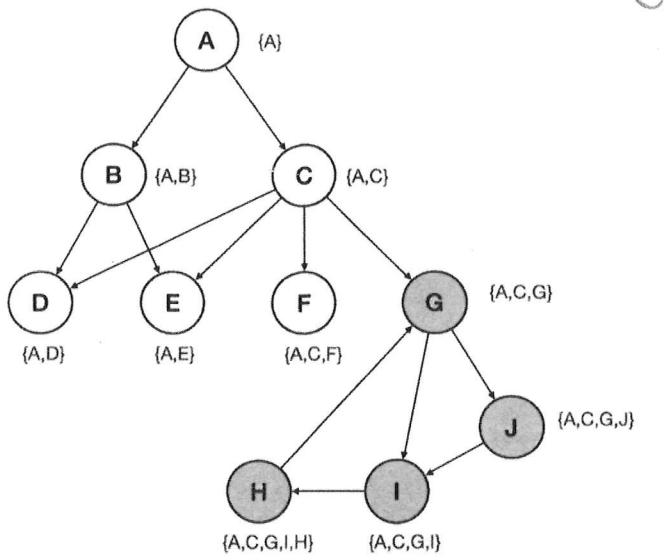


Fig. 4.2: CALock labels for a hierarchy containing strongly connected component (cyan)

Labelling strongly connected components

A strongly connected component is a maximal subgraph of a directed graph where every vertex is reachable from every other vertex. In Figure 4.2, the cyan vertices form a strongly connected component. Theorem 1 assumes that the graph does not contain strongly connected components. In real-life applications, maintaining this constraint is difficult. There are strategies to find strongly connected components in a directed graph [Sha81; Tar72; CM96] that can be contracted by vertex contraction [Gra+08]. We, however, maintain that finding and eliminating strongly connected components is not required to identify lock grains using CALock when the labels are assigned using Relation 4.2.

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 in different orders and hence, every vertex has the same set of guarding ancestors as the vertex which is the entry point to this connected component. The function `BFLabel` recurses over vertices in N until all the paths to a vertex are explored and its label does not change anymore (line 15). Therefore, Relation 4.2 is sufficient to label rooted directed graphs with strongly connected components. For example, in Figure 4.2, the vertices G, H, I, J form a strongly connected component. This component is connected to the rest of the graph via vertex G which is present in the labels of all the vertices in the component.

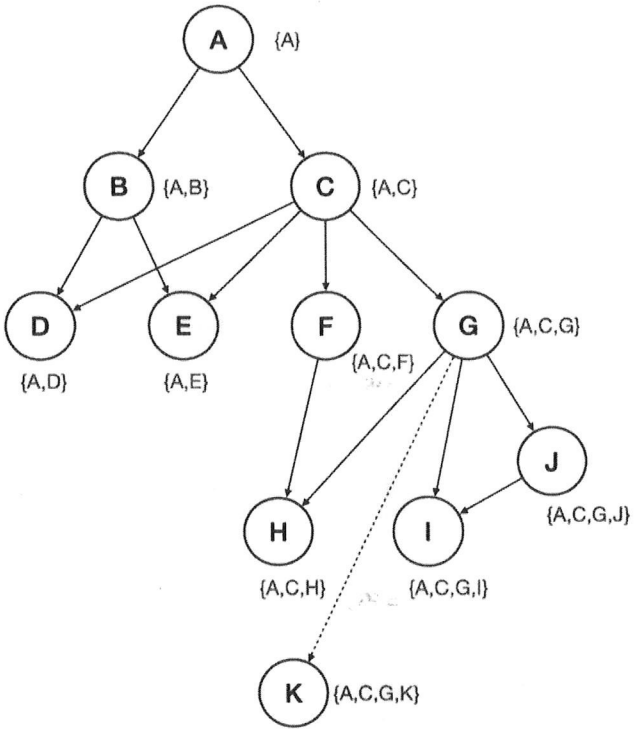


Fig. 4.3: CALock labels for a hierarchy with structural modifications

Relabelling graphs

The topology of a graph can change due to *structural modifications*. Then, the labels of the vertices need to be recomputed. The same function `AssignLabel(v)` can be used to relabel the graph. Relabelling begins at the vertex affected by the structural modification via the function `AssignLabel(v)`. In the same fashion as the initial labelling, the parents of the affected vertex are used to compute its label and then recursively of its children. Relabelling is terminated as soon as it reaches a fixpoint (line 15).

Consider the example in Figure 4.3. A new vertex K is inserted as a child of G. The label of K is computed by taking the intersection of the labels of its parents. Since K has only one parent, it's label is $\{A, C, G, K\}$. Unlike DomLock, MID and FlexiGran, CALock does not relabel the entire hierarchy when a structural modification occurs. It only relabels the affected vertices and their children. This is useful to not only parallelize structural modifications but also to reduce the overhead of relabelling the entire hierarchy.

5 CALock: Multi-Granularity locking and conflict detection

Once a hierarchy is labelled, threads can utilize the labels to lock the graph at different granularities. To this end, the CALock algorithm is proposed. The CALock algorithm is shown in Listing 2. It involves three steps.

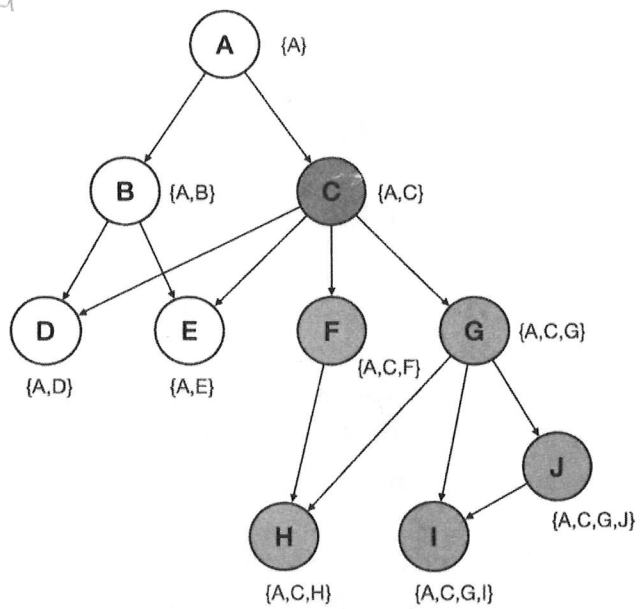
1. **Lock grain identification**: A thread identifies the grain it wishes to lock by using the labels of the vertices in its lock request. This grain is guarded by a lock guard which is to be locked by the thread.
2. **Lock request creation**: The thread proceeds to create a lock request for the guard and adds it to a lock pool. The thread then checks for conflicts with other lock requests in the pool and blocks if a conflict is detected.
3. **Lock conflict detection**: In CALock, two conditions indicate a lock conflict. If both conditions hold for a pair of lock requests, they are in conflict.

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

5.1 Lock grain identification

In order to lock a set of vertices, the approach in CALock is to find the smallest subgraph that contains all the vertices to be locked, such that all the paths to these vertices an be guarded to avoid race conditions. This achieved by finding the *Lowest Guarding Common Ancestor* (LGCA) of the vertices to be locked. As shown in Theorem 1, the LGCA is the deepest vertex in the graph that is an ancestor of all the lock targets. It is present on all the paths that lead to the vertices to be locked and guards them, serving as a lock *guard*.

For example, in Figure 5.1, to lock H and J, we find their LGCA by taking a set intersection of their labels i.e., {A, C, H} and {A, C, G, J} respectively. The set



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

Fig. 5.1: CALock labels

intersection is {A, C} in which is C is the deepest vertex. The thread then proceeds to create a lock request with C as the lock guard.

5.2 Lock requests and the lock pool

A lock request in CALock is a set of data that identifies the grain of a lock request in the hierarchy without the need to traverse the graph. To achieve this, the lock request contains the following information.

- **Thread ID**: of the thread requesting the lock.
- **Guard ID**: of the vertex being locked in order to lock the set of targets.
- **Guard label**: Label of the lock Guard vertex
- **Sequence number**: allocated to the lock request when it is added to the lock pool.
- **Activity Indicator**: used by other threads to wait in the presence of conflicts.
- **Lock mode**: read or write

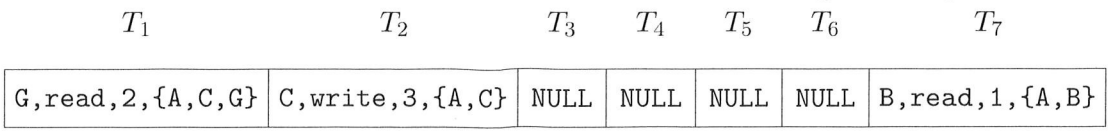


Fig. 5.2: Lock pool containing details of locks held by threads

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

Lock Pool

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

A snapshot of the lock pool is shown in Figure 5.2. It contains 3 lock requests from threads T_1, T_2 and T_7.

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

Tab. 5.1: Lock requests in the lock pool

With this additional information, the lock request is added to the pool (Listing 2 line 8). In order to conserve the first-come-first-serve order and prevent race conditions where a later lock request is granted first before the thread with an earlier request could detect the conflict, the three steps involving the assignment of a sequence number, setting the activity indicator and addition to the lock pool are done atomically by acquiring a mutex on the pool. The thread then proceeds to check for conflicts and blocks if one exists. In the absence of conflicts, the thread proceeds to its critical section.

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

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

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

5.3 Identifying lock conflicts

In MGL on graphs, conflict detection is not as simple as testing for read/write conflicts. Since lock guards have a grain that they protect, it is necessary to ensure that the grain remains exclusive for a writer. With CALock, two conditions indicate a lock conflict.

- **Mode conflict**: is checked by comparing the lock modes in the requests.
- **Grain overlap**: is checked using the guard ID and guard label of the requests. For two lock requests R_1 and R_2, if the guard ID of $R1$ is present in the guard label of R_2 or vice-versa, then the grains overlap.
- **Arrival order**: If there is a mode and grain conflict, the request with a lower sequence number is granted first.

A thread checks these conditions (line 11) by iterating over the pool from left to right (Listing 2 line 10). An iterating thread checks conflicts with all other threads and blocks at the first conflict it encounters. Once this conflict is resolved, the iterating thread proceeds checking conflicts with the remaining threads in the pool. Once it reaches the end of the lock pool, it is guaranteed to either have no conflicts or have priority over all conflicts because of its sequence number. We discuss this more in Chapter 6. When requesting a lock, any thread T can be blocked for a maximum of n turns where n is the maximum number of threads. After n blocks, the sequence number of T will be the lowest among all requests in the lock pool and it will be allowed to proceed. This is essential to prevent thread starvation.

Figure 5.3 shows the snapshot of a lock pool. Figure 5.3 shows the grains on a hierarchy for the locks requested by threads in Figure 5.2. This pool contains three active threads T_1, T_2 and T_7. T_7 arrives first and is assigned the sequence number 1. T_1 and T_2 arrive later and have sequence numbers 2 and 3 respectively. T_1 and T_2 conflict because they have overlapping grains. T_2 is requesting a lock on C which overlaps with the grain locked by T_1. This is detected by checking if C is present in

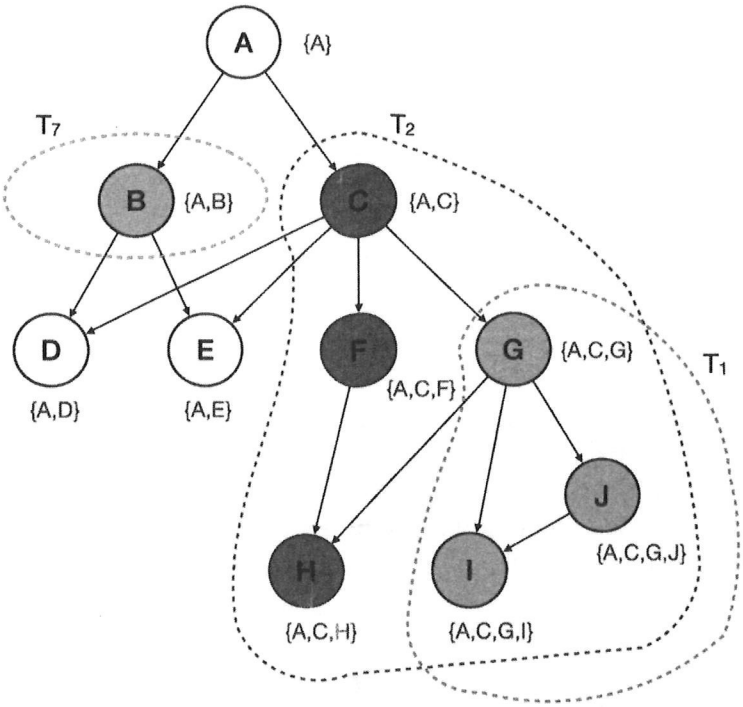


Fig. 5.3: Lock grains on the hierarchy for the locks requested by the threads in the lock pool(Figure 5.3)

the label of G i.e. $C \in \{A, C, G\}$. To resolve the conflict, T_1 is granted a lock since it arrived first and T_2 has to wait for T_1 to release the lock. T_7 does not have a conflict with any thread and acquires a lock on B.

5.4 Structural modifications, locking and relabelling

CALock can be utilized for static graphs whose structure does not change and for dynamic graphs that change at runtime. When a structural modification occurs, the grain in which the modification takes place needs to be relabelled. Unlike DomLock, MID and FlexiGran in which the relabelling happens under a global mutex, relabelling in CALock is done under the same lock that is acquired to perform the structural modification. Here, we explain the locking and relabelling mechanism for dynamic graphs.

Algorithm 2 Lock acquisition request in the lock pool

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

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

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

5.4.1 Vertex addition and deletion

Adding a vertex to the graph does not change the observable topology because this new vertex is not connected to the graph and hence it is also not reachable from the root. So, addition does not require locking or relabelling of any kind. Deleting a vertex that does not have any edges does not require synchronization either because a disconnected vertex is not reachable from the root of the graph.

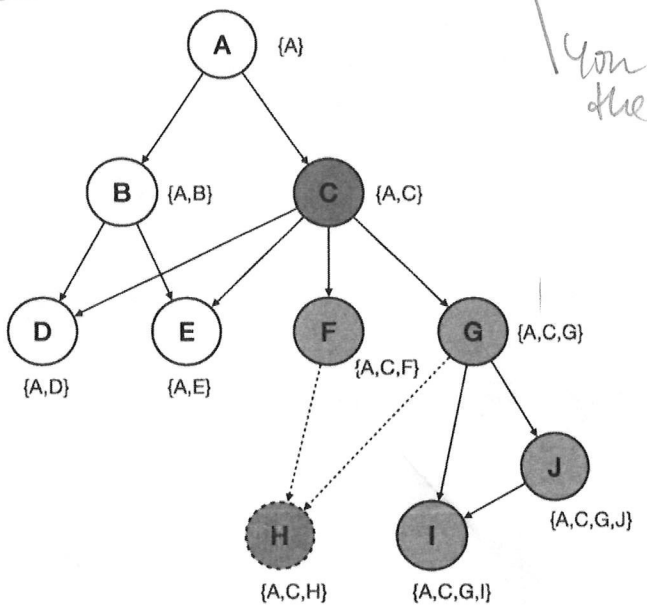


Fig. 5.4: Deleting vertex H in CALock requires a lock on C

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

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

5.4.2 Edge addition and deletion

Adding and deleting edges to the graph changes its topology and also the paths to the vertices. Both operations are performed under a write (exclusive) lock. When adding or deleting an edge between a source vertex u and a target vertex v, a write (exclusive) lock is taken on the LGCA of u and v. Both operations change the ancestors of a vertex so relabelling is initiated at the target (v) of the affected edge using `BFLabel()` function ~~in Algorithm~~ 1.

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

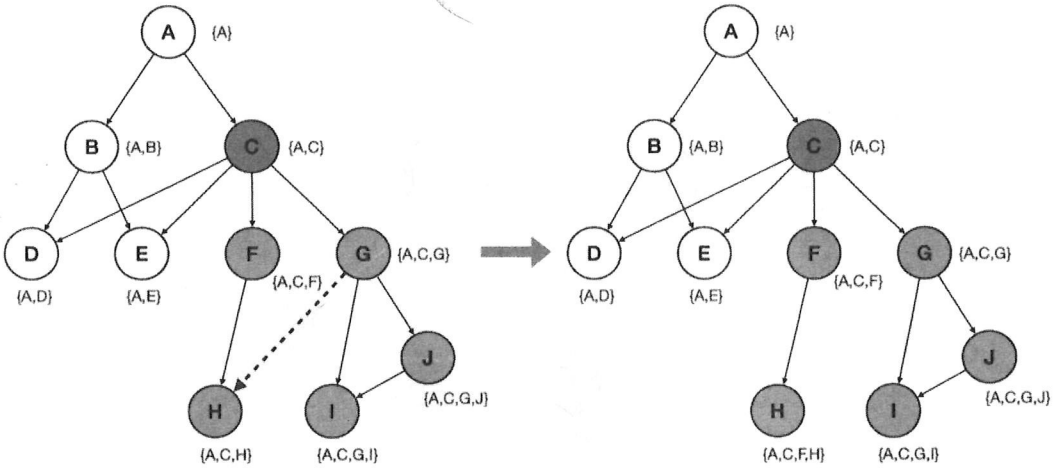


Fig. 5.5: Deleting the edge between G and H requires a lock on C and relabelling of H

6 Properties of CALock

The properties of any algorithm hold importance in guaranteeing behavioural stability. In this section, we show that CALock is safe, live and does not lead to thread starvation. 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. CALock requires the assignment of lock request sequence numbers, setting of the activity indicator and addition of the lock pool to happen atomically. In our implementation, we use a mutex to achieve this.

6.1 Correctness guarantees

6.1.1 Safety

Property A thread holding a write lock on a guard has exclusive access to the corresponding grain. This means that when a thread requests a write lock on a guard, it is granted the lock only if no other thread can access the grain protected by this guard.

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

At the entry to the critical section, T_1 (resp. T_2) iterates over the pool to check for conflicts of three kinds (Mode conflict, grain conflict, sequence number conflict). Since the iteration is always done in the same order, T_1 would detect a mode and grain conflict with T_2 and block if its sequence number is higher and vice-versa. Hence, T_1 and T_2 are never, simultaneously granted a lock on v. Therefore, CALock is safe.

6.1.2 Liveness

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

Discussion In a correct implementation of an application that uses CALock, when a thread leaves its critical section, it releases the lock it holds and this is guaranteed to happen i.e. threads do not sleep after lock acquisition. The lock pool prevents a thread from holding multiple locks. If a thread wishes to lock multiple vertices, it requests a lock on their LGCA. Since threads hold at most one lock at any given time, deadlocks never occur.

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

6.1.3 Fairness

Property When a thread requests a lock, it is granted the lock after a ~~defined~~ number of blocks which is equal to the number of places 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 ~~blocked~~ by a thread that has a lower sequence number. A thread can be blocked and bypassed by other threads in the presence of conflicts at most n times (n is the size of the lock pool). Since sequence numbers are assigned atomically and are monotonically increasing, a thread is granted its lock after at most n bypasses. After a thread is bypassed n times, the sequence number of that blocked thread would be the lowest and it would be granted the lock it requested. This ensures that no thread starves and CALock is fair.

6.2 Complexity analysis of CALock

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

6.2.1 Labelling and relabelling

The labeling 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 since each vertex is connected to every other vertex.
- **Label computation**: For each vertex, when it is visited during the traversal step, an 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. A vertex with more parents has more paths from the root and hence, 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, $e = |E|$ be the number of edges and d_{avg} be the average degree of a vertex. According to the Handshake lemma [Eul41], the average degree of a vertex is

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

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

$$T(BFS) = \theta(v + v.d_{avg}) \tag{6.2}$$

For each vertex, the size of its label is inversely proportional to the number of parents it has and consequently, label computation is inversely proportional to the average degree which is:

$$T(Label\ Computation) = \theta(\frac{v}{d_{avg}}) \tag{6.3}$$

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

$$T(Labelling) = \theta(v + v.d_{avg} + \frac{v}{d_{avg}}) \tag{6.4}$$

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 $O(v^2)$.

6.2.2 Conflict detection

When a thread requests a lock, it finds the LGCA of the lock targets and issues a lock request. The LGCA is computed via a set intersection of the labels of the lock targets. If the lock request is for q lock targets and the average label size is $\theta(\frac{v}{d_{avg}})$, the complexity of finding the lock guard is:

$$T(Lock\ Guard\ Computation) = \theta(q \times \frac{v}{d_{avg}}) \tag{6.5}$$

Since $q << v$ for real-world applications, the average case complexity can be reduced to:

$$T(Lock\ Guard\ Computation) = \theta(\frac{v}{d_{avg}}) \tag{6.6}$$

In the best case, the lock request is for a single vertex which does not require LGCA computation and the complexity is $\Omega(1)$. In the worst case, the lock request is for all vertices of the graph and the complexity is $O(\frac{v^2}{d_{avg}})$.

These labels are then used to detect conflicts. A thread checks conflicts with all other threads i.e. n times. For each conflict, a set membership test is performed. With an efficient set implementation, the membership can be tested in $O(1)$. The complexity of conflict detection is $\theta(n)$.

6.3 Complexity Comparison

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

Since intention locks do not have additional metadata, 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 $\theta(v^2 + \frac{v^2}{d_{avg}})$.

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

	Labelling	Lock Guard computation	Conflict detection
Intention Lock	-	-	$\theta(v^2 + \frac{v^2}{d_{avg}})$
DomLock	$\theta(v^2 + \frac{v^2}{d_{avg}})$	$\theta(v^2 + \frac{v^2}{d_{avg}})$	$O(n)$
MID	$\theta(v^2 + \frac{v^2}{d_{avg}})$	$\theta(v^2 + \frac{v^2}{d_{avg}})$	$O(n)$
FlexiGran	$\theta(v^2 + \frac{v^2}{d_{avg}})$	$\theta(v^2 + \frac{v^2}{d_{avg}})$	$O(n)$
CALock	$\theta(v^2 + \frac{v^2}{d_{avg}})$	$\theta(\frac{v}{d_{avg}})$	$O(n)$

Tab. 6.1: Average case complexities of MGL techniques

	Labelling	Lock Guard computation	Conflict detection
Intention Lock	-	-	$\theta(n)$
DomLock	$\theta(v^2)$	$\theta(v^2)$	$\theta(n)$
MID	$\theta(v^2)$	$\theta(v^2)$	$\theta(n)$
FlexiGran	$\theta(v^2)$	$\theta(v^2)$	$\theta(nv^2)$
CALock	$\theta(v^2)$	$\theta(1)$	$\theta(n)$

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

techniques perform either a DFS or a BFS traversal over the graph and have the same time complexity. CALock, however, is linear in the number of vertices when computing the lock guard. This is because CALock uses set intersection to compute the lock guard. DomLock, MID and Flexigran use integer intervals and then perform a DFS traversal to find the lock guard corresponding to the lock request. For very complex graphs, CALock labels are smaller than the integer intervals used by DomLock, MID and Flexigran. This results in a lower complexity for CALock in worst-case scenarios.

7 Experimental Evaluation

With benchmark experiments, we evaluate the performance of CALock and compare it with the state of the art against several parameters. Our evaluation uses the same benchmark experiments as DomLock, MID and FlexiGran [KN16; AN22; AN24] i.e. STMBench7[GKV07]. In this chapter, we present the experimental setup which includes the benchmark and the workloads. We then present the results of the experiments and discuss the implications of the results.

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. The benchmark is deployed on a Docker container under Ubuntu 20.04. To build the benchmark, GCC 12.1 and Cmake 3.22 are used. The compilation is done at the C++ 26 standard.

7.1 Benchmark Suite: STMBench7

STMBench is a well known benchmark based on the classical OO7 object oriented benchmark suite [CDN93]. It is used to evaluate the performance of hierarchical algorithms and data structures [Pro+19; Val+16; Fel+16; CC16; KPR15; FB15; RC14; KN16; AN22; AN24; KN18; GKN18; LZ14]

The hierarchy in STMBench7, as represented in Figure 7.1, consists of a *module* at the root level. This module consists of several levels of *complex assemblies*. Each of the deepest complex assemblies consists of a set of *base assemblies*. A *composite part* can be contained in several base assemblies. Each composite part contains a set of *atomic parts*. These atomic parts form a near-complete graph. The root of this graph is connected to a single composite part.

STMBench7 provides coarse-grain and medium-grain lock implementations. These are representatives of fixed granularity locking techniques, used in practice. The coarse lock is a reader-writer lock on the entire graph. Figure 7.1 shows the medium-lock grains for a module. Each level of complex assemblies is guarded by a single lock. Deeper in the graph, the atomic parts are guarded by a lock of their own.

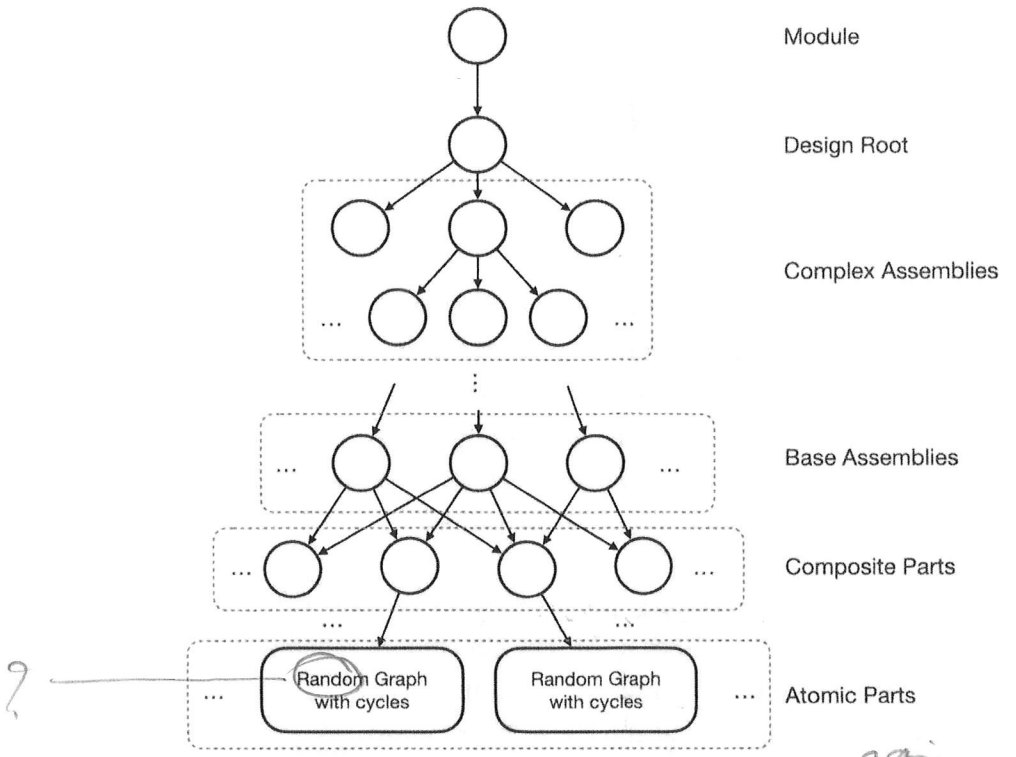


Fig. 7.1: Structure of a module in STMBench with medium lock boundaries

Structural modification operations take a mutex on the entire graph and happen in isolation. STMBench uses the following operations to stress the locking algorithms.

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

7.1.1 Synopsis

The research questions addressed by the benchmarks are the following:

§7.2 How long do different operation types take to complete under different locking strategies?

§7.3 What is the cost of maintaining metadata for CALock compared to DomLock, MID and FlexiGran?

§7.5 What are the relative sizes of lock grains between CALock, DomLock, MID?

§7.6.1 What is the performance of CALock compared to other lock strategies for locking on static graphs?

§7.6.2 What is the performance of CALock compared to other lock strategies for locking in dynamic graphs?

7.2 Per operation latency

Figure 7.2 shows the time to completion for different operations in STMBench7. For coarse and medium-grained locks, reads are fast and finish relatively quickly since they occur under pthread read locks which do not require additional metadata. Write operations, due to their nature are slower since they require exclusive access to the graph. However, with additional performance metrics (see Sections 7.6.1, 7.6.2), we observe that fixed-grain locks do not scale.

Intention locks often encounter deadlocks. This is because, unlike trees, the paths to a vertex are not unique. As such, a thread needs to intention-lock all the vertices on all the paths to a target vertex. Concurrent threads are not guaranteed to lock the all vertices in the same order and hence, deadlocks occur. When a deadlock is detected, the lock request is retried. If the deadlock persists, the lock request is rejected and counted as failed.

MGL techniques like DomLock, MID, FlexiGran and CALock are slower than coarse- and medium-grained locks. They are slow for reads and writes due to the presence of false subsumptions between grains causing spurious thread-blocks. For structural modifications, DomLock, MID and FlexiGran spend time relabelling the hierarchy which is very expensive.

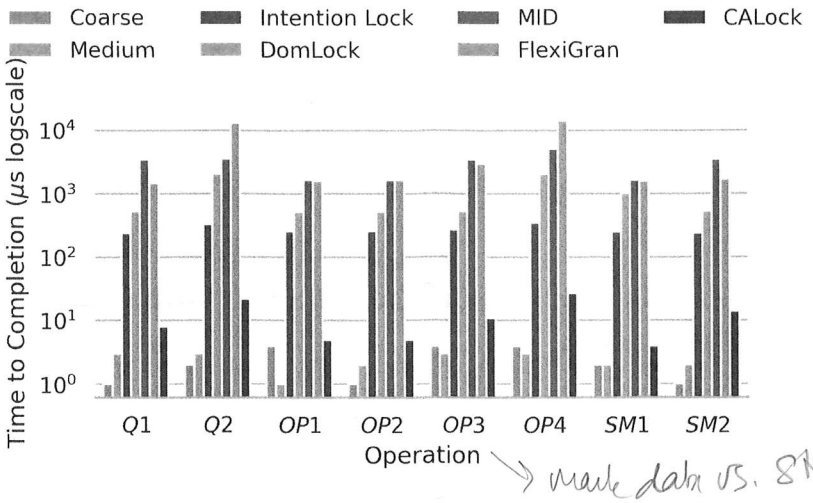


Fig. 7.2: Time to completion for different operations(lower is better)

CALock is faster than DomLock, MID and FlexiGran for reads and writes since it avoids false subsumptions (see Section 7.5). For structual modifications, unlike DomLock, MID and FlexiGran, the relabelling can be parallelised if the lock grains do not overlap. When deleting vertices from the hierarchy (SM1), CALock does not require relabelling.

7.3 Metadata management: bulk labelling and relabelling

7.3.1 Bulk labelling

We measure the time it takes to assign the labels to a graph when it is first created. This simulates loading data into a database. Coarse-grain, medium-grain locks and intention locks do not need labelling. This time is measured for three different sizes of the STMBench7 graph. The results are shown in Figure 7.3. We observe that DomLock is the fastest at preprocessing the hierarchy since a single depth-first-traversal is sufficient to compute the intervals. MID is slower than DomLock because it needs to compute two pairs of intervals for each vertex. One pair the intervals 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 DomLock but slower than MID because of the additional level information that needs to be computed per vertex.

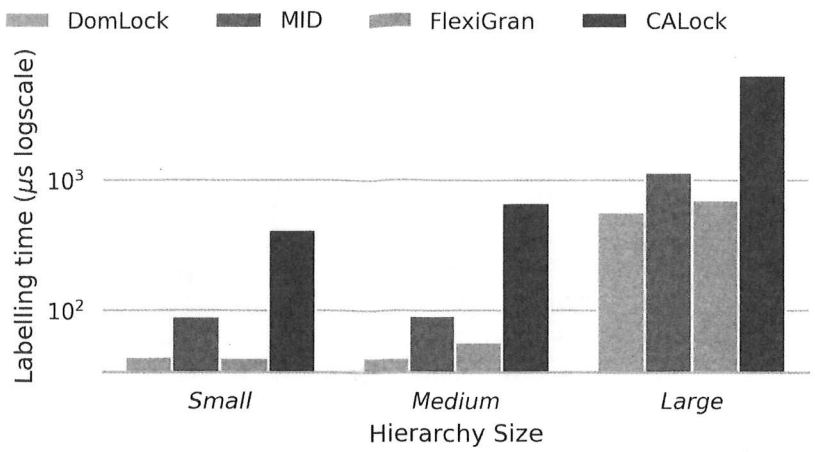


Fig. 7.3: Time to compute initial labels (lower is better)

CALock labels take the logest to preprocess since the labels are defined by a recursive breadth-first traversal with a fixpoint dependent on the number of paths to a vertex from the root.

7.3.2 Relabelling

Figure 7.4 shows the average time spent relabelling the graph per structural modification. ~~Both~~ Coarse, medium grained locks and intention locks do not have ~~additional metadata~~ and hence do not require relabelling. In CALock, relabelling is significantly faster than DomLock, MID and FlexiGran. This is because in DomLock, MID and FlexiGran a structural modification anywhere in the graph changes the depth-first traversal order of several vertices of the graph. A change in this traversal order requires re-computing a lot of intervals which propagate to the root of the hierarchy. Since the root is involved in the relabeling, intervals are computed under a mutex on the graph which prevents parallel relabelling of disjoint subgraphs leading to poor performance of DomLock, MID and FlexiGran.

In contrast, CALock relabels only the ~~affected~~ subgraphs directly affected by a structural modification. ~~CALock labels always popagate away from the root towards the leaves and are computed under the lock that is acquired to perform~~ the structural modification. Thus, multiple grains can be locked, modified and relabelled in parallel. CALock is 2× faster at relabelling than DomLock, MID and FlexiGran.

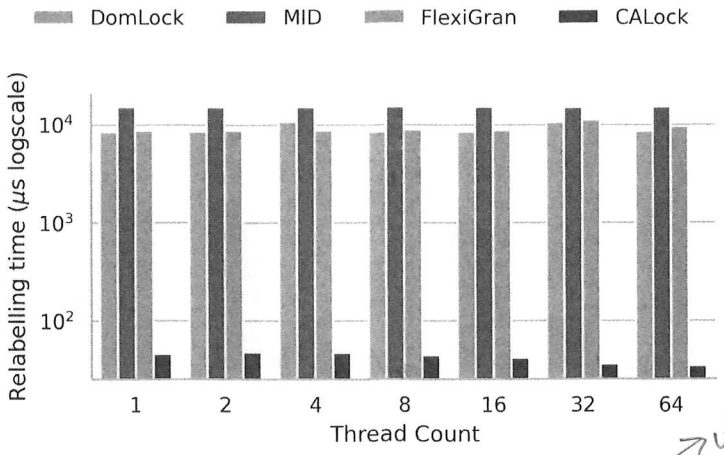


Fig. 7.4: Time spent relabelling the graph per modification operation (lower is better)

7.4 ~~Metadata~~ size in memory

All MGL techniques utilise metadata to identify the lock grains. DomLock utilizes integer ranges as labels, which ~~require less memory due to their compact representation~~. MID uses two pairs of integer ranges ~~to represent the intervals of the vertices~~. FlexiGran uses the DomLock intervals along with an integer to store the level of a vertex. In contrast, CALock employs sets of vertex identifiers as labels, leading to a larger memory footprint.

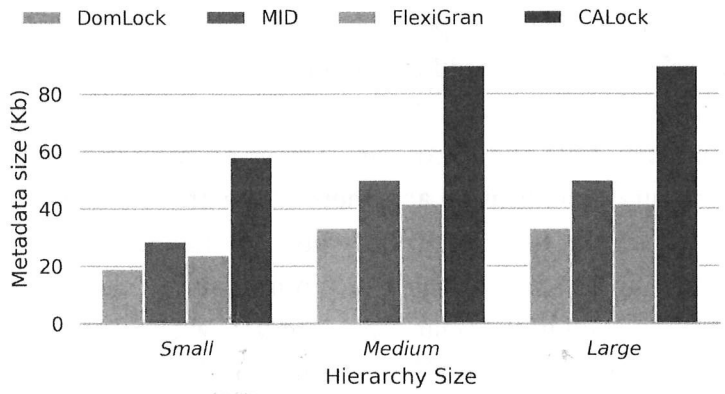


Fig. 7.5: Size of the metadata used for labelling

As shown in Figure 7.5, CALock's metadata consumes approximately 1.5× 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 low memory requirement however, the information about the exact topology of the graph is lost, leading to false subsumptions. CALock labels, being sets, contain

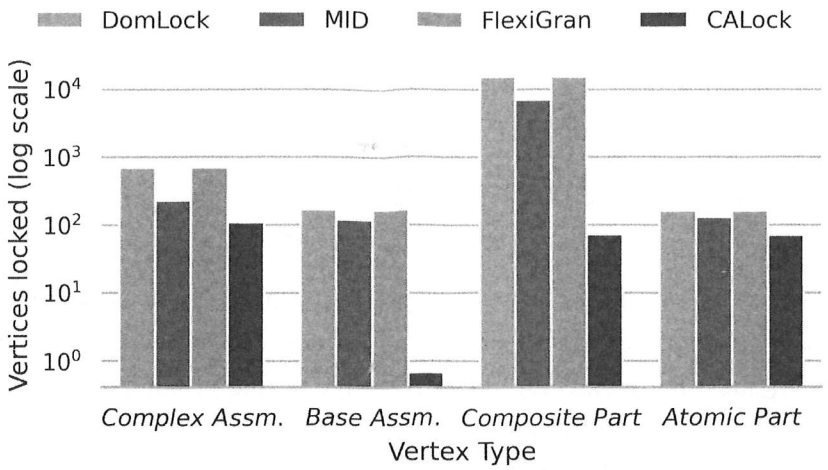


Fig. 7.6: Vertices locked per vertex type (lower is better)

more information allowing for smaller grain sizes and prevent the problem of false subsumptions.

7.5 Lock granularity and false subsumptions

Locking a vertex in MGL implicitly also locks all other vertices present in the lock grain. To compare the grain size for them, we measure the number of targets implicitly locked for a guard vertex in the STMBench7 graph. Figure ?? compares the granularity of DomLock, MID, FlexiGran and CALock.

Locking an atomic part has almost the same effect for all techniques with CALock being marginally better and reducing grain sizes. This is because, in STMBench7, the atomic parts are strongly connected and locking a single vertex often causes the whole graph of atomic parts to be locked. When locking higher up in the graph, the effects vary. Locking a complex assembly is more expensive with intervals from DomLock, MID and FlexiGran because complex assemblies often share composite parts leading to large grains 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, again, owing to false subsumptions.

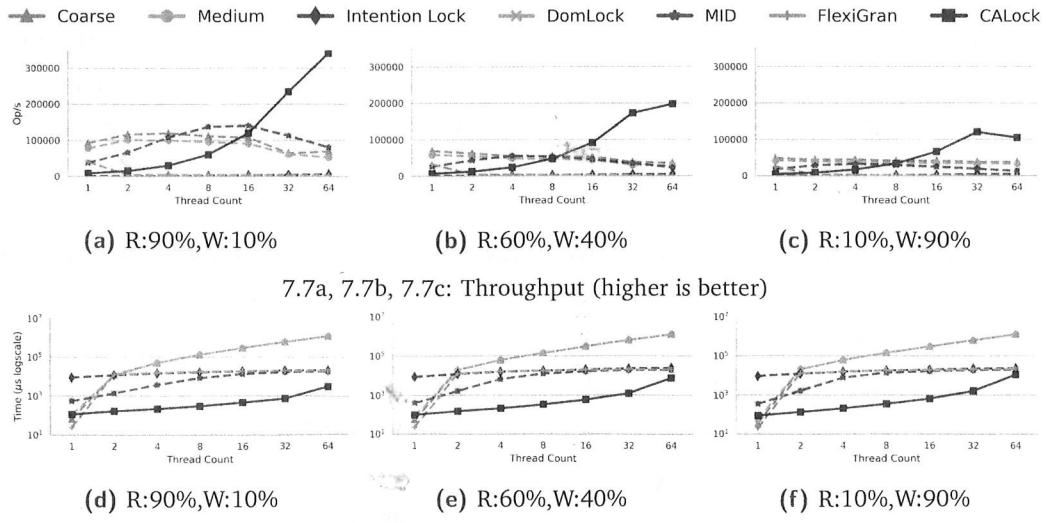
The granularity of the lock and its effect on subsumption is highly dependent on the topology of the graph. False subsumptions aggravate the problem of large grains

and lead to more grain overlaps between threads. The grain sizes of CALock are always smaller other locking techniques.

7.6 Overall locking performance

Sections 7.2, 7.3, 7.4 and 7.5 study individual parameters in isolation. Now, we study all of them together and evaluate their affects on overall performance of the locking techniques. For FlexiGran, the percentage of fine-grained locks is set to 50% as recommended by the authors in their paper [AN24]. In these set of benchmarks, we study both static and dynamic graphs in STMBench under the same workloads. Figures 7.7 and 7.8 show the throughput of different workloads on static graphs and dynamic graphs respectively. The charts in these figures are plotted with the number of concurrent threads on the x-axis and the throughput (op/s) or response time (ns) on the y-axis. Response time is measured from when the thread issues a lock request until the lock is granted.

7.6.1 Static Graphs



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

7.7a, 7.7b, 7.7c: Throughput (higher is better)

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

7.7d, 7.7e, 7.7f: Average time to grant lock request (lower is better)

Fig. 7.7: Performance with different workload types on static graphs (R: reads, W: writes).

Throughput figures 7.7a, 7.7b and 7.7c show that coarse-grain and medium-grain locks perform the best for up to 4 concurrent threads due to the additional computation of the lock grain required for DomLock, MID, FlexiGran and CALock. Beyond