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 : *17 Mars 2025*

Devant un jury composé de :

Gaël Thomas, Directeur de Recherche, INRIA	*Président du jury*
David Bromberg, Professeur, Université de Rennes	*Rapporteur*
Pascal Felber, Professeur, Université de Neuchâtel	*Rapporteur*
Sathya Peri, Professeur, IIT Hyderabad	*Examinateur*
Stefania Dumbrava, Maîtresse de conférences, ENSIIE & Télécom SudParis	*Examinatrice*
Mesaac Makpangou, Chargé de Recherche [HDR], INRIA	*Directeur de thèse*
Julien Sopena, Maître de Conférences, LIP6/Sorbonne Université	*Encadrant*
Marc Shapiro, Directeur de Recherche Émérite, INRIA & LIP6/Sorbonne Université	*Encadrant*
Swan Dubois, Maître de Conférences, LIP6/Sorbonne Université	*Encadrant*

To my Family

Remerciements

A PhD is a long journey. For me, it took 3 years, 3 supervisors and education in 3 countries to finally reach here. Throughout this time, I have been supported by many people, without whom I would not have been able to reach here. I would like to take this opportunity to thank them.

First and foremost, I would like to thank the rapporteurs of this thesis, Dr. David Bromberg (Professeur, Université de Rennes) and Dr. Pascal Felber (Professeur, Université de Neuchâtel). I am grateful for their time and effort in reviewing my work and providing valuable feedback. I would also like to thank the members of the jury, Dr. Gaël Thomas (Directeur de Recherche, INRIA), Dr. Sathya Peri (Professeur, IIT Hyderabad) and Dr. Stefania Dumbrava (Maîtresse de conférences, ENSIIE) for taking the time to read, evaluate and provide feedback on this body of work. The members of my CSD committee, Dr. Bernd Amann (Professeur, Sorbonne Université) and Dr. Ahmed Bouajjani (Professor, Université Paris Cité) have been a great source of support and guidance throughout my PhD. I am grateful for their time and effort in helping me navigate the challenges of a PhD and correcting the direction of my research.

The people without whom this thesis would not have been possible are my supervisors, Marc Shapiro, Julien Sopena and Swan Dubois. From asking me if I would like to do a PhD back in 2021 while I was in Germany, to the day I defended my thesis, I cannot thank Marc enough for his constant support, guidance and well-placed reality checks. Julien, you have been a great inspiration, not only in the rigour of research but also in correcting my French which, even after several corrections, is still awful. Swan, you have been a great mentor and helped me navigate the finer details of theoretical computer science. I am grateful for the opportunity to have worked with you all.

To the seniors in the lab with whom I have shared numerous lunches, coffee breaks and discussions, thank you for making my time in the lab enjoyable. Thank you, Pierre, Jonathan, and Alessio for taking care of DELYS and your help in my research. To the members of the DELYS team, Dr. Luciana Arantes, Dr. Philippe Darche, Dr. Frank Petit; Thank you for keeping this amazing research group running.

I have had the pleasure of working with some amazing colleagues during my time in the lab. Saalik, Benoît and Laurent have been great friends and colleagues, helping me navigate french bureaucracy, teaching me bits of french culture and also helping me with my research direction. I am grateful for your friendship and support. To my fellow PhDs, Aymeric, Baptiste, Célia, Daniel and Étienne; Thank you for your discussions and support. I wish you well for your future.

To the lehrstuhl softwaretechnik of RPTU, I extend my thanks, for introducing me to the wonderful world of parallel and distributed systems. I am grateful for the opportunity to have worked with Annette bieniusa, without whom I would not have been able to start my PhD.

Finally, the people without whom this thesis would not have been imaginable. My family, who dealt with years of daily phone and video calls, the fear of the COVID-19 pandemic, the uncertainty of being abroad and the stresses of a PhD, I am eternally grateful for your support. I dedicate this thesis to you Papa, Mummy and Yash. To my friends from France, Germany and India, thank you for your support, your love, and your friendship.

Thank you everyone.

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

Databases are critical components in the infrastructure of modern organizations, serving as systems for the efficient storage, management, and retrieval of vast amounts of data. As businesses and institutions increasingly pivot toward data-driven strategies, databases become paramount, enabling organizations to leverage data for informed decision-making and operational efficiency. The importance of databases extends across various sectors, from retail and finance to healthcare and scientific research, illustrating their versatility.

Consistency in databases is crucial for ensuring the accuracy and reliability of stored data, particularly in environments with concurrent access. It guarantees the database adheres to a defined set of rules and constraints. Without consistency, errors such as data corruption, conflicting updates, or invalid relationships between data entities can undermine the system's integrity. This is vital as consistent data forms the foundation for meaningful analytics, reliable application behaviour, and user trust.

In databases, managing data consistency during concurrent access is a foundational challenge. It can be specified through constraints or invariants, which are conditions on data. A database is in a consistent state when all specified invariants are satisfied. These constraints can be classified into two main types: **Physical** and **Logical**. Physical constraints typically govern relationships within the database. For example, *if record A refers to record B, then record B must exist for the database to remain consistent*. Logical constraints, on the other hand, involve higher-level properties of the data, such as *the sum of money across all accounts in a financial system remains constant*. These constraints collectively ensure that the database behaves predictably and correctly, even under concurrent access.

The ACID (Atomicity, Consistency, Isolation, Durability) model is a set of properties that ensure data validity. Under this model, a transaction is a series of accesses to the database involving reads and writes to specific data items. It is delimited by a *begin* operation that marks the start of the transaction and a *commit* operation that applies all the changes atomically. By grouping multiple accesses into a single unit of work, transactions provide a means to apply updates to the database as an indivisible operation. When applied to an initially consistent database, a correct

transaction must leave it consistent when it commits. This property ensures that the invariants hold before and after every transaction.

However, this notion of correctness is primarily concerned with the final state of the database, and it does not enforce consistency during the execution of a transaction. In practice, applications that interact with a database often assume that transactions will always observe a consistent database state. This expectation goes beyond the pre-application and post-application consistency invariants. To meet this expectation, transactions must be atomic and **isolated**. Isolation ensures that a transaction can execute as though it were the only transaction interacting with the database at that time. Without isolation, a transaction could read or write data in a temporarily inconsistent state, leading to incorrect or unpredictable outcomes. Locks are one of the mechanisms used to enforce isolation. By preventing simultaneous access to the same data by multiple transactions, locks ensure that no transaction observes an intermediate, inconsistent state created by another transaction.

Modern database systems have evolved to handle increasingly complex data models, such as hierarchical structures. Hierarchical data models are widely used in domains where relationships between data elements must be explicitly represented and managed. Examples include **File systems**, where directories and files form a hierarchy; **Organizational structures**, where departments and employees are arranged in levels; **Ontologies**, where concepts are related in a taxonomic order; **XML databases**, where data is structured in a tree-like format; and more recently, **Document stores**, where data is stored as nested objects that contain key-value pairs. These hierarchical models naturally capture relationships across different levels of abstraction, and they often involve multiple users or processes accessing the data concurrently. Typically, vertices represent data elements in these models, while edges capture their relationships or connections.

In such hierarchical models, transactions access data by acquiring locks on the vertices that contain the required data. To maintain atomicity and isolation, a transaction that needs to access multiple data items may require multiple locks, often at different levels of the hierarchy. This poses challenges because the hierarchy's topology and the transactions' access patterns are not constrained. Multi-granularity locking (MGL) is a well-established mechanism for addressing these challenges. MGL allows locks to be managed at varying levels of granularity, enabling *coarse-grain* locks at higher levels of the hierarchy and *fine-grain* locks at lower levels. This flexibility is critical for supporting efficient concurrency control. For example, a transaction that requires broad access to a large portion of the data can acquire a coarse-grain lock at a higher level. In contrast, another transaction that needs

access to specific items can acquire fine-grain locks at lower levels. This approach minimizes contention and maximizes parallelism.

However, implementing MGL effectively in systems with hierarchical and highly interconnected data introduces unique challenges. Hierarchical data structures often have complex topologies with relationships that span multiple levels. Efficiently managing locks in such environments requires careful consideration of both the hierarchy structure and the access patterns of concurrent transactions. Tailoring MGL to these scenarios is essential for achieving scalable and efficient concurrency control. By adapting MGL to hierarchical data models, it is possible to support a higher degree of parallelism while maintaining the consistency of the database. This research explores these adaptations, focusing on techniques that allow MGL to operate effectively in environments characterized by complex data relationships and diverse transaction workloads. These adaptations aim to achieve scalable, high-performance concurrency control that ensures consistency in even the most demanding hierarchical database systems.

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 a significant challenge. Scalability becomes a pressing issue as hierarchies grow more extensive and more intricate. This is particularly true in scenarios where the size and depth of the hierarchy expand significantly or where the relationships between elements deviate from strict hierarchical patterns, approaching those of graph-like data models. These complexities introduce obstacles that make straightforward adaptations of traditional MGL frameworks insufficient.

One of the core challenges lies in efficiently determining a **lockable unit** within these intricate structures. A lockable unit represents a portion of the hierarchy a transaction can safely operate in isolation without causing conflicts with other concurrent transactions. This task is often straightforward in hierarchies with simple topologies, like trees, because the structure guides lock placement. However, identifying appropriate lockable units becomes increasingly tricky in hierarchies characterized by significant depth and irregularity. This process must account for the complexity of the hierarchy's topology, which is dynamic and evolves as transactions update data. As a result, determining lockable units in such scenarios can become

computationally intensive, requiring careful consideration of the hierarchy's structure to maintain efficiency.

Once locks are established, managing and resolving their conflicts poses another substantial challenge. In simple hierarchical data models, conflicts are often limited to well-defined scenarios, such as two transactions attempting to access the same vertex or its immediate descendants. However, in more complex hierarchies, transactions may interact with multiple levels of the hierarchy simultaneously, creating more intricate conflict scenarios. The structure's irregularity and depth further complicate conflict detection and resolution, as transactions may inadvertently interact with distant or seemingly unrelated parts of the hierarchy. This increases the complexity of ensuring that transactions proceed without violating isolation.

To address these issues, it is necessary to extend existing MGL frameworks to better accommodate complex dynamic hierarchical data. Such extensions must focus on two critical aspects: **lock placement** and **conflict management**. Lock placement must be tailored to the unique characteristics of intricate hierarchies, ensuring that lockable units are defined to balance fine-grained control with the need to minimize overhead. Conflict management must scale effectively with the complexity of the hierarchy and the level of parallelism, providing mechanisms to detect and resolve conflicts promptly and efficiently. These enhancements must be designed to ensure that the fundamental goals of MGL—scalability, high performance, and data consistency—are upheld, even in environments with irregular and deeply nested structures.

Extending MGL to meet these requirements makes it possible to support concurrency control in even the most challenging hierarchical models. These adaptations aim to preserve the key benefits of MGL while addressing the unique demands posed by environments characterized by structural irregularity, increased depth, and distributed operation.

Contributions

In this work, we present CALock, a novel MGL protocol specifically designed to address the challenges of concurrency control in complex dynamic hierarchical data models. CALock extends the traditional MGL framework to support hierarchies with arbitrary depth, structural complexity and dynamic topology, enabling efficient and scalable concurrency management across diverse application domains. By adapting MGL principles to account for the intricacies of highly interconnected, dynamic

hierarchical data, CALock ensures high performance and data consistency, even under challenging workloads.

MGL techniques establish a granularity at which a transaction acquires a lock, optimizing access based on the specific needs of the transaction. CALock innovates in this area by employing a partial ordering of the vertices in the hierarchy to determine lock granularity dynamically. This ordering is defined through a vertex labelling scheme, where each vertex is assigned a label that represents a set of its common ancestors. These labels are computed recursively using a breadth-first traversal over the hierarchy, efficiently capturing the topology. At runtime, vertex labels provide a means to identify an appropriate lock granularity for a lock request, tailoring the locking strategy to the transaction's specific access pattern. As a hierarchy evolves, CALock adapts the lock granularity by efficiently updating vertex labels, ensuring that locks remain effective even as the hierarchy structure changes.

One of the core principles of CALock is to minimize the granularity of locks wherever possible. By doing so, CALock ensures that each lock covers only the smallest necessary portion of the hierarchy, thereby improving the level of parallelism available to concurrent transactions. An additional benefit of CALock is its ability to eliminate the need for fixed vertex ordering, a common requirement in many other MGL techniques. This design choice avoids the overhead associated with frequent relabelling of vertices as the hierarchy evolves, reducing computational costs while maintaining the same level of correctness and isolation guarantees as other MGL protocols.

CALock offers a robust solution for concurrency control in complex, irregular hierarchies through its novel approach to lock granularity and adaptive labelling mechanism. These features make it well-suited for environments requiring high scalability and dynamic adaptability.

Publications

The work presented in this thesis has been successively presented in the following publications:

1. A. Pandey, J. Sopena, M. Shapiro, and S. Dubois, "CALock: Multi-granularity locking in dynamic hierarchies," in 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Milan, Italy, June 2025.

2. A. Pandey, J. Sopena, and M. Shapiro, "Diversifying locks for effective synchronization in dynamic graphs," in EuroSys 2024 Doctoral Workshop, Athens, Greece, April 2024 Online.

3. A. Pandey, S. Dubois, M. Shapiro, and J. Sopena, "CALock: Multi-Granularity Locking in Directed Graphs," in ComPas 2023, Annecy, France, July 2023. Online.

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 multi-granularity locking, highlighting the limitations of current techniques.

Chapter 4 provides a theoretical framework for CALock, which includes the concept of the lowest common ancestors in graphs and the problem formulation for lock grain selection. It introduces CALock, our novel multi-granularity locking protocol, and describes its design and implementation. We also discuss the properties of CALock, such as safety, liveness, and fairness and present the complexity analysis of CALock to compare it with state-of-the-art techniques.

Chapter 5 describes the implementation of CALock, including the design choices and optimizations made to improve performance. We present an experimental evaluation of CALock, comparing its performance against state-of-the-art techniques in various scenarios using both micro-benchmarks and macro-benchmarks.

Finally, Chapter 6 concludes the thesis, summarizing our contributions and outlining future research directions in multi-granularity locking for complex hierarchical data models.

2 Background

Concurrency control is crucial in systems where multiple processes or threads simultaneously access shared resources, such as data structures, tables, or files. 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 introduction to multi-granularity locking. We discuss the structure of a hierarchy and the operations that can be performed on hierarchical data structures. We discuss the challenges of hierarchical data structures and the need for specialized locking techniques to manage concurrent access.

2.1 Hierarchical data structures

Hierarchical data models, rooted 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 created to manage the complex manufacturing data associated with the Apollo space program. IMS establishes a well-defined parent-child relationship between data entities, a 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, such 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 the designated root of the hierarchy such that there is a path from R to every other vertex in V.*

2.1.2 Data in hierarchical data structures

Hierarchical data structures are used to represent data that has a natural hierarchical relationship. This data is stored in the form of key-value pairs. Each vertex in a hierarchy has a unique identifier, type, and set of attributes. The data stored in the vertices can be of any kind, ranging from simple data types like integers and strings to complex data types like arrays and objects.

Vertices are connected by edges that represent the relationships between them. These relationships are directional and often labelled. The labels on the edges can be used to represent the type of relationship between the vertices. For example, in a file system, the edges can be labelled as "contains" to represent the relationship between a directory and the files it contains. Figure 2.1 shows an example of a hierarchical data structure with data on vertices and edges. In this hierarchy, vertices represent entities of type "Person". Each person has a unique identifier and the amount of food

they possess. Person vertices are connected via edges labelled `:isFriendTo` with a weight representing the strength of the friendship. So, we can infer that "*Asterix is close friends with Obelix and a casual friend with Getafix*".

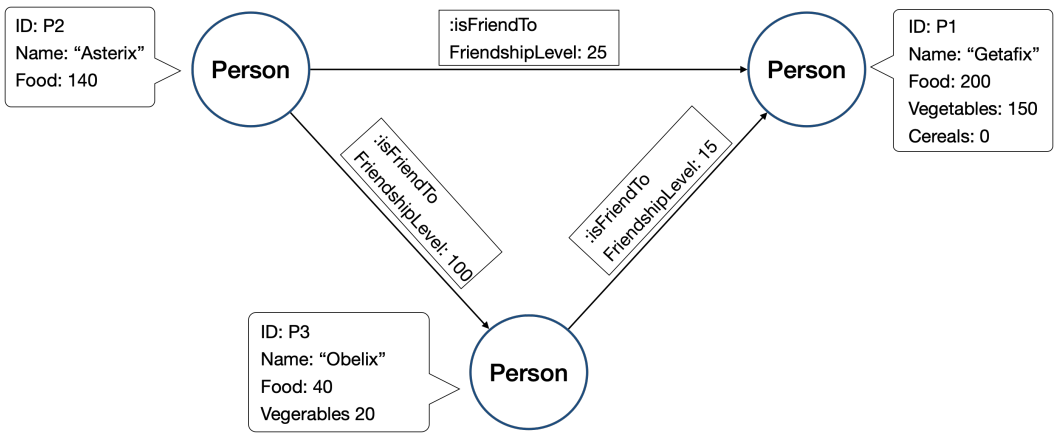


Figure 2.1: An example of a hierarchical data structure with data on vertices and edges.

2.1.3 Operations in hierarchical data structures

Hierarchical data structures support various operations that enable querying and modifying the data. These operations are essential for managing data in the vertices and relationships between vertices of the hierarchy. Common operations on hierarchical data structures fall into two categories:

- **Data read and write:** A data read and write operation involves accessing and/or updating attributes of vertices in a hierarchy. These operations are fundamental for interacting with the data and retrieving information from the structure. One or more vertices are often involved in a single read or write operation, depending on the application's specific requirements. 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 observable 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 Locking, consistency and performance

The primary goal of a multi-granularity locking protocol is to find an effective way of locking an entire *sub-graph*. In a hierarchy, each vertex can be locked individually. This can be done to read or write the data on the vertex. However, in many applications, it is sometimes necessary to lock an entire sub-graph to ensure that the data remains consistent. For example, a file system must lock the directory as a whole to ensure that another concurrent thread does not modify the files within the directory.

When write-locked, the requester has exclusive access to a vertex and implicitly to all its descendants. When read locked, a requester has shared access to the vertex and implicitly to all its descendants. Since MGL aims to lock entire sub-graphs, preventing another transaction from acquiring a lock on a vertex that is an ancestor of a locked vertex is essential. Assume a vertex v is locked in write mode. This implicitly locks the entire subtree rooted at v in write mode. If another transaction acquires a lock on an ancestor a of v, the sub-graph rooted at a, which also contains the sub-graph rooted at v, is implicitly locked. This violates the core invariant of MGL, which is to provide isolated sub-graph access to threads. To prevent this, MGL must ensure that another transaction locks no ancestor of a locked vertex.

An important concern of an application, after correctness, is performance. The performance of a correct locking protocol is determined by the rate at which locks can be granted, i.e., *locking throughput*. A correct application that is slow is often not useful in practice. In database design, this performance is driven by the choice of *lockable units*. A lockable unit is a set of data which should be atomically locked to ensure data consistency. A smaller lockable unit allows for more concurrency and better performance if transactions are simple, i.e., do not access many data items. Complex transactions access several data items and incur significant overhead in acquiring and releasing locks. A larger lockable unit is more efficient in such cases as it reduces this overhead. On the other hand, to ensure consistency, a larger lockable unit discriminates against transactions that access only a few data items by requiring them to wait. It is, therefore, desirable to vary the size of this lockable unit based on the access requirement of a transaction and have varying sizes of lockable units in the system. This is the motivation behind multi-granularity locking.

2.3 Multi-granularity locking: terminology and concepts

This section introduces the key concepts and terminology associated with multi-granularity locking (MGL) in hierarchical data structures. These concepts are essential for understanding the challenges and solutions MGL protocols present. In particular, concepts like "lock grain," which describe the lockable units in hierarchical locking, are critical for understanding the nuances of how concurrency is managed in 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.

Lock target The lock target refers to a specific vertex or a set of vertices within the hierarchy that a transaction accesses. A transaction may access a single lock target or a set of them. A lock target is defined independently of a locking protocol.

Lock guard The lock guard is an entity that serves as the point of synchronization for a given set of lock targets. A thread must acquire the corresponding lock guard to ensure data consistency when accessing a particular 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 depends on the locking protocol. We shall see examples of this in Chapter 3.

Lock grain Lock grain refers to the scope of implicitly locked data when a lock is acquired on a guard. A lock grain is a lockable unit protected by multi-granularity locks. A grain is rooted at a vertex and includes all descendants of that vertex. This root is designated as the lock guard for the grain.

Granularity The size of the grain is called its *granularity*, which varies from coarse-grained locks that encompass large portions of the hierarchy (e.g., entire sub-graphs) 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. In contrast, fine-grained locks allow for greater parallelism; they can introduce additional complexity in managing locks.

2.3.1 Access modes and conflicts

MGL protocols lock grains in either read or write mode. A read lock on a grain allows other transactions to acquire read locks on the same grain but prevents any transaction from acquiring a write lock. A write lock on a grain prevents any other transaction from acquiring a read or write lock. Two lock requests are compatible if they can be granted concurrently. MGL determines this compatibility by two conditions: **mode conflict** and **grain conflict**.

A mode conflict is dependent on the semantics of the lock. MGL uses the standard read/write lock semantics. A write lock guarantees lock grain exclusivity and is incompatible with any other lock request. A read lock allows shared access to a grain and is compatible with other read locks but not write locks. Some MGL protocols, like intention locks, further develop this concept by introducing additional modes to indicate the intent of a transaction to acquire a lock in a specific mode before an actual lock is acquired. Thus, An intention lock is incompatible with a read or write lock.

A grain conflict occurs when two lock requests target overlapping grains. The grain of a lock request is a set of vertices protected by a guard. If the grains of two lock requests have common vertices, they conflict. Different MGL protocols have different strategies for detecting grain conflicts. Intention locks use a deterministic DFS traversal combined with intention locks to prevent grain conflicts. As we discuss in Chapter 3, others use labelling schemes.

2.4 Acquiring MGL locks in a hierarchical data structure

MGL protocols implicitly lock sub-graphs rooted at a lock guard. If transactions can access vertices in the hierarchy randomly, grain conflicts will remain undetected and cause data inconsistency. To prevent this, MGL protocols often assume a defined order of operations for lock acquisition.

1. **Optional preprocessing:** Preprocessing is a one-off step required for certain MGL protocols to prepare a hierarchy for lock acquisition. During preprocessing, metadata required for a locking protocol to function is computed. This metadata, often stored in the vertices of the hierarchy, is then used to optimize lock grain identification and conflict detection.

2. **Preparing a lock request:** Once a transaction identifies the set of data it wishes to access, it must prepare a lock request. This request must include information to identify the lock grain and mode. Different MGL protocols may require additional information, as discussed in Chapter 3.

3. **Requesting a lock:** Once a lock request is prepared, it is submitted to the scheduling mechanism of the locking protocol that decides whether to grant or block the lock request. If blocked, the transaction waits until it is notified to resume. When granted, the transaction proceeds to perform the operation on the vertices. The locks are granted in a deterministic order, often from the root to the leaves of the hierarchy. This is done to guarantee that another transaction locks no ancestor of a locked vertex.

4. **Performing an operation:** the transaction can access the locked grain once a lock is granted. The transaction may read/write data on the vertices or perform structural modifications by adding/removing edges. Since the grain is isolated from other transactions, any changes made by the transaction are not visible to other transactions until the lock is released.

5. **Optional metadata maintenance:** For protocols that use metadata to identify grains, the transaction may need to update this metadata to reflect the changes made to the hierarchy. For example, if a vertex is deleted, the metadata must be updated to reflect this change. Not all MGL protocols require this step.

6. **Releasing the lock:** Once the transaction completes its operation, it releases the lock on the grain. This makes the updates visible to other transactions. This grain is now available for other transactions to lock and access. Locks should be released from the leaves to the root to ensure an ancestor is not unlocked before its descendants are unlocked.

The correctness argument of a locking protocol makes a few assumptions about the implementation and use.

1. **Atomicity of lock operations:** We assume 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 acquire a lock on the same grain.

2. **Cooperating threads:** The algorithm assumes that threads behave cooperatively without attempting to bypass or tamper with the locking mechanism. 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 scheduler, meaning a lock request is eventually granted in finite time, avoiding starvation where certain threads are indefinitely blocked. Fairness ensures that all threads have equitable access to shared resources.

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 transactions access a consistent version of the graph data, preventing issues such as stale data reads or lost updates.

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

2.5 Design guideline for a multi-granularity locking protocol

A locking protocol based on the steps described in Section 2.4 performs three key functions: Identifying the lock grain, detecting conflicts between lock requests, and managing the metadata required to implement the locking protocol. Each of these can be optimized to improve the system's throughput. However, these optimizations are often at odds with each other. As we will describe in more detail in Chapter 3, a locking protocol that uses a deterministic traversal to detect conflicts can achieve smaller grains but incurs a higher penalty for identifying lock conflicts. On the other hand, a locking protocol that eliminates traversals, using metadata to detect conflicts, needs to optimize metadata management to increase throughput. The three primary requirements of a correct MGL technique are:

$\mathbf{R_{Grain}}$ *Finding an appropriately sized grain for a request*. When a thread requests a lock on a set of lock targets, a locking protocol must determine a lock

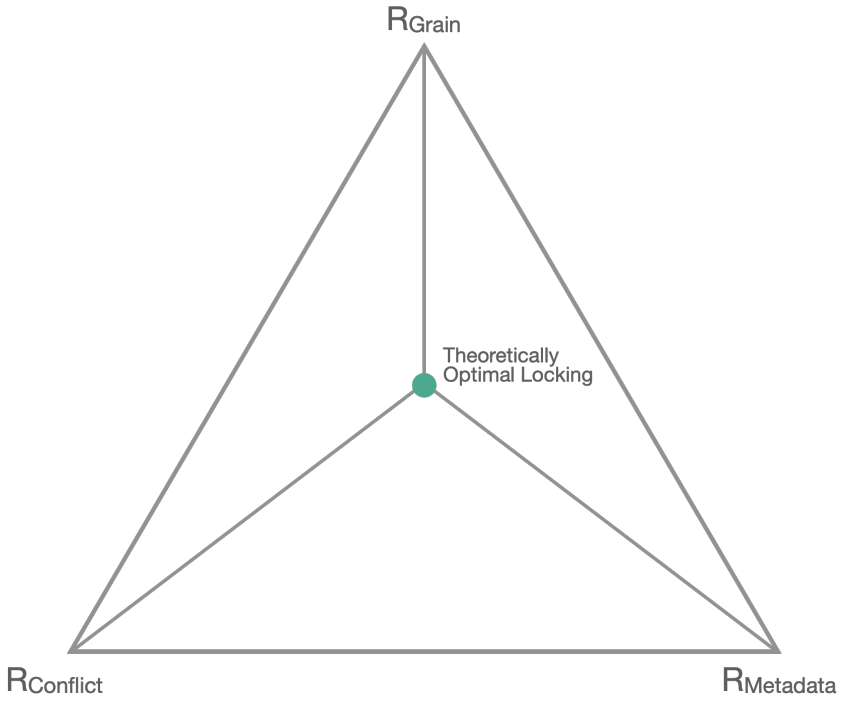


Figure 2.2: An optimal MGL technique balances the trade-offs between the three requirements to maximize throughput.

grain for this request such that a lock on said grain maximizes throughput. This involves considering the topology of the hierarchy and the relationships between vertices to minimize grain conflicts.

$R_{Conflict}$ *Efficiently detecting conflicts between locks*. Mode conflicts and grain conflicts between lock requests must be detected correctly to guarantee data consistency and efficiently to improve lock throughput. Mode conflicts can be detected by maintaining a lock table that records the lock mode for each transaction. Grain conflicts are more complex and often require sub-graph traversal to identify overlapping grains. A locking protocol must ensure that a locked grain is not accessed by a transaction with a conflicting mode. Several MGL protocols use additional metadata to optimize the detection of grain conflicts.

$R_{Metadata}$ *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 metadata management is prohibitively expensive for specific workloads, the locking protocol may not be practical for real-world applications that encounter those workloads.

The goal of a correct MGL technique is to maximize throughput. This throughput is measured by the number of lock requests that can be granted in a unit of time. Depending on the locking protocol, this can translate to optimizing the former three requirements. For example, a locking protocol that uses a deterministic traversal to detect conflicts can detect conflicts more efficiently by implementing graph-isomorphism tests. This, in turn, increases the system's throughput but incurs a higher memory footprint. A locking protocol that uses labelling omits traversals altogether and needs to optimize the metadata management to increase throughput.

3 Related work

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 its descendants in the same mode. This powerful concept allows for efficient locking of hierarchical data structures without explicitly locking large portions of the hierarchy. While powerful, implementing an MGL protocol that successfully achieves this goal is non-trivial.

The locking approaches discussed in this chapter use different mechanisms to address the requirements specified in Chapter 2, each with its trade-offs. It would be appropriate to claim that no approach is all-encompassing, and the locking approach choice depends on the application's specific requirements and the hierarchy being used. We present each lock mechanism's strengths, weaknesses and niche.

Finally, we present the trade-offs between them and motivate the need for a new hierarchical locking protocol that addresses the existing protocols' limitations by classifying them based on the requirements they fulfil.

3.1 Classical locking approaches for connected data

A common approach for synchronization in connected data, like lists and trees, 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.

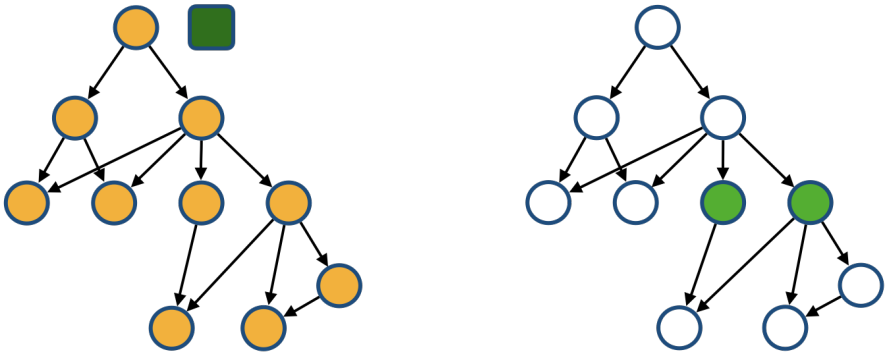
An alternative to lock coupling, targeted primarily for B-trees, is a variant called B-Link tree [LY81]. A B-link tree is an extension of the B-tree data structure that adds sibling pointers to improve concurrency and simplify lock management in multithreaded environments. In a traditional B-tree, nodes are connected in a

hierarchical structure, containing keys and pointers to child nodes. Still, 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 locking 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 only applies to tree-based structures and is not 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.

3.2 Fixed-grain locking techniques

Unlike classical approaches, which define a specific synchronization mechanism, read/write locks are often appropriated for hierarchical data by associating a lock guard with 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. All vertices with the same depth belong to the same grain, which does not depend on the lock request. We refer to this as *fixed-grain locking*. Fixed-grain locking has two extremes: We can 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

Figure 3.1: Fixed-grain locks in a hierarchical data structure (lock guard in green and the corresponding grain in yellow).

3.2.1 Coarse-grain locks

An oversimplified approach for correct thread synchronization is to guard an entire hierarchy with a single read-write 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 3.1a. 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.

3.2.2 Fine-grain locks

Another well-known approach to locking is to designate every target vertex as its own guard, i.e., every vertex has its own unique lock. We call this *fine-grain locking*. As shown in Figure 3.1b, 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.

3.3 Multi-granularity locking techniques

In contrast to fixed-grain approaches, *Multi-granularity locking* (MGL) [Gra+75] 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 transaction that requests a write (exclusive) lock on a guard vertex implicitly exclusively locks all its targets when its request is granted.

MGL 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 the lock request, lock requests with different lock targets 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].

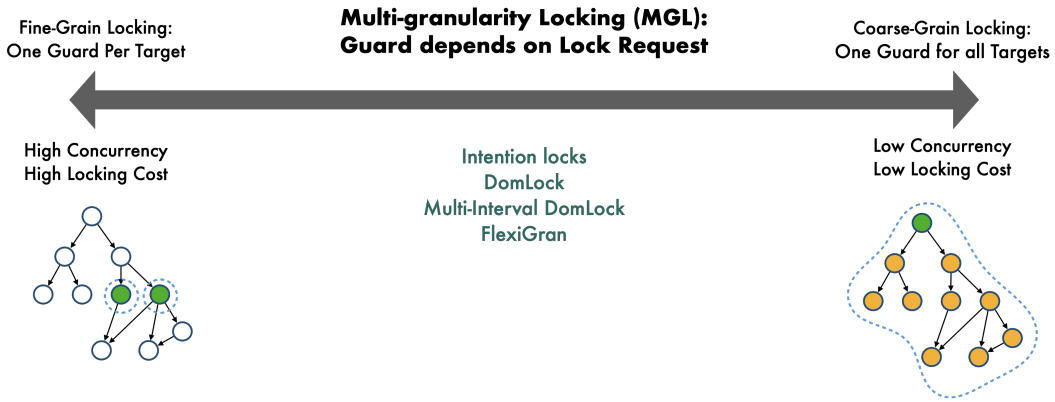


Figure 3.2: Multi-Granularity locking provides a balance between fine-grain and coarse-grain locking.

3.3.1 Intention lock

To lock a sub-graph rooted at a vertex v, it is necessary to prevent other threads from locking the ancestors of v in shared or exclusive mode. To achieve this, Gray et al. [Gra+75] introduced the concept of *intention locks*. Intention locks coordinate concurrent access by indicating a thread's intention to acquire more fine-grain locks within a hierarchy.

Intention locking was one of the first approaches to hierarchical locking designed to efficiently meet requirements $\mathbf{R_{Grain}}$ and $\mathbf{R_{Conflict}}$. Intention locking uses three new lock modes, IX (Intention Exclusive), IS (Intention Shared), and SIX (Shared with 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 any descendant of the vertex locked under IX, IS or SIX modes, respectively. In the intention lock protocol, a thread acquires these intention locks on all vertices on all 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 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

Table 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 3.3 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 (A) and acquire intention-exclusive locks (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 acquire an exclusive lock on G.

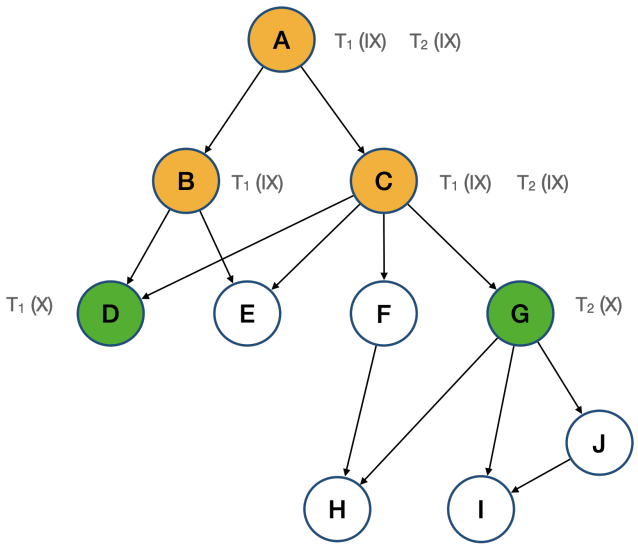


Figure 3.3: Multi granularity locking using intention locks. T_1 takes an exclusive lock on vertex D and T_2 takes an exclusive lock on vertex G.

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 to lock all paths. In the example of Figure 3.3, T_1 must perform the following two traversals:

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

Once a thread has acquired the lock and performed the necessary accesses, it releases the locks from the leaves to the root. This, again, requires traversals. In the example in Figure 3.3, T_1 must release the X lock on D. Then, backtrack towards the root to release the IX locks on B and C and finally release the IX lock on A. This involves the following two traversals:

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

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 and release 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 must be deterministic to prevent deadlocks.

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

3.3.2 DomLock

DomLock [KN16] is an 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 on all paths from the root to a target vertex v. A dominator 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 the root to v pass through d.*

A vertex can have several dominators. To maximize concurrency, DomLock uses the *dominator of maximum depth* as the lock guard. This dominator is called the immediate dominator.

Definition 3 (Immediate dominator). *A dominator $\mathcal{D}$ is an immediate dominator of another vertex v if no other dominator exists for v on the paths between $\mathcal{D}$ 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 descendants since any traversal to the target vertex or its descendants shall encounter the immediate dominator.

Preprocessing and metadata

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 a vertex v is denoted $I_v = [l_v, r_v]$. A vertex u is a dominator of another vertex v if the interval of u (I_u) subsumes the interval of v (I_v). Subsumption ($\prec$) is defined as follows.

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

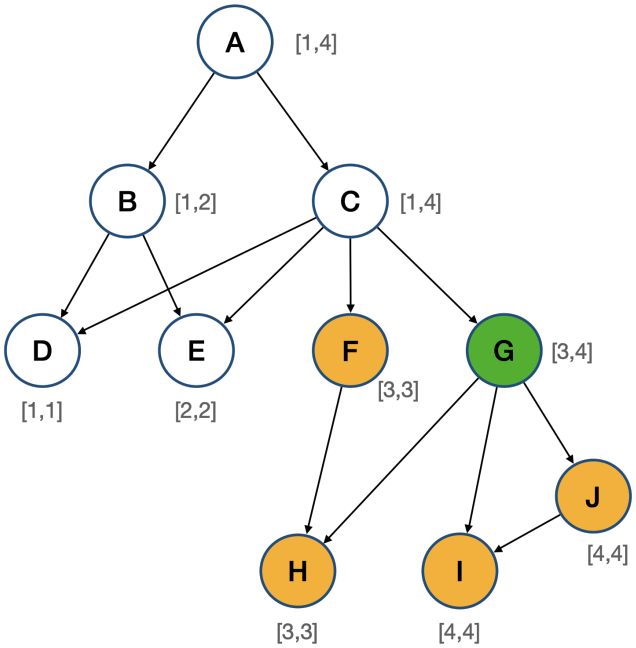


Figure 3.4: Hierarchy labelled with DomLock intervals and DomLock on G (lock guard) with the grain of the grain of the lock (yellow).

A post-order traversal of the hierarchy computes these intervals. Consider the example in Figure 3.4. A leaf is labelled with a unit interval. For example, vertex D is labelled with the interval $[1, 1]$ since it is the first vertex in the post-order traversal.

The interval of an internal vertex is computed from its children's. The l value of an internal vertex v is the minimum of the l values of its children, and its r value is the maximum of the r values of its children. For example, in Figure 3.4, 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 is a dominator of {H, I, J}

Lock acquisition and conflicts

To identify the lock grain, DomLock uses the intervals of the targets of a lock request. The immediate dominator is chosen as the lock guards, given a set of lock targets. This is done by finding the vertex with the smallest interval that subsumes the intervals of all lock targets. Finding the immediate dominator involves a depth-first search from the root of the hierarchy.

For example, in Figure 3.4, suppose a thread wants to acquire a lock on vertices H and J with intervals $[3, 3]$ and $[4, 4]$, respectively. The interval of the guard should subsume the intervals of H and J. The smallest possible interval to achieve this is $[3, 4]$. A depth-first search is performed to identify the deepest vertex with interval $[3, 4]$. DomLock identifies G as the lock guard for this lock request. The grain of G contains F, H, I and J since the interval of G subsumes the intervals of F, H, I and J which are $[3, 3]$, $[3, 3]$, $[4, 4]$ and $[4, 4]$ respectively.

By using numeric intervals, DomLock approximates the ancestor-descendant relationship in the hierarchy. The subsumption property aims to ensure that the interval of a vertex v only subsumes the intervals of all its descendants. However, this is not always the case. A post-order traversal assigns intervals. These intervals sometimes cause false positives when identifying the ancestor-descendant relationship. We call such false positives, *False subsumptions*. False subsumption occurs when the intervals of two vertices overlap, but they are not related by an ancestor-descendant relationship. For example, in Figure 3.4, G is a dominator of F, but F is not a descendant of G. Due to false subsumption, disjoint subgraphs are sometimes locked together since they are associated with the same lock guard. This leads to spurious lock conflicts and, consequently, performance degradation.

To test for a **grain conflict** between two lock requests, the intervals of the lock guards of the two requests are compared. Overlapping intervals indicate a conflict. For example, in Figure 3.4, along with a lock on G, another lock can be acquired on B since the intervals $[3, 4]$ and $[1, 2]$ respectively do not overlap. However, a lock on C will conflict with a lock on G since the interval of C ($[1, 4]$) overlaps with the interval of G ($[3, 4]$).

Metadata maintainance

Another drawback of DomLock is that it does not support dynamic hierarchies without relabelling a large part of the hierarchy. Since intervals are used to identify lock grains, they must be recomputed when a structural modification occurs.

For example, consider the hierarchy in Figure 3.5. If a vertex K is added as a child of G after H, K can only have interval $[4, 4]$. This forces I and J to get interval $[5, 5]$ and the interval of G must be recomputed to $[3, 5]$. This re-computation has to be done for all the ancestors of G as well up to the root of.

Furthermore, this re-computation might not be parallelizable with any other access. Since the interval of the root is recomputed as well, any other operation, whether

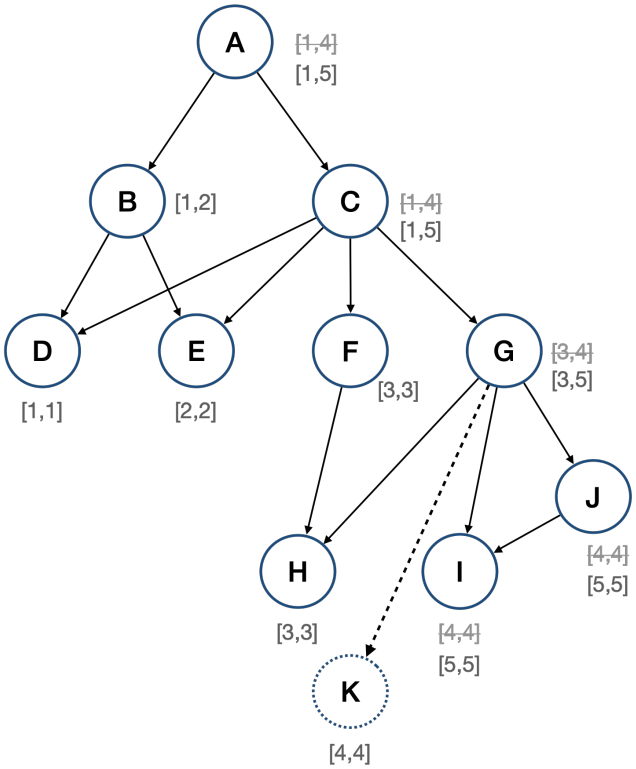


Figure 3.5: DomLock interval re-computation for a vertex insertion.

read, write or structural modification, has to wait until the re-computation of the root is complete. A structural modification is performed under a write lock on the root of a hierarchy to prevent concurrent accesses from interfering with the re-computation. This blocks all other accesses until the structural modification and the interval re-computation is complete.

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

3.3.3 Multi Interval DomLock (MID)

Multi Interval DomLock [MAN22] is a successor to DomLock that uses a pair of intervals per vertex to identify the immediate dominator. In addition to a DomLock interval, MID maintains another interval computed by a reverse post-order traversal of the hierarchy, called the "*DFS-on-image*". Figure 3.6 shows the MID intervals for a hierarchy.

Preprocessing and metadata

A vertex v is labelled with two intervals: the $\mathcal{D}$ interval ($ID_v = [lD_v, rD_v]$) and the $\mathcal{M}$ interval ($IM = [lM_v, rM_v]$). The only difference between the two intervals is the order of traversal used to compute them. $\mathcal{D}$ intervals being post-order and $\mathcal{M}$ intervals being reverse post-order. Leaves of the hierarchy are labelled with unit $\mathcal{D}$ and $\mathcal{M}$ intervals.

For example, in Figure 3.6, vertex H is labelled with $\mathcal{M}$ interval $[2, 2]$ in addition to its $\mathcal{D}$ interval $[3, 3]$ since it is the second vertex in the reverse post-order traversal. Like DomLock, the interval of an internal vertex is computed using its children's intervals. For both $\mathcal{D}$ and $\mathcal{M}$ intervals of a vertex, the l value of an internal vertex v is the minimum of the l values of the respective interval of its children and the r value of an internal vertex v is the maximum of the r values of the respective interval of its children.

For example, in Figure 3.6, vertex G has three children H, I and J. The $\mathcal{D}$ and $\mathcal{M}$ intervals of G are $[3, 4]$ and $[1, 2]$ respectively, since the minimum l values of its children are 3 and 1 and the maximum r values of its children are 4 and 2 respectively.

Intervals of a vertex u subsume ($\prec$) overlapping intervals of all descendants of u. A vertex u is a dominator of another vertex v if both $\mathcal{D}$ and $\mathcal{M}$ intervals of u subsume the respective intervals of v. Subsumption is defined as follows:

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

For example, in Figure 3.6, G is a dominator of F, H, I and J since both $\mathcal{D}$ and $\mathcal{M}$ intervals of G subsume the respective $\mathcal{D}$ and $\mathcal{M}$ intervals of F, H, I and J.

Lock acquisition and conflicts

In order to identify the lock grain, MID uses the $\mathcal{D}$ and $\mathcal{M}$ intervals of the targets of a lock request. The immediate dominator is chosen as the lock guard. The immediate dominator, with MID, is the deepest vertex that subsumes the $\mathcal{D}$ and $\mathcal{M}$ intervals of the lock targets. Like DomLock, finding the immediate dominator involves a depth-first search from the root of the hierarchy.

For example, in Figure 3.6, suppose a thread wants to acquire a lock on vertices H and J with $\mathcal{D}$ intervals $[3,3]$ and $[4,4]$ and $\mathcal{M}$ intervals $[2,2]$ and $[3,3]$ respectively. The lock guard is a vertex with $\mathcal{D}$ interval $[3,4]$ and $\mathcal{M}$ interval $[1,2]$. A depth-first search identifies G as the lock guard for this request.

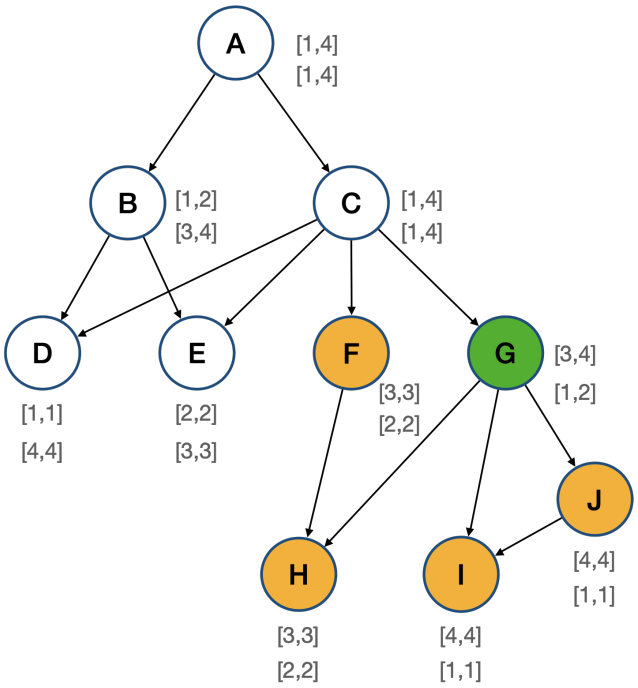


Figure 3.6: MID labels with lock on guard G with the grain of the lock (yellow)

The property of subsumption is used to identify the lock grain of a guard in MID. However, unlike DomLock, subsumption is tested on both $\mathcal{D}$ and $\mathcal{M}$ intervals. This is done to reduce the number of false subsumptions. However, MID still suffers from false subsumptions.

For example in Figure 3.6, G subsumes F, H, I and J since both intervals of G overlap with the respective intervals of F, H, I and J. 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 spurious conflicts.

Grain conflicts are identified by testing the $\mathcal{D}$ and $\mathcal{M}$ intervals of the lock guards for overlap. The grains of two lock guards are disjoint only if both $\mathcal{D}$ and $\mathcal{M}$ intervals are disjoint. Two disjoint grains can be locked concurrently.

Metadata maintenance

MID encounters double the penalty for recomputing vertex labels in dynamic hierarchies when a structural modification occurs. Since two intervals are computed per vertex, a post-order and a reverse post-order traversal are required to recompute them when a structural modification occurs. In some instances, the intervals of all vertices are recomputed, which is extremely expensive for large hierarchies.

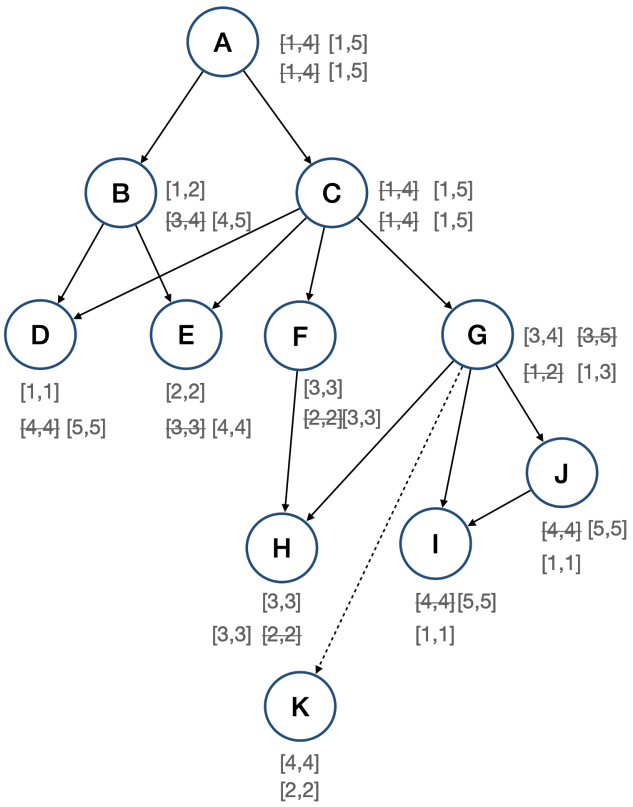


Figure 3.7: MID interval re-computation for a vertex insertion.

For example, when vertex K is inserted as a child of G as shown in Figure 3.7, at least one interval for every vertex is recomputed. These intervals are then propagated to the root of the hierarchy. Compared to DomLock, MID relabels more vertices for the same structural modification.

Like DomLock, a structural modification is performed under a write lock on the root of the hierarchy to prevent concurrent reads from interfering with the re-computation.

MID tries to reduce false subsumptions w.r.t to DomLock but does not eliminate them completely. The performance penalty for re-computation in dynamic hierarchies

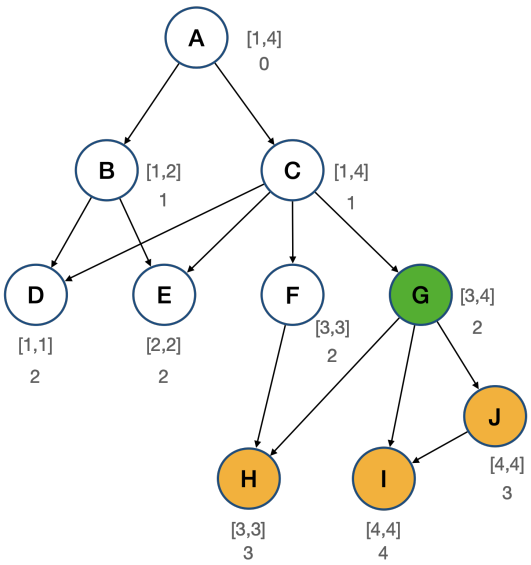


Figure 3.8: FlexiGran labels and vertex depth with lock on guard G with the grain of the lock (yellow).

is significantly higher in MID than in DomLock. As such, MID fulfils requirement $\mathbf{R_{Grain}}$ but incurs even severe penalties, compared to DomLock, against requirements $\mathbf{R_{Conflict}}$ and $\mathbf{R_{Metadata}}$.

3.3.4 Flexible granularity locking (FlexiGran)

FlexiGran [MAN24] aims to enable both MGL via DomLock and fine-grain locks to co-exist on the same hierarchy.

FlexiGran uses the DomLock intervals as vertex labels and uses vertex depth to determine ancestor-descendent relationships between identical intervals.

Preprocessing and metadata

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 their children's intervals. In addition to these intervals, FlexiGran computes each vertex's depth in the hierarchy. The depth of a vertex is the length of the shortest-longest path from the root to the vertex. Figure 3.8 shows a hierarchy's intervals and vertex depths labelled with FlexiGran.

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

Lock acquisition and conflicts

If a thread requests an 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 read/write lock is directly requested on the target.

Since fine-grain and MGL locks can exist in FlexiGran, 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 locks 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, they conflict if at least one of the 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

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

To test the compatibility of an MGL lock with a fine-grain lock, it is necessary to perform a reachability check between the guards of the MGL lock and the fine-grain lock. To do so, a search is initiated from the guard of the MGL lock, and if the fine-grain lock guard is reachable from the MGL lock guard, then the two locks are incompatible since they lie in the same MGL grain.

For example, in Figure 3.8, a hierarchical lock is acquired on G—this lock guards vertices H, I and J. 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. Unlike DomLock and MID, A FlexiGran lock on G does not subsume F as F is not a descendant of G since their depths are the same.

Metadata maintenance

Like DomLock, structural modifications in FlexiGran require re-computation of the intervals of the vertices. A single post-order traversal is enough to recompute the intervals and depths of vertices.

Like DomLock, a structural modification with FlexiGran is performed by acquiring a write lock on the root of the hierarchy since the root might be relabelled.

In summary, FlexiGran fulfils requirement $\mathbf{R_{Grain}}$ 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 re-computation of intervals in dynamic hierarchies.

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

Recall the primary requirements identified for MGL:

R_{Grain} Finding an appropriate lock guard for a lock request.

$R_{Conflict}$ Efficiently Detecting conflicts between locks

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

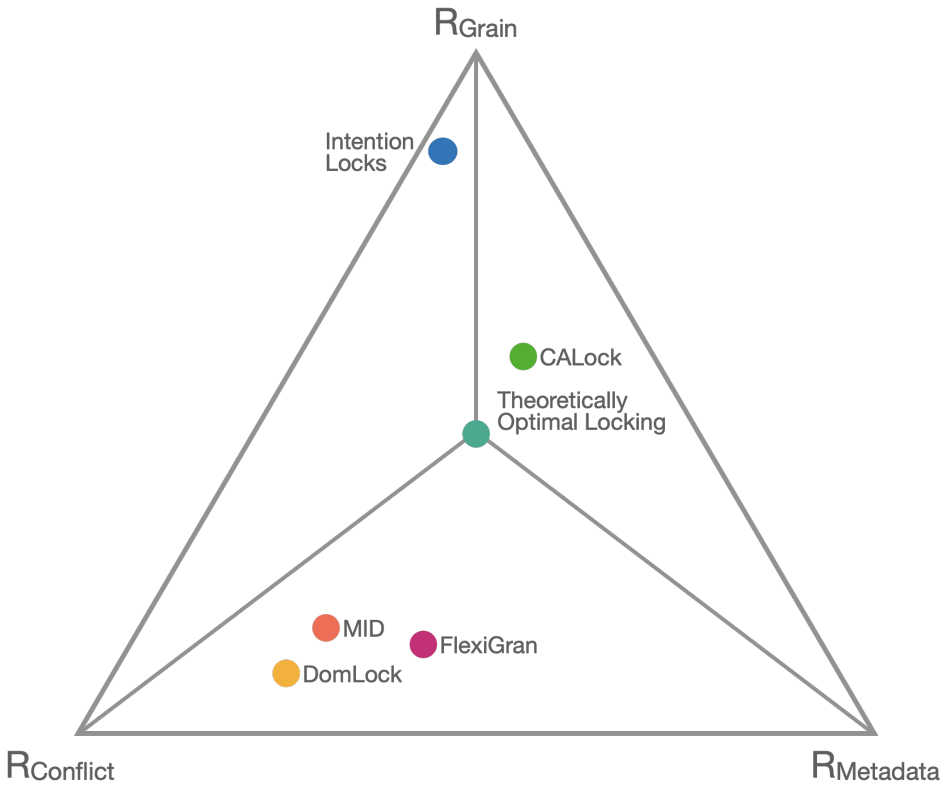


Figure 3.9: Trade-offs in MGL techniques.

MGL based only on paths, like Intention Lock, fulfils requirement $\mathbf{R_{Grain}}$ but incurs a significant performance penalty against requirement $\mathbf{R_{Conflict}}$ due to the number of traversals required to lock a vertex and detect conflicts.

MGL, using labels based on a vertex order such as DomLock, MID and FlexiGran, fulfils requirement $\mathbf{R_{Grain}}$ and incurs a penalty due to false subsumptions. In addition, due to the metadata required to implement the locking protocol, these techniques incur a performance penalty against requirement $\mathbf{R_{Metadata}}$ when structural modifications occur.

Algorithm	R_{Grain}	$R_{Conflict}$	$R_{Metadata}$
Intention Lock	Y	N	N/A
DomLock	N	Y	N
MID	N	Y	N
FlexiGran	N	Y	N
CALock	Y	Y	Y

Table 3.3: Comparison of MGL techniques against requirements.

Figure 3.9 and Table 3.3 show the trade-offs in the different MGL techniques. Intention locks offer the smallest lock grains at the cost of lock conflict detection. DomLock and MID offer efficient conflict detection at the cost of metadata management. FlexiGran offers better lock grains than others but incurs a performance penalty due to the expensive lock conflict detection. None of the techniques balances all the requirements to come close to a balanced MGL protocol.

3.5 Improving the efficiency of MGL techniques

The locking techniques discussed in this chapter are based on the graph's topology or the workload's nature. 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.

3.5.1 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 acquired one after the other on a path to the vertex to prevent concurrent access. In comparison, 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 Section 5.3, the performance of Intention locks degrades significantly when locking over hierarchies.

3.5.2 Label based techniques

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

3.5.3 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 sub-classification. Other path-based techniques use similar 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. CALock labelling scheme balances maintaining both topological information of the hierarchy while efficiently locking and relabelling.

3.6 CALock: A topological multi-granularity locking technique

We propose a new labelling scheme, based on path properties, that supports multi-granularity locking. *Common-Ancestor lock* (*CALock*) is based on a labelling that efficiently computes the closest common ancestor of a set of vertices in a rooted directed graph. By choosing the closest guarding common ancestor of a set of lock targets as their guard, CALock minimizes grain size. By avoiding a fixed ordering of vertices when using paths, 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 guarding ancestors, computed recursively via breadth-first traversal, which we discuss in Section 4.2.

4 CALock: Topological multi-granularity locking

This chapter presents the mathematical foundation, design, and properties of CALock labelling and locking mechanism. We begin by introducing the theoretical underpinnings of CALock, including the definitions of graphs, paths, and vertex relationships. We then discuss the relationships between paths and how they can be used via a labelling scheme to determine a locking grain. We then present the design of a locking and conflict detection protocol that uses said labelling scheme to determine the locking grain for a lock request. Next, We discuss how the labelling scheme handles structural modification operations. Finally, we present a discussion on the formal properties of CALock and its comparison with other state-of-the-art locking mechanisms viz-a-viz time complexity.

4.1 Topological labelling

CALock uses the topological information of a graph to determine the 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 common vertices 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 efficiently and effectively allows threads to determine a locking grain for their lock requests without traversing the hierarchy. CALock labelling is based on the concept of common ancestors introduced by Fischer and Huson [FH10]. Our labelling scheme extends their work on DAGs to general rooted graphs, which may contain connected components like cycles.

In this chapter, we present the formal definitions and theoretical underpinnings of the labelling scheme of CALock. We introduce 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 a locking grain. We formulate the locking grain problem mathematically and prove its correctness. Finally, we

present the CALock labelling function and discuss its implementation over different use cases.

4.1.1 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 designated 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 relation $A(u)$ gives the set of ancestors.*

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

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 $GA(u)$. This set is given by:*

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

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

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
$LGA(u)$	The Lowest guarding ancestor of u
T_G	LGA tree for a graph G
$LCA(u, v)$	The Lowest common ancestor of u and v
$LGCA(Q)$	The Lowest guarding common ancestor of Q
L_u	Label of vertex u

Table 4.1: Terms used in the definitions.

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:*

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

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

4.1.2 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 and Huson [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 be its corresponding LGA tree. Further, let v, w $\in$ 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. $LGCA_G(w_1, ..., w_k) = LGCA_G(w_1, LGCA_G(w_2, ..., w_k))$

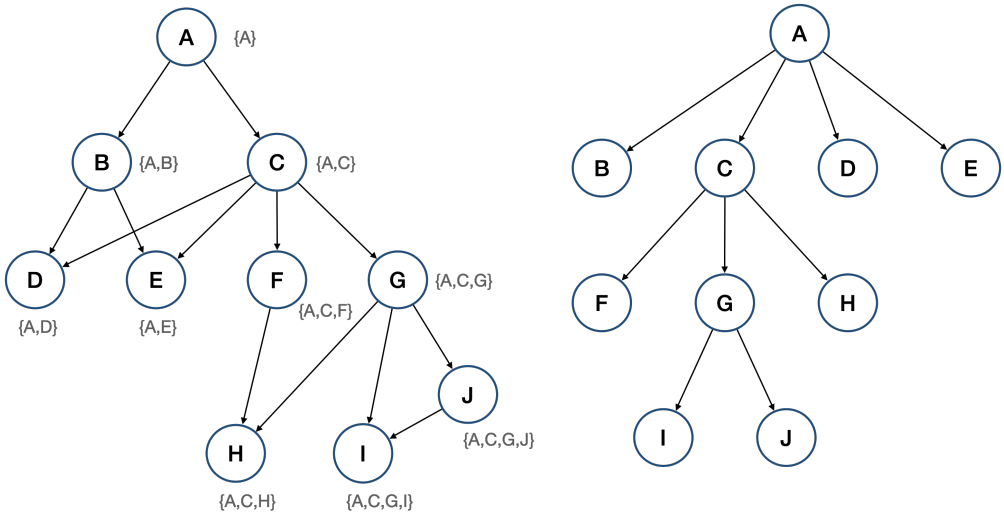


Figure 4.1: A rooted graph with CALock labels and its corresponding LGA tree.

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

4.1.3 Characteristic sets: sets of guarding ancestors

We use 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. Since T_G is a tree, this path is unique, and so is the label L_u. Figure 4.1 shows a rooted graph and corresponding LGA tree.

Computing this partial ordering of vertices in a graph is expensive since it requires examining all paths between the root and a vertex for all vertices in the graph. To compute the label of a vertex $u \in G$ without needing to compute the LGA-tree T_G, we use Lemma 1 to define a recursive function. The $LGCA$ of a set of vertices is the last vertex in an ordered intersection of the characteristic sets 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 must show that l is the deepest vertex in $L_v \cap L_w$.

Assume that a vertex $l' \neq l$ 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, 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 $L_v \cap L_w$. This 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. □

4.1.4 CALock labelling scheme

CALock uses the characteristic set L_u, a set of the guarding ancestors, as the label for a vertex u. In the implementation, a characteristic set is computed via a recursive relation. This recursive relation is defined using the definitions and lemmas from Section 4.1.2. In this section, we incrementally derive the relation and show its robustness against different topological extremes like cycles and connected components. An implementation of this function is shown in Algorithm 1.

Labelling function

We combine Theorem 1 with the definitions and lemmas from Section 4.1.2 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 Definition 10 as follows:

$$\begin{aligned} &LGA_G(v) = LGCA_G(p_1, LGCA_G(p_2, ..., p_k)) \\ &\text{where } p_1, p_2, ...p_k \text{ are the parents of } v \end{aligned} \tag{1}$$

Combining equation Equation (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.

$$L_v = \begin{cases} \{v\} & \text{v is the root} \\ \{\cap_{u \in parents(v)} L_u\} \cup \{v\} & \text{otherwise} \end{cases} \tag{2}$$

Labelling and relabelling a rooted graph

The recursive labelling function derived in Section 4.1.4 is used to compute the labels of each vertex in a hierarchy. An implementation of this function is shown in Algorithm 1. Label computation begins from the root and terminates when all paths have been explored. This function is robust to different topological extremes of the graph. Algorithm 1 can be used to label acyclic graphs and graphs with strongly connected components. The following sections discuss the labelling and relabelling of acyclic graphs and strongly connected components.

Labelling an acyclic graph Labelling starts at the graph's root with the function `AssignLabel(r)`. It uses Equation (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 4.2 vertex A has the label $\{A\}$.

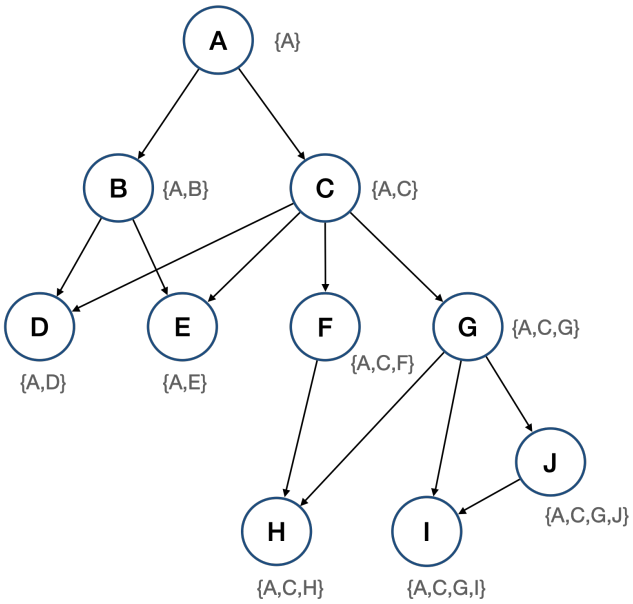


Figure 4.2: CALock labels on a hierarchy.

Then, the children of the vertex v are explored via a breadth-first traversal over the graph. A child's label is computed using the parents', via Equation (2). For example,

in Figure 4.2, 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 Equation (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).

This recursion reaches a fix-point in acyclic graphs after exploring all the paths to leaf vertices. By exploring 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 to avoid re-computation as much as possible.

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 )
```

Labelling a strongly connected component A strongly connected component is a maximal sub-graph of a directed graph where every vertex is reachable from every other. In Figure 4.3, the cyan vertices form a strongly connected component.

Ensuring that a graph does not contain strongly connected components is difficult in real-life applications. 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 undesirable in applications that store data in the graph's vertices. By contracting the vertices, we lose the information stored in the vertices

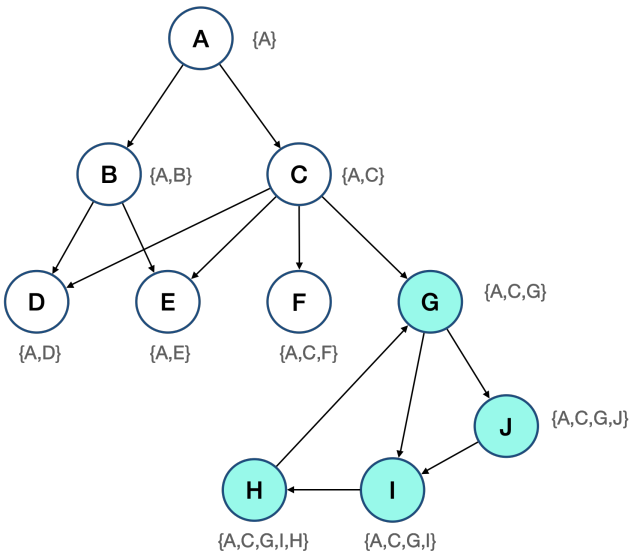


Figure 4.3: CALock labels for a hierarchy containing strongly connected component (cyan).

of the strongly connected component, which is unacceptable. With the recursive Equation (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 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 4.3, 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 change the paths that lead to a vertex from the root. Since the paths to vertices have changed, the labels of vertices need to be recomputed.

In the same fashion as the initial labelling, the parents of v are used to compute its label and recursively of its descendants. Relabelling terminates when it reaches a fix-point (line 15). Consider the example in Figure 4.4. When a new vertex K is added 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 for a mutex to

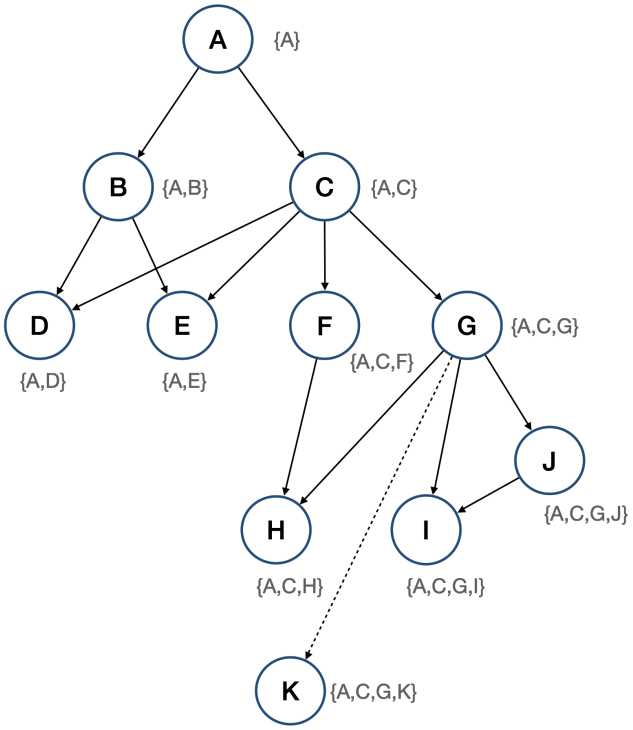


Figure 4.4: CALock labels for a hierarchy with structural modifications.

relabel the entire hierarchy and parallelize the relabelling process. CALock relabels the hierarchy under the same lock acquired to perform the structural modification.

Unlike the other state-of-the-art techniques, which relabel the graph by performing the same post-order traversal as preprocessing, 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 descendants. This is useful for parallelizing structural modifications and reducing the overhead of relabelling the entire hierarchy.

4.2 Multi-Granularity locking and conflict detection

In Section 4.1, we discuss the theory behind CALock labelling and introduce the concept of the lowest guarding common ancestor (LGCA) of vertices in a graph. We present `BFLabel()`, a function to preprocess a hierarchy. Each vertex in a hierarchy is assigned a label during preprocessing. These labels identify a grain for a lock request for a set of lock targets.

As presented in Chapter 2, a standard MGL lock protocol involves the following steps:

1. Optional preprocessing
2. Preparing a lock request
3. Requesting a lock
4. Performing an operation
5. Optional metadata maintenance
6. Releasing the lock

In this chapter, we will study steps 2-6 in detail for CALock. We will introduce the lock request schema for CALock as well as a lock pool, which is used to detect conflicts between requests. We will discuss how we avoid typical *time of check to time of use* (TOCTOU) race conditions between threads and specific optimizations made to the lock pool for fairness.

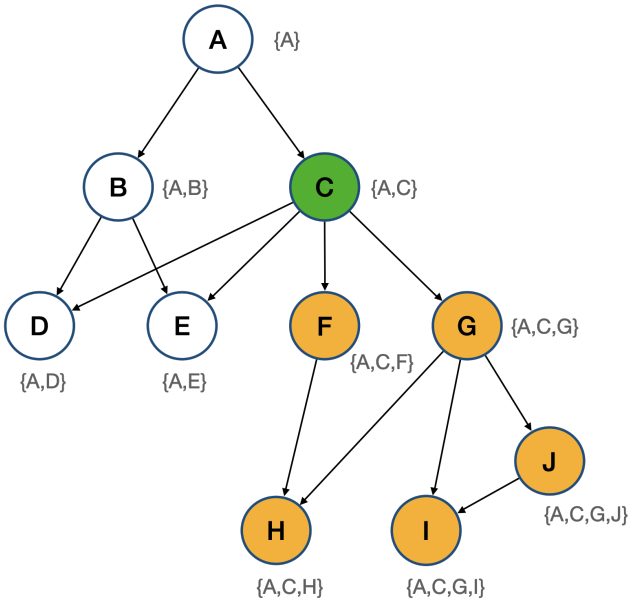
4.2.1 Lock request preparation: vertex labels and LGCA

After a hierarchy is preprocessed, threads start accessing data on vertices. This data access is a read or a write operation. Often, threads wish to access multiple vertices atomically. Once the set of vertices (lock targets) is identified, the thread accessing them needs to ensure isolation to guarantee correctness. To achieve this, it requests a lock on the *Lowest Guarding Common Ancestor* (LGCA) of the set of lock targets. With CALock, this is done by taking the set intersection of the labels of the lock targets. It follows from Theorem 1 that the last element in this set intersection is the LGCA of the set of lock targets and is the root of the grain that contains the target vertices.

For example, in Figure 4.5, to lock target vertices H and J, their LGCA is computed using the intersection of their labels, i.e., $\{A, C, H\} \cap \{A, C, G, J\} = \{A, C\}$ in which is C is the deepest vertex. The thread then proceeds to acquire a lock on C.

A lock request is created with data that identifies a grain without needing to traverse the hierarchy. A lock request contains the following information.

- **Thread ID**: The unique identifier of the thread making the lock request. This helps distinguish lock requests from different threads in the system.



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

Figure 4.5: CALock labels with a lock on C (green) and its grain (yellow).

- **Guard ID**: The unique identifier of the lock guard, which corresponds to the vertex being locked to control access to a grain of the hierarchy.
- **Guard label**: The label of the lock guard vertex.
- **Sequence number**: A unique number assigned to the lock request when it is added to the lock pool. This sequence number helps order lock requests and ensure lock acquisition fairness.
- **Activity indicator**: A boolean flag used by threads to monitor the status of a lock request. When a conflict occurs, threads use this indicator to determine whether they must wait for the current lock request to complete.
- **Lock mode**: Specifies the type of access the thread requests for the resource. This can be read (shared access, allowing concurrent reads) or write (exclusive access, preventing other reads or writes).

Using this data, a lock request is created by a thread. Since multiple lock targets are aggregated into a single grain using the LGCA of the targets, A thread requests a single lock at any given time. Lock requests are then added to a pool, which begins the lock acquisition process.

4.2.2 Requesting a lock: lock pool and lock conflicts

Once a thread has created a lock request object, it calls the `LOCK()` function to add it to a pool, where it is tested for conflicts with other threads. Algorithm 2 shows the procedure for acquiring a lock.

Lock pool

The lock pool in CALock is a central data structure used to manage and coordinate lock requests from multiple threads. It is an array where each entry corresponds to a thread, and an entry in the lock pool at a thread's index indicates an active lock request from that thread.

The first step taken by the scheduler is to assign a sequence number to the lock request (line 6). Sequence numbers help ensure fairness by granting locks in a first-come, first-served order. Then, the activity indicator for a request is set to true, and finally, it is added to the pool at the requesting thread's index.

Guaranteeing an FCFS order to grant locks is essential to ensure fairness. However, a race condition might occur between two concurrent threads 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.

For example, as shown in Figure 4.6, two threads T_1 and T_2 request a write lock on the same vertex v. Obviously, T_1 and T_2 cannot be granted their locks in parallel. One of them has to wait. T_1 is given sequence number 1 and T_2, which arrives after, is given sequence number 2. However, T_1 is pre-empted before it can insert its request into the pool, and T_2 adds its request first, tests for conflict, and finds no conflicting request in the pool. Then, T_1 resumes and inserts its entry into the lock pool. Both T_1 and T_2 will be granted their requests even though they conflict. Vertex v is not isolated any more. This is because T_2 tested for a conflict while T_1 had not added its request to the pool, and T_1 assumes priority since it arrived first and has sequence number 1.

Assigning a sequence number and adding the request to the pool is done atomically to avoid this race. We achieve this in our implementation with a mutex over the lock pool. Since these two operations are short, serializing threads with a mutex does not hinder performance.

T_1	T_2	Lock Pool
`seqNo` $\leftarrow 1$		[NULL, NULL]
	`seqNo` $\leftarrow 2$	[NULL, NULL]
	`addRequestToPool()`	[NULL, $\{v, wl, 2\}$]
	`testConflict()` $\rightarrow$ `false`	[NULL, $\{v, wl, 2\}$]
`addRequestToPool()`		[$\{v, wl, 1\}$, $\{v, wl, 2\}$]
	`lockAcquired` $\leftarrow$ true	[$\{v, wl, 1\}$, $\{v, wl, 2\}$]
`testConflict()` $\rightarrow$ `false`		[$\{v, wl, 1\}$, $\{v, wl, 2\}$]
`lockAcquired` $\leftarrow$ true		[$\{v, wl, 1\}$, $\{v, wl, 2\}$]

Figure 4.6: Threads T_1 and T_2 both acquire a write lock on vertex v due to a race between adding requests to the lock pool and testing for conflicts.

A lock request is guaranteed to be granted in FCFS order once it is added to the pool. After adding its request to the pool, a thread checks for conflicts with other requests.

Lock conflicts

In MGL on graphs, conflict detection involves checking two conditions to ensure correctness. These conditions arise from locks at different granularities (sub-graphs). The compatibility matrix for CALock is shown in Table 4.2

- **Grain overlap**: A grain overlap conflict occurs between two lock requests trying to lock overlapping grains. An overlap exists when the grains of the two lock requests have at least one common vertex. For example, if a thread is trying to lock a vertex, and another thread is holding a lock on a sub-graph that contains the vertex, the grains overlap. In this case, the finer-grained lock must be exclusive, meaning no conflicting lock can exist at a higher level.

  In CALock, Grain overlaps are checked using the requests' guard ID and guard label. 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 so 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.

Mode conflict is checked by comparing the lock modes for requests in the lock pool. A mode conflict exists if one request is a write lock and the other is a read lock.

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

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

Table 4.2: Lock compatibilities between read (rl) and write (wl) locks requested by threads $i \neq j$ on vertices x and y.

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

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}`
0	1	2	3	4	5	6

Figure 4.7: Lock pool containing active lock requests.

A snapshot of a lock pool is shown in Figure 4.7. The corresponding lock grains are shown in Figure 4.8. 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, T_2 and T_7 iterate over the pool from left to right and check for conflicts. T_1 finds a conflicting request at index 1 but has priority in the conflict since its sequence number is lower. Beyond index 1, it has no conflict in the pool and proceeds to acquire a read lock on G.

T_2 finds a conflicting request at index 0 and has a higher sequence number than T_1. So, T_2 waits for T_1 to release its lock. T_1 and T_2 conflict because they have overlapping grains. T_2 requests a lock on C, whose grain overlaps with the grain locked by T_1, rooted at G. This is confirmed by checking if C is present in the label of G, i.e., $C \in \{A, C, G\}$, which is true.

T_7 does not conflict with any request and acquires a lock on B.

While checking for conflicts, a thread blocks at each conflict it encounters if its sequence number is higher than the conflicting request. A higher sequence number

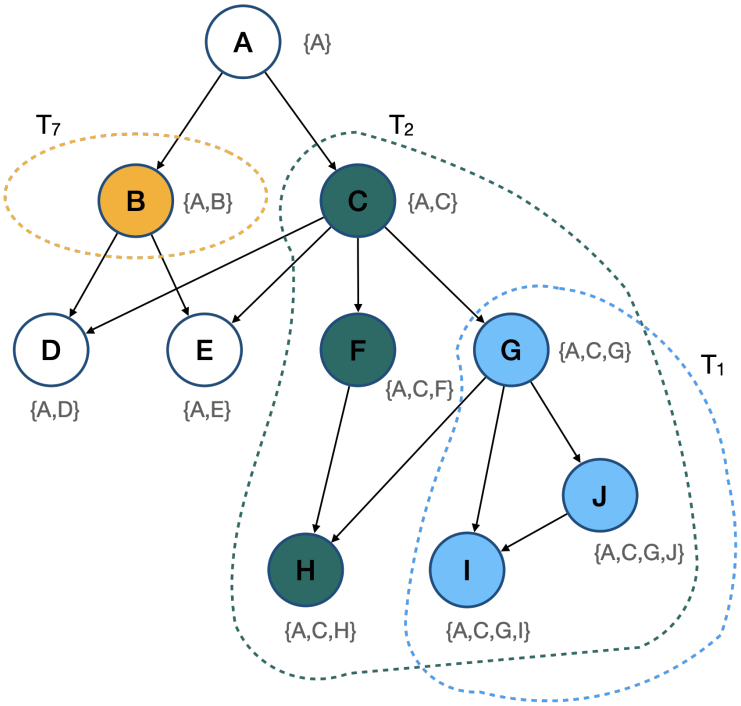


Figure 4.8: Lock grains on the hierarchy for the locks requested by the threads in the lock pool (Figure 4.7).

indicates that the thread arrived later and should wait for the conflicting thread to release the lock. Once a thread has iterated over the lock pool until the end, it is guaranteed to either have no conflicts with any request or have priority over all conflicts because its sequence number would be the lowest.

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( )
```

4.3 Metadata maintenance: structural modifications and relabelling

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

The same CALock algorithm, shown in Algorithm 2, can be utilized for dynamic graphs that change at runtime. To perform a structural modification, a thread acquires a write lock on the LGCA of the affected vertices or the LGA if only one vertex is involved in a structural modification. For example, when adding an edge between vertices u and v, a thread acquires a write lock on the LGCA of u and v.

A structural modification also involves a relabelling step for the affected grain. However, unlike DomLock, MID and FlexiGran, in which the relabelling happens by acquiring a write lock on the hierarchy, relabelling in CALock is done under the same lock acquired to perform the structural modification. Here, we explain the locking and relabelling mechanism for dynamic graphs.

4.3.1 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, vertex addition does not require locking or relabelling of any kind. The vertex gets a default label that contains its ID.

Deleting a vertex with no edges does not require synchronization either because such a vertex is not reachable from the root of the graph and does not have children that might be affected.

To delete a vertex that is reachable from the root, i.e., connected to the graph, a write lock is acquired on the LGA of the vertex to be deleted.

The LGA can be computed by taking the second-last element in the label of the vertex to be deleted. The LGA guards the grain containing this vertex. Once the grain is locked, the vertex is disconnected from the graph by deleting all its edges and then deleted. An example where vertex F is deleted is shown in Figure 4.9. To

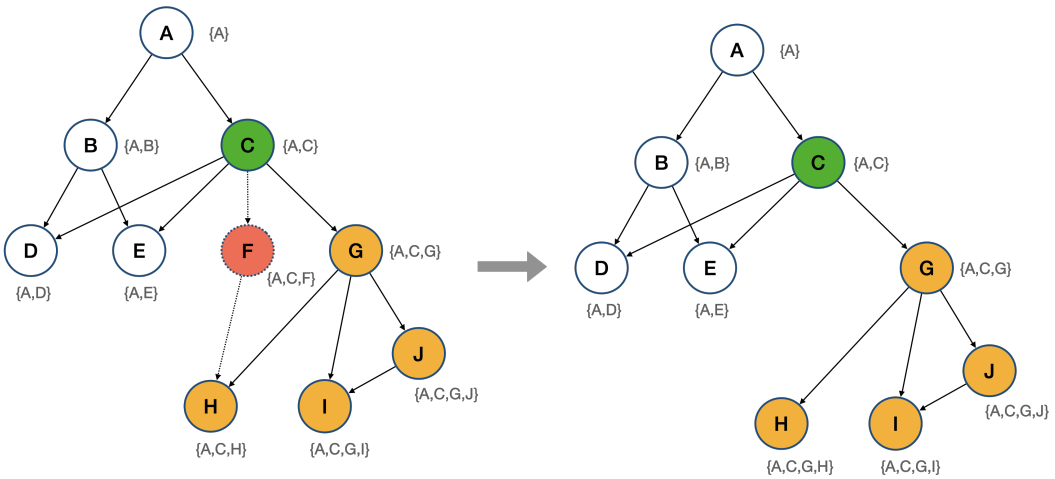


Figure 4.9: In CALock, Deleting vertex F requires a lock on C.

delete F, a lock is acquired on C, the second last vertex in the label of C, and is the LGA of F.

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 Figure 4.9, F has one child, H. So, the label of H is recomputed. After F is deleted, H has only one parent, G. So, the label of H is recomputed as $\{A, C, G, H\}$.

4.3.2 Edge addition and deletion

Adding and deleting an edge changes a graph's topology and the paths to the vertices. Both operations are performed under a write (exclusive) lock.

To add an edge between a source vertex u and a target vertex v, a write lock is acquired on the LGCA of u and v. This LGCA is computed via a set intersection of the labels of u and v, i.e. $L_u \cap L_v$. The last vertex in this intersection is the LGCA of u and v, which needs to be write-locked to isolate the grain containing both u and v.

Once a write lock is acquired, the operation can be performed. Adding or deleting an edge between two vertices changes the ancestors of the target vertex, so relabelling is initiated at the target vertex of the affected edge using `BFLabel()` function. An example where an edge is deleted between G and H is shown in Figure 4.10. Before deleting, a lock is acquired on C, the LGCA of G and H. After the edge is deleted, the

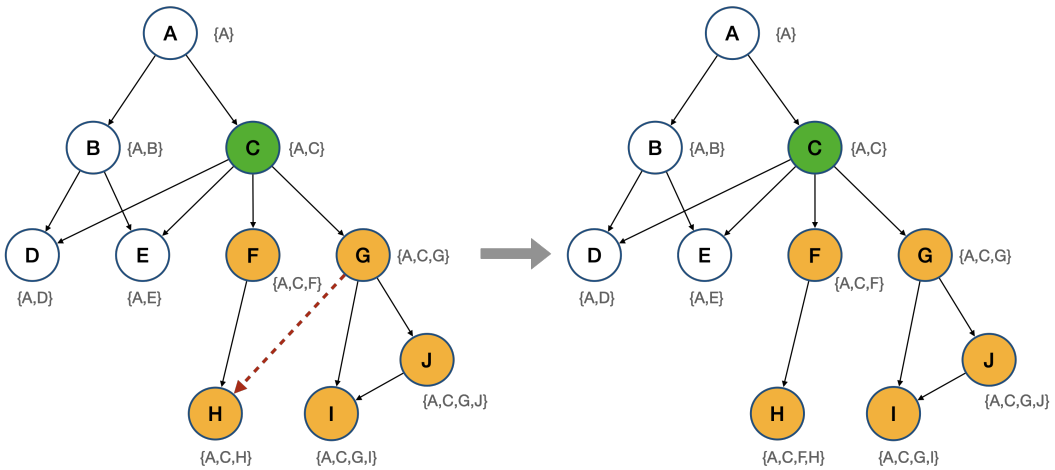


Figure 4.10: Deleting the edge between G and H requires a lock on C and relabelling H.

target of the deleted edge, H, is relabelled. H has only one parent, F. So, the label of H is recomputed as $\{A, C, F, H\}$.

When a vertex v 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.

4.4 Properties of CALock

A transaction uses the correctness guarantees of a locking protocol to assume certain guarantees regarding other transactions in the system. If all transactions adhere to the protocol, the system will remain correct.

This section shows 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. For instance, CALock requires the assignment of lock request sequence numbers, setting the activity indicator, and adding to the lock pool to happen atomically. In our implementation, we use a mutex to achieve this.

4.4.1 Correctness guarantees

Safety

Property A thread holding a write lock on a guard has exclusive access to the corresponding grain. Assuming 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, the lock is granted iff 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, 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 number is assigned atomically before adding the corresponding request to the pool.

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.

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 every thread eventually releases 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 them into a single request, 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.

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 with a lower sequence number. Other threads can block a thread 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 the lock it requested would be granted. This ensures that no thread starves and CALock is fair.

4.4.2 Complexity analysis of CALock

Beyond the correctness guarantees, the time complexity of the locking algorithm is crucial for its practical use in judging its suitability for various use cases. This section discusses the complexity of CALock labelling, relabelling and conflict detection and compares it with the other MGL techniques discussed in Chapter 3.

Labelling and relabelling

The labelling 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 unique 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

$$d_{avg} = \frac{2 \times e}{v} \tag{3}$$

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 paths from the root 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 labelling 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 $O(2v)$.

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 ($q << v$) 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 graph vertices, 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. A set membership test is performed for each position in the lock pool. An efficient set implementation can test the membership in $O(1)$. The complexity of conflict detection is:

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

4.4.3 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 4.3 and 4.4, 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	$\theta(v + \frac{v}{d_{avg}})$	$\theta(\frac{q}{d_{avg}})$	$\theta(n)$

Table 4.3: Average case complexities of MGL techniques (n is the number of threads).

	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	$O(2v)$	$O(v)$	$O(n)$

Table 4.4: 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 a 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. CALock labels can be computed faster than the integer intervals used by DomLock, MID and FlexiGran for very complex graphs. This results in a lower complexity for CALock in worst-case scenarios.

4.5 Summary

This chapter we presented CALock, a new labelling and multi-granularity locking protocol for hierarchical data. We discuss the theoretical underpinnings of CALock labelling and locking mechanisms using definitions of paths and common ancestors. We present a recursive labelling function used to label a hierarchy's vertices. Further, we discuss how the locking mechanism uses these labels to find an appropriate lock guard for a set of lock targets. CALock lock requests, which contain information about the lock guard, grain and lock mode, are added to a lock pool, which is used to detect conflicts.

We then discuss the formal properties of CALock. We show that CALock is safe, live and fair. Finally, we present the time complexities of different parts of the CALock lock mechanism and compare them with state-of-the-art MGL techniques. We show that CALock has a lower complexity than other MGL techniques in worst-case scenarios.

5 Implementation and evaluation

To analyse the performance of a locking mechanism, it is essential to implement and evaluate it with a benchmark which simulates a real-world workload. In this chapter, we present an implementation of CALock and assess its performance using the STMBench7 benchmark [GKV07]. STMBench7 is based on a CAD/CAM application and models a set of design objects into a hierarchy. We first provide an overview of the STMBench7 data model and then describe the implementation of CALock, focussing on the labelling mechanism, lock pool and conflict detection protocol. We compare the performance of CALock with coarse-grain locks, medium-grain locks, Intention Locks, DomLock, Multi-Interval DomLock, and FlexiGran.

5.1 Hierarchy schema for evaluation

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 5.1, 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 via a designated vertex called the *root part*.

5.2 Implementation of CALock

The implementation of CALock written in C++, chosen for its ability to deliver high performance and fine-grained control over data structures. Several optimizations

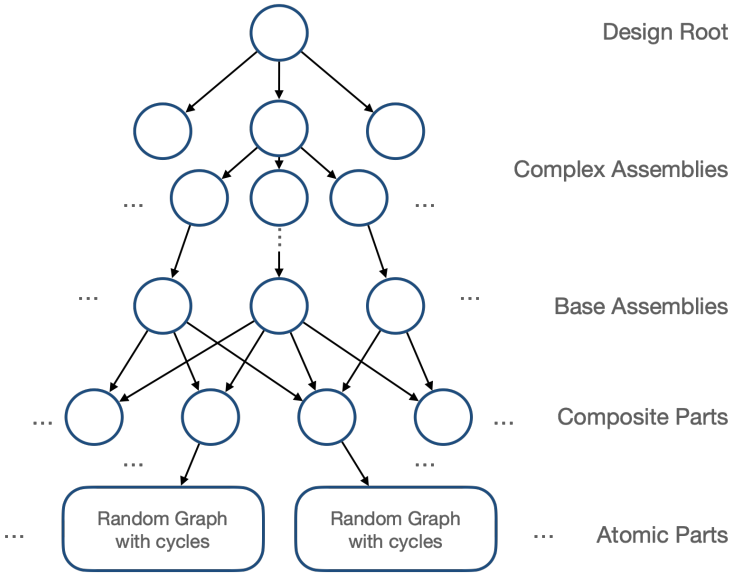


Figure 5.1: Structure of a module in STMBench.

have been integrated to ensure the protocol operates efficiently. 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 structural complexity.

5.2.1 CALock labelling

The CALock labelling is implemented as the `CALockLabelling` class, which provides a method `run()` to label the graph. Unlike the protocol description in Chapter 4, which uses characters as vertex IDs, our implementation uses integers. This allows for compact storage representation and faster membership tests. The hierarchy in STMBench consists of 4 different types of vertices. The implementation distinguishes each by assigning them an integer suffix based on their type. CALock IDs for vertices are constructed by appending this suffix to the STMBench vertex ID. The suffixes are 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.

Vertex labels

The label on each vertex is an ordered set containing the CALock IDs of the guarding ancestors of that vertex. In the implementation, to facilitate a 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. Listing 5.1 shows the `Vertex` class with the CALock label fields.

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

Since membership tests are frequently used in CALock when locking and conflict detection, the implementation uses an auxiliary `set` for membership tests. It bypasses the vector's linear search complexity. Together, a vector and a set provide constant-order time complexity for membership tests and label updates. The set and the vector are synchronized outside the critical path to contain the same CALock IDs.

```
class Vertex {
    ...
    int id;
    bool hasLabel;
    vector<int> pathLabel{};
    set<int> guardingAncestors{};
    bool isDeleted;
    ...
};
```

Listing 5.1: Vertex class with CALock label fields.

To find the LGA(the lowest guarding ancestor) of a single vertex, the last element of the `pathLabel` vector is returned (Listing 5.2). When finding the LGCA(the 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. This is shown in Listing 5.3.

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

```
    } else {
        return *pathLabel.rbegin();
    }
}
```

Listing 5.2: Finding LGA using path label.

```

int getLGCA(const vector<AtomicPart *> &aparts) const {
    list<int> guardingCommonAncestors{};
    auto it = aparts[0]->pathLabel.rbegin();
    auto end = aparts[0]->pathLabel.rend();
    for (auto i: aparts[0]->pathLabel) {
        for (auto a: aparts) {
            if (a->guardingAncestors.contains(i)) {
                guardingCommonAncestors.push_back(i);
            }
        }
    }
    return guardingCommonAncestors.rbegin();
}
```

Listing 5.3: Finding the LGCA of a set of atomic parts.

Labelling algorithm

The labelling algorithm is a breadth-first traversal over the hierarchy. It is implemented by keeping separate queues for each vertex type: Complex Assembly, Base Assembly, Composite Part, and Atomic Part. The implementation is shown in Listing 5.4. Labelling starts from the root of the hierarchy. The root of the hierarchy is a Complex Assembly. First, we add a suffix to the root's ID to get the CALock ID. The CALock ID is then set as the root's label. The root is then added to the queue of Complex Assemblies to initiate the labelling process.

All complex assemblies are labelled first using the function `traverse(cassmQ.front(), &cassmQ, &bassmQ)`. At the lowest level, complex assemblies contain base assemblies. While traversing, the algorithm labels the complex assembly and adds its children to the queue of complex assemblies (`cassmQ`) or base assemblies (`bassmQ`) based on their type. Once all complex assemblies are labelled, the algorithm proceeds to label base assemblies using function `traverse(bassmQ.front(), &cpartQ)`. This process continues until all vertices are labelled.

```
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;
    //Append Complex Assembly suffix (1) to the root ID to get
        CALock ID.
    rootLabel.push_back((root->getId() * 10) + 1);
    root->setPathLabel(rootLabel);
    cassmQ.push(root);
    // Label all complex assemblies.
    while (!cassmQ.empty()) {
        // Traverse a complex assembly and compute its label. Add
            its children to cassmQ or bassmQ
        traverse(cassmQ.front(), &cassmQ, &bassmQ);
        cassmQ.pop();
    }
    // Label all base assemblies.
    while (!bassmQ.empty()) {
        // Traverse a base assembly and compute its label. Add its
            children to cpartQ.
        traverse(bassmQ.front(), &cpartQ);
        bassmQ.pop();
    }
    // Label all composite parts.
    while (!cpartQ.empty()) {
        // Traverse a composite part and compute its label. Add its
             children to apartQ.
        traverse(cpartQ.front(), &apartQ);
        cpartQ.pop();
    }
}
```

Listing 5.4: BFS traversal for CALock labelling.

The `traverse()` method is overloaded for each vertex type. After computing the label of a vertex, the method adds the children of the vertex into their appropriate queues based on their type. Listing 5.5 shows the implementation of the `traverse()` method for a vertex of Complex Assembly type. The method traverses the children of a complex assembly and labels them. Each assembly gets a CALock ID by appending its type suffix to its STMBench ID. For instance, a base assembly is assigned an ID with suffix 2 (`int labelIdentifier = (assm->getId() * 10) + 2`).

```
void traverse(  ComplexAssembly *cassm,
                queue<ComplexAssembly *> *cassmQ,
                queue<BaseAssembly *> *bassmQ) const {
    list<int> currLabel = cassm->pathLabel;
```

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

Listing 5.5: Labelling a complex assembly.

The Atomic Part graph in STMBench is dense and can 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 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 post-order. The implementation of the `traverse` method for an Atomic Part is shown in Listing 5.6. Labelling atomic parts terminates once their labels reach a fix point (lines 2-7).

An atomic part stores a set of incoming and outgoing edges. The parents of an atomic part are located using the incoming edges (`apart->getFromConnections()`). The label of an atomic part is computed by successively removing guarding ancestors that are not present in the guarding ancestors of its parents. Once all the parents are tested, the atomic part is labelled with the guarding ancestors of its parents along with its own CALock ID. If the label of an atomic part changes, the algorithm recursively labels all children of the atomic part.

```
1   void traverse(AtomicPart *apart, Set<AtomicPart *> &visitedPartSet,
        list<int> currLabel) {
2       if (visitedPartSet.contains(apart)) {
```

```
3           return;
4       } else {
5           visitedPartSet.add(apart);
6           Set<Connection *> *fromConns = apart->getFromConnections();
7           SetIterator<Connection *> fiter = fromConns->getIter();
8           // Find ancestors that are in the atomic part's label but
                are not guarding ancestors of its parents.
9           set<int> removals;
10          for (int a: currLabel) {
11              bool allContain = true;
12              while (fiter.has_next()) {
13                  if (!fiter.next()->getSource()->guardingAncestors.
                        contains(a)) {
14                      allContain = false;
15                      break;
16                  }
17              }
18              if (!allContain) {
19                  removals.insert(a);
20              }
21              fiter = fromConns->getIter();
22          }
23          // Remove all ancestors which are not in the guarding
                ancestors of the parents.
24          currLabel.remove_if([removals](int l) { return removals.
                contains(l); });
25
26          if (currLabel != apart->pathLabel) {
27              // If the label has changed, set the new label and
                    label the children.
28              apart->setPathLabel(currLabel);
29              Set<Connection *> *toConns = apart->getToConnections();
30              SetIterator<Connection *> iter = toConns->getIter();
31              while (iter.has_next()) {
32                  Connection *conn = iter.next();
33                  currLabel.push_back((conn->getDestination()->getId
                        () * 10) + 4);
34                  traverse(conn->getDestination(), visitedPartSet,
                        currLabel);
35                  currLabel.pop_back();
36              }
37          }
38          visitedPartSet.remove(apart);
39      }
40  }
```

Listing 5.6: Labelling an atomic part.

5.2.2 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` inserted into the lock pool. This `lockObject` contains the following fields:

- `int Id`: Guard ID.
- `set<int> *guardingAncestors`: 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 5.7. The `atomic_flag` type is an atomic boolean, provided by C++ and is guaranteed to be lock-free. Operations on this type are guaranteed to occur in the same order as they are issued and are not reordered by compiler optimizations. This prevents nondeterministic thread blocks due to release/acquire reordering on some CPU architectures. Multiple atomic flags within the same program are totally ordered, preventing blocked threads from being notified before the lock is released.

```
1  class lockObject {
2  public:
3      set<int> *guardingAncestors;
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         guardingAncestors = ancestors;
12         mode = m;
13         Oseq = -1;
14         accessController.test_and_set();
15     }
16 };
```

Listing 5.7: `lockObject` class.

The lock pool is an array that contains `lockObjects`. The size is fixed and bounded to the number of concurrent hardware threads the system provides. Listing 5.8 shows the class definition of the CALock 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 5.8: `lockPool` 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 checks for conflicts and waits for resolution. The method is shown in Listing 5.9.

```
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 || l->mode == 1) &&
                // Grain Overlap.
                (reqObj->Id == l->Id ||
                reqObj->guardingAncestors->contains(l->Id) ||
                l->guardingAncestors->contains(reqObj->Id)) &&
```

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

Listing 5.9: Acquiring a lock on `reqObj`.

A lock request is first assigned its 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 (Listing 5.9 - lines 2-5).

Once a lock request is inserted into the pool, the method checks for conflicts with other lock requests by iterating over the pool from left to right. This check can be performed in parallel for all lock requests in the pool. The lock is granted if no conflict is detected, and the thread can proceed.

If a conflict is detected, the thread waits until the conflicting request releases its lock. This is done by waiting on the `accessController` of a conflicting lock request flag to become `false` (Listing 5.9 - line 21). Even when present in the lock pool, a request with activity indicator `false` is considered to have released its lock.

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

Listing 5.10: 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 5.10. Once waiting threads are notified, they resume execution and check for further conflicts in the lock pool.

After checking for conflicts with all lock requests in the pool and waiting on conflicting requests, a thread reaches the end of the lock pool (Listing 5.9 - line 6). At this point, the lock is always granted, as there are two possible scenarios:

a. The lock request does not conflict with any other lock request in the pool (Listing 5.9 - 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 5.9 - line 19).

5.2.3 Overall execution of an operation in CALock

The overall execution of an operation in CALock is shown in Listing 5.11. 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.

This operation first computes the set intersection of the labels of the base assembly and the composite part. This set intersection finds the LGCA of the base assembly and composite part.(Listing 5.11 - lines 8-19).

A lock request is created with the LGCA and the guarding ancestors in write mode (Listing 5.11 - line 21). The lock is acquired using the `acquireLock()` method of the lock pool. The thread waits for resolution as long as the lock is not granted (Listing 5.11 - line 24). Once granted, the operation is performed (Listing 5.11 - 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 parts for relabelling (Listing 5.11 - lines 28-30).

After relabelling is complete, the lock is released via the `releaseLock()` method of the lock pool (Listing 5.11 - 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->guardingAncestors.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
            ->guardingAncestors, 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 CALockRelabelling(dataHolder, tid);
29          r->cpartQ.push(cpart);
30          r->run();
31          // Release the lock.
32          caPool.releaseLock(l, tid);
33      }
34
35  }
```

Listing 5.11: Overall execution of an operation in CALock.

5.3 Performance evaluation of CALock

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

5.3.1 Experimental setup and code-base

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. GCC 12.1 and Cmake 3.22 are used to build

[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 benchmark. The compilation is done at the C++ 23 standard to enable atomic wait primitives. The optimization level is set to O3.

The implementation of STMBench7 and all the lock protocols is open source and available in a repository. The implementation contains scripts to configure the parameters, execute the benchmark and create performance charts. Optional raw data collection facilitates an in-depth understanding of the benchmark results.

5.3.2 Benchmark suite: STMBench7

STMBench7 provides coarse-grain and medium-grain lock implementations. These are representatives of fixed granularity locking techniques. The coarse-grain lock is a read-write lock on the entire graph. Figure 5.2 shows the medium-grain lock grains for a module. Complex assemblies are divided into levels, each 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.

Read-only operations

Q1 Reading an atomic part

Q2 Reading a set of atomic parts

OP1 Reading a set of complex assemblies

OP2 Reading a set of base assemblies

Write-only operations

OP3 Writing data to an atomic part

OP4 Writing data to a set of atomic parts

Structural modifications

SM1 Deleting a composite part

SM2 Adding an edge between a base assembly and a composite part

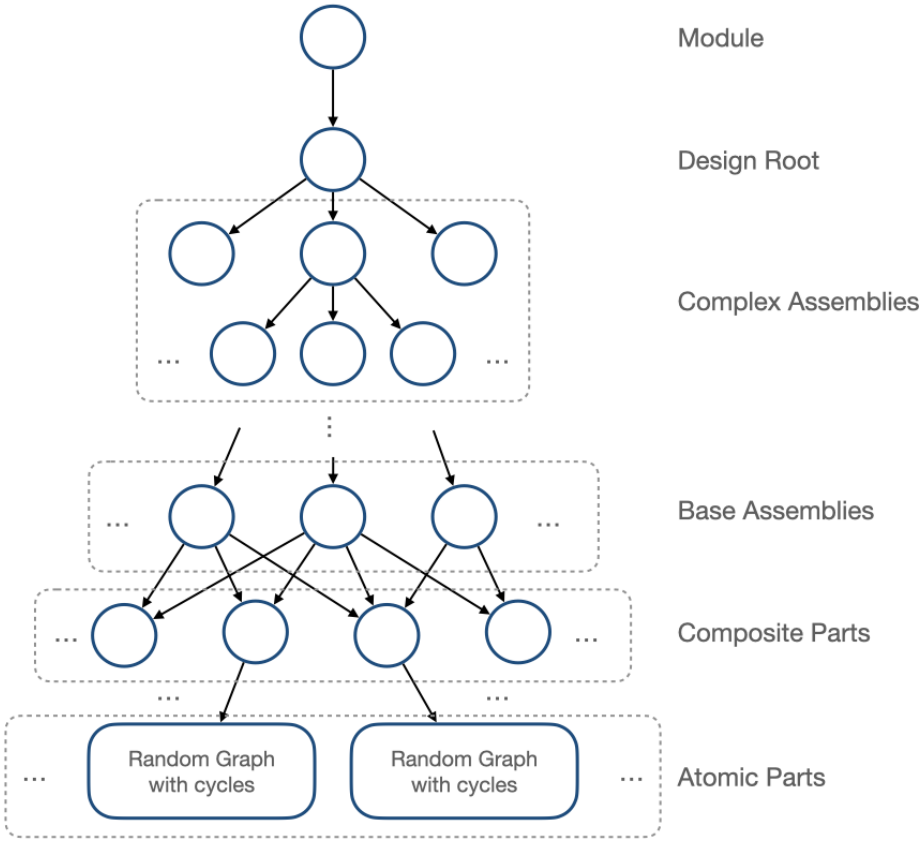


Figure 5.2: Structure of a module in STMBench with medium lock boundaries

5.3.3 Synopsis

To evaluate CALock, we compare it with coarse-grain locks, medium-grain locks, Intention locks, DomLock, MID and FlexiGran. To this end, we formulate research questions that scrutinize different aspects of the lock mechanism and address the claims made throughout this thesis. The questions we address are the following:

§5.3.4 How quickly does CALock grant a lock compared to other approaches for different operation types (Q1-SM2)?

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

§5.3.6 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?

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

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

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

5.3.4 Per operation response time

Question: *How quickly does CALock grant a lock compared to other approaches for different operation types (Q1-SM2)?*

Figure 5.3 shows the time to completion for different operations in STMBench7. Coarse-grain and medium-grain locks are the fastest at granting lock requests. However, they do not scale well (see Section 5.3.8, Section 5.3.8).

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. This cost is multiplied for requests with multiple target vertices (Q2, OP4), which leads to a higher response time.

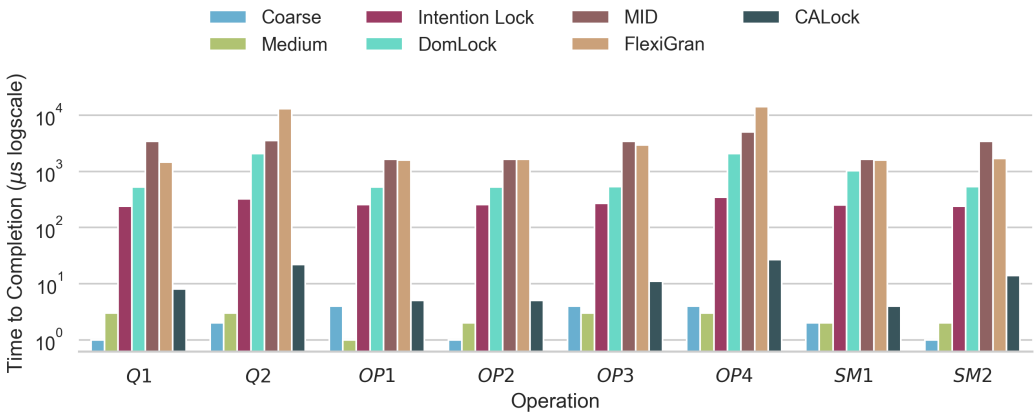


Figure 5.3: Time to completion for different operations in STMBench (lower is better).

In MGL techniques like DomLock, MID and FlexiGran, the time to completion is at least 300 times higher than coarse-grain, medium-grain and intention locks. Once a thread identifies the lock targets and finds their subsuming interval, a depth-first traversal is required to find the lock guard with the same interval. This traversal is especially expensive for vertices deeper in the hierarchy.

Between existing 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 involves testing overlap between a pair of intervals. FlexiGran is slower than MID because it requires testing overlap between intervals and level numbers. Conflict detection in FlexiGran MGL and FlexiGran fine-grained locks involves testing reachability between the guard of the MGL lock and the fine-grained lock. This is expensive and leads to a longer overall completion time.

CALock is at least 100 times faster to grant a lock than existing MGL techniques. This is because CALock eliminates needing 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 and Q2) are the fastest since they can progress 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 sub-graphs and is at least 100 times faster.

5.3.5 Metadata management: preprocessing and relabelling

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

MGL techniques rely on metadata to identify lock grains. DomLock and MID use integer intervals for subsumption testing, and 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 quickly negate any performance benefits. In this section, we examine the time required to compute the metadata initially (preprocessing) and the time needed to update it after a structural modification (relabelling). The benchmark is run on three different sizes of the STMBench7 graph. Table 5.1 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

Table 5.1: Sizes of the STMBench7 hierarchies.

Preprocessing

We measure the preprocessing 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 5.4. 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 a pair of intervals for each vertex. One is 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 takes the longest to preprocess a hierarchy since a recursive breadth-first traversal with a fix-point dependent on the number of paths to a vertex from the root defines the labels. CALock remains 10 times slower than MID and 20 times slower than DomLock for preprocessing.

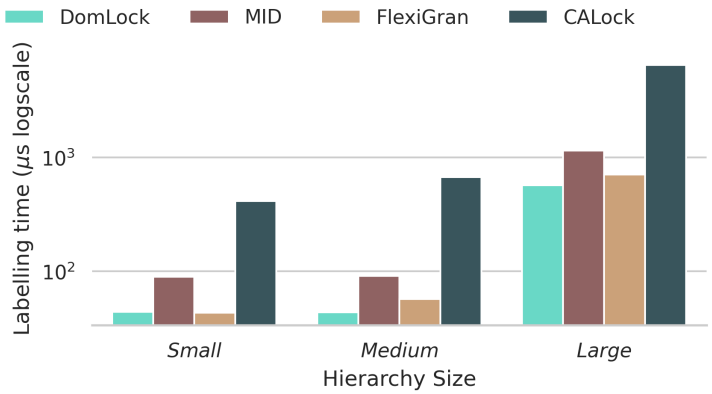


Figure 5.4: Time to compute initial labels (lower is better).

Relabelling

This benchmark studies 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 is performed under a global write lock and cannot be parallelized. Figure 5.5 shows the time DomLock, MID, FlexiGran and CALock take to relabel the graph after a structural modification.

CALock relabels only the sub-graphs 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 at least 100 times faster at relabelling than DomLock, MID and FlexiGran.

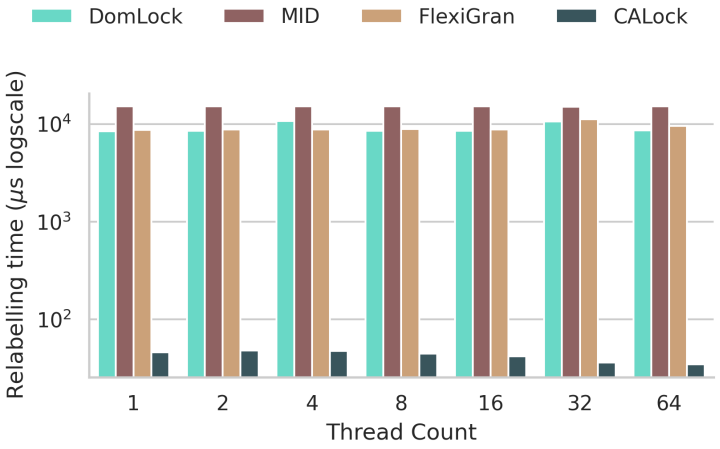


Figure 5.5: Time spent relabelling the graph per structural modification (lower is better).

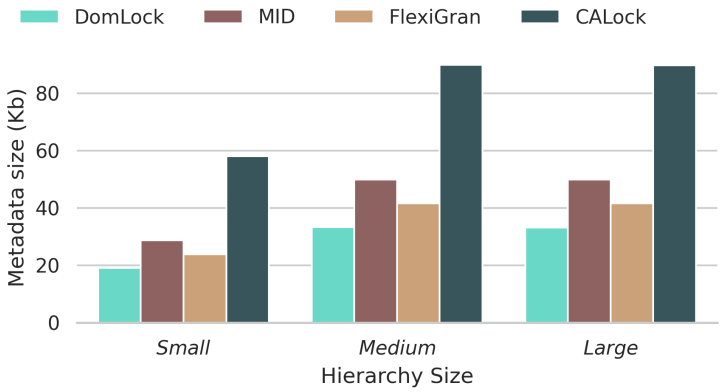


Figure 5.6: Size of the metadata used for labelling(lower is better).

5.3.6 Metadata size in memory

Question: *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?*

Label-based MGL techniques utilize metadata to identify the lock grains. DomLock utilizes compact integer ranges as labels. MID uses a pair of integer ranges to represent the intervals of the vertices. FlexiGran uses the DomLock intervals and 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 5.6, CALock's metadata consumes approximately 1.5 times more memory than DomLock, MID and FlexiGran. In DomLock, MID and FlexiGran, regardless of the topology of the graph, the label at each vertex consists of integers. This has a low memory requirement. However, 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 eliminating false subsumptions.

5.3.7 Lock granularity and false subsumptions

Question: *Does CALock eliminate false subsumptions and reduce grain size compared to DomLock, MID and FlexiGran?*

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.

We measure the number of targets implicitly locked for a guard vertex in the STMBench7 graph to compare the grain sizes for different locking protocols. Figure 5.7 compares the average granularity per vertex type for locks acquired using DomLock, MID, FlexiGran and CALock.

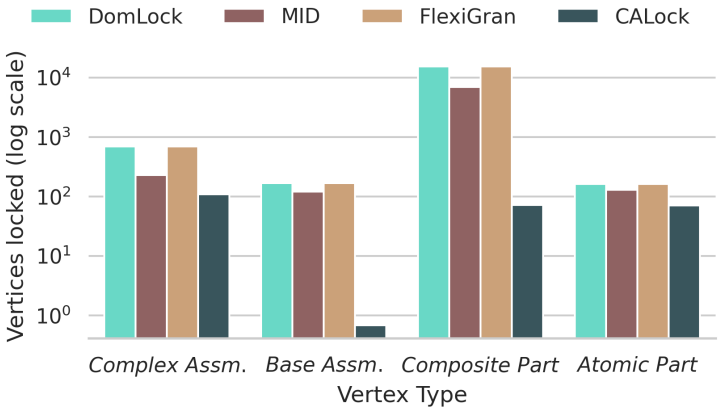


Figure 5.7: Vertices locked per vertex type (lower is better).

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 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 and composite parts due to false subsumptions. CALock is significantly better at this level, with grain sizes 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 complex assembly because a composite part is usually contained in multiple base assemblies. Combined with false subsumptions, this 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 than other locking techniques, as is evident in Figure 5.7.

5.3.8 Overall locking performance

In the previous sections, we analysed specific parameters individually: per-operation latency (Section 5.3.4), labelling and relabelling time (Section 5.3.5), metadata size (Section 5.3.6), and false subsumption ratio (Section 5.3.7). In this section, we combine these parameters to study their collective impact on overall performance, providing a more comprehensive evaluation of runtime evaluation.

In this set of benchmarks, we study data updates and structural modifications. Figures Figure 5.8 and Figure 5.9 show the throughput of workloads with data access only and workloads with structural modifications, 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.

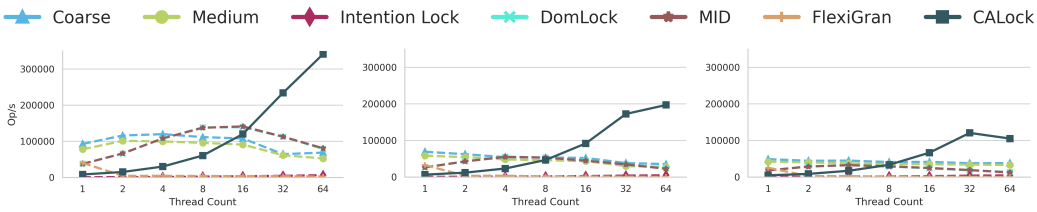
Data access

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

Throughput figures Figure 5.8a, Figure 5.8b and Figure 5.8c 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.

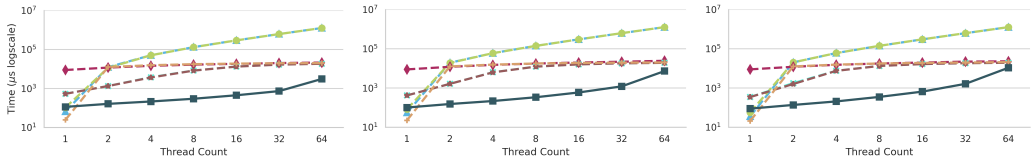
The graph in STMBench is irregular and has multiple paths leading to a vertex. DomLock, MID and FlexiGran intervals often lead to false subsumptions, causing threads to block (see Section 5.3.7). Intention locks exhibit poor performance due to the high number of paths leading to target vertices from the root of the STMBench hierarchy (see Figure 5.3).

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.



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

Figure 5.8a, Figure 5.8b, Figure 5.8c: Throughput (higher is better).



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

Figure 5.8d, Figure 5.8e, Figure 5.8f: Average time to grant lock request (lower is better).

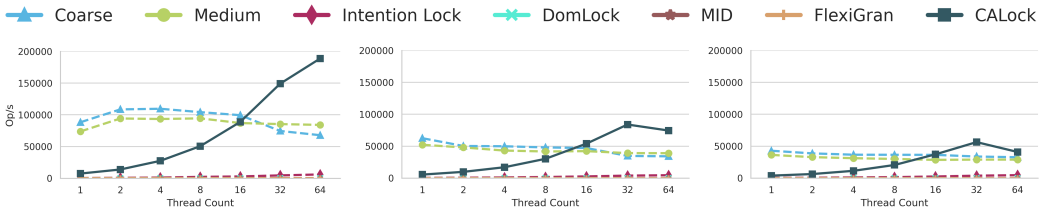
Figure 5.8: Performance with different workload types on static graphs (R: reads, W: writes).

Response time figures Figure 5.8d, Figure 5.8e and Figure 5.8f 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 increased 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.

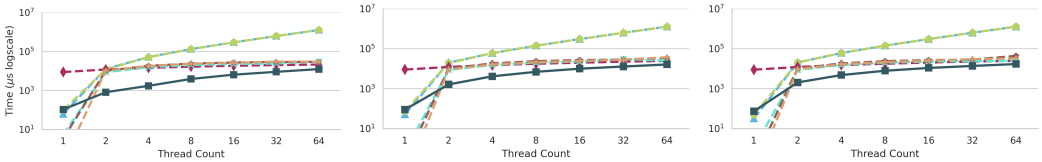
In interval-based MGL techniques like DomLock, MID and FlexiGran, once a lock request identifies an interval it wishes to lock, the thread traverses the hierarchy to find the corresponding lock guard with the requested interval (resp. interval pair for MID). This traversal is especially expensive when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

For all locking protocols, the overall wait time increases with the number of concurrent threads because of increased 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.



(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%

Figure 5.9a, Figure 5.9b, Figure 5.9c : 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%

Figure 5.9d, Figure 5.9e, Figure 5.9f: Average time to grant lock request (lower is better).

Figure 5.9: Performance with different workload types on dynamic graphs (R: reads, W: writes; SM: structural modifications).

Structural modifications

Question: *What is the performance of CALock compared to other lock strategies for workloads that also include structural modifications (Q1-SM2)?*

A structural modification changes the topology of the graph. This triggers relabelling for MGL locking techniques like DomLock, MID, FlexiGran and CALock.

Throughput figures Figure 5.9a, Figure 5.9b, and Figure 5.9c 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 5.8b, 40% of the operations are writes. Of these writes, 1% 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 5.9c), structural modifications can be as high as 0.9%. While in read-heavy workloads (Throughput Figure 5.9a), 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 than other algorithms because of the lack of parallelism due to additional relabelling required when a structural modification occurs. Note that the curves in Throughput figures Figure 5.9a,

Figure 5.9b and Figure 5.9c for DomLock, MID and FlexiGran are relatively flat, indicating a lack of scalability. CALock can parallelize structural modifications and is 2 times faster than coarse and medium-grained locks and about 8 times faster than DomLock, MID and FlexiGran.

Response time for Intention locks remains the same regardless of the proportion of structural modifications. However, as shown in Response time figures Figure 5.9d, Figure 5.9e and Figure 5.9f, 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 compared to static graphs, but CALock remains faster than all other lock techniques for even the most contended workload.

5.4 Summary of experimental results

With the results from the experiments using STMBench7 and micro-benchmarks, we can derive the following conclusions.

- Intention locks are the most expensive locking technique for any STMBench workload and are unsuitable for irregular graphs.
- CALock is faster than other locking techniques for all studied workloads with more than 8 concurrent threads.
- In read-heavy workloads without structural modifications, CALock is at least 3 times faster than 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 8 times 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 is 4 times faster than DomLock, MID and FlexiGran

- CALock is an appropriate locking technique if the graph is irregular (for which Intention locks are unsuitable) and the workload consists of structural modifications (for which DomLock, MID and FlexiGran are unsuitable).

6 Conclusion

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 various applications, from social networks to recommendation systems.

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

Intention locking has proven its mettle as an effective concurrency control mechanism when managing tree-like structures but has severe limitations when dealing with irregular semi-structured data like hierarchies. Modern *Multi-Granularity Locking* (MGL) techniques are not designed to handle the dynamic nature of connected data.

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 that 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 guarding common ancestor* (LGCA) of the vertices as the lock guard. 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. We have shown that CALock can significantly reduce contention and enhance throughput in multithreaded environments through rigorous experimentation and analysis.

This work's key contributions include developing a theoretical framework for CALock, implementing a prototype system, and conducting 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 this work will inspire further research and innovation in this area.

6.1 Contributions

6.1.1 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 for 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 determine 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*: To minimize grain size, 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. *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.

6.1.2 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 locking protocol leverages a *lock object*, a set of fields like guard ID, guard label, etc., to distinguish lock requests and check for conflicts between them. In addition, a lock pool guarantees safety and fairness when 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*: CALock protocol uses the labels of guards in the lock pool to test for grain overlaps. If the ID of a guard is present in the label of another lock guard, then there is a grain overlap. This reduces the time to detect conflicts between lock requests by eliminating traversals or sub-graph matching.

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

6.2 Summary of findings

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

- **Lock throughput improvement:** CALock demonstrates a significant improvement in throughput compared to contemporary MGL techniques. CALock is 4.5 times faster than DomLock and MID, the next best MGL techniques, for workloads with only data updates. CALock is objectively better in workloads containing structural modifications since DomLock, MID, and 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 and FlexiGran do not scale at all. The performance of 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 and reduces spurious blocks. Second, the lock protocol does not require traversals to acquire a lock and test for conflicts. This results in locks that are granted fast and with no 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 to block all other operations, unlike DomLock, MID and FlexiGran. This allows for parallel relabelling and ensures the system remains responsive even when the hierarchy within a locked grain changes.

The evaluation highlights CALock's advantages over existing MGL techniques regarding 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 data updates and structural changes.

6.3 Future work

In this work, we have focussed on the design and implementation of CALock for hierarchical data. Several avenues could benefit from CALock and avenues where CALock could be extended.

6.3.1 CALock labelling for connected data

Multidimensional data CALock labelling optimizes grain size and guarantees mutual exclusion for multi-granularity locking. However, CALock labelling, on its own,

can be used to optimize the indexing and querying of hierarchical data. CALock labels are similar to a Dewey decimal system [Swe83], which has proven its efficacy in searching multidimensional data by drilling down using labels. However, unlike the Dewey decimal system, CALock's labelling scheme is elastic and can be updated without expensive computation. Future work could study the suitability of path-based labelling techniques for data involving more than two search dimensions.

For index structures in JSON stores or SQL databases, CALock can be used as an effective synchronization mechanism to ensure index consistency 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 optimize hierarchical indices.

Graph data CALock is designed for semi-structured data and can be improved for generic graphs. The locking protocol uses a labelling scheme that depends on a hierarchy having a designated root. In most graph system use cases, such as 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 graph's structure to designate a root. For example, some users have more connections in a social network 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 the graph's root. 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 more likely to be traversed than others. We call such paths *hot paths* (η). Like CALock, a labelling scheme 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.

6.3.2 CALock for distributed synchronization

CALock is designed for multithreaded 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 effective hierarchical synchronization to manage resources.

CALock can be used 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 node at which an operation involving multiple actors can be synchronized. In actor systems with dynamic membership, this hierarchy can undergo structural changes. CALock can guard the synchronization hierarchy against breaking changes by preventing actors from joining or leaving the system while in a critical section and ensuring that the structural changes to a hierarchy are consistently applied across all actors.

Future work should explore the design and use of CALock for an 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 the BEAM VM.

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. 15).

[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. 100).

[BS77] Rudolf Bayer and Mario Schkolnick. "Concurrency of Operations on B-Trees". In: *Acta Informatica* 9 (1977), pp. 1–21 (cited on pp. 25, 42).

[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. 15).

[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. 100).

[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. 69).

[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. 69).

[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. 51).

[Dat00] C. J. Date. *An introduction to database systems (7 ed.)* Addison-Wesley-Longman, 2000 (cited on p. 15).

[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. 69).

[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. 69).

[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. 99).

[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 pp. 45, 47).

[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. 69).

[Geo11] Lars George. *HBase: The Definitive Guide: Random Access to Your Planet-Size Data*. O’Reilly, 2011 (cited on p. 16).

[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. 51).

[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. 28, 42).

[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 pp. 69, 80).

[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. 31, 69, 80).

[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. 69).

[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. 69).

[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. 25).

[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. 25).

[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. 69).

[Mic22] Microsoft. *Transaction locking and Row Versioning Guide - SQL Server*. 2022 (cited on p. 28).

[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. 38, 69, 80).

[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. 34, 69, 80).

[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. 69).

[Ray13] Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer, 2013 (cited on p. 15).

[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. 28).

[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. 69).

[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. 100).

[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. 51).

[Swe83] Russell Sweeney. “The Development of the Dewey Decimal Classification”. In: *J. Documentation* 39.3 (1983), pp. 192–205 (cited on pp. 43, 99).

[Tar72] Robert Endre Tarjan. “Depth-First Search and Linear Graph Algorithms”. In: *SIAM J. Comput.* 1.2 (1972), pp. 146–160 (cited on p. 51).

[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. 69).

[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. 100).

[Wya13] Derek Wyatt. *Akka Concurrency*. Sunnyvale, CA, USA: Artima Incorporation, 2013 (cited on p. 100).