MultiDomLock - A Multi-granular Locking Technique For Hierarchical Data Structures

A Project Report

submitted by

VIVEK DARSANAPU

in partial fulfilment of the requirements

for the award of the degree of

BACHELOR OF TECHNOLOGY

DEPARTMENT OF CSE
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

May 2016

THESIS CERTIFICATE

This is to certify that the thesis titled **MultiDomLock - A Multi-granular locking technique for Hierarchical Data Structures**, submitted by **Vivek Darsanapu**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Rupesh Nasre
Research Guide
Assistant Professor
Dept. of CSE
IIT-Madras, 600 036

Place: Chennai

Date: 5th May 2016

ABSTRACT

KEYWORDS: DomLock; Multi-granular Locking

We introduce an advanced technique, Multi-DomLock, for locking in Hierarchical data structures using the existing DomLock[1] technique. Dom-lock is a multi-granularity locking technique for hierarchical data structures which exploits the structural properties of the hierarchy in the data structure. Multi-DomLock generates multiple locking options from which we can choose depending on what we need the most, concurrency or locking cost. In contrast to DomLock, the technique tries to maximize the concurrency while keeping the locking cost nearly same as the earlier.

TABLE OF CONTENTS

CHAPTER 1

Introduction

Many real world applications such as distributed file systems, relational databases operate on underlying data structures, handling multiple real-time requests to them. These requests have to be dealt effectively in a way which allows maximum possible concurrency among them. The major factors affecting the performance of concurrent programs are the effectiveness of their underlying data structures and the synchronization method that resolves the conflict of race conditions.

The choice of the data structure holds the key to performance in handling large scale data. Hierarchy among data structures is an important property used to organize and reuse the existing models of data. A hierarchy is characterized by a containment relationship, in which the child nodes are contained within their parent nodes. The property of containment can be represented using a directed edge as it is not symmetrical. As a simple example, a file system forms a tree hierarchy with the folders being internal nodes and files being leaf nodes. As we can see, the nodes of the hierarchy can be of different types. By this, we can represent a hierarchical data structure in the form of a directed acyclic graph.

In addition to the hierarchy, handling large number of requests on such data structures requires an efficient synchronization method. Extreme locking methods such as coarse-grain locking and fine-grain locking have low scalability factor and perform poorly. In coarse-grain locking, lock is acquired at the root no matter which node is under operation. This reduces the concurrency heavily. Fine-grain locking technique locks only the requested nodes, thereby failing to take advantage of the hierarchy of the structure as the operations for an hierarchical data structure are target a hierarchical group instead of particular nodes. Therefore, in case of hierarchical structures, both the coarse-grain and the fine-grain locking mechanisms lead to reduced performance.

Multi-Granularity Locks (MGL) are designed for hierarchical data structures to exploit the structural properties. In MGL, instead of locking the root node or the requested nodes, a root node of the substructure which contains all the nodes, is locked. This leads to improvement in the concurrency as the nodes outside the substructure are still access-able for the other operations. Also note that, the locking cost is significantly low. Coarse-grain and fine-grain can be viewed as corner cases of the multi-granularity locks.

Intention locks[4], proposed by Gray et al., and DomLock[1], proposed by Saurabh Kalikar and Rupesh Nasre, are the examples of multi-granularity locks. In Intention Lock method, to acquire a lock on the substructure the nodes which are actually requested are locked under shared or exclusive mode, while the nodes which are in the path are specially locked. This type of distinguished locking helps in identifying the overlapped sub graphs which are being operated by multiple threads. Intention locks shows low performance in the cases of large and tall data structures, as the traversal cost in such structures is huge.

In DomLock, to acquire lock on the requested nodes, we lock the node which appears in all the paths from the root node to any of the requested node. But the DomLock method fails to find better concurrency options, solely concentrating on the locking cost. Here, we present the technique, Multi-DomLock, which is aimed to find such options.

The rest of the thesis is organized as follow. Chapter 2 presents existing DomLock approach and issues therein. Chapter 3 presents Multi-DomLock that solves these issues. Chapter 4 evaluates the performance of Multi-DomLock., and Chapter 6 concludes the work presented in the thesis.

CHAPTER 2

Background and Motivation

Given a directed acyclic graph rooted at node root, we define a dominator as follows.

Definition - Dominator: *A node d is a dominator of a node n if all the paths from root to n pass through d.*

Definition - Immediate Dominator: *A dominator d is an immediate dominator of a node n if there exists no other dominator for n on the paths between d, n and* $d \neq n$

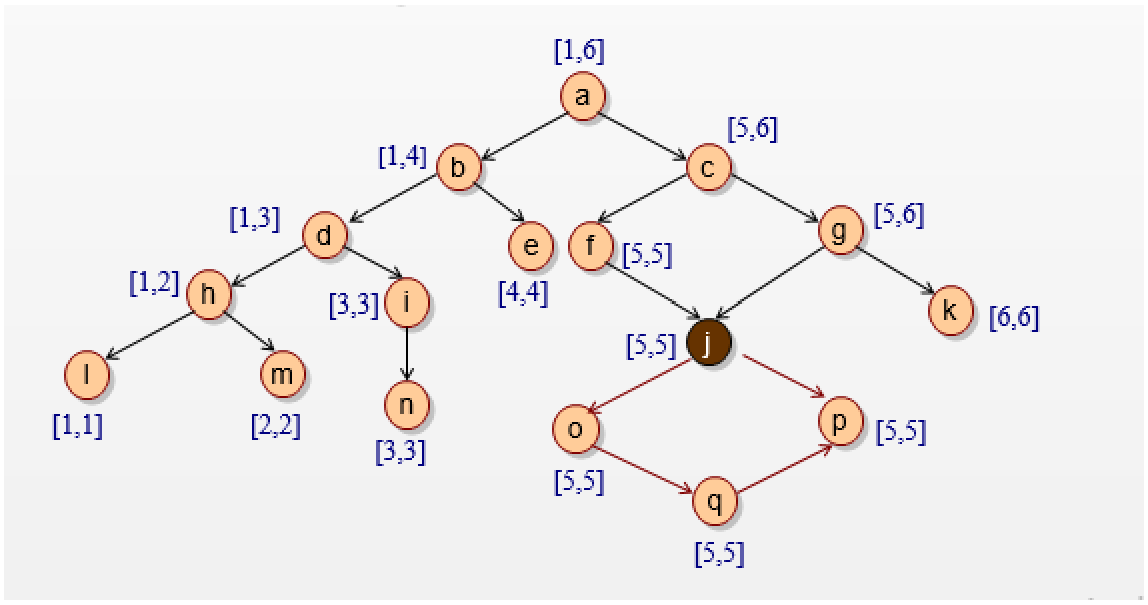

Figure 2.1: Example: A hierarchical data structure, each node's logic intervals

The dominator relationship between two nodes is reflexive and transitive. In our example of Figure 2.1, the root node *a* is the dominator for all the nodes in the data structure including itself. Nodes b and c act as dominators for the left and the right sub-structures of node a respectively. There may be more than one dominator for a given node but only a single immediate dominator is defined. For example, both a and c are dominators for f, but node c is the immediate dominator of node f. In the case of trees, a node's parent is its immediate dominator. For a given set of nodes, the least common ancestor will be the immediate dominator of the set. In case of a DAG, where a node may have more than one paths from the root node, the node that intervenes all the existing paths to that node becomes the dominator of it. For DAGs, the least common ancestor may or may not be the immediate dominator.

The core idea of the DomLock is to find the immediate dominator of the given set of nodes which are to be locked and acquire a lock on it to lock all the required nodes. To do so DomLock maintains preprocessed data in the form of logical intervals associated with every node. We define a function L : V $\rightarrow$ N x N, such that for any node v $\in$ V, L(v) is a pair, say [x, y] with x $\leq$ y, that denotes the logical interval associated with a node v. These logical intervals are used for the graph traversal to find the dominator. They maintain the following property in the hierarchy.

Subsumption Property: *For each node v $\in$ V, the logical interval L(v) subsumes (or super-ranges) L(v') for all the descendant elements v'$\in$ V reachable from v. Thus, if L(v) = [x1, y1] and L(v') = [x2, y2], then x1 $\leq$ x2 and y1 $\geq$ y2.*

By the above property, a domniator of a node super-ranges the interval of the particular node. And hence, a dominator of a given set of nodes *S* super ranges the intervals of all the nodes of *S*. DomLock finds the dominator node for a set of requested nodes to be locked using the above conclusion. There exist situations when it is possible that the dominator ends up locking extra nodes than the ones requested. For instance, in Figure 2.1, if a thread wants to lock nodes m, n and k then node a is the immediate common dominator. Locking a locks every node in the structure which is not required. Alternatively locking d and g will be a better option. This is core reason for evolution of Multi-Domlock, which can look through the options where locking cost is traded for concurrency. In general, if most of the requested nodes are part of a subgraph and a few belong to another, DomLock tend to lock the structure in such a way that does not support more concurrency. The better of way of doing it having two locks separately for each of the sub structures without effecting the connecting components of the graph.

CHAPTER 3

Multi-DomLock

In this section, we present Multi-DomLock, a technique to generate multiple locking options for a given Hierarchical data structure. The main idea of Multi-Domlock is to apply DomLock algorithm iteratively to find dominators for subsets of the requested nodes set. By doing so, we get separate dominators, for separate clusters of requested nodes.

To start with Multi-Domlock applies DomLock algorithm, *FindDominator()* on the graph to obtain the dominator of the set of nodes to be locked. In the next step, it picks one of the children of the dominator and applies the DomLock algorithm on it with the requested set *S* equal to the set of requested nodes which are reachable from the child. This step picks up the sub-dominator for the nodes reachable from the chosen child. By applying the same to all the children of previous dominator, we will have a series of sub-dominators which will cover the entire requested set of nodes which are to be locked. This can be repeated several times as the algorithm eventually ends up with fine-grain locking set.

Algorithm 1 buildS(ptr, S)

```
Require: node, set of nodes to be locked S
Ensure: : set of nodes reachable from given node ptr
1: answer = null;
2: for all n $\in$ S do
3:     if LogicInterval(ptr) overlaps with LogicInterval(n) then
4:         answer.add(n);
5:     end if
6: end for
7: return $answer$
       =0
```

In the running example, let us say, we have to lock m,n and q. Multi-Domlock starts by computing FindDominator() on the root node a, with S = {m,n,q}. The result will be the root node itself. Then it would apply FindDominator() on b with S = {m,n} which would result in node d. It also applies FindDominator() on node c with S = {q} which will result in node j. Now we have completed the procedure for all the children, the resultant sub-dominators will form a set which can lock the requested set. In the current example, we get this set as {d,j}. If the same procedure is continued for one more iteration the algorithm will give the dominators set as {m,n,q} which is nothing but the case of fine-grain locking.

Algorithm 2 MulitDomLock(root, S)

```
Require: root node root, set of nodes to be locked S
Ensure: : set of locking options for nodes in S
1:  ptr ← FindDominator(root, S);
2:  subDominators.add(ptr)
3:  currentChildren.clear();
4:  while subDominators ≠ S do
5:      for all x ∈ subDomniators do
6:          for all y ∈ ProperDescendants(x) do
7:              currentChildren.add(y);
8:          end for
9:      end for
10:     subDominators.clear();
11:     for all z ∈ currentChildren do
12:         childS = buildSet(z, S);
13:         tmp = FindDominator(z, childS);
14:         subDominators.add(tmp)
15:     end for
16:     results.add(subDominators);
17:     currentChildren.clear();
18: end while
        =0
```

The algorithm is guaranteed to terminate as we move down the directed acyclic graph, towards the requested nodes at least by one step. In addition to the above generated options we can also generate locking options by computing the combinations of sub-dominators found at different levels. For example, in Figure 2.1, if we need to lock the nodes {l,m,n,o,k}, the algorithm would return with the options of { {d,c}, {h,i,j,g},{l, m, n, o, k}}. But one can notice the possibility of locking {d,j,g} which exploits the hierarchy along with having lower locking cost. Generating such options can be done by applying the FindDominator() on the children of a dominator in a combinatorial way.(Note that the Multi-Dominator method evaluates all the children in one stretch.). This involves in larger processing time as the combinations can grow exponential which will be unsuited for large structures.

In order to chose from the options we can evaluate them for the attributes of concurrency and locking cost. The factor of concurrency can be inversely related to number of paths being locked in the structure.The other factor locking cost, can be defined in terms of the number of nodes physically locked. For instance, for a request to lock nodes m, l, k in Figure 2.1, fine-grained locking has a cost of 3 units, whereas the coarsest locking (at the root node a) would incur a cost of 1 unit, which is also the solution provided by DomLock. Another sound option of locking the nodes h and g has a cost of 2 units. Our goal is to choose a locking option in order to maximize the amount of path-concurrency and minimize the locking cost, considering the constraints involved.

CHAPTER 4

Experimental results

In this chapter,we measure performance of the options generated by Multi-Domlock. To do so, we spawn threads on a hierarchical data structure to perform a common critical task on selected nodes with Multi-DomLock as the locking method used to avert race conditions. The input parameters we take into consideration are size of the structure, size of the critical section and distribution of the nodes to be locked. Size of the structure is the total number of nodes in the data structure, *N*, size of critical section is number of nodes which are requested to be locked, *n*, and distribution is the order of skewness in position of the requested nodes. Total execution time of threads for few iterative levels of Multi-Domlock are plotted against the size of critical section for different distributions and sizes of a hierarchical tree data structure as follows:

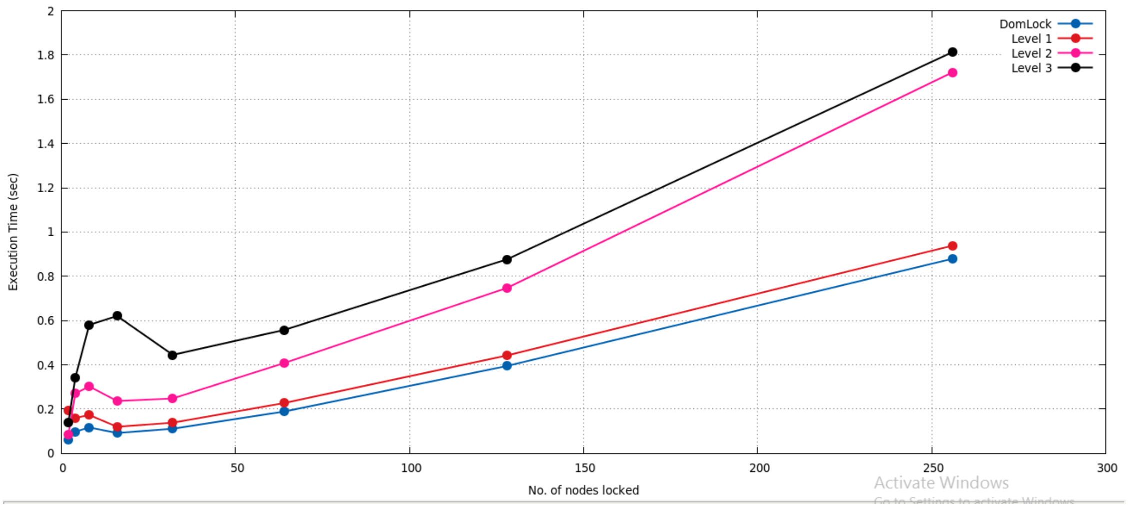

Figure 4.1: N = 1000000, Distribution=1

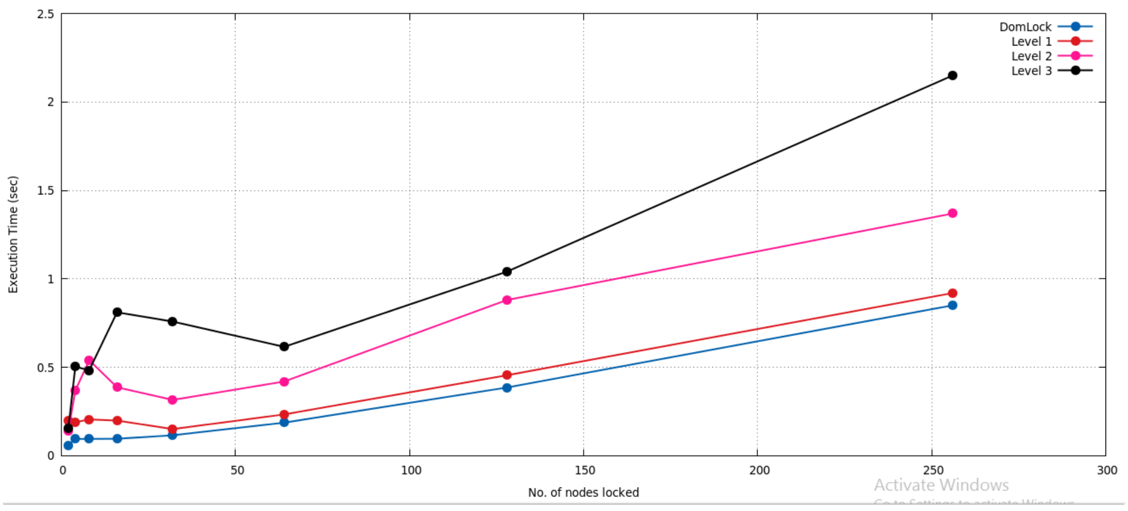

Figure 4.2: N = 1000000, Distribution=2

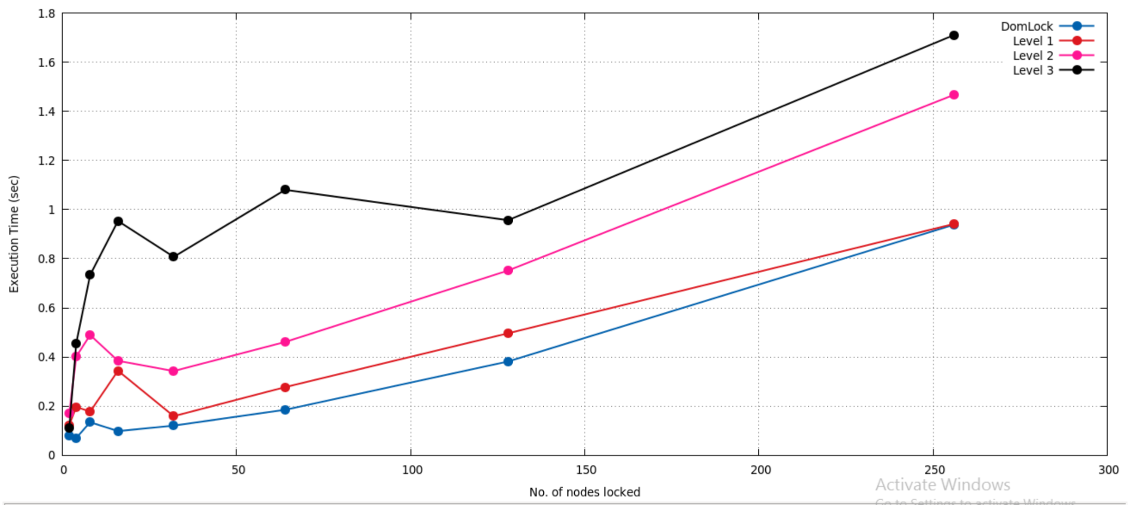

Figure 4.3: N = 1000000, Distribution=4

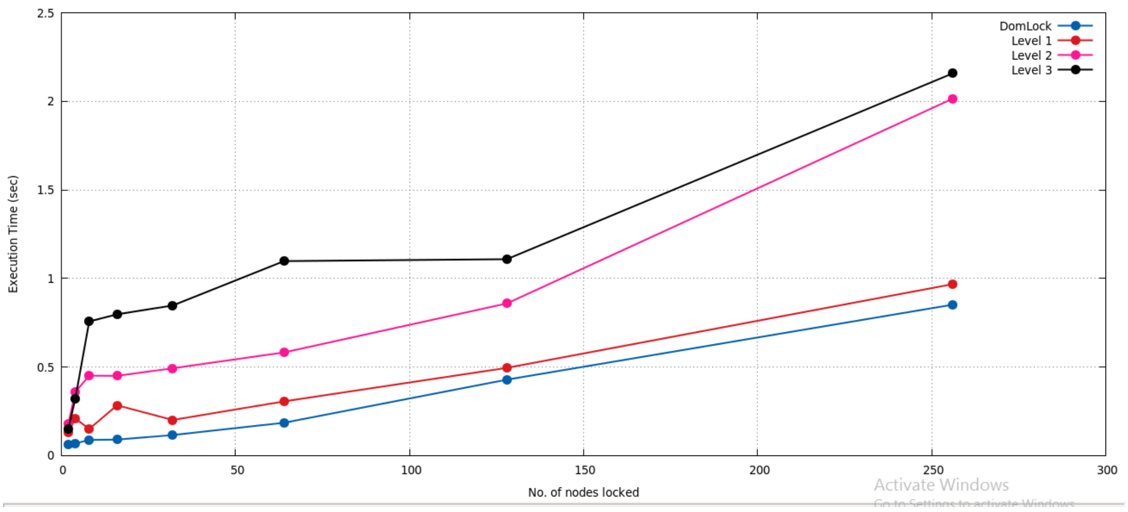

Figure 4.4: N = 1000000, Distribution=8

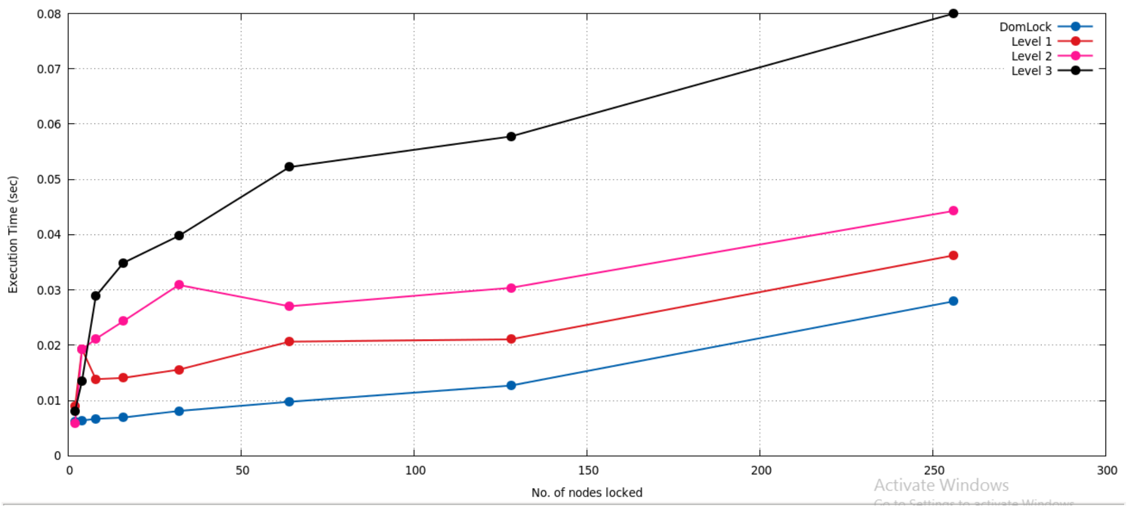

Figure 4.5: N = 10000, Distribution=1

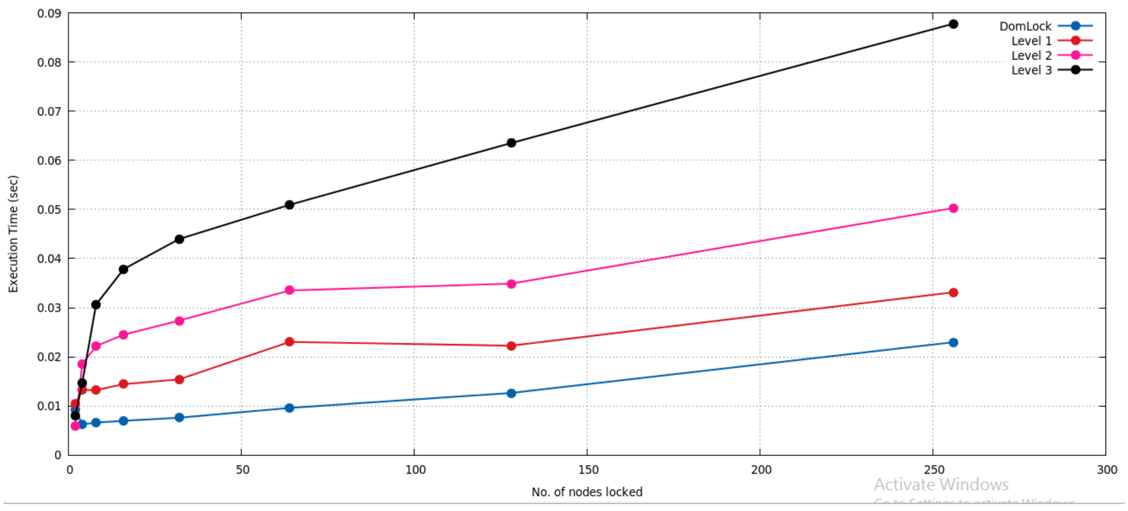

Figure 4.6: N = 10000, Distribution=2

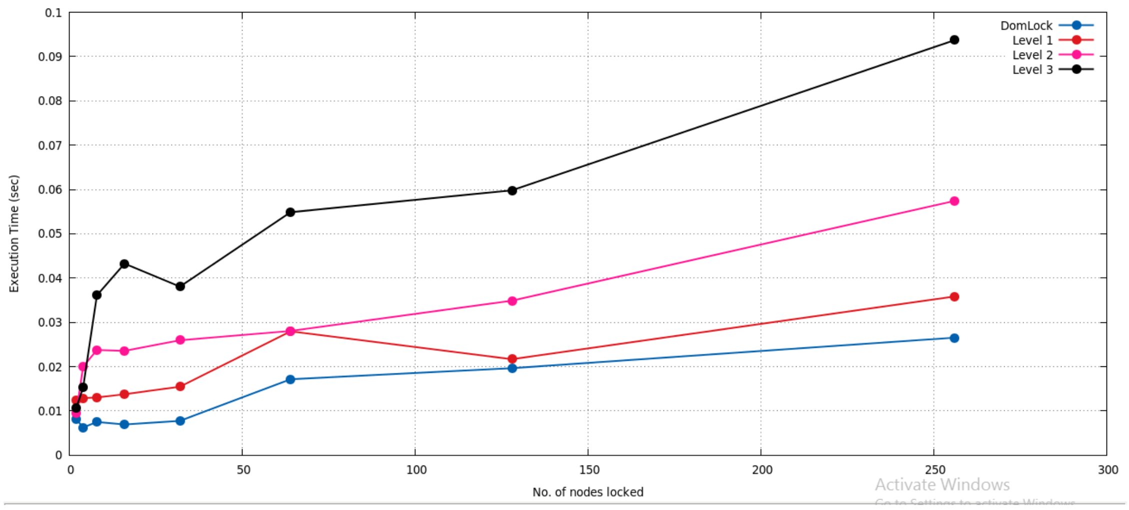

Figure 4.7: N = 10000, Distribution=4

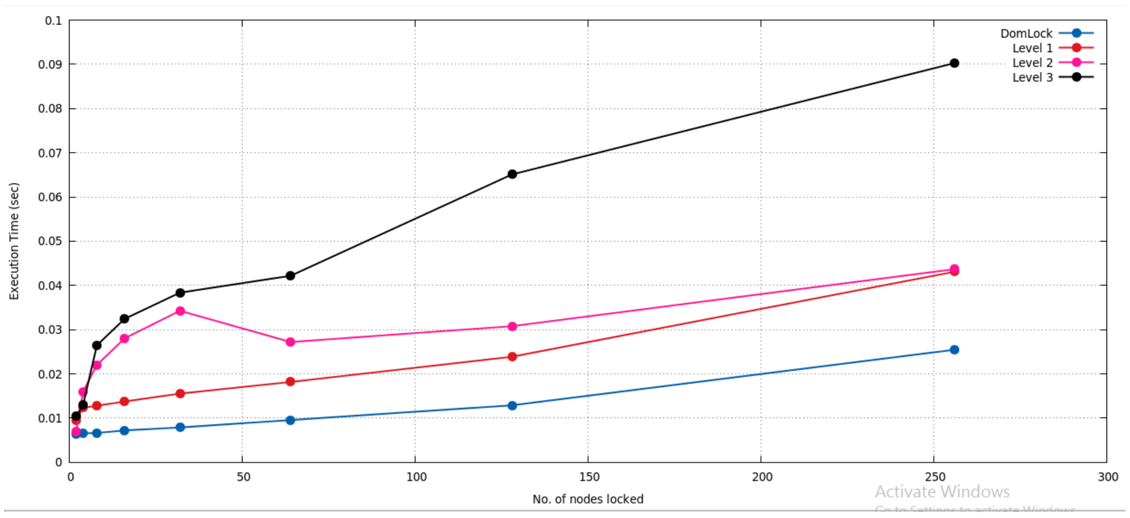

Figure 4.8: N = 10000, Distribution=8

We notice that as the distribution grows in skewness, the improvement in performance for the levels becomes more significant (in Figure. 4.3, 4.7, 4.8). Also as the algorithm goes deeper the execution time increases as it involves in considerable amount of computation time. This makes the set of dominators found at intermediate levels more effective than those which are found at lower levels.

CHAPTER 5

Related work

The field of databases has been a major source of locking protocols as for its necessity for them. Multi-Granularity locks are also designed to serve their purpose to database community. J. N. Gray putforth the idea of locking multiple granularities by multiple transactions. He was the one to propose Intention locks for shared databases. Intention locking is one of the most widely used locking technique in relational databases.

A locking protocol for dynamic databases was proposed by V. K. Chaudhri in 1995 which allows a node to be locked by a operation atmost once. But inorder to ensure that all of the ancestors are locked it does a path-based traversal aimilar to that of Intention locks.

Cherem in 2008, utilized MGL to transform atomic sections into program supporting normal locking constraints. Their methods uses points-to analysis to lock the points-to set of nodes to be locked. Deadlock conditions are avoided by using Intention locks. In Multi-Domlock although we lock multiple nodes, we do not encounter deadlocks as we always lock in the increasing order of the logical intervals.

CHAPTER 6

Conclusion

In this work, we modeled an algorithm which provides multiple locking options for hierarchical data structures. We showed how Multi-DomLock helps in identifying the more alternatives compared to the one provided by DomLock and how it shows better performance in skewed distributions. In future, we would like to develop a heuristic model to evaluate the effective options of dominator locking.

REFERENCES

[1] 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. Article No. 23. New York, NY, USA. 2016. ACM. ISBN: 978-1-4503-4092-2. doi:10.1145/2851141.2851164..

[2] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger *The notions of consistency and predicate locks in a database system..* Commun. ACM, 19(11):624-633, Nov. 1976. ISSN 0001-0782. doi: 10.1145/360363. 360369. URL http://doi.acm.org/10.1145/360363.360369.

[3] S. Cherem, T. Chilimbi, and S. Gulwani. *Inferring locks for atomic sections.* In Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’08, pages 304-315, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860- 2. doi: 10.1145/1375581.1375619. URL http://doi.acm.org/10. 1145/1375581.1375619.

[4] J. N. Gray, R. A. Lorie, and G. R. Putzolu. *Granularity of locks in a shared data base.* In Proceedings of the 1st International Conference on Very Large Data Bases, VLDB ’75, pages 428-451, New York, NY, USA, 1975. ACM. ISBN 978-1-4503-3920-9. doi: 10.1145/1282480.1282513. URL http://doi.acm.org/10.1145/1282480.1282513.

[5] V. K. Chaudhri and V. Hadzilacos. *Safe locking policies for dynamic databases..* In Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS ’95, pages 233-244, New York, NY, USA, 1995. ACM. ISBN 0-89791-730-8. doi: 10.1145/212433.212464. URL http://doi.acm.org/10.1145/212433.212464.