Abstract—In this paper, we propose an algorithm for maintaining a concurrent directed graph (for shared memory architecture) that is concurrently being updated by threads adding/deleting vertices and edges, one such example is shown in the Figure 1. The update methods of the algorithm are deadlock-free while the contains methods are wait-free. To the best of our knowledge, this is the first work to propose a concurrent data structure for an adjacency list representation of the graphs. We extend the lazy list implementation of concurrent set for achieving this.

We believe that there are many applications that can benefit from this concurrent graph structure. An important application that inspired us is Serialization Graph Testing (SGT) in databases and Transactional Memory. Motivated by this application, on this concurrent graph data-structure, we pose the constraint that the graph should be acyclic. We ensure this by checking graph acyclicity whenever we add an edge. To detect the cycle we propose a Wait-free reachability algorithm. We compare the performance of the proposed concurrent data structure with the coarse-grained locking implementation and we achieve significant speedups.

I. INTRODUCTION

Graph is a common data-structure that can model many real world objects & relationships. A graph represents pairwise relationships between objects along with their properties. Due to their usefulness, graphs are being used in various fields like genomics, networks, coding theory, scheduling, computational devices, networks, organization of similar and dissimilar objects, etc. The current trends of research on graphs are on social networks, semantic networks, ontology, protein structure, etc. Generally, these graphs are very large and dynamic in nature. Dynamic graphs are the once which are subjected to a sequence of changes like insertion, deletion of vertices and/or edges [3]. Online social networks (facebook, linkedin, google+, twitter, quora, etc.), are dynamic in nature with the users and the relationships among them changing over time. There are several important problem that can become challenging in such a dynamic setting: finding cycles, graph coloring, minimum spanning tree, shortest path between a pair of vertices, strongly connected components, etc. We have been specifically motivated by the problem of (SGT) scheduler [2, 8] from Databases. A database scheduler (as the name suggests) handles the concurrency control over a set of transactions running concurrently. A transaction is a block of code invoked by a thread/process to access multiple shared memory variables atomically. The scheduler commits a transaction if does not violate correctness (which is serializability); otherwise the transaction is aborted. The traditional solution employed by SGT & STMs to maintain dynamic graphs is to use a single coarse lock to protect the entire graph. Clearly, this implementation can be speeded up by providing finer granularities of synchronization. Each thread which has invoked a transaction should independently be able add/delete vertices/edges to independent parts of the graph.

It can be seen that this problem gets complicated if the graph is huge and multiple threads are concurrently accessing the graph and performing some operations. There are many efficient well-known algorithms for solving these problems in the sequential world. However, there are a very few works in the area of concurrent graphs shared memory setting. There has been a recent and relevant work on the area of concurrent graphs by Kallimanis and Kanellou [6]. Their work also discusses about concurrent graph which supports operations; addition/deletion of vertices/edges and dynamic traversal. But they represent dynamic graph in the form of adjacency matrix with fixed upper-limit on the total number of vertices. Hence, their work can not be used to build SGT scheduler. Moreover, it is not obvious how to extend their work to check for graph acyclicity.

II. CONCURRENT GRAPH DATA-STRUCTURE

A. Overview

The problems addressed in this paper are defined a concurrent directed graph $G = (V, E)$, which is dynamically being...
modified by a fixed set of concurrent threads. In this setting, threads may perform insertion / deletion of vertices or edges to the graph. We also maintain an invariant that the concurrent graph $G$ updated by concurrent threads should be acyclic. This means that the graph should preserve acyclicity at the end of every operation in the generated equivalent sequential history.

Serialization Graph Testing Algorithm which is our motivating example, assumes that all the transactions have unique ids. Once a transaction is deleted it does not back come again into the system. As a result, we assume that all the vertices are assigned unique keys and duplicate vertices are not allowed. We assume that if a vertex id has been removed, it will not be added again to the concurrent graph $G$.

III. MAINTAINING GRAPH ACYCLICITY

In this section, we consider the problem of maintaining an invariant of acyclicity in this concurrent dynamic graph data structure. As described earlier, the objective is to maintain an acyclic conflict graph of transactions for SGT. For a concurrent graph to be acyclic, the graph should maintain the acyclic property at the end of each operation in the equivalent sequential history. To achieve this, we try to ensure that the graph stays acyclic in all the global states. It is easy to see that a cycle can be created only on addition of edge to the graph. We modify the concurrent graph data structure presented in the earlier section to support this acyclic property. The sequential specification of $AddEdge$ is relaxed as follows: after a new directed edge has been added to the graph (in the shared memory), we verify if the resulting graph is acyclic. If it is, we leave the edge. Otherwise we delete the edge from the shared memory. Thus $AddEdge(u, v)$ may fail even if the edge $(u, v)$ was not already part of the graph.

A. Wait-Free Reachability Algorithm working Process

1) Start traversing the adjacency list of $v$ to find $u$ in the concurrent graph, without acquiring locks.
2) Traversed vertices are added to the local ReachSet and the vertex $v$ is marked to be explored.
3) Recursively explore (similar to breadth first traversal) the outgoing edges from the neighbors of $v$ to find $u$.
4) Do this until all the vertices in all the paths from $v$ to $u$ in $G$ have been explored or a cycle has been detected in the graph.

IV. RESULTS

We performed our tests on 10 core Intel Xeon (R) CPU E5-2630 v4 running at 2.02 GHz core frequency. Each core supports 2 hardware threads. Every core’s L1, L2 cache are private to that core; L3 cache (25MB) is shared across all cores of a processors. The tests were performed in a controlled environment, where we were the sole users of the system. The implementation was written in C++ and threading is achieved by using Posix threads.

In the experiments conducted, we start with an empty graph initially. When the program starts, it creates 150 threads and each thread randomly performs a set of operations chosen by a particular workload distribution. Here, the evaluation metric used is the time taken to complete all the operations. We measure speedup obtained against the sequential implementation and present the results for the following workload distributions: (a) Update-dominated: 25% $AddVertex$, 25% $AddEdge$, 10% $RemoveVertex$, 10% $RemoveEdge$, 15% $ContainsEdge$ and 15% $ContainsVertex$ (b) Contains-dominated: 7% $AddVertex$, 7% $AddEdge$, 3% $RemoveVertex$, 3% $RemoveEdge$, 40%
Figure 2: a) Speedup obtained against sequential v/s Number of operations/thread for \textit{Contains-dominated} workload b) \textit{Update-dominated} workload c) Acyclicity Data Structure

\textit{Contains\textit{Vertex} and 40\% Contains\textit{Edge} and the performance results are depicted in Figure 2}. Each data point is obtained after averaging for 5 iterations.

V. Conclusion & Future Direction

In this paper, we have shown how to construct a fully dynamic concurrent graph data structure, which allows threads to concurrently add/delete vertices/edges. The update methods of the algorithm are deadlock-free while the contains methods are wait-free. To the best of our knowledge, this is the first work to propose a concurrent data structure for an adjacency list representation of the graphs. The other known work on concurrent graphs by Kallimanis & Kanellou \cite{6} works on adjacency matrix and assume an upper-bound on the number of vertices while we make no such assumption.

We believe that there are many applications that can benefit from this concurrent graph structure. An important application that inspired us is SGT in databases and Transactional Memory. Motivated by this application, on this concurrent graph data-structure, we pose the constraint that the graph should be acyclic. We ensure this by checking graph acyclicity whenever we add an edge. To detect the cycle we have proposed a \textit{Wait-free} reachability algorithm. We have compared the performance of the concurrent data structure with the coarsely-grained locking implementation and we achieve a significant speedup. For proving the correctness of our algorithm, we have used the linearizability property \cite{5} and illustrated the proof sketch using linearization points and the complete pseudo code and proof sketch is described in the technical report \cite{7}. In the future, we plan to develop a more efficient concurrent graph data structure which can efficiently delete the incoming edges of a vertex. We also plan to extend this concurrent object by lifting the constraint of not allowing vertices to come again with the same key, once they have been deleted. The concurrency in \textit{wait-free} reachability algorithm can be further increased by using two-way searching technique. We also plan to think of other ways of cycle detection by extending ideas of incremental cycle detection algorithms \cite{1} to the concurrent setting.

\begin{thebibliography}{9}


\end{thebibliography}