delay imposed by the contention manager in responding to the conflicting successor transaction request. In this setting, Herlihy and Sun [77] proved that, in their protocol Ballistic, every *move* request is satisfied within time $n \cdot T_F + n \cdot T_M$ from when it is generated (no starvation) [77, Theorem 1]. As a corollary, they prove that if a request r is generated at time t, then all requests generated after time $t + n \cdot T_F$ will be ordered after r; all requests generated prior to time $t - n \cdot T_F$ will be ordered before r (bounded overtaking) [77, Corollary 1]. Moreover, they prove that there exists no set of finite number of requests $R = \{r_1, r_2, \cdots, r_f\}$ whose successor links form a cycle (no cycle) [77, Lemma 1].

As our protocol also satisfies these aforementioned properties, we can obtain the following symmetric result for Spiral:

Lemma 5.5.1 (responsiveness) Spiral *is (i) starvation-free; (ii) overtaking of requests can occur only during a bounded window of time; and (iii) there exists no set of finite number of requests* $R = \{r_1, r_2, \cdots, r_f\}$ *whose successor links form a cycle.*

We now proceed with establishing the existence of a directory path from the root node to a leaf node at any configuration of the system. Let C_0 be the state after some owner node u finished *publish* operation (Algorithm 6) but before any node issues a *move* or a *lookup* request. The following result is straightforward:

Lemma 5.5.2 *At initial configuration, there is a directory path from the root to a leaf node.*

Thus, the root node has a downward link which is not null at initial configuration. As the initial directory path is changed only by *move* requests, we proceed with the analysis considering that *move* requests are the only requests in the system. The root will always have a downward link even if subsequent *move* requests change it. The proof of Attiya et al. [10, Theorem 1] states that if there is a downward link at some node v, then there is a downward path from v to a leaf node. As Spiral also maintains the downward links in similar way, the result below follows immediately from Lemma 5.5.2, since there is always a downward link at the root.

Lemma 5.5.3 (directory path) *At any configuration, there is a directory path from the root node to a leaf node.*

Recall that partial downward paths may temporarily coexist with the directory path during the up and down phase of *move* requests but these paths are temporary and last up to the end of the phase. Thus, we can make the following observation.

Observation 1 *At quiescent configuration, all partial downward paths disappear except a directory path from the root to a leaf node.*

5.5.2 Performance of *publish* and *lookup* Requests

Theorem 5.5.4 (*publish* cost) *The publish operation has communication cost $\mathcal{O}(D \cdot \log^2 n)$.*

Proof. Note that the *publish* operation adds links on the publishing leaf node's spiral path towards the root. The theorem immediately follows from Lemma 5.3.3, by noticing that the number of levels in the hierarchy $h = \lceil \log D \rceil + 1$, and that a spiral path is trivially a canonical path. □

We now focus on the stretch for the *lookup* operation. Lets assume that, after a node v finished *publish* operation, some node w issues a *lookup* request r for the shared object ξ at v and there is no other *lookup* request in the system. If there is no *move* request in the system, it is trivial to see that a *lookup* request r from w finds the directory path to the owner node v, at level $\lceil \log(\mathrm{dist}(w,v)) \rceil + 1$ (Lemma 5.3.2), following the spiral path $p(w)$, where $\mathrm{dist}(w,v)$ is the distance of the owner node v from the requesting node w. When there are *move* requests in the system, they change the ownership of the shared object according to their arrival at the owner node. When searching for the object, directory path to the node that issued *move* request is formed from the fragments of spiral paths of the nodes that previously owned the object. In this scenario, the *lookup* request r from w may not find the directory path to the shared object at level $\lceil \log(\mathrm{dist}(w,v)) \rceil + 1$ because the directory path to v might be deformed significantly such that the $2^{(\lceil \log(\mathrm{dist}(w,v)) \rceil + 1)}$-neighborhood of w has no information about the owner node v. Nevertheless, we guarantee that every *lookup* request finds the directory path not much higher than level $\lceil \log(\mathrm{dist}(w,v)) \rceil + 1$. Particularly, we prove that every *lookup* request issued by some node w for the shared object ξ at some owner node v at distance $\mathrm{dist}(w,v) \leq 2^i$ will find the $slink$ towards the directory path to v at some level k in the spiral path $p(w)$, where $k = i + 4 + 2\log\log n + \log c_3$.

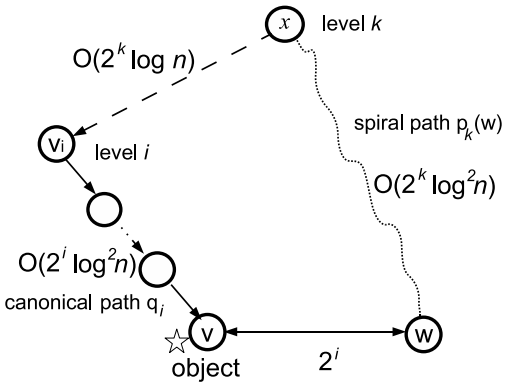


Figure 5.3: Special-parent.

Lemma 5.5.5 *If a node w issues a lookup request r for the shared object ξ currently owned by a node v which is at distance $\mathrm{dist}(w,v) \leq 2^i$ far from w, the spiral path $p(w)$ is guaranteed to either intersect with the directory path to v or find a slink to the directory path for the object at level at most k, where $k = i + 4 + 2\log\log n + \log c_3$.*

Proof. As shown in Figure 5.3, lets assume that v_i is the level i leader node in the canonical directory path q towards the owner node v and q_i is q's fragment up to level i. Assume also that

$x = \ell(X)$ is the leader of the cluster X at level $k = i + 4 + 2\log\log n + \log c_3$, which has a slink information to v_i (set by some *move* request following Algorithm 8), as in Definition 12. For a *lookup* request r issued by w to find the slink to v_i, X must include w, since the spiral path $p(w)$ of w visits the leaders of all clusters that contain it.

It suffices to show that the locality $\gamma_k = 2^{k-1}$ of X is at least the distance $\mathrm{dist}(v_i, w)$ between v_i and w to guarantee that X contains w. As $\mathrm{length}(q_i) \leq c_3 2^{i+2}\log^2 n$ (Lemma 5.3.3) and $\mathrm{dist}(w, v) \leq 2^i$, $\mathrm{dist}(v_i, w)$ is bounded by:

$$\begin{aligned}
\mathrm{dist}(v_i, w) &\leq \mathrm{length}(q_i) + \mathrm{dist}(w, v) \leq c_3 2^{i+2}\log^2 n + 2^i \\
&\leq c_3 2^{i+3} \cdot \log^2 n \leq 2^{\log c_3} \cdot 2^{i+3} \cdot 2^{\log\log^2 n} \\
&\leq 2^{i+3+\log\log^2 n + \log c_3} \leq 2^{i+3+2\log\log n + \log c_3} \\
&\leq 2^{k-1} = \gamma_k,
\end{aligned}$$

as needed. □

Next, we will bound the stretch of Spiral for *lookup*, $stretch_{\mathsf{Spiral},lookup} = C(r)/C^*(r)$, where $C(r)$ is the total communication cost of serving a *lookup* request r using the Spiral protocol and $C^*(r)$ is the optimal cost of serving r using the optimal algorithm.

Theorem 5.5.6 (*lookup* stretch) *The stretch of* Spiral *for a lookup operation is* $stretch_{\mathsf{Spiral},lookup} = \mathcal{O}(\log^4 n)$.

Proof. From Lemma 5.5.5, as a *lookup* request r from w for the shared object at owner node v is guaranteed to find a slink to the directory path q towards v at level at most k, the length of the spiral path $p_k(w)$ up to level k is bounded by $\mathrm{length}(p_k(w)) \leq c_3 2^{k+2}\log^2 n$ (Lemma 5.3.3), as a spiral path is trivially a canonical path. Similarly, $\mathrm{length}(q_i) \leq c_3 2^{i+2}\log^2 n$, where q_i is the canonical path of v up to v_i, the level i leader in q. From the locality property of $X(\in Z_k(w))$ (Lemma 5.5.5), $\mathrm{dist}(x, v_i) \leq \sigma\gamma_k \leq c 2^{k-1}\log n$, since $\sigma = \mathcal{O}(\log n)$ and $\gamma_k = 2^{k-1}$ as of Lemma

5.3.1. Therefore, the total cost of Spiral, $C(r)$, for the *lookup* request r (after substituting k by $i + 4 + 2\log\log n + \log c_3$) is bounded by:

$$\begin{aligned}
C(r) &= \text{length}(p_k(w)) + \text{dist}(x, v_i) + \text{length}(q_i) \\
&\leq c_3 2^{k+2}\log^2 n + c2^{k-1}\log n + c_3 2^{i+2}\log^2 n \\
&\leq c_3 2^{k+3}\log^2 n \leq c_3 2^{(i+4+2\log\log n+\log c_3)+3}\cdot\log^2 n \\
&\leq 128\cdot(c_3)^2\cdot 2^i\cdot 2^{2\log\log n}\cdot\log^2 n \\
&\leq 128\cdot(c_3)^2\cdot 2^i\cdot 2^{\log\log^2 n}\cdot\log^2 n \\
&\leq 128\cdot(c_3)^2\cdot 2^i\cdot\log^2 n\cdot\log^2 n \\
&\leq 128\cdot(c_3)^2\cdot 2^i\cdot\log^4 n.
\end{aligned}$$

As w and v are $\text{dist}(w, v) \leq 2^i$ apart, the optimal communication cost for the optimal protocol is at least bounded by $C^*(r) \geq 2^{i-1}$ for the *lookup* request r to get the shared object at v following the shortest path between w and v in G. Hence, the *lookup* stretch for Spiral is,

$$stretch_{\mathsf{Spiral},lookup} = \frac{C(r)}{C^*(r)} \leq \frac{128\cdot(c_3)^2\cdot 2^i\cdot\log^4 n}{2^{i-1}} = \mathcal{O}(\log^4 n),$$

as needed. □

5.5.3 Performance of *move* Requests in Sequential Executions

We give the performance analysis of Spiral in sequential executions. As *move* requests do not overlap with each other in sequential executions, the system attains quiescent configuration after a request is served and until a next request is issued for the shared object, i.e. a next request will be issued only after the current request finishes. We perform the following execution setup: Lets define a sequential execution of a set $\mathcal{E}$ of $l + 1$ requests $\mathcal{E} = \{r_0, r_1, \cdots, r_l\}$, where r_0 is the initial *publish* request and the rest are the subsequent *move* requests (we do not include *lookup*

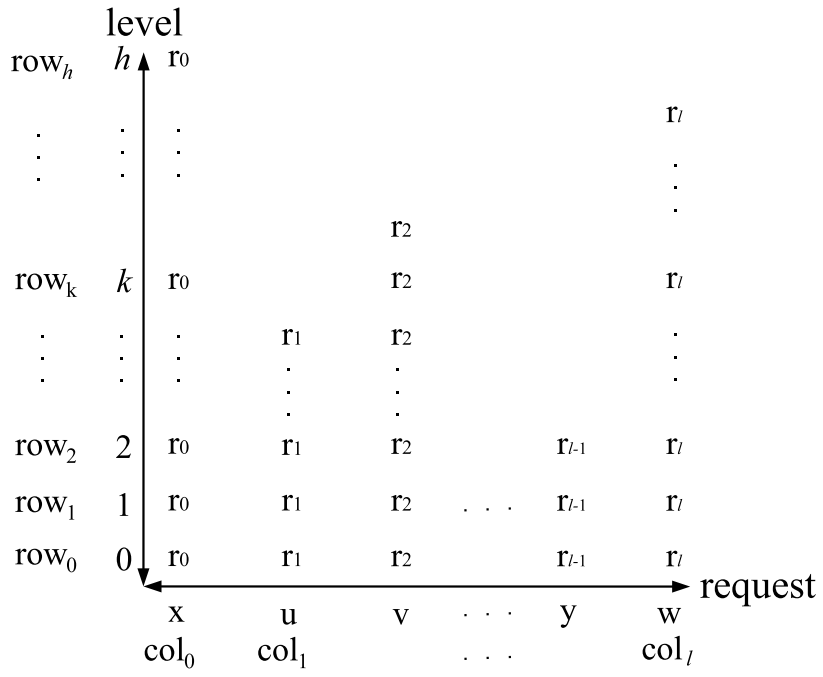


Figure 5.4: Illustration of a sequential execution.

operations in $\mathcal{E}$ since they do not add or remove links in the directory, and hence do not impact the performance of other *move* or *lookup* operations).

We define a two-dimensional array of size $h+1 \times l+1$ to help prove the stretch, where $h+1$ and $l+1$ are the number rows and columns, respectively. Moreover, we denote the $h+1$ rows as $\{row_0, row_1, \cdots, row_h\}$, and the $l+1$ columns as $\{col_0, col_1, \cdots, col_l\}$. All the locations of the array are initially empty ($\perp$) and $[0,0]$ is assumed to be the lower left corner element and $[h,l]$ be the upper right corner element.

Each $col_i, 0 \leq i \leq l$ keeps track of the levels visited by each request r_i in the hierarchy $\mathcal{Z}$ while searching for the object. The *peak level* for a request r_i is the maximum level reached by r_i in $\mathcal{Z}$. As $\mathcal{E}$ starts with r_0, the *publish* request, the peak level reached by r_0 is h, the maximum level in $\mathcal{Z}$, and r_0 is registered at all the locations of col_0 starting from $col_0[0]$ to $col_0[h]$. For each non-initial request (i.e., the *move* request) $r_i \in \mathcal{E}, i > 0$, issued by some node v, we keep track of it by registering at all the locations from 0 to k of col_i till the request sees the directory path at the leader node of some cluster at its peak level k in the spiral path $p(v)$. We have that $k \leq h$.

An example for one such execution of requests in $\mathcal{E}$ is given in Fig. 5.4, where $x, u, \cdots, w$ in the horizontal axis are $l+1$ requests in $\mathcal{E}$ and $0, 1, \cdots, h$ in the vertical axis are the levels. The entries in the first column are from the *publish* request $r_0 \in \mathcal{E}$ issued by a leaf node x that reached to level h (following Algorithm 6). The entries in the rest of the columns are from *move* requests r_1 to r_l when registering them in all the locations of the respective columns according to the peak level they reached while searching for the object in the up phase (following Algorithm 8).

Let $C^*(\mathcal{E})$ denote the total communication cost of serving all the requests in $\mathcal{E}$ using the optimal algorithm. Let $C(\mathcal{E})$ denote the total communication cost of serving all the requests in $\mathcal{E}$ using the Spiral protocol. We will bound the stretch $stretch_{\mathsf{Spiral},sequential} = \max_{\mathcal{E}} C(\mathcal{E})/C^*(\mathcal{E})$. For simplicity, in the stretch analysis, we consider only the cost incurred by the up phase of each *move* request. When we consider also the cost incurred by the down phase of each request, the stretch increases by a factor of 2 only.

We proceed with the necessary definitions that we use in the performance analysis (particularly, in the proofs of Lemmas 5.5.7 and 5.5.8) given below. For any $c, d, 0 \leq c < d \leq l$, we define a *valid pair* $W^j_{(c,d)}$ of two non-empty entries in $row_j, 0 \leq j \leq h$, as $W^j_{(c,d)} = (row_j[c], row_j[d])$, such that $row_j[c] \neq \perp$ and $row_j[d] \neq \perp$, and $\forall e, c < e < d$, $row_j[e] = \perp$. In other words, $W^j_{(c,d)}$ is a pair of two subsequent non-empty entries in a row. For example, as shown in Fig. 5.4, $W^k_{(0,1)}$ is not a valid pair, but $W^k_{(0,2)}$ is a valid pair because the entry between location 0 and 2 is $\perp$ in row_k.

Moreover, we denote by S_j the total count of the number of entries $row_j[i], 0 \leq i \leq l$, such that $row_j[i] \neq \perp$, and by W_j the total number of pairs $W^j_{(c,d)}$ in it. We have that $W_j = S_j - 1$. For example, if we consider only the five requests listed in Fig. 5.4 at row_k, $S_k = 3$ and $W_k = 2$. Note also that there are at most $S_j = l$ entries and $W_j = l - 1$ pairs in each $row_j, 0 \leq j \leq h$, for $\mathcal{E}$.

Lemma 5.5.7 *For the sequential execution $\mathcal{E}$, $C^*(\mathcal{E}) \geq \max_{1 \leq k \leq h}(S_k - 1)2^{k-1}$, where $h = \lceil \log D \rceil + 1$.*

Proof. Denote by $C^*_k(\mathcal{E})$ the optimal communication cost of the optimal protocol for all the requests in $\mathcal{E}$ that reach level k in the hierarchy $\mathcal{Z}$, while probing for the shared object in their up

phase. According to the execution setup, S_k are the number of requests in $\mathcal{E}$ that reach level k, and W_k are the total number of valid pairs at that level. It is known from Lemma 5.3.2 that, if the spiral paths $p(u)$ and $p(v)$ of any two requests r_i and r_{i+1} issued by the leaf nodes u and v intersect at level k, $\mathrm{dist}(u, v) \geq 2^{k-1}$, since otherwise their spiral paths would intersect at level $k-1$ or lower. Therefore, the optimal communication cost $C_k^*(\mathcal{E})$ is bounded by at least the total distance between the W_k valid pairs of requests at level k, i.e., $C_k^*(\mathcal{E}) \geq W_k \cdot 2^{k-1} \geq (S_k - 1) \cdot 2^{k-1}$, as $W_k = S_k - 1$. Considering all the levels from 1 to h, it is safe to say that the optimal communication cost, $C^*(\mathcal{E})$, is bounded by at least the maximum over $C_k^*(\mathcal{E})$, $1 \leq k \leq h$. Therefore, $C^*(\mathcal{E}) \geq \max_{1 \leq k \leq h} C_k^*(\mathcal{E}) \geq \max_{1 \leq k \leq h}(S_k - 1) \cdot 2^{k-1}$. We do not consider cost for level 0 in optimal cost because there is no communication that reaches that level. □

Lemma 5.5.8 *For the sequential execution $\mathcal{E}$, $C(\mathcal{E}) \leq \sum_{k=1}^{h} c_3(S_k - 1)2^{k+2} \log^2 n$, where $h = \lceil \log D \rceil + 1$.*

Proof. Similar to Lemma 5.5.7, denote by $C_k(\mathcal{E})$ the total communication cost of Spiral for all the requests in $\mathcal{E}$ that reach level k in the hierarchy $\mathcal{Z}$, while probing the shared object in their up phase. According to the execution setup, $C_k(\mathcal{E})$ is at most the total communication cost for serving S_k requests in $\mathcal{E}$ that reach level k using Spiral. It is known from Lemma 7.2.2 that the total communication cost for each request that reaches level k in the hierarchy $\mathcal{Z}$ is bounded by at most $c_3 2^{k+2} \log^2 n$. Since, in row_k, there are $S_k - 1$ non-initial requests that reach level k, we have that $C_k(\mathcal{E}) \leq (S_k - 1)c_3 2^{k+2} \log^2 n$. By combining the costs for each levels, the total communication cost of Spiral is bounded by $C(\mathcal{E}) = \sum_{k=1}^{h} C_k(\mathcal{E}) \leq \sum_{k=1}^{h} c_3 \cdot (S_k - 1) \cdot 2^{k+2} \cdot \log^2 n$, in the worst-case. We do not consider communication cost for level 0 in total cost because there is no communication that reaches that level. □

We now give the central result of the sequential analysis:

Theorem 5.5.9 (*move* stretch in sequential executions) *The stretch of the* Spiral *protocol is $stretch_{\mathsf{Spiral},sequential} = \mathcal{O}(\log^2 n \cdot \log D)$ for sequential executions.*

Proof. Since the execution $\mathcal{E}$ is arbitrary, we obtain, from Lemmas 5.5.7 and 5.5.8, the stretch of the Spiral protocol bounded by

$$\begin{aligned}
stretch_{\mathsf{Spiral},sequential} &\leq \frac{C(\mathcal{E})}{C^*(\mathcal{E})} \\
&\leq \frac{\sum_{k=1}^{h} c_3(S_k-1)2^{k+2}\cdot\log^2 n}{\max_{1\leq k\leq h}(S_k-1)\cdot 2^{k-1}} \\
&\leq \frac{8c_3\log^2 n\sum_{k=1}^{h}(S_k-1)\cdot 2^{k-1}}{\max_{1\leq k\leq h}(S_k-1)\cdot 2^{k-1}} \\
&\leq \frac{8c_3\log^2 n\cdot h\cdot\max_{1\leq k\leq h}(S_k-1)}{\max_{1\leq k\leq h}(S_k-1)} \\
&\leq 8c_3\log^2 n\cdot h \\
&\leq 8c_3\log^2 n\cdot(\lceil\log D\rceil+1) \\
&= \mathcal{O}(\log^2 n\cdot\log D),
\end{aligned}$$

as $h=\lceil\log D\rceil+1$. □

5.5.4 Performance of *move* Requests in Concurrent Executions

The performance analysis of Spiral given in Section 5.5.3 is valid only for sequential executions and does not apply to concurrent executions because the adversary is not allowed to gain by ordering concurrent requests in a smarter way. A sequential execution assumes that the Spiral protocol and the optimal algorithm would queue requests in the same order. However, a concurrent execution can change the order and hence affect the performance. In this section, we study the following *one-shot* instance of the concurrent execution: At time t, as soon as a node finished publishing a shared object ξ using a *publish* operation started at time 0, $R\subseteq V$ of nodes issue a *move* request each concurrently and no further requests occur. We calculate the total communication cost of all the requests including the *publish* operation (similar as of Section 5.5.3) to provide the stretch in one-shot situation.

For the performance analysis we assume that the network model is synchronous (the protocol does not require synchrony for correctness) along with the assumptions of Section 5.2. We assume that a time unit is of duration required for a message send by a node to reach a destination node that is a unit distance far from it. We define for level i *period* of time duration $\Phi(i) = 4c_3 2^i \log^2 n, 0 \leq i \leq h$ (Lemma 5.3.3), i.e., the longest distance traversed in level i (including sub-levels) following the canonical (or spiral) path. Moreover, the $\Phi(\cdot)$ are aligned in such a way that $\Phi(i)$ and $\Phi(i-1)$ starts at the same time, i.e., two periods of duration $\Phi(i-1)$ can be perfectly accommodated at a level i period $\Phi(i)$. We assume also that all requests proceed in rounds. A *round* is of duration $\Phi(h)$, where $h = \lceil \log D \rceil + 1$, and it has h overlapping aligned periods. In a round, there is 1 period for level h, 2 periods for level $h-1$, 4 periods for level $h-2$, and so on, so that there are 2^{h-k} periods for level k. In a period, each leader node in the canonical path (or spiral path) can exchange a message with each of its neighbors (parents or children). A leader node $w_{i,\chi}$ of the highest sub-level cluster at level i in a spiral path $p(w)$ forwards the request to a leader node $w_{i+1,0}$ of the lowest sub-level cluster at level $i+1$ at the end of its phase $\Phi(i)$. Similarly, any request that arrives to a leader $w_{i,j}$ of a sub-level j cluster at level i is processed during $\Phi(i)$ and sent to higher sub-level cluster towards $w_{i,\chi}$.

We proceed with a short description of the execution of concurrent requests: At time zero, a node issues a *publish* operation r_0 to publish the object. As soon as the *publish* operation r_0 finishes at time t, l nodes issue one *move* request each concurrently, namely $\mathcal{R} = \{r_0, r_1, r_2, \cdots, r_l\}$ (we include $r_0 \in \mathcal{R}$ for convenience). All the l non-initial requests are forwarded to their parent nodes at level 1 at the end of period $\Phi(0)$, following their spiral paths. When level 1 cluster leaders in the respective spiral paths of the requesting nodes receive one request each, they simply forward it to the parent node at level 2 at the end of period $\Phi(1)$; if two requests are received at level 1, one will be forwarded to the parent node at level 2 following the spiral path of the forwarded request, while the other request will be "deflected" down to level 0 along the directory path formed by the previous request that was forwarded to level 2. For more than 2 requests, the above scenario occurs repetitively.

Similar to Section 5.5.3, lets denote by $C^*(\mathcal{R})$ the total communication cost of the optimal algorithm to serve all the requests in $\mathcal{R}$, and by $C(\mathcal{R})$ the total communication cost of the Spiral protocol to serve those requests, in concurrent executions. We will bound the stretch $stretch_{\mathsf{Spiral},concurrent} = \max_{\mathcal{R}} C(\mathcal{R})/C^*(\mathcal{R})$. For simplicity, we consider only the cost incurred by the up phase of each request; if we consider also the cost incurred by the down phase, the stretch increases by a factor of 2 only.

Moreover, similar to Section 5.5.3, lets say $Q_k, 0 \leq k \leq h$, where $h = \lceil \log D \rceil + 1$, are the total number of requests in $\mathcal{R}$ (including the *publish* operation r_0) that reach level k, following their spiral paths, while searching for the directory path towards the shared object.

Lemma 5.5.10 *In concurrent execution $\mathcal{R}$, for the Q_k requests that reach level k in the hierarchy $\mathcal{Z}$, $C^*(\mathcal{R}) \geq \max_{1 \leq k \leq h} |Q_k - 1| \cdot 2^{k-1}$, where $h = \lceil \log D \rceil + 1$.*

Proof. We can observe that the optimal ordering of the concurrent requests is related to the *Steiner tree* problem [110], i.e., the Steiner tree of the source nodes of Q_k whose requests reach level k is a lower bound for $C_k^*(\mathcal{R})$. As any pair in Q_k has source nodes u and v with distance at least $\mathrm{dist}(u, v) \geq 2^{k-1}$ (Lemma 5.3.2), the cost of the Steiner tree to cover all Q_k requests is at least $|Q_k - 1| \cdot 2^{k-1}$. Therefore, $C_k^*(\mathcal{R}) \geq |Q_k - 1| \cdot 2^{k-1}$. Considering all the levels from 1 to h, it is safe to say that the optimal communication cost of the optimal algorithm is at least bounded by the maximum cost of the Steiner tree over $C_k^*(\mathcal{R}), 1 \leq k \leq h$. That is, $C^*(\mathcal{R}) \geq \max_{1 \leq k \leq h} C_k^*(\mathcal{R}) \geq \max_{1 \leq k \leq h} |Q_k - 1| \cdot 2^{k-1}$, where $h = \lceil \log D \rceil + 1$, as needed. □

As we know from Lemma 5.3.3 that total communication cost for each request that reaches level k in $\mathcal{Z}$ is bounded by $c_3 2^{k+2} \log^2 n$ (i.e., the canonical path length up to level k), we have that, for Q_k requests that reach level k, $C_k(\mathcal{R}) \leq |Q_k - 1| \cdot c_3 2^{k+2} \log^2 n$, the cost of Spiral for that level, as there are $|Q_k - 1|$ non-initial requests. For all the levels from 1 to h, from the arguments similar as of Lemma 5.5.8, we can immediately have the following lemma for $C(\mathcal{R})$:

Lemma 5.5.11 *In concurrent execution $\mathcal{R}$, for the Q_k requests that reach level k in the hierarchy $\mathcal{Z}$, $C(\mathcal{R}) \leq \sum_{k=1}^{h} c_3 \cdot |Q_k - 1| \cdot 2^{k+2} \log^2 n$, where $h = \lceil \log D \rceil + 1$.*

Now we give the central result of the concurrent analysis. It follows immediately from a proof similar to Theorem 5.5.9.

Theorem 5.5.12 (*move* stretch in concurrent executions) *The move stretch of the* Spiral *protocol is* $stretch_{\mathsf{Spiral},concurrent} = \mathcal{O}(\log^2 n \cdot \log D)$ *for concurrent executions.*

5.6 Experiments

Motivated from the nice theoretical performance guarantees of Spiral in serving *move* and *lookup* operations, we now aim to investigate how these properties translate in real world through a thorough experimental evaluation. For the experimental evaluation, we adapt the Erdős-Rényi model [47] and generate random graphs of different sizes, ranging from 10 nodes to 2,000 nodes. Particularly, we use the $G(n, \rho)$ variant of the Erdős-Rényi model [47] where a graph G is constructed connecting nodes randomly such the each edge is included in G with probability $0 < \rho < 1$ independent from every other edge. The graphs we use in the experiments are generated setting $p = 0.5$. The weight of each edge is also chosen independently from the weight of every other edge at random from 1 to 10. The results are presented and analyzed for *move* and *lookup* operations on a single shared object. We defer the multiple objects experimentation for future work; however, if we assume that a node can issue a request for another object only after its current request for an object finishes, the results for a single object extend also to multiple objects. The object operations (*move* and *lookup*) are generated uniformly at random in the sense that a bottom-level node which issues a request is selected randomly among the available nodes of the graph every time a request is issued. We implement Spiral in sequential, one-shot, and dynamic executions of *move* and *lookup* operations ranging from 10 to 10,000 operations. Since Arrow uses a pre-selected spanning tree and Spiral uses a hierarchical directory $\mathcal{Z}$ constructed in Section 5.3, we also implement Arrow [43] for the performance comparison of Spiral. We first initialize $\mathcal{Z}$ for the object by creating a downward path from the root to a bottom-level node through a *publish* operation.

The performance of the protocols Spiral and Arrow is measured with respect to the communication cost. We measure the total communication cost through the sum of the weights of the

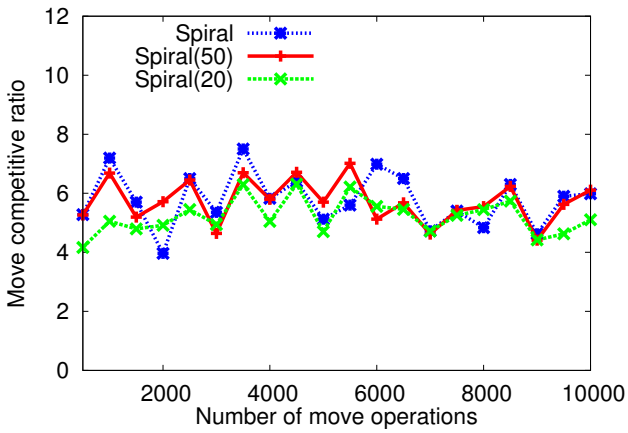


Figure 5.5: Performance of Spiral for sequential and dynamic *move* operations in a random network of 128 nodes. Lower is better.

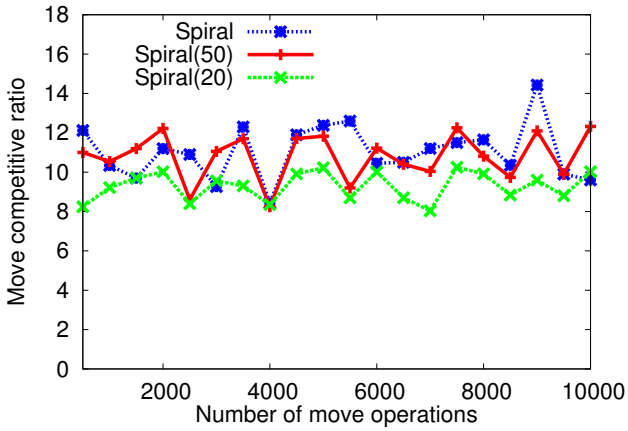


Figure 5.6: Performance of Spiral for sequential and dynamic *move* operations in a random network of 512 nodes. Lower is better.

edges the requests traverse following the protocols to reach their predecessor nodes. The optimal communication cost is measured with respect to the sum of the weights of the edges the requests must traverse if they would have asked to follow shortest paths in G to reach predecessor nodes. We then compare the total communication cost with the optimal communication cost and present the results in terms of competitive ratio. The results presented in this section are the average of 10 experiments. For the experiments, we assume that the execution proceeds in steps such that every node can receive, process, and send a message in each step. In the figures, we denote by Spiral a sequential execution, and by Spiral(t) a dynamic execution in which a new *move* request is gener-

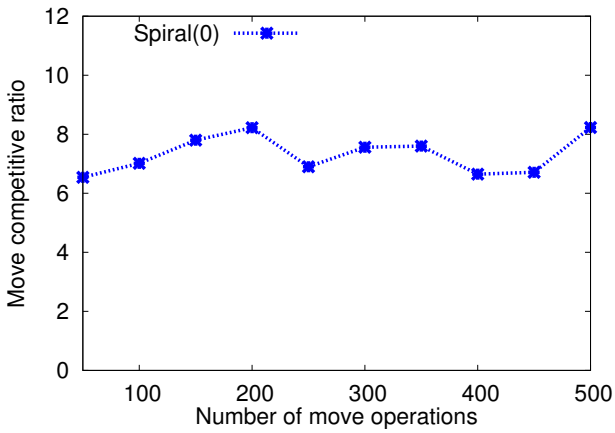


Figure 5.7: Performance of Spiral for one-shot concurrent *move* operations in a random network of 512 nodes. Lower is better.

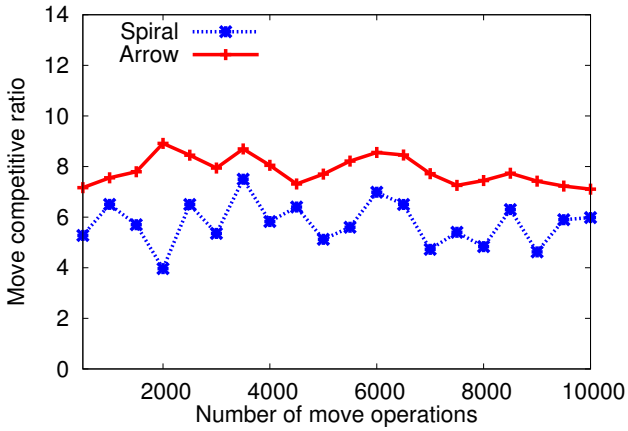


Figure 5.8: Performance comparison of Spiral and Arrow for sequential *move* operations in a random network of 128 nodes. Lower is better.

ated in every t steps from a randomly chosen node in the graph. Therefore, Spiral(0) denotes an one-shot execution.

We start with the performance of Spiral for a set *move* operations in sequential and dynamic executions. Figs. 5.5 and 5.6 show the performance of Spiral in executing 10 to 10,000 *move* operations in random networks of 128 nodes and 512 nodes, respectively. The results show that Spiral performs slightly better in terms of communication cost when there are large number of active *move* requests at each step of the execution. Fig. 5.7 shows the performance of Spiral in executing 10 to 500 *move* operations in a random network of 512 nodes when all of them are issued at the

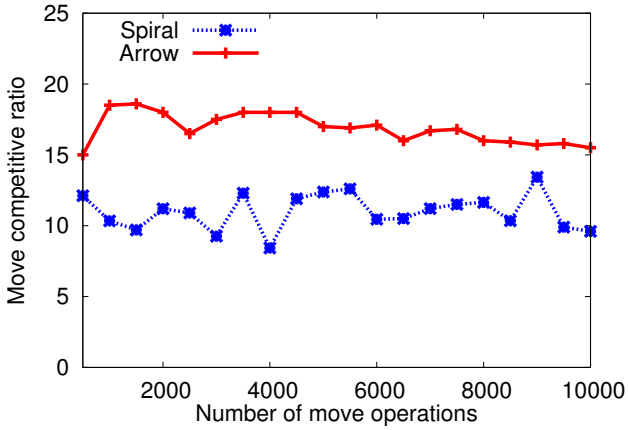


Figure 5.9: Performance comparison of Spiral and Arrow for sequential *move* operations in a random network of 512 nodes. Lower is better.

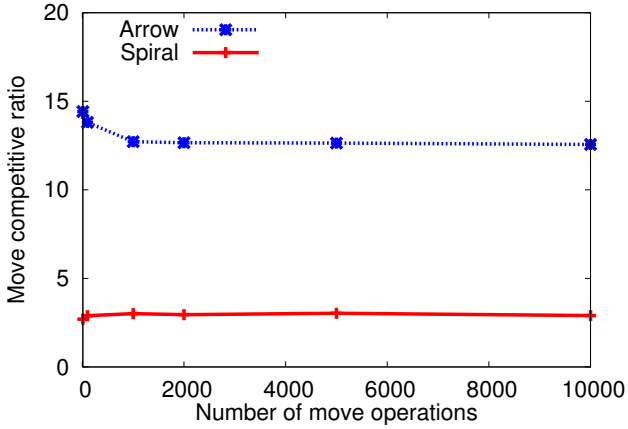


Figure 5.10: Performance comparison of Spiral and Arrow in the worst-case scenario of the sequential execution of *move* operations in a ring network of 128 nodes. Lower is better.

same time step and no further requests are issued thereafter. The reason behind better performance in this one-shot execution, comparing to sequential and dynamic scenarios given in Fig. 5.6, is that downward paths become less deformed when requests are issued concurrently so that requests can be served visiting only the lower levels of the hierarchy minimizing the communication cost.

We are now interested to see how the performance of Spiral compares with the performance of Arrow. This comparison is interesting in the sense that Arrow uses a spanning tree that is different from the overlay structure used by Spiral. The performance comparison of Spiral and Arrow for 10 to 10,000 sequential *move* operations in a random network of 128 nodes is given in Fig. 5.8.

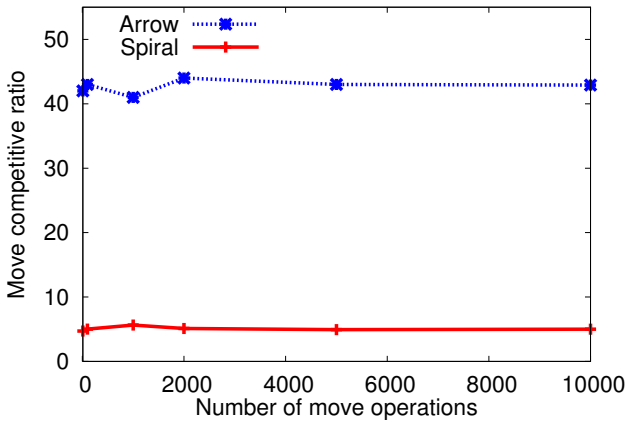


Figure 5.11: Performance comparison of Spiral and Arrow in the worst-case scenario of the sequential execution of *move* operations in a ring network of 512 nodes. Lower is better.

Similarly, Fig. 5.9 shows the results in a random network of 512 nodes for the same set of *move* operations of Fig. 5.8. The results show that Spiral performs 1.12 – 1.65 times better in a 128 nodes network and 1.1 – 1.57 times better in a network of 512 nodes in comparison to Arrow. The comparable performance of Spiral and Arrow is due to the fact that the model that we used for generating random network allows for low cost spanning tree.

Figs. 5.8 and 5.9 showed that the performance of Arrow is comparable to Spiral, despite the use of a spanning tree, in random networks generated using the Erdős-Rényi model [47]. Therefore, we are now interested to see how Spiral and Arrow perform in ring networks. We generated two ring networks of size 128 and 512 nodes, respectively, and ran Spiral and Arrow. We assumed a worst-case scenario in which Arrow needs to serve the sequence of *move* requests that are issued alternatively by the nodes that are the leaves of the spanning tree of the ring network. The comparison results from this study are given in Figs. 5.10 and 5.11. As shown in Fig. 5.10, the competitive ratio of Spiral is approximately 6 times better in comparison with the competitive ratio of Arrow in the ring network of 128 nodes. Note that the edge weights of the ring networks were assigned randomly at uniform from 1 to 10 independently of every other edge. Similarly, in the ring network of 512 nodes, Spiral is approximately 9 times better in comparison to Arrow. This is because every request needs to go through the root node of the spanning tree most of the times

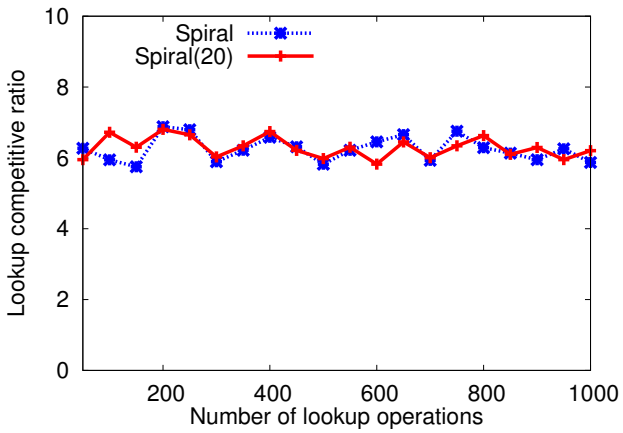


Figure 5.12: Performance of Spiral for sequential and concurrent *lookup* operations in a random network of 128 nodes. Lower is better.

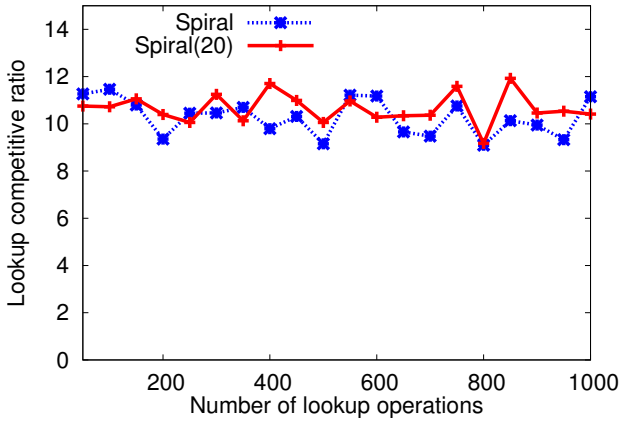


Figure 5.13: Performance of Spiral for sequential and concurrent *lookup* operations in a random network of 512 nodes. Lower is better.

to reach the predecessor node using Arrow, however, in Spiral, the predecessor node is found in a level that is proportional to the shortest distance between the requesting and the predecessor node.

We now study the performance of Spiral in serving *lookup* operations. We compare the results for Spiral when *lookup*s are not overlapped with *move* requests in a sequential execution of *move* requests and when looks are overlapped with *move* requests in a dynamic execution of *move* requests that are issued in every 20 steps. Fig. 5.12 shows the results of this study in executing 50 to 1,000 *lookup* operations in a random network of 128 nodes. We executed a lookup operation after every 10 *move* operations while running *move* operations of Figs. 5.5 and 5.6. The results for a

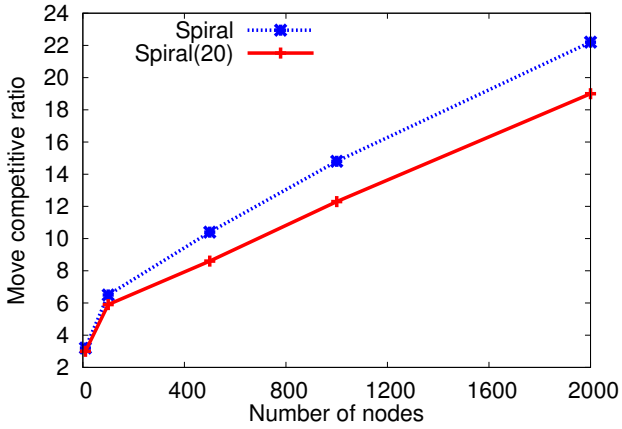


Figure 5.14: Performance of Spiral for 1,000 sequential and dynamic *move* operations in random networks of size ranging from 10 to 2,000 nodes. Lower is better.

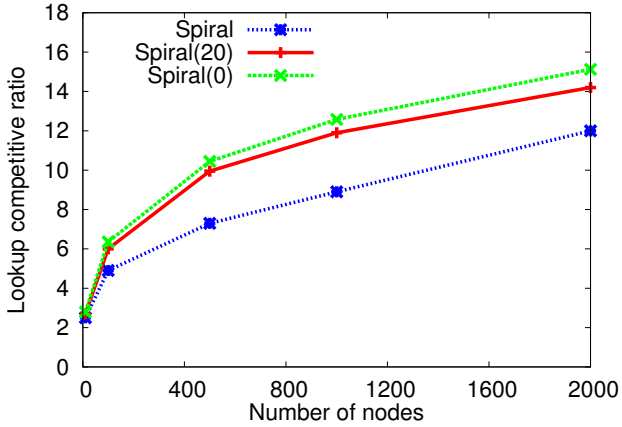


Figure 5.15: Performance of Spiral when a *lookup* operation is issued with non-overlapping and overlapping *move* operations in random networks of size ranging from 10 to 2,000 nodes. Lower is better.

similar setting in a random network of 512 nodes is given in Fig. 5.13. The performance of Spiral is slightly worse in the dynamic execution in comparison to its performance in the sequential execution. The slight increase in cost is due to the forwarding of the *lookup* requests to special-child nodes that lose their $link$ pointers due to concurrent *move* operations while *lookup*s are in transit. However, the impact in performance is very low.

We are also interested to see how the performance of Spiral changes with the network size in both sequential and dynamic executions. We give some results in this scenario in Fig. 5.14

for 1,000 *move* operations in the randomly generated networks of size ranging from 10 to 2,000 nodes. The results show that the difference in performance of Spiral increases with the increase in the network size. The results for a *lookup* operation in the similar setting as of Fig. 5.14 is given in Fig. 5.15. The results for Spiral(0) in Fig. 5.15 are for the setting where a *lookup* operation is issued by a node concurrently with $n/2$ one-shot *move* requests. Fig. 5.15 shows that Spiral achieves the best performance for a *lookup* operation in non-overlapping executions. Moreover, in overlapping scenarios, the increase in competitive ratio depends with the number of active *move* requests at any step of execution. In summary, these results show that Spiral achieves scalable performance in different size networks.

5.7 Summary and Discussions

In this chapter, we considered the problem of implementing transactional memory in large-scale distributed networked systems, where processors are placed in the nodes of a general network and communicate through a message passing environment. We presented and analyzed a novel directory-based DTM protocol, called Spiral, for shared objects, designed for the distributed data-flow implementation of software transactional memory on large-scale distributed networked systems. The protocol supports both shared and exclusive access to items, by providing *lookup* and *move* operations, respectively, and also the *publish* operation to publish the created object. This protocol is based on efficient hierarchical directory construction based on sparse covers with appropriately defined locality parameters. The total ordering imposed among the clusters at same level and also the clusters at different levels tolerates race conditions while serving concurrent requests. To the best of our knowledge, Spiral is the first DTM protocol for distributed transactional memory that achieves poly-logarithmic stretch in general networks.

With this work, we are left with several intriguing directions for future work. First, Spiral can be extended to accommodate non-FIFO communication and tolerate unreliable communication links by adapting some of the techniques used in the Combine protocol [10]. Second, we analyzed Spiral for the batch set of sequential and concurrent requests for a single shared object, a natural

extension is to consider multiple shared objects, dynamic processing of incoming requests, and the respective competitive performance analysis. Third, similar to the previous proposals [10, 43, 77, 143], we assumed full knowledge of the participating nodes, or equivalently, we assumed the static physical network. When nodes enter or leave the physical network, it may be necessary to rebuild the hierarchy. One natural direction is to extend Spiral for dynamic networks, where nodes join and leave over time. Fourth, a self-stabilizing algorithm (similar to [137]) can also be designed for Spiral as fault-tolerance is an important issue in distributed setting because if the node that is currently holding a shared object crashes, that object will become unavailable. At last, our theoretical analysis can be complemented by studying the experimental performance of Spiral on various networks and also comparing the performance experimentally with prior distributed consistency protocols, e.g., [10, 43, 77, 143].

Chapter 6
Distributed Systems: Dynamic Analysis Framework

6.1 Introduction

In this chapter[1], we present a novel analysis framework for distributed hierarchical directories for an arbitrary set of dynamic (online) requests. In order to analyze distributed hierarchical directories, we model the network as a weighted graph, where graph nodes correspond to processors and graph edges correspond to communication links between processors. The hierarchical directories are constructed based on some well-known clustering techniques (e.g., sparse covers, maximal independent sets) where network nodes are organized into $h+1$ levels. In every level, we select a set of leader nodes; higher level leaders coarsen the lower level set of leaders. The leader nodes can be selected arbitrarily at the construction of the hierarchy. At the bottom level (level 0) each node is a leader, while in the top level (level h) there is a single special leader node called the $root$.

We consider an execution of an arbitrary set of dynamic (online) requests, e.g. *publish*, *lookup*, and *move*, which are initiated at arbitrary moments of time by any (bottom level) node. In our analysis, the goal is to minimize the *total communication cost* for the request set. Previously, a dynamic analysis approach is given for spanning tree based implementations Arrow [89] and Relay [145]. Note that this approach can not be directly extended to analyze distributed directories that are based on well-known hierarchical clustering techniques, e.g. Ballistic [77], Spiral [129], STALK [42], and LLS [1]. Recently, Attiya *et al.* [10] provided an analysis of their overlay

[1]This chapter accepted for publication in:
Gokarna Sharma and Costas Busch. An Analysis Framework for Distributed Hierarchical Directories. *Algorithmica*, Preprint, 2013. http://link.springer.com/article/10.1007/s00453-013-9803-2

tree based distributed directory based protocol, Combine, considering online concurrent requests. However, their analysis is also similar to previous analysis approaches for hierarchical clustering based distributed directories for sequential and concurrent requests, and hence can not be applied to analyze them for dynamic requests. To the best of our knowledge, ours is the first formal dynamic performance analysis of hierarchical clustering based distributed directories which are designed to implement a large class of fundamental coordination problems in distributed systems.

6.1.1 Contributions

We present a generic algorithm for implementing a distributed hierarchical directory that can support dynamic requests and prove an upper bound on the competitive ratio of this algorithm by providing a novel analysis framework. Particularly, we prove $\mathcal{O}(\eta \cdot \varphi \cdot \sigma^5 \cdot h)$ competitive ratio for the general algorithm in implementing a large interesting class of distributed hierarchical directories for any arbitrary set of (online) *move* requests in dynamic executions, where η is a write size related parameter, φ is a stretch related parameter, and σ is a growth related parameter on the hierarchy, respectively. A node u in each level k has a *write set* of leaders which helps to implement the *move* requests. The parameter η expresses what is the maximum size of the write set of leaders of the node u among all the levels in the hierarchy; the parameter φ expresses how far the leaders in the write set of u can appear beyond a minimum radius around u; and the parameter σ expresses the minimum radius growth ratio.

The competitive ratio bound given above increases linearly with the maximum number of leaders for all the levels in the distributed hierarchical directory. Someone may say that the competitive ratio can be made sub-linear in η (the write set size parameter) by contacting the leaders in the write set in parallel, hence shortening the route. However, as noticed in [77, 129], this process may introduce race conditions in the concurrent execution of *move* operations in some of the applications of hierarchical directories, e.g. distributed transactional memories. This is due to the fact that the concurrent *move* requests that are contacting their overlapping parent sets may miss one another. The hierarchical directory algorithm still functions correctly as the concurrent *move* requests

eventually meet at the root level in the worst-case. Moreover, we measure the communication cost. Thus, even if an application allows a node to contact the leaders in its write set in parallel, we still need to contact all the leaders in its write set for updating the downward pointers in the hierarchy due to a *move* operation, i.e., the total communication cost is the same in both the sequential and parallel scenarios. In the analysis we fix the duration of time windows (defined later in Section 6.3.1) in such a way that all the leaders in the write set of a node at any level can be contacted one after another in a sequential order before the time windows expire. Therefore, our approach makes the generic algorithm free of possible race conditions even in concurrent executions.

We focused only on *move* requests since they are the most costly operations. The cost due to a *publish* operation is the fixed initial cost which is compensated by the *move* and *lookup* operations issued thereafter. The *lookup* operations have always small cost even when considered individually and do not impact performance of other *move* or *lookup* operations. This is because *lookup* operations do not modify (i.e., add or remove) downward pointers in the hierarchy and the read set of leaders for any node (defined similar to the write set of leaders) used to route *lookup* operations at any level is typically no larger than the write set of leaders at that level for that node used to route *move* operations. Further, we consider only one shared object as in [89, 145] and also a single hierarchical structure per object as in previous directory protocols [77, 129].

We apply our framework and the competitive analysis of the generic algorithm to analyze several variants of distributed hierarchical directory based protocols, Spiral [129], Ballistic [77], Awerbuch and Peleg's tracking a mobile user [13, 15] (hereafter AP-algorithm), and other several tracking algorithms for sensor and mobile ad hoc networks, namely LLS [1], STALK [42], GLS [93], and position based multi-zone routing method [5]. We obtain the following results.

- Spiral: we provide $\mathcal{O}(\log^2 n \cdot \log D)$ competitive ratio for Spiral in general networks, where n is the number of nodes and D is the diameter, respectively, of the network. Spiral is designed for the *data-flow* distributed implementation of software transactional memory in large-scale distributed systems, where transactions are immobile (running at some particular node) and shared objects are moved to those nodes that need them [10, 43, 77, 129, 145].

In a previous work [129], we have shown that Spiral is $\mathcal{O}(\log^2 n \cdot \log D)$ competitive in both sequential executions which consist of non-overlapping sequence of requests and one-shot concurrent executions where all requests appear simultaneously. Here, we provide the analysis for arbitrary dynamic requests which subsumes these previous bounds. It is worthy to note here that this dynamic analysis can also be extended to analyze Spiral for the case where requests execute concurrently. As only one node needs to be locked at a time in the spiral path, the Spiral protocol does not require some kind of mutual exclusion to support concurrent requests and hence helps to avoid race conditions that may occur in concurrent executions without the extra cost that may be incurred due to a mutual exclusion algorithm. The concurrent execution bound for Spiral using this analysis framework will also be the same as the dynamic execution bound.

- AP-algorithm: we provide $\mathcal{O}(\log^2 n \cdot \log D)$ competitive ratio for AP-algorithm in general networks. The AP-algorithm is appropriate for a general mobile user tracking problem that arises in many applications in the distributed setting, e.g. sensor networks. It has been proven in [15] that the algorithm is $\mathcal{O}(\log n \cdot \log D + \log^2 D/\log n)$ competitive in sequential executions and $\mathcal{O}(\log^2 n \cdot \log D + \log^2 D/\log n)$ competitive in one-shot concurrent executions. Our analysis subsumes these results, since it considers the more general case of arbitrary dynamic executions. Moreover, this analysis framework can be used to analysis AP-algorithm for the case where requests execute concurrently.

- Ballistic: we provide $\mathcal{O}(\log D)$ competitive ratio for Ballistic in constant-doubling dimension networks. This protocol is also for the *data-flow* distributed implementation of software transactional memory. It was shown in [77] that Ballistic is $\mathcal{O}(\log D)$ competitive in sequential executions. Again, our analysis subsumes this result. Herlihy and Sun [77] noticed that, during concurrent execution of move operations, Ballistic may need to lock simultaneously multiple parent nodes in the same level and probe them sequentially. This is because, due to the use of overlapping parent sets, *move* requests that concurrently probe them may miss

each other. Our analysis framework can be used to analyze Ballistic in concurrent executions, if the race conditions, as outlined in [77], are avoided. We also get $\mathcal{O}(\log D)$ competitive ratio for Combine [10] in constant-doubling dimension networks, using the hierarchy of [77] as the overlay tree to run Combine.

- **Other protocols**: we provide $\mathcal{O}(\log D)$ competitive ratio for LLS in unit disk graphs, for any arbitrary set of online *move* requests for an object in dynamic executions. This protocol is for providing location service in ad hoc networks. It was shown in [1] that LLS is $\mathcal{O}(\log d)$ competitive for the amortized cost of updating the directory due to a cumulative movement of distance d by a moving node under some assumptions. The competitive bound we give for LLS considers the worst-case. Moreover, we provide the same $\mathcal{O}(\log D)$ competitive ratio for STALK in geometric networks (similar to the model defined in [21]) for an arbitrary set of *move* requests for an object. STALK was shown to be $\mathcal{O}(\log D)$-competitive in [41, 42]. Again, our results subsume these bounds. We also consider position based multi-zone routing method due to Amouris *et al.* [5] and GLS due to Li *et al.* [93]. However, no worst-case competitive bounds are given by the authors for these algorithms and these algorithms exhibit several limitations. The routing method of [5] requires each node in the network to maintain information about every other node (i.e., location updates are flooded). GLS makes little effort to handle updates due to *move* operations and also the out-of-date information, so that even the *move* operations for the object at nearby nodes need to reach to the root of the directory hierarchy to find that nearby node. If these limitations are removed, our framework can also be applied to analyze them in dynamic executions.

The logarithmic factors in the competitive ratio are mainly due to the properties of the hierarchical clustering techniques used in the protocols. Utilizing improved clustering techniques and/or considering specific networks may result in better factors in the competitive ratio. The general network bounds for Spiral and AP-algorithm are within a poly-log factor from optimal, in light of the $\Omega(\log n/\log\log n)$ lower bound proved by Alon *et al.* [4] in certain topologies (e.g., the hypercube, any highly expanding graph, any network with sufficiently large girth, and any highly

expanding graph), for Awerbuch and Peleg's *mobile user tracking* problem [13, 15]. To the best of our knowledge, this is the first competitive dynamic analysis for distributed hierarchical directories.

Someone may consider using the spanning tree T of Gupta [61] (constructed by transforming the randomized tree structure of Fakcharoenphol *et al.* [49]), which guarantees that the expected distance in the tree T for every two nodes in the graph is at most $\mathcal{O}(\log n)$ times their distance in the graph, and run the Arrow protocol on T. An expected bound on the stretch can be proved using the dynamic analysis of Herlihy *et al.* [72] for the Arrow protocol on the spanning tree T. However, the (worst-case) stretch for T can be still as large as D. This is because, for example, in ring networks, the minimum distance is 1 and the maximum distance is $n/2$ between every two nodes in the graph. This results in a competitive ratio of $(D \cdot \log D)$, which is significantly larger than the polylogarithmic competitive ratio of our solution. That is, our solution yields good behavior every time for any arbitrary set of dynamic requests because it bounds the worst-case communication cost, whereas this solution yields good behavior only in the expected case because the spanning tree construction is randomized and hence it bounds only the "expected" communication cost.

Our analysis framework captures both the time and the distance restrictions in ordering dynamic requests through a notion of *time windows*. All the nodes proceed in time windows; in a window, each node might initiate new requests and each node can exchange a message with each of its neighbors in the hierarchy at the end of the window. For obtaining an upper bound, we consider a synchronous execution where time is divided into windows of appropriate duration for each level. For obtaining a lower bound, given an optimal ordering of the requests, we consider the communication cost provided by a Hamiltonian path that visits each request node exactly once according to their order. The lower bound holds for any asynchronous execution of the requests (details in Section 6.3.1). We perform the analysis level by level. The time window notion combined with a Hamiltonian path allows us to analyze the competitive ratio for the requests that reach some level. After combining the competitive ratio of all the levels, we obtain the overall competitive ratio.

6.1.2 Chapter Organization

The rest of the chapter is organized as follows. We give a generic distributed hierarchical directory algorithm in Section 6.2. In Section 6.3, we present a novel dynamic analysis framework based on time windows. We analyze the generic algorithm of Section 6.2 in Section 6.4. Through the framework, we analyze Spiral, Ballistic, AP-algorithm, and other several directory protocols in Section 6.5, and conclude the chapter in Section 6.6 with a short discussion.

6.2 An Online Algorithm

6.2.1 Network Model

We model a distributed network as a weighted undirected graph $G = (V, E, \mathfrak{w})$ similar to the model given in Section 5.2 of Chapter 5.

6.2.2 Hierarchy

Algorithm 9 presents a generic distributed hierarchical directory algorithm, denoted by $\mathcal{A}$. It is based on a hierarchy with $h + 1$ levels of leaders $\mathcal{Z} = \{Z_0, Z_1, \ldots, Z_h\}$ of a network $G = (V, E)$, such that $Z_{k+1} \subseteq Z_k$. In other words, the leaders are partitioned recursively such that, at level 0, each node $v \in V$ is a leader by itself, namely, $Z_0 = V$; and the highest level Z_h contains a single leader root with leader node r. Communication between leader nodes occurs through shortest paths.

Each node $v \in V$ has, at level k, a *write* set of leaders, $Write_k(v) \subseteq Z_k$, and a *read* set of leaders $Read_k(v) \subseteq Z_k$ (Lines 2–6 of Algorithm 9). For convenience, $Write_0(v) = Read_0(v) = v$. The write set of leaders are used to route *move* requests from requesting nodes to their predecessor nodes[2] in the hierarchy, and the read set of leaders are used to route *lookup* requests from requesting nodes to the current owner node of the object ξ (we provide details on how this is done in Algorithm 9 and Section 6.2.3).

[2]As the algorithm forms a distributed queue, the predecessor node of a requesting node is the node that issued the request that is ordered before the request from the requesting node in the distributed queue; see Section 6.3 for details.

We define the following parameters which will be useful later in the analysis.

- $\phi_k(v)$: the maximum radius of the farthest node in $Write_k(v)$ from any node $v \in V$, that is, $\phi_k(v) = \max_{u \in Write_k(v)} \mathrm{dist}(v, u)$.
- ϕ_k: the maximum radius of the farthest node in the write set of any node in the hierarchy at level k, that is, $\phi_k = \max_{v \in V} \phi_k(v)$.
- ϕ'_k: a minimum radius such that if two nodes are within distance ϕ'_k, then they must have a common leader in their write sets at level k. In other words, $\forall u, v \in V, \mathrm{dist}(v, u) \leq \phi'_k \Longrightarrow Write_k(u) \cap Write_k(v) \neq \emptyset$.
- φ: the stretch of maximum versus minimum radius in the write set, that is, $\varphi = \max_{0 \leq k \leq h} \frac{\phi_k}{\phi'_k}$. Typically, $\varphi \geq 1$, since $\phi_k \geq \phi'_k$.
- σ: is the minimum radius growth ratio, such that $\phi'_k = \sigma^{k-1}$, for $k > 0$. Typically, $\sigma \geq 2$.
- η: the maximum write set size for any node v in any level of the hierarchy, namely, $\eta = \max_{0 \leq i \leq h, v \in V} |Write_i(v)|$.

6.2.3 Shared Object Operations

Let ξ be a shared object which we want to access through the distributed directory. At any time there is an owner node, denoted $Owner(\xi)$, which holds the object and is allowed to modify it. The directory hierarchy $\mathcal{Z}$ is a data structure that enables one to find and modify the object whenever needed.

We now describe how the algorithm $\mathcal{A}$ supports *publish*, *lookup*, and *move* in $\mathcal{Z}$. Each leader node t at some level k has a $Pointer_t(\xi)$ pointing towards one of the leaders in level $k-1$ (otherwise it is $\perp$ (null)). A downward chain of pointers will lead to the owner of the object at level 0.

Suppose that some node s issues a $publish(\xi)$ operation. Node s initiates an update of pointer directions from level 1 up to level h such that any downward chain leads to s. At each vertex t in

Algorithm 9: A generic distributed hierarchical directory algorithm for an object ξ

1 **Initialization:**
2 On input graph $G = (V, E)$ build a hierarchy of leaders $\mathcal{Z} = \{Z_0, Z_1, \cdots, Z_h\}$, such that:
3 $Z_{k+1} \subseteq Z_k$, $0 \leq k < h$;
4 Every node $v \in V$ at Z_0 is a leader by itself;
5 Z_h consists of a single leader with leader node r (the root of the hierarchy);
6 Each node $v \in V$ has a *write* set of leaders at level k, $Write_k(v) \subseteq Z_k$, and a *read* set $Read_k(v) \subseteq Z_k$ (with $Write_0(v) = Read_0(v) = v$);
7 **Publish object ξ by node s:**
8 **For** *all* layers $1 \leq k \leq h$ and for *all* $t \in Write_k(s)$ **do**:
9 Set downward pointer of t, $Pointer_t(\xi)$, to point towards any leader in $Write_{k-1}(s)$;
10 **Lookup object ξ by node v:**
11 $k \leftarrow 1$;
12 **Until** $Pointer_t(\xi) \neq \perp$ for any $t \in Read_k(v)$ **do**
13 $k \leftarrow k + 1$;
14 Go to $Owner(\xi)$ following the chain of downward pointers, and send a copy of ξ to v;
15 **Move object ξ to node v:**
16 $k \leftarrow 1$;
17 **Until** $Pointer_t(\xi) \neq \perp$ for any $t \in Write_k(v)$ **do**
18 Set $Pointer_t(\xi)$ of all $t \in Write_k(v)$ to point to any leader in $Write_{k-1}(v)$;
19 $k \leftarrow k + 1$;
20 $old \leftarrow Pointer_t(\xi)$ (where $Pointer_t(\xi) \neq \perp$ and $t \in Write_k(v)$);
21 Set $Pointer_t(\xi)$ of all $t \in Write_k(v)$ to point to any leader in $Write_{k-1}(v)$;
22 Go to $Owner(\xi)$ following the chain of downward pointers starting from old, and at the same time set older downward pointers to $\perp$;
23 As soon as $Owner(\xi)$ is reached, move ξ to v (hence, $Owner(\xi) \leftarrow v$);

$Write_k(s)$ the pointer $Pointer_t(\xi)$ is set to point toward any leader in $Write_{k-1}(s)$ (Lines 7–9 of Algorithm 9). Note that after the publish the $Pointer_r(\xi)$ at the root will not be $\perp$ thereafter.

In order to implement a $lookup(\xi)$ operation, the requesting node v successively queries the vertices in its read set, $Read(v)$, until hitting a vertex t at level k that has a non-null pointer $Pointer_t(\xi)$, which leads to the current owner of ξ (Lines 10–14 of Algorithm 9). Therefore, following the chain of downward pointers the owner node can be reached, and a copy of the object can be obtained by v. The execution of a $move(\xi)$ operation, invoked at some requesting node v, consists of: (i) inserting the pointer $Pointer_v(\xi)$ pointing to any leader in $Write_{i-1}(v)$ for all the leaders $t \in Write_i(v)$ at each level $i < k$ (effectively setting $Owner(\xi) \leftarrow v$ through the new chain of downward pointers) until hitting a vertex t at level k that has a pointer $Pointer_t(\xi)$

leading to the current owner s of ξ; and (ii) deleting $Pointer_t(\xi)$ at all the vertices t in the chain of downward pointers towards s (Lines 15–23 of Algorithm 9). As soon as the current owner is reached, ξ is moved to v. The assumption we made here about setting the pointers for all the leaders $t \in Write_i(v)$ at each level $i < k$ pointing to any leader in $Write_{i-1}(v)$ is to accommodate different techniques used in previous directory protocols. Particular implementation may vary, for example, Spiral orders the $\mathcal{O}(\log n)$ parents at any level into sub-levels such that only one downward pointer needs to be set at any leader node. An example of a *move* operation in Algorithm 9 can be found in Fig. 5.1).

In concurrent execution scenarios the operations in the algorithm require coordination to avoid deadlocks or blocking. For example, updates to the pointers of the write set of a node should all occur in an atomic manner. The various instantiations of the generic algorithm that we describe in Section 6.5 take care of this issue by using different distributed coordination techniques.

6.3 Analysis Framework

We now proceed with describing the framework to analyze the generic online Algorithm $\mathcal{A}$ for a set of arbitrary *move* requests. We identify a *move* request r by the tuple $r = (u, t)$, where u is the leaf node in the cluster hierarchy $\mathcal{Z}$ that initiates the *move* request and $t \geq 0$ is the time when the request is initiated in the system. As soon as the request is initiated it starts searching for the predecessor node following the write set of leaders upward in the hierarchy. We denote by $\mathcal{R} = \{r_0 = (v_0, t_0), r_1 = (v_1, t_1), \ldots\}$ the arbitrary finite set of dynamic (online) *move* requests, where the requests $r_i \in \mathcal{R}$ are indexed according to their initiation time, i.e., $i < j \Longrightarrow t_i \leq t_j$.

We are interested in bounding the competitive ratio of the online algorithm $\mathcal{A}$ in ordering the requests in $\mathcal{R}$ compared with the ordering provided by an optimal algorithm that uses the shortest paths in the original network. Since passing the object from one owner to the next can take some time, the effect of Algorithm $\mathcal{A}$ on the distinct *move* requests is similar to a *distributed queue* which orders the requests in a distributed fashion. When a request tries to join the queue, the online algorithm delivers a message to that request's predecessor node in the queue. The ordering

is considered complete as soon as the request reached the predecessor node. Suppose that a request $r_1 = (v_1, t_1)$ is ordered by Algorithm 9 after another request $r_2 = (v_2, t_2)$. The ordering of r_1 from a node v_1 is considered complete as soon as v_2 is informed that r_1 is the successor of r_2. As the ordering of r_1 after r_2 is provided by the algorithm, it should not always be necessary for r_1 to initiate after r_2 to be ordered behind r_2 in the queue. For example, let us assume that $t_1 < t_2$ for the above requests and there are no other requests in the system. Assume also that the location of the possible predecessor node, denoted by v_3, for both the requests is such that $\mathrm{dist}(v_1, v_3) > c \cdot \mathrm{dist}(v_2, v_3)$ for some constant $c > 1$. In this case, even if r_1 is initiated before r_2, it may not always find the predecessor node v_3 before r_2 finds it due to the distance it needs to travel to reach v_3. This phenomenon, called *overtaking*, may happen for requests depending on their initiation times and their distance from the possible predecessor nodes. This overtaking is *bounded* in the sense that it happens only for a finite time and only for a finite number of dynamic requests. Therefore, it guarantees that every dynamic *move* operation eventually completes execution. We use this bounded overtaking phenomenon in defining (dense/sparse) subsequences in Section 6.4.

Moreover, note that the predecessor node may not necessarily be the owner of the object at the time the request completed its queuing process. As soon as the predecessor node gets the object from its predecessor in the queue (and finishes the operations on it, if any), it will send the object to the requesting node (the successor node of that predecessor node in the queue), invalidating its own copy if needed. Therefore, each requesting node will eventually receive the object according to the distributed global order provided by the online algorithm.

To bound the total competitive ratio of Algorithm 9 due to a set of arbitrary *move* requests, we proceed with defining time windows and give some basic results in Section 6.3.1 which will be useful later in the analysis of the algorithm in Section 6.4. The analysis relies on two separate competitive bounds derived based on the dense windows (Section 6.4.1) and the sparse windows (Section 6.4.2) for each level; the dense and sparse windows at each level are defined according to the number of requests that are inside a particular time window at that level. By combining the bounds of dense and sparse windows for all the levels, we get the desired competitive bound.

6.3.1 Windows

Time windows is an essential ingredient of our analysis framework. We divide time into fixed duration periods which allows us to obtain upper bounds for the communication cost and also respective lower bounds. Our lower bound is valid also for asynchronous execution of requests. In a synchronous execution, we assume that the communication link latency (or delay) is predictable in the sense that the latency is exactly one to send a message by a node to a destination node that is a unit distance far from it. In an asynchronous execution, communication link latencies are not predictable in the sense that latencies may not be exactly one for a destination node at a unit distance, i.e., messages can be arbitrarily fast or slow. We can have a notion of time duration in asynchronous executions by assuming that each message has a delay of at most one time unit. It is a commonly used approach, e.g. [72], based on the intuition that messages eventually arrive to their destinations after a finite amount of time.

We now discuss how our lower bound (in a synchronous execution model) is valid in asynchronous executions. Note that our lower bound is the optimal communication cost which is computed assuming an optimal queue order of requests provided by an optimal algorithm that has complete knowledge about all the requests $\mathcal{R}$. In the optimal ordering, the optimal communication cost is the actual distance in the original network G between the consecutive requesting nodes in the optimal queue order. Therefore, assuming that an optimal algorithm also has to cope up with the worst-case communication link latencies in asynchronous executions [72], we can say that the lower bound is independent of synchrony assumptions and valid for both synchronous and asynchronous execution of requests.

At each level k a window represents the time that a node needs to reach and modify the pointers of all the leader nodes in its write set (this is $|\eta \cdot \phi_k|$). In other words, this is the duration of a time window in our analysis given in Section 6.4. This duration is sufficient for any level k node $u \in V$ to contact all the nodes in its write set $Write_k(u)$ at that level one after another in a serial order. Recall that we do not contact the leaders in the write set of a node in parallel because, as noticed in [77, 129], due to the use of overlapping parent sets, *move* requests that concurrently probe those

leaders may miss each other in some applications of distributed directories. Moreover, we measure the communication cost. Therefore, even if an application allows a node to contact the leaders in its write set in parallel, the total communication cost is the same in both the sequential and parallel scenarios.

We define for level k the *time window* W_k of time duration $|\eta \cdot \phi_k|, 1 \leq k \leq h$, and 1 for $k = 0$. Assuming an execution starts at time 0, we can have the sequence of windows for each level k, $0 \leq k \leq h$, i.e., $\mathbf{W}_k = \{W_k^0, W_k^1, \ldots\}$, where W_k^0 is the first window at level k, W_k^1 is the second window at level k, and so on. These windows have the property that W_k^{j+1} starts immediately after W_k^j expires. When the notations are clear, we simply denote by W_k one of the windows in $\mathbf{W}_k$.

Hereafter, assume for simplicity the worst value for ϕ_k, namely $\varphi \cdot \sigma^{k-1} = \phi_k$ (this doesn't affect the results of the analysis). We also consider $\sigma \geq 2$, which is the case for the algorithms in Section 6.5. The windows are aligned in such a way that W_k and one of W_{k-1} start at the same time. For one window at level h, there are σ windows at level $h-1$, σ^2 windows at level $h-2$, and so on, so that there are σ^{h-k} windows at level k. When we consider the windows of all the levels, there are h overlapping aligned windows for one window at the root level. Fig. 6.1 depicts the window alignment for $\sigma = 2$. As shown in the figure, the windows at every level $k+1$ are twice longer than the windows at level k; this is a consequence of having a minimum radius growth ratio σ that doubles between every consecutive levels. In general cases of hierarchical directories, the time windows at level $k+1$ are σ times longer than the time windows at level k due to the minimum radius growth ratio σ.

We assume that the execution starts at time 0; at that time the root node (the first predecessor node) which has the object is known. The time windows defined above impose restrictions to the algorithm in the sense that they control when to forward requests to higher and lower levels from the current level. Therefore, time windows may add some additional delay in the upper bound cost, i.e., the upper bound gets worse. However, time windows do not affect the lower bound cost because the lower bound analysis can be done without assuming synchronous execution and hence it is valid for any asynchronous execution of requests.

We assume that the message exchange (forwarding of requests to parent and child levels from the current level) happens at the end of the window. (The assumption is for the analysis; the algorithm executes correctly without this assumption.) We would like to note here that as windows and their durations are fixed already for all the levels (Fig. 6.1), some kind of particular message exchange is not required to start new windows at higher and lower levels from any current level. Recall also that the forwarding of requests to higher and lower levels is done at the end of the window to make sure that they can reach and modify the pointers of all the leader nodes in their write sets at the current level. The requests that are initiated to the system at level 0 are forwarded to level 1 at the end of the window W_0 for level 0. Level 1 leaders forward requests to level 2 at the end of the window W_1. This proceeds at higher levels and in a similar way to the downward direction. Therefore, at the end of a window, each level k leader node can exchange a message with its leader neighbors at level $k+1$ or level $k-1$. A leader node y_k at level k forwards the request to a leader node y_{k+1} at level $k+1$ at the end of its window W_k (see paths of r_i and r_j in Fig. 6.1) in the up phase of the request. Similarly, a leader node y_k may forward the request to a leader node y_{k-1} at level $k-1$ at the end of its window W_k in the down phase of the request. There may be the case that the current window W_{k+1} at level $k+1$ is not yet expired when the window W_k is ready to send the requests at its end. In this situation, we impose one more restriction on message exchange such that the messages will be delayed until the current window at $k+1$ (or $k-1$) expires. Hence, the requests that need to be sent to level $k+1$ (to level $k-1$) from level k are sent as soon as a new W_{k+1} window (a new W_{k-1} window) starts (see path of r_j in Fig. 6.1). Depending on the minimum radius growth ratio σ, the requests from at most σ consecutive windows at level $k-1$ are sent to a level k window at a time in the worst-case.

We proceed by proving some basic results on windows. Assume that when in the same window requests update the same pointers then higher priority is given to older requests. In many occasions we will use request r_i to refer to the respective node v_i.

We now prove the first basic result which bounds the initiation time difference of any two requests that reach level k inside some windows. We say that a request from some node *reaches*

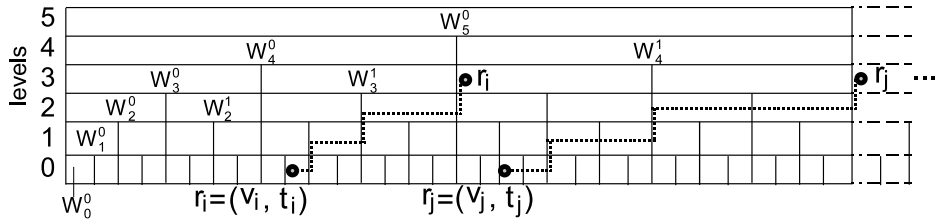


Figure 6.1: Illustration of time windows for $\sigma = 2$

level k if the request probes the leaders of the node in its write set at level k while searching for the predecessor node. We define by *respective window* for a request for level k the window in which the request reaches that level. Lemma 6.3.1 below shows that the initiation time difference of any two requests at any level k is determined by the number of windows between the respective windows of the requests and the window size at that level. Particularly,

Lemma 6.3.1 *Let $r_i = (v_i, t_i) \in \mathcal{R}$ and $r_j = (v_j, t_j) \in \mathcal{R}$ be two requests that reach level k inside respective windows W_k^p and W_k^q, and $q - p - 1 = m$ for some integer $m > 1$. Then the difference in their initiation time is at least $t_j - t_i \geq (m-1) \cdot \eta \cdot \phi_k$.*

Proof. Recall that any request r_i, as soon as it is initiated, starts searching for the predecessor node following its parent nodes in its higher levels in the hierarchy $\mathcal{Z}$. To reach level k, r_i needs to traverse the hierarchy $\mathcal{Z}$ at least the total time of duration $\eta \cdot \phi_{k-1}$ up to level $k-1$. The request r_i then reaches level k at the starting of the window. Due to the properties of time windows and the restriction we impose in when to forward the requests to the higher levels, the total time will be at most $\eta \cdot \phi_k$ in the worst-case. This is due to the case when the current window W_l in the path of r_i at level $l < k$ has not been expired yet at the time when r_i is ready to jump to the next level $l+1$, so that r_i must wait for the next window at level l to begin for it to be able to jump to that level.

Therefore, if a request r_i reaches level k at time t, r_i must be initiated in time between $t_i \in [t - \eta \cdot \phi_k, t - \eta \cdot \phi_{k-1}]$, where t is the current time; otherwise, it must have reached level k in the different window. Therefore, we look at the worst path distance for one request and the best path distance for another request, that is, r_i reached level k traversing only ϕ_{k-1} distance and r_j reached level k traversing ϕ_k distance. When there are m windows between the windows W_k^p and W_k^q in

which r_i and r_j reached level k, for some integer constant $m > 1$, the initiation time difference is $t_j - t_i \geq (t - \eta \cdot \phi_{k-1}) - (t - m \cdot \eta \cdot \phi_k) \geq m \cdot \eta \cdot \phi_k - \eta \cdot \phi_{k-1} \geq m \cdot \eta \cdot \phi_k - \frac{\eta \cdot \phi_k}{\sigma} \geq (m-1) \cdot \eta \cdot \phi_k$. The inequality $t_j - t_i \geq (t - \eta \cdot \phi_{k-1}) - (t - m \cdot \eta \cdot \phi_k)$ is true because of the number of windows between the windows in which requests r_i and r_j reach level k. The lemma follows. □

We now prove the second basic result (Lemma 6.3.2) which bounds the minimum shortest path distance between any two nodes in the original network when requests originating from those nodes reach level $k-1$ inside the same window W_{k-1}^i and level k inside the same window W_k^j. We define a notion of *meet* for two requests r_1 and r_2, respectively, from nodes v_1 and v_2, which will be useful in the proof of Lemma 6.3.2 below. Loosely speaking, we can say that two requests meet at some level if the downward paths (set by them through downward pointers) intersect at that level at some leader node. According to our definition, two requests do not need to be at the same leader node x at some level k at the same time to intersect their downward paths. Formally, if we examine *move* operations in Algorithm 9, when a downward pointer is added at a leader node x in the write set of any node at any level k, it points to any leader in the write set of that node at level $k-1$. Therefore, if the request r_2 from node v_2 discovers a non-null downward pointer at leader node x at level k and the request r_1 from node v_1 is the *move* (or *publish*) request that was last to visit x and hence added the non-null downward pointer seen by r_2, we say that r_1 is met by r_2 at that level k at the leader x. After r_2 sees the non-null downward pointer set by r_1, it starts its down phase following the non-null downward pointer at x.

Lemma 6.3.2 *Suppose $r_i = (v_i, t_i) \in \mathcal{R}$ and $r_j = (v_j, t_j) \in \mathcal{R}, j > i$, are two requests that reach level k. If they both fall inside the same window W_{k-1}^i at level $k-1$ and also inside the same window W_k^j at level k, then* $\mathrm{dist}(v_i, v_j) \geq \sigma^{k-2}$.

Proof. Recall that hierarchical directories use write sets and downward pointers associated with them to coordinate the *move* requests that are trying to join the distributed queue. Moreover, we use two other constraints that are imposed in the way the hierarchy can be built. The first is the minimum distance within which two nodes must have a common leader as a given level grows

exponentially with the levels. The second one is the leaders at level k are a subset of leaders at level $k-1$. We now proceed as follows. As both requests fall inside the same window at level $k-1$ and then continue to the same window at level k, they must not have met each other at level $k-1$. In other words, one request did not see the downward pointers previously set by another request, i.e., the write sets at level $k-1$ of the nodes of these two requests do not overlap, otherwise one request would have been diverted behind the another one following downward pointers. Moreover, according to our construction, two requests can not follow each other after they see the downward pointers previously set. Therefore, these two requests must have been initiated from the nodes that are at distance at least σ^{k-2} from each other so that they do not have a common leader at level $k-1$, i.e., $\mathrm{dist}(v_i, v_j) \geq \sigma^{k-2}$. □

We now prove the third basic result (Lemma 6.3.3). This lemma shows that if two requests reach level k but the shortest distance between the nodes that initiated those requests in the graph is less than σ^{k-1}, then there must exist a third request, that was initiated from some other node, that changed the downward paths so that those requests could not meet at the level that is lower than k. Particularly, we prove the following result.

Lemma 6.3.3 *Suppose $r_i = (v_i, t_i) \in \mathcal{R}$ and $r_j = (v_j, t_j) \in \mathcal{R}, j > i$, are two requests that reach level k. If $\mathrm{dist}(v_i, v_j) < \sigma^{k-1}$ then there must exist a third request $r_l = (v_l, t_l) \in \mathcal{R}$ whose initiation time happens between the time r_i is initiated and the time r_j is initiated, such that either $\mathrm{dist}(v_i, v_l) \geq \sigma^{k-4}$ or $\mathrm{dist}(v_l, v_j) \geq \sigma^{k-4}$.*

Proof. If $\mathrm{dist}(v_i, v_j) < \sigma^{k-1}$ we have that the write sets of the respective nodes intersect at level $k-1$. Let A be the intersection of the write sets. If r_i and r_j both reach level k, then r_j must have missed the downward pointers toward r_i in all the levels up to $k-1$. Thus, there must exist a third request $r_l = (v_l, t_l)$, which was initiated between the time r_i was initiated and the time r_j was initiated, which has deleted the downward pointers in A set by r_i. Suppose now that all requests with initiation time between the time r_i was initiated and the time r_j was initiated are at distance less than σ^{k-4} from r_i. Therefore, all these intermediate requests are within distance less than

$2\sigma^{k-4} \leq \sigma^{k-3}$ from each other. Therefore, the write sets of these nodes at level $k-2$ all intersect with each other. Which implies that no request will reach level $k-1$. Therefore, r_l would not exist, a contradiction. Similarly, it cannot be that all the intermediate requests are within distance less than σ^{k-4} from r_j. Therefore, the claim follows. □

6.4 Analysis of the Online Algorithm

We proceed with necessary definitions that we use in the performance analysis of the online algorithm $\mathcal{A}$. We denote by S_k^j the total count of the number of requests that reach level k inside some window W_k^j. We call the level k windows that have $S_k^j \geq \sigma + 1$ the *dense windows* and the rest of the level k windows (which have $S_k^j < \sigma + 1$) the *sparse windows*, where σ is the minimum radius growth ratio. In some well-known hierarchical directories, e.g. Ballistic [77], AP-algorithm [15], and Spiral [129], where $\sigma = 2$, we have that $S_k^j \geq 3$ for dense windows and $S_k^j < 3$ for sparse windows. The reason behind considering the windows with $S_k^j \geq \sigma + 1$ and $S_k^j < \sigma + 1$ separately is that we need always at least $\lceil S_k^j/\sigma \rceil \geq 2$ requests inside any window that are at least σ^{k-2} far from each other in the graph G (as implied by Lemma 6.3.2). This will help to establish a non-trivial lower bound in the communication cost for ordering all the requests in $\mathcal{R}$ that reach level k. For $S_k^j < \sigma + 1$ windows (i.e. sparse windows), the goal is to transform them into the case of dense windows and apply a similar analysis. In Section 6.4.2, we describe how to transform sparse windows into dense windows case such that there are exactly two requests in each window.

We are interested in obtaining bounds for the communication cost measured as the sum of the distances traversed by all messages. We will bound the competitive ratio $CR_{\mathcal{A}} = \max_{\mathcal{R}} C(\mathcal{R})/C^*(\mathcal{R})$, where $C(\mathcal{R})$ and $C^*(\mathcal{R})$ are the total communication cost and the optimal cost, respectively, of serving all the requests in $\mathcal{R}$ using the online algorithm $\mathcal{A}$ and the optimal algorithm. For convenience, we analyze the competitive ratio of $\mathcal{A}$ for the dense windows and the sparse windows separately. Hence, $CR_{\mathcal{A}} \leq CR_A(\mathcal{R}) + CR_B(\mathcal{R})$, where $CR_A(\mathcal{R})$ is the competitive ratio of $\mathcal{A}$ for serving all the requests inside dense windows and $CR_B(\mathcal{R})$ is the competitive ratio of $\mathcal{A}$ for serving all the requests inside sparse windows.

6.4.1 Dense Windows

In this section, we analyze the total communication cost $C_A(\mathcal{R})$ and the optimal cost $C_A^*(\mathcal{R})$ for dense windows, and bound the competitive ratio $CR_A(\mathcal{R}) = C_A(\mathcal{R})/C_A^*(\mathcal{R})$. We will first focus on a single dense window W_k^j (i.e., a window with $S_k^j \geq \sigma+1$). We give bounds for the total and the optimal communication cost for W_k^j which will be useful when we later analyze the performance for all the dense windows in $\mathbf{W}_k$.

We denote by $C_A(W_k^j(\mathcal{R}))$ the total communication cost of serving requests that reach level k inside a dense window W_k^j by the online algorithm $\mathcal{A}$, and by $C_A^*(W_k^j(\mathcal{R}))$ the respective optimal communication cost. Note that, for simplicity, we consider only the cost incurred by the up phase of each *move* request. When we consider the down phase of each request the cost increases by a factor of 2 only.

We prove following two lemmas (using Lemma 6.3.2). The first lemma bounds the total communication cost $C_A(W_k^j(\mathcal{R}))$ for serving requests that reach level k inside a dense window W_k^j by the online algorithm $\mathcal{A}$. The second lemma bounds the optimal cost $C_A^*(W_k^j(\mathcal{R}))$ for the requests that reach level k inside a dense window W_k^j. Denote by $\{r_1, r_2, \cdots, r_l\}$ a sequence of requests inside a level k window W_k^j. We define a notion of *request pair* that is useful in the proof of Lemma 6.4.2 and also in Section 6.4.2. A request pair is defined as a set of two consecutive requests in $\{r_1, r_2, \cdots, r_l\}$ such that the sequence can be seen as a collection of request pairs $\{(r_1, r_2), (r_2, r_3), \cdots, (r_{l-1}, r_l)\}$. According to our definition, if we denote total number of requests inside W_k^j by S_k^j, then there will be exactly $S_k^j - 1$ number of request pairs.

Lemma 6.4.1 $C_A(W_k^j(\mathcal{R})) \leq 2 \cdot S_k^j \cdot \eta \cdot \phi_k$.

Proof. The communication cost due to a node request at any level is bounded by $\eta \cdot \phi_k$. Therefore, for a request to reach level k, the total cost it incurs is at most $2 \cdot \eta \cdot \phi_k$, since $\sigma \geq 2$, and $\phi_k/\phi_{k-1} \leq \sigma$. Since there are S_k^j requests in W_k^j, $C_A(W_k^j(\mathcal{R})) \leq 2 \cdot S_k^j \cdot \eta \cdot \phi_k$. □

Lemma 6.4.2 $C_A^*(W_k^j(\mathcal{R})) \geq \lceil (S_k^j - 1)/\sigma \rceil \cdot \sigma^{k-2}$.

Proof. We use Lemma 6.3.2 and the restriction imposed by the time windows in forwarding the requests from the current level to its immediate higher (and lower) level. In the best situation, the requests that are forwarded to level k are the requests that fall inside a single window at level $k-1$ (that ends just before the level k window starts). However, in the worst case, the requests that are forwarded to level k are the requests that fall inside σ consecutive windows at level $k-1$. This is due to the fact that the current window at level k might not yet expire when the window at level $k-1$ is ready to forward the requests that fall inside it to level k, as pointed out in Section 6.3.1. Moreover, according to the time windows construction where there are σ windows at level $k-1$ for one window at level k, requests from at most σ consecutive windows at level $k-1$ are forwarded to a level k window at any time. Therefore, if there are $S_k^j \geq \sigma+1$ requests inside the window W_k^j at level k, due to the pigeonhole principle, at least $\lceil S_k^j/\sigma \rceil \geq 2$ requests must come from the same window W_{k-1}^i at level $k-1$. Moreover, it is known from Lemma 6.3.2 that, if two requests from any two bottom level nodes u and v reach level k and they both fall inside some window W_{k-1}^i at level $k-1$ and some window W_k^j at level k, then $\mathrm{dist}(u,v) \geq \sigma^{k-2}$. Therefore, the optimal communication cost $C_A^*(W_k^j(\mathcal{R}))$ is bounded by at least the distance between the $\lceil (S_k^j-1)/\sigma \rceil$ request pairs in W_k^j, i.e., $C_A^*(W_k^j(\mathcal{R})) \geq \lceil (S_k^j-1)/\sigma \rceil \cdot \sigma^{k-2}$. □

Among all the dense windows $\mathbf{W}_k$ for level k, we define a subsequence of dense windows $\mathfrak{W}_k^\alpha = \{W_k^\alpha, W_k^{\alpha+\lambda_d}, W_k^{\alpha+2\lambda_d}, \cdots\} \subset \mathbf{W}_k$ such that $\alpha \in \{0,1,2\}$ for $\lambda_d = 3$. Thus, there will be λ_d *dense subsequences* in $\mathbf{W}_k$. The intuition behind including every third dense window in a dense subsequence is to guarantee that all the requests in window $W_k^{\alpha+i\lambda_d}$ are initiated in the system at least $\eta \cdot \phi_k$ time before any request in window $W_k^{\alpha+(i+1)\lambda_d}, i \geq 0$, is initiated in the system (Lemma 6.3.1). This guarantees that all the requests inside window $W_k^{\alpha+i\lambda_d}$ are ordered before any request inside window $W_k^{\alpha+(i+1)\lambda_d}, i \geq 0$, by the online algorithm. Therefore, overtaking can happen between the requests inside a single window of $\mathfrak{W}_k^\alpha$ only, i.e., the overtaking is bounded in both time and the number of requests. We prove the following lemma.

Lemma 6.4.3 *For any two requests $r_a = (v_a, t_a) \in W_k^{\alpha+j\lambda_d}$, $r_b = (v_b, t_b) \in W_k^{\alpha+(j+1)\lambda_d}$, $j \geq 0$, in the dense subsequence $\mathfrak{W}_k^\alpha$, $t_b - t_a \geq \eta \cdot \phi_k$.*

Proof. As $m = 2$ for $\lambda_d = 3$ in the dense subsequence $\mathfrak{W}_k^\alpha$, from Lemma 6.3.1, we have that $t_b - t_a \geq (2-1) \cdot \eta \cdot \phi_k \geq \eta \cdot \phi_k$. □

We proceed with giving an upper bound in the total communication cost $C_A(\mathfrak{W}_k^\alpha(\mathcal{R}))$ for all the requests in the dense subsequence $\mathfrak{W}_k^\alpha$. We fix $\mathfrak{S}_k^\alpha = \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} S_k^i$, the total number of requests inside all the windows of the dense subsequence $\mathfrak{W}_k^\alpha$, where $|\mathfrak{W}_k^\alpha|$ is the total number of windows in $\mathfrak{W}_k^\alpha$. The following result follows from Lemma 6.4.1 by summing up the communication cost due to $\mathfrak{S}_k^\alpha$ requests in the dense subsequence $\mathfrak{W}_k^\alpha$.

Lemma 6.4.4 *For the requests in a dense subsequence $\mathfrak{W}_k^\alpha$, $C_A(\mathfrak{W}_k^\alpha(\mathcal{R})) \leq 2 \cdot \mathfrak{S}_k^\alpha \cdot \eta \cdot \phi_k$.*

We now bound the optimal cost $C_A^*(\mathfrak{W}_k^\alpha(\mathcal{R}))$ for all the requests in the dense subsequence $\mathfrak{W}_k^\alpha$. The main idea here is to show that $C_A^*(\mathfrak{W}_k^\alpha(\mathcal{R}))$ is at least the cost due to a minimum cost Hamiltonian path that visits each vertex of $\mathfrak{W}_k^\alpha(\mathcal{R})$ exactly once. On our way, we use a notion of directed dependency graph. Note that the lower bound is for any asynchronous execution for the involved requests.

We start with necessary definitions. Let $\mathfrak{R} = \{r_1, r_2, \cdots\} \subset \mathcal{R}$ denote a subset of requests in $\mathcal{R}$. The *directed dependency graph* $H(\mathfrak{R}) = (V', E', \mathfrak{w}')$ has requests as vertices V', where $|V'| = |\mathfrak{R}|$, a directed edge from any vertex $r_i \in V'$ to any other vertex $r_j \in V'$ such that $(v_i, v_j) \in E'$ and $(v_j, v_i) \in E'$, and edge weight function $\mathfrak{w}' : E' \to \mathbb{R}^+$. Note that $H(\mathfrak{R})$ is a *directed complete graph* − there are two directed edges between every pair of vertices. The directed edge weights in $H(\mathfrak{R})$ are assigned as given below:

$$\forall i, j, \mathfrak{w}'(v_i, v_j) = \max\left\{\mathrm{dist}(v_i, v_j), t_i - t_j\right\}.$$

Note that $\mathfrak{w}'(v_i, v_j)$ may be different than $\mathfrak{w}'(v_j, v_i)$. We argue that the time in $\mathfrak{w}'(v_i, v_j)$ translates to the communication cost as there is always a request that is searching for the predecessor node as soon as it is initiated in the system until it is ordered behind the predecessor.

Each possible ordering for any algorithm for the requests in $\mathfrak{R}$ is given by a *directed Hamiltonian path*, that visits each vertex exactly once, on the graph $H(\mathfrak{R})$. Out of the possible orderings, the order which minimizes the ordering cost will be the *lowest cost directed Hamiltonian path*. Since the graph $H(\mathfrak{R})$ is a directed complete graph, there is always a Hamiltonian path. An example of a Hamiltonian path is given in Fig. 6.2 for $\mathfrak{W}_k^\alpha$ with $|\mathfrak{W}_k^\alpha| = 4$, where N_s is the starting node and N_t is the ending node of the path.

Observation 2 *The optimal communication cost $C^*(\mathfrak{R})$ for the requests $\mathfrak{R}$ is at least the* lowest cost directed Hamiltonian path *in the graph $H(\mathfrak{R})$.*

We now consider the directed dependency graph $H(\mathfrak{W}_k^\alpha(\mathcal{R}))$ for all the requests in the dense subsequence $\mathfrak{W}_k^\alpha$. We divide vertices in $H(\mathfrak{W}_k^\alpha(\mathcal{R}))$ into $|\mathfrak{W}_k^\alpha|$ groups, denoted as $H_i, 1 \leq i \leq |\mathfrak{W}_k^\alpha|$, such that H_i corresponds to a window $W_k^i \in \mathfrak{W}_k^\alpha$, where $|\mathfrak{W}_k^\alpha|$ is the total number of windows in the dense subsequence $\mathfrak{W}_k^\alpha$. In other words, a group constitutes a dense window in $\mathfrak{W}_k^\alpha$. We order the groups H_i from left to right. If we look at a particular group H_i, there are some directed edges between vertices inside H_i, some directed edges going out to the groups on both sides (left and right of H_i), and some directed edges coming into H_i from the groups on both sides (see Fig. 6.2). We focus on a subgraph $H^{sub}(\mathfrak{W}_k^\alpha(\mathcal{R}))$ of the graph $H(\mathfrak{W}_k^\alpha(\mathcal{R}))$ such that, for any two vertices $u, v \in H_i$, $\mathrm{dist}(u, v) \geq \sigma^{k-2}$. As argued in Lemma 6.4.2, there will be at least $\lceil S_k^i/\sigma \rceil$ vertices in each group H_i after removing such vertices. Denote by P some directed Hamiltonian path on $H^{sub}(\mathfrak{W}_k^\alpha(\mathcal{R}))$ (see Fig. 6.2) and by P^* the lowest cost directed Hamiltonian path among all P.

We can make the following observations for the requests in the graph $H^{sub}(\mathfrak{W}_k^\alpha(\mathcal{R}))$. As $H^{sub}(\mathfrak{W}_k^\alpha(\mathcal{R}))$ only includes vertices of each $H_i \in H(\mathfrak{W}_k^\alpha(\mathcal{R}))$ such that $\mathrm{dist}(u, v) \geq \sigma^{k-2}$ holds for any two vertices $u, v \in H_i$, we have the following observation.

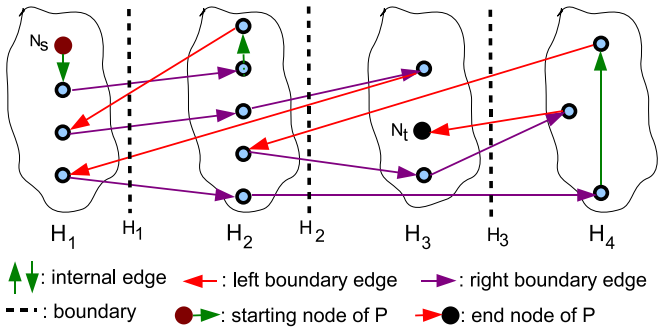


Figure 6.2: Illustration of a Hamiltonian path P starting from the node $N_s \in H_1$ and ending in the node $N_t \in H_3$ for the dense subsequence $\mathfrak{W}_k^\alpha$ with $|\mathfrak{W}_k^\alpha| = 4$. The left boundary edges of a group H_3 are $|\overline{E}_3^{b,left}| = 2$ and the right boundary edges of H_3 are $|\overline{E}_3^{b,right}| = 2$. Moreover, the left external edges of H_3 are $|E_3^{ext,left}| = 1$ and the right external edges of H_3 are $|E_3^{ext,right}| = 1$.

Observation 3 *For any two requests* $r_a = (v_a, t_a) \in H_i$ *and* $r_b = (v_b, t_b) \in H_i$, $\mathfrak{w}'(v_a, v_b) = \mathfrak{w}'(v_b, v_a) \geq \text{dist}(v_a, v_b) \geq \sigma^{k-2}$.

Arguing along the lines of Lemma 6.3.1 for the vertices inside two groups $H_i, H_j, j > i$, we can have the following observation on the edge weights. This is because of the fact that for any request $r_b \in H_j, j > i$, to be ordered behind any request $r_a \in H_i$ in level k, r_j needs time at least $(j - i) \cdot \eta \cdot \phi_k$ which translates to the equivalent weight of the directed edge from H_j to H_i.

Observation 4 *For any two requests* $r_a = (v_a, t_a) \in H_i$ *and* $r_b = (v_b, t_b) \in H_j$, $j > i$, $\mathfrak{w}'(v_a, v_b) \geq 0$ *and* $\mathfrak{w}'(v_b, v_a) \geq (j - i) \cdot \eta \cdot \phi_k$.

In each group H_i, there are two types of edges, internal and external. The *internal* edges E_i^{int} are all the edges (u', v') from any vertex $u' \in H_i$ to any other vertex $v' \in H_i$. The *external* edges E_i^{ext} are all the edges (u', v') from any vertex $u' \in H_i$ to any other vertex $v' \in H_j, j \neq i$. Moreover, the external edges E_i^{ext} of H_i are of two types, that go to the groups on the left ($H_{<i}$), which we denote by $E_i^{ext,left}$ (the *left external* edges), and that go to the groups on the right ($H_{>i}$), which we denote by $E_i^{ext,right}$ (the *right external* edges). We have that $E_i^{ext} = E_i^{ext,left} \cup E_i^{ext,right}$.

We define the *boundary* of H_i as a dotted vertical line on its right (see Fig. 6.2) which shows the interaction between $H_{>i}$ and $H_{\leq i}$. Consider a Hamiltonian path P on $H^{sub}(\mathfrak{W}_k^\alpha(\mathcal{R}))$. We define the *boundary edges* as follows (see Fig. 6.2). For H_i, let $\overline{E}_i^{b,right}$ (the *right boundary* edges)

be the set of edges (u', v') in P which satisfy the condition that $u' \in H_{\leq i}$ and $v' \in H_{>i}$. All the right boundary edges $\overline{E}_i^{b,right}$ will cross the boundary of H_i and point to right groups. Similarly, let $\overline{E}_i^{b,left}$ (the *left boundary* edges) be the set of edges (u', v') in P which satisfy the condition that $u' \in H_{>i}$ and $v' \in H_{\leq i}$. All the left boundary edges $\overline{E}_i^{b,left}$ will cross the boundary of H_i and point to it or the groups on the left of it. We can prove the following relation between $\overline{E}_i^{b,left}$ and $E_i^{ext,left}$ for any group H_i.

Lemma 6.4.5 $|\overline{E}_i^{b,left}| \geq |E_i^{ext,right}| - 1$ *for each group* H_i.

Proof. If path P visits all the vertices in $H_{\leq i}$ before visiting any vertex of $H_{>i}$, we are done. Otherwise, if path P visits only some of the vertices in $H_{\leq i}$ before crossing the boundary of H_i to the right, the path P must visit the rest of the vertices via the left boundary edges $\overline{E}_i^{b,left}$. Thus, there must be at least $|E_i^{ext,right}| - 1$ left boundary edges $\overline{E}_i^{b,left}$. □

The following observation is straightforward. According to the way we defined $\overline{E}_i^{b,left}$ and $E_i^{ext,left}$, if all the left boundary edges $\overline{E}_i^{b,left}$ from each group H_i point to the immediate previous group H_{i-1}, we have that $\sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |\overline{E}_i^{b,left}| = \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{ext,left}|$. In the case when left boundary edges from a group H_i point to any group H_k, these left boundary edges are counted in $\overline{E}_i^{b,left}$ of all the groups that are between group H_k and H_i.

Observation 5 $\sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |\overline{E}_i^{b,left}| \geq \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{ext,left}|$.

In a directed Hamiltonian path P due to the optimal algorithm, some edges (u, v) are between the vertices of a particular group $H_i, 1 \leq i \leq |\mathfrak{W}_k^i|$ (denoted P_{int}), and some are between the vertices of groups H_i and $H_j, j \neq i$ (denoted P_{ext}). Thus, the Hamiltonian path $P = P_{int} \cup P_{ext}$ (union of the edges from both groups). Denote by

$$\begin{aligned} C(P) &= C(P_{int}) + C(P_{ext}) \\ &= C(P_{int}) + C(P_{ext,left}) + C(P_{ext,right}) \end{aligned}$$

the total cost of any Hamiltonian path P, where $P_{ext,left}$ is the set of left external edges $E_i^{ext,left}, 1 \leq i \leq |\mathfrak{W}_k^\alpha|$, in P and $P_{ext,right}$ is the set of right external edges $E_i^{ext,left}, 1 \leq i \leq |\mathfrak{W}_k^\alpha|$, in P. In other words, as depicted in Fig. 6.2, any Hamiltonian path P contains edges that are either internal to a group in the sense that they point from one node to another node inside that group (green edges in Fig. 6.2) or external to a group in the sense that they point to nodes in different groups. In the external case, they either point from a node in any of the right groups to a node in any of the left groups (red edges in Fig. 6.2) or from a node in any of the left groups to a node in any of the right groups (purple edges in Fig. 6.2). We now bound the minimum cost of any Hamiltonian path P.

The following observation is straightforward from the way we defined $H^{sub}(\mathfrak{W}_k^\alpha(\mathcal{R}))$. This is because $H^{sub}(\mathfrak{W}_k^\alpha(\mathcal{R}))$ only includes those vertices of each $H_i \in H(\mathfrak{W}_k^\alpha(\mathcal{R}))$ where $\mathrm{dist}(u,v) \geq \sigma^{k-2}$ holds between every two vertices $u, v \in H_i$.

Observation 6 $C(P_{int}) \geq \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{int}| \cdot \sigma^{k-2}$.

Lemma 6.4.6 $C(P_{ext,left}) \geq \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |\overline{E}_i^{b,left}| \cdot \eta \cdot \phi_k$.

Proof. According to Observation 4, the total cost due to a left boundary edge e that starts from some group H_l and ends at some other group $H_j, j \leq i < l$, is at least $(l-j) \cdot \eta \cdot \phi_k$. As each edge e can be seen as the sum of its $l-j$ number of fragments connecting all the groups starting from H_l to H_j, the lemma follows. □ □

We are now ready to prove the lower bound $C_A^*(\mathfrak{W}_k^\alpha(\mathcal{R}))$ for the requests in the dense subsequence $\mathfrak{W}_k^\alpha$.

Lemma 6.4.7 *For the requests in a dense subsequence* $\mathfrak{W}_k^\alpha, C_A^*(\mathfrak{W}_k^\alpha(\mathcal{R})) \geq \frac{1}{4} \cdot \mathfrak{S}_k^\alpha \cdot \sigma^{k-3}$.

Proof. We have that $C_A^*(\mathfrak{W}_k^\alpha(\mathcal{R})) \geq C(P^*)$, where P^* is on $H^{sub}(\mathfrak{W}_k^\alpha(\mathcal{R}))$. Therefore, we bound the optimal cost $C(P^*)$ of P^*, which is at least

$$C(P^*) \quad \geq \quad C(P_{int}^*) + C(P_{ext,left}^*) + C(P_{ext,right}^*).$$

From Observation 4, $C(P^*_{ext,right}) \geq 0$. Therefore, from Observation 6 and Lemma 6.4.6,

$$C(P^*) \;\geq\; C(P^*_{int}) + C(P^*_{ext,left}) \geq \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{int}| \cdot \sigma^{k-2} + \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |\overline{E}_i^{b,left}| \cdot \eta \cdot \phi_k.$$

As $\sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |\overline{E}_i^{b,left}| \geq \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{ext,left}|$ (Observation 5) and $|\overline{E}_i^{b,left}| \geq |E_i^{ext,right}| - 1$ (Lemma 6.4.5), the above equation reduces to

$$\begin{aligned} C(P^*) \;&\geq\; \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{int}| \cdot \sigma^{k-2} + \frac{1}{2} \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |\overline{E}_i^{b,left}| \cdot \eta \cdot \phi_k + \frac{1}{2} \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |\overline{E}_i^{b,left}| \cdot \eta \cdot \phi_k \\ &\geq\; \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{int}| \cdot \sigma^{k-2} + \frac{1}{2} \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{ext,left}| \cdot \eta \cdot \phi_k + \\ &\quad \frac{1}{2} \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} \Big(|E_i^{ext,right}| - 1 \Big) \eta \cdot \phi_k. \end{aligned}$$

When $|E_i^{ext,right}| < 2$ for each group H_i, $1 \leq i \leq |\mathfrak{W}_k^\alpha|$, then $|\overline{E}_i^{b,left}| \geq 0$ (Lemma 6.4.5). Therefore, for each group H_i, there must be the case that the Hamiltonian path P^* visited all the vertices in $H_{\leq i}$ before visiting any vertex of $H_{>i}$. Thus, the above equation reduces to $C(P^*) \geq \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{int}| \cdot \sigma^{k-2}$. As there are at least $\lceil S_k^i/\sigma \rceil$ vertices in each group H_i, we have that $\sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{int}| \geq \frac{\mathfrak{S}_k^\alpha}{2 \cdot \sigma}$. Hence, $C(P^*) \geq \frac{\mathfrak{S}_k^\alpha}{2 \cdot \sigma} \cdot \sigma^{k-2} \geq \frac{1}{4} \cdot \mathfrak{S}_k^\alpha \cdot \sigma^{k-3}$.

$$\begin{aligned} |E_i^{ext,right}| \geq 2 \;&\implies\; |E_i^{ext,right}| - 1 \geq \frac{|E_i^{ext,right}|}{2} \\ &\implies\; \frac{1}{2} \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} \Big(|E_i^{ext,right}| - 1 \Big) \eta \cdot \phi_k \geq \frac{1}{2} \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} \left(\frac{|E_i^{ext,right}|}{2} \right) \eta \cdot \phi_k \\ &\implies\; \frac{1}{2} \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} \Big(|E_i^{ext,right}| - 1 \Big) \eta \cdot \phi_k \geq \frac{1}{4} \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{ext,right}| \cdot \eta \cdot \phi_k. \end{aligned}$$

Therefore, since $\phi_k = \varphi\sigma^{k-1} \geq \sigma^{k-2}$,

$$\begin{aligned} C(P^*) &\geq \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{int}| \cdot \sigma^{k-2} + \frac{1}{2}\sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{ext,left}| \cdot \eta \cdot \phi_k + \frac{1}{4}\sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{ext,right}| \cdot \eta \cdot \phi_k \\ &\geq \frac{1}{4}(\sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{int}| + \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{ext,left}| + \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{ext,right}|)\sigma^{k-2}. \end{aligned}$$

As there are at least $\lceil S_k^i/\sigma \rceil$ vertices in each group H_i and each vertex is attached to at least one edge with a source vertex, we have that

$$\sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{int}| + \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{ext,left}| + \sum_{i=1}^{|\mathfrak{W}_k^\alpha|} |E_i^{ext,right}| \geq \frac{\mathfrak{S}_k^\alpha}{\sigma}.$$

Therefore,

$$C(P^*) \geq \frac{1}{4} \cdot \frac{\mathfrak{S}_k^\alpha}{\sigma} \cdot \sigma^{k-2} \geq \frac{1}{4} \cdot \mathfrak{S}_k^\alpha \cdot \sigma^{k-3}.$$

The lemma follows as $C_A^*(\mathfrak{W}_k^\alpha(\mathcal{R})) \geq C(P^*)$. □

We now bound the total communication cost $C_A(\mathcal{R})$ of the online algorithm $\mathcal{A}$ and the optimal communication cost $C_A^*(\mathcal{R})$ for serving all the requests in $\mathcal{R}$ that are inside dense windows. For the total communication cost, we first sum the total communication cost of serving all the requests inside all the dense subsequences in a level and later combine the cost for all the levels. For the optimal communication cost, we first find the optimal communication cost of serving all the requests inside all the dense subsequences in a level and later take the maximum among the optimal cost for all the levels.

Lemma 6.4.8 *For the execution $\mathcal{R}$, the total communication cost of the online algorithm $\mathcal{A}$ for all the requests inside the dense windows (of all the levels) is $C_A(\mathcal{R}) \leq 2 \cdot \sum_{k=1}^{h} \sum_{\alpha=0}^{\lambda_d - 1} (\mathfrak{S}_k^\alpha \cdot \eta \cdot \phi_k)$.*

Proof. From Lemma 6.4.4, we have that for one dense subsequence $\mathfrak{W}_k^\alpha$ of dense windows at level k, the total communication cost of the online algorithm $\mathcal{A}$ is $C_A(\mathfrak{W}_k^\alpha(\mathcal{R})) \leq 2 \cdot \mathfrak{S}_k^\alpha \cdot \eta \cdot \phi_k$.

Moreover, there are total λ_d dense subsequences. Therefore, the total communication cost for the dense windows in $\mathbf{W}_k$ is bounded by $C_A(\mathbf{W}_k) \leq \sum_{\alpha=0}^{\lambda_d-1} C_A(\mathfrak{W}_k^\alpha(\mathcal{R})) \leq 2 \cdot \sum_{\alpha=0}^{\lambda_d-1}(\mathfrak{S}_k^\alpha \cdot \eta \cdot \phi_k)$. By combining the costs of the dense windows for each level, the total communication cost of $\mathcal{A}$ is bounded by

$$C_A(\mathcal{R}) \leq \sum_{k=1}^{h} C_A(\mathbf{W}_k) \leq \sum_{k=1}^{h} \left(2 \cdot \sum_{\alpha=0}^{\lambda_d-1} (\mathfrak{S}_k^\alpha \cdot \eta \cdot \phi_k) \right) \leq 2 \cdot \sum_{k=1}^{h} \sum_{\alpha=0}^{\lambda_d-1} (\mathfrak{S}_k^\alpha \cdot \eta \cdot \phi_k),$$

where h is the number of cluster levels. (We do not consider communication costs for level 0 in total cost because there is no communication that reaches that level.) □

Lemma 6.4.9 *For the execution $\mathcal{R}$, the optimal communication cost for all the requests inside dense windows (of all the levels) is $C_A^*(\mathcal{R}) \geq \frac{1}{4} \cdot \max_{1 \leq k \leq h} \cdot \max_\alpha \left(\mathfrak{S}_k^\alpha \cdot \sigma^{k-3} \right)$.*

Proof. As there are total λ_d dense subsequences and $C_A^*(\mathfrak{W}_k^\alpha(\mathcal{R})) \geq \frac{1}{4} \cdot \mathfrak{S}_k^\alpha \cdot \sigma^{k-3}$ (Lemma 6.4.7) for one dense subsequence $\mathfrak{W}_k^\alpha$, the optimal communication cost for the dense windows in $\mathbf{W}_k$ is at least $C_A^*(\mathbf{W}_k) \geq \max_\alpha C_A^*(\mathfrak{W}_k^\alpha(\mathcal{R})) \geq \frac{1}{4} \cdot \max_\alpha(\mathfrak{S}_k^\alpha \cdot \sigma^{k-3})$. Considering all the dense windows from level 1 to level h, the optimal communication cost $C_A^*(\mathcal{R})$ is bounded by at least the maximum over $C_A^*(\mathbf{W}_k), 1 \leq k \leq h$, where h is the number of cluster levels. Therefore, $C_A^*(\mathcal{R}) \geq \max_{1 \leq k \leq h} C_A^*(\mathbf{W}_k) \geq \max_{1 \leq k \leq h} \left(\frac{1}{4} \cdot \max_\alpha(\mathfrak{S}_k^\alpha \cdot \sigma^{k-3}) \right) \geq \frac{1}{4} \cdot \max_{1 \leq k \leq h} \cdot \max_\alpha \left(\mathfrak{S}_k^\alpha \cdot \sigma^{k-3} \right)$. (We do not consider costs for level 0 in optimal communication cost since there is no communication that reaches that level.) □

We are now ready to bound the competitive ratio $CR_A(\mathcal{R})$ for the dense windows.

Theorem 6.4.10 $CR_A(\mathcal{R}) = \mathcal{O}(\eta \cdot \varphi \cdot \sigma^2 \cdot h)$.

Proof. We obtain from Lemmas 6.4.8 and 6.4.9 the competitive ratio of the online algorithm $\mathcal{A}$ for the dense windows bounded by

$$
\begin{aligned}
CR_A(\mathcal{R}) &= \frac{C_A(\mathcal{R})}{C^*_A(\mathcal{R})} \\
&\leq \frac{2 \cdot \sum_{k=1}^{h} \sum_{\alpha=0}^{\lambda_d - 1} \left(\mathfrak{S}_k^\alpha \cdot \eta \cdot \phi_k\right)}{\frac{1}{4} \cdot \max_{1 \leq k \leq h} \cdot \max_\alpha \left(\mathfrak{S}_k^\alpha \cdot \sigma^{k-3}\right)} \\
&\leq \frac{8h\lambda_d \cdot \max_{1 \leq k \leq h} \cdot \max_\alpha \left(\mathfrak{S}_k^\alpha \cdot \eta \cdot \phi_k\right)}{\max_{1 \leq k \leq h} \cdot \max_\alpha \left(\mathfrak{S}_k^\alpha \cdot \sigma^{k-3}\right)} \\
&\leq \frac{8h\lambda_d \cdot \max_{1 \leq k \leq h} \cdot \max_\alpha \left(\mathfrak{S}_k^\alpha \cdot \eta \cdot \varphi \cdot \sigma^2\right)}{\max_{1 \leq k \leq h} \cdot \max_\alpha \mathfrak{S}_k^\alpha} \\
&\leq 8h\lambda_d \eta \varphi \sigma^2 \cdot \frac{\max_{1 \leq k \leq h} \cdot \max_\alpha \mathfrak{S}_k^\alpha}{\max_{1 \leq k \leq h} \cdot \max_\alpha \mathfrak{S}_k^\alpha} \\
&\leq 8 \cdot h \cdot \lambda_d \cdot \eta \cdot \varphi \cdot \sigma^2 \\
&= \mathcal{O}(\eta \cdot \varphi \cdot \sigma^2 \cdot h),
\end{aligned}
$$

since $\lambda_d = 3$ and $\phi_k / \sigma^{k-3} \leq \varphi \cdot \sigma^2$. □

6.4.2 Sparse Windows

In this section, we analyze the total communication cost $C_B(\mathcal{R})$ and the optimal communication cost $C^*_B(\mathcal{R})$ for serving requests inside sparse windows, and bound the competitive ratio $CR_B(\mathcal{R}) = C_B(\mathcal{R})/C^*_B(\mathcal{R})$. Recall that a level k window W_k^j is sparse if $S_k^j \leq \sigma$. We consider a subsequence of sparse windows of $\mathbf{W}_k$ (the set of all windows at level k) for the competitive ratio.

Due to S_k^j requests inside each sparse window, it may not always be the case that these (at most) σ requests satisfy the requirements for the lower bound derivation. Therefore, our goal in the analysis that follows is to transform each sparse window scenario into a dense window case such that there are exactly two requests in each sparse window that are at least σ^{k-4} far in the graph G. Note that in dense windows the distance lower bound was σ^{k-2}; here however it becomes σ^{k-4} because of Lemma 6.3.3.

Similar to the subsequences of dense windows, we consider the subsequence of sparse windows $\mathcal{Q}_k^\beta = \{W_k^\beta, W_k^{\beta+\lambda_s}, W_k^{\beta+2\lambda_s}, \cdots\} \subset \mathbf{W}_k$ such that $\beta \in \{0, 1, 2\}$ for $\lambda_s = 3$. Thus, there will be λ_s *sparse subsequences* in $\mathbf{W}_k$. Similar to Lemma 6.4.3, for any two requests $r_a = (v_a, t_a) \in W_k^{\beta+j\lambda_d}$

and $r_b = (v_b, t_b) \in W_k^{\beta+(j+1)\lambda_d}$, $j \geq 0$, of $\mathcal{Q}_k^\beta$, $t_b - t_a \geq \eta \cdot \phi_k$. This provides bounded overtaking such that all the requests inside window $W_k^{\beta+j\lambda_d}$ of $\mathcal{Q}_k^\beta$ will be ordered before any request inside window $W_k^{\beta+(j+1)\lambda_d}$, $j \geq 0$, by the online algorithm.

Next, we will focus on a sparse subsequence $\mathcal{Q}_k^\beta$. We give bounds on the total and the optimal communication cost for all the requests in the sparse subsequence $\mathcal{Q}_k^\beta$ and these results extend to all sparse windows in $\mathbf{W}_k$.

Denote by $\mathcal{P}_k^\beta = \{r_1, r_2, r_3, \cdots\}$ a sequence of requests in the sparse subsequence $\mathcal{Q}_k^\beta$ such that each window $W_k^i \in \mathcal{Q}_k^\beta$ has one request r_i (chosen arbitrarily among the σ it contains). In other words, $|\mathcal{P}_k^\beta| = |\mathcal{Q}_k^\beta|$. As $S_k^j \leq \sigma$, for each window $W_k^j \in \mathcal{Q}_k^\beta$, the total cost computed via $\mathcal{P}_k^\beta$ for $\mathcal{Q}_k^\beta$ will increase by a factor of σ only.

Using the request pair definition given in Section 6.4.1, $\mathcal{P}_k^\beta$ can be seen as a collection of request pairs $\mathcal{P}_k^\beta = \{(r_1, r_2), (r_2, r_3), (r_3, r_4), \cdots\}$. Each request pair $(r_a, r_{a+1}) \in \mathcal{P}_k^\beta$ has the property that $t_{a+1} - t_a \geq \eta \cdot \phi_k$, however there may be the case that $\mathrm{dist}(r_a, r_{a+1}) < \sigma^{k-1}$. We define another sequence of request pairs $\widetilde{\mathcal{P}}_k^\beta = \{(r'_1, r''_1), (r'_2, r''_2), (r'_3, r''_3), \cdots\}$ for the sequence of requests in $\mathcal{P}_k^\beta$ using a transformation given below.

i. If $\mathrm{dist}(r_a, r_{a+1}) \geq \sigma^{k-1}$ in the graph G for any two subsequent requests $r_a \in \mathcal{P}_k^\beta$ and $r_{a+1} \in \mathcal{P}_k^\beta$, we fix $r'_a = r_a$ and $r''_a = r_{a+1}$.

ii. if $\mathrm{dist}(r_a, r_{a+1}) < \sigma^{k-1}$ in the graph G for any two subsequent requests $r_a \in \mathcal{P}_k^\beta$ and $r_{a+1} \in \mathcal{P}_k^\beta$, then according to Lemma 6.3.3, there exists an ordering request r_c (it can be from the same level k or the lower) after r_a and before r_{a+1} in time such that either $\mathrm{dist}(r_a, r_c) \geq \sigma^{k-4}$ or $\mathrm{dist}(r_c, r_{a+1}) \geq \sigma^{k-4}$. We fix r'_a and r''_a following the criteria given below:

   a. If there is the case that $\mathrm{dist}(r_a, r_c) \geq \sigma^{k-4}$, then we fix $r'_a = r_a$ and $r''_a = r_c$.

   b. If there is the case that $\mathrm{dist}(r_c, r_{a+1}) \geq \sigma^{k-4}$, then we fix $r'_a = r_c$ and $r''_a = r_{a+1}$.

The transformation from $\mathcal{P}_k^\beta$ to $\widetilde{\mathcal{P}}_k^\beta$ guarantees that $\mathrm{dist}(r'_a, r''_a) \geq \sigma^{k-4}$ for any request pair $(r'_a, r''_a) \in \widetilde{\mathcal{P}}_k^\beta$. However, the timing requirement of at least $\eta \cdot \phi_k$ for any two requests r_1 and r_2

in the subsequent request pairs of $\widetilde{\mathcal{P}}_k^\beta$ may be violated. We satisfy the timing requirement through special sparse subsequences on $\widetilde{\mathcal{P}}_k^\beta$.

We define a *special sparse subsequence* $\widehat{\mathcal{P}}_k^\gamma = \{(r_1'', r_1'''), (r_2'', r_2'''), (r_3'', r_3'''), \cdots\} \subset \widetilde{\mathcal{P}}_k^\beta$ for the pair of requests in $\widetilde{\mathcal{P}}_k^\beta$ by including every third request pair of $\widetilde{\mathcal{P}}_k^\beta$, where $\gamma \in \{0, 1, \cdots, \lambda_s' - 1\}$ for $\lambda_s' = 3$. Therefore, there will be λ_s' special sparse subsequences in $\widetilde{\mathcal{P}}_k^\beta$. Moreover, the requests in $\widehat{\mathcal{P}}_k^\gamma$ satisfy the following lemma for the timing requirement (due to Lemmas 6.3.1 and 6.4.3).

Lemma 6.4.11 *For any two consecutive request pairs* (r_a'', r_a''') *and* (r_{a+1}'', r_{a+1}''') *in* $\widehat{\mathcal{P}}_k^\gamma$, $t_{a+1}'' - t_a''' \geq \eta \cdot \phi_k$.

The special sparse subsequence $\widehat{\mathcal{P}}_k^\gamma$ has exactly two requests in each window it contains and the requests in subsequent windows satisfy the timing property. Therefore, each request pair in $\widehat{\mathcal{P}}_k^\gamma$ can be treated as a group H_i in the dense window analysis. From this point on, the analysis proceeds similar to the case of dense windows, where now each pair corresponds to a "dense window" (note that a pair may not actually reside in the same window, but we will assume it does, without affecting correctness, in order to perform the lower bound analysis). Similar to Theorem 6.4.10, we can obtain the following theorem for the competitive ratio $CR_B(\mathcal{R})$ for the sparse windows (the term σ^5 comes from using the σ^{k-4} distance in the pairs).

Denote by $C_B(\mathcal{Q}_k^\beta(\mathcal{R}))$ the total communication cost and by $C_B^*(\mathcal{Q}_k^\beta(\mathcal{R}))$ the optimal communication cost for all the requests in the sparse subsequence $\mathcal{Q}_k^\beta$. The lemmas given below for $\mathcal{Q}_k^\beta$ follow similarly to Lemmas 6.4.4 and 6.4.7. We fix $\mathfrak{S}_k^\gamma = |\widehat{\mathcal{P}}_k^\gamma|$ the total number of request pairs in $\widehat{\mathcal{P}}_k^\gamma$.

Lemma 6.4.12 *For the requests in a sparse subsequence* $\mathcal{Q}_k^\beta$, $C_B(\mathcal{Q}_k^\beta(\mathcal{R})) \leq 2 \cdot \sigma \cdot \sum_{\gamma=0}^{\lambda_s'-1}(\mathfrak{S}_k^\gamma \cdot \eta \cdot \phi_k)$.

Lemma 6.4.13 *For the requests in a sparse subsequence* $\mathcal{Q}_k^\beta$, $C_B^*(\mathcal{Q}_k^\beta(\mathcal{R})) \geq \frac{1}{4} \cdot \max_\gamma(\mathfrak{S}_k^\gamma \cdot \sigma^{k-5})$.

The following lemmas bound $C_B(\mathcal{R})$ and $C_B^*(\mathcal{R})$.

Lemma 6.4.14 *For the execution $\mathcal{R}$, the total communication cost of the online algorithm $\mathcal{A}$ for all the requests inside the sparse windows (of all the levels) is* $C_B(\mathcal{R}) \leq 2 \cdot \sigma \cdot \sum_{k=1}^{h} \sum_{\beta=0}^{\lambda_s-1} \sum_{\gamma=0}^{\lambda'_s-1} (\mathfrak{S}_k^{\gamma} \cdot \eta \cdot \phi_k)$.

Lemma 6.4.15 *For the execution $\mathcal{R}$, the optimal communication cost for all the requests inside sparse windows (of all the levels) is* $C_B^*(\mathcal{R}) \geq \frac{1}{4} \cdot \max_{1 \leq k \leq h} \cdot \max_{\beta} \cdot \max_{\gamma} \left(\mathfrak{S}_k^{\gamma} \cdot \sigma^{k-5}\right)$.

Theorem 6.4.16 $CR_B(\mathcal{R}) = \mathcal{O}(\eta \cdot \varphi \cdot \sigma^5 \cdot h)$.

Proof. We obtain from Lemmas 6.4.14 and 6.4.15 the competitive ratio of the online algorithm $\mathcal{A}$ for the sparse windows bounded by

$$\begin{aligned}
CR_B(\mathcal{R}) &= \frac{C_B(\mathcal{R})}{C_B^*(\mathcal{R})} \\
&\leq \frac{2 \cdot \sigma \cdot \sum_{k=1}^{h} \sum_{\beta=0}^{\lambda_s-1} \sum_{\gamma=0}^{\lambda'_s-1} (\mathfrak{S}_k^{\gamma} \cdot \eta \cdot \phi_k)}{\frac{1}{4} \cdot \max_{1 \leq k \leq h} \cdot \max_{\beta} \cdot \max_{\gamma} (\mathfrak{S}_k^{\gamma} \cdot \sigma^{k-5})} \\
&\leq \frac{8\lambda_s \lambda'_s h\sigma \cdot \max_{1 \leq k \leq h} \cdot \max_{\beta} \cdot \max_{\gamma} (\mathfrak{S}_k^{\gamma} \cdot \eta \cdot \phi_k)}{\max_{1 \leq k \leq h} \cdot \max_{\beta} \cdot \max_{\gamma} (\mathfrak{S}_k^{\gamma} \cdot \sigma^{k-5})} \\
&\leq \frac{8\lambda_s \lambda'_s h\sigma \cdot \max_{1 \leq k \leq h} \cdot \max_{\beta} \cdot \max_{\gamma} (\mathfrak{S}_k^{\gamma} \cdot \eta \cdot \varphi \cdot \sigma^4)}{\max_{1 \leq k \leq h} \cdot \max_{\beta} \cdot \max_{\gamma} \mathfrak{S}_k^{\gamma}} \\
&\leq 8\lambda_s \lambda'_s h\eta\varphi\sigma^5 \cdot \frac{\max_{1 \leq k \leq h} \cdot \max_{\beta} \cdot \max_{\gamma} \mathfrak{S}_k^{\gamma}}{\max_{1 \leq k \leq h} \cdot \max_{\beta} \cdot \max_{\gamma} \mathfrak{S}_k^{\gamma}} \\
&\leq 8 \cdot \lambda_s \cdot \lambda'_s \cdot h \cdot \eta \cdot \varphi \cdot \sigma^5 = \mathcal{O}(\eta \cdot \varphi \cdot \sigma^5 \cdot h),
\end{aligned}$$

since $\lambda_s = \lambda'_s = 3$ and $\phi_k/\sigma^{k-5} \leq \varphi \cdot \sigma^4$. □

6.4.3 Complexity of the Online Algorithm

We now prove the main theorem of the analysis. Since the execution $\mathcal{R}$ is arbitrary, we obtain from Theorem 6.4.10 of the dense window analysis (Section 6.4.1) and Theorem 6.4.16 of the sparse window analysis (Section 6.4.2), the competitive ratio of the online algorithm $\mathcal{A}$ bounded by $CR_{\mathcal{A}} \leq CR_A(\mathcal{R}) + CR_B(\mathcal{R})$.

Theorem 6.4.17 *The competitive ratio of the online algorithm $\mathcal{A}$ is $CR_{\mathcal{A}} = \mathcal{O}(\eta \cdot \varphi \cdot \sigma^5 \cdot h)$ for any arbitrary set of (online) move requests in dynamic executions.*

6.5 Analysis of Existing Directories

In this section, we analyze several existing distributed hierarchical directory based protocols, particularly Spiral [129], Ballistic [77], AP-algorithm [13, 15], and also some hierarchical directory based tracking algorithms for sensor and mobile ad hoc networks, namely STALK [41, 42] and LLS [1]. Recall that Spiral and Ballistic were for the data-flow distributed implementation of software transactional memory, whereas AP-algorithm, STALK, and LLS, were for the mobile user tracking problem in sensor and mobile ad hoc networks. Moreover, Spiral and AP-algorithm are suitable for arbitrary network topologies, Ballistic is suitable for constant-doubling metric topologies, STALK is suitable for geometric network topologies as defined in [21], and LLS is suitable for unit disk graph topologies.

The Spiral Protocol: it uses a hierarchical sparse cover, more specifically $(\mathcal{O}(\log n), \mathcal{O}(\log n))$-labeled cover hierarchy (details in [129]), developed based on well-known ideas for clustering the graph to approximate graph distance metrics by distributions over tree metrics [19, 49], on a general metric network. It has $h + 1 = \mathcal{O}(\log D)$ levels. It was shown in [129] that the $(\mathcal{O}(\log n), \mathcal{O}(\log n))$-labeled sparse cover hierarchy can be constructed in deterministic polynomial time. The sparse cover hierarchy of Spiral can be converted to the hierarchy of leaders $\mathcal{Z}$ by considering only the leader nodes of all the clusters in the hierarchy. At level 0 each node in V is a leader as each cluster consists of only one node. At the root level (level h), there is only one cluster that contains all nodes V, so the leader of that cluster is considered for $\mathcal{Z}$. In any level i of $\mathcal{Z}$, $1 \leq i \leq h-1$, each node $u \in V$ belongs to exactly $\mathcal{O}(\log n)$ clusters, where each cluster is treated as different and the leaders of these clusters are considered for $\mathcal{Z}$. Based on $\mathcal{Z}$, a *spiral path*, denoted as $p(u)$, is built by visiting designated leader nodes (leader nodes are chosen arbitrarily among the nodes in the cluster) in all the clusters that u belongs to starting from level 0 up to h. The downward paths are obtained from the fragments of spiral paths that are created after the updates in the hierarchy by *move* operations. Moreover, the requests are served using the spiral paths in their up phase and downward paths in their down phase. The Spiral hierarchy

has the property that $\eta = \mathcal{O}(\log n)$, $\phi_k = \mathcal{O}(2^k \log n)$, and $\phi'_k = 2^{k-1}$ for any level $0 \leq k \leq h$, since $\sigma = 2$. Therefore, $\varphi = \phi_k/\phi'_k = \mathcal{O}(2^k \log n)/2^{k-1} = \mathcal{O}(\log n)$. We note that distributed coordination is achieved by performing the η pointer accesses per level separately at sub-levels according to a labelling of the clusters. Hence, from Theorem 6.4.17, we obtain:

Theorem 6.5.1 $CR_{\mathsf{Spiral}} = \mathcal{O}(\log^2 n \cdot \log D)$ *in dynamic executions.*

The Ballistic **Protocol:** it uses a sequence of connectivity graphs as a directory hierarchy (details in [77]), obtained using a distributed maximal independent set algorithm (e.g. [95]), on a constant-doubling metric network. It has $h + 1 = \mathcal{O}(\log D)$ levels. Due to the use of maximal independent set of leaders, the Ballistic hierarchy directly translates to the hierarchy of leaders $\mathcal{Z}$. Let $\mathcal{Z} = \{Z_0, Z_1, \ldots, Z_h\}$ be the Ballistic hierarchy. It guarantees that: (i) at level 0 each node in V belongs to the connectivity graph Z_0. The level 0 leaders are a maximal independent set of this graph and two nodes x and y are connected if and only if $\mathrm{dist}(x, y) < 2^1$; (ii) in any level i of $\mathcal{Z}$, $1 \leq i \leq h - 1$, only leader nodes in level $i - 1$ join the connectivity graph Z_i. Nodes x and y are connected in Z_i if and only if $\mathrm{dist}(x, y) < 2^{i+1}$. Moreover, the level i leaders are a maximal independent set of the Z_i graph; and (iii) the highest level Z_h contains exactly one node which is called the $root$ node. The neighboring level nodes are connected by edges to form the tree overlay. For the analysis of Ballistic for an arbitrary set of dynamic requests, similar to Spiral hierarchy, we assign different labels for the $\eta = \mathcal{O}(1)$ number of *move parent* nodes of each leader node x at every level i, which are subset of parents within distance $4 \cdot 2^{i+1}$ of x. Moreover, we have that $\sigma \leq 2$. Similarly, according to the hierarchy construction, $\phi_k = \mathcal{O}(2^k)$, and $\phi'_k = 2^{k-1}$ for any level $0 \leq k \leq h$. Hence, $\varphi = \mathcal{O}(1)$. Therefore, from Theorem 6.4.17 substituting η by $\mathcal{O}(1)$, φ by $\mathcal{O}(1)$, and h by $\mathcal{O}(\log D)$, we obtain the following theorem. This theorem holds also for Combine [10] using the Ballistic hierarchy described above as an overlay tree to run Combine in constant-doubling metric networks.

Theorem 6.5.2 $CR_{\mathsf{Ballistic}} = \mathcal{O}(\log D)$ *in dynamic executions.*

The AP-algorithm: it uses a hierarchical directory composed of a hierarchy of $h = \lceil \log D \rceil + 1$ *regional directories* $RD_i, 1 \leq i \leq h$ (details in [15]). The regional directories on higher levels of the hierarchy based on coarser decompositions of the network (i.e., decomposition into larger regions). The purpose of the regional directory RD_i at level i of the hierarchy is to enable a potential searcher to track any user residing within distance 2^i from it. The regional directory construction is based on the concept of *regional matching*. This matching concept relies on a read set $Read(v) \subseteq V$ and a write set $Write(v) \subseteq V$, defined for every vertex v, similar to the one given in Section 6.2. That is, a vertex v writes the information about every user it currently has to all the vertices in $Write_i(v)$; the searcher for the user from the node w queries all the vertices in $Read_i(w)$. Consider the collection $\mathcal{RW}$ of all pairs of sets, namely $\mathcal{RW} = \{Read(v), Write(v) | v \in V\}$. The collection $\mathcal{RW}$ is a 2^i-*regional matching* (for some integer $m \geq 1$) if $Write(v) \bigcap Read(u) \neq \emptyset$ for all $v, u \in V$ such that $\mathrm{dist}(u, v) \leq 2^i$. It was shown in [15] that it is possible to construct an m-regional matching $\mathcal{RW}_{m,k}, m, k \geq 1$, to support *move* operations, with $Deg_{write}(\mathcal{RW}_{m,k}) = 1$ and $Rad_{write}(\mathcal{RW}_{m,k}) = 2k - 1$, where $Deg_{write}(\mathcal{RW}) = \max_{v \in V} |Write(v)|$ and $Rad_{write}(\mathcal{RW}) = \frac{1}{m} \max_{u,v \in V} \{\mathrm{dist}(u, v) | u \in Write(v)\}$. Therefore, we have that $\eta = \mathcal{O}(\log n)$ and $\varphi = \mathcal{O}(\log n)$ in AP-algorithm in dynamic executions, and $\sigma = 2$. Hence, from Theorem 6.4.17, we obtain:

Theorem 6.5.3 $CR_{\mathsf{AP-algorithm}} = \mathcal{O}(\log^2 n \cdot \log D)$ *in dynamic executions.*

The LLS Algorithm: it uses a virtual hierarchical cover of the $M \times M$ plane consisting of exponentially decreasing squares (details in [1]). This partitioning is similar to [15]. It has $h + 1 = \mathcal{O}(\log D)$ levels. Moreover, a level k square has size 2 times the size of the level $k - 1$ square. Therefore, $\varphi = \mathcal{O}(1)$ and also $\sigma = 2$. According to the construction, each process in the move request tracking path has at most 16 nodes to examine, i.e., the number of move parent nodes for each process $\eta = \mathcal{O}(1)$. Hence, through Theorem 6.4.17 we obtain that LLS has competitive ratio $\mathcal{O}(\eta \cdot \varphi \cdot \sigma^5 \cdot h) = \mathcal{O}(\log D)$. It was shown in [1] that LLS is $\mathcal{O}(\log d)$ competitive for the amortized cost of updating the directory due to a cumulative movement of distance d by a moving node. This was by assuming some restricted locality aware version of the unit disk graph network

in the analysis and also some assumptions on the cost metric. However, we note that our analysis of LLS is from the worst-case perspective without such assumptions. We summarize the result in the theorem below.

Theorem 6.5.4 $CR_{\mathsf{LLS}} = \mathcal{O}(\log D)$ *in dynamic executions.*

The STALK Algorithm: it assumes a hierarchical partitioning of processors over locations in geometric networks (details in [41]). The hierarchical structure it maintains is similar to [15]. It has $h + 1 = \mathcal{O}(\log_\sigma D)$ levels, where $\sigma \geq 3$ such that the radius of a level k cluster is at least $\phi'_k = \sigma^k$. Moreover, they have the radius of a level k cluster at most $\phi'_k = m\sigma^k$, where $m \geq 2/\sqrt{3}$. Therefore, $\varphi = \mathcal{O}(1)$. According to the construction, each process in the tracking path has at most one child, i.e., the number of move parent nodes for each process $\eta = \mathcal{O}(1)$. Hence, as σ and m are constants, through Theorem 6.4.17 we obtain that STALK has competitive ratio $\mathcal{O}(\eta \cdot \varphi \cdot \sigma^5 \cdot h) = \mathcal{O}(m \cdot 3^5 \cdot \log_3 D) = \mathcal{O}(\log D)$. We summarize the result in the theorem below.

Theorem 6.5.5 $CR_{\mathsf{STALK}} = \mathcal{O}(\log D)$ *in dynamic executions.*

6.6 Summary and Discussions

We presented and analyzed a framework for distributed hierarchical directories for an arbitrary set of dynamic online requests. We also analyzed several existing distributed directory protocols through the framework. This is the first such analysis of distributed directories that do not use pre-selected spanning trees as an underlying hierarchy. This technique is appealing in the sense that it gives a methodology to analyze a large interesting class of hierarchical directories for any arbitrary set of requests and it subsumes previous techniques for sequential and concurrent execution analysis. For future work, it will be interesting to see whether the linear dependency of the competitive ratio on η can be removed (or at least made sub-linear), without introducing race conditions, for distributed hierarchial directories in some specific topologies. Moreover, we considered FIFO

links on the hierarchical directory which may not be very realistic given that a single logical link may map various physical links. It will be interesting to explore the cost of implementing FIFO logical links when needed.

Chapter 7
NUMA Systems: Load Balanced Model

7.1 Introduction

In the context of DDPs, previous approaches: Arrow [43], Relay [143], Combine [10], Ballistic [77], and Spiral [129], focused only on stretch bounds for various network topologies (see Table 2.2 in Chapter 2 for their properties) and they do not control the congestion. Moreover, DDPs used in [3, 32, 33] have not been analyzed even for the stretch bounds. The network congestion can also affect the overall performance of the algorithm and sometimes it is a major bottleneck. Network congestion is a significant issue for the NUMA systems where multicore chips are connected with each other through high speed interconnect communication links and they are suitable for high performance distributed and parallel computing. We measure the network congestion as the worst node or edge utilization (the maximum number of times the object requests use any edge or node in the network while accessing the shared object).

In this chapter[1], we present MultiBend, a consistency algorithm for shared objects, that is suitable for d-dimensional mesh networks and is load balanced in the sense that it has low congestion (maximum edge utilization), and at the same time maintains low stretch. Mesh networks are appealing due to their use in parallel and distributed computing [2, 34, 96, 120]. Mesh networks are cost-effective and provide great performance solution for diverse applications, simple expansion for future growth, and scalable connection properties. Mesh topologies are used as an underlying

[1]Portions of this chapter published in:
Gokarna Sharma and Costas Busch. Towards Load Balanced Distributed Transactional Memory. *Proceedings of the 18th International European Conference on Parallel Computing* (Euro-Par), LNCS 7484, pp. 403–414, 2012. http://link.springer.com/chapter/10.1007/978-3-642-32820-6_41

backbone network in many distributed clusters and supercomputers. For example, 65,000 nodes of IBM Blue Gene/L are interconnected as a $64 \times 32 \times 32$ 3-dimensional mesh or torus [2]. Recently, IBM Blue Gene/Q integrated 5-dimensional torus [34], where a torus is a variation of the mesh topology.

MultiBend combines in a novel way a consistency algorithm protocol with a routing algorithm to achieve low stretch and load balancing. The low stretch is achieved through a hierarchical directory which we first introduced in [129] for general networks and we adapted here for the mesh network. The load balancing is achieved through an *oblivious routing* approach (e.g., [16, 28, 96]) tailored to the d-dimensional mesh; in particular, we use the oblivious routing algorithm in Busch *et al.* [28]. A routing algorithm is *oblivious* if every path that is selected for each request to route to its destination is chosen independently of every other path. Oblivious routing is preferred as it does not make any assumptions on the network traffic. DDPs and oblivious routing algorithms have been used before separately. Here we combine these algorithms for the first time to achieve simultaneously low stretch and load balancing. The combination is possible because both DDPs and oblivious routing algorithms work on some form of hierarchy of clusters. Therefore, the routing of messages between any two consecutive levels in consistency algorithm hierarchies can be done obliviously to obtain low congestion on the edges that are used by the nodes of the clusters of the consecutive levels. Later, combining the congestion bounds for all the levels, we can obtain a concentration result on congestion. Low stretch is obtained exploiting the properties of cluster hierarchies.

MultiBend is different from previous DDPs, Arrow [43], Relay [143], Combine [10], Ballistic [77], and Spiral [129] (see Table 2.2 in Chapter 2). Although previous DDPs use some kind of hierarchical structures, their constructions are useful only to minimize stretch and they can not be exploited to control congestion. Moreover, MultiBend is different from *distributed hash table protocols* (DHTs), e.g., Chord [134], CAN [108], Pastry [113], and Tapestry [146], that are developed for peer-to-peer networks. We list some of the differences here. DHTs store key-value pairs by assigning keys (or objects) to different nodes; a node will store the values for all the keys

for which it is responsible. MultiBend handles mobile objects whereas objects (or keys) are not mobile in DHTs. Congestion in MultiBend is for each node and edge, whereas in DHTs it is only for some special DHT nodes. Moreover, stretch in MultiBend is related to path graph distances, whereas in DHTs it is the number of hops in special DHT nodes.

7.1.1 Theoretical Contributions

MultiBend works for any arbitrary execution of the *move* operations in $\mathcal{E}$. However, in the analysis, we consider only the *sequential* and the *concurrent (one-shot)* execution of the set $\mathcal{E}$ of *move* operations in MultiBend. In sequential case, we consider an initial *publish* operation followed by a non-overlapping sequence of l *move* operations. For concurrent case, we assume the *one-shot* scenario where all the $l \leq n$ *move* operations come to the system at the same time after an initial *publish* operation and no further requests occur. We discuss later in Section 7.4.2 how MultiBend can be analyzed for the *dynamic* execution where requests in $\mathcal{E}$ arrive to the system in arbitrary moments of time.

Note that OPT might order the requests of $\mathcal{E}$ differently than a consistency algorithm. Let π^* be the order of OPT and π be the order of the consistency algorithm.

- Sequential execution: π^* is same as π in the sequential (or non-overlapping) execution of requests, therefore, the cost of OPT must be at least $A^*(\mathcal{E}) \geq \sum_{i=1}^{l} |\mathrm{dist}(s_i, t_i)|$, the shortest path distance between each source and destination node pair in the π order.

- Concurrent execution: π^* may not be same as the order π provided by the consistency algorithm in the concurrent execution of requests. However, $A^*(\mathcal{E})$ of OPT must be at least the sum of the *Steiner tree* [110] distances of the locations of the request nodes.

For the *move* operations in both sequential and concurrent (one-shot) executions, MultiBend guarantees $\mathcal{O}(d \log n)$ amortized stretch and $\mathcal{O}(d^2 \log n)$ approximation of the optimal congestion on any edge in d-dimensional mesh networks, where n is the number of nodes in the mesh (see Section 7.7 for the approximation bound on node congestion). If we fix the number of nodes n,

then d is at most $\mathcal{O}(\log n)$ for d-dimensional mesh networks with equal d in every dimension. In this scenario, MultiBend achieves stretch bound of $\mathcal{O}(\log^2 n)$ and congestion approximation bound of $\mathcal{O}(\log^3 n)$; this stretch bound of MultiBend either outperforms or matches the bounds of all previous DDPs (refer Table 2.1 for the various properties of previous DDPs, their bounds, and their comparison with our results). Moreover, MultiBend minimizes congestion, whereas all previous DDPs do not address this issue. For fixed d, the *move* stretch of MultiBend is optimal within a loglog factor comparing to the $\Omega(\log n/\log\log n)$ lower bound due to Alon *et al.* [4] for the mobile user tracking problem; the congestion approximation is also optimal within a constant factor in light of the $\Omega(\frac{C^*}{d}\log n)$ lower bound on the approximation ratio of an oblivious algorithm due to Maggs *et al.* [96].

The communication cost of the *publish* operation is proportional to the diameter of the mesh network (i.e., $\mathcal{O}(d \cdot n)$) and it is a fixed initial cost which is only considered once and compensated by the costs of the *move* (or *lookup*) operations which are issued thereafter. The stretch of a *lookup* operation $\wp$ from node s to the owner node o can be defined similarly as of *move* stretch, which is $\frac{|p|}{|\mathrm{dist}(s,o)|}$, where $|p|$ is the number of edges the path p of the request $\wp$ uses in MultiBend and $|\mathrm{dist}(s,o)|$ is the number of edges in the shortest path between s and o. The stretch of *lookup* operations is $\mathcal{O}(d^2)$ in MultiBend even when they are considered individually while their overall edge congestion has $\mathcal{O}(d^2\log n)$ approximation in the d-dimensional mesh. Moreover, MultiBend is shown to be *correct* in the sense that it eventually enqueues every *move* request in the distributed queue and each request is enqueued only once. To the best of our knowledge, this is the first consistency algorithm that achieves low stretch in a load balanced way.

7.1.2 Practical Contributions

We complement the theoretical analysis of MultiBend by the extensive simulations in a 16×16 nodes 2-dimensional mesh network. We simulate MultiBend and some of its variants considering many different sequences of *move* and *lookup* operations on a single shared object and multiple shared objects. The evaluation results show that MultiBend has a very reasonable distance stretch

property along with its low congestion benefits compared to prior DDPs in different execution settings. The simulation is compared to two well-known DDPs, Arrow [43] and Ballistic [77]. Our choice of Arrow and Ballistic for the performance comparison among existing DDPs (Arrow [43], Relay [143], Combine [10], Ballistic [77], and Spiral [129]) is due to the fact that Relay works similar to Arrow, the overlay construction of Combine resembles the overlay construction used in Ballistic, and MultiBend uses some of the algorithmics of Spiral. In particular, our results show that MultiBend is better by at least a factor of 6.85 in balancing the load, in the worst-case, in the 16×16 nodes 2-dimensional mesh network; the distance stretch results of MultiBend are comparable to previous DDPs.

7.1.3 Chapter Organization

The rest of the chapter is organized as follows. We proceed with network model and the construction of a hierarchy for the 2-dimensional mesh in Section 7.2. We present MultiBend protocol in Section 7.3 for the 2-dimensional mesh. We then analyze our protocol for both stretch and congestion in Section 7.4. In Section 7.5, we extend MultiBend for the d-dimensional mesh, where d is not assumed to be fixed. We then present simulation results of the implementation of MultiBend for a single shared object and multiple shared objects in Section 7.6. We conclude the chapter in Section 7.7 with a short discussion.

7.2 Preliminaries

7.2.1 Network Model

We begin with some necessary definitions which are adapted from [28, 129]. We represent a distributed network as a d-dimensional mesh. The d-dimensional mesh $M = (V, E)$ is a d-dimensional grid of nodes (network machines) V, where $|V| = n$, with side length m_i in each dimension such that $n = \prod_{i=1}^{d} m_i$, and edges (interconnection links between machines) $E \subseteq \binom{V}{2}$. Each computing node $u \in V$ is connected with each of its $2d$ neighbors (except the nodes at the

boundaries of the mesh). We denote by $|E|$ the number of edges in M. We consider that the number of nodes, m_i, at each dimension of the mesh network is a power of 2 and m_i is equal in every dimension. However, our results hold also for mesh networks (including d-dimensional) where dimensions are within a constant factor of each other. If the dimensions are within a constant factor, the worst-case stretch and congestion bound increase is also within the same factor. A path p in M is a sequence of nodes with respective sequence of edges connecting the nodes, such that the length of the path p, denoted $\mathrm{length}(p)$, is the number of edges it uses. A *sub-path* of p is any path obtained by a subsequence of consecutive edges in p; we may also refer to a sub-path as a *fragment* of p. Let $\mathrm{dist}(u, v)$ denote the shortest path length (distance) between nodes u and v.

Consider a routing problem Π defined as a set of pairs of source and destination nodes. A routing algorithm for Π provides paths from every source to its respective destination. An algorithm is *oblivious* if the path choice for each pair of source-destination is independent of the path choices of any other pair. The edge (node) congestion C is the maximum number of times any edge (node) is is used by the object requests. Let C^* denote the optimal congestion attainable by any routing algorithm. We have symmetric definitions for node congestion. For a sub-mesh $M' \subseteq M$ (i.e., M' is any mesh that contains inside M), let $\mathrm{out}(M')$ denote the number of edges at the boundary of M', which connect nodes in M' with nodes outside M'. Consider some sub-mesh M' of the network M. Let Π' denote the messages (pairs of sources and destinations) in Π which have either their source or destination in M', but not both. All the messages in Π' will cross the boundary of M'. The paths of these messages will cause congestion at least $|\Pi'|/\mathrm{out}(M')$. Define the boundary congestion of M' to be $B(M', \Pi) = |\Pi'|/\mathrm{out}(M')$. For the problem Π, the boundary congestion $B = \max_{M' \subseteq M} B(M', \Pi)$, the maximum over all its sub-meshes. Clearly, $C^* \geq B$.

We assume that M represents a network in which nodes do not crash, it implements FIFO communication between nodes, and messages are not lost. We also assume that, upon receiving a message, a node is able to perform a local computation and send a message in a single atomic step. MultiBend can be extended to accommodate non-FIFO communication and tolerate unreliable communication links by adapting some of the techniques of Attiya *et al.* [10].

7.2.2 Hierarchical Directory for the 2-Dimensional Mesh

We describe here the hierarchical directory construction for the 2-dimensional mesh, later we discuss how to extend it to higher dimensions ($d > 2$) in Section 7.5. The hierarchical directory construction for the 2-dimensional mesh is interesting because it is simple to construct but shows the benefits of our approach in controlling both stretch and congestion. This hierarchical construction is later used in Section 7.3 to run our MultiBend protocol. In particular, we describe how to represent the 2-dimensional mesh with equal side lengths $m = 2^k, k \geq 0$, as a hierarchy of sub-meshes. We decompose the 2-dimensional mesh M into two types of sub-meshes, type-1 (see Fig. 7.1) and type-2 (see Fig. 7.2), as described below, adapting some notations from Busch *et al.* [28].

- *Type-1 sub-meshes.* There are $k + 1$ levels of type-1 sub-meshes, $i = 0, 1, \cdots, k$. The mesh M itself is the only level k sub-mesh. Every level i sub-mesh can be partitioned into 4 sub-meshes by dividing each side by 2. Each resulting sub-mesh is a type-1 sub-mesh at level $i - 1$. According to this decomposition, at level i, there are $2^{2(k-i)}$ sub-meshes each with side length $m_i = 2^i$. Note that the level 0 sub-meshes are the individual nodes of the mesh.

- *Type-2 sub-meshes.* There are $k - 1$ levels of type-2 sub-meshes, $i = 1, \cdots, k - 1$. The type-2 sub-meshes at level i are obtained by taking the type-1 sub-meshes of that level and shifting them by $-m_i/2$ simultaneously in both dimensions. Some of the shifted sub-meshes are entirely within M and the remaining of the shifted sub-meshes are partially overlapped with M. For the partially overlapped sub-meshes, we keep only their intersection with M. According to this construction, we have that both sides are of length at least $m_{i-1} = 2^{i-1}$ for all the type-2 sub-meshes at level i.

The decomposition described above satisfies the following properties: (i) The type-1 (and also type-2) sub-meshes at a given level are disjoint; (ii) Each type-1 or type-2 sub-mesh at level i can be partitioned into type-1 sub-mesh(es) at level $i - 1$; and (iii) Each type-1 or type-2 sub-mesh at level i is completely contained in a sub-mesh at level $i + 1$ of type-1, type-2, or both.

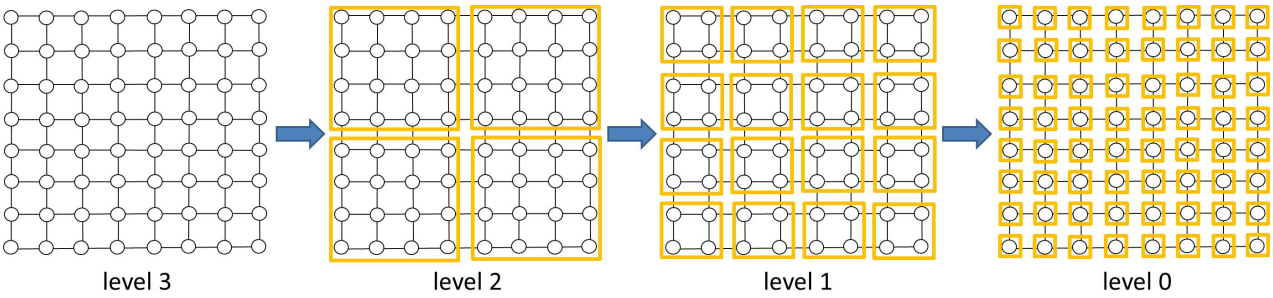


Figure 7.1: Illustration of the decomposition of the $2^3 \times 2^3$ 2-dimensional mesh into type-1 sub-meshes.

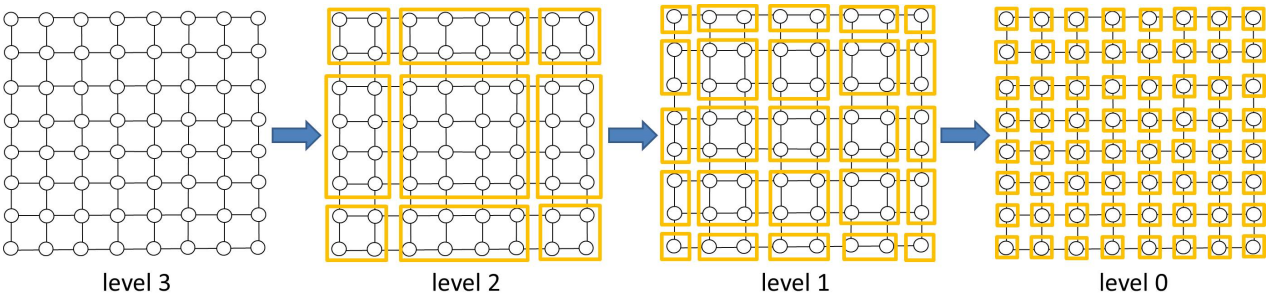


Figure 7.2: Illustration of the decomposition of the $2^3 \times 2^3$ 2-dimensional mesh into type-2 sub-meshes. The decompositions of level 0 and level 3 are omitted from the hierarchy of sub-meshes as they match type-1 decompositions of those levels.

We now define a hierarchy of sub-meshes. The sub-mesh hierarchy $\mathcal{Z} = \{Z_0, Z_1, \ldots, Z_k\}$, is a hierarchy of $k+1$ levels of sub-meshes such that: (i) At level k all nodes in M belong to exactly one sub-mesh, i.e., mesh M itself is the only level k sub-mesh; (ii) At level 0 each node in M is the one sub-mesh by itself; and (iii) In any level i, $1 \leq i \leq k-1$, Z_i contains type-1 and type-2 sub-meshes of level i. We have that $k+1 = \mathcal{O}(\log n)$ in $\mathcal{Z}$ as side lengths of the sub-meshes increase by a factor of 2 between two consecutive levels.

Since there are exactly two types of sub-meshes at any level $0 < i < k$ of $\mathcal{Z}$, we assign sub-level 1 to type-2 sub-mesh and sub-level 2 to type-1 sub-mesh (see Figs. 7.1 and 7.2). This assignment extends the definition of level to sub-level using (i, j), where i is the level and $j \in \{1, 2\}$ is the sub-level. For level 0 and level k we have only one sub-level as there are only type-1 sub-meshes. We assign level $(0, 2)$ to level 0 sub-mesh and level $(k, 1)$ to level k sub-mesh.

Moreover, using this (i, j) definition, we can denote the sub-mesh at any level (i, j) by $X_{i,j}$. In the following, we sometime write sub-level (i, j) instead of level (i, j).

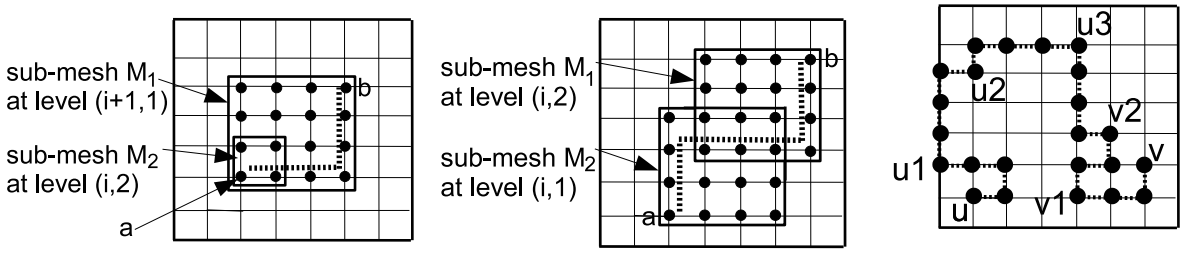
7.2.3 Multi-bend Paths

We define a path $p(u)$ for each node $u \in V$ which we will refer to as the "multi-bend" path of u. The path $p(u)$ is built by visiting a sequence of predetermined leader nodes in all the sub-meshes that u belongs to starting from level 0 up to k. In each level, the sub-meshes are visited according to the lexicographical ordering of their sub-levels.

In every sub-mesh $X_{i,j}$ at level (i, j) a *leader* node is chosen arbitrarily at the initialization of the hierarchy which we denote as $\ell_{i,j} = \ell(X_{i,j})$. If one node is the leader on many level sub-meshes, we add a virtual copy node of it and create a virtual link between the virtual copy and itself in subsequent sub-meshes. Since the top most Z_k consists of a single level $(k, 1)$ sub-mesh $X_{k,1}$ (which is the mesh M by itself) it has a unique leader which we denote by $\ell_{k,1} = \ell(X_{k,1}) = r$ (the root). Trivially, every node $u \in V$ is a leader of its own sub-mesh at level 0, i.e., $\ell_{0,2} = u$ for every node $u \in V$. Note that $\ell(X_{i,j})$ for any level (i, j) sub-mesh $X_{i,j}$ is changed for every request by electing a new leader uniformly at random among the nodes of $X_{i,j}$. This step is done to control congestion.

As needed later in the formal definition of the multi-bend path $p(u)$ for each node $u \in V$, we denote by $\ell_{i,j}(u)$ the leader node of the level (i, j) sub-mesh $X_{i,j}$ in which u belongs to (i.e., $u \in X_{i,j}$). Moreover, we sometime denote the sub-mesh $X_{i,j}$ itself by $X_{i,j}(u)$ to signify that $X_{i,j}$ contains u. When the context is clear, we write $X(u)$ instead of $X_{i,j}(u)$. According to the construction of type-1 and type-2 sub-meshes at each level, there is exactly one sub-mesh at each level (i, j) in which node u belongs to.

From an abstract point of view, the multi-bend path bends (changes dimensions) multiple times while it visits sub-mesh leaders of higher levels. The name of the protocol is inspired from this bending property of multi-bend paths. To be able to bound the congestion, when the multi-bend path needs to visit the leader node of the subsequent sub-mesh from the leader node of the current



(a) One-bend path (b) Two-bend path (c) Multi-bend path

Figure 7.3: Illustration of one-bend, two-bend, and multi-bend paths in the $2^3 \times 2^3$ 2-dimensional mesh

sub-mesh, we ask it to follow only the nodes that are contained in the current and the subsequent sub-meshes.

A *one-bend* path consists of two straight lines, one line in each dimension which meet at a corner where the bend occurs. The one-bend path is sufficient to satisfy our criteria for the multi-bend path when the subsequent sub-mesh completely contains the current sub-mesh. For example, consider the scenario in Fig. 7.3a, where sub-mesh M_2 is completely contained in sub-mesh M_1. In this case, it is possible to visit any node of M_2 from any node in M_1 using a one-bend path, without visiting any node that in not contained in M_1 or M_2. According to the mesh decomposition and the construction of $\mathcal{Z}$ given in Section 7.2.2, this is exactly the case between the leaders of the sub-meshes at level $(i+1,1)$ and at level $(i,2), 0 < i < k$, in the multi-bend path because any type-1 or type-2 sub-mesh at level i is completely contained in a sub-mesh at level $i+1$ of type-1, type-2, or both.

We sometime need two-bend paths between two subsequent leaders in MultiBend. According to the mesh decomposition and the construction of $\mathcal{Z}$ given in Section 7.2.2, every level i of $\mathcal{Z}$ has actually two sub-levels $(i,1)$ and $(i,2)$, where level $(i,1)$ has all type-2 sub-meshes and level $(i,2)$ has all type-1 sub-meshes. Moreover, some type-2 sub-meshes at level $(i,1)$ are not completely contained in some type-1 some-meshes at level $(i,2)$. In this situation, a one-bend path is not always sufficient to visit the leader of the sub-mesh at level $(i,1)$ from the leader of the sub-mesh at level $(i,2)$ in a multi-bend path, without visiting the nodes outside those sub-meshes, and hence a two-bend path is needed between them. A *two-bend* path consists of three straight lines, two lines

in one dimension and they meet at the corners of the third line in other dimension where two bends occur. For example, consider the scenario in Fig. 7.3b, where sub-mesh M_2 is not completely contained in sub-mesh M_1. In this case, it is possible to visit any node of M_2 from any node in M_1 using a two-bend path, without visiting any node that in not contained in M_1 or M_2.

For any pair of nodes $u, v \in V$, let $s(u, v)$ denote a dimension-by-dimension (i.e., change in path from one dimension to the other dimension in every bend) shortest path (an at most two-bend path) from u to v. For any set of nodes $u_1, u_2, \ldots, u_f \in V$, let $s(u_1, u_2, \ldots, u_f)$ denote the concatenation of shortest paths $s(u_1, u_2),\ s(u_2, u_3), \ldots,\ s(u_{f-1}, u_f)$. The multi-bend path $p(u)$ is formed by taking the concatenation of the shortest paths that connect the ascending sequence of leaders of sub-meshes in which u belongs to starting from node u at sub-level $(0, 2)$ in the bottom level up to node r at sub-level $(k, 1)$ in the root level. The shortest paths that connect the subsequent leaders are either one-bend or two-bend paths. For example, see Fig. 7.3c, which shows a multi-bend path from node u to node v in a $2^3 \times 2^3$ 2-dimensional mesh, where u_1, u_2, u_3, v_2, and v_1 are the leader nodes of the clusters the multi-bend path $p(u)$ of u visits to reach v. Formally, the multi-bend path of node u is:

$$p(u) = s(u, \ell_{1,1}(u), \ell_{1,2}(u), \ldots, \ell_{k-1,1}(u), \ell_{k-1,2}(u), r).$$

We say that two multi-bend paths *intersect* if they have a common node. We also say that two multi-bend paths intersect at level i if they visit the same leader node at level i (they may intersect outside leaders but we do not consider that). The lemma below follows from the properties of the sub-mesh hierarchy $\mathcal{Z}$.

Lemma 7.2.1 *For any two nodes $u, v \in V$, their multi-bend paths $p(u)$ and $p(v)$ intersect at level at most $\lceil \log(\mathrm{dist}(u, v)) \rceil + 1$.*

Proof. According to the definition of multi-bend paths, $p(u)$ and $p(v)$ of nodes u and v visit the ascending sequence of leaders of sub-meshes in which they belong to. Suppose $\iota = \lceil \log(\mathrm{dist}(u, v)) \rceil + 1 \leq k$. From the definition of $\mathcal{Z}$, any sub-mesh at level ι has a side length at

least $2^{\iota-1} \geq \mathrm{dist}(u, v)$. Thus, some (type-1 or type-2) sub-mesh $X \in Z_\iota$ will contain both u and v. Therefore, the paths $p(u)$ and $p(v)$ intersect in leader node $\ell(X)$ of the sub-mesh X. □

7.2.4 Canonical Paths

In the analysis of MultiBend, we will examine paths obtained from fragments of multi-bend paths; the fragments are formed while the object moves. These paths start at level 0 and may go up to the root. We will refer to such paths as *canonical*. Formally, a canonical path q up to sub-level $(\alpha, \beta) \leq (k, 1)$ is

$$q = s(x_{0,2}, x_{1,1}, x_{1,2}, x_{2,1}, x_{2,2}, \ldots, x_{\alpha,\beta}),$$

such that $x_{i,j}$'s are leader nodes along the path. A canonical path can be either *partial* when the top node is below level k (below the root), or *full* when the top node is the root. A multi-bend path $p(u)$ is a full canonical path. Any prefix of a multi-bend path is a partial canonical path. We continue to bound the length of a canonical path when we use the sub-mesh hierarchy $\mathcal{Z}$. Particularly, we prove the following lemma.

Lemma 7.2.2 (canonical path length) *For any canonical path q up to level α (any sub-level $(\alpha, 1)$ or $(\alpha, 2)$),* $\mathrm{length}(q) \leq 2^{\alpha+4}$.

Proof. Define the function $\mathrm{next}(i, j)$ (resp. $\mathrm{prev}(i, j)$) which returns the sub-level immediately higher (resp. lower) than the sub-level (i, j). Consider two consecutive nodes $x_{i,j}, x_{\mathrm{next}(i,j)} \in q$, where $(0, 2) < (i, j) < (\alpha, \beta)$ ($\beta \in \{1, 2\}$ in the 2-dimensional mesh decomposition). From the definition of canonical paths, there is a node $y \in V$ with $x_{i,j} = \ell_{i,j}(y)$ and $x_{\mathrm{next}(i,j)} = \ell_{\mathrm{next}(i,j)}(y)$. Therefore, $\mathrm{dist}(x_{i,j}, x_{\mathrm{next}(i,j)}) = \mathrm{dist}(\ell_{i,j}(y), \ell_{\mathrm{next}(i,j)}(y)) \leq \mathrm{dist}(y, \ell_{i,j}(y)) + \mathrm{dist}(y, \ell_{\mathrm{next}(i,j)}(y))$.

We explore the following two cases:

i. $\mathrm{next}(i, j) = (i, j + 1)$: sub-meshes $X_{i,j}(y)$ and $X_{\mathrm{next}(i,j)}(y)$ are at the same level i. We have that the length of at least one side of the sub-mesh $X_{i,j}(y)$ is 2^i and also for sub-mesh $X_{\mathrm{next}(i,j)}(y)$, at least one side is of length 2^i. Since the path between them is constructed by

the dimension to dimension path (with at most 2 bends) $\text{dist}(x_{i,j}, x_{\text{next}(i,j)}) \leq (2^i + 2^i) \leq 2^{i+1}$.

ii. $\text{next}(i,j) = (i+1, 1)$: sub-meshes $X_{i,j}(y)$ and $X_{\text{next}(i,j)}(y)$ are at levels i and $i+1$, respectively. We have that at least one side of the sub-mesh $X_{i,j}(y)$ is of length 2^i and for sub-mesh $X_{\text{next}(i,j)}(y)$ at least one side is of length 2^{i+1}. This gives $\text{dist}(x_{i,j}, x_{\text{next}(i,j)}) \leq (2^i + 2^{i+1}) \leq 2^{i+2}$.

By adding the length of the paths we have that, the path length q for each level i, denoted as q_i, is the sum of the path lengths obtained for cases i and ii, i.e., $q_i \leq 2^{i+1} + 2^{i+2} \leq 2^{i+3}$. The path q is the concatenation of paths constructed by the dimension to dimension shortest paths in sub-meshes of (at least one) sides $2^1, 2^2, \cdots, 2^{\alpha-1}, 2^\alpha$. Therefore the canonical path length q up to level α is: $\text{length}(q) = \sum_{i=1}^{\alpha} q_i \leq \sum_{i=1}^{\alpha} 2^{i+3} \leq 2^{\alpha+4}$. □

7.3 The MultiBend Protocol

We present the MultiBend protocol (Algorithm 10) which is a consistency algorithm for shared objects. For simplicity, we describe it here for a 2-dimensional mesh M using the 2-dimensional mesh decomposition of Section 7.2.2 and for one shared object; the general case for d-dimensional mesh is given in Section 7.5. Moreover, we consider here only one shared object as it is typical in the consistency algorithm literature [10, 43, 77, 129, 143]. We perform some experiments for supporting multiple objects through MultiBend in Section 7.6.

7.3.1 Protocol Overview

Consider some shared object ξ. The protocol guarantees that any moment of time only one node holds the shared object ξ which is the *owner* of the object. The owner is the only node who can modify the object (write the object); the other nodes can only access the object for read.

MultiBend is implemented on the sub-mesh hierarchy $\mathcal{Z}$ constructed in Section 7.2.2. Only the bottom level nodes of $\mathcal{Z}$ can issue requests (*publish*, *lookup*, and *move*) for the shared object

ξ, while nodes in higher levels of $\mathcal{Z}$ are used to propagate the requests in the mesh. The basic objective of MultiBend is to maintain a *directory path* in $\mathcal{Z}$ which is a directed path from the root node r to the bottom-level node that is the current owner of ξ. The directory path is updated whenever ξ moves from one node to another. Initially, the directory path is formed from the multi-bend path $p(v)$ of the object creator node v. As soon as the object ξ is created, v publishes ξ by visiting the leaders in its multi-bend path $p(v)$ towards the root r, making each parent leader node pointing to its child leader (Fig. 5.1a). These leader pointers correspond to path segment between the two consecutive leaders and the concatenation of these path segments from the root r to v form the initial directory path. A *move* request from node u for the object ξ at the owner node v is served by following leader ancestors in its multi-bend path $p(u)$, setting downward links toward it until $p(u)$ intersects the directory path to the owner node, and resetting the directory path it follows while descending towards the owner node v (Figs. 5.1c$-$5.1f); the directory path now points to the requesting node u. As soon as the *move* request reaches the owner node, the object is forwarded from the owner node (node v in Fig. 5.1a) to the requesting node (node u in Fig. 5.1a) along some shortest path in the mesh (Fig. 5.1h). This shortest path is the actual path that the object traverses. Fig. 7.3c depicts a possible path that the *move* operation of Fig. 5.1 follows in the mesh M to reach the owner node of the object. A *lookup* operation is served similar to *move* without modifying the directory path.

7.3.2 Protocol Description

We now provide the protocol description in detail. We define the notion of parent node before giving details of *lookup* and *move*. We denote *parent* node y of a node x in the multi-bend path $p(u)$ as $y = \text{parent}_{p(u)}(x)$, i.e., if y is the sub-level (i, j) sub-mesh leader in $p(u)$ then x is the leader of the immediate lower sub-level sub-mesh leader. Note that the leader of a level 0 sub-mesh is the node itself.

Moreover, we define the notion of special-parent node, which will be useful in reducing the *lookup* cost (see discussion in Section 7.3.3). A *special-parent* node of y, denoted as

Algorithm 10: MultiBend

1 **When** y **receives** $m = \langle v,$ *up*, *publish*$\rangle$ **from** x: // Publish operation
2 set $y.link = x$; **if** y is not a root node **then** send m to $\text{parent}_{p(v)}(y)$;
3 **When** y **receives** $m = \langle u, phase,$ *lookup*$\rangle$ **from** x: // Lookup operation
4 **if** $m = \langle u,$ *up*, *lookup*$\rangle$ **then** // up phase
5 **if** $y.link = \perp$ **then**
6 **if** $y.slink$ list is empty **then**
7 elect a leader w at sub-mesh containing $\text{parent}_{p(u)}(y)$; send m to w;
8 **else** elect a leader w at sub-mesh containing first pointer of $y.slink$ list;
9 send $\langle u,$ *down*, *lookup*$\rangle$ to w;
10 **else** elect a leader w at sub-mesh containing $y.link$; send $\langle u,$*down*,*lookup*$\rangle$ to w;
11 **if** $m = \langle u,$ *down*, *lookup*$\rangle$ **then** // down phase
12 **if** y is a leaf node **then**
13 send the read-only copy of ξ to u and remember u;
14 **else** elect a leader w at sub-mesh containing $y.link$; send m to w;
15 **When** y **receives** $m = \langle u, phase,$ *move*$\rangle$ **from** x: // Move operation
16 **if** $m = \langle u,$ *up*, *move*$\rangle$ **then** // up phase
17 assign $oldlink \leftarrow y.link$ and set $y.link = x$;
18 add y in $slink$ list of y's special parent;
19 **if** $oldlink = \perp$ **then**
20 elect a leader w at sub-mesh containing $\text{parent}_{p(u)}(y)$; send m to w;
21 **else** send $\langle u,$ *down*, *move*$\rangle$ to $oldlink$;
22 **if** $m = \langle u,$ *down*, *move*$\rangle$ **then** // down phase
23 **if** y is in the $slink$ list **then** erase y from $slink$;
24 **if** y is not a leaf node **then** $oldlink \leftarrow y.link$; $y.link \leftarrow \perp$; send m to $oldlink$;
25 **else** send the writable copy of ξ to u;
26 invalidate(ξ) from the owner node v and the read-only copies from other nodes;

$\text{sparent}_{p(u)}(y)$, at sub-level (i, j) in the multi-bend path $p(u)$ is the leader node of one of the sub-meshes $X(u) \in Z_\eta$ at level η, where $\eta = i + 5$, i.e., $\text{sparent}_{p(u)}(y)$ is some *ancestor* leader node of y at level η in $p(u)$. Every node knows its special parent and has a special downward pointer, $slink$, towards a *special-child* node from its special-parent $\text{sparent}_{p(u)}(y)$ (otherwise it is $\perp$). We maintain a list of $slink$ pointers if one node is the special parent for the leaders of several sub-meshes.

We are now ready to provide details of *publish*, *lookup*, and *move* operations. The *publish*(ξ) operation issued by the creator node v assigns downward pointers along the edges of $p(v)$ directed toward v. The pseudo-code for *publish* is given in Lines 1 and 2 of Algorithm 10. As for example, Fig. 5.1a shows hierarchy $\mathcal{Z}$ after a successful *publish* operation. The *move*(ξ) operation issued by

Algorithm 11: Leader election procedure

```
1 select a node w in the sub-mesh containing leader z uniformly at random;
2 copy information at old leader z to new leader w;
3 inform the parent and child of z about the new leader w;
4 construct a sub-path p_i from w_{i-1} to w by picking a dimension by dimension shortest path (where
5    the sub-path is either one-bend or two-bend);
```

a node u is implemented in two phases: (i) in the *up phase*, it is sent from u upward in the hierarchy $\mathcal{Z}$ along $p(u)$ towards the root r until it intersects at a node (i.e. node x) with the directory path; (ii) in the *down phase*, it follows the directory path from node x to the object owner; then the owner sends a copy of ξ to u (along some shortest path in M). In the up phase, the *move* operation sets the directions of the edges in the fragment of $p(u)$ between u and x to point toward u. In the down phase, it deletes the downward pointers (or links) in the fragment of the directory path from x to v, making the new directory path points toward u. Through this process, when the *move*(ξ) operation from u reaches v in its down phase, u obtains a writable copy of ξ from v invalidating the old copy of ξ at v and modifying the directory path (Figs. 5.1c−5.1h). The pseudo-code for *move* is given in Lines 14–25 of Algorithm 10. Moreover, this process has resulted to a canonical directory path that consists of two multi-bend path fragments, a fragment of u's multi-bend path between r and x and a fragment of v's multi-bend path between x and u. Subsequent *move* operations may result into further fragmentation of the directory path into multiple (more than two) multi-bend path fragments.

The *lookup*(ξ) operation issued by a node u is served similarly as of *move*(ξ), but downward pointers are not added and existing downward pointers are not deleted, hence not modifying the existing directory path. The pseudo-code for *lookup* is given in Lines 3–13 of Algorithm 10. Through this process, when the *lookup*(ξ) operation from u reaches v in its down phase, u obtains a read-only copy of ξ from v without invalidating ξ from v and without modifying the existing directory path.

7.3.3 Need of Special Parent

A *lookup* request from any node w for the object ξ at the owner node v may not find the directory path to v at level $\log\lceil(\text{dist}(w,v))\rceil + 1$ leader node X of $\mathcal{Z}$ where their multi-bend paths $p(w)$ and $p(v)$ intersect. This is because after several *move* operations the directory path may become highly fragmented and hence the directory path does not pass through X where $p(w)$ and $p(v)$ intersect. The notion of a special parent node helps to avoid this situation and guarantees efficient *lookup*s, such that whenever a downward link is formed at a node z the special parent of z is also informed about z holding a downward pointer. The special parent has the property that any nearby *lookup* close to z will either reach z or its special parent. In the up phase of the *lookup* request (Lines 4–9 of Algorithm 10), it is forwarded to level $\text{next}(i,j)$ from level (i,j) only if both $link$ and $slink$ pointers are $\perp$ for the leader $\ell(X)$ of the level (i,j) sub-mesh X. The $slink$ pointers are set by *move* operations in their up phase and existing $slink$ pointers are deleted by *move* operations when they follow the directory path previously set by other *move* operations in their down phase (Lines 17 and 22 of Algorithm 10). We prove in Section 7.4 that *lookup* operations are always efficient using special parents (see Lines 6, 8, 17, and 22 of Algorithm 10).

7.3.4 Load Balancing

MultiBend (Algorithm 10) uses a leader election procedure (Algorithm 11) such that *lookup* and *move* requests can be served in a load balanced way. The procedure works as follows: Let z be a leader node of the sub-mesh M' in $\mathcal{Z}$. We elect a new leader at M' by selecting a node $w \in M'$ uniformly at random. After the leader is elected, the information at old leader z is moved to new leader w and the parent and child of z are informed about the new leader w. The pointers inside M' are also updated to point to the new leader. After that, sub-path p_i from w_{i-1} (a leader of the sub-mesh that is sending a message to M') to w is formed by picking a dimension by dimension shortest path; the sub-path p_i is one-bend if sub-mesh containing w and the sub-mesh containing w_{i-1} are both type-1 sub-meshes, otherwise, p_i is of at most two bend path. If the sub-path is the two-bend path then it is picked by a random ordering of dimensions on a random node. The *lookup*

operation uses this procedure to elect the leader as shown in Lines 7, 8, 9, and 13 of Algorithm 10. For *move*, the procedure is invoked at Line 19 of Algorithm 10.

The use of leader election procedure incurs extra cost to the actual cost of the *move* and *lookup* operations. This is because this procedure requires some rounds of message exchanges between the old leader and the new leader, and also with the parent and child of the old leader to inform them about the new leader. We note that the pointer update cost is low in comparison to the cost of serving the requests because only the information in the nearby region needs to be updated due to the new leader. We argue that this step facilitates congestion control. This is because when a fixed leader is used, the node congestion on that leader is proportional to the number of requests that visit that leader. Moreover, in the fixed leader case, edge congestion can also be proportional to the number of requests as all the requests use fixed edges along the shortest path between two subsequent leaders. We study the impact of the extra cost due to the use of leader election procedure in the performance of MultiBend through simulations in Section 7.6.

A multi-bend path selection approach we use plays major role in controlling edge congestion because it minimizes the overutilization of edges by random ordering of dimensions while connecting two randomly selected subsequent sub-mesh leaders in the hierarchy. In other words, our approach forms multi-bend paths that change dimensions independently for each operation, starting first in the horizontal or vertical direction, and may follow different set of nodes every time we route requests between two different leaders. We also note that, if the congestion requirement on edges (or nodes) can be relaxed by the factor of ρ, then leader change is needed only after every ρ requests in our approach. Our simulation results in Section 7.6 show the trade-off between stretch and congestion in various leader change frequencies.

Moreover, we observe that at any time a request locks at most three nodes (level $\mathrm{prev}(i,j)$, (i,j), and $\mathrm{next}(i,j)$) along the multi-bend path or a directory path. In concurrent situations this might be a problem. This is because we need to lock more than one node (at most three nodes) in the multi-bend path to do the random leader election as described in Algorithm 11, otherwise directory information necessary for generating a new path may get lost. Therefore, in the concurrent

execution of *move* requests, we need to make sure that the nodes that are affected by the random leader election should be kept locked until all the steps of Algorithm 11 have been executed. We can use the notion of *conflict graph* for each level such that neighbors in the conflict graph can not perform the random leader election at the same time (that is, they can not be in the critical section at the same time). But the non-neighbors can be in the critical section at any time. Using this setup, the performance of MultiBend for *move* operations in concurrent executions will be the same as its performance for the sequence of *move* operations in sequential executions (see Sections 7.4.1 and 7.4.2). Note also that the special parent node does not need to be locked because only one specific $slink$ pointer value needs to be updated at any time.

7.4 Performance Analysis

We give the stretch and congestion analysis of MultiBend for sequential and concurrent (one-shot) executions. The correctness proof of MultiBend is omitted as it can be easily proven by extending the correctness proofs of Ballistic [77], Combine [10], and Spiral [129].

7.4.1 Performance in Sequential Executions

Move Cost: We now give the analysis of MultiBend in sequential executions. As *move* requests are non-overlapping in sequential executions, the system attains quiescent configuration after a request is served and until a next request is issued, i.e., a next request will be issued only after the current request finishes. Let us define a sequential execution of a set $\mathcal{E}$ of $l+1$ requests $\mathcal{E} = \{r_0, r_1, \cdots, r_l\}$ for the object ξ, where r_0 is the initial *publish* request and the rest are the subsequent *move* requests (we do not include *lookup* operations in $\mathcal{E}$ since they do not add or remove links in the directory hierarchy $\mathcal{Z}$, and hence do not impact the performance of other *move* or *lookup* operations).

For the sake of analysis, similar as in [129], we define a two-dimensional array B of size $(k+1) \times (l+1)$, where $k+1$ and $l+1$ are the number of rows and columns, respectively. The $k+1$ rows of B can be denoted as $\{row_0, row_1, \cdots, row_k\}$, and the $l+1$ columns of B can be

denoted as $\{col_0, col_1, \cdots, col_l\}$. All the locations of the array B are initially empty ($\perp$). We fix that $[0,0]$ is the lower left corner element and $[k,l]$ be the upper right corner element. The levels visited by each request r_i in the hierarchy $\mathcal{Z}$ while searching for the object are registered in each $col_i, 0 \leq i \leq h$. The maximum level reached by a request r_i before it finds the downward pointer ($link$ or $slink$) in $\mathcal{Z}$ is called the *peak level* for that request. We have that $h \leq k$. The peak level reached by r_0 (the *publish* request) is always k, the maximum level in $\mathcal{Z}$, and r_0 is registered at all the locations of col_0 starting from $col_0[0]$ and ending in $col_0[k]$.

Let $A^*(\mathcal{E})$ denote the optimal cost for serving requests in $\mathcal{E}$ through OPT and $A(\mathcal{E})$ denote the total communication cost for serving requests in $\mathcal{E}$ using MultiBend. We will bound the stretch $\max_{\mathcal{E}} A(\mathcal{E})/A^*(\mathcal{E})$. For simplicity, we consider only the cost incurred by the up phase of each *move* request. When we consider also the cost incurred by the down phase, the stretch increases by a factor of 2 only. For any $c, d, 0 \leq c < d \leq l$, a *valid pair* $W^j_{(c,d)}$ of two non-empty entries in $row_j, 0 \leq j \leq h$ is defined as $W^j_{(c,d)} = (row_j[c], row_j[d])$, such that $row_j[c] \neq \perp$ and $row_j[d] \neq \perp$, and if $d - c > 1$, then $\forall e, c+1 \leq e \leq d-1$, $row_j[e] = \perp$. In other words, $W^j_{(c,d)}$ is a pair of two subsequent non-empty entries in a row. Moreover, we denote by S_j the total count of the number of entries $row_j[i], 0 \leq i \leq l$, such that $row_j[i] \neq \perp$, and by W_j the total number of valid pairs ($W^j_{(c,d)}$) in it. We have that $W_j = S_j - 1$.

Theorem 7.4.1 (*move* stretch) *The move stretch of* MultiBend *is* $\mathcal{O}(\log n)$ *for sequential executions.*

Proof. Let $A^*_h(\mathcal{E})$ be the optimal communication cost for serving requests in $\mathcal{E}$ that reach level h in the hierarchy $\mathcal{Z}$. According to the execution setup, S_h is the number of requests in $\mathcal{E}$ that reach level h, and W_h is the total number of valid pairs at that level. For any two subsequent requests that originate from nodes u and v and reach level h, $\text{dist}(u,v) \geq 2^{h-1}$ (according to Lemma 7.2.1), since otherwise their multi-bend paths would intersect at level $h-1$ or lower. Therefore $A^*_h(\mathcal{E}) \geq W_h \cdot 2^{h-1} \geq (S_h - 1)2^{h-1}$, as $W_h = S_h - 1$. Considering all the levels from 1 to k, we can say that optimal cost $A^*(\mathcal{E})$ is at least $A^*(\mathcal{E}) \geq \max_{1 \leq h \leq k} A^*_h(\mathcal{E}) \geq \max_{1 \leq h \leq k}(S_h - 1)2^{h-1}$.

Similarly, let $A_h(\mathcal{E})$ be the total communication cost of MultiBend for all the requests in $\mathcal{E}$ that reach level h in the hierarchy $\mathcal{Z}$, while probing the shared object in their up phase. According to the execution setup, $A_h(\mathcal{E})$ is the total communication cost for serving S_h requests that reach level h using MultiBend. We have that $A_h(\mathcal{E}) \leq (S_h - 1)2^{h+4}$ (Lemma 7.2.2). By combining the cost for each level, $A(\mathcal{E}) = \sum_{h=1}^{k} A_h(\mathcal{E}) \leq \sum_{h=1}^{k}(S_h - 1)2^{h+4}$, in the worst-case. We do not need to consider level 0 for $A^*(\mathcal{E})$ and $A(\mathcal{E})$ because there is no communication at that level.

Since the execution $\mathcal{E}$ is arbitrary and $\sum_{h=1}^{k}(S_h - 1)2^{h+4} \leq k \cdot \max_{1\leq h\leq k}(S_h - 1)2^{h+4}$,

$$\begin{aligned}\max_{\mathcal{E}} \frac{A(\mathcal{E})}{A^*(\mathcal{E})} &\leq \frac{k \cdot \max_{1\leq h\leq k}(S_h - 1)2^{h+4}}{\max_{1\leq h\leq k}(S_h - 1)2^{h-1}} \\ &\leq 32 \cdot k \leq 32 \cdot (\lceil \log n \rceil + 1) = \mathcal{O}(\log n),\end{aligned}$$

as $k = \lceil \log n \rceil + 1$. □

Note that the *move* stretch of Theorem 7.4.1 does not take into account the cost of leader election procedure (Algorithm 11). As described in Section 7.3.4, the leader election procedure incurs extra cost. We argue here that the cost of leader election is low as only limited number of nodes are involved in the leader election process. We analyze the impact of the cost of leader election procedure in the performance of MultiBend through simulations in Section 7.6, which shows that the cost due to the leader election procedure is approximately 3 times more than the cost due to actual move operations. Nevertheless, this increase in cost in turn helps us in obtaining significantly low congestion approximation (from linear on n to logarithmic on n).

Congestion: We relate the congestion of the paths selected by MultiBend to the optimal congestion C^*. In particular, we prove the following theorem (this bound is valid for both *move* and *lookup* operations, as both operations do random leader change in the same way). We use the following Chernoff bound.

Lemma 7.4.2 (Chernoff bound) *Let $Y_1, Y_2, \cdots, Y_n$ be independent Poisson trials such that, for $1 \leq i \leq n$, $P[Y_i = 1] = pr_i$, and $P[Y_i = 0] = qr_i = 1 - pr_i$, where $0 < pr_i, qr_i < 1$. Then, for $Y = \sum_{i=1}^{n} Y_i, \mu = E[Y] = \sum_{i=1}^{n} pr_i$, and any $\delta \geq 2e - 1$, $P[Y > (1+\delta)\mu] < 2^{-\mu(1+\delta)}$.*

Theorem 7.4.3 (congestion) MultiBend *achieves $\mathcal{O}(\log n)$ approximation on congestion with high probability.*

Proof. Recall that every request from its source node to its destination node is routed by MultiBend by selecting some paths. Precisely, these paths are the multi-bend paths. Let e denote an edge in the mesh graph M and $C(e)$ denote the load on e (the number of times the edge e is used by the paths of the requests). We bound the probability that some multi-bend path uses edge e. Consider a fragment of a path p from a sub-mesh M_1 to a sub-mesh M_2, which we call the sub-path p_i of p, such that $M_1 \subseteq M_2$ and e is a member of M_2. If M_1 is of type-1 then all of its sides are equal to m_ℓ, where ℓ is the level of M_1. Then the sub-path p_i uses edge e with probability at most $2/m_\ell$. Moreover, a one-bend sub-path is enough to route the request from M_1 to M_2. We deal with the case of type-2 sub-meshes later.

Let P' be the set of paths that go from M_1 to M_2 (or vice-versa). Let $C'(e)$ denote the congestion that the paths P' cause on e. Using the similar argument as given in previous paragraph for an edge e, the upper bound in $C'(e)$, denoted as $E[C'(e)]$, is bounded by $E[C'(e)] \leq 2|P'|/m_\ell$. This is because, we can write $P' = P_1 \cup P_2$, where P_1 is the set of sub-paths from M_1 to M_2 and P_2 is the set of sub-paths from M_2 to M_1. Therefore, expected congestion on edge e due to sub-paths in P_1 is $2|P_1|/m_\ell$ and due to sub-paths in P_2 is $2|P_2|/m_\ell$. Summing the congestion due to P_1 and P_2, we get the desired bound $E[C'(e)] \leq 2|P'|/m_\ell$.

Moreover, from the definition of the boundary congestion, $B \geq B(M_1, \Pi) \geq |P'|/\mathrm{out}(M_1)$. Thus, $C^* \geq |P'|/\mathrm{out}(M_1)$. Since M_1 has all sides of length m_ℓ nodes, $\mathrm{out}(M_1) \leq 4m_\ell$. Therefore, $E[C'(e)] \leq 8C^*$. We charge this congestion to sub-mesh M_2. Between every sub-level $(i, 2)$ sub-meshes, $1 \leq i \leq k-1$, as M_1 of sub-level $(i, 2)$ is completely contained in M_2 of sub-level $(i+1, 2)$ and there are at most $k < \lceil \log n \rceil + 1$ levels, the expected congestion on edge e, denoted as $E[C(e)]$, is bounded by $E[C(e)] \leq 8C^*(\lceil \log n \rceil + 1)$.

According to our construction, there is only one type-2 sub-mesh M_1' between every two type-1 sub-meshes M_1 and M_2 in the sub-mesh hierarchy $\mathcal{Z}$. As the type-2 sub-mesh M_1' may not be the proper subset of M_2, the set of paths from M_1 to M_1' may go through four possible type-2 sub-meshes and they may bend at most two times before they reach to the leader node of M_2. This will increase the congestion by at most the factor of 4 between every two type-1 sub-meshes M_1 and M_2. Moreover, since only sub-meshes up to level $k < \lceil \log n \rceil + 1$ can contribute to the congestion on edge e and there are at most $(\lceil \log n \rceil + 1)$ levels, $E[C(e)] \leq 32C^*(\lceil \log n \rceil + 1)$.

As every request selects its path independently of every other request (Algorithm 11), we now derive a concentration result on the congestion C, using a standard Chernoff bound given in Lemma 7.4.2. Let $Y_i = 1$ if a multi-bend path p_i uses edge e; otherwise $Y_i = 0$. Then $E[C(e)] = E[\sum_i Y_i] \leq 32C^*(\lceil \log n \rceil + 1) \leq 32C^*(\log n + 2)$. For $|E| > 4$, we have that $E[C(e)] \leq 32C^* \log(|E|n)$. As $C^* \geq 1$, using Lemma 7.4.2,

$$P[C(e) > 32\kappa C^* \log(|E|n)] \quad < \quad 2^{-32k \log(|E|n)} < 2^{\log((|E|n)^{-32\kappa})} < (|E|n)^{-32\kappa},$$

for some constant $\kappa = 2e + 1$. Taking the union over all the edges $e \in E$,

$$P\left[\max_{e \in E} C(e) > 32\kappa C^* \log(|E|n)\right] < \frac{1}{(|E|n)^{32\kappa - 1}}.$$

As $|E| = \mathcal{O}(n^2)$, we achieve $C = \mathcal{O}(C^* \log n)$ with high probability. □

This congestion bound is valid for object operations on a single shared object. Whenever multiple objects support is needed for a consistency algorithm, a directory hierarchy can be constructed for each shared object. In this scenario, one interesting question is whether the leader election procedure still required to minimize the congestion. We address this question through extensive simulations in Section 7.6.

Publish Cost: We prove the following theorem for the communication cost of any *publish* operation.

Theorem 7.4.4 (*publish* cost) *The publish operation has communication cost $\mathcal{O}(n)$.*

Proof. A *publish* operation adds downward links on the publishing leaf node's multi-bend path towards the root node r in the sub-mesh hierarchy $\mathcal{Z}$. Moreover, notice that the number of levels in the hierarchy $k < \log n + 2$ (Lemma 7.2.1) and a multi-bend path is trivially a canonical path. Therefore, the theorem immediately follows from Lemma 7.2.2. □

Lookup Cost: For the *lookup* stretch, assume that some node w issues a *lookup* request $\wp$ for the shared object ξ and there is no other *lookup* request in the system. We prove the following theorem for any *lookup* operation.

Theorem 7.4.5 (*lookup* stretch) *The stretch of* MultiBend *is $\mathcal{O}(1)$ for any lookup operation.*

Proof. We explore two different cases of a *lookup* request execution: one when there is no *move* request in the system and the other when there are *move* requests in the system. If there is no *move* request in the system, it is trivial to see that a *lookup* request $\wp$ from w finds the directory path to the owner node v at level $\lceil \log(\text{dist}(w, v)) \rceil + 1$ (Lemma 7.2.1), following the multi-bend path $p(w)$, where $\text{dist}(w, v)$ is the distance of the owner node v from the requesting node w.

When there are *move* requests in the system, the *lookup* request $\wp$ from w may not find the directory path to the shared object at level $\lceil \log(\text{dist}(w, v)) \rceil + 1$ because the directory path to v might have been deformed significantly such that the level $2^{(\lceil \log(\text{dist}(w,v)) \rceil + 1)}$ parents of w in its multi-bend path $p(w)$ have no information about the owner node v. Nevertheless, we can prove that if a node w issues a *lookup* request $\wp$ for the shared object ξ currently owned by a node v which is at distance $\text{dist}(w, v) = 2^i$ far from w, the multi-bend path $p(w)$ is guaranteed to either intersect with the directory path to v or find a $slink$ to the directory path for the object at level at most η, where $\eta = i + 5$.

The intuition behind the proof is as follows. Let us assume that $x = \ell(X)$ is the leader of the sub-mesh X at level $\eta = i + 5$, which has a *slink* information to level i leader v_i (set by some previous *move* request), where v_i is the level i leader node in the directory path to the owner node

v. For the *lookup* $\wp$ to find *slink* to v_i, X must include w (since the multi-bend path $p(w)$ of w visits the leaders of all the sub-meshes that contain it). Now, it suffices to show that the side length of X is at least the distance $\mathrm{dist}(v_i, w)$ to guarantee that X contains w. As the canonical path up to level i from v (the owner node), denoted by q_i, is of length 2^{i+4} (Lemma 7.2.2) and $\mathrm{dist}(w, v) = 2^i$, $\mathrm{dist}(v_i, w) \leq \mathrm{length}(q_i) + \mathrm{dist}(v, w) \leq 2^{i+4} + 2^i \leq 2^{i+5}$. That is, some sub-mesh should be of side length at most 2^{i+5} to have such information, and thus, such sub-mesh will be at level $\eta = i + 5$. Therefore, X contains w.

We are now ready to bound the stretch $A(\wp)/A^*(\wp)$ of MultiBend for a *lookup* operation $\wp$, where $A(\wp)$ is the total communication cost of serving the *lookup* request $\wp$ using the MultiBend protocol and $A^*(\wp)$ is the optimal cost of serving the *lookup* request $\wp$ through OPT. The total cost of MultiBend for $\wp$ is at most the sum of the distances $\mathrm{length}(p_\eta(w))$ (the length of the spiral path $p(w)$ up to level η), $\mathrm{dist}(x, v_i)$ (the path length between level η leader x and the level i leader v_i in the canonical path towards the owner node v), and $\mathrm{length}(q_i)$ (the canonical path length of v up to v_i, the level i leader in q). Therefore, the total cost of MultiBend for the *lookup* request $\wp$ (after substituting η by $i + 5$) is bounded by:

$$\begin{aligned} A(\wp) &= \mathrm{length}(p_\eta(w)) + \mathrm{dist}(x, v_i) + \mathrm{length}(q_i) \\ &\leq 2^{\eta+4} + 2^{\eta+3} + 2^i \\ &\leq 2^{i+5+4} + 2^{i+5+3} + 2^i \leq 2^{i+10}. \end{aligned}$$

As w and v are $\mathrm{dist}(w, v) = 2^i$ apart, the optimal cost is at least $A^*(\wp) \geq 2^i$ for the *lookup* request $\wp$ to get the shared object ξ at v following the shortest path between w and v in the mesh M. Hence, the stretch of MultiBend for a *lookup* operation is $\frac{A(\wp)}{A^*(\wp)} \leq \frac{2^{i+10}}{2^i} = \mathcal{O}(1)$, as needed. □

7.4.2 Performance in Concurrent Executions

The performance analysis of MultiBend given above does not apply to concurrent executions because the adversary is not allowed to gain by ordering the requests in a smarter way, i.e., the

orderings provided by both MultiBend and OPT are the same. Concurrent executions can change the order of the requests in execution and hence affect the performance of the MultiBend protocol. In one-shot execution, all requests come concurrently (at the same time) in the system. We study the following one-shot instance of concurrent execution. At time t as soon as a *publish* operation started at time 0 finished execution, $R \subseteq V$ nodes issue a *move* request concurrently and no further requests occur. We divide the time into periods and rounds such that a level i round has i non-overlapping aligned periods, and we assume that all requests proceed in rounds. Now when two or more requests reach to level i one is forwarded towards level $i+1$ and other(s) is "deflected" down following the directory path set by the previously upward forwarded request in the hierarchy $\mathcal{Z}$. Defining total and optimal cost for one-shot execution similar to sequential execution, the optimal cost for any level i is given by the *Steiner tree* [110] of the requests that reach that level. The total cost analysis is similar as of sequential execution, and also the analysis for approximation on congestion, and *lookup* and *publish* bounds. Therefore, we summarize the bounds in concurrent executions in the theorem below.

Theorem 7.4.6 *The move stretch of* MultiBend *is* $\mathcal{O}(\log n)$ *for concurrent (one-shot) executions. It achieves* $\mathcal{O}(\log n)$ *approximation on congestion with high probability. Moreover, the* publish *operation has* $\mathcal{O}(n)$ *cost and the* lookup *operation has* $\mathcal{O}(1)$ *stretch.*

The performance of MultiBend can also be analyzed for requests that are initiated in arbitrary moments of time (i.e., dynamic executions). This analysis can capture the execution scenarios where requests are neither completely sequential as considered in Section 7.4.1 nor completely one-shot as considered in Section 7.4.2. Using the technique proposed recently by Sharma and Busch [126], we can show that Theorem 7.4.6 holds for MultiBend also in dynamic executions.

7.5 Extensions to the d-Dimensional Mesh

The 2-dimensional sub-mesh hierarchy (Section 7.2.2) can be generalized directly for the sub-mesh hierarchy construction of the d-dimensional mesh, but the distance stretch becomes $\mathcal{O}(2^d \log n)$

for *move* operations. Moreover, the approximation on congestion also becomes $\mathcal{O}(2^d \log n)$. Therefore, we outline an alternative decomposition that has $\mathcal{O}(d \log n)$ approximation on the distance stretch and $\mathcal{O}(d^2 \log n)$ approximation on the edge congestion for *move* operations in d-dimensional mesh networks. Recall that we do not fix d, i.e. the dimension d is not assumed to be constant. The decomposition will have type-1 sub-meshes and other shifted sub-meshes. We set $\lambda = \max\{1, m_\ell/2^{\lceil \log d+1 \rceil}\}$, where m_ℓ is the side length of the level ℓ type-1 sub-mesh. The type-1 sub-meshes at level ℓ are shifted by $(j-1)\lambda$ nodes in each dimension to get the type-j sub-meshes for $j > 1$. If the resulting sub-mesh is not entirely within the mesh M, we only keep the part of it that is overlapped with M. Therefore, all the type-j sub-meshes are not squares like type-1 sub-meshes. According to this decomposition, there will be at most $2(d+1)$ different types of sub-meshes at any level. The hierarchy $\mathcal{Z}$ is formed similar to the 2-dimensional mesh but now there will be $2(d+1)$ sub-levels at each level (instead of 2 sub-levels in the 2-dimensional mesh case). A multi-bend path $p(u)$ for a node u is formed by taking the concatenation of the shortest paths that connect the ascending sequence of leaders of the sub-meshes in which u belongs to starting from node u (sub-level $(0, \mathcal{O}(d))$ to the root node r (sub-level $(k, 1)$). The canonical paths can also be defined similarly to Section 7.2.2. We can prove following results in d-dimensional meshes.

Lemma 7.5.1 (canonical path length in d-dimensional mesh) *In d-dimensional mesh networks, for any canonical path q up to level α (any sub-level $(\alpha, \beta), 1 \leq \beta \leq \mathcal{O}(d)$),* $\text{length}(q) \leq \mathcal{O}(d2^\alpha)$.

Proof. As defined in Section 7.2.4, a canonical path q is the concatenation of paths constructed by the dimension to dimension shortest paths in the sub-meshes starting from the lowest level $(0, \mathcal{O}(d))$ to the level $(\alpha, \beta), 1 \leq \beta \leq \mathcal{O}(d)$. Moreover, $\mathcal{O}(d)$ sub-levels in each level are visited in the ascending order by the canonical path. In the d-dimensional mesh, the canonical path length increases by a factor of $\mathcal{O}(d)$ in comparison to the canonical path length for the 2-dimensional mesh given in Lemma 7.2.2 because, given any two nodes s and t in the d-dimensional mesh, there is some sub-mesh of some type-j that completely contains s and t and has side length $\mathcal{O}(d \cdot \text{dist}(s,t))$ [28]. Therefore, in the worst-case, the total path length from type-1 sub-mesh at level i to type-1

mesh at level $i+1$ is $\mathcal{O}(d2^i)$. Combining the cost for all the $\mathcal{O}(\log n)$ levels similar to Lemma 7.2.2, we get the desired bound. □

Theorem 7.5.2 (*publish* cost for d-dimensional mesh) *In d-dimensional mesh networks, any publish operation by* MultiBend *has cost $\mathcal{O}(d \cdot n)$.*

Proof. Similar to theorem 7.4.4, as $k < \log n + 2$ and a multi-bend path is trivially a canonical path, the theorem follows immediately from Lemma 7.5.1. □

Theorem 7.5.3 (*lookup* stretch in d-dimensional mesh) *In d-dimensional mesh networks, the stretch of* MultiBend *is $\mathcal{O}(d^2)$ for any lookup operation.*

Proof. Similar to Theorem 7.4.5, even if there are *move* requests in the system, we can prove that if a node w issues a *lookup* request $\wp$ for the shared object ξ currently owned by a node v which is at distance $\mathrm{dist}(w,v) = 2^i$ far from w, the multi-bend path $p(w)$ is guaranteed to either intersect with the directory path to v or find a $slink$ to the directory path for the object at level at most η, where $\eta = i + 1 + \log d + \log c_1$, where c_1 is some constant. This is because, similar to Theorem 7.4.5, as $\mathrm{length}(q_i) \leq c_1 d 2^i$ (Lemma 7.5.1), we have that $\mathrm{dist}(v_i, w) \leq \mathrm{length}(q_i) + \mathrm{dist}(v, w) \leq c_1 d 2^i + 2^i \leq c_1 d 2^{i+1} \leq 2^{i+1+\log d + \log c_1}$. Therefore, some sub-mesh should be of side length at most $i+1+\log d+\log c_1$ to have such information, thus such sub-mesh will be at level $i+1+\log d+\log c_1$.

Moreover, similar to Theorem 7.4.5, the total cost $A(\wp)$ of MultiBend for the *lookup* operation $\wp$ is at most the sum of the distances $\mathrm{length}(p_\eta(w))$ (the length of the spiral path $p(w)$ up to level η), $\mathrm{dist}(x, v_i)$ (the path length between level η leader x and the level i leader v_i in the canonical path towards the owner node v), and $\mathrm{length}(q_i)$ (the canonical path length of v up to v_i, the level i leader in q). Therefore (after substituting n by $i + 1 + \log d + \log c_1$),

$$\begin{aligned} A(\wp) &= \mathrm{length}(p_\eta(w)) + \mathrm{dist}(x, v_i) + \mathrm{length}(q_i) \\ &\leq c_1 d 2^\eta + c_1 d 2^{\eta-1} + 2^i \\ &\leq c_1 d 2^{i+1+\log d+\log c_1} + c_1 d 2^{i+\log d+\log c_1} + 2^i \\ &\leq c_1 d 2^{i+2+\log d+\log c_1}. \end{aligned}$$

The optimal cost is at least $A^*(\wp) \geq 2^i$ for the *lookup* request $\wp$, as w and v are $\text{dist}(w, v) = 2^i$ apart. Hence, the stretch $\frac{A(\wp)}{A^*(\wp)} \leq \frac{c_1 d 2^{i+2+\log d + \log c_1}}{2^i} = \mathcal{O}(d^2)$, as c_1 is a constant. □

Theorem 7.5.4 (*move* stretch in d-dimensional mesh) *In d-dimensional mesh networks,* MultiBend *has $\mathcal{O}(d \log n)$ stretch for* move *operations.*

Proof. Let $A_h^*(\mathcal{E})$ be the optimal communication cost for serving requests in $\mathcal{E}$ that reach level h in the hierarchy $\mathcal{Z}$. According to the execution setup, S_h is the number of requests in $\mathcal{E}$ that reach level h, and W_h is the total number of valid pairs at that level. Similar to Theorem 7.4.1, $A^*(\mathcal{E}) \geq \max_{1 \leq h \leq k}(S_h - 1)2^{h-1}$.

Similarly, let $A_h(\mathcal{E})$ be the total communication cost of MultiBend for all the requests in $\mathcal{E}$ that reach level h in the hierarchy $\mathcal{Z}$, while probing the shared object in their up phase. Similar to Theorem 7.4.1, we have that $A_h(\mathcal{E}) \leq (S_h - 1)c_1 d 2^h$ (using Lemma 7.5.1), where c_1 is some constant. Therefore, by combining the cost for each level, $A(\mathcal{E}) = \sum_{h=1}^{k} A_h(\mathcal{E}) \leq \sum_{h=1}^{k}(S_h - 1)c_1 d 2^h$, in the worst-case.

Since the execution $\mathcal{E}$ is arbitrary and $\sum_{h=1}^{k}(S_h - 1)c_1 d 2^h \leq k \cdot \max_{1 \leq h \leq k}(S_h - 1)c_1 d 2^h$, the *move* stretch is bounded by

$$\max_{\mathcal{E}} \frac{A(\mathcal{E})}{A^*(\mathcal{E})} \leq \frac{k \cdot \max_{1 \leq h \leq k}(S_h - 1)c_1 d 2^h}{\max_{1 \leq h \leq k}(S_h - 1)2^{h-1}} \leq c_1 \cdot d \cdot k = \mathcal{O}(d \log n),$$

as $k = \lceil \log n \rceil + 1$ and c_1 is a constant. □

Theorem 7.5.5 (congestion for d-dimensional mesh) *In d-dimensional mesh networks,* MultiBend *achieves $\mathcal{O}(d^2 \log n)$ approximation on congestion with high probability.*

Proof. Similar to Theorem 7.4.3, consider the formation of a sub-path p_i from a sub-mesh M_1 to a sub-mesh M_2, where p_i is formed by following the shortest path from a randomly chosen node v_1 in M_1 to a randomly chosen node v_2 in M_2. This path is a two-bend dimension-by-dimension path with random ordering of dimensions. Assume also that M_1 is of type-1 and e be an edge of M_2. It has been proven in Busch *et al.* [28] that the sub-path p_i uses edge e with probability at most $\frac{2}{da^{d-1}}$,

where a is a side length of the sub-mesh M_1. Let P' be the set of paths that go from M_1 to M_2, or vice versa. Let $C'(e)$ be the congestion that the messages P' cause on e. Similar to Theorem 7.4.3, $E[C'(e)] \leq \frac{2|P'|}{da^{d-1}}$.

As $B \geq B(M_1, \Pi) \geq |P'|/\text{out}(M_1)$, we have that $C^* \geq |P'|/\text{out}(M_1)$. Since each side of M_1 has a nodes, $\text{out}(M_1) \leq 2da^{d-1}$. Therefore, $E[C'(e)] \leq 4C^*$. We charge this congestion to sub-mesh M_2. As there are at most $k < \lceil \log n \rceil + 1$ levels and at each level there are $\mathcal{O}(d)$ different types of sub-meshes, the expected congestion on edge e, denoted as $E[C(e)]$, is bounded by $E[C(e)] \leq \mathcal{O}(dC^*(\lceil \log n \rceil + 1))$.

According to our construction, there are $\mathcal{O}(d)$ different type sub-meshes M_1' between every two type-1 sub-meshes M_1 and M_2 in the sub-mesh hierarchy $\mathcal{Z}$. As $\mathcal{O}(d)$ different types sub-meshes M_1' may not be the proper subset of M_2, the set of paths from M_1 to one of the M_1' may go through $\mathcal{O}(d)$ different types of sub-meshes. This will increase the congestion by at most an another factor of $\mathcal{O}(d)$ between every two type-1 sub-meshes M_1 and M_2. Therefore, $E[C(e)] \leq \mathcal{O}(d^2C^*(\lceil \log n \rceil + 1))$.

As every request selects its path independently of every other request, similar to the two-dimensional case given in Theorem 7.4.3, we get a concentration result on the congestion C such that $C = \mathcal{O}(C^* d^2 \log n)$ with high probability, applying the standard Chernoff bound (Lemma 7.4.2), and using the fact that $|E| = \mathcal{O}(d \cdot n)$ and $d = \mathcal{O}(n)$. □

7.6 Experimental Results

Motivated from the nice theoretical properties of MultiBend in controlling both distance stretch and congestion, we now aim to investigate how these properties translate in practice through extensive simulations. We perform our simulations in a 16×16 nodes 2-dimensional mesh network, unless otherwise stated; we defer evaluations under a very general case of d-dimensional mesh networks for future work. The results are analyzed for a shared object and also for multiple shared objects in the mesh, and the operations (*publish*, *lookup*, and *move*) are performed for that object/those objects. The object operations are generated uniformly at random, that is, any

bottom-level node which issues a request is selected randomly among the nodes of the mesh. We implement several variants of MultiBend and two prior DDPs Arrow and Ballistic as described in Section 7.6.1. We assume a non-overlapping execution of various sequences of *move* and *lookup* operations. We initialize the directory hierarchy for each object (before serving any *lookup* and *move* operation through it) by creating a downward path from the root to a bottom-level node, which serves as an initial directory path for future operations on that object. This initialization process is the *publish* operation for that object.

The communication cost of the protocols for a set of operations is measured with respect to the total number of hops the set of operations traverses in the mesh, following the protocols, to reach the predecessor nodes that own the objects they requested. The optimal communication cost for the protocols for a set of operations is measured through the total number of hops that the operations need to traverse to reach their predecessor nodes following a Manhattan path in the mesh (assuming that they know their predecessor nodes). We do not consider the cost involved in sending the object, after it is found, from the predecessor node to the requesting node. If we consider this cost, the results we give increase at most by a factor of 2 only. For the congestion, we count the number of times any edge in the mesh is used by the set of operations, which we refer by *load per edge*.

For simplicity, we number the nodes of the 16×16 mesh by 0 to 255. We assume that the nodes with number from 0 to 15 are placed in the first row in the increasing order. Similarly, the nodes with number from number 16 to 31 are placed in the second row in the increasing order, and so on. We represent all 480 edges of the 16×16 mesh as $\{(e_{0,1}, e_{1,2}, \cdots, e_{14,15}), (e_{0,16}, e_{1,17}, \cdots, e_{15,31}), (e_{16,17}, e_{17,18}, \cdots, e_{30,31}), (e_{16,32}, e_{17,33}, \cdots, e_{31,47})$ $, \cdots, (e_{240,241}, e_{241,242}, \cdots, e_{254,255})\}$ connecting each of their neighbors (at most 4 and at least 2). We order the edges by first considering edges connecting subsequent nodes in the horizontal direction of the first row (i.e., horizontal edges), then considering edges connecting the nodes in the first row to the nodes in the second row (i.e., vertical edges), and so on. Moreover, we refer them by the integer numbers from 1 to 480 according to the order such that 1 denotes the edge

$e_{0,1}$, 2 denotes the edge $e_{1,2}$, and so on. We extend these notions to other mesh sizes appropriately, whenever required.

7.6.1 Protocol Variants Used in Experiments

The following variants of MultiBend are used for the simulations involving a single shared object.

- MultiBend: is the same protocol described in Section 7.3 which performs leader change every time an operation visits a leader. However, it does not take into account the cost due to the leader election procedure (Algorithm 11).
- MultiBend-Leader: is a variant of MultiBend which considers the extra cost incurred due to the leader election procedure along with the actual cost due to *move* and *lookup* operations. It also performs leader change at each operation.
- MultiBend-Leader(32): is a variant of MultiBend-Leader in which the leader election procedure is called after every 32 operations.
- MultiBend-Leader(1024): is a variant of MultiBend-Leader in which the leader election procedure is called after every 1024 operations.
- MultiBend-Static: is a variant of MultiBend where leaders are assigned to the sub-meshes of the directory hierarchy uniformly at random at the start of the directory construction and no further leader change occurs. Thus, this protocol does not incur extra leader election cost. Moreover, this protocol uses fixed paths to connect subsequent sub-mesh leaders for every object operation. Note that MultiBend chooses independently the multi-bend paths for each object operation.
- MultiBend-One: is a variant of MultiBend in which the path construction between every two sub-meshes is done by using just one-bend shortest paths. Note that MultiBend may sometime use two-bend shortest paths. It has leader election cost.

The following variants of MultiBend are used for the simulations involving multiple objects. All these protocols do not have leader election cost as the leaders are fixed all the time.

- MultiBend-Static-First: is a variant of MultiBend-Static which we use for supporting multiple objects. We create a directory for each different object. Each object is assumed to be identified with a unique integer between 0 to $\omega - 1$, where ω is the total number of objects in the network. In the directory $\mathcal{Z}_s$ for an object s, the leader for each sub-mesh $M_i \in \mathcal{Z}_s$ is assigned, at the start of the directory construction, in such a way that the leader for that sub-mesh is the node at position $s \mod \nu$ in the order of the nodes that are inside M_i, where ν is the number of nodes in M_i (the order of the nodes is provided in a fixed row-major order inside the sub-mesh in our evaluation). Moreover, this protocol uses fixed paths to connect two subsequent sub-mesh leaders for each object operation.

- MultiBend-Static-Last: is a variant of MultiBend-Static in which, for each sub-mesh M_i, the leader for that sub-mesh is the node at position $\nu - (s \mod \nu)$ in the node order, where ν is the number of nodes inside M_i. This protocol also uses fixed paths. The motivation behind considering this variant for comparison is that it may provide different congestion trade-off than MultiBend-Static-First as requests sometime need to traverse many edges to reach the next sub-mesh leader from the leader of the current sub-mesh due to its hierarchy construction.

- MultiBend-Static-Random: is a variant of MultiBend-Static that we use for supporting multiple objects. In this variant, the leaders of the directory for each object are assigned uniformly at random at the time of directory construction. This protocol also uses fixed paths.

- MultiBend-Static-First-Two: is a variant of MultiBend-Static-First in which the path construction between every two sub-meshes is done independently for each object operation by picking a dimension-by-dimension shortest path connecting the leaders similar to MultiBend. Note that three aforementioned variants use fixed paths for each object operation.

The intuition behind using this variant in experiments is to see how the protocols that work based on static leaders perform if we use paths used by MultiBend instead of fixed paths.

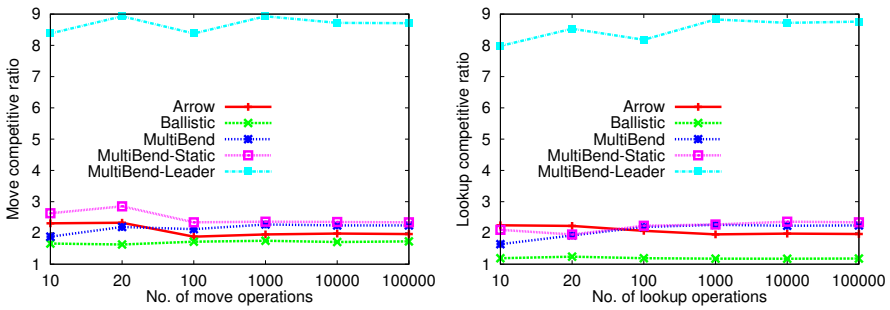
The performance of aforementioned MultiBend variants is also compared through simulations with the following two prior DDPs. Note that these protocols do not control congestion.

- Arrow [43]: operates on a fixed spanning tree, where every node holds a pointer to one of its neighbors in the tree, indicating the direction towards the node that owns the object. The final node in the path formed by the trail of pointers indicates the location of the owner node that is either holding the object or going to hold the object soon. (In a non-overlapping execution, the owner node already holds the object, so the requests do not need to wait for the object to arrive.) The object requests change the direction of the pointers to point to the new location so that future object requests will also be served efficiently. In other words, it maintains a distributed queue through *path reversal* [102].

- Ballistic [77]: is a location-aware consistency algorithm. Similar to MultiBend, this protocol is *hierarchical*: nodes are organized as clusters at different levels, where clusters at every level are built upon maximal independent sets of leader nodes of the clusters in the previous level. In this protocol, object requests are synchronized by path reversal similar to Arrow: when two requests meet at some intermediate node, the second request is diverted behind the first request. (A similar property holds also for our protocol.) Note that Ballistic uses *move-parent* and *lookup-parent* sets for searching the downward pointers at the parent level from the current level (details in [77]). Ballistic does not take into account the parent set probing cost. Thus, we define a variant Ballistic-Probing which takes into account the probing cost.

7.6.2 Single Object Results

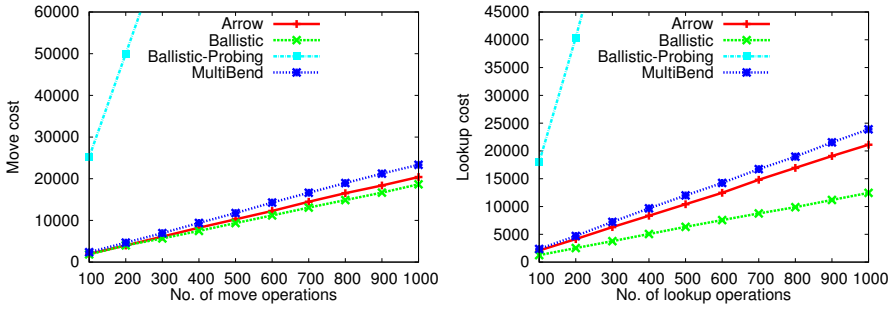
We start with the distance stretch results and later present the congestion results. The cost comparison of MultiBend variants (namely MultiBend, MultiBend-Static, and MultiBend-Leader) with Arrow and Ballistic for up to 100,000 *move* and *lookup* operations is given in Fig. 7.4. For the



(a) 100,000 move operations (b) 100,000 lookup operations

Figure 7.4: The stretch comparison of MultiBend variants and prior DDPs

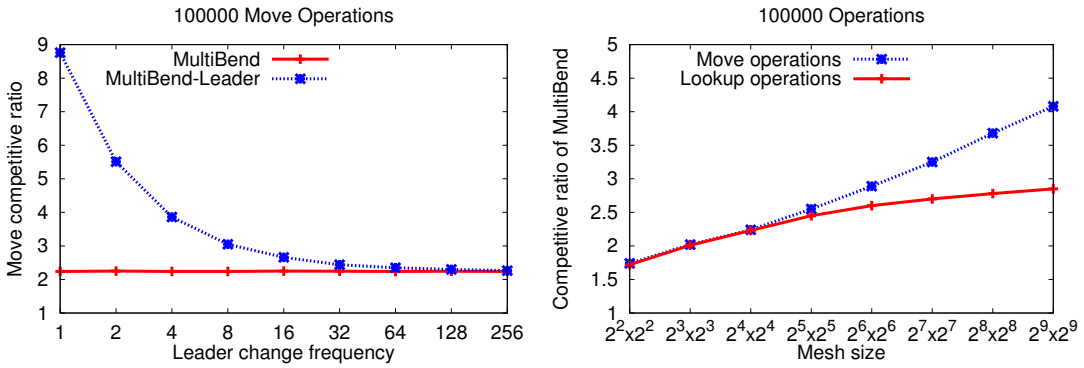
.



(a) 1000 move operations (b) 1000 lookup operations

Figure 7.5: The cost comparison for up to 1000 move and lookup operations

move operations, as shown in Fig. 7.4a, the performance of MultiBend and MultiBend-Static is comparable to the performance of Arrow and Ballistic. More specifically, the performance of Arrow and Ballistic is slightly better than MultiBend variants in most of the cases. Ballistic is better as we did not consider the move-parent set probing cost of it. As 11 leaders in the move-parent set are consulted on average by a leader at each level in our simulations, Ballistic-Probing, which takes into account this probing cost, performs much worse (see Fig. 7.5a for the comparison of the costs of Ballistic and Ballistic-Probing). The reason Arrow performs better is due to nice neighbor growth and connection properties of the network topology we used for evaluation, which facilitates Arrow to follow comparatively shorter paths than MultiBend. Even after taking into account the leader election cost in MultiBend-Leader, its *move* and *lookup* competitive ratios increase by the factor of approximately 4 only. The benefit is that MultiBend-Leader significantly minimizes the load per edge compared to Arrow and Ballistic.



(a) leader change frequency (b) mesh size change

Figure 7.6: The impact of leader change frequency and mesh sizes in the performance competitive ratio of MultiBend variants for 100,000 operations

For the *lookup* operations, the performance gap of MultiBend variants in comparison to Arrow and Ballistic is lower than the performance gap in *move* operations (Fig. 7.4b). However, Ballistic-Probing, which takes into account also the probing cost performs significantly worse. We consider only Ballistic for further comparisons due to its cost and congestion similar to compared protocols; note that probing cost is essential in Ballistic and this probing cost makes Ballistic less suitable in practice scenarios. On the average 15 leaders in the lookup-parent sets were consulted in Ballistic by a leader at each level in our simulations. These results can be seen in Fig. 7.5b.

We saw in Fig. 7.4 that in comparison to prior DDPs, the actual performance competitive ratio (and also the cost) of MultiBend for *move* and *lookup* operations is approximately a factor of 4 times more. This is due to the fact that the leaders are changed every time MultiBend visits them while serving the *move* or *lookup* operations. Therefore, we were interested to study how the distance competitive ratio changes if we minimize the leader change frequency. Fig. 7.6a shows the distance stretch results of that study. The results suggest that the impact of the leader election cost in the performance competitive ratio decreases significantly with the increase in the leader change frequency. As we can see in the figure, the impact of leader election cost decreases by more than a factor of 2 when new leaders are elected after every 4 *move* operations. The impact of leader election cost becomes negligible if we elect new leaders after every 16 operations or more. We achieved similar results for the performance competitive ratio of *lookup* operations.

Figure 7.7: The load comparison of Arrow, Ballistic, and MultiBend for 100,000 *move* operations (the worst load per edge: Arrow 50,015 at edge 172, Ballistic 31,997 at edge 434, and MultiBend 7297 at edge 8)

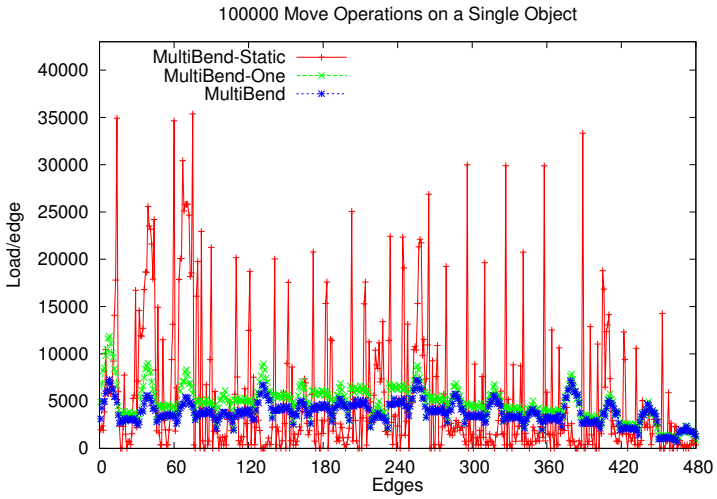


Figure 7.8: The load comparison of MultiBend variants for 100,000 *move* operations (the worst load per edge: MultiBend-Static 35,369 at edge 172, MultiBend-One 11,858 at edge 8, and MultiBend 7297 at edge 8)

Moreover, we study how the performance competitive ratio changes with the change in the mesh network size for the fixed number of *move* and *lookup* operations. Our simulation results show that the competitive ratio increase is within a logarithmic factor of the increase in the side length of the mesh. As shown in Fig. 7.6b, the performance competitive ratio for 100,000 *move*

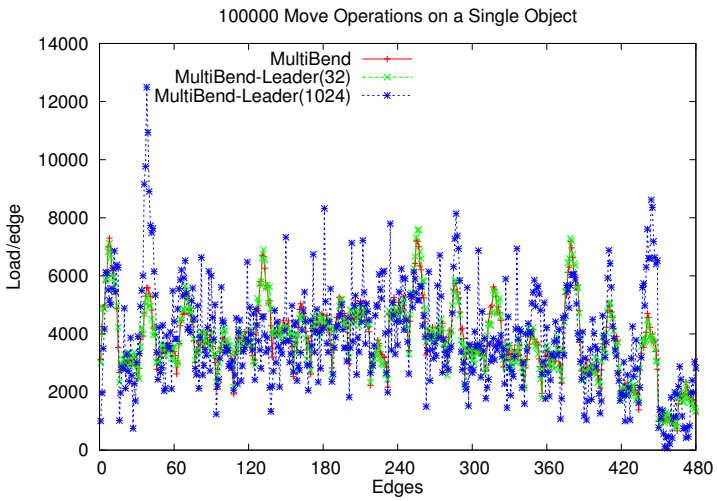


Figure 7.9: The comparison of MultiBend variants for the load per edge due to leader change frequency (the worst load per edge: MultiBend-Leader(32) 7590 at edge 257, MultiBend-Leader(1024) 12,495 at edge 38, and MultiBend 7297 at edge 8)

operations is 2.05 in a 8×8 mesh, whereas the ratio is 4.20 in a 512×512 mesh. For *lookup* operations, due the efficient use of special-parent nodes in finding the shortcuts to the downward paths, the performance competitive ratio stabilizes to a constant when mesh size gets larger. The results in Fig. 7.6b depict that the *lookup* competitive ratio is within a factor of 3 for the 512×512 nodes mesh network.

We now provide the congestion results. Fig. 7.7 shows the load per edge comparison of Arrow, Ballistic, and MultiBend for the 100,000 randomly generated *move* operations. As Arrow uses a pre-selected minimum cost spanning tree as a directory hierarchy, only limited number of edges (out of 480 edges) are used many times while the rest of the edges (more specifically, the edges that do not constitute to the spanning tree) are not used at all for load distribution. That property can be confirmed from the results of Fig. 7.7 where some edges are used thousands of times, while other are not used at all. Similarly, in comparison to Arrow results, Ballistic distributes load to more edges, however it also suffers from the limitation that some edges are used significantly many times than the least used edges. MultiBend tries to use all the edges in the mesh network uniformly thanks to the independently selected controlled dimension-by-dimension paths for each operation that connect two subsequent sub-mesh leaders. Similar congestion results were achieved for the

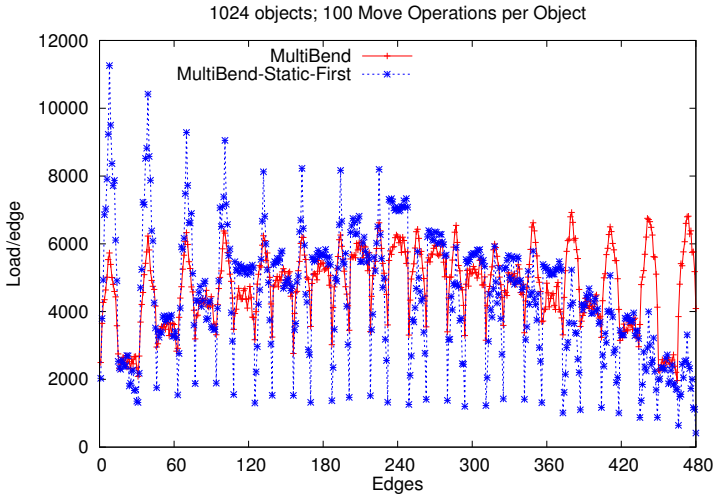


Figure 7.10: The load comparison of MultiBend variants for 1024 objects with 100 *move* operations per object (the worst load per edge: MultiBend-Static-First 11,258 at edge 8 and MultiBend 6926 at edge 380)

lookup operations; we omit *lookup* results as they show similar behavior as of *move* operations in all our experimental settings.

The load per edge comparison for the MultiBend variants for 100,000 *move* operations is given in Fig. 7.8. The results show that MultiBend minimizes the load per edge drastically. Moreover, despite the similar stretch performance, MultiBend-One performs worse in congestion control compared to MultiBend. This is due to the use of only one-bend paths in MultiBend-One which can not control the congestion on edges as efficiently as original paths used by MultiBend. We now study the effect of leader change frequency in the load at each edge of the mesh in Fig. 7.9 for 100,000 *move* operations. The load increase per edge in MultiBend-Leader(32) in comparison to MultiBend is not that significant, while the load increase in MultiBend-Leader(1024) is very significant. If we increase the leader change frequency, the load per edge converges toward the load per edge performance of MultiBend-Static given in Fig. 7.8.

7.6.3 Multiple Objects Results

The aforementioned simulation results suggest that MultiBend provides strong congestion control for a single shared object. We now study the distance stretch properties and the load balancing

benefits of MultiBend for supporting multiple objects. The simulations involving the distance stretch in supporting multiple objects showed similar results as of Fig. 7.4; hence, we only consider congestion properties here.

We construct a hierarchy directory for each shared object to support multiple objects. We focus our experiments on evaluating whether MultiBend still benefits from the leader election procedure when there are a large number of objects, e.g., greater than the number of nodes in a sub-mesh. It seems in the first sight that when there are large number of objects the leader election procedure is not needed. This is because in the long term every node will become a leader of a directory hierarchy which helps in minimizing the congestion without the use of a leader election subroutine, avoiding the extra cost. Therefore, if we let each directory use a different node when assigning a leader at the beginning of directory construction, it should alleviate the need of a leader election procedure.

Our simulation results show that the load is more balanced without the need of leader election procedure in the multiple objects scenario in comparison to the single object scenario (compare MultiBend-Static results of Fig. 7.8 with MultiBend-Static-First results of Fig. 7.10). However, this static assignment of different nodes as leaders in the multiple objects scenario may not always guarantee low load per edge similar to MultiBend (we will discuss a comparatively bad example in Fig. 7.13). Therefore, our approach is proven to be useful in controlling congestion at all times in the case of multiple objects as well.

Recall that the congestion benefit obtained using MultiBend is not only because of the frequent random leader election but also because of the use of independently selected controlled multi-bend paths for each object operation to connect two subsequent sub-mesh leaders in the hierarchy. These controlled paths change dimensions independently for each operation, starting first in the horizontal or vertical direction, and may follow different set of nodes every time we route requests between two different leaders. Fig. 7.10 compares the congestion obtained while running MultiBend and MultiBend-Static-First for 1024 objects with 100 *move* operations per object. The load per edge performance of MultiBend-Static-First is almost the factor of 2 worse in comparison to the

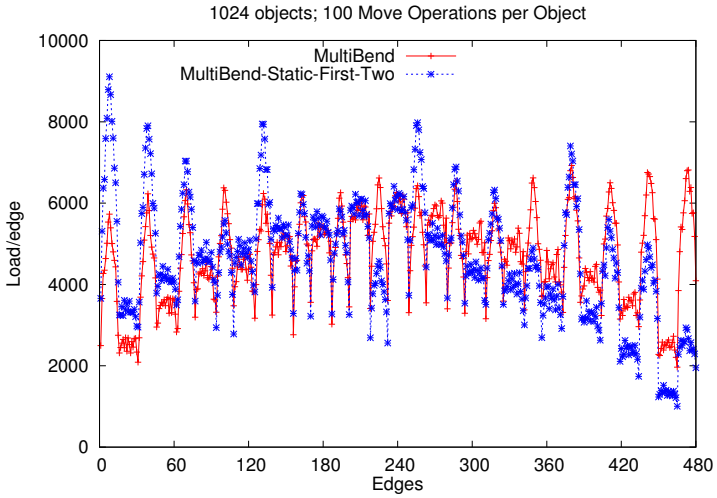


Figure 7.11: The load comparison of MultiBend variants for 1024 objects with 100 *move* operations per object (the worst load per edge: MultiBend-Static-First-Two 9108 at edge 8 and MultiBend 6926 at edge 380)

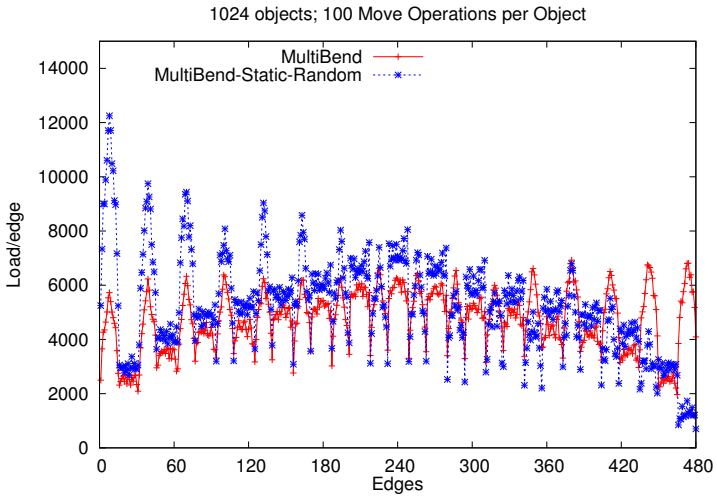


Figure 7.12: The load comparison of MultiBend variants for 1024 objects with 100 *move* operations per object (the worst load per edge: MultiBend-Static-Random 12,248 at edge 8 and MultiBend 6926 at edge 380)

performance of MultiBend. The trade-off is that MultiBend-Static-First obtains that load performance without the use of the leader election procedure. Moreover, the congestion performance of MultiBend-Static-First for multiple objects is better by almost a factor of 3 in comparison to the performance of MultiBend-Static given in Fig. 7.8 for a single shared object.

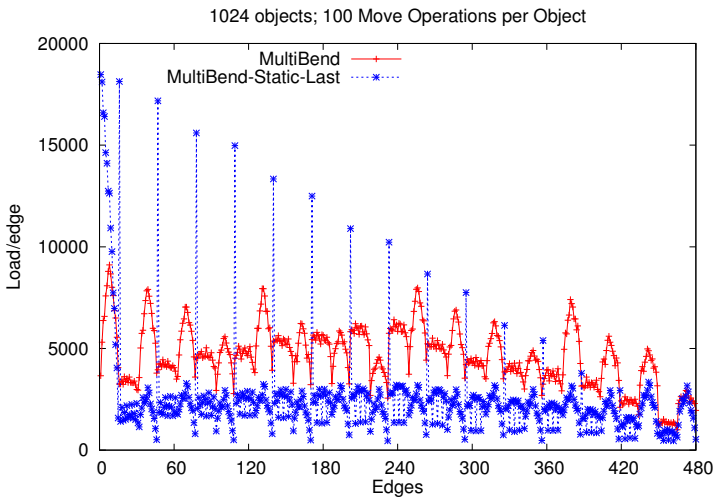


Figure 7.13: The load comparison of MultiBend variants for 1024 objects with 100 *move* operations per object: a comparatively bad example (the worst load per edge: MultiBend-Static-Last 18,470 at edge 1 and MultiBend 6926 at edge 380)

Note also that MultiBend-Static-First does not use independently selected dimension-by-dimension shortest paths to connect the subsequent sub-meshes leaders while serving the operations, but still achieves the performance that is only 2 times worse compared to the performance of MultiBend. Therefore, we were interested to see how these independently selected controlled paths impact the congestion performance of MultiBend-Static-First. Fig. 7.11 shows the impact of such paths in congestion. MultiBend-Static-First-Two which uses such independently selected controlled dimension-by-dimension shortest paths for each object operation shows the improvement in congestion in comparison to MultiBend-Static-First by a factor of 1.23 (compare the congestion results of MultiBend-Static-First in Fig. 7.10 and MultiBend-Static-First-Two in Fig. 7.11). That is, the improvement on the worst load at any edge is 2150 (the different between the blue peaks in the figures). The performance of MultiBend variant MultiBend-Static-Random for the same execution setting is given in Fig. 7.12.

We now discuss a comparatively bad example. We used MultiBend-Static-Last in which the leader assignment in the beginning of the directory construction follows reverse node order in the sub-mesh. That is, if there are 32 nodes in a sub-mesh, for the object numbered 33 the leader node in that sub-mesh is $32 - (33 \mod 32) = 31$. For object number 31, the leader will be 1 for

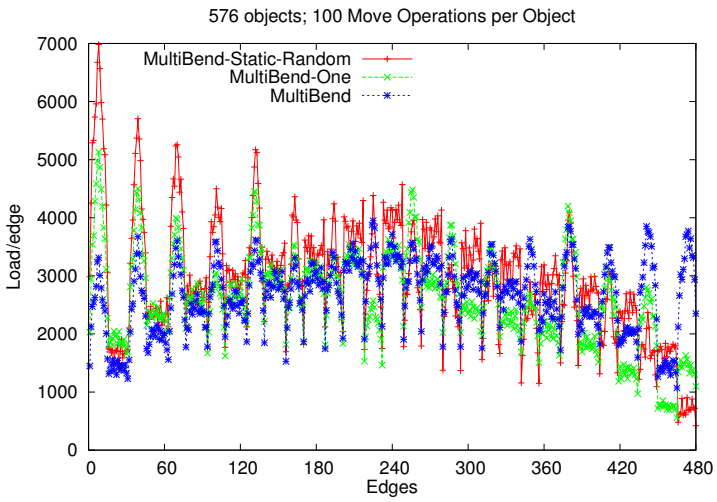


Figure 7.14: The load comparison of MultiBend variants for 576 objects with 100 operations per object (the worst load per edge: MultiBend-Static-Random 6983 at edge 8, MultiBend-One 5134 at edge 8, and MultiBend 3950 at edge 225)

that mesh. If we compare the worst congestion at any edge in this static assignment of leaders, as shown in Fig. 7.13, MultiBend-Static-Last performs worse in comparison to MultiBend-Static-First-Two by a factor of 2.03, MultiBend-Static-First by a factor of 1.64, and MultiBend-Static-Random by a factor of 1.51. This is due to the overutilization of some of the edges by MultiBend-Static-Last while serving object requests.

We also compare the performance of MultiBend variants when there are uneven number of objects in each node of the mesh. For example, if there are 576 objects, 3/4-th fraction of the total nodes in the network get 2 objects each and 1/4-th fraction of the network nodes get 3 objects each, following a uniform distribution starting from the first node of the 16×16 mesh. In this setting, as depicted in Fig. 7.14, MultiBend still performs significantly better in controlling congestion thanks to the combination of random leader election procedure and independently selected dimension-by-dimension paths. Moreover, MultiBend-One performs slightly worse than MultiBend due to the use of only one-bend paths, while MultiBend-Static-Random's performance is within a factor of 2 of MultiBend.

To summarize our findings, the simulation results on multiple objects suggest that the leader election approach allows us to achieve low congestion compared to static variants. However, con-

sidering the extra cost due to leader election in MultiBend, the trade-off between stretch and congestion can be made depending on applications while supporting multiple objects. Nevertheless, in applications where there are small number of objects, the leader election technique used in MultiBend is indeed a true facilitator for congestion control.

7.7 Summary and Discussions

In this chapter, we presented and analyzed a novel load balanced directory-based consistency algorithm, called MultiBend, for shared objects, that is suitable for d-dimensional mesh networks. We also evaluated MultiBend for its distance stretch and congestion benefits considering several sequences of *move* and *lookup* operations on a single shared object and multiple shared objects. The evaluation results confirm the theoretical and practical benefits of MultiBend.

As there are (at most) $2d$ neighbors for any node in d-dimensional mesh networks, MultiBend guarantees $\mathcal{O}(d^3 \log n)$ approximation of the optimal congestion on any node. This is because a node can be used by any of the $2d$ edges, therefore increasing the edge congestion by a $\mathcal{O}(d)$ factor. This node congestion approximation is also optimal within a constant factor for fixed d.

For the future work, we plan to explore stretch and congestion bounds of MultiBend for the case of d-dimensional mesh networks with uneven dimensions. Moreover, we plan to extend MultiBend for dynamic networks where nodes enter and leave at any time and make it fault-tolerant.

Chapter 8
Distributed and NUMA Systems: Time and Communication Trade-offs

8.1 Introduction

In this chapter, we evaluate the performance of the execution schedules in data-flow distributed implementations of transactional memory in distributed and NUMA systems using the total communication cost for moving objects to transactions, and the execution time for completing all transactions. We give bounds and trade-offs for the communication cost and execution time. In other words, we give a scheduling problem instance where execution time and communication cost cannot be simultaneously optimized. Minimizing execution time implies high communication cost and vice versa. Therefore, there is no single algorithm that can optimize both parameters simultaneously. We also give several hardness results for both the communication cost and execution time. To the best of our knowledge this is the first comprehensive study of performance bounds for distributed transactional memory schedules with multiple objects per transactions.

8.1.1 Contributions

We give a comprehensive set of bounds for scheduling problem instances where transactions use multiple objects. We first give near optimal upper bounds for the communication cost. We then explore the execution time and non-trivial lower bounds, and also provide upper bounds. Finally, we explore trade-offs between communication cost and execution time.

Communication cost: We first observe that the problem of minimizing the communication cost is NP-hard with a reduction from the graph TSP problem. A TSP instance is transformed to a transaction scheduling problem by having each city node represented with a transaction, where all transactions share a single object. Note that the hardness holds even with a single shared object.

We then continue with upper bounds for communication cost. We use a universal TSP tour to schedule the transactions. A universal TSP tour [83] defines a traversal order to visit all nodes in the graph so that any subsequence of nodes is also an approximate TSP tour for the respective subset of nodes. By executing the transactions in order according to the TSP tour we guarantee that each object follows approximately a TSP tour. The overall schedule has communication cost within $\mathcal{O}(\log^4 n/\log\log n)$ factor from optimal, where n is the number of nodes of the network.

Execution time: Optimization of execution time is an NP-hard problem (reduction from vertex-coloring), and also hard to approximate within any factor smaller than the number of nodes. We give an $\Delta+1$ approximation algorithm for the execution time, where Δ is the maximum number of other transactions that a transaction conflicts with. This bound is obtained with a greedy coloring of a weighted conflict graph where nodes are transactions and a conflict among transactions is represented with an edge with weight to distance between the transactions.

We then explore lower bounds. During execution each transaction follows a walk that visits all the objects that request it (the TSP tour length of an object is no more than twice the shortest walk length). A trivial lower bound on the execution time is the longest of any object shortest walk. A interesting question is whether there are efficient schedules when the shortest walks are small (and other parameter are also low, such as conflicts number and objects per transactions). We answer this question to the negative, namely, where each shortest object walk has length $\mathcal{O}(n^{5/6})$, while any execution schedule requires time $\Omega(n)$. The same instance has $\mathcal{O}(\log n)$ objects per transaction and $\Delta = \mathcal{O}(n^{2/3})$; thus, the $\Omega(n)$ execution time does not follow directly from these parameters. This problem instance demonstrates a significant asymptotic gap between the objects' walks and the execution time.

Time and communication trade-offs: We give a problem instance where it is impossible to simultaneously optimize execution time and communication cost. In this problem instance the execution time is $\Omega(n^{2/3})$ and the communication cost is $\Omega(n)$. We give a schedule with optimal execution time $\mathcal{O}(n^{2/3})$. However, we show that any schedule with optimal execution time must have suboptimal communication cost $\Omega(n^{4/3})$. We obtain a symmetric impossibility result with respect to optimal communication cost. That is, we give a schedule with optimal communication cost $\mathcal{O}(n)$. However, we show that any schedule with optimal communication cost must have suboptimal execution time $\Omega(n)$.

These impossibility results imply that we cannot have a single algorithm that minimizes both execution time and communication cost simultaneously, which justifies the independent study of these two optimization problems.

8.1.2 Chapter Organization

The rest of the chapter is organized as follows. In Section 8.2 we provide hardness results and an upper bound for communication cost. In Section 8.3 we examine the execution time for which we give hardness results, and upper and lower bounds. Execution time and communication trade-offs are presented in Section 8.4. We conclude in Section 8.5 with a short discussion.

8.2 Communication Cost Bounds

The problem of minimizing the communication cost is NP-hard, by a reduction from the graph TSP (traveling salesperson problem) which is an NP-complete problem. Any graph TSP instance can be directly converted to a transactional scheduling problem instance on the same graph such that each node has a transaction and there is a single object. The transactional memory schedule has optimal communication cost if and only if the respective TSP tour cost is optimal.

Given a graph G, we can approximate the optimal communication using a *universal TSP tour*. Jia *et al.* [83], prove that there exists a universal TSP tour Q that traverses all the nodes in G so

that for any subset of nodes S the induced sub-tour in Q approximates the optimal tour for S (in the induced subgraph) within a factor of $\mathcal{O}(\log^4 n/\log\log n)$.

We can use the universal tour Q to construct a schedule for the objects as follows. We execute the transactions in sequence according to the order that they appear in Q. Once a transaction finishes execution, it passes each object to the next transaction that requires the object according to the order in Q. We refer to this as the *universal TSP schedule*. We can prove the following result.

Theorem 8.2.1 (Communication Cost Upper Bound) *The universal TSP schedule guarantees communication cost within $\mathcal{O}(\log^4 n/\log\log n)$ factor from optimal.*

Proof. Let $\mathcal{T}_{r_i} \subseteq \mathcal{T}$ denote the set of transactions that request object $r_i \in R$. Let G_{r_i} denote the induced subgraph of G consisting only of the nodes where the transactions in $\mathcal{T}_{r_i}$ reside. Let C_{r_i} denote the total distance traversed by object r_i for visiting the nodes in G_{r_i} when following the universal TSP schedule. Let $C^*_{r_i}$ denote the optimal cost for traversing the nodes in G_{r_i}. The universal TSP schedule guarantees that $C_{r_i}/C^*_{r_i} \leq \lambda$, where $\lambda = \mathcal{O}(\log^4 n/\log\log n)$.

The total communication cost is equal to the sum of the individual costs for the objects, $C = \sum_{i=1}^{k} C_{r_i}$. The optimal cost is $C^* = \sum_{i=1}^{k} C^*_{r_i}$, since objects do not combine. Therefore,

$$C = \sum_{i=1}^{k} C_{r_i} \leq \sum_{i=1}^{k} \lambda \cdot C^*_{r_i} = \lambda \sum_{i=1}^{k} C^*_{r_i} = \lambda C^*,$$

which implies that $C/C^* = \mathcal{O}(\log^4 n/\log\log n)$, as needed. □

8.3 Execution Time Bounds

In this section we focus on finding schedules that minimize the execution time. We first prove that the problem is NP-hard. We then show that it is impossible to obtain execution time close to the shortest walk length of any object. We also give approximation algorithms.

8.3.1 Hardness for Execution Time

We will reduce the vertex coloring problem to this problem. Consider an arbitrary unweighted graph H. The coloring problem aims to find the chromatic number $\chi(H)$ which is the smallest number of distinct colors that can be assigned to the nodes of H so that no two adjacent nodes receive the same color (valid coloring). The vertex coloring problem is NP-hard.

We can transform in polynomial time any vertex coloring problem instance in an arbitrary graph H to a transaction scheduling problem in a graph G. We construct G to be isomorphic with H such that each edge in G has weight 1. Each node in G holds a transaction. For each edge $e = (u, v)$ in H we create an object in G to be used by the respective transactions in the adjacent nodes in G.

Given a transaction execution schedule in G, we can find a valid vertex coloring in H by simply converting the time that each transaction executes to a color (color value is equal to time value). It can be shown that an execution schedule in G has duration χ time steps if and only if H has a valid coloring with χ colors. Therefore, a schedule for the transactions in G is optimal if and only if the respective coloring in H is optimal. Consequently, the execution time optimization problem is NP-hard.

Note that the reduction above is 1-gap-preserving. Since, for all $\epsilon > 0$, approximating the chromatic number within a factor $n^{1-\epsilon}$ is NP-hard [147], approximating the optimal time within a factor $n^{1-\epsilon}$ is NP-hard too. Note that the inaproximate hardness result holds for transactional memory graphs where all edges have uniformly the same weight (weight 1 in the reduction). If a graph is not uniformly weighted then the reduction gap may change and it could be less hard to approximate the optimal value with a smaller factor.

8.3.2 Upper Bound for Execution Time

Here we give an approximation algorithm for the optimal time schedule. Consider a transaction scheduling problem instance in a graph G. Let Z denote the transaction conflict graph, such that each node in Z is a transaction and each edge represents a conflict between the adjacent

transactions, that is, the transactions share one or more objects. The graph H is weighted so that the weight on an edge is the distance between the respective transactions in G.

A valid coloring of Z with non negative integer colors guarantees that any two adjacent transactions will receive colors which differ at least at much as the weight of the edge connecting them. For any transaction T_i in Z let $\gamma(T_i)$ denote the *weighted-degree*, which is the sum of the weights of the edges adjacent to T_i. Let Γ denote the maximum weighted-degree in Z. We can color the conflict graph with a simple *greedy* algorithm, node by node, by assigning the first available color to each node, and we can obtain a coloring with $\Gamma + 1$ colors. We refer to the resulting schedule as the *greedy schedule*.

Let Δ denote the maximum (unweighted) node degree in Z, that is, Δ is the maximum number of adjacent nodes for any node in Z. The maximum edge weight, denoted $w_{\max}$, in Z is a lower bound for the schedule time, since it takes at least so much time to transfer a shared object in G from one transaction to another that it conflicts. Since $\Gamma \leq w_{\max}\Delta$, the algorithm above provides a $\Delta + 1$ approximation for the optimal execution time.

Theorem 8.3.1 (Execution Time Upper Bound) *The greedy schedule provides a $\Delta + 1$ approximation to the optimal time schedule.*

8.3.3 Lower Bound for Execution Time

The *shortest walk* of an object in G minimizes the total distance to visit all the transactions that require the object. Note that a TSP tour length of an object is no more than twice its shortest walk length. The maximum shortest walk of any object in G is a lower bound for the execution time. Here we give an instance on a graph G with n nodes, such that the shortest walk of any object is $\mathcal{O}(n^{5/6})$ (asymptotically smaller than n), and yet, the only possible schedule is almost sequential with execution time $\Omega(n)$, giving a significant gap between the walk lower bound and the execution time. This problem instance has small number of objects per transaction, $\mathcal{O}(\log n)$, and each transaction conflicts with $\Delta = \mathcal{O}(n^{2/3})$ other objects.

Consider a graph G that is a $s \times 2s^2$ grid consisting of s rows and $2s^2$ columns and the number of nodes is $n = 2s^3$. Each node is connected to four neighbor nodes (up, down, left, right) by an edge of weight 1; the nodes at the corner or sides are connected to two or three neighbors, respectively. Divide the grid into $s \times s$ consecutive and node-disjoint sub-grids $G_1, \ldots, G_{2s}$. Denote the odd subsequence of sub-grids as $H_1, \ldots, H_s$ (where H_z corresponds to G_{2z-1}).

Each node in H_z holds a transaction. There are s *internal objects* $o_1, \ldots, o_s$, such that object o_z is used by all the transactions in the zth sub-grid H_z. Initially, each o_z object resides in the top left corner node in its respective sub-grid.

There are $2s$ *external objects* $q_1, \ldots, q_{2s}$. In each sub-grid H_z, each object q_z will be used by a random set of s transactions. Initially, all external objects reside in the top-left corner node of H_1.

Lemma 8.3.2 *Each internal object has shortest walk length $\mathcal{O}(s^2) = \mathcal{O}(n^{2/3})$. Each external object has shortest walk length $\mathcal{O}(s^{5/2}) = \mathcal{O}(n^{5/6})$.*

Proof. Each internal object has shortest walk of length exactly $s^2 - 1$, since it can visit the nodes of the respective sub-grid row by row, in a zig-zag way.

Consider an external object q_i. The shortest walk of q_i within a sub-grid H_z may be of length $\mathcal{O}(s \cdot \sqrt{s})$, which corresponds to the worst case scenario where the s nodes that request q_i are spaced at distance $\sqrt{s}$ from each other in Q_z. By adding up all the walk lengths in the s sub-grids, the total walk length is $\mathcal{O}(s \cdot s \cdot \sqrt{s}) = \mathcal{O}(s^{5/2})$. □

There is a *sequential schedule* which executes the transactions in sequence one after the other starting from the transactions in H_1, then passing the objects to H_2 and executing the transactions in H_2, then passing the objects in H_3 and so on. In each H_z the transactions can execute in time $\mathcal{O}(s^2)$, and since there are s sub-grids we obtain the following result:

Lemma 8.3.3 *The sequential schedule executes all transactions in time $\mathcal{O}(s^3) = \mathcal{O}(n)$.*

We will show now that any execution schedule requires time which is asymptotically greater than s^3, and hence, the sequential schedule is optimal. In order to prove the central impossibility result we use the following lemma.

Lemma 8.3.4 *With high probability, in any specific sub-grid H_z, $1 \leq z \leq s$, any set of λ transactions uses at least $\lambda/2$ different external objects, where $1 \leq \lambda \leq s$.*

Theorem 8.3.5 (Lower Bound for Sparse Instance) *With high probability, there is a choice of external object transaction assignments, such that every execution schedule in G has duration $\Omega(s^3) = \Omega(n)$.*

Proof (sketch). Consider an arbitrary time window W of $s-1$ time steps. It suffices to prove that it is impossible to execute more than $4s+1$ transactions in W.

For the sake of contradiction suppose that $4s+2$ transactions or more execute during W. We divide the set of transactions which execute in W into sets $A_1, \ldots, A_s$, such that set A_z consists of all transactions which execute in sub-grid H_z. Clearly, $\sum_{z=1}^{s} |A_z| \geq 4s+2$.

We have any two pairs A_{z_1} and A_{z_2}, $z_1 \neq z_2$, cannot share any object, since the minimum distance in G between any two nodes in the respective sub-grids H_{z_1} and H_{z_2} is at least s, and since the duration of W is $s-1$ there is not enough time to transfer any object between the two sub-grids during time period W.

No more than $s-1$ transactions can execute in each A_z within period W, since all transactions in A_z share the internal object o_z in H_z. Therefore, $|A_z| \leq s-1$. From Lemma 8.3.4 each set A_z requires at least $|A_z|/2$ external objects (with high probability). Since the external objects from the different sets are disjoint, the total number of different external objects that are required is at least $\sum_{z=1}^{s} |A_z|/2 \geq (4s+2)/2 \geq 2s+1$. This is a contradiction since there are $2s$ external objects in total. □

8.4 Time and Communication Trade-offs

We show that there is a scheduling problem instance on a graph G in which the communication cost and execution time cannot be minimized simultaneously. In Section 8.4.1 we describe this problem instance. In Section 8.4.2 we provide a pipelined schedule, with small execution time, while in Section 8.4.3 we provide a sequential schedule with small communication cost. We generalize this

observation by proving that any schedule in G that attempts to optimize time must have suboptimal communication cost, and vice versa. This implies that there is no single algorithm that can optimize both execution time and communication cost simultaneously.

8.4.1 Problem Instance Description

For proving this result, we consider an grid graph $G = (V, E)$ with $n = a \times c$ nodes, which consists of a rows and c columns of nodes (Figure 8.1). Each node in G connects with an edge to four neighbors (up, down, left, right), except for the nodes at corners or borders which have two or three neighbors, respectively. Each edge has weight 1. Graph G consists of a sequence of k subgraphs $G_1, G_2, \ldots, G_k$, each of size of $a \times (a + 3b)$, where $1 \leq b \leq a$ and $k = c/(a + 3b)$, where the number of columns c is a multiple of $a + 3b$. Each subgraph G_i is further divided into four grid subgraphs A_i, B_i, C_i, D_i, such that A_i has size $a \times a$, and each of B_i, C_i, D_i has size $a \times b$.

In each G_i there is a transaction in every node of A_i, B_i, and D_i, while C_i does not have any transactions at all. There is a set of *global* objects $o_1, \ldots, o_a$ which are requested by a subset of transactions in every G_i, $1 \leq i \leq k$. Each object o_j is requested by all transactions residing in the jth row of G. For example, Figure 8.1 highlights the set of transactions in the 5th row (from the top) that use global object o_5. Each global object o_j initially resides in the leftmost column of A_1 and row j.

There are also transactions which are *internal* to each G_i, requested only by transactions inside G_i (and specifically in A_i, B_i and D_i). In A_i there is an object p_i requested by all the transactions of A_i. Initially, p_i resides in the top left corner of A_i. In B_i there is a set of b objects $Q_i = \{q_{i,1}, \ldots, q_{i,b}\}$, such that object $q_{i,j}$ is requested by all transactions in the jth column of B_i. Initially, object $q_{i,j}$ resides in the top node of the jth column of B_i. The same objects in Q_i are also requested by the transactions in D_i, so that object $q_{i,j}$ is requested by all transactions in the jth column of D_i. Therefore, each object $q_{i,j} \in Q_i$ is requested by the jth column transactions in both B_i and D_i. Figure 8.1 highlights the sets of transactions to use $q_{1,2}$ and $q_{2,2}$.

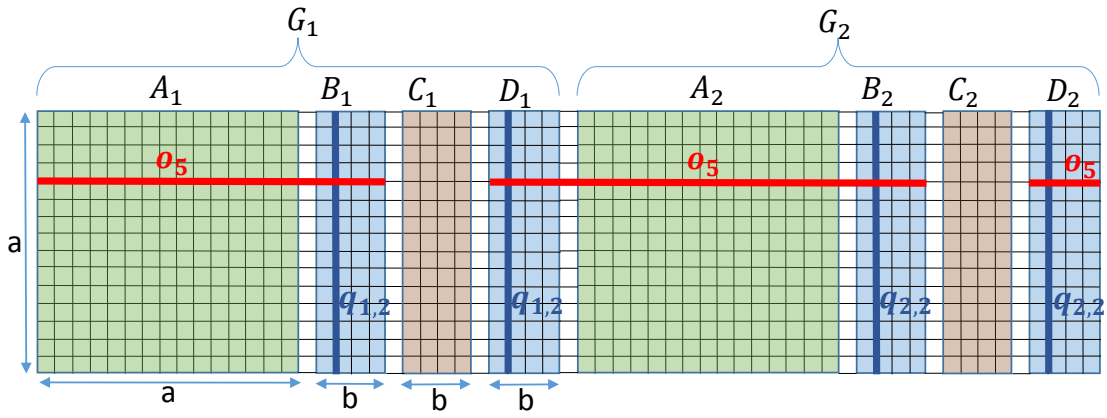


Figure 8.1: The graph G for the time-communication impossibility result, with $k = 2$, $a = 16$ and $b = 5$.

8.4.2 Fast Pipelined Schedule

We present a "pipelined" schedule where the global objects traverse the graph G by visiting $G_1, G_2, \ldots, G_k$ in a pipelined fashion. This execution has small execution time but high communication cost. Global object o_1 first traverses G_1 then G_2 and so on until it exits from G_k. At the time that o_1 enters G_2, object o_2 starts traversing G_1. Furthermore, when o_1 enters G_3, object o_2 enters G_2 and o_3 starts traversing G_1. This pipelined schedule continues until the last global object o_a exits from G_k. The pipeline has k stages, where stage i, $1 \leq i \leq k$, consists of the execution of transactions in G_i.

Lemma 8.4.1 *In the pipelined schedule, a global object o_j traverses all requested transactions in G_i in time $\mathcal{O}(a)$ with communication cost $\Omega(a + b^2)$.*

Proof. First we consider the execution time. We calculate the time that it takes for the global object o_j to traverse G_i. We start by showing that it takes $\mathcal{O}(a)$ time for global object o_j to traverse A_i. It takes at most a time steps for p_i to reach the jth row. Once o_j and p_i are together in the leftmost node of the jth row, it takes $2a$ time steps to traverse all transactions in the row. The reason is that the two objects o_j and p_i need to appear simultaneously in each node on the jth row, so that the respective transaction on that node executes. Since at most one object can traverse an edge at a time step at any given direction, and there is only one path with length one between two

adjacent nodes, it takes at least two time steps for the two objects to move from one node to an adjacent node in the jth row.

We then continue to show that it takes $\mathcal{O}(b)$ time for o_j to traverse B_i. While o_j was traversing A_i, all the objects in Q_i can move in parallel to the jth row of B_i. Thus, once o_j enters B_i on the jth row, it takes at most b time steps to traverse the row.

We can also show that it takes $\mathcal{O}(b)$ time for o_j to traverse C_i and D_i combined. The objects in Q_i and object o_j can move from B_i to D_i as a convoy, one object following the other along the jth row. In this way it takes $\mathcal{O}(b)$ time steps until all objects reach the respective transactions in the jth row of D_i, and o_j is positioned in the leftmost node of the jth row of D_i. It takes additional $\mathcal{O}(b)$ steps for q_j to move along the jth row of D_i to execute the transactions.

By combining all the above times we get that o_j traverses all the requested transactions in G_i in $\mathcal{O}(a + b) = \mathcal{O}(a)$ time, since $b \leq a$.

We continue now with the communication cost. We will add all the path lengths that are followed by the involved objects. The total cost for traversing A_i is $\Omega(a)$ since the paths followed by o_j and p_i have length $\Omega(a)$, since the row has width a. The cost for traversing B_i, C_i and D_i combined is $\Omega(b^2)$, since each object $q_{i,j} \in Q_i$ follows a path from B_i to D_i of length $\Omega(b)$, and $|Q_i| = b$. Therefore, the total communication cost is $\Omega(a + b^2)$. □

Lemma 8.4.2 *The pipelined schedule executes all transactions in time $\mathcal{O}(a(a + k))$ with communication cost $\Omega(ka(a + b^2))$.*

Proof. From Lemma 8.4.1, it takes $\mathcal{O}(a)$ time for a global object to traverse a subgraph G_i. Similar to the proof of Lemma 8.4.1, we can also show that the internal objects of G_i can be repositioned to their origins in time $\mathcal{O}(a)$, which is useful for the traversal of the next global object to traverse G_i. Thus, we have a pipeline with k stages for a global objects, where each stage takes $\mathcal{O}(a)$ time to complete. Therefore, the total time until the last object finishes traversal is $\mathcal{O}(a(a + k))$.

From Lemma 8.4.1, $\Omega(a + b^2)$ communication cost is necessary for a global object to traverse subgraph G_i. Since any G_i is visited by a global objects, and $1 \leq i \leq k$, the total communication cost of the pipelined execution is $\Omega(ka(a + b^2))$. □

8.4.3 Slow Sequential Schedule

We describe a "sequential" execution where all the global objects start in G_1, and then after all finish with G_1 they all move to G_2, and after all objects finish with G_2 they all move to G_3, and so on. Thus all the global objects visit together all subgraphs G_i sequentially. This schedule has high execution time but low communication cost (compared to the pipelined schedule). There are in total k rounds in this schedule, where in round i all the global objects appear only in G_i.

Lemma 8.4.3 *Each round has time duration $\Omega(a^2)$ and the communication cost $\mathcal{O}(a^2)$.*

Proof. Consider round i. We first give a execution time bound. Every transaction in G_i will execute during round i. It takes $\Omega(a^2)$ time to execute the transactions in A_i because the internal object p_i has to traverse a^2 transactions. It takes $\Omega(a)$ time to execute the transactions in B_i and D_i. This is because the objects in Q_i can move in parallel along their respective columns in B_i (which requires $\Omega(a)$ time). However, the objects in Q_i have to be skewed along their columns, in order to allow the global objects to execute in parallel too. The skewing is at most b positions between $q_{i,1}$ and $q_{i,b}$ along their respective columns, and any two $q_{i,j}$ and $q_{i,j+1}$ differ by one position, with $q_{i,j+1}$ positioned higher, $1 \leq j < b$. Once the global objects and the objects in Q_i finish with B_i they all move to D_i in $\Omega(b)$ time, such that the global objects move in parallel followed by the objects in O_i which move in convoy too. Then in D_i the transactions will execute in $\Omega(a)$ time as in B_i. Thus, $\Omega(a+b) = \Omega(a)$ time is required to complete the execution in B_i and D_i. Combining the above bounds, we have in total $\Omega(a^2 + a) = \Omega(a^2)$ time for round i.

For the communication cost we have to consider the path lengths traversed by all the objects. Each global object traverses a total distance of $a + 3b = \mathcal{O}(a)$, giving cost of $\mathcal{O}(a^2)$ for the a global objects. The internal object p_i traverses a path of length a^2. Each object in Q_i traverses a path of distance $\mathcal{O}(a+b) = \mathcal{O}(a)$, giving a cost of $\mathcal{O}(ab)$ for all objects in Q_i. Therefore, the total communication cost is $\mathcal{O}(a^2 + ab) = \mathcal{O}(a^2)$. □

Since there are k rounds, the following result follows immediately from Lemma 8.4.3.

Lemma 8.4.4 *The total execution time for the sequential schedule is $\Omega(ka^2)$ and the communication cost is $\mathcal{O}(ka^2)$.*

8.5 Summary and Discussions

In this chapter, we gave bounds and trade-offs for the communication cost and execution time for transaction scheduling in distributed and NUMA systems. We showed that there are scheduling instances where execution time and communication cost can not be simultaneously minimized; minimizing execution time implies high communication cost and vice versa. After that, we gave algorithms that only minimize communication cost or the execution time. For the future work, it is interesting to provide similar trade-offs for the execution time and congestion. This is a natural direction in the sense that communication cost and congestion can be minimized simultaneously in some particular topologies, e.g. d-dimensional meshes.

Chapter 9
Conclusions and Future Work

9.1 Overall Dissertation Summary

In this dissertation, we made following contributions for transaction scheduling in three different TM systems that provide three different communication cost models:

- **Tightly-coupled systems:** We proposed two models for transaction scheduling in tightly-coupled systems that extend the original one-shot transaction scheduling model and provide several trade-offs in the competitive ratio. Our execution window model (Chapter 3) provided scheduling algorithms with competitive ratio bounds that are within a poly-log factor of $\mathcal{O}(s)$; the scheduling algorithms for the one-shot scheduling model only obtained tight $\Theta(s)$ competitive ratio. Our balanced workload model (Chapter 4) provided scheduling algorithms with competitive ratio bounds that are sub-linear in s, which is a significant improvement compared to $\mathcal{O}(s)$ bound for the one-shot scheduling model, albeit with two minimalistic assumptions.

- **Distributed networked systems:** We presented a distributed consistency algorithm (Chapter 5) for the data-flow implementation of transactional memory in distributed networked systems that are based on general network topologies. We proved that this algorithm achieves poly-log stretch for shared object operations in sequential and one-shot executions. We then provided a dynamic analysis framework (Chapter 6) and presented stretch bounds for several directory algorithms in dynamic execution of shared object operations.

- **NUMA systems:** We presented a distributed consistency algorithm (Chapter 7) for the data-flow implementation of transactional memory in NUMA systems that are based on mesh based topologies. We showed that both stretch and congestion can be minimized simultaneously.

- **Distributed networked and NUMA systems:** We provided a trade-off result (Chapter 8) for transactional memory implementation in distributed networked and NUMA systems in the sense that the communication cost and the execution time (makespan) can not be minimized simultaneously. That is, if we try to minimize makespan we can not minimize communication cost and if we try to control communication cost the makespan increases significantly.

9.2 Future Directions

9.2.1 Tightly-coupled Systems

A natural direction is to investigate the transaction scheduling problem for a combination of window-based model with the balanced workload model to achieve competitive ratio close to $\mathcal{O}(\sqrt{s})$ for windows of transactions. The significance of this is that the previous bounds in the literature considered only one-shot problems and do not generalize well in the window-based model. For example, the bound of $\mathcal{O}(s)$ from [9] in the one-shot model becomes $\mathcal{O}(s \cdot N)$ in the window-based model.

Another natural direction is to determine what is the smallest balancing ratio (number of writes vs total number of reads and writes) that can maintain the same formal bounds in the balanced workload model. Moreover, it is interesting to investigate special cases of transaction conflict graphs which can enable makespan competitive ratios asymptotically smaller than $\mathcal{O}(\sqrt{s})$. Such graphs can represent interesting access patterns to shared resources. It is also interesting to consider scheduling algorithms for *mini-transactions* – simple atomic operations on a small number of locations [8].

Further, it is interesting to investigate how is the performance affected when we take into account the latency to access shared variables. Different processors may have different access times to the shared variables because they may reside in different levels of the memory hierarchy and in different caches. This affects both the lower and upper bounds of the makespan analysis. To properly model the access time variance one idea is to consider a weighted conflict graph and derive new lower and upper bounds on the makespan that take into account the edge weights of the graph. Moreover, it is interesting to design and analyze transaction scheduling algorithm using different performance metrics such as throughput, average response time, aborts per commit ratio, etc. Experimental evaluations for (combinations of) these metrics appeared in several papers, e.g. [7, 46, 121]. It is interesting also to experimentally evaluate these newly designed scheduling algorithms and existing algorithms [7, 45, 59, 142].

9.2.2 Distributed Networked Systems

Ballistic, Relay, and Combine (and also Spiral) have all been analyzed for a single shared object only. Thus, a natural extension is to handle multiple shared objects $\xi_1, \ldots, \xi_k$. In order to handle the case of multiple objects, one idea is to follow a universal TSP (traveling salesperson problem) approach [56, 63, 83]. A *universal TSP approach* computes a TSP tour Q for all the nodes in the network by going through all the nodes in some specific order. Now if we need to visit subset S of nodes inducing a sub-tour, the TSP tour Q approximates the optimal tour for S (in the induced subgraph) within a factor of $\mathcal{O}(\log^4 n/\log\log n)$ [83]. For each shared object ξ_i, we can then compute an approximate TSP tour for the object which visits all the nodes that have transactions that request the object (e.g. for *move*, namely, write operations, or for *lookup*). The TSP tour for the object is obtained by visiting in sequence the involved nodes in the universal TSP tour. For a transaction to execute in a node v, all the objects of the transaction must appear in v. Once all the objects appear in v, the transaction executes and then each object moves to the next node according to the order specified in their respective tours. Eventually, all the transactions will execute. The use of a universal TSP guarantees that there will be no deadlocks. In addition, the total communication

cost of this approach will be close to optimal (poly-log approximation), assuming that the TSP tours of the objects are good approximations (typically, poly-log approximations). This idea will also be helpful in analyzing makespan.

For the experimental evaluation, it will be interesting to extend the HyFlow framework [115] − a Java framework implementation for STM in distributed systems − by including the aforementioned distributed directory algorithms. Moreover, it will be interesting to extend the STAMP benchmarks that are originally designed for tightly-coupled TM systems to support distributed implementations of the TM systems.

9.2.3 NUMA Systems

For TM implementation in NUMA architectures, it will be interesting to explore load and distance competitive ratio bounds of MultiBend for the case of d-dimensional networks with uneven dimensions. Moreover, it will be interesting to extend MultiBend for dynamic networks where nodes join and leave the network over time and make it fault-tolerant. This extension for dynamic networks also applies to algorithms designed for distributed networked systems (as existing algorithms can not handle dynamic networks). Moreover, the problem of incorporating the consistency algorithms in a full-fledged distributed TM system remains as an important open problem.

Bibliography

[1] Ittai Abraham, Danny Dolev, and Dahlia Malkhi. Lls: a locality aware location service for mobile ad hoc networks. In *Proceedings of the 2004 joint workshop on Foundations of mobile computing (DIALM-POMC)*, pages 75–84, New York, NY, USA, 2004. ACM.

[2] Narasimha R. Adiga, Matthias A. Blumrich, Dong Chen, Paul Coteus, Alan Gara, Mark Giampapa, Philip Heidelberger, Sarabjeet Singh, Burkhard D. Steinmacher-Burow, Todd Takken, Mickey Tsao, and Pavlos Vranas. Blue gene/l torus interconnection network. *IBM J. Res. Dev.*, 49(2-3):265–276, 2005.

[3] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The mit alewife machine: A large-scale distributed-memory multiprocessor. Technical report, Cambridge, MA, USA, 1991.

[4] Noga Alon, Gil Kalai, Moty Ricklin, and Larry J. Stockmeyer. Lower bounds on the competitive ratio for mobile user tracking and distributed job scheduling. *Theor. Comput. Sci.*, 130(1):175–201, 1994.

[5] K.N. Amouris, S. Papavassiliou, and Miao Li. A position-based multi-zone routing protocol for wide area mobile ad-hoc networks. In *Proceedings of the IEEE 49th Vehicular Technology Conference (VTC)*, volume 2, pages 1365–1369, 1999.

[6] Mohammad Ansari, Christos Kotselidis, Mikel Lujan, Chris Kirkham, and Ian Watson. On the performance of contention managers for complex transactional memory benchmarks. In *Proceedings of the 8th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 83–90, Washington, DC, USA, 2009. IEEE Computer Society.

[7] Mohammad Ansari, Mikel Lujn, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In Andr Seznec, Joel Emer, Michael O'Boyle, Margaret Martonosi, and Theo Ungerer, editors, *High Performance Embedded Architectures and Compilers*, volume 5409 of *Lecture Notes in Computer Science*, pages 4–18. Springer Berlin / Heidelberg, 2009.

[8] Hagit Attiya. The inherent complexity of transactional memory and what to do about it. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing (PODC)*, pages 1–5, New York, NY, USA, 2010. ACM.

[9] Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir. Transactional contention management asanon-clairvoyant scheduling problem. *Algorithmica*, 57(1):44–61, 2010.

[10] Hagit Attiya, Vincent Gramoli, and Alessia Milani. A provably starvation-free distributed directory protocol. In *Proceedings of the 12th international conference on Stabilization, safety, and security of distributed systems (SSS)*, pages 405–419, 2010.

[11] Hagit Attiya and Alessia Milani. Transactional scheduling for read-dominated workloads. *J. Parallel Distrib. Comput.*, 72(10):1386–1396, 2012.

[12] Hagit Attiya, Hadas Shachnai, and Tami Tamir. Local labeling and resource allocation using preprocessing. *SIAM J. Comput.*, 28(4):1397–1414, March 1999.

[13] B. Awerbuch and D. Peleg. Sparse partitions. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science (FOCS)*, volume 2, pages 503–513, 1990.

[14] B. Awerbuch and M. Saks. A dining philosophers algorithm with polynomial response time. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science (FOCS)*, volume I, pages 65–74, Washington, DC, USA, 1990. IEEE Computer Society.

[15] Baruch Awerbuch and David Peleg. Concurrent online tracking of mobile users. *SIGCOMM Comput. Commun. Rev.*, 21(4):221–233, 1991.

[16] Yossi Azar, Edith Cohen, Amos Fiat, Haim Kaplan, and Harald Räcke. Optimal oblivious routing in polynomial time. *J. Comput. Syst. Sci.*, 69(3):383–394, 2004.

[17] Tongxin Bai, Xipeng Shen, Chengliang Zhang, William N. Scherer III, Chen Ding, and Michael L. Scott. A key-based adaptive transactional memory executor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2007.

[18] Brenda S. Baker and Edward G. Coffman, Jr. Mutual exclusion scheduling. *Theor. Comput. Sci.*, 162(2):225–243, August 1996.

[19] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 184–193, 1996.

[20] Yair Bartal and Adi Rosen. The distributed k-server problem–a competitive distributed translator for k-server algorithms. *Journal of Algorithms*, 23(2):241 – 264, 1997.

[21] Yigal Bejerano and Israel Cidon. An efficient mobility management strategy for personal communication systems. In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking (MobiCom)*, pages 215–222, 1998.

[22] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC)*, pages 1–1, 2011.

[23] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. Proactive transaction scheduling for contention management. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 156–167, 2009.

[24] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. Bloom filter guided transaction scheduling. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 75–86, 2011.

[25] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, pages 127–138, 2008.

[26] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 247–258, 2008.

[27] Hans L. Bodlaender and Klaus Jansen. Restrictions of graph partition problems. part i. *Theor. Comput. Sci.*, 148(1):93–109, August 1995.

[28] Costas Busch, Malik Magdon-Ismail, and Jing Xi. Optimal oblivious path selection on the mesh. *IEEE Trans. Comput.*, 57(5):660–671, May 2008.

[29] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.

[30] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 157–166, 2013.

[31] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of The IEEE International Symposium on Workload Characterization (IISWC)*, pages 35–46. IEEE, 2008.

[32] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112–1118, 1978.

[33] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, 1990.

[34] Dong Chen, Noel Eisley, Philip Heidelberger, Sameer Kumar, Amith Mamidala, Fabrizio Petrini, Robert Senger, Yutaka Sugawara, Robert Walkup, Burkhard Steinmacher-Burow, Anamitra Choudhury, Yogish Sabharwal, Swati Singhal, and Jeffrey J. Parker. Looking under the hood of the ibm blue gene/q network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 69:1–69:12, 2012.

[35] Intel Corporation. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`.

[36] Intel Corporation. A first look at the intel quickpath interconnect. `http://www.intel.com/intelpress/files/A_First_Look_at_the_Intel(r)_QuickPath_Interconnect.pdf`.

[37] Intel Corporation. Who moved the goal posts? the rapidly changing world of cpus. http://software.intel.com/en-us/articles/who-moved-the-goal-posts-the-rapidly-changing-world-of-cpus/.

[38] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís Rodrigues. D2stm: Dependable distributed software transactional memory. In *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 307–313, Washington, DC, USA, 2009. IEEE Computer Society.

[39] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: Streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 67–78, 2010.

[40] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)*, pages 336–346, New York, NY, USA, 2006. ACM.

[41] Murat Demirbas, Anish Arora, Tina Nolte, and Nancy Lynch. Brief announcement: Stalk: a self-stabilizing hierarchical tracking service for sensor networks. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing (PODC)*, pages 378–378, New York, NY, USA, 2004. ACM.

[42] Murat Demirbas, Anish Arora, Tina Nolte, and Nancy Lynch. A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In Teruo Higashino, editor, *Principles of Distributed Systems*, volume 3544 of *Lecture Notes in Computer Science*, pages 299–315. Springer Berlin Heidelberg, 2005.

[43] Michael J. Demmer and Maurice Herlihy. The arrow distributed directory protocol. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, pages 119–133, 1998.

[44] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In Shlomi Dolev, editor, *Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin / Heidelberg, 2006.

[45] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the 27th Annual ACM symposium on Principles of Distributed Computing (PODC)*, pages 125–134, New York, NY, USA, 2008. ACM.

[46] Aleksandar Dragojević, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *Proceedings of the 28th Annual ACM symposium on Principles of Distributed Computing (PODC)*, pages 7–16, New York, NY, USA, 2009. ACM.

[47] Paul Erdős and Alfréd Rényi. On random graphs I. *Publ. Math. Debrecen*, 6:290–297, 1959.

[48] Guy Even, Magnús M. Halldórsson, Lotem Kaplan, and Dana Ron. Scheduling with conflicts: online and offline algorithms. *J. of Scheduling*, 12(2):199–224, April 2009.

[49] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. *J. Comput. Syst. Sci.*, 69(3):485–497, 2004.

[50] Uriel Feige and Joe Kilian. Zero knowledge and the chromatic number. *J. Comput. Syst. Sci.*, 57(2):187–199, 1998.

[51] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Trans. Parallel Distrib. Syst.*, 21(12):1793–1807, 2010.

[52] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP)*, pages 237–246, New York, NY, USA, 2008. ACM.

[53] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. Hardware transactional memory for gpu architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 296–307, 2011.

[54] M. R. Garey and R. L. Grahams. Bounds for multiprocessor scheduling with resource constraints. *SIAM J. Comput.*, 4(2):187–200, 1975.

[55] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[56] Igor Gorodezky, Robert D. Kleinberg, David B. Shmoys, and Gwen Spencer. Improved lower bounds for the universal and a priori tsp. In *Proceedings of the 13th international conference on Approximation, and the 14th International conference on Randomization, and combinatorial optimization: algorithms and techniques (APPROX/RANDOM)*, pages 178–191, 2010.

[57] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust Contention Management in Software Transactional Memory. In *Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency on Object-Oriented Languages (SCOOL)*, 2005.

[58] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In Pierre Fraigniaud, editor, *Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 303–323. Springer Berlin / Heidelberg, 2005.

[59] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *Proceedings of the 24th Annual ACM symposium on Principles of Distributed Computing (PODC)*, pages 258–264, New York, NY, USA, 2005. ACM.

[60] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, March 2007.

[61] Anupam Gupta. Steiner points in tree metrics don't (really) help. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 220–227, 2001.

[62] Anupam Gupta, Mohammad T. Hajiaghayi, and Harald Räcke. Oblivious network design. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm (SODA)*, pages 970–979, 2006.

[63] Mohammad T. Hajiaghayi, Robert Kleinberg, and Tom Leighton. Improved lower and upper bounds for universal tsp in planar metrics. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm (SODA)*, pages 649–658, 2006.

[64] Magnús M. Halldórsson, Guy Kortsarz, Andrzej Proskurowski, Ravit Salman, Hadas Shachnai, and Jan Arne Telle. Multicoloring trees. *Inf. Comput.*, 180(2):113–129, January 2003.

[65] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Michael Chen, Christos Kozyrakis, and Kunle Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6):92–103, November 2004.

[66] 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. The ibm blue gene/q compute chip. *IEEE Micro*, 32(2):48–60, 2012.

[67] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, 2003.

[68] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[69] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, August 2008.

[70] D. Hasenfratz, J. Schneider, and R. Wattenhofer. Transactional memory: How to perform load adaption in a simple and distributed manner. In *Proceedings of the 2010 International Conference on High Performance Computing and Simulation (HPCS)*, pages 163–170, Washington, DC, USA, 2010. IEEE.

[71] Danny Hendler, Alex Naiman, Sebastiano Peluso, Francesco Quaglia, Paolo Romano, and Adi Suissa. Exploiting locality in lease-based replicated transactional memory via task migration. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 121–133, 2013.

[72] Maurice Herlihy, Fabian Kuhn, Srikanta Tirthapura, and Roger Wattenhofer. Dynamic analysis of the arrow distributed protocol. *Theor. Comp. Syst.*, 39(6):875–901, 2006.

[73] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23th International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529, Washington, DC, USA, 2003. IEEE Computer Society.

[74] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, 41(10):253–262, 2006.

[75] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22th Annual ACM symposium on Principles of Distributed Computing (PODC)*, pages 92–101, New York, NY, USA, 2003. ACM.

[76] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[77] Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.

[78] Maurice Herlihy, Srikanta Tirthapura, and Roger Wattenhofer. Competitive concurrent distributed queuing. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 127–133, 2001.

[79] Maurice Herlihy, Srikanta Tirthapura, and Roger Wattenhofer. Competitive concurrent distributed queuing. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing (PODC)*, pages 127–133, 2001.

[80] Dorit S. Hochbaum, editor. *Approximation algorithms for NP-hard problems*. PWS Publishing Co., Boston, MA, USA, 1997.

[81] Cray Inc. Cray xt^{TM} system overview. `http://docs.cray.com/books/S-2423-22/S-2423-22.pdf`.

[82] Sandy Irani and Vitus Leung. Scheduling with conflicts, and applications to traffic signal control. In *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 85–94, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[83] Lujun Jia, Guolong Lin, Guevara Noubir, Rajmohan Rajaraman, and Ravi Sundaram. Universal approximations for tsp, steiner tree, and set cover. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 386–395, 2005.

[84] S. Khot. Improved inapproximability results for maxclique, chromatic number and approximate graph coloring. In *Proceedings of the 42th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 600–609, Washington, DC, USA, 2001. IEEE Computer Society.

[85] Junwhan Kim and B. Ravindran. Scheduling transactions in replicated distributed software transactional memory. In *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 227–234, 2013.

[86] Junwhan Kim and Binoy Ravindran. On transactional scheduling in distributed transactional memory ystems. In *Proceedings of the International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 347–361, 2010.

[87] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Distm: A software transactional memory framework for clusters. In *Proceedings of the 2008 37th International Conference on Parallel Processing (ICPP)*, pages 51–58, 2008.

[88] Robert Krauthgamer and James R. Lee. Navigating nets: simple algorithms for proximity search. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 798–807, 2004.

[89] Fabian Kuhn and Roger Wattenhofer. Dynamic analysis of the arrow distributed protocol. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 294–301, 2004.

[90] Raju Kumar, Riccardo Crepaldi, Hosam Rowaihy, Albert F. Harris III, Guohong Cao, Michele Zorzi, and Thomas F. La Porta. Mitigating performance degradation in congested sensor networks. *IEEE Trans. Mob. Comput.*, 7(6):682–697, 2008.

[91] F. T. Leighton, B. M. Maggs, and S. B. Rao. Packet routing and job-shop scheduling in $O(congestion + dilation)$ steps. *Combinatorica*, 14(2):167–186, 1994.

[92] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, 1991.

[93] Jinyang Li, John Jannotti, Douglas S. J. De Couto, David R. Karger, and Robert Morris. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th annual international conference on Mobile computing and networking (MobiCom)*, pages 120–130, New York, NY, USA, 2000. ACM.

[94] Kai Lu, Ruibo Wang, and Xicheng Lu. Brief announcement: Numa-aware transactional memory. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing (PODC)*, pages 69–70, 2010.

[95] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986.

[96] B. Maggs, F. Meyer auf der Heide, B. Voecking, and M. Westermann. Exploiting locality for data management in systems of limited bandwidth. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 284–293, 1997.

[97] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 79–90, 2010.

[98] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 198–208, 2006.

[99] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. Technical Report TR 893, Computer Science Department, University of Rochester, 2006.

[100] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA)*, pages 69–80, New York, NY, USA, 2007. ACM.

[101] Rajeev Motwani, Steven Phillips, and Eric Torng. Non-clairvoyant scheduling. *Theor. Comput. Sci.*, 130(1):17–47, August 1994.

[102] Mohamed Naimi, Michel Trehel, and André Arnold. A log (n) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34(1):1–13, 1996.

[103] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory Comput. Syst.*, 32(3):241–280, 1999.

[104] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[105] Rajmohan Rajaraman, Andreá W. Richa, Berthold Vöcking, and Gayathri Vuppuluri. A data tracking scheme for general networks. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures (SPAA)*, pages 247–254, 2001.

[106] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems (ASPLOS-X)*, pages 5–17. ACM, 2002.

[107] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. Metatm/txlinux: transactional memory for an operating system. *SIGARCH Comput. Archit. News*, 35(2):92–103, June 2007.

[108] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, 2001.

[109] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7(1):61–77, 1989.

[110] Gabriel Robins and Alexander Zelikovsky. Improved steiner tree approximation in graphs. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 770–779, 2000.

[111] Paolo Romano, Luis Rodrigues, Nuno Carvalho, and Joäo Cachopo. Cloud-tm: harnessing the cloud with distributed transactional memories. *SIGOPS Oper. Syst. Rev.*, 44(2):1–6, 2010.

[112] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 47–56, 2010.

[113] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[114] Mohamed M. Saad and Binoy Ravindran. Snake: control flow distributed software transactional memory. In *Proceedings of the International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 238–252, 2011.

[115] Mohamed M. Saad and Binoy Ravindran. Supporting stm in distributed systems: Mechanisms and a java framework. In *TRANSACT*, pages 1–9. 2011.

[116] David Sainz and Hagit Attiya. Relstm: A proactive transactional memory scheduler. In *International Workshop on Transactional Computing (TRANSACT)*, pages 1–8, 2013.

[117] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th Annual ACM symposium on Principles of Distributed Computing (PODC)*, pages 240–248, New York, NY, USA, 2005. ACM.

[118] William N. Scherer III and Michael L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs (CSJP)*, St. John's, NL, Canada, Jul 2004.

[119] Johannes Schneider and Roger Wattenhofer. Bounds on contention management algorithms. *Theor. Comput. Sci.*, 412(32):4151–4160, 2011.

[120] Steven L. Scott and Et Al. The cray T3E network: Adaptive routing in a high performance 3D torus. In *Proceedings of Hot Interconnects IV Symposium*, pages 147–156, 1996.

[121] Gokarna Sharma and Costas Busch. A competitive analysis for balanced transactional memory workloads. In *Proceedings of the 14th International Conference on Principles of Distributed Systems (OPODIS)*, pages 348–363, 2010.

[122] Gokarna Sharma and Costas Busch. On the performance of window-based contention managers for transactional memory. In *Proceedings of the 13th International Workshop on Advanced in Parallel and Distributed Computational Models (APDCM)*, pages 559–568, 2011.

[123] Gokarna Sharma and Costas Busch. A competitive analysis for balanced transactional memory workloads. *Algorithmica*, 63(1-2):296–322, 2012.

[124] Gokarna Sharma and Costas Busch. Towards load balanced distributed transactional memory. In *Proceedings of the International European Conference on Parallel Computing (Euro-Par)*, pages 403–414, 2012.

[125] Gokarna Sharma and Costas Busch. Window-based greedy contention management for transactional memory: Theory and practice. *Distrib. Comput.*, 25(3):225–248, 2012.

[126] Gokarna Sharma and Costas Busch. An analysis framework for distributed hierarchical directories. *Algorithmica*, pages 1–32, 2013.

[127] Gokarna Sharma and Costas Busch. An analysis framework for distributed hierarchical directories. In *Proceedings of the 14th International Conference on Distributed Computing and Networking (ICDCN)*, pages 378–392, 2013.

[128] Gokarna Sharma and Costas Busch. Transactional memory: Models and algorithms. *SIGACT News*, 45(2):74–103, 2014.

[129] Gokarna Sharma, Costas Busch, and Srivathsan Srinivasagopalan. Distributed transactional memory for general networks. In *Proceedings of the 2012 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1045–1056, 2012. To appear in Distributed Computing.

[130] Gokarna Sharma, Brett Estrade, and Costas Busch. Window-based greedy contention management for transactional memory. In Nancy Lynch and Alexander Shvartsman, editors, *Distributed Computing*, volume 6343 of *Lecture Notes in Computer Science*, pages 64–78. Springer Berlin / Heidelberg, 2010.

[131] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[132] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 141–150, 2009.

[133] Michael F. Spear, Virendra J. Marathe, William N. Scherer, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 179–193, 2006.

[134] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, 2001.

[135] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple reservations and the oklahoma update. *IEEE Parallel Distrib. Technol.*, 1(4):58–71, November 1993.

[136] Kunal Talwar. Bypassing the embedding: algorithms for low dimensional metrics. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing (STOC)*, pages 281–290, 2004.

[137] Srikanta Tirthapura and Maurice Herlihy. Self-stabilizing distributed queuing. *IEEE Trans. Parallel Distrib. Syst.*, 17(7):646–655, 2006.

[138] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT)*, pages 127–136, 2012.

[139] Ruibo Wang, Kai Lu, and Xicheng Lu. Investigating transactional memory performance on ccnuma machines. In *Proceedings of the 18th ACM international symposium on High performance distributed computing (HPDC)*, pages 67–68, 2009.

[140] Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, Lan Gao, and Depei Qian. Software transactional memory for gpu architectures. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1:1–1:10, 2014.

[141] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 261–272. IEEE, 2007.

[142] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 169–178, New York, NY, USA, 2008. ACM.

[143] Bo Zhang and Binoy Ravindran. Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory. In *Proceedings of the 13th international conference on Principles of Distributed Systems (OPODIS)*, pages 48–53, 2009.

[144] Bo Zhang and Binoy Ravindran. Brief announcement: queuing or priority queuing? on the design of cache-coherence protocols for distributed transactional memory. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing (PODC)*, pages 75–76, New York, NY, USA, 2010. ACM.

[145] Bo Zhang and Binoy Ravindran. Dynamic analysis of the relay cache-coherence protocol for distributed transactional memory. In *Proceedings of the 2010 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–11, 2010.

[146] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, 2004.

[147] David Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing*, 3(1):103–128, 2007.

Appendix: Copyright Forms for Published Materials

Confirmation of your Copyright Transfer

Dear Author,

Please note: This e-mail is a confirmation of your copyright transfer and was sent to you only for your own records.

The copyright to this article, including any graphic elements therein (e.g. illustrations, charts, moving images), is hereby assigned for good and valuable consideration to Springer-Verlag effective if and when the article is accepted for publication and to the extent assignable if assignability is restricted for by applicable law or regulations (e.g. for U.S. government or crown employees). Author warrants (i) that he/she is the sole owner or has been authorized by any additional copyright owner to assign the right, (ii) that the article does not infringe any third party rights and no license from or payments to a third party is required to publish the article and (iii) that the article has not been previously published or licensed.

The copyright assignment includes without limitation the exclusive, assignable and sublicensable right, unlimited in time and territory, to reproduce, publish, distribute, transmit, make available and store the article, including abstracts thereof, in all forms of media of expression now known or developed in the future, including pre- and reprints, translations, photographic reproductions and microform. Springer may use the article in whole or in part in electronic form, such as use in databases or data networks for display, print or download to stationary or portable devices. This includes interactive and multimedia use and the right to alter the article to the extent necessary for such use.

An author may self-archive an author-created version of his/her article on his/her own website and/or the repository of Author's department or faculty. Author may also deposit this version on his/her funder's or funder's designated repository at the funder's request or as a result of a legal obligation, provided it is not made publicly available until 12 months after official publication by Springer. He/she may not use the publisher's PDF version, which is posted on www.springerlink.com, for the purpose of self-archiving or deposit. Furthermore, Author may only post his/her own version, provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at www.springerlink.com".

Prior versions of the article published on non-commercial pre-print servers like arXiv.org can remain on these servers and/or can be updated with Author's accepted version. The final published version (in pdf or html/xml format) cannot be used for this purpose. Acknowledgement needs to be given to the final publication and a link must be inserted to the published article on Springer's website,

accompanied by the text "The final publication is available at springerlink.com". Author retains the right to use his/her article for his/her further scientific career by including the final published journal article in other publications such as dissertations and postdoctoral qualifications provided acknowledgement is given to the original source of publication.

Author is requested to use the appropriate DOI for the article. Articles disseminated via www.springerlink.com are indexed, abstracted and referenced by many abstracting and information services, bibliographic networks, subscription agencies, library networks, and consortia.

After submission of the agreement signed by the corresponding author, changes of authorship or in the order of the authors listed will not be accepted by Springer.

This is an automated e-mail; please do not reply to this account. If you have any questions, please go to our help pages.

Thank you very much.

Kind regards,

Springer Author Services

Article Details

Journal title	**Article title**
Distributed Computing	Window-Based Greedy Contention Management for Transactional Memory: Theory and Practice
DOI	**Corresponding Author**
10.1007/s00446-012-0159-7	Gokarna Sharma
Copyright transferred to	**Transferred on**
Springer-Verlag	Tue Jan 24 08:15:05 CET 2012

Confirmation of your Copyright Transfer

Dear Author,

Please note: This e-mail is a confirmation of your copyright transfer and was sent to you only for your own records.

The copyright to this article, including any graphic elements therein (e.g. illustrations, charts, moving images), is hereby assigned for good and valuable consideration to Springer Science+Business Media, LLC effective if and when the article is accepted for publication and to the extent assignable if assignability is restricted for by applicable law or regulations (e.g. for U.S. government or crown employees). Author warrants (i) that he/she is the sole owner or has been authorized by any additional copyright owner to assign the right, (ii) that the article does not infringe any third party rights and no license from or payments to a third party is required to publish the article and (iii) that the article has not been previously published or licensed.

The copyright assignment includes without limitation the exclusive, assignable and sublicensable right, unlimited in time and territory, to reproduce, publish, distribute, transmit, make available and store the article, including abstracts thereof, in all forms of media of expression now known or developed in the future, including pre- and reprints, translations, photographic reproductions and microform. Springer may use the article in whole or in part in electronic form, such as use in databases or data networks for display, print or download to stationary or portable devices. This includes interactive and multimedia use and the right to alter the article to the extent necessary for such use.

An author may self-archive an author-created version of his/her article on his/her own website and/or the repository of Author's department or faculty. Author may also deposit this version on his/her funder's or funder's designated repository at the funder's request or as a result of a legal obligation, provided it is not made publicly available until 12 months after official publication by Springer. He/she may not use the publisher's PDF version, which is posted on www.springerlink.com, for the purpose of self-archiving or deposit. Furthermore, Author may only post his/her own version, provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at www.springerlink.com".

Prior versions of the article published on non-commercial pre-print servers like arXiv.org can remain on these servers and/or can be updated with Author's accepted version. The final published version (in pdf or html/xml format) cannot be used for this purpose. Acknowledgement needs to be given to the final publication and a link must be inserted to the published article on Springer's website,

the final publication and a link must be inserted to the published article on Springer's website, accompanied by the text "The final publication is available at springerlink.com". Author retains the right to use his/her article for his/her further scientific career by including the final published journal article in other publications such as dissertations and postdoctoral qualifications provided acknowledgement is given to the original source of publication.

Author is requested to use the appropriate DOI for the article. Articles disseminated via www.springerlink.com are indexed, abstracted and referenced by many abstracting and information services, bibliographic networks, subscription agencies, library networks, and consortia.

After submission of the agreement signed by the corresponding author, changes of authorship or in the order of the authors listed will not be accepted by Springer.

This is an automated e-mail; please do not reply to this account. If you have any questions, please go to our help pages.

Thank you very much.

Kind regards,

Springer Author Services

Article Details

Journal title	Article title
Algorithmica	A Competitive Analysis for Balanced Transactional Memory Workloads
DOI	**Corresponding Author**
10.1007/s00453-011-9532-3	Gokarna Sharma
Copyright transferred to	**Transferred on**
Springer Science+Business Media, LLC	Mon May 23 19:07:14 CEST 2011

Confirmation of your Copyright Transfer

Dear Author,

Please note: This e-mail is a confirmation of your copyright transfer and was sent to you only for your own records.

The copyright to this article, including any graphic elements therein (e.g. illustrations, charts, moving images), is hereby assigned for good and valuable consideration to Springer-Verlag Berlin Heidelberg effective if and when the article is accepted for publication and to the extent assignable if assignability is restricted for by applicable law or regulations (e.g. for U.S. government or crown employees). Author warrants (i) that he/she is the sole owner or has been authorized by any additional copyright owner to assign the right, (ii) that the article does not infringe any third party rights and no license from or payments to a third party is required to publish the article and (iii) that the article has not been previously published or licensed.

The copyright assignment includes without limitation the exclusive, assignable and sublicensable right, unlimited in time and territory, to reproduce, publish, distribute, transmit, make available and store the article, including abstracts thereof, in all forms of media of expression now known or developed in the future, including pre- and reprints, translations, photographic reproductions and microform. Springer may use the article in whole or in part in electronic form, such as use in databases or data networks for display, print or download to stationary or portable devices. This includes interactive and multimedia use and the right to alter the article to the extent necessary for such use.

Authors may self-archive the Author's accepted manuscript of their articles on their own websites. Authors may also deposit this version of the article in any repository, provided it is only made publicly available 12 months after official publication or later. He/she may not use the publisher's version (the final article), which is posted on SpringerLink and other Springer websites, for the purpose of self-archiving or deposit. Furthermore, the Author may only post his/her version provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be provided by inserting the DOI number of the article in the following sentence: "The final publication is available at Springer via http://dx.doi.org/[insert DOI]".

Prior versions of the article published on non-commercial pre-print servers like arXiv.org can remain on these servers and/or can be updated with Author's accepted version. The final published version (in pdf or html/xml format) cannot be used for this purpose. Acknowledgement needs to be given to the final publication and a link must be inserted to the published article on Springer's website, by

inserting the DOI number of the article in the following sentence: "The final publication is available at Springer via http://dx.doi.org/[insert DOI]". Author retains the right to use his/her article for his/her further scientific career by including the final published journal article in other publications such as dissertations and postdoctoral qualifications provided acknowledgement is given to the original source of publication.

Articles disseminated via http://link.springer.com are indexed, abstracted and referenced by many abstracting and information services, bibliographic networks, subscription agencies, library networks, and consortia.

After submission of the agreement signed by the corresponding author, changes of authorship or in the order of the authors listed will not be accepted by Springer.

This is an automated e-mail; please do not reply to this account. If you have any questions, please go to our help pages.

Thank you very much.

Kind regards,

Springer Author Services

Article Details

Journal title	Article title
Distributed Computing	Distributed Transactional Memory for General Networks
DOI	**Corresponding Author**
10.1007/s00446-014-0214-7	Gokarna Sharma
Copyright transferred to	**Transferred on**
Springer-Verlag Berlin Heidelberg	Mon Mar 24 12:01:40 CET 2014

Confirmation of your Copyright Transfer

Dear Author,

Please note: This e-mail is a confirmation of your copyright transfer and was sent to you only for your own records.

The copyright to this article, including any graphic elements therein (e.g. illustrations, charts, moving images), is hereby assigned for good and valuable consideration to Springer Science+Business Media New York effective if and when the article is accepted for publication and to the extent assignable if assignability is restricted for by applicable law or regulations (e.g. for U.S. government or crown employees). Author warrants (i) that he/she is the sole owner or has been authorized by any additional copyright owner to assign the right, (ii) that the article does not infringe any third party rights and no license from or payments to a third party is required to publish the article and (iii) that the article has not been previously published or licensed.

The copyright assignment includes without limitation the exclusive, assignable and sublicensable right, unlimited in time and territory, to reproduce, publish, distribute, transmit, make available and store the article, including abstracts thereof, in all forms of media of expression now known or developed in the future, including pre- and reprints, translations, photographic reproductions and microform. Springer may use the article in whole or in part in electronic form, such as use in databases or data networks for display, print or download to stationary or portable devices. This includes interactive and multimedia use and the right to alter the article to the extent necessary for such use.

An author may self-archive an author-created version of his/her article on his/her own website and/or the repository of Author's department or faculty. Author may also deposit this version on his/her funder's or funder's designated repository at the funder's request or as a result of a legal obligation, provided it is not made publicly available until 12 months after official publication by Springer. He/she may not use the publisher's PDF version, which is posted on http://link.springer.com, for the purpose of self-archiving or deposit. Furthermore, Author may only post his/her own version, provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at http://link.springer.com".

Prior versions of the article published on non-commercial pre-print servers like arXiv.org can remain on these servers and/or can be updated with Author's accepted version. The final published version (in pdf or html/xml format) cannot be used for this purpose. Acknowledgement needs to be given to the final publication and a link must be inserted to the published article on Springer's website,

accompanied by the text "The final publication is available at link.springer.com". Author retains the right to use his/her article for his/her further scientific career by including the final published journal article in other publications such as dissertations and postdoctoral qualifications provided acknowledgement is given to the original source of publication.

Author is requested to use the appropriate DOI for the article. Articles disseminated via http://link.springer.com are indexed, abstracted and referenced by many abstracting and information services, bibliographic networks, subscription agencies, library networks, and consortia.

After submission of the agreement signed by the corresponding author, changes of authorship or in the order of the authors listed will not be accepted by Springer.

This is an automated e-mail; please do not reply to this account. If you have any questions, please go to our help pages.

Thank you very much.

Kind regards,

Springer Author Services

Article Details

Journal title	Article title
Algorithmica	An Analysis Framework for Distributed Hierarchical Directories
DOI	**Corresponding Author**
10.1007/s00453-013-9803-2	Gokarna Sharma
Copyright transferred to	**Transferred on**
Springer Science+Business Media New York	Mon Jun 10 18:34:15 CEST 2013

Consent to Publish

Lecture Notes in Computer Science

Springer

Title of the Book or Conference Name: Euro-Par 2012

Volume Editor(s): Christos Kaklamanis, Theodore Papatheodorou and Paul Spirakis

Title of the Contribution: Towards Load Balanced Distributed Transactional Memory

Author(s) Name(s): Gokarna Sharma and Costas Busch

Corresponding Author's Name, Address, Affiliation and Email: Gokarna Sharma, Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA gokarna@csc.lsu.edu

When the Author is more than one person the expression "Author" as used in this agreement will apply collectively unless otherwise indicated.

§ 1 Rights Granted

The copyright to this Contribution is transferred to Springer-Verlag GmbH Berlin Heidelberg (hereinafter called Springer-Verlag). The copyright transfer covers the sole right to store, reproduce, publish, disseminate, and sell throughout the world the said Contribution and parts thereof, as well as its translations in any foreign languages, in all forms and media of expression – such as in its electronic form (offline, online) – now known or developed in the future.

Springer will take, either in its own name or in that of Author, any necessary steps to protect these rights against infringement by third parties. It will have the copyright notice inserted into all editions of the Contribution according to the provisions of the Universal Copyright Convention (UCC) and dutifully take care of all formalities in this connection, either in its own name or in that of Author.

The parties acknowledge that there may be no basis for claim of copyright in the United States to a Contribution prepared by an officer or employee of the United States government as part of that person's official duties. If the Contribution was performed under U.S. Government contract, but Author is not a U.S. Government employee, Springer grants the U.S. Government royalty-free permission to reproduce all or part of the Contribution and to authorize others to do so for U.S. Government purposes.

§ 2 Rights Retained by Author

Author retains, in addition to uses permitted by law, the right to communicate the content of the Contribution to other scientists, to share the Contribution with them in manuscript form, to perform or present the Contribution or to use the content for non-commercial internal and educational purposes, provided that the Springer publication is mentioned as the original source of publication in any printed or electronic materials. Author retains the right to republish the Contribution in any collection consisting solely of Author's own works without charge but must ensure that the publication by Springer is properly credited and that the relevant copyright notice is repeated verbatim.

Author may self-archive an author-created version of his/her Contribution on his/her own website and/or in his/her institutional repository, as well as on a non-commercial archival repository such as ArXiv/CoRR and HAL, including his/her final version. Author may also deposit this version on his/her funder's or funder's designated repository at the funder's request or as a result of a legal obligation. Author may not use the publisher's PDF version, which is posted on www.springerlink.com, for the purpose of self-archiving or deposit. Furthermore, Author may only post his/her version provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link should be accompanied by the following text: "The original publication is available at www.springerlink.com". Author retains the right to use his/her Contribution for his/her further scientific career by including the final published paper in his/her dissertation or doctoral thesis provided acknowledgement is given to the original source of publication. Author also retains the right to use, without having to pay a fee and without having to inform the publisher, parts of the Contribution (e.g. illustrations) for inclusion in future work, and to publish a substantially revised version (at least 30% new content) elsewhere, provided that the original Springer Contribution is properly cited.

§ 3 Warranties

Author warrants that the Contribution is original except for such excerpts from copyrighted works (including illustrations, tables, animations and text quotations) as may be included with the permission of the copyright holder thereof, in which case(s) Author is required to obtain written permission to the extent necessary and to indicate the precise source. Springer has the right to permit others to use individual illustrations within the usual limits.
Author warrants that he/she has power to grant the rights in accordance with Clause "Rights Granted", that he/she has not assigned such rights to third parties, that the Contribution has not heretofore been published in whole or in part, contains no libelous statements and does not infringe on any copyright, trademark, patent, statutory rights or proprietary rights of others, including rights obtained through licenses; and that Author will indemnify Springer against any cost, expenses or damages for which Springer may become liable as a result of any breach of this warranty.

§ 4 Delivery of the Work and Publication

Author agrees to deliver to the responsible Volume Editor(s) (for conferences, usually one of the Program Chairs), on a date to be agreed upon, the manuscript created according to the Springer Instructions for Authors. Springer agrees to publish the said Contribution at its own cost and expense. After submission of the Consent to Publish form signed by the Corresponding Author, changes of authorship, or in the order of the authors listed, will not be accepted by Springer.

§ 5 Author's Discount

Author is entitled to purchase for his/her personal use (directly from Springer) books published by Springer at a discount of 33 1/3% off the list price as long as there is a contractual arrangement between Author and Springer. Resale of such copies or of free copies is not permitted.

§ 6 Governing Law and Jurisdiction

This agreement shall be governed by, and shall be construed in accordance with, the laws of the Federal Republic of Germany. The courts of Berlin, Germany shall have the exclusive jurisdiction.

Corresponding Author signs for and accepts responsibility for releasing this material on behalf of any and all Co-Authors.

Signature of Corresponding Author: **Date:** 05/29/2012

. . . . . .

☐ I'm an employee of the US Government and transfer the rights to the extent transferable (Title 17 §105 U.S.C. applies)

31.01.2012
14:00

Vita

Gokarna Sharma received his Bachelors degree in Computer Engineering from Tribhuvan University, Nepal, in 2004, and the Dual Degree European Masters in Computational Logic (EMCL) from the Vienna University of Technology, Austria, in 2007 and from the University of Bozen-Bolzano, Italy, in 2008. In summer 2008, he was a summer consultant at the Enabling Computing Technologies (ECT) research domain of Alcatel-Lucent Bell Laboratories, Murray Hill, New Jersey, USA. He started Doctor of Philosophy study in Computer Science at the School of Electrical Engineering and Computer Science, Louisiana State University, Baton Rouge, USA, in August 2008. Gokarna Sharma will formally receive his Doctor of Philosophy degree in August 2014. His current research interests are on parallel and distributed computing: algorithms, systems, and data structures, distributed sensor networks, and network and graph algorithms.