Thèse présentée pour l'obtention du grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Spécialité

Ingénierie / Systèmes Informatiques

École doctorale

Informatique, Télécommunication et Électronique Paris (ED130)

CALock : Topological Multi-Granularity Locking for Hierarchical data

---

Ayush Pandey

Soutenue publiquement le : *21 Mars 2025*

Devant un jury composé de :

David Bomberg, Professeur, Université de Rennes	*Rapporteur*
Pascal Felber, Professeur, Université de Neuchâtel	*Rapporteur*
Esther Pacitti, Professeure, Université de Montpellier 2	*Examinatrice*
Gaël Thomas, Chercheur senior, INRIA	*Examinateur*
Sathya Peri, Professeur, IIT Hyderabad	*Examinateur*
Marc Shapiro, Directeur de Recherche Émérite, INRIA, Sorbonne Université, LIP6	*Directeur de thèse*
Mesaac Makpangou, Chargé de Recherche [HDR], INRIA	*Directeur de thèse*
Julien Sopena, Maître de Conférences, Sorbonne Université, LIP6	*Encadrant*
Swan Dubois, Maître de Conférences, Sorbonne Université, LIP6	*Encadrant*

To my Family

Copyright:
Except where otherwise noted, this work is licensed under
https://creativecommons.org/licenses/by-nc-nd/4.0/

Abstract

Hierarchies serve as a fundamental structure across various disciplines, modelling hierarchical relationships in computer science, biology, social networks, and logistics. However, dynamic, concurrent updates in real-world systems necessitate synchronisation techniques to maintain data consistency.

This work explores a novel approach, called CALock, to synchronise operations on a hierarchy, based on a novel labelling scheme that facilitates multi-granularity locking. Our approach addresses both concurrent data access and structural modification. CALock exploits the hierarchical topology, via a new labelling scheme, to identify common ancestors of vertices. This enables a thread to efficiently identify an appropriate lock granule. Leveraging variable lock granularity optimizes operations across the hierarchy while ensuring consistency and performance.

We provide a detailed discussion of the CALock labeling and the locking algorithm, prove its properties, and evaluate it experimentally. On static hierarchies, CALock remains competitive with previous labeling schemes. When structural modifications change the hierarchy, CALock has better concurrency and throughput. Indeed, CALock improves throughput by up to 4.5×, and response time by up to 1.5× for workloads that contain structural modifications.

Keywords: Multi-granularity locking, Hierarchical data, Graphs, Locking, Synchronization, Graph topology, Ancestors.

Résumé

Les hiérarchies servent de structure fondamentale dans diverses disciplines, modélisant les relations hiérarchiques en informatique, en biologie, dans les réseaux sociaux ou en logistique. Cependant, les mises à jour concurrentes dans les systèmes réels nécessitent de la synchronisation pour maintenir la cohérence des données. Notre travail explore une nouvelle approche, appelée CALock, pour synchroniser les opérations sur une hiérarchie, en utilisant un schéma d'étiquetage qui facilite le verrouillage à granularité multiple.

Notre approche concerne tout à la fois l'accès concurrent aux données ainsi que les modifications de structure. CALock exploite la topologie hiérarchique, par le biais d'un nouveau schéma d'étiquetage, permettant d'identifier les ancêtres communs de sommets. Cela permet à un fil d'exécution d'identifier le grain de verrouillage approprié de façon efficace. L'utilisation de grains de verrouillage multiples optimise les opérations sur la hiérarchie, tout en garantissant la cohérence et les performances.

Nous présentons une discussion détaillée de l'étiquetage CALock et de l'algorithme de verrouillage. Nous prouvons leurs propriétés, et nous les évaluons de manière expérimentale. Sur des hiérarchies statiques, CALock reste compétitif par rapport aux schémas d'étiquetage précédents. En présence de modifications de structure, CALock améliore la concurrence et le débit. En particulier, CALock améliore le débit jusqu'à 4,5×, et le temps de réponse par jusqu'à 1,5×, pour des charges contenant des modifications structurelles.

Mots-clés: Verrouillage multi-granularité, Données hiérarchiques, Graphes, Verrouillage, Synchronisation, Topologie des graphes, Ancêtres.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

In database systems, maintaining data consistency and integrity during concurrent access is a foundational challenge. As systems grow in scale and complexity, managing data concurrency effectively becomes critical, especially in scenarios where multiple transactions access shared data simultaneously. Locking mechanisms provide a structured approach to concurrency control, with multi-granularity locking (MGL) being a particularly valuable technique. MGL has been widely studied in traditional database settings, especially for table spaces organized in tree-structured data, where it efficiently manages hierarchical access patterns.

However, as data models evolve to incorporate intricate hierarchies and highly connected structures like graphs and networks, traditional approaches to MGL require adaptation. Hierarchical data models are common in domains such as file systems, organizational structures, ontologies, and XML databases. These models naturally capture relationships across different levels of abstraction, where data elements can be accessed concurrently by multiple users or processes. Such concurrent access raises the possibility of conflicts and inconsistencies, particularly when different parts of the hierarchy or network are accessed simultaneously.

To address these issues, MGL provides a mechanism to manage locks at varying levels of granularity, allowing coarse-grained locks at higher levels of the hierarchy and fine-grained locks at lower levels. This flexibility supports efficient concurrency control by enabling transactions to lock large portions of data when broad access is required or smaller segments for more specific operations. However, implementing MGL effectively in systems with complex hierarchies and interconnected data structures introduces unique challenges. Balancing performance with fairness becomes crucial, as the system's performance depends on minimizing lock contention and avoiding deadlocks, while ensuring that locks at different levels can coexist without causing unnecessary blocks or resource wastage.

In summary, as data structures grow more intricate, a deeper understanding of multi-granularity locking tailored to hierarchical and graph-based data is essential for achieving scalable, efficient concurrency control. This research aims to address these complexities, exploring adaptations of MGL that can support parallelism and data integrity in environments characterized by complex, multi-level connections.

1.1 Problem Statement

While multi-granularity locking (MGL) is widely applied in simple hierarchical data models, adapting it to handle complex, irregular, and deeply nested hierarchies remains challenging. Traditional MGL approaches often face scalability issues in environments where hierarchies expand significantly in size and depth, or where relationships become less rigid, as in graph-like data models. One core challenge lies in efficiently determining the optimal placement of MGL locks within these intricate structures. This process must consider both the structural depth and irregularity of the hierarchy, which can make lock placement a computationally intensive task.

Furthermore, once locks are set, identifying and managing conflicts between them becomes increasingly complex, especially as parallel transactions interact at multiple levels of the hierarchy. These interactions can introduce significant synchronization challenges, particularly in distributed environments with high latency or frequent partitioning. Addressing these issues requires an extension of existing MGL frameworks that supports not only efficient lock placement and conflict detection but also maintains high performance and data consistency across complex hierarchical structures.

1.2 Contributions

In this work, we present CALock, a novel MGL protocol designed to address the challenges of concurrency control in complex hierarchical data models. CALock extends the traditional MGL framework to support hierarchies with arbitrary depth and complexity, enabling efficient and scalable concurrency management.

MGL techniques establish the granularity at which a transaction acquires a lock, to optimize access based on the transaction's needs. In CALock, we use a topological partial ordering of the vertices in the hierarchy to establish this granularity. This ordering is defined through a vertex labelling scheme. At runtime, these labels help determine the most suitable lock granularity for a given lock request based on the transaction's access pattern.

CALock attempts to minimize the granularity of a lock, ensuring that each lock covers only the necessary portion of a hierarchy. This approach avoids fixed vertex ordering and the need for frequent relabelling, while still providing the same locking guarantees as other MGL techniques. Each vertex label consists of its set of common

ancestors, computed recursively through breadth-first traversal, allowing CALock to achieve flexible and efficient locking in complex, dynamic hierarchies.

1.3 Thesis Structure

The remainder of this thesis is organized as follows:

Chapter 2 presents an overview of concurrency control, multi-granularity locking and the challenges associated with complex hierarchical data models.

Chapter 3 reviews existing research in the field of multi-granularity locking, highlighting the limitations of current techniques.

Chapter 4 provides a theoretical framework for CALock which includes the concept of lowest common ancestors in graphs, and the problem formulation for optimal locking grain selection and the proof of optimality of CALock labelling.

Chapter 5 introduces CALock, our novel multi-granularity locking protocol, and describes its design and implementation. We also discuss the properties of CALock like safety, liveness and fairness.

Chapter 6 discusses the formal properties of CALock, including safety, liveness, and fairness. We also present the complexity analysis of CALock and compare it against state-of-the-art techniques.

Chapter 7 describes the implementation of CALock, including the design choices and optimizations made to improve performance.

Chapter 8 presents an experimental evaluation of CALock, comparing its performance against state-of-the-art techniques in a variety of scenarios using both, micro-benchmarks and macro-benchmarks.

Finally, Chapter 9 concludes the thesis, summarizing our contributions and outlining future research directions in the field of multi-granularity locking for complex hierarchical data models.

Background

2

Contents

Concurrency control is a crucial component in systems where multiple processes or threads simultaneously access shared resources, such as data structures, database tables, and system resources. Effective concurrency control mechanisms employ various synchronization techniques to maintain the safety and consistency of these shared resources. As noted by Raynal [Ray13], "Synchronization is a set of rules and mechanisms that allows the specification and implementation of sequencing properties on statements issued by the processes so that all the executions of a multiprocess program are correct."

In this chapter, we provide an overview of concurrency control mechanisms, focusing on locking. We introduce a few classical locking mechanisms, their role in synchronization and in ensuring data consistency. We discuss additional challenges brought forth by hierarchical data structures and the need for specialized locking techniques to manage concurrent access to them. Finally, we introduce the concept of multi-granularity locking and its role in balancing the trade-offs between the extremes of locking in hierarchical data structures.

2.1 Hierarchical data structures

Hierarchical data models, which have their roots in the early days of computer science, play a critical role in the representation and storage of structured information. The inception of hierarchical data can be traced back to the 1960s with the development of the Information Management System (IMS) by IBM [Bla98]. IMS was one of the earliest database management systems designed specifically for organizing hierarchical data and was initially created to manage the complex manufacturing data associated with the Apollo space program. IMS establishes a well-defined parent-child relationship between data entities, which is a defining characteristic of hierarchical models [Dat00].

Over the decades, hierarchical data structures have remained pertinent, particularly in domains that require nested or tree-like relationships. They are widely utilized in XML data management, file system hierarchies, and organizational charts [ABS99]. The relevance of hierarchical models has only increased in contemporary big-data contexts, where they are often integrated with other data models, such as graphs, to address the challenges posed by the complexity and interconnectedness of modern data environments.

For instance, hierarchical data models are fundamental in NoSQL databases like Apache HBase, where the efficient storage and rapid retrieval of large-scale, semi-structured data are critical [Geo11]. The structured nature of hierarchical models aligns well with applications in social networks, content management systems, and cloud-based storage solutions, where understanding and managing data relationships is essential. Consequently, hierarchical data models remain foundational to modern computing, providing an effective framework for organizing and retrieving complex, nested information.

2.1.1 Structure of a Hierarchy

A hierarchy is a tree like structure with an additional property that a vertex can have multiple parents. Formally, a hierarchy is defined as follows:

Definition 1. *A hierarchy is a directed graph H=(V, E, R) and R $\in$ V where*

- *V is a finite set of vertices.*
- *$E \subset V \times V$ is a set of directed edges where each edge (u,v) represents a parent-child relationship between vertices u (the parent) and v (the child).*

- *R is a designated root of the hierarchy such that there is a path from R to every other vertex in V.*

2.1.2 Operations in Hierarchical Data Structures

Hierarchical data structures support a variety of operations that enable the traversal, modification, and querying of the data. These operations are essential for managing the relationships between vertices and maintaining the integrity of the hierarchy. Some common operations on hierarchical data structures include:

- **Search and Query:** Searching and querying operations involve locating specific vertices or paths within the hierarchy based on predefined criteria. These operations are essential for retrieving relevant information from the data structure. These operations result in read-only access to the hierarchy. While Data reads focus on vertices, search and query operations focus on relationships between vertices and use the edges to navigate the hierarchy.

- **Data Read and Write:** Reading and writing operations involve accessing and updating the attributes of vertices in the hierarchy. These operations are fundamental for interacting with the data and retrieving information from the structure. Often, one or more vertices are involved in a single read or write operation, depending on the specific requirements of the application. These operations do not alter the topology of the hierarchy.

- **Structural Modifications:** Inserting and deleting vertices or edges in a hierarchical structure can alter the topology of the hierarchy. These operations are crucial for creating new data and maintaining the integrity of the relationships between vertices. We call such operations *structural modifications*.

2.2 Need for Concurrency Control in Hierarchical Data Structures

Locking mechanisms in hierarchical data structures are essential for ensuring data consistency and integrity in concurrent environments where multiple processes or threads may attempt to access or modify data simultaneously. Given the inherently nested and interdependent relationships among vertices in hierarchical structures,

specialized locking techniques are often required to effectively manage concurrent access.

The primary objective of locking in these structures is to prevent race conditions, which occur when two or more operations interfere with one another, potentially resulting in an inconsistent or incorrect data state. By implementing appropriate locking strategies, it is possible to coordinate access to shared resources, thus safeguarding the integrity of the data and maintaining the correctness of operations performed within the hierarchy.

This work on locking focuses on a multicore system where threads concurrently access a shared graph. Such use-cases provide the possibility to use a single centralized scheduler which decides on the ordering of the lock requests. In distributed deployments like HPC clusters where the communication can be guaranteed to be quasi-instantaneous, locking can still be useful since synchronization primitives can be implemented without worrying about catastrophic network partitions. Geo-distributed graphs and synchronization are out of the scope of this work. As such, locking in any form is not an efficient solution for use-cases where the graph is geo-distributed and the communication latency (ergo the synchronization delay) is high.

The requirement for any locking technique over graphs is to protect the vertices and edges of the graph against concurrent write access. Writes can modify the data on the vertices or add/remove edges from the graph.

A lock protocol typically involves five main steps:

1. **Requesting a Lock:** A thread initiates the locking process by submitting a request to acquire a lock on a specific set of vertices that it intends to access for a write operation. This request is an attempt to ensure that only the requesting thread can modify said vertices during the operation, thereby maintaining data integrity and avoiding interference from other concurrent operations.

2. **Detecting Conflicts:** Upon receiving a lock request, the locking protocol detects conflicts with locks held by other threads. This step is critical to preserve mutual exclusion. If a conflict is detected, the protocol determines whether the requesting thread should proceed or be temporarily blocked.

3. **Acquiring a Lock:** If conflict detection confirms no overlapping locks, the locking protocol grants the requested lock, allowing the thread to proceed with its intended write operation. In cases where a conflict exists, the requesting thread is put into a blocked state, where it waits until the conflict is resolved,

typically by the release of the conflicting lock. This blocking mechanism is designed to prevent busy-waiting and reduce system resource usage.

4. **Performing an Operation:** With the lock granted, the thread performs its designated read or write operation on the locked vertices. The lock ensures that no other thread can access these vertices concurrently for a write operation. Upon completing the operation, the thread prepares to release the lock, signaling the end of its exclusive access.

5. **Releasing the Lock:** After finishing the operation, the thread releases the lock on the vertices, allowing other threads to request and potentially acquire locks on the same vertices. This release phase reopens access to the locked vertices, ensuring that other threads waiting to modify or read the same graph segments can proceed according to the locking protocol's scheduling and prioritization rules.

An implementation of a locking protocol makes a few assumptions on the implementation and use.

1. **Atomicity of Lock Operations:** It is assumed that lock acquisition and release operations are atomic, meaning they are indivisible and cannot be interrupted mid-operation. This atomicity ensures that no two threads can simultaneously request and acquire a lock on the same resource, which is essential to prevent race conditions.

2. **Non-Malicious Threads:** The algorithm assumes that threads behave cooperatively, following established protocols without attempting to bypass or tamper with the locking mechanism. Also, threads do not attempt to access vertices without acquiring an appropriate lock. This assumption excludes scenarios where threads might act maliciously or fail to adhere to the locking protocol, simplifying the conflict detection and lock management processes.

3. **Fairness in Lock Scheduling:** Many locking algorithms assume a fair scheduling mechanism, meaning that each lock request is eventually granted in a finite amount of time, avoiding starvation where certain threads are indefinitely blocked. This fairness ensures that all threads have equitable access to shared resources, which is particularly important in high-throughput environments like HPC.

4. **Deterministic Conflict Detection:** It is assumed that the conflict detection mechanism can deterministically identify conflicts between lock requests and consistently enforce access restrictions. This assumption is crucial for ensuring

that threads either proceed with their operations or are correctly blocked when a conflict is detected.

5. **Consistency of Shared State:** The algorithm assumes that the state of shared data remains consistent and is correctly updated before each lock release. This assumption ensures that subsequent operations access an accurate version of the graph data, preventing issues such as stale data reads or lost updates.

Together, these prerequisite steps and assumptions form the foundation for effective concurrency management, allowing the locking algorithm to maintain data integrity while maximizing system efficiency.

2.3 Role of Locking in Concurrency Control

As computing systems increasingly rely on concurrent processing, effective concurrency control mechanisms are essential for ensuring the integrity and reliability of shared resources. Locking mechanisms are a foundational strategy in managing access to these resources, allowing for mutual exclusion and preventing conflicts that can arise when multiple threads attempt to modify shared data simultaneously.

In this section, we explore the critical roles that locking plays in concurrency control. We first examine how locks prevent race conditions by ensuring that only one thread can modify shared data at a time. Next, we discuss the importance of locks in maintaining data consistency through atomic operations. Finally, we address the challenges of deadlock prevention and the strategies that can be employed to minimize this risk. Through this overview, we highlight the significance of locking mechanisms in supporting robust and reliable concurrent programming.

2.3.1 Safety

Race conditions are a fundamental concern in concurrent programming, occurring when multiple threads access and modify shared data simultaneously without proper synchronization mechanisms in place. This lack of coordination can lead to unpredictable and erroneous outcomes, as the final state of the data may depend on the timing of the thread executions rather than a well-defined sequence of operations. Locks play a critical role in preventing race conditions by ensuring that only one

thread can modify shared data at any given time. By acquiring a lock before accessing the data, a thread effectively prevents other threads from making concurrent modifications, thereby establishing a controlled environment for data access.

The implementation of locks creates critical sections—regions of code that must be executed by only one thread at a time. When a thread acquires a lock, it enters the critical section and can safely perform operations on the shared data, knowing that no other thread can interfere during this period. Once the operations are complete, the thread releases the lock, allowing other waiting threads to enter the critical section. This mechanism not only prevents race conditions but also enhances the reliability of concurrent programs, as it provides a straightforward method for enforcing mutually exclusive access to shared resources.

Locks are instrumental in ensuring data consistency across concurrent operations. In computing, consistency refers to the property that shared data remains in a valid state throughout the execution of concurrent transactions. Locks help maintain this consistency by enforcing atomicity, a key aspect of transaction management that dictates that operations on shared data should be treated as indivisible units. This means that a series of operations can either be completed in their entirety or not executed at all, effectively preventing intermediate states that could lead to data corruption.

For instance, consider a banking application where a user initiates a transfer of funds from one account to another. This operation typically involves several steps: debiting the source account, crediting the destination account, and possibly updating transaction logs. If these operations are interrupted by another thread accessing the same accounts, the integrity of the transaction may be compromised, leading to scenarios such as double spending or incorrect balances. By using locks to protect these operations, the system ensures that each transfer is executed atomically, thus maintaining the integrity and consistency of the financial data. This mechanism is crucial not only for ensuring correctness in transactional systems but also for building trust in applications that rely on shared data.

2.3.2 Liveness

While locks are essential for ensuring safe and consistent access to shared resources, their improper use can lead to deadlocks—situations where two or more threads are waiting indefinitely for each other to release locks, thereby causing a standstill in program execution. Deadlocks can severely impact system performance and responsiveness, making it critical to implement strategies for their prevention.

Several techniques can be employed to mitigate the risk of deadlocks in systems utilizing locks. One common approach is lock ordering, which mandates that all threads acquire locks in a predefined sequence. By adhering to a global order for lock acquisition, the system effectively eliminates circular wait conditions, which are a primary cause of deadlocks. For example, if Thread A acquires Lock 1 and Thread B acquires Lock 2, both threads should be programmed to request subsequent locks in the same order. This strategy significantly reduces the chances of deadlock occurrences, as it prevents the formation of cycles in the waiting graph.

Another useful technique is the implementation of timeout mechanisms. By allowing threads to specify a maximum waiting period for acquiring locks, the system can reduce the likelihood of indefinite waiting. If a thread cannot acquire the desired lock within the specified time-frame, it can abandon its attempt and either retry later or perform alternative actions. This not only aids in breaking potential deadlocks but also enhances overall system responsiveness, as it prevents threads from being indefinitely stalled.

2.4 Requirements

The design of a locking mechanism for hierarchical data structures must satisfy several key requirements to ensure the correctness, efficiency, and scalability of the system. These requirements are essential for guiding the development and evaluation of the locking algorithm. They can be categorized into four main areas of concern:

R_{VGuard} *Identifying a lock guard for every vertex of a hierarchy*. For a given lock target, the lock protocol must effectively and efficiently identify the appropriate lock guard that protects this target. This is essential to ensure that the lock granularity can be minimized while maintaining data consistency. If a protocol fails to identify an appropriate guard, it may lead to unnecessary contention and reduced concurrency.

R_{RGuard} *Finding an appropriate lock guard for a request*. When a thread requests a lock on a set of lock targets, the locking protocol must determine an appropriate lock guard for this request that maximizes concurrency and minimizes contention with other lock requests. This involves considering the topology of the hierarchy and the relationships between vertices to minimize conflicts where two threads request locks on overlapping grains.

$R_{Conflict}$ *Efficiently detecting conflicts between locks.* Since a correct locking protocol is workload agnostic, it must allow a thread to efficiently detect conflicts with other lock requests. In MGL, this involves testing the ancestor-descendant relationships between two lock guards which in turn implies grain overlap. If a thread requests a lock on a vertex, then to prevent conflicts, a locking protocol must ensure that no ancestor or descendant of this vertex is locked by another thread in a conflicting mode.

$R_{Metadata}$ *Housekeeping the metadata required to implement the locking protocol.* The additional metadata required to implement the locking protocol and the conflict detection mechanism must be managed efficiently to minimize the overhead of using that particular locking protocol. If the metadata management is prohibitively expensive for certain workloads, the locking protocol may not be practical for real-world applications that encounter those workloads.

2.5 Locking in Hierarchical Data Structures

Before delving into the detailed discussion of locking and its relevance to hierarchical data, it is essential to establish the foundational terminology that will be used throughout this thesis. Hierarchical locking, as a mechanism for managing concurrent access to data structured in a hierarchy, involves a variety of concepts that require precise definitions to avoid ambiguity and ensure clarity.

In particular, concepts like "lock grain," which describe levels of granularity in hierarchical locking, are critical for understanding the nuances of how concurrency is managed in hierarchical and graph-based data models. While some of these terms are standard in the literature, others are specific to the context of this thesis, reflecting the unique challenges and solutions presented by hierarchical data structures.

2.5.1 Key terms in hierarchical locking

Understanding the terminology associated with hierarchical locking is crucial for effectively discussing and implementing locking mechanisms in hierarchical data structures. Below are some fundamental terms used in this context:

Lock Target

The lock target refers to a specific vertex or a set of vertices within the hierarchy that a lock is intended to protect. Identifying the appropriate lock target is critical for preventing conflicts and ensuring that concurrent operations do not interfere with each other. The selection of lock targets can influence the efficiency of locking protocols, as improper targeting may lead to unnecessary contention or reduced concurrency.

Lock Guard

The lock guard is an entity that serves as the point of synchronization for a given set of lock targets. When a thread wishes to access a particular target, it must acquire the corresponding lock guard to ensure exclusive access to the target. The lock guard acts as a sentinel, preventing other threads from modifying the protected data until the lock is released. The mapping between a lock target and its corresponding lock guard is dependent on the locking protocol. We shall see examples of this in Chapter 3.

Lock Grain and Granularity

Lock grain refers to the scope of the data that is being locked when a lock is acquired on a guard. The size of the grain is called its *granularity*. In hierarchical locking, this can vary from coarse-grained locks that encompass large portions of the hierarchy (e.g. entire subgraphs) per lock guard to fine-grained locks that target individual vertices or smaller groups of vertices per lock guard. The choice of lock grain (resp. granularity) directly impacts the level of concurrency and performance in a system; while fine-grained locks allow for greater parallelism, they can introduce additional complexity in managing locks.

2.5.2 Classical locking approaches for connected data

A common approach for synchronization in connected data, like graphs, is lock coupling [BS77], also known as hand-over-hand locking. In hand-over-hand locking, a thread locks a vertex before traversing to one of its children in the structure, unlocking the parent once the child is locked. This allows for per vertex synchronization, which improves concurrency by permitting multiple threads to work on different

parts of the hierarchy simultaneously. However, as Leis et al. [LHN19] show, lock coupling can lead to suboptimal locking performance on modern hardware.

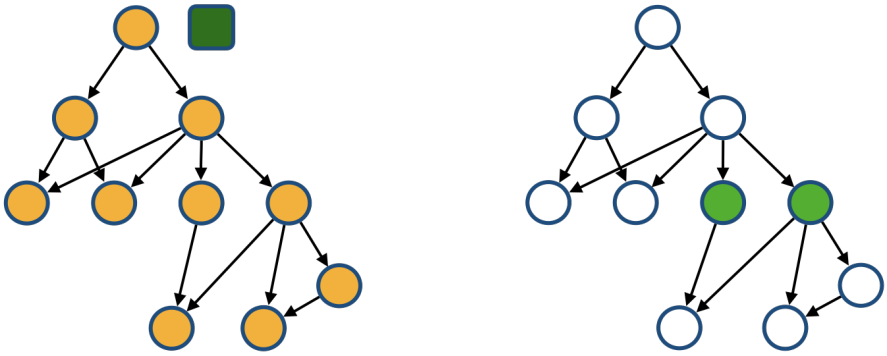
An alternative to lock coupling, targeted specially for B-trees is a variant called B-Link tree [LY81]. A B-link tree is an extension of the B-tree data structure that adds sibling pointers to improve concurrency and simplify lock management in multithreaded environments. In a traditional B-tree, nodes are connected in a hierarchical structure with each node containing keys and pointers to child nodes, but concurrent modifications require complex locking strategies to maintain consistency. B-link trees address this by adding *right sibling* pointers between nodes at the same level, allowing threads to traverse the tree even during insertions or deletions without needing to lock the entire structure. This enables more efficient concurrent access, as threads encountering locked nodes can follow sibling links to find the next valid node. B-link trees require additional mechanisms to ensure that pointer updates are atomic since multiple threads may attempt to modify the same pointers concurrently. Additionally, this approach is only applicable to tree-based structures and is not be directly transferable to other hierarchical data models.

Classical locking techniques for hierarchical data structures are often designed with a synchronization hypothesis. A hypothesis defines how and when synchronization mechanisms (like locks, barriers, or other coordination tools) are applied to ensure threads or processes do not interfere with each other. Lock coupling uses a simple hypothesis that a thread must acquire a lock on a vertex before traversing to its children. B-link trees introduce a more sophisticated hypothesis by allowing threads to follow sibling pointers when encountering locked nodes, thus reducing contention and improving concurrency. Such design choices limit their use in general purpose applications where the topology of a linked structure is not known a priori.

2.5.3 Fixed Grain locking

Unlike classical approaches which define a specific synchronization mechanism, reader-writer locks are often appropriated for hierarchical data by associating a lock guard for a predefined set of lock targets. For example, a unique lock guard can be assigned for all vertices of the same depth from the root of the hierarchy. Since this guard is fixed and does not depend on the lock request, we refer to this as *fixed-grain locking*.

Fixed-grain locking has two extremes: We can either have a single lock guard for all the vertices in the hierarchy (coarse-grain locking) or a unique lock guard for each target vertex (fine-grain locking).



(a) Coarse-grain lock (b) Fine-grain locks

Fig. 2.1: Fixed-grain locks in a hierarchical data structure (lock guard in green and the corresponding grain in yellow).

Coarse-Grain Locking An oversimplified approach for correct thread synchronization is to guard an entire hierarchy with a reader-writer lock such that any thread accessing the hierarchy must first acquire this lock. This approach is called *coarse-grain locking* and is shown in Figure 2.1a. A thread that wishes to access any vertex in the hierarchy must first acquire a lock on the entire data structure and consequently block all other writers until it releases this lock. Coarse-grain locking is simple to implement and has extremely low locking overhead but suffers from the highest possible contention as all threads must acquire the same lock to access any part of the hierarchy.

Fine-Grain Locking Another well known approach to locking is when every target vertex is its own guard i.e. every vertex has its own unique lock. We call this *fine-grain locking*. As shown in Figure 2.1b, vertices are locked individually and a thread that needs to access multiple vertices must acquire multiple locks. Fine-grain locking has the advantage of reducing contention as threads can access disjoint parts of the hierarchy concurrently. However, the overhead of acquiring multiple locks makes this approach less efficient.

The approaches shown in Figures 2.1a and 2.1b have their own trade-offs. Coarse-grain locking causes contention by unnecessarily blocking threads that access disjoint parts of the hierarchy. Fine-grain locking, on the other hand, introduces significant overhead due to the need to acquire multiple locks for accessing multiple vertices with the additional requirement of deadlock detection and prevention/resolution.

2.5.4 Multi-Granularity Locking

In contrast to fixed-grain locking, multi-granularity locking adapts the granularity of locks based on the structure of the hierarchy and the lock request. According to Gray et al. [Gra+75], "Multi-granularity locking involves locking a guard in a hierarchy such that a single lock guard is sufficient to protect all the target vertices protected by that guard i.e. a thread that requests a write (exclusive) lock on a guard vertex implicitly exclusively locks all its targets when its request is granted".

Multi-granularity locking (MGL) [Gra+75] techniques find a balance between the two extremes of fixed-grain locking based on the topology of the hierarchy. Since *granularity* depends on the topology of the graph and on the lock request, lock requests with the same set of lock targets for different hierarchies can have different granularities, hence the name. A classical example of MGL is *Intention locks* [RS77] which are often used in database indices to optimize hierarchical access [Mic22].

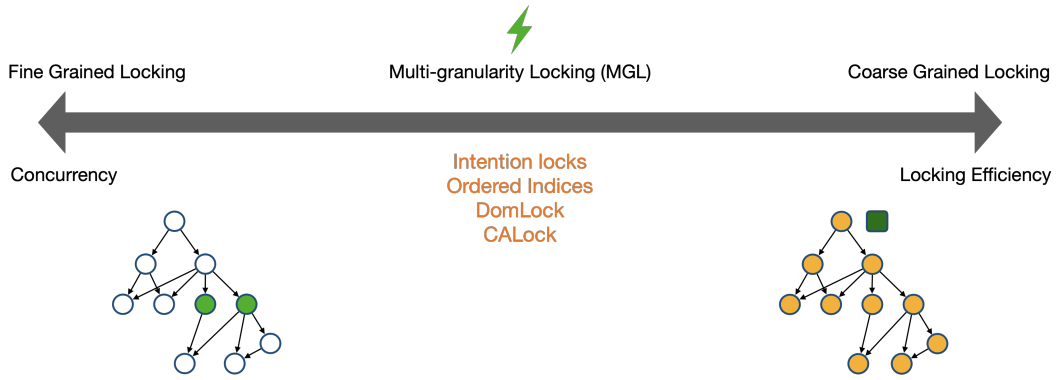


Fig. 2.2: Multi-Granularity locking provides a balance between Fine-grain and Coarse-grain locking.

Structural modifications, which alter the topology of a hierarchy are especially challenging for MGL techniques. There already exist effective MGL protocols for hiaraarchies that do not undergo structural modifications, this is not the case for graphs whose structure can mutate. We discuss this in Chapter 3.

Related Work

3

Contents

Multi-granularity locking is a well-known solution to the problem of hierarchical locking in database systems. With MGL, locking a vertex in a hierarchy in a specific mode implicitly locks some or all its descendants in the same mode. This is a powerful concept that allows for efficient locking of hierarchical data structures without the need to explicitly lock large portions of the hierarchy. While powerful, implementing an MGL protocol that successfully achieves this goal is non-trivial.

The different locking approaches discussed in this chapter use different mechanisms to address the requirements specified in Section 2.4, each with its own trade-offs. It would be appropriate to claim that no approach is all encompassing and the choice of the locking approach depends on the specific requirements of the application and the hierarchy being used.

In this chapter, we explore some existing locking protocols and how successful they are in addressing the requirements of MGL. We present their strengths, weaknesses and the niche they occupy. Finally, we present the trade-offs between them and finally, motivate the need for a new hierarchical locking protocol which addresses the limitations of the existing protocols by classifying them into categories based on the requirements they fulfil.

3.1 Existing Lock techniques

3.1.1 Level-based locks

Level-based locking is a synchronization technique employed in hierarchical data structures to control concurrent access by just utilizing their structure. Level-based locks form lock grains based on some topological property of the hierarchy. One approach to level-based locking is to associate a lock guard per level of the hierarchy. This level is determined by the depth of vertices in the hierarchy. Vertices with the same depth are associated with a single lock guard. As shown in Figure 3.1, the hierarchy is divided into levels based on the depth of the vertices and a lock guard is associated with each level. This guard protects all vertices in that level. To lock a target vertex, a thread acquires the lock associated with the level of that target and locks all the vertices in that level. For example, in Figure 3.1, in order to lock target vertex D, vertices E, F and G are implicitly locked as well since they are in the same level and protected by the same guard as D.

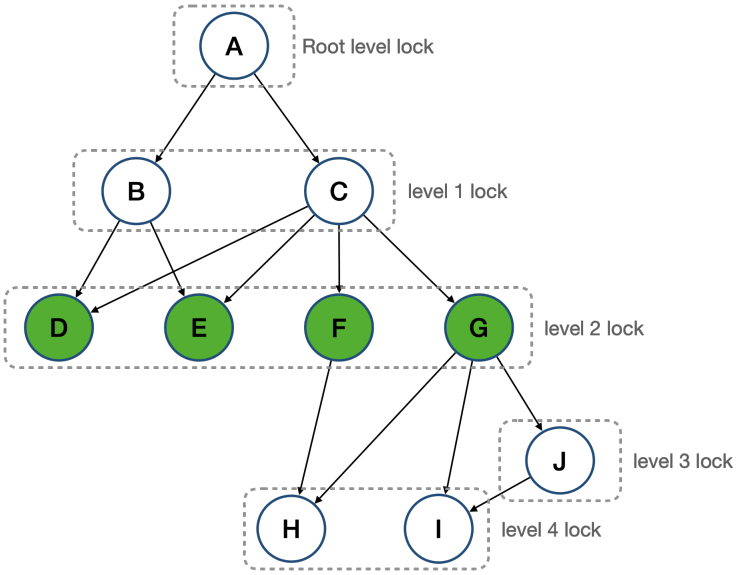


Fig. 3.1: Hierarchical locks with fixed grains per level and a lock on level 2

Level-based locks fulfil only requirement $\mathbf{R_{VGuard}}$ since the lock guard is fixed for a set of vertices. However, since the lock grain is fixed by level, acquiring a lock on a specific level implicitly locks all the vertices in that level. This leads to unnecessary conflicts between conflicts that access disjoint sets of vertices in the same level. For example, consider two threads T_1 and T_2 that want to lock vertices D and E respectively. Since the lock guard for level 2 is the same, T_1 and T_2 will always conflict with each other even though they are accessing different vertices.

3.1.2 Intention Lock

Intention locks are used to coordinate concurrent access by indicating a thread's intention to acquire more fine-grain locks within a hierarchy. Unlike traditional locks that are placed directly on lock targets, intention locks are acquired at higher levels of a hierarchy to signal that a thread plans to acquire finer-grained locks on specific target vertices which exist deeper in the hierarchy. These intention locks protect the paths that lead to the target vertices from the root. By guarding these paths, intention locks ensure that the target vertices can be accesses exclusively by writers.

Intention locking was one of the first approaches to hierarchical locking designed to efficiently meet requirements $\mathbf{R_{VGuard}}$ and $\mathbf{R_{Conflict}}$. Gray et al. [Gra+75] introduced three new lock modes: IX (Intention Exclusive), IS (Intention Shared) and SIX (Shared Intention Exclusive), which allow a thread to signify its intention to acquire an exclusive lock, a shared lock or a shared lock which can be upgraded to an exclusive lock on a descendant of a vertex locked under IX, IS or SIX modes, respectively. In the intention lock protocol, a thread acquires these intention locks on all the paths from the root to the lock target, which is then locked in either a shared (S) or exclusive (X) mode. Thus, all the ancestors of a target vertex are locked in an intention mode, and the target vertex is locked in a shared or exclusive mode.

The protocol requires that threads place these locks in a depth-first manner. This ordering ensures that two concurrent threads trying to lock the same target will always acquire the locks in the same order, allowing them to detect conflicts deterministically. Table 3.1 shows the compatibility matrix for the intention locks.

Mode	NL	IS	IX	S	SIX	X
NL	Y	Y	Y	Y	Y	Y
IS	Y	Y	Y	Y	Y	N
IX	Y	Y	Y	N	N	N
S	Y	Y	N	Y	N	N
SIX	Y	Y	N	N	N	N
X	Y	N	N	N	N	N

Tab. 3.1: Compatibility matrix for intention lock modes. **NL**: NoLock, **IS**: Intention Shared, **IX**: Intention Exclusive, **S**: Shared, **SIX**: Shared Intention Exclusive, **X**: Exclusive

Consider the example in Figure 3.2 where a thread T_1 wishes to acquire an exclusive lock (X) on target vertex D and another thread, T_2 on target vertex G. With intention locks, both threads start from the root i.e. A and acquire intention exclusive locks

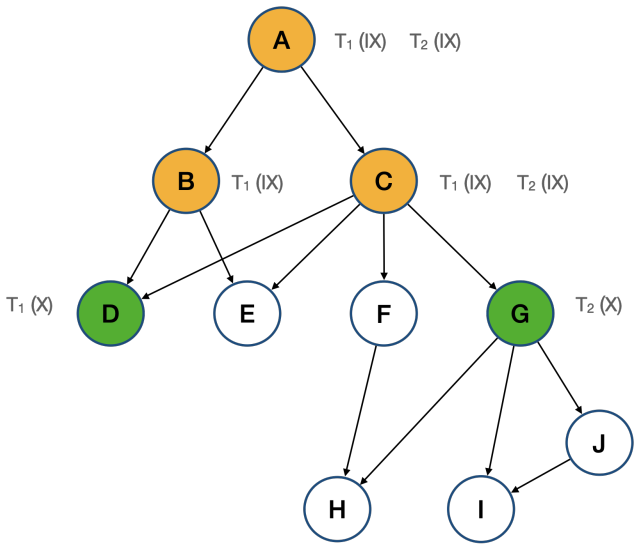


Fig. 3.2: Multi granularity locking via Intention locks.
T_1 takes an exclusive lock on vertex D and T_2 takes an exclusive lock on vertex G

(IX) on the vertices from the root to their lock targets i.e. D and G respectively. Let's assume that T_1 is faster than T_2 and manages to acquire its locks first. T_1 acquires IX on vertices A, B and C, in this order, and acquires an exclusive lock on D.

When T_2 tries to acquire a lock on G, it acquires IX on vertices A and C, in this order, before trying to acquire a lock on G. When T_2 tries to acquire a lock on A, it encounters a pre-existing IX lock acquired by T_1. Since two IX locks are compatible, as seen in Table 3.1, T_2 can acquire the IX lock on A. Similarly, T_2 can acquire the IX lock on C and then proceed to acquire an exclusive lock on G.

By locking the vertices on the path to the target vertex under IS and IX modes, always in a depth-first manner, intention locking protocol correctly identifies lock grain overlaps. However, this process requires multiple traversals from the root to the lock target so that all paths can be locked. In the example of Figure 3.2, T_1 must perform the following two traversals:

- $A \rightarrow B \rightarrow D$
- $A \rightarrow C \rightarrow D$

In a rooted tree, every vertex is reachable from the root via a unique path. This, however, is not the case with DAGs. In DAGs, as the number of paths from the root to a target vertex increases, the number of traversals required to acquire an S or X lock on that vertex also increases because intention locks are to be acquired on all paths to the target vertex. This leads to a significant performance degradation.

Along with the added performance penalty, the order in which multiple paths are traversed needs to be deterministic to prevent deadlocks.

So, while intention locks fulfil requirements $\mathbf{R_{VGuard}}$ and $\mathbf{R_{RGuard}}$, requirement $\mathbf{R_{Conflict}}$ incurs a significant performance penalty.

3.1.3 DomLock

DomLock [KN16] is a MGL technique that uses the concept of dominators to identify lock grains instead of relying on explicitly locking paths that lead to a target. A dominator is a vertex that lies on all the paths from the root to a target vertex v and is thus an effective lock guard for its descendants since locking a dominator is sufficient to lock all the paths to v and the descendants of v.

Definition 2 (Dominator)**.** *A vertex d is a dominator of another vertex v if all the paths from root to v pass through d.*

A vertex can have several dominators in a hierarchy. In order to maximize concurrency, by minimizing the number of locks required for correct exclusive access, DomLock uses the dominator of maximum depth as the lock guard. This dominator is called the immediate dominator.

Definition 3 (Immediate Dominator)**.** *Immediate Dominator: A dominator d is an immediate dominator of another vertex v if there exists no other dominator for v on the paths between d and v.*

This immediate dominator is the deepest vertex that lies on all paths from the root of the hierarchy to a target vertex. Therefore, it is sufficient to lock this immediate dominator to lock all the paths to the target vertex and its descendants since, any traversal to the target vertex or its descendants shall encounter the immediate dominator.

Labelling through numeric intervals In order to identify the dominators of a vertex more efficiently, DomLock uses a labelling scheme that assigns a pair of integers to each vertex. This pair, called the *interval* of the vertex is denoted $I_v = [l_v, r_v]$. This interval I_v subsumes ($\prec$) the intervals of all the descendants of v. Subsumption is defined as follows:

$$u \text{ is a dominator of } v \iff I_u \prec I_v \iff l_v \leq l_u \wedge r_v \geq r_u \tag{3.1}$$

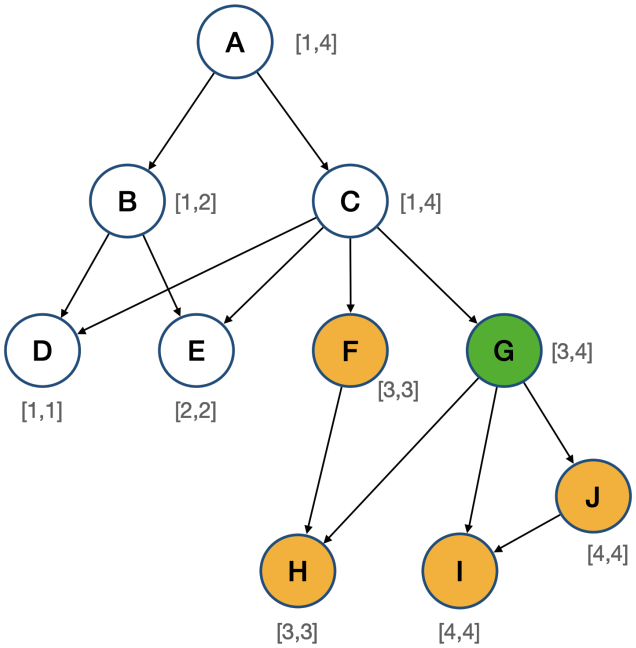


Fig. 3.3: Hierarchy labelled with DomLock intervals and DomLock on G (lock guard) with the grain of the grain of the lock (yellow).

These intervals are computed by performing a post-order traversal of the hierarchy. Consider the example in Figure 3.3. The leaves are labelled with unit intervals. For example, vertex D is labelled with the interval $[1, 1]$ since it is the first vertex in the post-order traversal.

For an internal vertex, the interval is computed via its children's. The l value of an internal vertex v is the minimum of the l values of its children and the r value of an internal vertex v is the maximum of the r values. For example, in Figure 3.3, vertex G has three children H, I and J. The interval of G is $[3, 4]$ since the minimum l value of its children is 3 and the maximum r value of its children is 4. In turn, G $\prec$ {H, I, J}

Lock Grain identification Lock grain identification in DomLock is based on the intervals of the vertices. The grain of a lock guard g contains any vertex v such that v is a descendant of g and $I_g \prec I_v$. In practice, this involves a depth first search from the root of the hierarchy to identify the deepest vertex that satisfies the subsumption property to ensure that the granularity is as small as possible by locking the immediate dominator. For example, in Figure 3.3, the grain of G contains F, H, I and J since the interval of G overlaps with the intervals of F, H, I and J which are $[3, 3]$, $[3, 3]$, $[4, 4]$ and $[4, 4]$ respectively.

Herein lies the first major drawback of DomLock. Due to the subsumption property and the manner in which numeric intervals are assigned to vertices, *False subsumptions* can occur. Subsumption is a property that exists between two vertices related via an ancestor-descendant relationship. A *False subsumption* occurs when the intervals of two vertices overlap, but they are not related by an ancestor-descendant relationship. For example, in Figure 3.3, G is the dominator of F but F is not a descendant of G. Sometimes, due to a false subsumption disjoint subgraphs are locked together. This leads to spurious lock conflicts and consequently, performance degradation.

Label recomputation Another drawback of DomLock is that it does not support dynamic hierarchies without having to relabel a large part of the hierarchy. Since intervals are used to identify lock grains and determine the ancestor-descendant relationship, the intervals of the vertices have to be recomputed when a structural modification occurs.

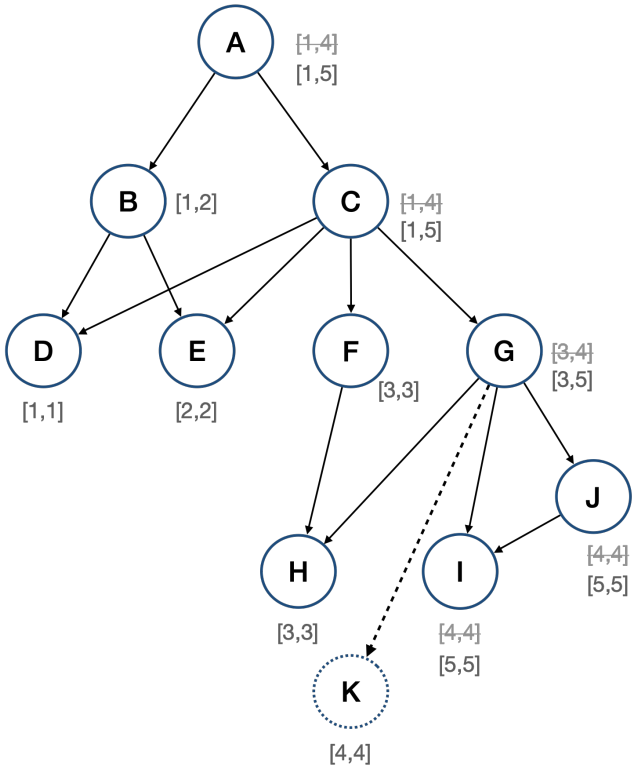


Fig. 3.4: DomLock interval recomputation for a vertex insertion

For example, consider the hierarchy in Figure 3.4. If a vertex K is added as a child of G after H, K gets the interval $[4, 4]$. I and J get the interval $[5, 5]$. Following this,

the interval of G is recomputed as $[3, 5]$. This recomputation has to be done for all the ancestors of G as well, which includes the root of the hierarchy.

An additional drawback is that this recomputation is not parallelizable. Since the interval of the root is recomputed as well, any other operation, read, write or structural modification, has to wait until the recomputation is complete. This is done in order to prevent incorrect dominator identification due to improper intervals. In order to prevent concurrent reads from interfering with the recomputation, a mutex has to be placed on the root of the hierarchy. This lock is held until the recomputation is complete. In dynamic hierarchies, structural modifications can lead to a significant performance penalty due to the lack of parallelism in the relabelling.

While DomLock addresses requirements $\mathbf{R_{VGuard}}$ and $\mathbf{R_{RGuard}}$, false subsumptions and the lack of support for dynamic hierarchies incur significant penalties against requirement $\mathbf{R_{Conflict}}$ and $\mathbf{R_{Metadata}}$ respectively.

3.1.4 Multi Interval DomLock (MID)

Multi Interval DomLock [MAN22] is a successor to DomLock which uses a pair of intervals on each vertex to identify the immediate dominator. MID maintains in addition to the DomLock intervals, another interval computed by a reverse post-order traversal of the hierarchy called the *'DFS-on-image'*. Figure 3.5 shows the MID intervals for a hierarchy.

Labeling through a pair of numeric intervals Each vertex is labelled with two intervals: the *D* interval ($ID_v = [lD_v, rD_v]$) and the *M* interval ($IM = [lM_v, rM_v]$). The only difference between the two intervals is the order of traversal of vertices. D intervals being post-order and M intervals being reverse post-order. Like DomLock, the leaves of the hierarchy are labelled with unit D and M intervals.

For example, in Figure 3.5, vertex H is labelled with a M interval $[2, 2]$ in addition to the D interval $[3, 3]$ since it is the second vertex in the reverse post-order traversal. Again, like DomLock, for an internal vertex, the interval is computed via the intervals of its children. For both, D and M intervals of a vertex, the l value of an internal vertex v is the minimum of the l values of the D intervals its children and the r value of an internal vertex v is the maximum of the r values of the D intervals of its children.

For example, in Figure 3.5, vertex G has three children H, I and J. The D and M intervals of G are $[3,4]$ and $[1,2]$ respectively since the minimum l value of its children is 3 and 1 and the maximum r value of its children is 4 and 2 respectively.

Intervals of a vertex v subsume ($\prec$) the overlapping intervals of all descendants of v. Subsumption is defined as follows:

$$\begin{aligned} u \text{ is a dominator of } v &\iff ID_u \prec ID_v \wedge IM_u \prec IM_v \\ &\iff lD_v \leq lD_u \wedge rD_v \geq rD_u \wedge lM_v \leq lM_u \wedge rM_v \geq rM_u \end{aligned} \tag{3.2}$$

For example, in Figure 3.5, G is a dominator of F, H, I and J since both D and M intervals of G overlap with the respective D and M intervals of F, H, I and J.

Lock Grain identification The property of subsumption is used to identify the lock grain of a guard in MID. However, unlike DomLock, subsumption is tested on both pairs of intervals. While testing subsumption, both D and M intervals are tested for overlap. The grains of two lock guards are disjoint only if both D and M intervals are disjoint. This is done in an effort to reduce the number of false subsumptions. However, as we shall see, MID still suffers from false subsumptions.

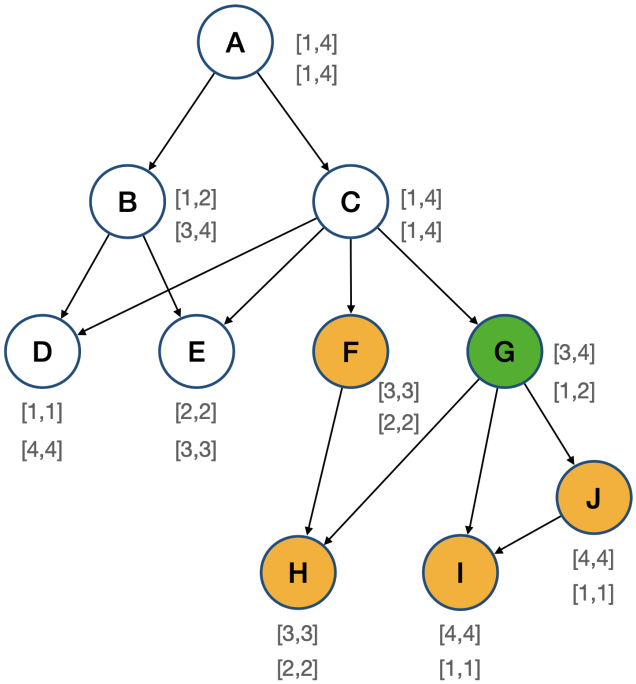


Fig. 3.5: MID labels with lock on guard G with the grain of the lock (yellow)

For example in Figure 3.5, G subsumes F, H, I and J since both intervals of G overlap with the respective intervals of F, H, I and J. So, Even though MID uses two intervals per vertex, there is still a false subsumption. G subsumes F but F is not a descendant of G. A thread that wishes to lock F when G is locked will be blocked due to F and G being included in the same grain even though topologically, they are not. Like DomLock, MID also suffers from poor performance due to such spurious conflicts.

Label recomputation MID encounters double the penalty for recomputing vertex labels in dynamic hierarchies when a structural modification occurs. In DomLock, a single post-order traversal is enough to compute all the intervals for a hierarchy. In MID, two traversals are required to compute the intervals when a structural modification occurs. In certain cases, the intervals of all vertices are recomputed which is extremely expensive for large hierarchies.

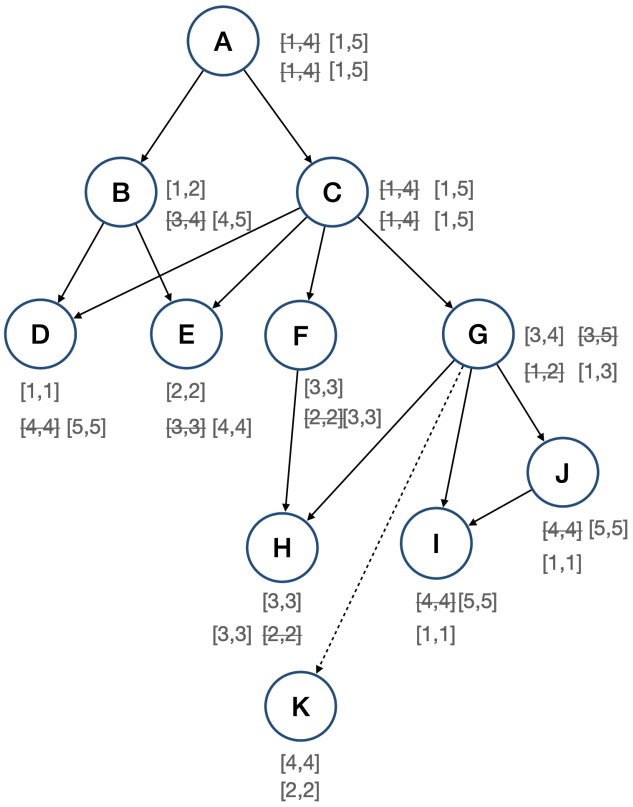


Fig. 3.6: MID interval recomputation for a vertex insertion

For example, when a vertex K is inserted as a child of G as shown in Figure 3.6, At least one interval for every vertex is recomputed. These intervals are then propagated to the root of the hierarchy. In order to prevent concurrent reads from interfering

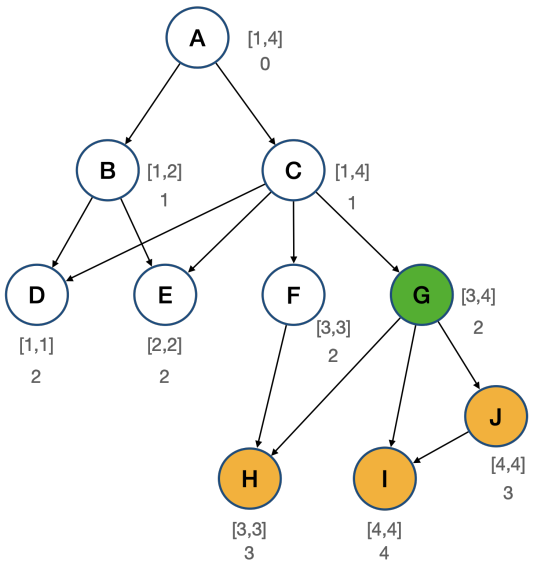


Fig. 3.7: FlexiGran labels and vertex depth with lock on guard G with the grain of the lock (yellow)

with the recomputation, a mutex is acquired on the root of the hierarchy. This lock is held until the recomputation is complete.

MID tries to eliminate false subsumptions present in DomLock but does not do so successfully. The performance penalty for recomputation in dynamic hierarchies is doubled in MID as compared to DomLock. As such, MID fulfills requirements $\mathbf{R_{VGuard}}$ and $\mathbf{R_{RGuard}}$ but incurs even severe penalties against requirements $\mathbf{R_{Conflict}}$ and $\mathbf{R_{Metadata}}$ than DomLock.

3.1.5 Flexible granularity Locking (FlexiGran)

FlexiGran [MAN24] aims to enable the existence of MGL and fixed-grain locks on the same hierarchy. DomLock is used as the MGL locking technique in FlexiGran and the fixed-grain locks are fine-grain locks i.e. every vertex is its own guard. FlexiGran uses the intervals of DomLock as vertex labels and uses vertex depth to determine ancestor-descendent relationships between two identical intervals.

Labelling through numeric intervals In FlexiGran, a post-order traversal is performed to compute the labels of vertices. Like DomLock, the leaves of the hierarchy are labelled with unit intervals and internal vertices are labelled with intervals computed from the intervals of their children. In addition to these intervals, FlexiGran also computes the depth of each vertex in the hierarchy. The depth of a vertex is the

length of the shortest-longest path from the root to the vertex. Figure 3.7 shows the intervals and vertex depths of a hierarchy labelled with FlexiGran.

This shortest-longest path is useful for depth determination in the presence of connected components like cycles. In a cycle, the path is recursive and based on the method of computation, the longest path can be infinite. The shortest-longest path breaks this recursion to a limit. This limit is the length of a path from the root to a vertex which contains the target vertex at most twice, as is the case with cycles.

Lock Grain identification FlexiGran uses the depth information in addition to the intervals to determine the ancestor-descendant relationship between two vertices. If a thread requests a MGL lock on a vertex via flexigran, then a lock is acquired via the DomLock protocol by performing a depth first search to find the immediate dominator of the target vertex. If a thread requests a fine-grain lock, then a reader/writer lock is directly requested on the target.

Since both fine-grain and MGL locks can exist in flexigran, the process of testing lock conflicts involves multiple steps.

Table 3.2 shows the compatibility matrix for FlexiGran locks. **F** and **H** indicate the kind of lock acquired, fine-grain lock and multi-granularity lock respectively. **R** and **W** indicate the lock mode. So, **FR** is a fine-grain read lock.

When checking if two MGL are compatible, the intervals of their guards are tested for overlap. Disjoint intervals indicate that the two locks are DomLock compatible. When checking two fine-grain locks for conflict, the lock guards for the lock requests are compared. If two fine-grain lock requests are on the same vertex, then they are in conflict if at-least one of the lock requests is for a write lock.

	Lock On Ancestor				Lock On Descendent			
Lock Mode	**FR**	**FW**	**HR**	**HW**	**FR**	**FW**	**HR**	**HW**
FR	Y	Y	Y	N	Y	Y	Y	Y
FW	Y	Y	N	N	Y	Y	Y	Y
HR	Y	Y	Y	N	Y	N	Y	N
HW	Y	Y	N	N	N	N	N	N

Tab. 3.2: Flexigran compatibility matrix showing the protocol for the co-existence of hierarchical and fine-grained locks in a system. **F**: Fine-grained, **H**: Hierarchical, **R**:Read, **W**: Write

To test the compatibility of a MGL lock with a fine-grain lock, it is required to perform an ancestor-descendant relationship check between the guards of the MGL lock and

the fine-grain lock. To do so, a traversal is performed and if the fine-grain lock guard is reachable from the MGL lock guard, then the two locks are incompatible since they are related by an ancestor-descendant relationship and are in the same MGL grain. For example, in Figure 3.7, a hierarchical lock is acquired on G. This lock guards vertices H, I and J. Unlike DomLock and MID, FlexiGran does not subsume F since F is not a descendant of G since their depths are the same. In this situation, if a fine-grain lock is requested on F, it would be compatible with the hierarchical lock on G since there is no path from G to F or vice-versa.

Label recomputation Like DomLock, structural modifications in FlexiGran require recomputation of the intervals of the vertices. A single post-order traversal is enough to recompute the intervals and depths of vertices. Since the root might be involved in the structural modification, a lock is placed on the root to prevent concurrent reads from interfering with the recomputation. This restricts concurrency and leads to a degradation in performance.

As such, FlexiGran fulfills requirements $\mathbf{R_{VGuard}}$ and $\mathbf{R_{RGuard}}$ and incurs a significant performance penalty against requirement $\mathbf{R_{Conflict}}$ due to the expensive compatibility checks required to detect conflicts between MGL and fine-grain locks. FlexiGran also incurs a performance penalty against requirement $\mathbf{R_{Metadata}}$ due to the non-parallel recomputation of intervals in dynamic hierarchies.

3.2 Trade-offs between state-of-the-art techniques

Recall the four primary requirements identified for MGL:

$\mathbf{R_{VGuard}}$ Identifying a lock guard for every vertex of a hierarchy.

$\mathbf{R_{RGuard}}$ Finding an appropriate, optimal lock guard for a request.

$\mathbf{R_{Conflict}}$ Efficiently Detecting conflicts between locks

$\mathbf{R_{Metadata}}$ Housekeeping the metadata required to implement the locking protocol.

Level-based locks are inefficient and only fulfil requirement $\mathbf{R_{VGuard}}$. MGL based on traversals like Intention Lock, fulfills requirements $\mathbf{R_{VGuard}}$ and $\mathbf{R_{RGuard}}$ but incurs a significant performance penalty against requirement $\mathbf{R_{Conflict}}$ due to the sheer number of traversals required to lock a vertex.

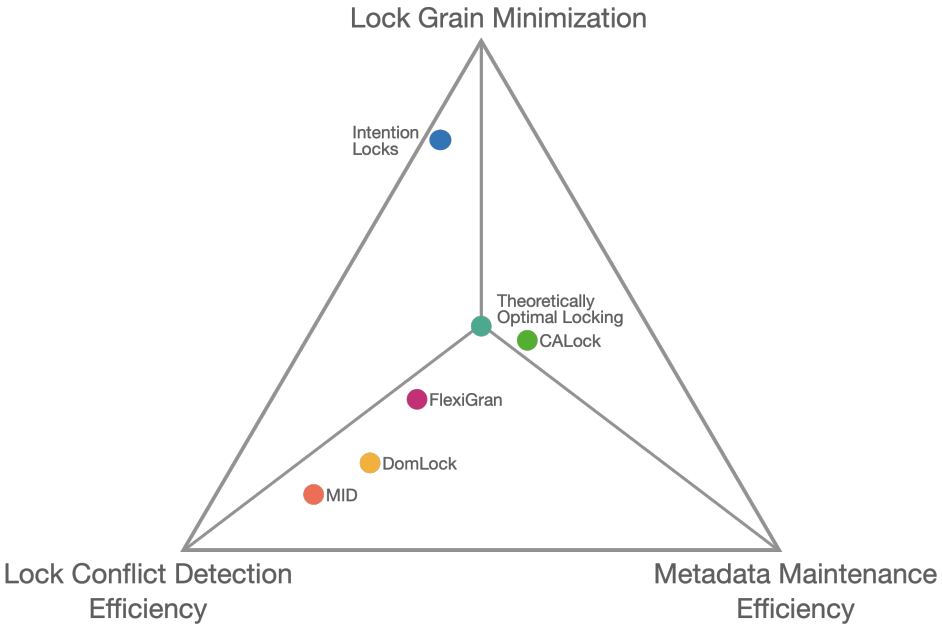


Fig. 3.8: Trade-offs in MGL techniques

Algorithm	R_{VGuard}	R_{RGuard}	$R_{Conflict}$	$R_{Metadata}$
Level-based locks	Y	N	N	N
Intention Lock	Y	Y	N	N
DomLock	Y	Y	N	N
MID	Y	Y	N	N
FlexiGran	Y	Y	N	N
CALock	Y	Y	Y	Y

Tab. 3.3: Comparison of MGL techniques against requirements

Label-based techniques like DomLock, MID and FlexiGran also fulfil requirements $\mathbf{R_{VGuard}}$ and $\mathbf{R_{RGuard}}$ but incur a performance penalty against requirement $\mathbf{R_{Conflict}}$ due to false subsumptions. In addition, due to the metadata required to implement the locking protocol, these techniques incur a performance penalty against requirement $\mathbf{R_{Metadata}}$ as well when used in dynamic hierarchies. FlexiGran that implements MGL and fine-grain locks suffers from poor performance due to the expensive compatibility checks and label recomputation.

Figure 3.8 and Table 3.3 shows the trade-offs in the different MGL techniques. Intention locks offer the most optimal lock grains at the cost of lock conflict detection efficiency. DomLock and MID offer efficient conflict detection at the cost of metadata management. FlexiGran offers better lock grains but incurs a performance penalty

due to the expensive lock conflict detection. None of the techniques balances all the requirements to come close to an optimal MGL protocol.

3.3 Improving efficiency of MGL techniques

The locking techniques discussed in chapter 3 are based on a broad set of assumptions. These assumptions are based either on the topology of the graph or the nature of the workload. Most MGL techniques enforce an ordering of vertices either via reachability thereby utilizing paths to vertices or by a static ordering that ignores the topology of the hierarchy.

3.3.1 Path based techniques

Path based techniques like lock coupling [BS77] and intention locks [Gra+75] utilize paths from the root of a graph to the lock target. Lock coupling uses a pair of locks that are acquired one after the other on a path to the vertex to prevent concurrent access to the vertex. While intention locking places a set of locks on the path to the vertex to prevent concurrent access. These techniques are efficient for trees and tree-like structures where the path to a vertex from the root is unique. However, generalizing these techniques to graphs or even hierarchies is not appropriate. As we will see in the 8, the performance of Intention locks degrades significantly when locking over hierarchies.

3.3.2 Label based techniques

Other MGL techniques make use of labelling strategies which identify an ordering of vertices which is later used by the locking protocol to identify the lock guard and grains. Techniques like DomLock, MID and FlexiGran fall into this category. By using a predefined ordering, for example a post-order, these techniques can very efficiently identify the lock guard and grain. However, the topological detail of the graph is lost.

Other techniques like Toggle [KN19] circumvent locking all together and use thread schedulers to prevent race conditions. While scheduling techniques are efficient, they involve redesigning an application and are not readily useable for general developers.

3.3.3 Unified Path and Label based techniques

Unified techniques for labelling are explored in literature in the domain of metadata management for large scale, efficient searching applications. The Dewey Decimal System [Swe83] is the earliest example of a path based labelling technique used to identify elements of a hierarchy based on their subclassification. Other path based techniques use similar expensive representations of paths.

A path based labelling technique that preserves the topological ordering of vertices in the labels and facilitates efficient locking would be an ideal solution. However, the additional metadata required to implement this unified technique should not be prohibitively expensive. DomLock and its successors sacrifice labelling performance for locking performance to an extent that makes them unusable for hierarchies that undergo structural modifications. Our labelling scheme balances maintaining both topological information of the hierarchy while being efficient for locking and also being efficient for relabelling.

3.4 CALock: A topological multi-granularity locking technique

We propose a new labeling scheme based on path properties, that supports multi-granularity locking. *Common-Ancestor lock* (*CALock*) is based on a labeling that efficiently computes the closest common ancestor of a set of vertices in a rooted directed graph. By choosing the closest common ancestor of a set of lock targets as their guard, CALock minimizes grain size. In using paths, by avoiding a fixed ordering of vertices, CALock circumvents expensive relabelling while providing the same locking guarantees as other MGL techniques. The label of a vertex in the graph is a set of its common ancestors, computed recursively via breadth-first traversal which we discuss in chapter 5.

CALock: Topological labelling

4

Contents

CALock uses the topological information of a graph to determine the optimal locking grain for a lock request. It does so by using the paths to a vertex from the root of the hierarchy and finding the set of vertices that are common on all these paths. This path based grain identification can become expensive for systems with high lock throughput. To avoid grain computation becoming a bottleneck, we develop a labelling scheme that eficiently and effectively allows threads to determine an optimal locking grain for their lock requests without needing to traverse the hierarchy.

In this chapter, we present the formal definitions and theoretical underpinnings of the labelling scheme of CALock. We begin by introducing the basic concepts of graphs, paths, and vertex relationships. Next we discuss the bounds of commonality between vertices in a graph and how they can be used to determine an optimal locking grain.n We formulate the optimal locking grain problem mathematically and prove its correctness. Finally we present the CALock labelling function and discuss its implementation over different use cases.

4.1 Graphs, Paths and Vertex Relationships

Let $G = (V, E)$ be a directed graph. V is its set of vertices, connected by directed edges in the set $E \subseteq V \times V$. A rooted graph has a distinguished vertex $r \in V$ such that all other vertices are reachable from r.

A pair of vertices (u, v) can be connected by a sequence of edges, called the path between u and v, noted p. The set of vertices of p is noted $\mathcal{V}(p)$. The length l_p of this path is the size of $\mathcal{V}(p)$ i.e. $l_p = |\mathcal{V}(p)|$.

In the general case, several such paths may exist between a pair of vertices. The set of paths between u and v is noted $\mathcal{P}_{(u,v)}$. The depth $\delta(v)$ of a vertex v is the length of the shortest path in the set $\mathcal{P}_{(r,v)}$.

Definition 4 (Ancestor and Descendent). *An ancestor of a vertex u is a vertex $v \in G$ that lies on a path from r to u. The vertex u is then called the descendant of vertex v. The set of ancestors is given by the relation $A(u)$.*

$$A(u) = \{v \in V | \exists p \in \mathcal{P}_{(r,u)}, v \in p\}$$

Definition 5 (Guarding Ancestor). *A guarding ancestor of a vertex u is a vertex $v \in G$ that lies on all paths from r to u. The set of guarding ancestors of u is noted L_u. This set is given by the relation $GA(u)$.*

$$GA(u) = \{v \in A(u) | \forall p \in \mathcal{P}_{(r,u)} \Rightarrow v \in p\}$$

Definition 6 (Lowest guarding Ancestor). *The lowest guarding ancestor $LGA(u)$ of a vertex u is a guarding ancestor of u with the maximum depth.*

Definition 7 (LGA-Tree). *The LGA-tree T_G of G is an auxiliary tree that has the same vertices as V and whose edges are defined such that the parent vertex of $v \neq r$ is the LGA of v in G.*

Definition 8 (Common Ancestor). *A common ancestor (CA) of two vertices u and v is a vertex c that is an ancestor of both u and v. The set of common ancestors is given by the relation $CA(u, v)$.*

$$CA(u, v) = \{c \in V | c \in A(u), c \in A(v)\}$$

Definition 9 (Lowest Common Ancestor). *The lowest common ancestor $LCA(u, v)$ of two vertices u and v is a common ancestor of u and v with the maximum depth.*

Term	Meaning
G	Graph
V	Vertices of G
E	Edges of G
r	Root of G
u, v, w	Vertices
Q	Set of vertices
$\delta(v)$	Depth of vertex v
$GA(u)$	Guarding ancestors of u
$CA(u, v)$	Common ancestors of u and v
$LGA(u)$	Lowest guarding ancestor of u
T_G	LGA tree for a graph G
$LCA(u, v)$	Lowest common ancestor of u and v
$LGCA(Q)$	Lowest guarding common ancestor of Q
L_u	Label of vertex u

Tab. 4.1: Terms used in the definitions

4.2 Lowest Guarding Common Ancestor

The Lowest Guarding Common Ancestor $LGCA(Q)$ of a set of vertices Q is the deepest vertex that is both a guarding ancestor and a common ancestor of all vertices in Q. Fischer et. al [FH10] derive the following relationships between the LGA and the LGCA of vertices in a rooted DAG.

Lemma 1. *Let G be a DAG, rooted at r, and T_G its corresponding LGA-tree. Further, let v, w ∈ V be two arbitrary vertices in G. Then* $LGCA_G(v, w) = LCA_{T_G}(v, w)$

Lemma 2. *For a vertex* $v \in G : v \neq r, LGA_G(v) = LGCA_G(parents_G(v))$

Definition 10. $LGCA_G(w_1, ..., w_k) = LGCA_G(w_1, LGCA_G(w_2, ..., w_k))$

Definition 11. $LCA_{T_G}(u_1, u_2, ..., u_n) = LCA_{T_G}(u_1, LCA_{T_G}(u_2, ..., u_n))$

4.3 Characteristic sets: Sets of Guarding ancestors

We take inspiration from the LGA-tree T_G to label the vertices of a rooted DAG *G*. A vertex u is labelled with an ordered set containing the vertices on the path from the

root of T_G to u. L_u contains the set of guarding ancestors of u sorted by their depth in T_G. Since T_G is a tree, this path is unique and so is the label L_u.

In order to compute the label of a vertex $u \in G$ without needing to compute the LGA-tree T_G, we take the set intersection of the labels of u's ancestors in G. The $LGCA$ of a set of vertices is the last vertex in an ordered intersection of the sets of guarding ancestors of those vertices. We have the following theorem:

Theorem 1. *$LGCA_G(v, w)$ is the vertex of the maximum depth in $L_v \cap L_w$*

Proof. Let $l = LGCA_G(v, w)$ be the LGCA of two vertices $\{v, w\}$, We need to show that l is the deepest vertex in $L_v \cap L_w$.

Assume that there is a vertex $l' \neq l$ that lies on all paths from l to v and l to w in G.

Since l' lies on the paths in the sets $\mathcal{P}_{(l,v)}$ and $\mathcal{P}_{(l,w)}$, inductively, l' also lies on all the paths from the root (r) of the graph to the vertices v and w. If l' lies on these paths then it is a guarding ancestor of v and w and should be present in their characteristic sets.

The following holds true: $(l' \in L_v) \wedge (l' \in L_w) \equiv l' \in L_v \cap L_w$.

Since l' lies on the paths to v and w after l, by the definition of depth $\delta(v)$ of a vertex, $\delta(l') > \delta(l)$.

Now we have two conclusions, $l' \in L_v \cap L_w$ and $\delta(l') > \delta(l)$.

Since l' is deeper than l, it should be the deepest member of $L_v \cap L_w$. Which means, l' is the LGCA of v and w. But this contradicts our original assumption that l is the LGCA of v and w. This means our assumptions on l' contradict the definition of LGCA. Either $l' = l$ or l' is the LGCA and not l. □

4.4 CALock labelling scheme

CALock label for a vertex is the characteristic set. In the implementation, we use this set, computed via a recursive relation to determine the lock guard for a set of lock targets. This recursive relation is computed using the definitions and lemmas from Section 4.2. In this section, we incrementally derive the relation and show its robustness against different topological extremes of like cycles and connected components. An implementation of this function is shown in Algorithm 1.

4.4.1 Recursive labelling function

We combine theorem 1 with the definitions and lemmas from Section 4.2 to derive a recursive function that we use to label the vertices in a graph. To this end, Lemma 2 can be rewritten using Definition 10 as follows:

$$LGA_G(v) = LGCA_G(p_1, LGCA_G(p_2, ..., p_k)) \\ \text{where } p_1, p_2, ...p_k \text{ are the parents of } v \tag{4.1}$$

Combining equations 4.1 and Theorem 1; $LGA_G(v)$ is the deepest vertex in $L_{p_1} \cap L_{p_2} \cap ... \cap L_{p_k}$ where $p_1, p_2, ...p_k$ are the parents of v.

Therefore, the sets of guarding ancestors of the parents of v, are enough to compute the lowest guarding ancestor of v. We use this property to recursively compute the labels of the vertices in a graph using the following function.

$$L_v = \begin{cases} \{v\} & \text{v is the root} \\ \{v\} \cup \{\cap_{u \in parents(v)} L_u\} & \text{otherwise} \end{cases} \tag{4.2}$$

4.4.2 Labelling and relabelling a graph

The recursive labelling function derived in Section 4.4.1 is used to compute the labels of each vertex in a hierarchy. An implementation of this function is shown in Algorithm 1. This function is robust to different topological extremes of the graph. Not only can it be used to label acylic graphs but also, graphs with strongly connected components. We discuss the labelling and relabelling of acyclic graphs and strongly connected components in the following sections.

Labelling an acyclic graph

Labelling starts at the root of the graph with the function `AssignLabel(r)`. It uses Relation 4.2 implemented in lines 12 - 17 in Algorithm 1 to assign the labels. The root vertex does not have parents, so the label of a root is the set containing the ID of the root itself (Lines 8 - 11). For example, in Figure 4.1 vertex A has the label $\{A\}$.

Then, the children of the vertex v are explored via a breadth-first traversal over the graph. Using the labels of the parents, the label of the child is computed through

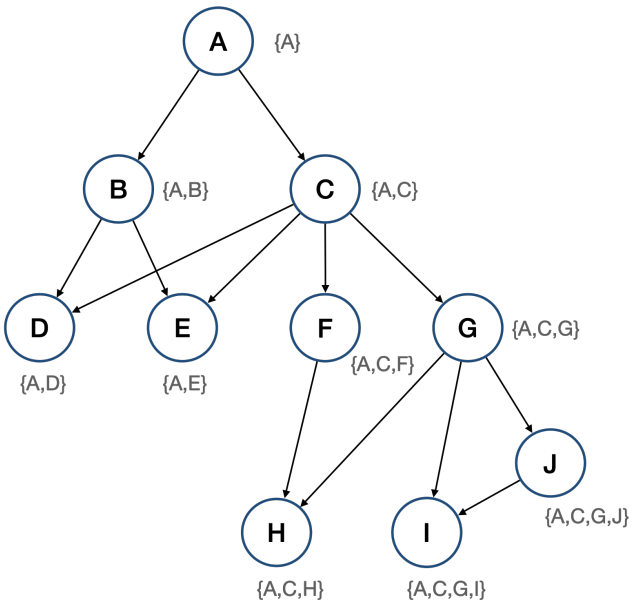


Fig. 4.1: CALock labels on a hierarchy

Relation 4.2. For example, in Figure 4.1, the label of vertex B is $\{A, B\}$ since it has only one parent. Vertex E has two parents which have the labels $\{A, B\}$ and $\{A, C\}$ respectively. Their set intersection is $\{A\}$. Using Relation 4.2, the label of E is $\{A, E\}$. The graph is explored and labelled until the recursion reaches a fix-point and terminates (line 15).

In acyclic graphs, this recursion reaches a fix-point when all the paths to the leaf vertices have been explored. By exploring the vertices in a depth-first manner, we ensure that the label of a vertex is computed only after the labels of all its parents have been computed so-as-to avoid recomputation as much as possible.

Labelling a strongly connected component

Strongly connected components are subgraphs that exhibit special properties in terms of connectivity of vertices. A strongly connected component is a maximal sub-graph of a directed graph where every vertex is reachable from every other. In Figure 4.2, the cyan vertices form a strongly connected component.

Algorithm 1 Labelling the graph

```
1:  procedure AssignLabels(v)
2:      queue.push(v)
3:      while queue.hasNext( ) do
4:          v ← queue.Next( )
5:          BFLabelA(v, queue)

6:  procedure BFLabelA(v, queue)
7:      C ←children(v)
8:      if parents(v) = ∅ then
9:          v.label ← {v}
10:         queue.push( C )
11:         return
12:     P ←parents(v)
13:     tempLabel ←Intersection(P.labels)
14:     tempLabel.append(v)
15:     if v.label ≠ tempLabel then
16:         v.label ← tempLabel
17:         queue.push( C )
```

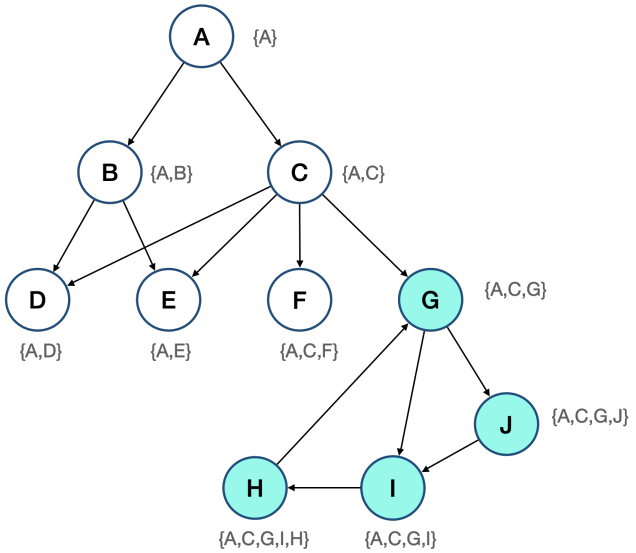


Fig. 4.2: CALock labels for a hierarchy containing strongly connected component (cyan)

Theorem 1 assumes that the graph does not contain strongly connected components. In real-life applications, maintaining this constraint is difficult. One approach to handling strongly connected components is to eliminate them by contracting the vertices in the component to a single vertex [Sha81; Tar72; CM96; Gra+08]. By doing so, we treat the strongly connected component as a single vertex in the graph. However, this approach alters the graph semantically which is not desirable in

applications that store data in the vertices of the graph. By contracting the vertices, we lose the information stored in the vertices of the strongly connected component which is not acceptable. With the recursive relation 4.2 we do not need to eliminate strongly connected components to label the vertices of a graph.

Assume that $N \subseteq V$ is a set of vertices of a graph that are strongly connected. Since every vertex in N is reachable from every other vertex in N, the path set $\mathcal{P}_{(u,v)}$ for any pair of vertices $u, v \in N$ contains the same vertices and hence, every vertex in N has the same set of guarding ancestors. `BFLabel` recurses over all vertices in N until the paths labels of vertices in N do not change any more (line 15). For example, in Figure 4.2, the vertices G, H, I, J form a strongly connected component.

Relabelling a graph

The topology of a graph can change due to *structural modifications* which add or remove vertices and edges from a graph. Such modifications alter the topology of the graph and hence the paths that lead to a vertex from the root. When a structural modification occurs, the labels of the vertices need to be recomputed.

Unlike the other state-of-the-art techniques, which relabel the graph by performing the same post-order traversal from the left-most leaf, in CALock relabelling begins at the vertex affected by the structural modification via `AssignLabel(v)` function. Unlike DomLock, MID and FlexiGran, CALock does not relabel the entire hierarchy when a structural modification occurs. It only relabels the affected vertices and their children. This is useful to not only parallelize structural modifications but also to reduce the overhead of relabelling the entire hierarchy.

In the same fashion as the initial labelling, the parents of v are used to compute its label and recursively of its children. Relabelling terminates when it reaches a fix-point (line 15).

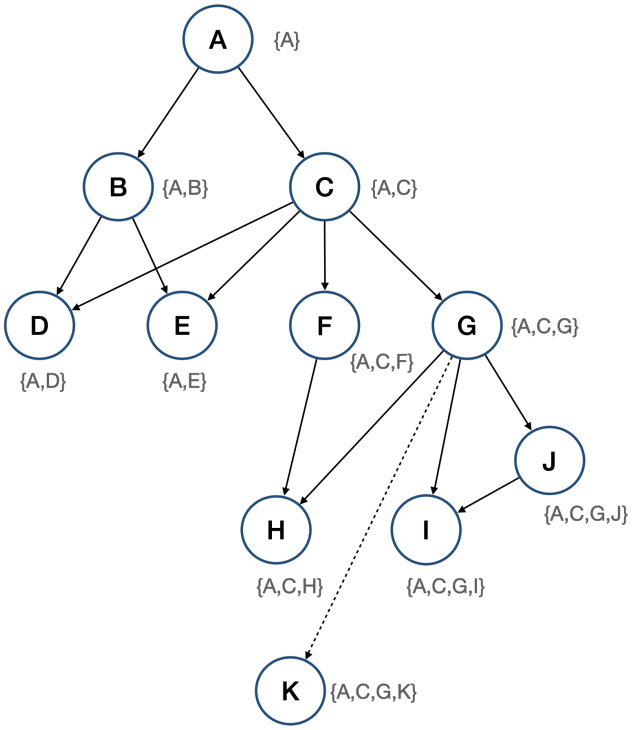


Fig. 4.3: CALock labels for a hierarchy with structural modifications

Consider the example in Figure 4.3. Let us insert a new vertex K as a child of G. The label of K is the intersection of the labels of its parents i.e. $\{A, C, G, K\}$. Since K is a leaf vertex, the recursion terminates. No other vertex in the hierarchy is relabelled. This allows CALock to efficiently handle structural modifications in the hierarchy by eliminating the need of a mutex for relabelling the entire hierarchy and by parallelizing the relabelling process.

5 CALock: Multi-Granularity locking and conflict detection

Contents

In Chapter 4, we discussed the formal theory behind CALock. A recursive function `BFLabel()` was introduced to assign labels to the vertices in the graph. These labels are used to identify the lock grain of a lock request. Labelling is performed as an initialization step and is not required during runtime. Once a hierarchy is labelled, threads can utilize the labels to lock the graph at different granularities. From the locking steps discussed in Chapter 2, Section 2.2, we know that a thread wishing to lock a set of vertices must first create a lock request and then check for conflicts with other lock requests. Once the lock request is granted, a thread can proceed to its critical section, following which, it releases the lock.

In CALock, a lock request contains information derived from the labels of the vertices in the lock request which is used to identify the lock guard. Lock requests are added to a concurrent data structure called a lock pool. This pool is used to detect conflicts between lock requests. After adding a lock request to the pool, a thread checks for conflicts with other lock requests in the pool. If no conflict is detected, the thread proceeds to its critical section. This process is shown in Listing 2. The main steps are:

1. **Lock grain identification**: A thread uses the labels of the vertices in its lock request to identify the grain it wants to lock. The thread then determines the lock guard associated with this grain and locks it.
2. **Lock request creation**: The thread proceeds to create a lock request for the guard and adds it to a lock pool.

3. **Lock conflict detection**: After adding its request, the thread evaluates potential conflicts by comparing its request with other entries in the pool. In CALock, two conditions indicate a lock conflict. If both conditions hold for a pair of lock requests, they are in conflict.

4. **Operation execution**: If no conflict is detected, the thread proceeds to execute its operation.

5. **Lock release**: After the operation is complete, the thread releases the lock on the grain.

In this chapter, we look at each of these steps in detail with examples and corner cases. We also discuss the implementation of CALock

5.1 Lock grain identification

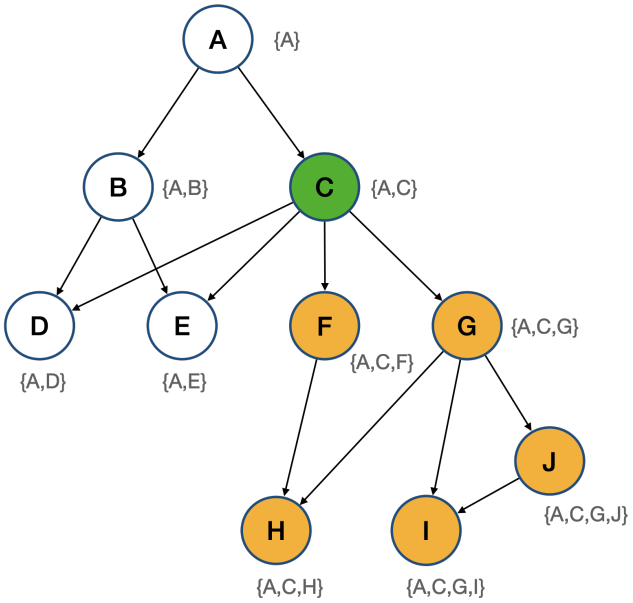
The lock grain of a set of target vertices in CALock is the smallest sub-graph that contains all the targets, such that all paths to these targets can be guarded against concurrent conflicting access. The guard corresponding to a lock request is the *Lowest Guarding Common Ancestor* (LGCA) of the targets. As shown by Theorem 1, the LGCA is the deepest vertex in the graph that is an ancestor of all the lock targets. It is present on all the paths that lead to the vertices to be locked and guards them.

For example, in Figure 5.1, Let us lock target vertices H and J. The intersection of their labels is $\{A, C, H\} \cap \{A, C, G, J\} = \{A, C\}$ in which is C is the deepest vertex. The thread then proceeds to acquire a lock on C.

5.2 Lock requests and the lock pool

A lock acquire request in CALock contains data that identifies its grain without the need to traverse the graph. The lock request contains the following information.

- **Thread ID**: The unique identifier of the thread making the lock request. This helps distinguish lock requests originating from different threads in the system.
- **Guard ID**:The unique identifier of the lock guard, which corresponds to the vertex being locked to control access to a specific portion of the hierarchical data.



Lock on C: {A,C} with grain: {F, G, H, I, J}

Fig. 5.1: CALock labels

- **Guard label**: The label of the lock guard vertex.
- **Sequence number**: A unique number assigned by the lock manager to the lock request when it is added to the lock pool. This sequence number helps in ordering lock requests and ensuring fairness in lock acquisition.
- **Activity Indicator**: A flag or marker used by other threads to monitor the status of this lock request. When a conflict occurs, threads use this indicator to determine whether they need to wait for the current lock request to complete.
- **Lock mode**: Specifies the type of access the thread is requesting for the resource. This can be either read (shared access, allowing concurrent reads) or write (exclusive access, preventing other reads or writes).

These fields will be used to detect conflicts with other lock requests. As soon as the thread identifies the LGCA, it issues a lock request by adding its request to a pool used to identify conflicts.

Lock Pool

The lock pool in CALock is a central data structure used to manage and coordinate lock requests from multiple threads. It is essentially an array where each entry corresponds to a thread, and the state of each thread's entry reflects whether it is

holding a lock or waiting for one. When a thread requests a lock, it is assigned a sequence number and sets its activity indicator to true, signaling that it is actively using the lock. The lock pool helps detect conflicts between lock requests, and CALock uses the sequence numbers to ensure fairness by granting locks in a first-come, first-served manner. Threads check for conflicts and, if necessary, block until the active thread releases the lock, maintaining synchronization and preventing thread starvation.

The procedure of acquiring a lock is specified in listing **??**. Initially and when the thread is not holding a lock, its entry is `NULL`. When a thread arrives in the lock pool, it is given a sequence number (line 6) and it sets its activity indicator to true (line 7) indicating that it is active and using the lock. In order to remain fair and prevent thread starvation, CALock uses sequence numbers to resolve conflicts. The *sequence numbers* are an indication of the order in which the locks are requested. In the presence of conflicts, threads are granted locks in a *first come first serve* order. Other threads check for conflicts with this thread and block until the activity indicator is set to false.

An example of a lock pool containing 3 requests from threads T_1, T_2 and T_7 is shown in Figure 5.2.

Thread ID	Guard ID	Guard Label	Seq. No.	Active	Mode
1	G	{A,C,G}	2	true	read
2	C	{A,C}	3	true	write
7	B	{A,B}	1	true	read

Tab. 5.1: Lock requests in the lock pool

The order in which threads arrive is important to ensure fairness. To prevent a race condition where a thread with a higher sequence number and conflicting request is granted a lock before a thread with a lower sequence number could add its request to the pool, the assignment of a sequence number and the addition of the lock request to the pool is done atomically.

Once a lock request is added to the pool, it is guaranteed to be granted in first come first served order. After adding its request to the pool, a thread proceeds to check for conflicts with other requests in the pool.

5.3 Identifying lock conflicts

In MGL on graphs, conflict detection involves checking two conditions to ensure correctness. These conditions arise from the existence of locks at different granularities (sub-graphs).

- **Grain Overlap**: A grain overlap conflict occurs when a writer attempts to acquire a lock at a finer granularity (e.g., a vertex or edge) while another thread holds a lock at a coarser granularity (e.g., a sub-graph or the entire graph). In this case, the finer-grained lock must be exclusive, meaning no conflicting lock can exist at a higher level. For example, if a writer locks a vertex, no lock at the sub-graph level can coexist with it, as the vertex is part of a locked sub-graph. This is checked using the guard ID and guard label of the requests. For two lock requests R_1 and R_2, if the guard ID of $R1$ is present in the guard label of R_2 or vice-versa, then the grains overlap.

- **Mode Conflict**: A mode conflict occurs when the grains protected by two lock guards overlap in such a way that a writer's lock would violate exclusivity. Specifically, if a writer attempts to lock a vertex or edge, and another lock already protects a grain containing said vertex/edge that overlaps with the writer's lock, the two locks cannot coexist. Ensuring non overlapping grains guarantees that no other transaction modifies or reads the data protected by the writer's lock during its operation. This is checked by comparing the lock modes in the requests. If one request is a write lock and the other is a read lock, then a mode conflict exists.

A thread checks these conditions (line 11) for its request against existing locks by iterating over the pool from left to right (Listing 2 line 10) ensuring deterministic conflict detection. This ordering guarantees fairness and prevents starvation by eliminating the risk of priority inversion.

	$rl_i(x)$	$wl_i(x)$
$rl_j(y)$	✓	✤
$wl_j(y)$	✤	✤

✓ compatible.
✤ compatible iff grain of x is disjoint from grain of y.

Tab. 5.2: Lock compatibilities between read (rl) and write (wl) locks requested by threads $i \neq j$ on vertices x and y.

A requesting thread checks conflicts with existing requests in the lock pool and blocks at the first conflict it encounters. The compatibility matrix for CALock is shown in Table 5.2 Once this conflict is resolved, the iterating thread proceeds checking

T_1	T_2	T_3	T_4	T_5	T_6	T_7
G,read,2,{A,C,G}	C,write,3,{A,C}	NULL	NULL	NULL	NULL	B,read,1,{A,B}

Fig. 5.2: Lock pool containing details of locks held by threads

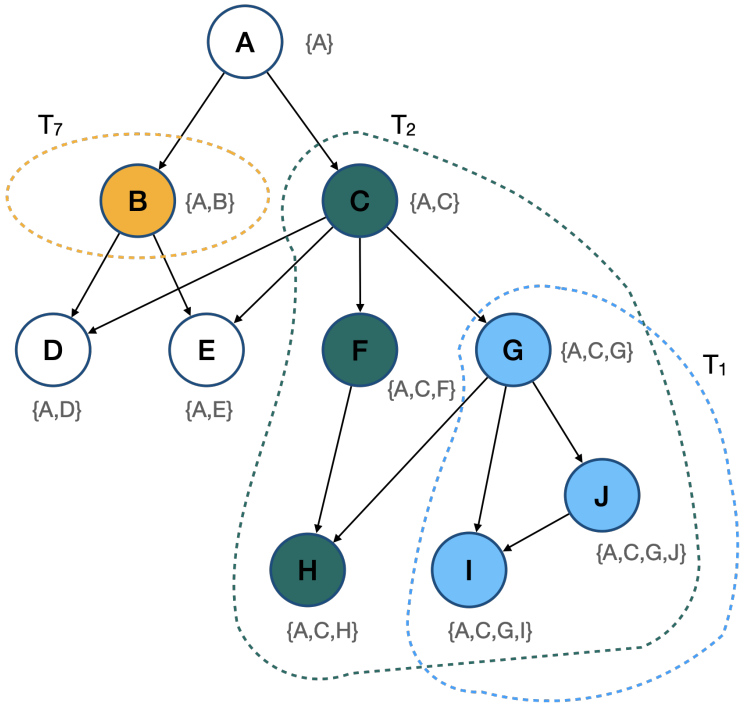


Fig. 5.3: Lock grains on the hierarchy for the locks requested by the threads in the lock pool(Figure 5.3)

conflicts with the remaining threads in the pool. Once it reaches the end of the lock pool, it is guaranteed to either have no conflicts or have priority over all conflicts because its sequence number would be the lowest. We discuss this more in Chapter 6.

When requesting a lock, a thread T can be blocked for a maximum of n turns where n is the maximum number of threads. After at most n blocks, the sequence number of T will be the lowest among all requests in the lock pool. As such, it will be allowed to proceed since it will be the oldest request in the pool and should be served first according to the first come first served ordering. This is to prevent starvation.

Figure 5.2 shows the snapshot of a lock pool. Figure 5.3 shows the grains on a hierarchy for the locks requested by threads in Figure 5.2. This pool contains requests from three threads T_1, T_2 and T_7. Suppose, T_7 arrives first and is assigned sequence number 1. T_1 and T_2 arrive later and have sequence numbers 2 and 3

respectively. T_1 and T_2 have overlapping grains because T_2 is requesting a lock on C which overlaps with the grain locked by T_1. This is detected by checking if C is present in the label of G i.e. $C \in \{A, C, G\}$. T_1 is granted a lock since it arrived first, and T_2 has to wait for T_1 to release the lock. T_7 does not conflict with any request and acquires a lock on B.

Algorithm 2 Lock acquisition request in the lock pool

```
1: GSeq : Global sequence number for lock requests
2: Mutex : Mutex used when adding requests to the lock pool
3: Condition : Atomic boolean value used by waiting threads when in conflict

4: procedure LOCK(req, threadID)
5:     LOCK(Mutex)
6:     req.Seq ← Gseq + +
7:     req.condition.TESTANDSET(true)
8:     Pool[threadId] ← req
9:     UNLOCK(Mutex)
10:    for all lock ∈ Pool \ threadID do
11:        if lock ≠ NULL
12:            ∧(req.HASRWCONFLICT(lock))
13:            ∧(lock.guardID ∈ req.label ∨ req.guardID ∈ lock.label)
14:            ∧(req.Seq > lock.Seq) then
15:            thread.BLOCKANDWAIT(lock.condition)
16:    return true

17: procedure UNLOCK(lock)
18:    lockPool.REMOVE(lock)
19:    lock.condition.CLEAR( )
20:    lock.condition.NOTIFY_ALL( )
```

5.4 Structural modifications, locking and relabelling

In the earlier sections, we discussed data updates. The scope of a data modification is limited to vertices. As such, a data modification requires locking only vertices. Now, we study structural modifications. Structural modifications involve changes to the graph's topology, such as adding or deleting vertices and edges. These modifications require careful handling to ensure consistency. In this section, we will explore the mechanisms for locking and relabelling in CALock when structural changes occur.

The same CALock algorithm, shown in Figure 2 can be utilized for dynamic graphs that change at runtime. In order to perform a structural modification, a thread

acquires a write lock on the LGCA of the affected vertices. For example, when adding an edge between vertices u and v, a thread acquires a write lock on the LGCA of u and v.

The lock protocol for a structural modification involves the following steps:

1. **Lock grain identification**: Identifying the grain of the vertices being modified. This is done by finding the LGCA of the vertices.
2. **Lock request creation**: A lock request is created with the identified grain and *write* mode.
3. **Lock conflict detection**: The thread checks for mode and grain conflicts with other lock requests in the pool.
4. **Operation execution**: If no conflicts are detected, the thread proceeds to perform the structural modification. Vertices/edges are added or deleted.
5. **Grain Relabelling**: Due to a change in the topology, the labels of the vertices in the affected grain may need to be updated. This is done by recursively calling the `BFLabel()` function on the children of the affected vertices.
6. **Lock release**: After relabelling terminates, the thread releases the lock on the grain.

In addition to the steps required to perform a data update, a structural modification also involves a relabelling step for the affected grain. However, unlike DomLock, MID and FlexiGran in which the relabelling happens under a global mutex, relabelling in CALock is done under the same lock that is acquired to perform the structural modification. Here, we explain the locking and relabelling mechanism for dynamic graphs.

5.4.1 Vertex addition and deletion

Adding a vertex to the graph does not change the observable topology because this new vertex is not connected to the graph, and hence it is also not reachable from the root. So, addition does not require locking or relabelling of any kind. The vertex gets a default label that contains its ID. Deleting a vertex that does not have any edges does not require synchronization either because such a vertex is not reachable from the root of the graph and also does not have children that might be affected.

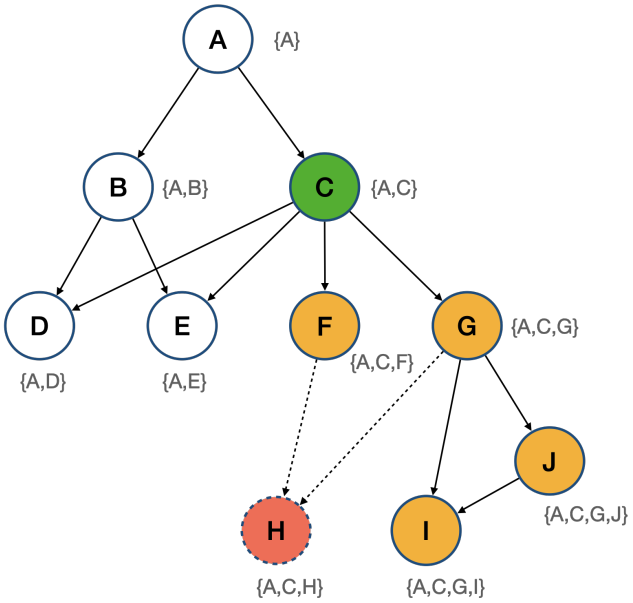


Fig. 5.4: In CALock, Deleting vertex H requires a lock on C

To delete a vertex that is reachable from the root, i.e., connected to the graph, a lock is required. To do this, a write (exclusive) lock is taken on the LGA of the vertex that needs to be deleted. The LGA guards all the paths reaching this vertex and prevents inconsistent access to it. Once the lock is acquired, the vertex is disconnected from the graph and then deleted. Figure 5.4 shows an example where vertex H is deleted. To delete H, a lock is acquired on C which is the LGA of H.

After a vertex is deleted, relabelling might be necessary if the deleted vertex's descendants are still connected to the graph, since the set of their ancestors has changed. Relabelling is done by recursively calling the function `BFLabel()` in Algorithm 1 on the children of the deleted vertex. IN the figure, H does not have any children so, relabelling is not necessary in this case.

5.4.2 Edge addition and deletion

Adding and deleting an edge changes the topology of a graph and also the paths to the vertices. Both operations are performed under a write (exclusive) lock. When adding or deleting an edge between a source vertex u and a target vertex v, a write (exclusive) lock is acquired on the LGCA of u and v. Both operations change the ancestors of the target vertex so relabelling is initiated at the target (v) of the affected edge using `BFLabel()` function.

When a vertex has only one incoming edge, deleting that edge disconnects it from the graph. In this case, relabelling can be omitted because the target vertex v has no parents and is also not reachable from the root of the graph.

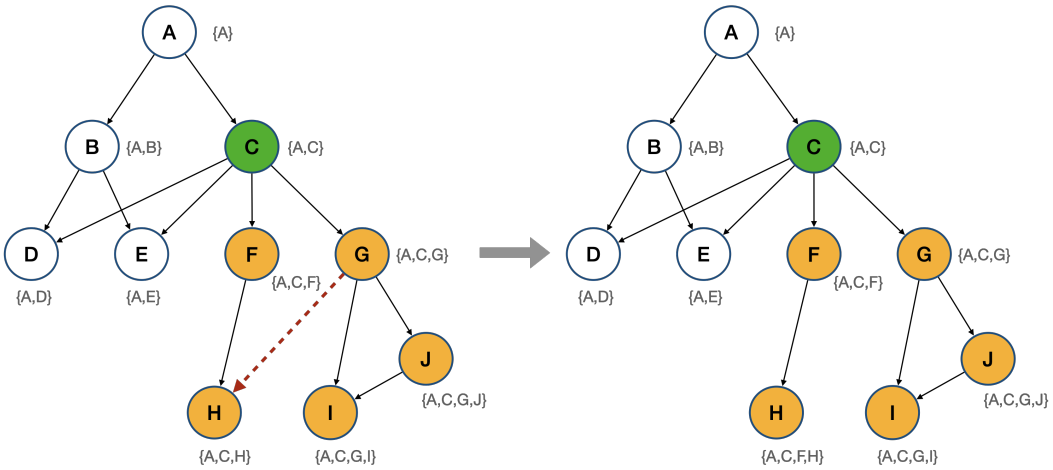


Fig. 5.5: Deleting the edge between G and H requires a lock on C and relabelling of H

Properties of CALock

6

Contents

The properties of any algorithm hold importance in guaranteeing behavioural stability. In this chapter, we show that CALock is safe, live and fair. However, it's crucial to emphasize that the discussion of these formal properties only holds true value when the implementation of the locking algorithm is correct and the threads do not try to circumvent the locking protocol or act maliciously. CALock requires the assignment of lock request sequence numbers, setting of the activity indicator and addition of the lock pool to happen atomically. In our implementation, we use a mutex to achieve this.

6.1 Correctness guarantees

6.1.1 Safety

Property A thread holding a write lock on a guard has exclusive access to the corresponding grain. Assuming that all threads acquire the lock on a guard corresponding to the targets they access, the system guarantees that when a thread requests a write lock on a guard, it is granted the lock only if no other thread can access the grain protected by this guard.

Discussion By contradiction, assume that two threads T_1 and T_2 are granted a write lock on the same vertex v. Then the lock pool would contain, at indices 1 and 2 respectively, two lock requests R_1 and R_2 with the same lock target v and lock mode wl. However, they have different sequence numbers since a sequence numbers is assigned atomically before the corresponding request is added.

Then, T_1 (resp. T_2) iterates over the pool to check for conflicts (Mode conflict, grain conflict). Since all threads iterate over the pool from left to right, T_1 would detect a mode and grain conflict with T_2 and block at index 2 in the pool if its sequence number is greater than that of T_2. Respectively, T_2 would detect a mode and grain conflict with T_1 and block at index 1 in the pool if its sequence number is greater than that of T_1. Hence, T_1 and T_2 are never, simultaneously granted a lock on v. Therefore, CALock is safe.

6.1.2 Liveness

Property When a thread requests a lock, it is guaranteed to be granted the lock eventually, i.e., the lock acquisition algorithm does not block forever assuming every thread eventually releases the lock it holds.

Discussion The CALock protocol assumes that threads do not sleep after lock acquisition. The lock protocol finishes with a thread releasing the lock it holds. The lock pool prevents a thread from holding multiple locks since it only contains one position per thread. If a thread wishes to lock multiple vertices, it consolidates the need to lock multiple vertices into a single request in the form of a lock on their LGCA. Since a thread holds at most one lock at any given time, deadlocks never occur as there is no circular wait.

The absence of deadlocks combined with the assumed progress of threads ensures that CALock is live.

6.1.3 Fairness

Property When a thread requests a lock, it is guaranteed to be granted its request after being blocked at most n times, where n is the number of positions in the lock pool. The lock acquisition algorithm does not lead to a starvation of threads.

Discussion CALock uses a FIFO mechanism to grant locks. When a thread is blocked, it is always by a thread that has a lower sequence number. A thread can be blocked by other threads in the presence of conflicts at most n times (n is the size of the lock pool). After a thread is bypassed at most n times, the sequence number of that blocked thread would be the lowest and it would be granted the lock it requested. This ensures that no thread starves and CALock is fair.

6.2 Complexity analysis of CALock

Beyond the correctness guarantees, the complexity of the locking algorithm is crucial for its practical use. In this section, we discuss the complexity of CALock labelling, relabelling and conflict detection and compare with the other MGL techniques discussed in Chapter 3.

6.2.1 Labelling and relabelling

The labeling and relabelling algorithm involves two operations.

- **Traversal**:Traversal to compute paths is a recursive BFS over the graph starting from the root. The number of edges examined is proportional to the average degree of vertices. In the worst case, where the graph is complete, the degree of any vertex is equal to the number of vertices in the graph.
- **Label computation**: When a vertex is visited during traversal, the intersection of its parents' labels is calculated. The number of elements in the label of a vertex is inversely proportional to the number of its parents. Since the label of a vertex is the set of its guarding ancestors which lie on all paths to a vertex, a vertex with more parents has more disjoint paths from the root and fewer guarding ancestors. Consequently, a vertex with more parents has a smaller label. We can approximate this in terms of the vertex degree as well.

For a graph $G = (V, E)$, let $v = |V|$ be the number of vertices, $e = |E|$ the number of edges and d_{avg} the average degree of a vertex. According to the Handshake lemma, the average degree of a vertex is

$$d_{avg} = \frac{2 \times e}{v} \tag{6.1}$$

The complexity of breadth-first traversal over a graph that contains v vertices, with average degree d_{avg} is:

$$T(BFS) = \theta(v + v.d_{avg})$$

The size of the label of a vertex is inversely proportional to the number of parents it has and consequently, the time complexity of the set intersection required for label computation is inversely proportional to the average degree which is $\theta(\frac{1}{d_{avg}})$

The combined complexity of these operations for the labeling the entire hierarchy is

$$T(Labelling) = \theta((v + v.d_{avg}) \times \frac{1}{d_{avg}}) = \theta(v + \frac{v}{d_{avg}})$$

In the best case, the graph contains only one vertex. The best case complexity is $\Omega(1)$. In the worst case, the average degree of vertices is 1. Thus, the worst-case complexity is $O(2v)$.

6.2.2 Lock guard computation and Conflict detection

When a thread requests a lock, it computes the LGCA of the lock targets to be the guard and issues a lock request. The LGCA is computed via a set intersection of the labels of the lock targets. If the lock request is for q lock targets and the time complexity of computing the LGCA is $\theta(\frac{1}{d_{avg}})$, the complexity of finding the lock guard is:

$$T(Lock\ Guard\ Computation) = \theta(\frac{q}{d_{avg}})$$

In the best case, the lock request is for a single vertex which does not require LGCA computation and the complexity is $\Omega(1)$. In the worst case, the lock request is for all vertices of the graph and the complexity is $O(v)$. A thread checks conflicts with all other threads i.e. n times, where n is the size of the lock pool. For each position in the lock pool, a set membership test is performed. With an efficient set implementation, the membership can be tested in $O(1)$. The complexity of conflict detection is:

$$T(Conflict\ Detection) = \theta(n)$$

6.3 Complexity Comparison

Based on the complexity analysis, we compare CALock with the other MGL techniques discussed in Chapter 3. The average and worst-case complexities of the labelling, lock guard computation and conflict detection operations are summarized in Tables 6.1 and 6.2 respectively.

	Labelling	Lock Guard computation	Conflict detection
Intention Lock	-	-	$\theta(v + v.d_{avg})$
DomLock	$\theta(v + v.d_{avg})$	$\theta(v + v.d_{avg})$	$\theta(n)$
MID	$\theta(v + v.d_{avg})$	$\theta(v + v.d_{avg})$	$\theta(n)$
FlexiGran	$\theta(v + v.d_{avg})$	$\theta(v + v.d_{avg})$	$\theta(n \times (v + v.d_{avg}))$
CALock	$\theta(v + \frac{v}{d_{avg}})$	$\theta(\frac{q}{d_{avg}})$	$\theta(n)$

Tab. 6.1: Average case complexities of MGL techniques

	Labelling	Lock Guard computation	Conflict detection
Intention Lock	-	-	$O(v^2)$
DomLock	$O(v^2)$	$O(v^2)$	$O(n)$
MID	$O(v^2)$	$O(v^2)$	$O(n)$
FlexiGran	$O(v^2)$	$O(v^2)$	$O(nv^2)$
CALock	$O(2v)$	$O(v)$	$O(n)$

Tab. 6.2: Worst case complexities of MGL techniques (n is the number of threads)

Since intention locks do not require metadata in the form of labelling, their complexity is only associated with conflict detection. Conflicts in Intention locks are detected by performing a DFS traversal over the graph. Consequently, the complexity of conflict detection is $\theta(v + v.d_{avg})$.

For label based techniques, the labelling and lock guard computation are proportional to the average degree of vertices. When labelling the hierarchy, all label based techniques perform either a DFS or a BFS traversal over the graph and have the same time complexity. CALock, however, is linear in the number of vertices when computing the lock guard. This is because CALock uses set intersection to compute the lock guard. DomLock, MID and Flexigran use integer intervals and then perform a DFS traversal to find the lock guard corresponding to the lock request. For very complex graphs, CALock labels are smaller than the integer intervals used by

DomLock, MID and Flexigran. This results in a lower complexity for CALock in worst-case scenarios.

7 Implementation

Contents

This chapter provides a comprehensive overview of the implementation of CALock, focusing on its key components: the *labeling mechanism*, *locking protocol*, *lock pool*, and *conflict detection*. The implementation is written in C++, chosen for its ability to deliver high performance and fine-grained control over data structures. To ensure the protocol operates efficiently, several optimizations have been integrated. These include techniques to reduce the size of the labels which conserve memory. Additionally, the implementation minimizes the number of index accesses required to locate the Lowest Guarding Common Ancestor (LGCA), a pivotal step in the locking protocol. These optimizations are designed to balance computational efficiency and scalability, making CALock suitable for hierarchical data structures with varying levels of complexity.

7.1 Implementation of CALock Labelling

The CALock labelling is implemented as the `CALockLabelling` class which provides a method `run()` to label the graph. Unlike the description of the protocol in Chapters 5 and 4, which use characters as vertex IDs, the implementation uses integers. This allows for a more compact storage representation as well as faster membership tests. The hierarchy in STMBench consists of 4 different types of vertices. The implementation distinguishes each by assigning them a fixed vertex ID integer suffix. The vertex IDs are assigned as follows:

- Complex Assembly: `<Identifier>::1`

- Base Assembly: `<Identifier>::2`
- Composite Part: `<Identifier>::3`
- Atomic Part: `<Identifier>::4`

Using this representation, a vertex with STMBench identifier 52 of type `Atomic Part` has a CALock ID 524.

7.1.1 Vertex labels

The label on each vertex is an ordered set containing the vertex IDs of the guarding ancestors of that vertex. In the implementation, to facilitate fast membership test to detect conflicts as well as finding the LGCA, each vertex contains a `std::Set` of guarding ancestors and a `std::Vector` containing an order of these guarding ancestors.

A vector in C++ provides constant time insertion but is linear for a membership test. To check if an element exists in a vector a search is performed which, in the C++ standard template library, is done via the `std::find` function.

Since membership tests are frequently used in CALock when locking and also during conflict detection, the implementation uses an auxiliary Set for membership tests and bypasses the linear search complexity of the vector. Together, a vector and a set provide constant order time complexity for membership test as well as label updates.

```
1 class Vertex {
2     ...
3     int id;
4     bool hasLabel;
5     vector<int> pathLabel{};
6     set<int> guardingAncestors{};
7     bool isDeleted;
8     ...
9 };
```

Listing 7.1: Vertex class with CALock label fields

In order to find the LGA(lowest guarding ancestor) of a single vertex, the last element of the `pathLabel` vector is returned. When finding the LGCA(lowest guarding common ancestor) of a set of vertices, the intersection of the path labels of the vertices is computed. The last element of the resulting vector is the LGCA.

```
int getLSCA() const {
    if (pathLabel.size() == 1) {
        return *(--pathLabel.rbegin());
    } else {
        return *pathLabel.rbegin();
    }
}
```

Listing 7.2: Finding LGCA using path label

7.1.2 Labelling algorithm

The labelling algorithm is a breadth first traversal over the hierarchy. It is implemented by keeping separate queues for each type of vertex, i.e., Complex Assembly, Base Assembly, Composite Part, and Atomic Part. As the traversal progresses, vertices are labelled in the order of their types, which ensures that the guarding ancestors of a vertex are labelled before the vertex itself. Labelling starts from the root of the hierarchy. The root is labelled, and its children are added to their respective queues. The implementation of the labelling algorithm is shown in Listing 7.3.

```
void CALockLabelling::run(int tid) const{
    queue<ComplexAssembly *> cassmQ;
    queue<BaseAssembly *> bassmQ;
    queue<CompositePart *> cpartQ;
    queue<AtomicPart *> apartQ;


    ComplexAssembly *root = dataHolder->getDesignRoot();
    vector<int> rootLabel = root->pathLabel;
    rootLabel.push_back((root->getId() * 10) + 1);
    root->setPathLabel(rootLabel);
    cassmQ.push(root);

    while (!cassmQ.empty()) {
        traverse(cassmQ.front(), &cassmQ, &bassmQ);
        cassmQ.pop();
    }
    while (!bassmQ.empty()) {
        traverse(bassmQ.front(), &cpartQ);
        bassmQ.pop();
    }
    while (!cpartQ.empty()) {
        traverse(cpartQ.front(), &apartQ);
        cpartQ.pop();
    }
```

```
26      while(!apartQ.empty()){
27          Set<AtomicPart *> visitedPartSet;
28          traverse(apartQ.front(), visitedPartSet, apartQ.front()->
                pathLabel);
29          apartQ.pop();
30      }
31  }
```

Listing 7.3: BFS traversal for CALock labelling

During labelling, the `traverse` method is called for each vertex in the queue to label it. The `traverse` method is overloaded for each type of vertex. After computing the label of a vertex, the method adds the children of the vertex into their appropriate queues based on their type. Listing 7.4 shows the implementation of the `traverse` method for a vertex of Complex Assembly type.

```
1   void traverse(  ComplexAssembly *cassm,
2                   queue<ComplexAssembly *> *cassmQ,
3                   queue<BaseAssembly *> *bassmQ) const {
4
5       list<int> currLabel = cassm->pathLabel;
6       Set<Assembly *> *subAssm = cassm->getSubAssemblies();
7       SetIterator<Assembly *> iter = subAssm->getIter();
8       bool childrenAreBase = cassm->areChildrenBaseAssemblies();
9
10      while (iter.has_next()) {
11          Assembly *assm = iter.next();
12          // If children are Complex Assemblies, add them to the
                Complex Assembly queue.
13          if (!childrenAreBase) {
14              int labelIdentifier = (assm->getId() * 10) + 1;
15              currLabel.push_back(labelIdentifier);
16              assm->setPathLabel(currLabel);
17              cassmQ->push((ComplexAssembly *) assm);
18          } else {
19          // If children are Base Assemblies, add them to the Base
                Assembly queue.
20              int labelIdentifier = (assm->getId() * 10) + 2;
21              currLabel.push_back(labelIdentifier);
22              assm->setPathLabel(currLabel);
23              bassmQ->push((BaseAssembly *) assm);
24          }
25          currLabel.pop_back();
26      }
27  }
```

Listing 7.4: Labelling a complex assembly

```
void traverse(AtomicPart *apart,
    Set<AtomicPart *> &visitedPartSet,
    list<int> currLabel) const {

    if (apart == NULL || visitedPartSet.contains(apart)) {
        return;
    } else {
        visitedPartSet.add(apart);
        Set<Connection *> *fromConns = apart->getFromConnections();
        SetIterator<Connection *> fiter = fromConns->getIter();

        // Find elements that need to be removed from the label.
        set<int> removals;
        for (int a: currLabel) {
            while (fiter.has_next()) {
                if (!fiter.next()->getSource()->criticalAncestors.
                    contains(a)) {
                    removals.insert(a);
                    break;
                }
            }
            fiter = fromConns->getIter();
        }

        currLabel.remove_if([removals](int l) { return removals.
            contains(l); });

        // If the label has changed, update it in the vertex and
            traverse the children.
        if (currLabel != apart->pathLabel) {
            apart->setPathLabel(currLabel);
            Set<Connection *> *toConns = apart->getToConnections();
            SetIterator<Connection *> iter = toConns->getIter();
            while (iter.has_next()) {
                Connection *conn = iter.next();
                currLabel.push_back((conn->getDestination()->getId
                    () * 10) + 4);
                traverse(conn->getDestination(), visitedPartSet,
                    currLabel);
                currLabel.pop_back();
            }
        }
        visitedPartSet.remove(apart);
    }
}
```

Listing 7.5: Labelling an atomic part

Atomic Part graph in STMBench is dense and contain many cycles. Due to this, atomic Parts are handled differently since an atomic part vertex can be visited multiple times. An atomic part graph is connected to the hierarchy under a composite part via a single vertex called the `rootPart`. The `rootPart` is designated as the root of the atomic part graph. Once this root is labelled, the algorithm traverses the graph of atomic parts in a post-order fashion. The implementation of the `traverse` method for an Atomic Part is shown in Listing 7.5.

7.2 Lock Requests and the Lock Pool

Once a thread wishes to lock a set of targets, it creates a lock request. The lock request is implemented as a `lockObject` which is inserted into the lock pool. This `lockObject` contains the following fields:

- `int Id`: Guard ID.
- `set<int> *criticalAncestors`: Set of guarding ancestors.
- `int mode`: Mode of the lock (Read/Write).
- `long Oseq`: Sequence number to decide the arrival order of the lock requests.
- `atomic_flag accessController`: Flag which conflicting threads use to wait on this thread.

The `lockObject` class is shown in Listing 7.6.

```
1  class lockObject {
2  public:
3      set<int> *criticalAncestors;
4      int Id;
5      int mode;
6      long Oseq;
7      atomic_flag accessController = ATOMIC_FLAG_INIT;
8
9      lockObject(int pId, set<int> *ancestors, int m) {
10         Id = pId;
11         criticalAncestors = ancestors;
12         mode = m;
13         Oseq = -1;
14         accessController.test_and_set();
15     }
16 };
```

Listing 7.6: Lock Object class

The lock pool is an array that contains `lockObjects`. The size is fixed and bounded to the number of concurrent hardware threads provided by the system. Listing 7.7 shows the class definition of the lock pool.

```
class CAPool {
public:
    mutex lockPoolLock;
    lockObject *locks[SIZE];
    shared_mutex threadMutexes[SIZE];
    condition_variable_any threadConditions[SIZE];
    long int Gseq;

    CAPool() {
        Gseq = 0;
        for (int i = 0; i < SIZE; i++) {
            locks[i] = nullptr;
        }
    }
    bool acquireLock(lockObject *reqObj, int threadID) {
        //Check for conflicts and wait for resolution
    }
    void releaseLock(lockObject *l, int threadId) {
        //Release the lock
    }
};
```

Listing 7.7: Lock Pool class

When acquiring a lock, a thread inserts a `lockObject` corresponding to its lock request into the pool and waits for the lock to be granted. The `acquireLock` method is responsible for checking for conflicts and waiting for resolution. The method is shown in Listing 7.8.

```
bool acquireLock(lockObject *reqObj, int threadID) {
    lockPoolLock.lock();
    reqObj->Oseq = ++Gseq;
    locks[threadID] = reqObj;
    lockPoolLock.unlock();
    for (int i = 0; i < SIZE; i++) {
        // A thread won't run into conflict with itself.
        if (locks[i] != nullptr) {
            auto l = locks[i];
            if (l != nullptr &&
                // Mode Conflict.
                (reqObj->mode == 1 ||
                (reqObj->mode == 0 && l->mode == 1)) &&
                // Grain Overlap.
                (reqObj->Id == l->Id ||
```

```
16                  reqObj->criticalAncestors->contains(l->Id) ||
17                  l->criticalAncestors->contains(reqObj->Id)) &&
18                  // Arrival Order.
19                  (reqObj->Oseq > l->Oseq)) {
20                  // Wait for resolution and notification.
21                  l->accessController.wait(true);
22              }
23          }
24      }
25      return true;
26  }
```

Listing 7.8: Acquiring a lock

When acquiring a lock, the lock request is first assigned a sequence number (`Oseq`) and then inserted into the lock pool. This assignment and insertion into the pool is done under a mutex lock on the pool to prevent a race condition between the assignment of a sequence number and insertion into the pool.

Once a lock request is inserted into the pool, the method checks for conflicts with other lock requests in the pool. This check can be performed in parallel for all lock requests in the pool. If no conflict is detected, the lock is granted and the thread can proceed to its operation.

If a conflict is detected, the thread waits for resolution. This is done by waiting on the `accessController` of a conflicting lock request flag to become `false` (Listing 7.8 - line 21).

```
1       void releaseLock(lockObject *l, int threadId) {
2           locks[threadId] = nullptr;
3           l->accessController.clear();
4           l->accessController.notify_all();
5       }
```

Listing 7.9: Releasing a lock

When a thread releases a lock, it removes the lock request from the pool, changes its `accessController` flag to `false`, and notifies all waiting threads. The `releaseLock` method is shown in Listing 7.9. Once waiting threads are notified, they resume execution and proceed to check for further conflicts in the lock pool.

After checking for conflicts with all lock requests in the pool and having waited on conflicting requests, a thread reaches the end of the lock pool (Listing 7.8 - line 6). At this point, the lock is always granted as there are two possible scenarios:

a. The lock request is not in conflict with any other lock request in the pool (Listing 7.8 - lines 12, 13).

b. The lock request has priority over all conflicting lock requests in the pool because its sequence number is smaller than all conflicting lock requests (Listing 7.8 - line 19).

7.3 Overall execution of an operation in CALock

The overall execution of an operation in CALock is shown in Listing 7.10. The operation is executed in the `run` method of the operation class. The listing shows a structural modification in which a composite part is added to a base assembly.

The operation first computes the set intersection of the labels of the base assembly and the composite part (Listing 7.10 - lines 8-12). This set intersection is used to find the LGCA of the base assembly and the composite part. If the LGCA is disconnected, the base assembly is used as the LGCA (Listing 7.10 - lines 14-19).

A lock request is then created with the LGCA and the guarding ancestors in write mode (Listing 7.10 - line 21). The lock is acquired using the `acquireLock` method of the lock pool. As long as the lock is not granted, the thread waits for resolution (Listing 7.10 - line 24). Once granted, the operation is performed (Listing 7.10 - line 26).

Since a structural modification has occurred, the hierarchy is relabelled. This is done by executing a relabelling operation and adding the composite part to the queue of composite parts for relabelling (Listing 7.10 - lines 28-30).

After relabelling is complete, the lock is released using the `releaseLock` method of the lock pool (Listing 7.10 - line 32).

```
1  int CAStructuralModification3::run(int tid) const {
2      CompositePart *cpart = dataHolder->getCompositePart(cpartId);
3      BaseAssembly *bassm = dataHolder->getBaseAssembly(bassmId);
4
5      list<int> lockLabel = {};
6
7      // Find the set intersection of the labels of bassm and cpart.
8      for (int a: bassm->pathLabel) {
9          if (cpart->criticalAncestors.contains(a)) {
10             lockLabel.push_back(a);
11         }
12     }
```

```
13
14      // find the LGCA of the bassm and cpart.
15      pair<DesignObj *, bool> lo = lscaHelpers::getLockObject(
            lockLabel);
16      //if cassm is disconnected, use bassm as the LGCA.
17      if (!lo.second) {
18          lo.first = bassm;
19      }
20      //create a lock request with the LGCA and the guarding
            ancestors in write mode.
21      auto *l = new lockObject(lo.first->getLabellingId(), &lo.first
            ->criticalAncestors, 1);
22
23      // Acquire the lock.
24      if (caPool.acquireLock(l, tid)) {
25          // Perform the operation.
26          bassm->addComponent(cpart);
27          // Relabel the hierarchy because a structural modification
                has occurred.
28          auto *r = new CALockRelabeling(dataHolder, tid);
29          r->cpartQ.push(cpart);
30          r->run();
31          // Release the lock.
32          caPool.releaseLock(l, tid);
33      }
34
35  }
```

Listing 7.10: Overall execution of an operation in CALock

8 Experimental Evaluation

Contents

We evaluate the performance of CALock and experimentally compare it with the state of the art. Our evaluation uses the same benchmarks as DomLock, MID and FlexiGran [KN16; MAN22; MAN24] i.e. STMBench7[GKV07]. In this chapter, we first present the experimental setup and then the results of the experiments and discuss their implications.

We run our experiments on a machine with an AMD EPYC 7642 CPU, with 48 Cores, a base clock of 2.3 GHz and 512 GB of RAM [1]. The benchmark is deployed on a Docker container under Ubuntu 20.04. To build the benchmark, GCC 12.1 and Cmake 3.22 are used. The compilation is done at the C++ 23 standard to enable the use of atomic wait primitives. The optimization level is set to O3.

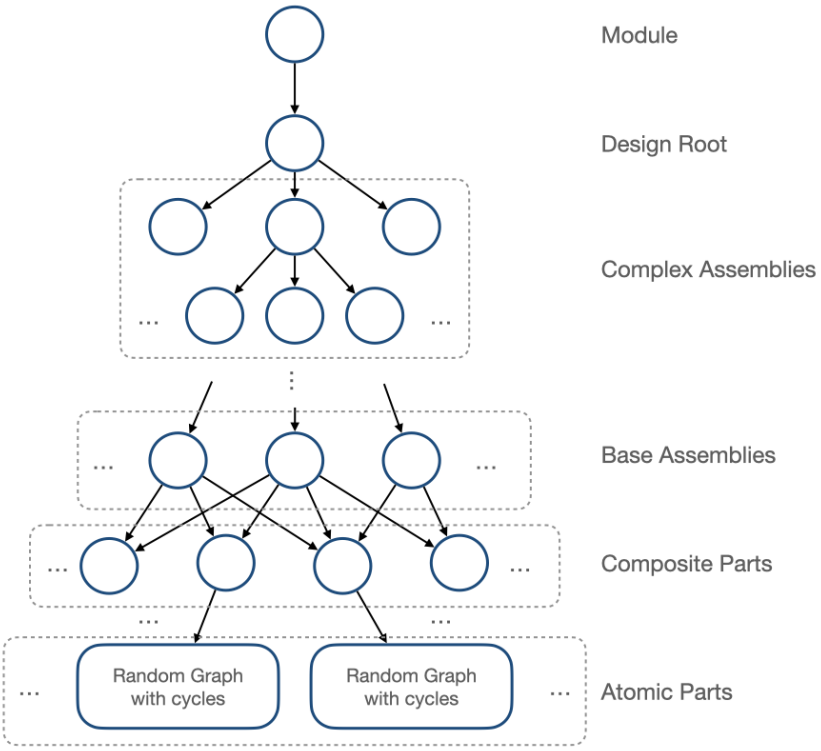


Fig. 8.1: Structure of a module in STMBench with medium lock boundaries

8.1 Benchmark Suite: STMBench7

STMBench is a well known benchmark based on the classical OO7 object oriented benchmark suite [CDN93]. It is widely used to evaluate the performance of hierarchical algorithms and data structures [Pro+19; Val+16; Fel+16; CC16; KPR15; FB15; RC14; KN16; MAN22; MAN24; KN18; GKN18; LZ14]

The hierarchy in STMBench7, as represented in Figure 8.1, consists of a *module* at the root level. This module consists of several levels of *complex assemblies*. Each of the deepest complex assemblies consists of a set of *base assemblies*. A *composite part* can be contained in several base assemblies. Each composite part contains a set of *atomic parts*. These atomic parts form a near-complete graph. The root of this graph is connected to a single composite part.

STMBench7 provides coarse-grain and medium-grain lock implementations. These are representatives of fixed granularity locking techniques, used in practice. The coarse-grain lock is a reader-writer lock on the entire graph. Figure 8.1 shows

[1]Experiments presented in this thesis were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

the medium-grain lock grains for a module. Complex assemblies are divided into levels and each level is guarded by its own lock. Deeper in the graph, the set of atomic parts, under each composite part, is guarded by its own lock. A structural modification operation acquires a mutex on the entire graph.

STMBench provides a set of operations that stress the locking mechanism being evaluated. From these operations, Q1 and Q2 are read-only operations, OP1 and OP2 are read-write operations and SM1 and SM2 are structural modification operations.

1. Reading an atomic part (Q1).
2. Reading a set of atomic parts (Q2).
3. Reading a complex assembly (OP1).
4. Reading a set of complex assemblies (OP1).
5. Reading a base assembly (OP2).
6. Reading a set of base assemblies (OP2).
7. Writing data to an atomic part (OP3).
8. Writing data to a set of atomic parts (OP4).
9. Deleting a composite part (SM1).
10. Adding an edge between a base assembly and a composite part (SM2).

8.1.1 Synopsis

The research questions addressed by our benchmarks compare CALock with DomLock, MID and FlexiGran:

§8.2 How quickly does CALock grant a lock compared to DomLock, MID, FlexiGran for lock requests of different operation types (Q1-SM2)?

§8.3 What is the cost of labelling a hierarchy with CALock labels compared to integer intervals for DomLock, MID and FlexiGran?

§8.4 How expensive is it to maintain a set of guarding ancestors for each vertex in the hierarchy for CALock compared to integer intervals for DomLock, MID and FlexiGran?

§8.5 Does CALock eliminate false subsumptions and reduce grain size compared to DomLock, MID and FlexiGran?

§8.6.1 What is the performance of CALock compared to other lock strategies for workloads with only data updates (Q1-OP4)?

§8.6.2 What is the performance of CALock compared to other lock strategies for workloads that also include structual modifications (Q1-SM2)?

8.2 Per operation response time

Figure 8.2 shows the time to completion for different operations in STMBench7. Coarse-grain and Medium-grain locks are the fastest to grant lock requests. However, they do not scale well (see Sections 8.6.1, 8.6.2).

Intention locks are slower than coarse-grain and medium-grain locks because they require a depth-first traversal to acquire intention locks on the paths that lead to a target vertex. The cost of acquiring intention locks is linear in the number of unique paths to a target vertex. For requests with multiple target vertices (Q2, OP4), this cost is multiplied and leads to a higher response time.

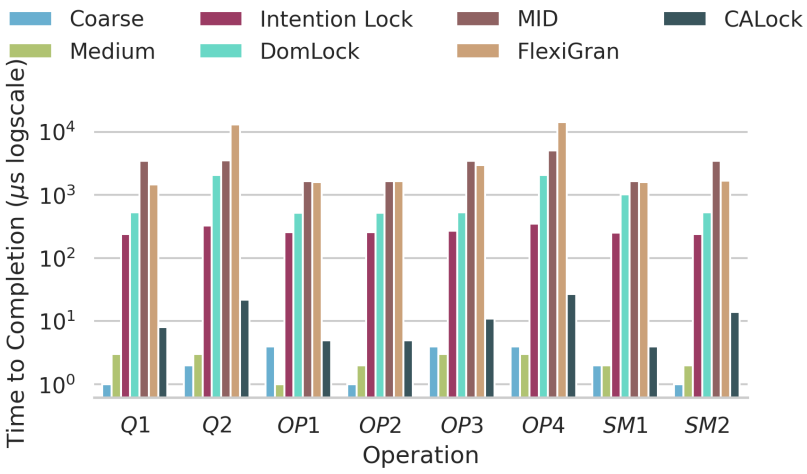


Fig. 8.2: Time to completion for different operations in STMBench (lower is better)

In MGL techniques like DomLock, MID and FlexiGran, the time to completion is at least $10^{2.5}$ times higher than coarse-grain, medium-grain and intention locks. This is because once a thread identifies the lock targets and finds their interval range, a depth first traversal is required to find the lock guard with the same interval range. This traversal is especially expensive for vertices deeper in the hierarchy.

Between MGL techniques, DomLock is the fastest since conflict detection requires only testing overlap between a pair of intervals. MID is slower than DomLock because it requires testing overlap between two pairs of intervals. FlexiGran is slower than MID because it requires testing overlap between intervals and level numbers. Conflict detection in FlexiGran MGL and FlesiGran fine-grained locks involves testing reachability between the guard of the MGL lock and the fine-grained lock. This is expensive and leads to an overall higher time to completion.

CALock is at least 10^2 times faster to grant a lock than other MGL techniques. This is because CALock eliminates the need for a traversal to find the lock guard. Instead, a set intersection of the labels is used to find the LGCA which serves as the lock guard. This is significantly faster than the depth-first traversal required by Intention locks, DomLock, MID and FlexiGran.

Read only operations(Q1, Q2) are the fastest since they can happen in parallel, even in the same grain. Write operations (OP3, OP4) are slower since they require exclusive access to the grain. Structural modifications often take the longest since they require exclusive access to the entire graph with DomLock, MID and FlexiGran. CALock parallelizes the relabelling of disjoint subgraphs and is at least 10^2 times faster.

8.3 Metadata management: bulk labelling and relabelling

MGL techniques rely on metadata to identify lock grains. DomLock and MID use integer intervals for subsumption testing, while FlexiGran combines intervals with a level number. CALock, on the other hand, utilizes sets of vertex identifiers. Regardless of the metadata type, excessive housekeeping to maintain its accuracy can easily negate any performance benefits. In this section, we examine the time required to compute the metadata initially (bulk labeling) and the time needed to update it after a structural modification (relabeling). The benchmark is run on three different sizes of the STMBench7 graph. Table 8.1 lists the size of the STMBench7 hierarchies.

Graph Size	Vertices	Edges
Small	234,737	4,393,682
Medium	2,334,257	43,759,682
Large	23,329,457	437,419,682

Tab. 8.1: Sizes of the STMBench7 hierarchies

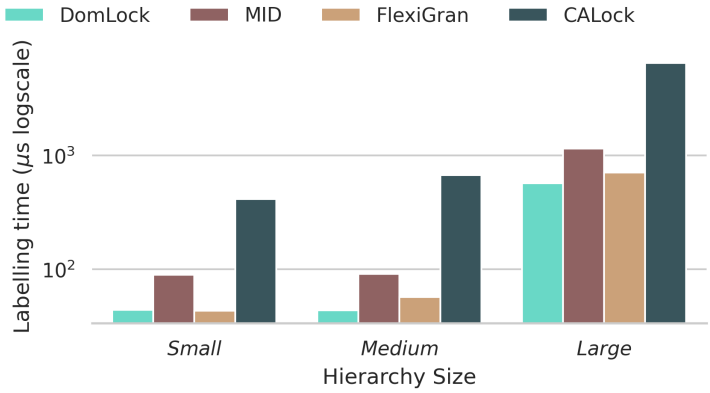


Fig. 8.3: Time to compute initial labels (lower is better)

8.3.1 Bulk labelling

We measure the pre-processing time, i.e., the time it takes to assign the labels to a graph for label based MGL techniques like DomLock, MID, FlexiGran and CALock. This simulates loading data into a database. This time is measured for three different sizes of the STMBench7 graph. The results are shown in Figure 8.3. We observe that DomLock is the fastest since a single depth-first-traversal is sufficient to compute the intervals. MID is 8 times slower than DomLock because it needs to compute two pairs of intervals for each vertex. One computed by a depth-first traversal and another by a depth-first traversal on the mirror image of the graph. FlexiGran is faster than MID but slightly slower than DomLock because of the additional level information that needs to be computed per vertex.

CALock labels take the longest to pre-process since the labels are defined by a recursive breadth-first traversal with a fix-point dependent on the number of paths to a vertex from the root. CALock remains 10 times slower than MID and 20 times slower than DomLock for pre-processing.

8.3.2 Relabelling

Figure 8.4 shows the average time spent relabelling the graph per structural modification. Coarse grain, medium grain locks and intention locks do not have additional metadata and hence do not require relabelling. DomLock, MID and FlexiGran are significantly slow since they relabel the entire graph after a structural modification. This relabelling prevents any concurrent operations. Figure 8.4 shows time taken by DomLock, MID, FlexiGran and CALock to relabel the graph after a structural modification.

In contrast, CALock relabels only the subgraphs directly affected by the structural modification under the same lock as the structural modification. Thus, multiple grains can be locked, modified and relabelled in parallel. CALock is 100 times faster at relabelling than DomLock, MID and FlexiGran.

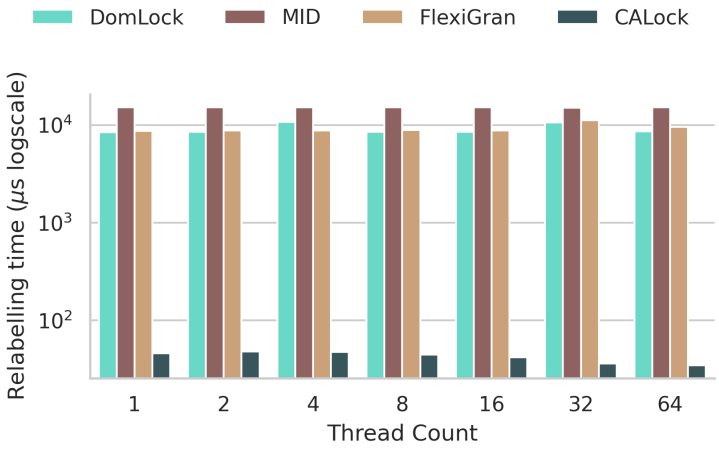


Fig. 8.4: Time spent relabelling the graph per structural modification (lower is better)

8.4 Size in memory

Label based MGL techniques utilize metadata to identify the lock grains. DomLock utilizes integer ranges as labels, which are compact. MID uses two pairs of integer ranges to represent the intervals of the vertices. FlexiGran uses the DomLock intervals along with an integer to store the level of a vertex. In contrast, CALock employs sets of vertex identifiers as labels, leading to a larger memory footprint.

As shown in Figure 8.5, CALock's metadata consumes approximately 1.5× more memory than DomLock, MID and FlexiGran. In DomLock, MID and FlexiGran,

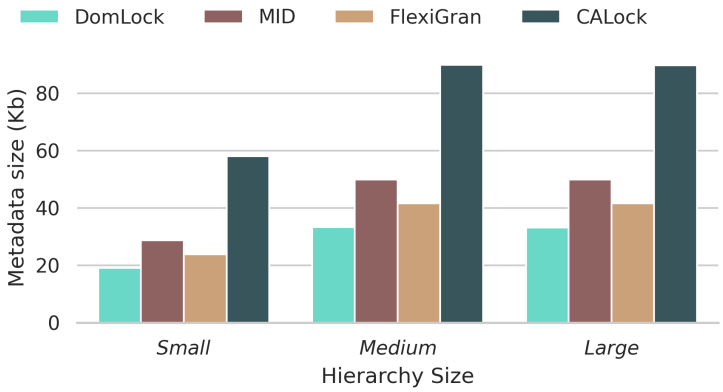


Fig. 8.5: Size of the metadata used for labelling in STMBench (lower is better)

regardless of the topology of the graph, the label at each vertex consists of integers. This has low memory requirement however, the information about the exact topology of the hierarchy cannot be inferred from the labelling itself, leading to false subsumptions. CALock labels, being sets, contain more information allowing for smaller grain sizes and avoid false subsumptions.

8.5 Lock granularity and false subsumptions

Locking a vertex in MGL implicitly also locks all other vertices present in the lock grain. This allows threads to use a single guard for multiple targets such that the guard is the root of the smallest sub-graph containing the targets. A smaller grain size allows more parallelism and reduces the probability of conflicts.

To compare the grain sizes for different locking protocols, we measure the number of targets implicitly locked for a guard vertex in the STMBench7 graph. Figure 8.6 compares the granularity of DomLock, MID, FlexiGran and CALock.

Locking an atomic part has almost the same effect with all algorithms with CALock marginally reducing grain sizes. When locking higher up in the graph, the effects vary. Locking a complex assembly is more expensive with DomLock, MID and FlexiGran because complex assemblies often share composite parts. Using a complex assembly as a lock guard causes all shared composite parts to be locked leading to large grains. With DomLock, MID and FlexiGran, the grain size is larger 8 times bigger than CALock due to false subsumptions.

When locking base assemblies, with DomLock, MID and FlexiGran, multiple base assemblies are locked due to the one-to-many relationship between base assemblies

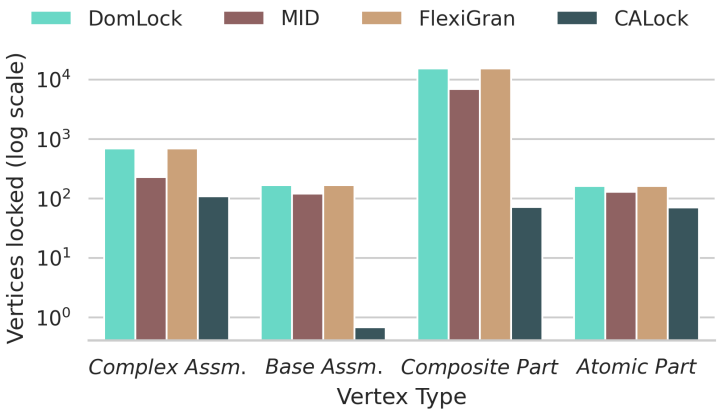


Fig. 8.6: Vertices locked per vertex type (lower is better)

and composite parts, again, owing to false subsumptions. CALock is significantly better at this level with grain sizes being 100 times smaller than DomLock, MID and FlexiGran.

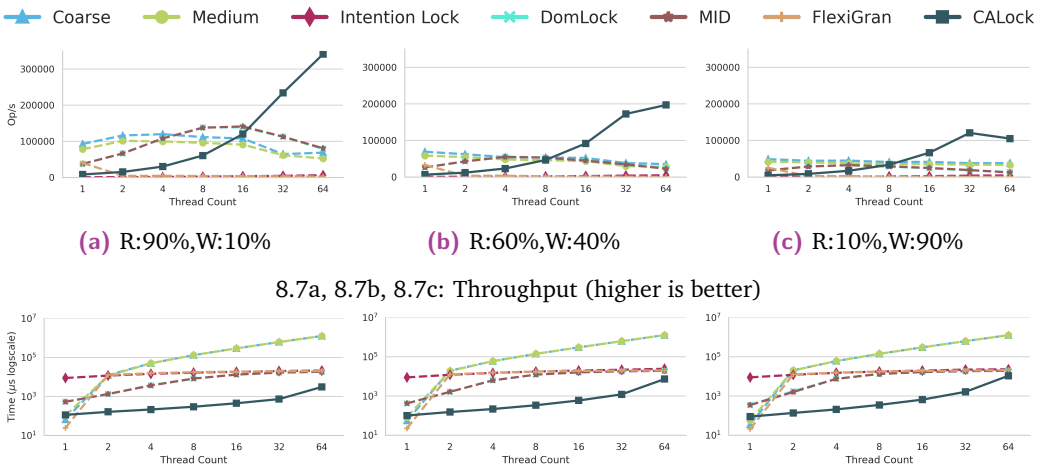
Composite parts exhibit a similar trend to base assemblies. CALock has a grain size 100 times smaller than DomLock, MID and FlexiGran. When a composite part is a lock target, the guard is often a base assembly or a complex assembly because a composite part is often contained in multiple base assemblies. This, combined with false subsumptions, leads to large grain sizes for DomLock, MID and FlexiGran.

The granularity of the lock and its effect on subsumption is highly dependent on the topology of the graph. False subsumptions aggravate the problem of large grains and lead to more grain overlaps between threads. The grain sizes of CALock are always smaller other locking techniques as is evident in Figure 8.6.

8.6 Overall locking performance

Sections 8.2, 8.3, 8.4 and 8.5 studied individual parameters in isolation. Here, we study them together and evaluate their effects on overall performance. In these set of benchmarks, we study both data updates and structural modifications. Figures 8.7 and 8.8 show the throughput of different workloads on static graphs and dynamic graphs respectively. The charts in these figures are plotted with the number of concurrent threads on the x-axis and the throughput (op/s) or response time (μs) on the y-axis. Response time is measured from when the thread issues a lock request until the lock is granted.

8.6.1 Data Uodates



(a) R:90%,W:10% (b) R:60%,W:40% (c) R:10%,W:90%

8.7a, 8.7b, 8.7c: Throughput (higher is better)

(d) R:90%,W:10% (e) R:60%,W:40% (f) R:10%,W:90%

8.7d, 8.7e, 8.7f: Average time to grant lock request (lower is better)

Fig. 8.7: Performance with different workload types on static graphs (R: reads, W: writes).

Throughput figures 8.7a, 8.7b and 8.7c show that coarse-grain and medium-grain locks have better throughput for up to 4 concurrent threads due to the additional computation of the lock grain required for DomLock, MID, FlexiGran and CALock. Beyond 4 threads, MGL techniques give better throughput. However, the performance of DomLock and MID stagnates at 8 threads.

The graph in STMBench is irregular and has multiple paths leading to a vertex. DomLock, MID and Flexigran intervals often lead to false subsumptions (see Section 8.5). Intention locks exhibit poor performance due to the high number of paths leading to target vertices from the root of the STMBench hierarchy (see Section8.2).

CALock successfully minimizes the size of the lock grains and allows threads to lock disjoint grains in parallel, achieving better scalability than DomLock, MID and FlexiGran. We observe that CALock is at least 2 times faster than DomLock for 32 threads and at least 4.5 times faster than DomLock for 64 threads.

Response time figures 8.7d, 8.7e and 8.7f compare the wait time per thread between coarse-grain locks, medium-grain locks, Intention locks, DomLock, MID, FlexiGran and CALock. Response time increases with the number of threads due to an increase in conflicts. For a single thread, coarse-grain and medium-grain locks are the fastest, but their performance suffers with any form of parallelism.

Between the MGL techniques, FlexiGran and Intention lock take the longest to grant a lock on average. With Intention locks, this is because of the cost of traversals

required to acquire intention locks on vertices. With FlexiGran, lock conflict detection is expensive because of the coexistence of MGL and fine-grain locks. DomLock and MID are faster than FlexiGran but, remain significantly slower than CALock.

In interval-based MGL techniques like DomLock, MID and FlexiGran, once a lock request identifies an interval it wishes to lock, the thread traverses the hierarchy to find the corresponding lock guard with the requested interval (or. interval pair for MID). This traversal is especially expensive when locking deeper in the hierarchy. CALock on the other hand computes the guard vertex directly by a set intersection, not requiring a traversal giving faster response times for lock requests.

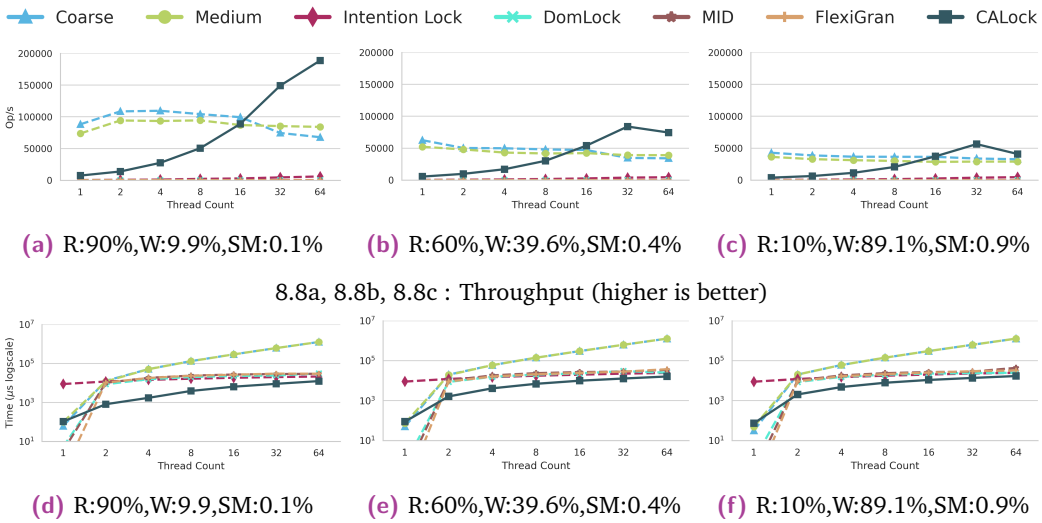
For all locking protocols, the overall wait time increases with the number of concurrent threads, because of an increase in the number of conflicts due to overlapping grains. For 64 threads, CALock is 6 times faster than DomLock in a read-dominated load and 1.5 times faster in a write-dominated load.

8.6.2 Structural Updates

A structural modification changes the topology of the graph. This triggers relabelling for MGL locking techniques like DomLock, MID, FlexiGran and CALock. Throughput figures 8.8a, 8.8b and 8.8c show the performance of locking algorithms when reads and writes interleave with structural modifications. We set structural modifications to 1% of the total writes. For example in Figure 8.7b, 40% of the operations are writes. Out of these writes, 10% are structural modifications, i.e. $40\% \times 1\% = 0.4\%$ structural modifications. The remaining 39.6% are data writes.

In a write-heavy workload (Throughput figure 8.8c), structural modifications can be as high as 0.9%. While in read-heavy workloads (Throughput figure 8.8a), they are as low as 0.1%. Even under a small fraction of structural modifications, we observe that Intention locks, coarse-grain and medium-grain locks do not scale with the number of threads.

DomLock, MID and FlexiGran perform significantly worse compared to other algorithms because along with the lack of parallelism due to an additional relabelling required when a structural modification occurs. Note that the curves in Throughput figures 8.8a, 8.8b and 8.8c for DomLock, MID and FlexiGran are relatively flat, indicating a lack of scalability. CALock can parallelize structural modifications and is 2x faster than coarse and medium-grained locks and about 8x faster than DomLock, MID and FlexiGran.



(a) R:90%,W:9.9%,SM:0.1% (b) R:60%,W:39.6%,SM:0.4% (c) R:10%,W:89.1%,SM:0.9%

8.8a, 8.8b, 8.8c : Throughput (higher is better)

(d) R:90%,W:9.9,SM:0.1% (e) R:60%,W:39.6%,SM:0.4% (f) R:10%,W:89.1%,SM:0.9%

8.8d, 8.8e, 8.8f: Average time to grant lock request (lower is better)

Fig. 8.8: Performance with different workload types on dynamic graphs (R: reads, W: writes; SM: structural modifications).

Response time for Intention locks remains same regardless of the proportion of structural modifications. However, as shown in Response time figures 8.8d, 8.8e and 8.8f, response time for DomLock, MID and FlexiGran is longer for workloads with structural modifications compared to workloads without. Response time for CALock also increases for dynamic graphs when compared to static graphs but CALock remains faster than all other lock techniques for even the most contended workload.

8.7 Summary of experimental results

8.7.1 Summary

With the results from the experiments using STMBench7 and microbenchmarks, we can derive the following conclusions.

- Intention locks are the most expensive locking technique for any STMBench workload and are not suitable for irregular graphs.
- CALock is faster than DomLock, coarse-grain locks and medium-grain locks for all studied workloads with more than 8 concurrent threads.

- In read heavy workloads without structural modifications, CALock is at least 3 times faster compared to any other locking technique.
- In read heavy workloads with structural modifications, CALock is 1.5 times faster than coarse-grain and medium-grain locks and 8x faster than DomLock, MID and FlexiGran.
- In write heavy workloads without structural modifications, CALock is 2 times faster than coarse-grained and medium-grained locks and 4 times faster than DomLock, MID and FlexiGran
- In write heavy workloads with structural modifications, CALock performs as good as coarse-grain and medium-grain locks and 4 times faster than DomLock, MID and FlexiGran
- CALock is an appropriate locking technique if the graph is irregular and the workload consists of structural modifications (for which DomLock, MID and FlexiGran are unsuitable).

Conclusion

9

Contents

A connected structure is a fundamental way of representing data. Semi-structured connected data has become mainstream. First, with the advent of NoSQL stores, and now with the rise of graph databases. These databases are used in a wide range of applications, from social networks to recommendation systems.

While representing connected data effectively is a problem in itself, managing consistent access to this data is another challenge. Another dimension of complexity is added when the topology of the connected data changes. To guarantee consistency while maintaining performance in the face of these challenges, we need efficient concurrency control mechanisms designed specifically for connected data.

To this end, the first technique was *Intention Locks*. While intention locking has proven its mettle as an effective concurrency control mechanism when managing tree like structures, it has severe limitations when dealing with irregular semi-structured data like hierarchies. Modern *Multi-Granularity Locking* (MGL) techniques have been proposed to address the limitations of intention locks. However, these techniques are not designed to handle the dynamic nature of connected data. They are not efficient in scenarios where the topology of the data changes frequently.

An effective multi-granularity locking mechanism for hierarchical data exploits the topology of the hierarchy to optimize the assignment of lock grains. Intention locks achieve this by using special lock modes to indicate the intention of a transaction to

lock a node and its descendants. Newer MGL techniques use a labeling scheme to identify lock grains efficiently.

In this thesis, we explored the design, implementation, and evaluation of CALock, a novel labelling and concurrency control mechanism for hierarchical data. CALock uses a path based labelling scheme which assigns a label to each vertex in the hierarchy. This label is an ordered set of all *guarding ancestors* of the vertex. Locking one of these guarding ancestors is sufficient to lock the vertex. To minimize the grain size of a lock guard CALock uses the *lowest guarding ancestor* (LGA) of a vertex as the lock guard. To lock multiple vertices, CALock uses the *lowest common guarding ancestor* (LGCA) of the vertices as the lock guard. By using the lowest guarding ancestor as the lock guard, CALock minimizes grain size and maximizes parallelism by reducing the probability of grain overlaps that cause thread blocks.

Our research has demonstrated the effectiveness of CALock in improving system performance and reliability in various scenarios. Through rigorous experimentation and analysis, we have shown that CALock can significantly reduce contention and enhance throughput in multi-threaded environments.

The key contributions of this work include the development of a theoretical framework for CALock, the implementation of a prototype system, and a comprehensive evaluation using benchmark applications. Our findings indicate that CALock outperforms traditional locking mechanisms in terms of scalability and efficiency.

In our humble opinion, this thesis has made significant strides in advancing the state of the art in concurrency control. The development and evaluation of CALock represent a meaningful contribution to the field, and we hope that this work will inspire further research and innovation in this area.

9.1 Contributions

9.1.1 CALock: Path based labelling scheme

Our first contribution is the design of a labelling scheme that effectively captures the topology of a hierarchy. This eliminates the need of traversals to find a guarding ancestor or the lock guard for a lock request. CALock labelling scheme achieves the following:

1. *Effectively identifying guarding ancestors*: The labelling scheme allows a thread to identify the guarding ancestors of a vertex without traversing the hierarchy.

2. *Efficiently finding lock guards*: For a lock request containing one or more vertices, the labels of the vertices are used to find a set of lock guards for that request. This eliminates the need for traversals.

3. *Optimizing the lock guard*: A consideration to minimize grain size is to choose the deepest guarding ancestor as the lock guard. This facilitates a greater degree of parallelism by reducing the probability of grain overlaps that cause thread blocks.

4. *Eliminating false subsumptions*: False subsumptions occur when a thread is blocked by a lock guard that is not an ancestor of the vertex it is trying to lock. CALock eliminates false subsumptions because the labelling scheme ensures that the grains are well-defined, and a guard only protects its descendants.

5.

6. *Supporting dynamic hierarchies*: The labelling scheme is designed to support dynamic hierarchies. When a structural modification occurs, the labels of the affected vertices are updated to reflect the change. This relabelling can also be parallelized in CALock since only the vertices in the affected grain are relabelled.

9.1.2 CALock: Locking protocol

The second contribution of our work is a multi-granularity protocol that leverages the CALock labelling scheme to optimize concurrency control in hierarchical data. The lock protocol leverages a *lock object*, which is a set of fields like guard ID, guard label etc., to distinguish lock requests as well as check for conflicts between them. In addition, a lock pool guarantees safety and fairness in granting lock requests. The CALock protocol achieves the following:

1. *Efficient lock acquisition*: The CALock protocol allows a thread to acquire a lock on a vertex without traversing the hierarchy by using the LGCA of a target vertex in the lock request. Inserting the ID of the LGCA in the lock pool via a lock object is sufficient to indicate the existence of a lock.

2. *Efficient lock conflict detection*: To test for grain overlaps, CALock protocol uses the labels of guards in the lock pool. If the ID of a guard is present in the label of another lock guard, then there is a grain overlap. This significantly reduces the time taken to detect conflicts between lock requests by eliminating traversals or sub-graph matching.

3. *Fair locking*: The CALock protocol ensures fairness in lock acquisition by using a lock pool. Each lock request is assigned a sequence number which is used to determine the order in which locks are granted. This prevents starvation and prevents priority inversion.

9.2 Summary of Findings

The evaluation of CALock is conducted through a series of benchmarks designed to measure its performance against other MGL techniques. The key results are summarized here:

- **Lock throughput improvement:** CALock demonstrates a significant improvement in throughput compared to other MGL techniques. CALock is 4.5 times faster than DomLock and MID, which are the next best MGL techniques, for workloads that consist of only data updates. In workloads that contain structural modifications, CALock is objectively better since DomLock, MID, FlexiGran exhibit extremely poor performance.

- **Scalability:** The throughput and response time of CALock scales with the number of concurrent threads accessing the hierarchy. Intention locks do not scale at all and so does FlexiGran. The performance DomLock and MID improves slightly until 8 threads but then starts to degrade.

- **Reduced lock wait time:** CALock reduces the time threads spend waiting for locks. This is achieved due to two primary reasons. First, the labelling scheme minimizes grain size. Second, the lock protocol does not require traversals to acquire lock and test for conflicts. This results in locks which are not only granted fast but also have a lower probability encountering false subsumptions.

- **Handling dynamic hierarchies:** CALock labelling is designed to handle dynamic hierarchies. When a structural modification changes the topology of the hierarchy, CALock updates the labels without needing a second lock to relabel unlike DomLock, MID and FlexiGran. This allows for parallel

relabelling and ensures that the system remains responsive even when the hierarchy within a locked grain is changing.

The evaluation highlights CALock's significant advantages over existing MGL techniques in terms of throughput, scalability, and adaptability to dynamic hierarchies. By minimizing lock wait times, efficiently handling structural modifications, and scaling effectively with increasing concurrency, CALock addresses the limitations of state-of-the-art approaches like DomLock, MID, and FlexiGran. These results demonstrate that CALock is a robust and efficient solution for hierarchical data synchronization, making it particularly well-suited for workloads involving both data updates and structural changes.

9.3 Future Work

In this work, we have focussed on the design and implementation of CALock for hierarchical data. There are several avenues of improvement which could benefit from CALock as well as avenues where CALock could be extended.

9.3.1 Indexing and querying dimensional data

CALock labelling is designed to optimize grain size and guarantee mutual exclusion for lock requests. However, CALock labelling can also be used to optimize indexing and querying in hierarchical data. CALock labels appear similar to a Dewey decimal system [Swe83] which has proven its efficacy in searching multidimensional data by drilling down using labels. Variants of Dewey decimal system have been developed to improve its efficiency but still, adding a new category or changing classes requires a relabeling of the entire hierarchy. Future work could study the suitability of path based labelling techniques for data involving more than two search dimensions.

Similarly, for index structures in JSON stores or SQL databases, CALock can be used an effective synchronization mechanism to ensure the consistency of index with data. Similar work has been done by Finis et al. [Fin+15] for indexing XML data. Future work should explore avenues of using the concept of LGCA and LGA to optimise hierarhical indices.

9.3.2 Distributed CALock

CALock is designed for multi-threaded environments. However, the challenges of concurrency control are even more pronounced in distributed systems. While highly available distributed systems wouldn't benefit from multi-granularity locking due to the inherent need for progress even when partitions occur, HPC environments could benefit from CALock. Online scheduling systems like YARN [Vav+13] can benefit from the effective hierarchical synchronization to manage resources.

CALock can be used in conjunction with the actor model to facilitate effective thread synchronization. Sang et al. [San+20] designed AEON for actor systems developed in C++ which uses a static hierarchy to identify the nod at which an operation involving multiple actors can be synchronized. In actor systems with dynamic membership, this hierarchy can undergo structural changes. CALock can not only be used to guard the synchronization hierarchy against breaking cchanges by preventing actors from joining or leaving the system while in a critical section but also to ensure that the structural changes to a hierarchy are consistently applied across all actors.

Future work should explore the deign and use of CALock for actor model which could be incorporated into an actor framework like Akka [Wya13], Orleans [Byk+11] or be used for an actor based programming language like Erlang [AVW91] by incorporating CALock in th BEAM VM.

9.3.3 Optimizing CALock for generic graph Systems

One big drawback of CALock is that it designed for semi-structured data. The locking protocol makes use of a labelling scheme which depends on a hierarchy having a designated root. In most graph system use cases like social media, recommendation systems, and logistics, the underlying graph data is unstructured and often does not have a designated root to start labelling.

Future work should explore the suitability of CALock labelling for generic graphs by using heuristics about the structure of the graph to designate a root. For example, in a social network, some users have more connections than others. Let's call such a user a *popular* (σ). Consequently, the probability of a σ user having a path to all users is higher than a non-σ user. Thus, a σ user can be designated as the root of the graph. With this designated root, CALock labelling can be used to optimize concurrency control in graph systems.

A second dimension of structure can be introduced in graphs through workload analysis. For example, in a recommendation system, an edge in the graph has a weight associated with it. With every edge having a weight, the graph consists of paths which are more likely to be traversed than others. We call such paths *hot paths* (η). A labelling scheme, like that of CALock, can be used to prefer hot paths. In doing so, the locking scheme will provide a higher priority to locks on hot paths. This should in theory lead to better performance.

9.4 Closing remarks

Throughout this thesis we delve into the problem of concurrency control in hierarchical data. We have designed, implemented and evaluated CALock, a novel labelling scheme and locking protocol. CALock is designed to optimize concurrency control in hierarchical data by minimizing grain size through its labelling scheme. The evaluation shows promise for CALock in terms of throughput, scalability and response time, especially for dynamic hierarchies where CALock outperforms existing MGL techniques by a significant margin. We believe that CALock is a significant contribution to the field of concurrency control and hope that this work will inspire further research and innovation in this area.

Bibliography

[ABS99] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999 (cited on p. 12).

[AVW91] Joe L Armstrong, RH Virding, and Mike C Williams. "Erlang User's Guide and Reference Manual Version 3.2". In: *Ellemtel Utvecklings AB, Sweden* (1991) (cited on p. 96).

[BS77] Rudolf Bayer and Mario Schkolnick. "Concurrency of Operations on B-Trees". In: *Acta Informatica* 9 (1977), pp. 1–21 (cited on pp. 20, 39).

[Bla98] K. R. Blackman. "Technical note: IMS celebrates thirty years as an IBM product". In: *IBM Systems Journal* 37.4 (1998), pp. 596–603 (cited on p. 12).

[Byk+11] Sergey Bykov, Alan Geller, Gabriel Kliot, et al. "Orleans: cloud computing for everyone". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. SOCC '11. Cascais, Portugal: Association for Computing Machinery, 2011 (cited on p. 96).

[CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. "The OO7 Benchmark". In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*. Ed. by Peter Buneman and Sushil Jajodia. ACM Press, 1993, pp. 12–21 (cited on p. 78).

[CC16] Fernando Miguel Carvalho and João Cachopo. "Optimizing memory transactions for large-scale programs". In: *Journal of Parallel and Distributed Computing* 89 (Mar. 2016), pp. 13–24 (cited on p. 78).

[CM96] Joseph Cheriyan and Kurt Mehlhorn. "Algorithms for Dense Graphs and Networks on the Random Access Computer". In: *Algorithmica* 15.6 (1996), pp. 521–549 (cited on p. 47).

[Dat00] C. J. Date. *An introduction to database systems (7 ed.)* Addison-Wesley-Longman, 2000 (cited on p. 12).

[Fel+16] Pascal Felber, Shady Issa, Alexander Matveev, and Paolo Romano. "Hardware read-write lock elision". In: *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*. Ed. by Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues. ACM, 2016, 34:1–34:15 (cited on p. 78).

[FB15] Ricardo Filipe and João Barreto. "Nested Parallelism in Transactional Memory". en. In: *Transactional Memory. Foundations, Algorithms, Tools, and Applications: COST Action Euro-TM IC1001*. Ed. by Rachid Guerraoui and Paolo Romano. Cham: Springer International Publishing, 2015, pp. 192–209 (cited on p. 78).

[Fin+15] Jan Finis, Robert Brunel, Alfons Kemper, et al. “Indexing Highly Dynamic Hierarchical Data”. In: *Proc. VLDB Endow.* 8.10 (2015), pp. 986–997 (cited on p. 95).

[FH10] Johannes Fischer and Daniel H. Huson. “New common ancestor problems in trees and directed acyclic graphs”. In: *Inf. Process. Lett.* 110.8-9 (2010), pp. 331–335 (cited on p. 43).

[GKN18] K. Ganesh, Saurabh Kalikar, and Rupesh Nasre. “Multi-granularity Locking in Hierarchies with Synergistic Hierarchical and Fine-Grained Locks”. In: *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*. Ed. by Marco Aldinucci, Luca Padovani, and Massimo Torquati. Vol. 11014. Lecture Notes in Computer Science. Springer, 2018, pp. 546–559 (cited on p. 78).

[Geo11] Lars George. *HBase: The Definitive Guide: Random Access to Your Planet-Size Data*. O’Reilly, 2011 (cited on p. 12).

[Gra+08] Tracy Grauman, Stephen G. Hartke, Adam S. Jobson, et al. “The hub number of a graph”. In: *Inf. Process. Lett.* 108.4 (2008), pp. 226–228 (cited on p. 47).

[Gra+75] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. “Granularity of Locks in a Large Shared Data Base”. In: *Proceedings of the International Conference on Very Large Data Bases (VLDB), Framingham, Massachusetts, USA*. Ed. by Douglas S. Kerr. ACM, 1975, pp. 428–451 (cited on pp. 23, 27, 39).

[GKV07] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. “STMBench7: a benchmark for software transactional memory”. In: *Proceedings of the Second European Conference on Computer Systems (EuroSys), Lisbon, Portugal, March 21-23, 2007*. Ed. by Paulo Ferreira, Thomas R. Gross, and Luís Veiga. ACM, 2007, pp. 315–324 (cited on p. 77).

[KN16] Saurabh Kalikar and Rupesh Nasre. “DomLock: a new multi-granularity locking technique for hierarchies”. In: *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Barcelona, Spain, March 12-16, 2016*. Ed. by Rafael Asenjo and Tim Harris. ACM, 2016, 23:1–23:12 (cited on pp. 29, 77, 78).

[KN18] Saurabh Kalikar and Rupesh Nasre. “NumLock: Towards Optimal Multi-Granularity Locking in Hierarchies”. In: *Proceedings of the 47th International Conference on Parallel Processing (ICPP), Eugene, OR, USA, August 13-16, 2018*. ACM, 2018, 75:1–75:10 (cited on p. 78).

[KN19] Saurabh Kalikar and Rupesh Nasre. “Toggle: Contention-Aware Task Scheduler for Concurrent Hierarchical Operations”. In: *Euro-Par 2019: Parallel Processing: 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings*. Ed. by Ramin Yahyapour. Vol. 11725. Lecture Notes in Computer Science. Springer, 2019, pp. 142–155 (cited on p. 39).

[KPR15] Junwhan Kim, Roberto Palmieri, and Binoy Ravindran. “On Scheduling in Distributed Transactional Memory: Techniques and Tradeoffs”. In: *Handbook on Data Centers*. Ed. by Samee U. Khan and Albert Y. Zomaya. New York, NY: Springer, 2015, pp. 1267–1283 (cited on p. 78).

[LY81] Philip L. Lehman and S. Bing Yao. “Efficient Locking for Concurrent Operations on B-Trees”. In: *ACM Trans. Database Syst.* 6.4 (1981), pp. 650–670 (cited on p. 21).

[LHN19] Viktor Leis, Michael Haubenschild, and Thomas Neumann. “Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method”. In: *IEEE Data Eng. Bull.* 42.1 (2019), pp. 73–84 (cited on p. 21).

[LZ14] Peng Liu and Charles Zhang. “Unleashing concurrency for irregular data structures”. In: *36th International Conference on Software Engineering (ICSE), Hyderabad, India, 31 May - 07 June, 2014*. Ed. by Pankaj Jalote, Lionel C. Briand, and André van der Hoek. ACM, 2014, pp. 480–490 (cited on p. 78).

[Mic22] Microsoft. *Transaction locking and Row Versioning Guide - SQL Server*. 2022 (cited on p. 23).

[MAN24] Anju Mongandampulath Akathoott and Rupesh Nasre. “FlexiGran: Flexible Granularity Locking in Hierarchies”. In: *Euro-Par 2024: Parallel Processing*. Ed. by Jesus Carretero, Sameer Shende, Javier Garcia-Blas, et al. Cham: Springer Nature Switzerland, 2024, pp. 3–17 (cited on pp. 35, 77, 78).

[MAN22] Anju Mongandampulath Akathoott and Rupesh Nasre. “Multi-Interval DomLock: Toward Improving Concurrency in Hierarchies”. In: *ACM Trans. Parallel Comput.* 9.3 (2022), 12:1–12:27 (cited on pp. 32, 77, 78).

[Pro+19] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, et al. “Renaissance: benchmarking suite for parallel applications on the JVM”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 31–47 (cited on p. 78).

[Ray13] Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer, 2013 (cited on p. 11).

[RS77] Daniel R. Ries and Michael Stonebraker. “Effects of Locking Granularity in a Database Management System”. In: *ACM Trans. Database Syst.* 2.3 (1977), pp. 233–246 (cited on p. 23).

[RC14] Hugo Rito and João P. Cachopo. “ProPS: A Progressively Pessimistic Scheduler for Software Transactional Memory”. In: *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*. Ed. by Fernando M. A. Silva, Inês de Castro Dutra, and Vítor Santos Costa. Vol. 8632. Lecture Notes in Computer Science. Springer, 2014, pp. 150–161 (cited on p. 78).

[San+20] Bo Sang, Patrick Eugster, Gustavo Petri, Srivatsan Ravi, and Pierre-Louis Roman. “Scalable and serializable networked multi-actor programming”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 198:1–198:30 (cited on p. 96).

[Sha81] Micha Sharir. “A strong-connectivity algorithm and its applications in data flow analysis”. In: *Computers & Mathematics with Applications* 7.1 (1981), pp. 67–72 (cited on p. 47).

[Swe83] Russell Sweeney. “The Development of the Dewey Decimal Classification”. In: *J. Documentation* 39.3 (1983), pp. 192–205 (cited on pp. 40, 95).

[Tar72] Robert Endre Tarjan. “Depth-First Search and Linear Graph Algorithms”. In: *SIAM J. Comput.* 1.2 (1972), pp. 146–160 (cited on p. 47).

[Val+16] Tiago M. Vale, João A. Silva, Ricardo J. Dias, and João M. Lourenço. “Pot: Deterministic Transactional Execution”. en. In: *ACM Transactions on Architecture and Code Optimization* 13.4 (Dec. 2016), pp. 1–24 (cited on p. 78).

[Vav+13] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, et al. “Apache Hadoop YARN: yet another resource negotiator”. In: *ACM Symposium on Cloud Computing, SOCC ’13, Santa Clara, CA, USA, October 1-3, 2013*. Ed. by Guy M. Lohman. ACM, 2013, 5:1–5:16 (cited on p. 96).

[Wya13] Derek Wyatt. *Akka Concurrency*. Sunnyvale, CA, USA: Artima Incorporation, 2013 (cited on p. 96).