Non-Blocking Graph Data Structures

Jiewen Hu

Abstract

In this project, I developed a non-blocking graph data structure optimized for parallel computing. The implementation, tested on multiple architectures, focused on directed graphs using various synchronization methods. The performance, assessed through Erdős–Rényi graph simulations, showed significant scalability, especially in high-thread environments.

1 Background

Graphs, with their intricate network of vertices and edges, are fundamental in modeling complex relationships and interactions in various fields, including computational biology, social network analysis, and computer networks. Traditionally, graph computations have been constrained by sequential data structures, which inherently limit the speed and efficiency of processing, particularly in scenarios with large datasets. This limitation becomes acutely apparent in one of my researches of virus evolution in computational biology, where simulations on sequential graphs are not only time-intensive but also lack the capability for effective parallelization. The development of a parallelizable, specifically non-blocking, graph data structure emerges as a vital need in this context. By embracing non-blocking paradigms, we can significantly enhance the performance and scalability of graph computations, paving the way for more rapid and comprehensive analyses in various computational domains.

1.1 Graph

Since undirected graph could be implemented by making a copy of directed graph and reverse the edge directions, this project mainly focus on the implementation of directed graph. An abstract directed graph is given as $G = (V, E)$, where V is the set of vertices and E is the set of directed edges (ordered pair of vertices). Each edge connects an ordered pair of vertices belonging to V.

1.2 Operations

The operations I want to implement including the basic modification and lookup operations, as well as a search function:

1. **AddVertex**: This operation inserts a new vertex into the graph.

2. **RemoveVertex**: This function removes an existing vertex from the graph.

3. **ContainsVertex**: A lookup operation that checks whether a vertex exists in the graph.

4. **AddEdge**: This operation adds a new edge to the graph.

5. **RemoveEdge**: This function removes an existing edge from the graph.

6. **ContainsEdge**: A lookup operation that checks whether an edge exists in the graph.

2 Approach

In order to compare the performance of different synchronization techniques, I implemented four data structure: Sequential baseline, Coarse-grained, Fine-grained, and Lock-free. I will explain the implementation in details in this section.

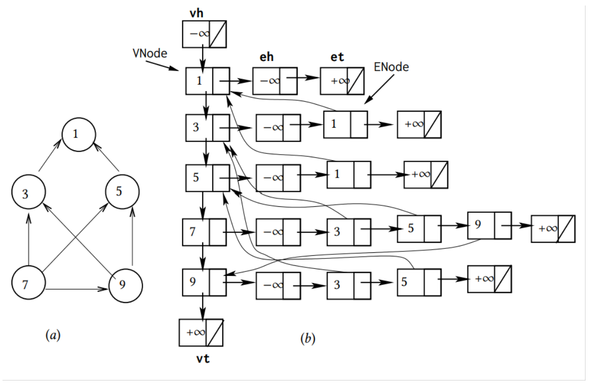


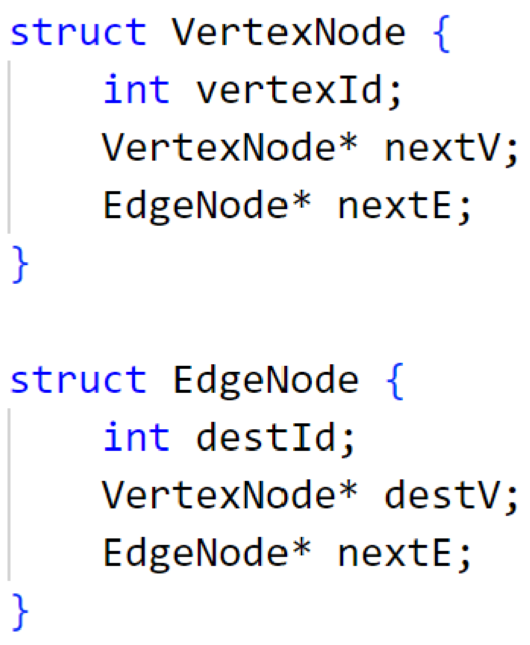
Figure 1: Illustration of nested linked lists for adjacency list

2.1 Data Structure

The graph is structured as an adjacency list, where each vertex in the graph is represented as a node in a primary sorted linked list, VertexList. Each vertex-node in this list then roots a secondary linked list, EdgeList, which contains all the outgoing edges (or edge-nodes) for that vertex. These edge-nodes are also maintained in a sorted order. In addition, the edge-nodes maintain pointers to the corresponding vertex-nodes to enable efficient graph traversals, as shown in Figure 1. The basic data structure used for sequential implementation is provided in Figure 2.

2.2 Sequential

In traditional sequential graph data structures, operations are focused on simplicity and direct manipulation. Key operations include adding or re-



```
struct VertexNode {
    int vertexId;
    VertexNode* nextV;
    EdgeNode* nextE;
}

struct EdgeNode {
    int destId;
    VertexNode* destV;
    EdgeNode* nextE;
}
```

Figure 2: Sequential data structure

moving vertices and edges, and checking for their existence. These tasks are performed by straightforward traversal and modification of the graph's lists or arrays that represent vertices and edges. Additionally, traversal algorithms like Breadth-First Search are implemented using simple iterative or recursive approaches. The overall design emphasizes ease of understanding and implementation, without the complexities of handling concurrent access or modifications.

2.3 coarse-grained

The coarse-grained implementation is largely based on the sequential implementation by adding global mutex to both vertex list and corresponding edge list.

2.4 fine-grained

In the fine-grained implementation of our graph data structure, we employ a 'hand-over-hand' locking strategy, as discussed in our lectures. This approach involves applying locks only to the nodes we are currently modifying and their immediate predecessors, which contain crucial information about these nodes. However, a significant challenge in this approach arises from the interplay between the vertex list and the edge lists.

Particularly, when removing a vertex, we face the complexity of also needing to remove nodes from various edge lists that reference this vertex. This scenario could potentially lead to locking a substantial number of vertex nodes to locate and eliminate the related edges. This situation presents a risk of increased contention and reduced efficiency in our fine-grained locking scheme.

To address this issue, we introduce a compromise by adopting a strategy of checking the existence of both endpoints of an edge during edge-related operations. This approach transforms edge deletion into a lazy operation. When one endpoint of an edge is absent from the graph, such edges are deemed invalid and can be ignored or removed at a later stage. This strategy allows us to treat the vertex list and the edge list as two separate entities, significantly reducing the complexity and lock contention.

Under this modified scheme, vertex nodes are locked only during vertex-specific operations, and edge nodes are locked solely for edge-related tasks. This separation ensures that operations on vertices and edges are more independent, reducing the likelihood of multiple locks and thereby enhancing the overall efficiency and scalability of our fine-grained implementation. It's a balancing act that maintains the integrity and consistency of the graph while optimizing for concurrent access and modifications.

2.4.1 Lock-Free Data Structure Implementation

The intricacies of lock-free data structure implementation necessitate a detailed discussion on the rationale behind each operation within our concurrent graph model.

2.4.2 Locating Vertices and Edges

The functions `locV` and `locE` are instrumental in the lock-free infrastructure of our graph. Their primary function is to determine the precise location within the graph's data structure where a vertex or an edge should be situated. These functions are especially critical in a dynamic environment where multiple threads may be concurrently modifying the graph.

The cornerstone of `locV` and `locE` is the provision of an atomic operation that ensures the necessary conditions are met before any modifications are made. For instance, to add a vertex, `locV` ensures that the vertex does not already exist within the graph. Similarly, when adding an edge with `locE`, the function verifies that while both connecting vertices are present, the edge itself is not.

Illustrated in Figure 3, the `locV` function begins by accepting an input tuple $< v, k >$, where v denotes the starting point for the search and k the vertex index being sought. The function initializes two pointers, `pv` (previous vertex) and `cv` (current vertex), which are vital for the traversal of the linked list, beginning from `v`, the head of the list. During its execution, `locV` traverses the nodes, bypassing any that are marked for deletion—a process signifying logical removal yet physical presence within the list. This step is crucial for maintaining the integrity of the list during concurrent updates and is achieved through the Compare-And-Swap (CAS)

```
101: procedure LocV($v$, $k$)
102:   while (true) do
103:     $pv \leftarrow v$; $cv \leftarrow pv$.vnxt;
104:     while (true) do
105:       $cn \leftarrow cv$.vnxt;
106:       while (isMrkd ($cn$)) $\wedge$($cv.k < k$)) do
107:         if ($\neg CAS$($pv$.vnxt, $cv$, $cv$.vnxt)) then
108:           goto 102;
109:         end if
110:         $cv \leftarrow cn$; $cn \leftarrow$ UnMrkdRf($cv$.vnxt);
111:       end while
112:       if ($cv.k \geq k$) then return $\langle pv, cv \rangle$;
113:       end if
114:       $pv \leftarrow cv$; $cv \leftarrow cn$;
115:     end while
116:   end while
117: end procedure
```

Figure 3: Pseudo code of locating vertex

operation. If the CAS fails due to modifications by other threads, `locV` restarts from the outer loop, ensuring the operation's atomicity.

The traversal proceeds until `locV` identifies a vertex whose key $cv.k$ is not smaller than the sought key k, signaling the location where the vertex resides or should be inserted. The function culminates by returning the tuple $< pv, cv >$, pinpointing the exact location for the sought key and enabling further operations, such as insertion or removal, to be performed accurately and safely within a concurrent setting.

As shown in Figure 4, the `locE` function works similarly but need to check more properties like whether its ends are deleted as described in fine-grained section.

```
118: procedure LocE($v$, $k$)
119:   while (true) do
120:     $pe \leftarrow v$; $ce \leftarrow pe$.enxt;
121:     while (true) do
122:       $cnt \leftarrow ce$.enxt; VNode vn ← ce.ptv;
123:       while (isMrkd ($vn$) $\wedge$ ¬ isMrkd ($cnt$)) do
124:         if (¬CAS($ce$.enxt, $cnt$, MrkdRf ($cnt$))) then
125:           goto Line 119;
126:         end if
127:         if (¬CAS($pe$.enxt, $ce$, $cnt$)) then goto Line 119;
128:         end if
129:         $ce \leftarrow cnt$; $vn \leftarrow ce$.ptv;
130:         $cnt \leftarrow$ UnMrkdRf($ce$.enxt);
131:       end while
132:       while (isMrkd (cnt)) do
133:         v.ecnt.FetchAndAdd (1);
134:         if (¬ CAS($pe$.enxt, $ce$, $cnt$)); then goto 119;
135:         end if
136:         $ce \leftarrow cnt$; $vn \leftarrow ce$.ptv;
137:         $cnt \leftarrow$ UnMrkdRf($ce$.enxt);
138:       end while
139:       if (isMrkd ($vn$)) then  goto Line 123;
140:       end if
141:       if ($ce.l \geq k$) then return $\langle pe, ce \rangle$
142:       end if
143:       $pe \leftarrow ce$; $ce \leftarrow cnt$;
144:     end while
145:   end while
146: end procedure
```

Figure 4: Pseudo code of locating edge

2.4.3 Adding Vertices and Edges

With the aid of the locating functions, the addition of vertices and edges becomes straightforward. The procedure entails moving to the correct position using the locating functions and then attempting to add the vertex or edge using a Compare-And-Swap (CAS) operation to ensure synchronization. If this CAS operation fails due to concurrent modifications, the process must retry from the beginning, invoking the locating function anew to establish the correct position within the dynamically changing data structure. Figure 5 presents the pseudocode for the `AddVertex` and `AddEdge` functions.

```
Operation AddVertex ($k$)
  while (true) do
    $\langle pv, cv \rangle \leftarrow$ locV (vh, $k$);
    if ($cv.k = k$) then return false;
    else
      $nv \leftarrow$ CVNode ($k$); $nv$.vnxt $\leftarrow cv$;
      if (CAS($pv$.vnxt, $cv$, $nv$)) then
        return true;
      end if
    end if
  end while
end Operation
```

```
Operation AddEdge($k$, $l$)
  ⟨ u, v, st ⟩ ← ConVPlus($k$, $l$);
  if (st = false) then
    return "VERTEX NOT PRESENT" ;
  end if
  while (true) do
    if (isMrkd ($u$) $\vee$ isMrkd ($v$)) then
      return "VERTEX NOT PRESENT";
    end if
    $\langle pe, ce \rangle \leftarrow$ locE ($u$.enxt, $l$);
    if ($ce.l = l$) then
      return "EDGE PRESENT";
    end if
    $ne \leftarrow$ CENode ($l$);
    $ne$.enxt $\leftarrow ce$;
    $ne$.ptv $\leftarrow v$;
    if (CAS($pe$.enxt, $ce$, $ne$)) then
      u.ecnt.FetchAndAdd (1);
      return "EDGE ADDED";
    end if
  end while
end Operation
```

Figure 5: Pseudo code of adding operations

2.4.4 Removing Vertices and Edges

The removal operations incorporate a more intricate approach in the lock-free context. A key principle for an efficient lock-free removal is the distinction between making an element invalid, termed "logical removal," and its subsequent elimination from the data structure, known as "physical removal." Logical removal is a swift operation that marks an element as deleted, whereas physical removal, which may involve memory deallocation, tends to be more time-consuming. Direct implementation of physical removal can lead to unauthorized access to the element during its deletion or cause threads to stall within the CAS loop. Therefore, logical removal is executed first, ensuring that other threads can safely interact with the data structure without encountering the deleted element.

Figure 6 illustrates the lock-free implementations of `RemoveVertex` and `RemoveEdge`. In the

`RemoveVertex` operation, `MrkdRf(cn)` signifies that the node cn is marked for logical removal. Once the node is successfully marked, the algorithm proceeds to the physical removal phase. It is important to note that logical removal also establishes exclusivity, preventing other threads from modifying the logically removed element. This ensures that there is no need for additional looping when performing the physical removal step.

2.4.5 Containing Vertices and Edges

The containing operations are direct application of Locating operations, as shown in Figure 7.

2.5 Debugging Methodology

Debugging lock-free algorithms presents unique challenges due to the complexity of concurrent interactions. Despite the theoretical frameworks available in academic literature, practical implementations are prone to subtle bugs that can be difficult to detect and resolve. To thoroughly investigate and rectify issues within my implementation, I have employed a two-pronged approach for assembling debugging datasets:

```
Operation REMOVEVERTEX($k$)
  while (true) do
    $\langle pv, cv \rangle \leftarrow$ LOCV (vh, $k$);
    if ($cv.k \neq k$) then
      return false;
    end if
    $cn \leftarrow cv$.vnxt;
    if ($\neg$ ISMRKD ($cn$)) then
      if (CAS($cv$.vnxt, $cn$, MRKDRF ($cn$))) then
        if (CAS($pv$.vnxt, $cv$, $cn$)) then
          $break$;
        end if
      end if
    end if
  end while
  return true;
end Operation
```

```
Operation REMOVEEDGE($k$, $l$)
  ⟨ u, v, st ⟩ ← CONVPLUS($k$, $l$);
  if (st = false) then
    return "VERTEX NOT PRESENT";
  end if
  while (true) do
    if (ISMRKD ($u$) $\vee$ ISMRKD ($v$)) then
      return "VERTEX NOT PRESENT";
    end if
    $\langle pe, ce \rangle \leftarrow$ LOCE ($u$.enxt, $l$);
    if ($ce.l \neq l$) then
      return "EDGE NOT PRESENT";
    end if
    $cnt \leftarrow ce$.enxt;
    if ($\neg$ ISMRKD ($cnt$)) then
      if (CAS($ce$.enxt, $cnt$, MRKDRF ($cnt$))) then
        $u$.ecnt.$FetchAndAdd$(1);
        if (CAS($pe$.enxt, $ce$, $cnt$)) then $break$;
        end if
      end if
    end if
  end while
  return "EDGE REMOVED";
end Operation
```

Figure 6: Pseudo code of removing operations

1. **Edge Cases:** A suite of over thirty manually crafted edge case scenarios was developed to stress-test the implementation. This suite includes tests on an empty graph to ensure baseline correctness, as well as dummy function tests. Dummy functions artificially extend computational time within a thread, thereby

```
Operation ContainsVertex($k$)
  $cv \leftarrow$ vh.vnxt;
  while ($cv.k < k$) do
    $cv \leftarrow$ UnMrkdRf ($cv$.vnxt);
  end while
  if ($cv.k = k \land \neg$ isMrkd ($cv$)) then
    return true;
  else return false;
  end if
end Operation
```

```
Operation ContainsEdge($k$, $l$)
  $\langle$ u, v, st $\rangle \leftarrow$ ConCPlus($k$, $l$);
  if (st = false) then
    return "VERTEX NOT PRESENT";
  end if
  $ce \leftarrow u$.enxt;
  while ($ce.l < l$) do
    $ce \leftarrow$ UnMrkdRf ($ce$.enxt);
  end while
  if ($ce.l = l \land \neg$ isMrkd ($u$) $\land \neg$ isMrkd ($v$) $\land \neg$ isMrkd ($ce$)); then
    return "EDGE FOUND" ;
  else
    return "VERTEX OR EDGE NOT PRESENT";
  end if
end Operation
```

Figure 7: Pseudo code of containing operations

increasing the likelihood of concurrent conflicts and revealing potential synchronization issues.

2. **Large Randomized Cases:** Given the project's focus on large-scale graph processing, manual test case generation is impractical. Therefore, I have utilized automated tools to generate a diverse set of randomized datasets. These datasets encompass a variety of operation combinations and are designed to simulate real-world usage patterns, allowing me to identify and address issues that emerge under complex operational conditions.

This comprehensive testing strategy ensures a robust examination of the lock-free graph implementation, facilitating the identification of concurrency-related bugs that might not surface under conventional testing methodologies.

3 Result

3.1 Experiment

Experiments were conducted on a personal computer equipped with a 13th Generation Intel® Core™ i9 Processor operating at 4.10 GHz, with a total of 20 threads available. The test environment was set up using randomly generated Erdős–Rényi graphs with a connection probability of 0.25 and an initial size of 1000 nodes.

Three distinct sets of experiments were performed, each measuring the number of graph operations completed within a span of 20 seconds, varying the number of threads utilized. The operations were randomly selected from three different pools, each with a specified frequency for the operations: {AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge, ContainsEdge$\}$. The frequency distributions for the operations were as follows: $\{0.25, 0.25, 0.10, 0.10, 0.15, 0.15\}$ for a mix of operations; $\{0.3, 0.3, 0.2, 0.2, 0, 0\}$ for an emphasis on modification operations; and $\{0, 0, 0, 0, 0.5, 0.5\}$ for an emphasis on lookup operations only. These configurations were intended to simulate scenarios of mixed operations, modification-centric operations, and lookup-centric operations, respectively.

3.2 Main Results

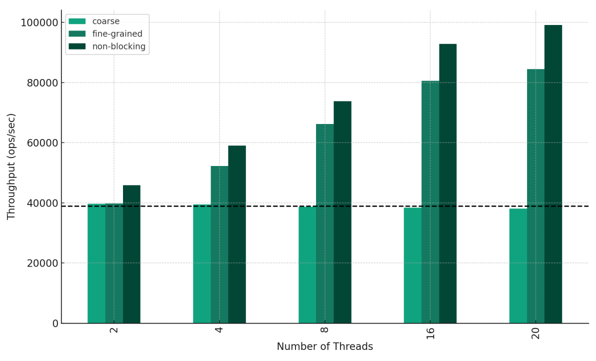
The performance evaluation of the different graph data structure implementations is presented in Figure 6, where subfigures (a), (c), and (e) depict the throughput for each implementation and subfigures (b), (d), and (f) display the corresponding speedup relative to the sequential baseline, indicated by the dotted lines.

Our observations reveal that the non-blocking algorithm demonstrates significant scalability with respect to the number of threads: both throughput and speedup values increase substantially as more threads are introduced. Conversely, the coarse-grained lock-based version exhibits a decline in performance with an increasing thread count. This performance degradation is symptomatic of the inherent limitation of coarse-grained locks—contention for the global lock escalates with more threads, leading to a bottleneck. This contention is so detrimental that, once concurrency is introduced, the coarse-grained lock-based version underperforms in comparison to the sequential implementation.

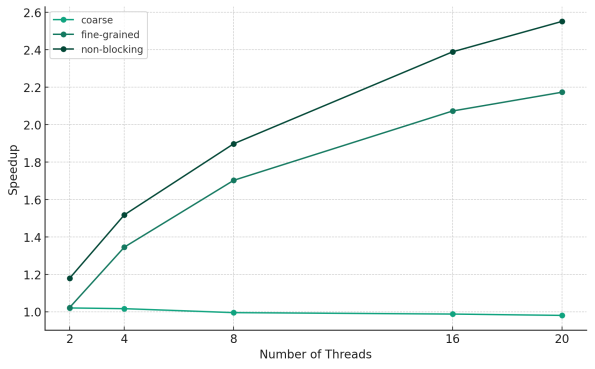
Interestingly, the lookup-centric operations do not parallelize well, with the parallelized data structures showing inferior performance to the sequential baseline, even with 20 threads. Specifically, the throughput of the non-blocking implementation with a single thread is markedly lower than the baseline (681,667 ops/s compared to 5,474,498 ops/s). This discrepancy is likely attributable to the inherently low cost of sequential lookup operations. Despite being lock-free, the overhead associated with Compare-And-Swap (CAS) operations is non-negligible and impacts performance adversely.

3.3 Analysis

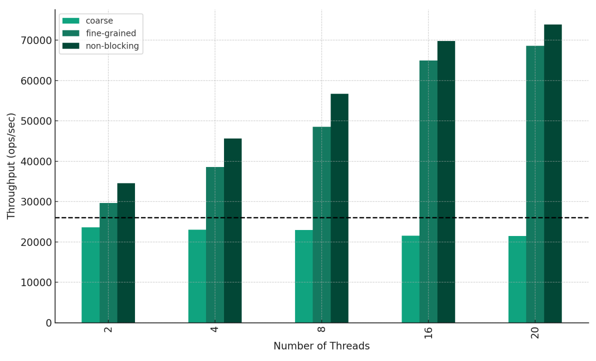
The performance evaluation of the lock-free graph data structure provides insights into the factors that limited the speedup achieved by the various implementations. To understand these limitations, we consider several aspects of parallel computing.



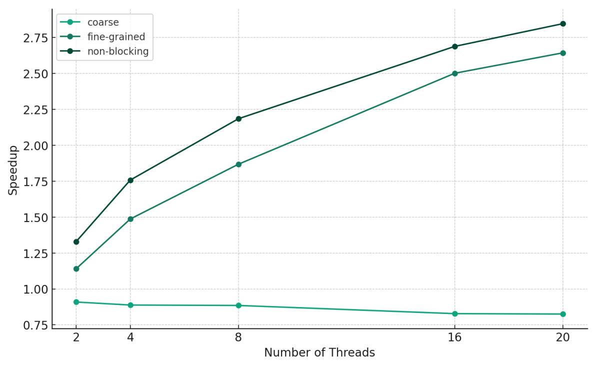
(a) Throughout for mixed operations



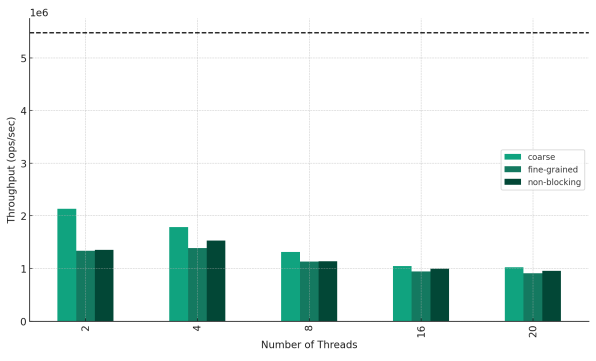
(b) Speedup for mixed operations



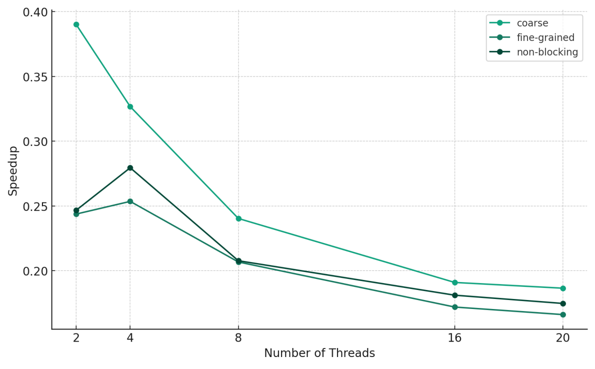
(c) Throughout for modification-centric operations



(d) Speedup for modification-centric operations



(e) Throughout for lookup-centric operations



(f) Speedup for lookup-centric operations

Figure 8: Experimental Results

3.3.1 Cache Analysis

A systematic examination of cache utilization was conducted to identify potential bottlenecks in the non-blocking graph structure. Performance counters were accessed using the `perf` tool, with results summarized in Table 1 for mixed operations and Table 2 for lookup-centric operations.

The data from Table 1 reveal a significant increase in cache miss rates correlating with the rise in thread count. This trend suggests an escalation in cache invalidations as concurrent threads modify shared data, leading to an increase in cache write-backs. An interesting observation from Table 2 is the disproportionate growth in cache references during lookup-centric operations compared to mixed operations. This could indicate a rise in the number of retries following failed Compare-And-Swap (CAS) operations, contributing to an increased computational burden not proportionally distributed among threads. Such inefficiency in workload distribution likely impedes the attainment of ideal speedup, particularly at higher thread counts, and could account for the observed suboptimal performance in lookup-centric tasks.

3.3.2 Parallelism and Dependencies

The lock-free implementation exhibited significant scalability with an increasing number of threads, suggesting that the algorithm successfully exploited the available parallelism to some extent. However, the increase in throughput was not linear, indicating potential dependencies that could have prevented full parallelization. For instance, while the non-blocking algorithm is designed to minimize dependencies, the Compare-And-Swap operations still introduce some level of inherent serialization, which might have limited the speedup. In addition, it is noticeable that the throughout value in lookup-centric operations experiment is extremly high, which could reach the bandwidth for data communication between caches. The experiment on Bridge Machine, which shows a quick convergence, also supports this idea.

3.3.3 Communication and Synchronization Overhead

Synchronization overhead, particularly in the form of CAS operations in the non-blocking implementation, is a likely contributor to the limited speedup. CAS operations, although lock-free, incur a cost that becomes more pronounced as the number of threads increases, as observed in the lower throughput when compared to the sequential baseline. This overhead is particularly evident in the case of lookup-centric operations, which are inherently fast in a sequential context but become burdened by the additional synchronization mechanisms required for concurrent execution.

3.3.4 Choice of Machine Target

The choice to implement the algorithm on a CPU with a 13th Generation Intel® Core™ i9 Processor was based on the assumption that a multicore CPU environment would be well-suited to a lock-free, highly concurrent algorithm. While a GPU might offer a higher degree of parallelism, the branching and irregular data access patterns of graph algorithms make CPUs a more suitable choice, particularly given the overheads associated with GPU programming and data transfer between the CPU and GPU memory spaces.

4 Conclusion

The experimental results demonstrate the scalability potential of non-blocking graph data structures, particularly when dealing with a high number of threads. The non-blocking approach showcased a significant improvement in throughput and speedup compared to other synchronization methods. However, the performance gains were not linear, indicating the presence of inherent limitations, such as synchronization overhead and cache-related inefficiencies. Our cache analysis revealed a considerable increase in cache misses with higher thread counts, suggesting synchronization and contention issues as the primary culprits limiting speedup. Despite these challenges, the CPU-based implementation proved to be a sound choice for our lock-free algorithm, given the irregular data access patterns of graph algorithms and the overheads associated with GPU programming. Future work should aim to minimize synchronization overhead, optimize cache utilization, and explore the potential of fine-grained optimizations to enhance the performance of lock-free graph data structures further. The current study lays the groundwork for the continued evolution of parallel graph processing techniques, paving the way for faster and more efficient analyses in computational domains that rely heavily on graph-based data

References

1. Chatterjee, B., Peri, S., Sa, M., & Singhal, N. (2018). A Simple and Practical Concurrent Nonblocking Unbounded Graph with Lin-

Number of threads	Cache Miss	Cache Reference	Cache Miss rate
1	457,563	3,557,408,371	0.01%
2	1,344,829	7,287,514,109	0.02%
4	15,943,978	15,228,431,708	0.11%
8	214,703,410	29,619,492,476	0.73%

Table 1: Cache information for mix operations

Number of threads	Cache Miss	Cache Reference	Cache Miss rate
1	5,139,618	43,150,133	11.91%
2	9,140,947	502,141,398	1.82%
4	13,158,330	1,979,104,546	0.67%
8	13,781,365	4,882,690,070	0.28%

Table 2: Cache information for lookup-centric operations

earizable Reachability Queries. arXiv preprint arXiv:1809.00896.

2. Peri, S., Reddy, C. K., & Sa, M. (2019). An Efficient Practical Concurrent Wait-Free Unbounded Graph. IEEE Xplore. https://ieeexplore.ieee.org/document/8855669

3. Hong, B. (2008). A Lock-free Multi-threaded Algorithm for the Maximum Flow Problem. Drexel University, Philadelphia, PA 19104. ISBN 978-1-4244-1694-3. ©2008 IEEE.