MULTI-CORE PARALLEL GRAPH ALGORITHMS

MULTI-CORE PARALLEL GRAPH ALGORITHMS

BY
BIN GUO, M.Sc., B.Eng.

A Thesis
submitted to the Department of Computing and Software
and the School of Graduate Studies
of McMaster University
in partial fulfilment of the requirements
for the degree of
Doctor of Philosophy

Doctor of Philosophy (2022) McMaster University
(Department of Computing and Software) Hamilton, Ontario, Canada

TITLE:	Multi-Core Parallel Graph Algorithms
AUTHOR:	Bin Guo M.Sc., (Computer Science) Winnipeg University B.Eng., (Packaging Engineering) North University of China
SUPERVISOR:	Dr. Emil Sekerinski
NUMBER OF PAGES:	xiv, 140

Lay Abstract

Graphs are important data structures to model real networks like social networks, communication networks, hyperlink networks, and model-checking networks. These network graphs are becoming larger and larger. Analyzing large data graphs requires efficient parallel algorithms executed on multicore machines. In this thesis, we focus on two graph problems, graph trimming and core maintenance. The graph trimming is to remove the vertices without outgoing edges, which may repeatedly cause other vertices to be removed. For each vertex in the graph, the core number is a parameter to indicate the density; the core maintenance is to maintain the core numbers of vertices when edges are inserted or removed dynamically, without recalculating all core numbers again. We evaluate our methods on a 16-core or 64-core machine over a variety of real and synthetic graphs. The experiments show that our parallel algorithms are much faster compared with existing ones.

Abstract

Large sizes of real-world data graphs, such as social networks, communication networks, hyperlink networks, and model-checking networks, call for fast and scalable analytic algorithms. The shared-memory multicore machine is a prevalent parallel computation model that can handle such volumes of data. Unfortunately, many graph algorithms do not take full advantage of such a parallel model. This thesis focuses on the parallelism of two graph problems, graph trimming and core maintenance. Graph trimming is to prune the vertices without outgoing edges; core maintenance is to maintain the core numbers of vertices when inserting or removing edges, where the core number of a vertex can be a parameter of density in the graph. The goal of this thesis is to develop fast, provable, and scalable parallel graph algorithms that perform on shared-memory multicore machines. Toward this goal, we first discuss the sequential algorithms and then propose corresponding parallel algorithms. The thesis adopts a three-pronged approach of studying parallel graph algorithms from the algorithm design, correctness proof, and performance analysis. Our experiments on multicore machines show significant speedups over various real and synthetic graphs.

To my family.

Acknowledgements

There are many people I would like to thank. Without them, my thesis would not have been possible.

First and foremost, I thank my supervisor Emil Sekerinski for giving me guidance and inspiration during my Ph.D. studies. He introduced me to parallel algorithms and taught me the knowledge necessary for writing this thesis. He also gave me a lot of useful career advice. I am very grateful he spent hours with me every week answering all of my questions.

I thank the rest of my thesis committee members, Ryszard Janicki, Ned Nedialkov, and Richard Paige, for providing me with useful feedback to improve this thesis. I also thank Richard Paige for giving me advice on job applications.

During my Ph.D. studies, I thank the opportunity to work as a teaching assistant for George Karakostas and my supervisor Emil Sekerinski.

I thank my fellow students, Fatemeh Kazemi Vanhari, Wahalatantrige Senanayaka, Tianyu Zhou, Kefu Zhu, Jason Nagy, Shucai Yao, and Spencer Park.

I am especially thankful to my wife, Yan Chen. She stood by my side during all of my ups and downs, and her constant support and encouragement enabled me to finish writing this thesis. Finally, I am grateful to my parents for always motivating me to pursue my dreams. Without them, I would not have come this far.

Contents

List of Figures

List of Tables

Notation and Abbreviations

Notation

Graph Trimming

$G = (V, E)$	a graph with n vertices and m edges
$G^T = (V, E^T)$	a transposed graph of the directed graph $G = (V, E)$
$u(G).deg_{in}$	the in-degree of u in G
$u(G).deg_{out}$	the out-degree of u in G
$u(G).deg$	the degree of u in G
$u(G).post$	the successors of u in G
$u(G).pre$	the predecessor of u in G
$u(G).S$	the supporting set used by AC-6-Trimming
$u(G).status$	the status (`LIVE` or `DEAD`) of u in G
Deg_{in}	the maximal in-degree among all vertices in G
Deg_{out}	the maximal out-degree among all vertices in G
α	the number of peeling steps in G
Q	a propagation queue
$\mathcal{P}$	the total number of workers
Q_p	a private queue set used by workers p
$\|Q_p\|$	the upper-bound size of waiting sets among $\mathcal{P}$ workers

Core Maintenance

$\vec{G} = (V, \vec{E})$	an constructed DAG by the k-order for edge insertion core maintenance
$u \mapsto v \in \vec{E}(\vec{G})$	a directed edge in an constructed graph
$\mathbb{O} = \mathbb{O}_0\mathbb{O}_1 \dots \mathbb{O}_k$	a sequence indicates the k-order $\preceq$
$u(\vec{G}).d^*_{in}$	the remaining in-degree of u
$u(\vec{G}).d^+_{out}$	the candidate out-degree of u
$u(G).mcd$	the max-core degree of u
$u(G).core$	the core number of u
V^*	the candidate set
V^+	the searching set
$\Delta G = (V, \Delta E)$	an inserted graph to G

Abbreviations

DAG	Directed Acyclic Graph
SCC	Strongly Connected Component
BFS	Breadth-First Search
DFS	Depth-First Search
FW-BW	Forward-Backward Algorithm
CSP	Constraint Satisfaction Problem
AC	Arc-Consistency
OM	Order Maintenance
CSR	Compressed Sparse Row

Declaration of Academic Achievement

1. **Bin Guo**, Emil Sekerinski. Efficient parallel graph trimming by arc-consistency. The Journal of Supercomputing 78, 15269–1531 (2022). 45 pages.

2. **Bin Guo**, Jason Nagy, and Emil Sekerinski. Universal Design of Interactive Mathematical Notebooks on Programming. SIGCSE 2022, V. 2, pages 1132 (1-page poster).

3. **Guo, Bin**, Sekerinski, Emil. New Parallel Order Maintenance Data Structure. arXiv preprint arXiv:2208.07800 (2022). 13 pages.

4. **Guo, Bin**, Sekerinski, Emil. Simplified Algorithms for Order-Based Core Maintenance. arXiv e-prints: arXiv-2201 (2022). 11 pages.

5. **Guo, Bin**, Sekerinski, Emil. Parallel Order-Based Core Maintenance in Dynamic Graphs. arXiv preprint arXiv:2210.14290v1 (2022). 15 pages.

Part I

Introduction and Preliminaries

Chapter 1

Introduction

Graphs are fundamental data structures in computer science. They have been studied and analyzed for hundreds of years beginning with the famous Königsberg Bridge problem studied by Euler in 1736. Essentially, graphs are mathematical representations of relationships between objects such as individuals, knowledge, and positions. In the graph, each vertex represents an object and each edge represents some relationship between a pair of objects. Graphs are used to model a real system in numerous applications, like social networks (Takac and Zabovsky, 2012), graph pattern matching (Chen et al., 2019), communication networks (Kumar et al., 2010), knowledge graphs (Xiaoping et al., 2021), and model verification (Hojati et al., 1993), data is organized into graphs with vertices for objects and edges for their relationships.

The data graphs can be *static graphs*, which can not be changed after generated. However, in many real-world applications, such as determining the influence of individuals in spreading epidemics in dynamic complex networks (Miorandi and De Pellegrini, 2010) and tracking the actual spreading dynamics in dynamic social media networks (Pei et al., 2014), the data graphs continuously change over time. The changes correspond to the insertion and deletion of edges. That means each edge has a time stamp to indicate the time of updating. Graphs of this kind are called *dynamic graphs*.

Since many real-world applications can be modeled as graphs, graph analytics has attracted much attention from both research and industry communities. Many algorithms are proposed to analyze large data graphs, including graph trimming, Strong Connected Component (SCC) decomposition, k-core decomposition, k-truss decomposition, etc. One big issue is that data graphs are growing rapidly in today's data-driven world. For example, 2.9 billion people use Facebook each day; the 2016 release of the DBpedia data set already has 6.6 million entities (vertices) and 13 billion pieces of information (edges). Reducing the running time of programs lowers overall costs. For example,

the rental costs of machines on Amazon EC21[1] is proportional to the usage time. In addition, reducing the time-to-completion of tasks has been shown to increase worker productivity as well as end-user experience.

The Parallel Model. One naive method to handle such large data graphs is to increase the clock speeds of single-core machines. Moore's law states that the transistor density doubles approximately every 18 months, and along with Dennard scaling, which states that transistor power density is constant (Moore, 1998). This has historically corresponded to increases in clock speeds of single-core machines of roughly 30% per year since the mid-1970s. However, since around the mid-2000s, Dennard scaling no longer continued to hold due to physical limitations of the hardware. As a result, hardware vendors have turned to developing processors with multiple cores for delivering improved performance. In other words, processor frequency is no longer increasing due to the power wall and single-thread performance is no longer increasing as the benefits of caching, pipelining, etc. are maxed out. However, the number of transistors per processor is still increasing–linearly on a logarithmic scale, i.e. exponentially, doubling every 18 months as predicted by Gordon Moore. That is why multicore machines have become prevalent in recent years. These machines are referred to as *shared-memory multicore* machines for the different cores access to a shared global memory (Shun, 2017a). We analyze parallel algorithms in the *work-depth* model (Cormen et al., 2009; Shun, 2017b), where the *work*, denoted as $\mathcal{W}$, is the total number of operations that are used by the algorithm and the *depth*, denoted as $\mathcal{D}$, is the longest length of sequential operations (JéJé, 1992). The expected running time is $\mathcal{O}(\mathcal{W}/\mathcal{P} + \mathcal{D})$ when using $\mathcal{P}$ *workers*. Here, a worker is a working process corresponding to a physical core for a multicore processor. The formal definition of the above model is provided in Section 2.2.

Our Goal. Due to the prevalence of shared-memory multicore machines and the rapidly increased data graphs, it is urgent to parallelize graph analytic algorithms. Normally, we can not trivially transform an algorithm from a sequential version to an efficient parallel version. There are four challenges in finding efficient parallel algorithms for the share-memory multicore model. First, compared with the existing work, we desire new parallel or concurrent algorithms with improved work, depth, and space complexities. Second, parallel or concurrent algorithms are hard to argue for correctness, and testing is not reliable to detect all errors (Andrews, 1991; Herlihy et al., 2020). This is because the concurrent execution leads to nondeterministic interleaving of steps and may cause *data races*, e.g. multiple working processes access the

[1] `http://aws.amazon.com/ec2/pricing/`

same memory locations for writing. Third, these parallel or concurrent algorithms should minimize synchronization overhead, requiring a combination of locks, atomic primitives, software transactional memory, etc. Forth, the parallel strategies should achieve work-load balance on multiple cores for implementation. Our research focuses on parallelizing various graph algorithms on multicore architecture. The goal is not just the theoretical improvements, e.g. devising new parallel algorithms and also proving the correctness, but the experimental studies on multicore machines, e.g. cache-friendly algorithms always have good performance since the cache hit rate is high. The goal of this thesis is the following:

We seek to develop fast, provable, and scalable parallel graph algorithms that perform on shared-memory multicore machines.

In this thesis, we take *two-steps* toward this goal. First, we propose a fast sequential algorithm. Second, we parallelize such sequential algorithms on shared-memory multicore machines. The thesis adopts a three-pronged approach of studying parallel graph algorithms from the algorithm design, correctness proof, and performance analysis.

1.1 Studied Problems

In this thesis, we study two graph problems, *Graph Trimming* and *Core Maintenance*. For Core Maintenance specifically, we study the problem of *Order Maintenance*.

Graph Trimming. Given a direct graph $G = (V, E)$, graph trimming is about removing vertices without outgoing edges. Removing such unqualified vertices may continuously cause other vertices $u \in V$ without outgoing edges to be trimmed. The trimming process will terminate when no vertices can be trimmed.

The large size of data graphs motivates graph trimming approaches, such as cycle detection (Lowe, 2016), k-core decomposition (Batagelj and Zaversnik, 2003), and in particular graph decomposition (Hong et al., 2013). For instance, for the communication network *wiki-talk* (Kumar et al., 2010) with 2.4 million vertices, surprisingly 94.5% of the vertices can be trimmed, which greatly reduces the graph size for subsequent processing.

Order Maintenance. The *Order-Maintenance* (OM) data structure (Dietz and Sleator, 1987; Bender et al., 2002; Utterback et al., 2016) maintains a total order of unique items in an order list, denoted as $\mathbb{O}$, by three operations including inserting, deleting, and comparing the order of two items. The naive

idea is to use a balanced binary search tree (Cormen et al., 2022). All three operations can be performed in $O(\log N)$ time, where N is the total number of items in $\mathbb{O}$. In (Dietz and Sleator, 1987; Bender et al., 2002), by using labels to indicate the order, all three operations are optimized to $O(1)$ amortized running time.

There are many applications that require manipulating an ordered list of items. In (Marchetti-Spaccamela et al., 1996; Haeupler et al., 2012), given a directed graph, a *topological order* of all vertices are maintained, where u precedes v in topological order for all edges (u, v) in the graph. In (Zhang et al., 2017; Guo and Sekerinski, 2022c), given a undirected graph, a *k-order* of all vertices are used for core maintenance, where u is precedes v in the peeling steps of core decomposition (Batagelj and Zaversnik, 2003; Cheng et al., 2011; Khaouid et al., 2015; Montresor et al., 2012; Wen et al., 2016) for all edges (u, v) in the graph. Similarly, such k-order can be used for truss maintenance (Zhang and Yu, 2019a). Also, ordered sets are widely used in Unified Modeling Language (UML) Specification (Martin et al., 2003), e.g., a display screen (an OS's representation) has a set of windows, but furthermore, the set is ordered, so do the ordered bag and sequence.

In this thesis, we apply the OM data structure to maintain the order of all vertices in a graph, which can be used in the core maintenance problem to improve its performance.

Core Maintenance. Given an undirected graph $G = (V, E)$, the *k-core* is the maximal subgraph of G where each vertex has a degree at least k. The *core number* of each vertex $u \in V$ is the largest k such that u is contained in the k-core of G (Batagelj and Zaversnik, 2003; Kong et al., 2019). The *core decomposition* of G is to compute the core number for each $u \in V$. Additionally, the *k-shell* is the set of vertices that are part of the k-core but not part of the $(k + 1)$-core. In other words, a k-shell includes all vertices that have core numbers equal to k. The k-core is to find the dense subgraphs from data graphs. Such a dense part of a data graph always has special properties in real networks.

The core decomposition is incredibly useful in static graphs (Batagelj and Zaversnik, 2003; Cheng et al., 2011; Khaouid et al., 2015; Montresor et al., 2012; Wen et al., 2016) due to its linear running time (Batagelj and Zaversnik, 2003). In (Kong et al., 2019), Kong et al. summarize a large number of applications in biology, social networks, community detection, ecology, information spreading, etc. Especially in (Burleson-Lesser et al., 2020), Lesser et al. investigate the k-core robustness in ecological and financial networks. In a survey (Malliaros et al., 2020), Malliaros et al. summarize the main research work related to k-core decomposition from 1968 to 2019. In static graphs, the core decomposition has been extensively studied (Batagelj and Zaversnik,

2003; Cheng et al., 2011; Khaouid et al., 2015; Montresor et al., 2012; Wen et al., 2016).

However, in many real-world applications, such as determining the influence of individuals in spreading epidemics in dynamic complex networks (Miorandi and De Pellegrini, 2010) and tracking the actual spreading dynamics in dynamic social media networks (Pei et al., 2014), the data graphs are dynamic graph and continuously change over time. The changes correspond to the insertion and deletion of edges, which may have an impact on the core numbers of some vertices in the graph. When each time an edge is inserted or removed, it is time-consuming to recalculate the core numbers of all vertices by traversing the whole graphs. A better approach is only to update the core number of affected vertices. The problem of maintaining the core numbers for dynamic graphs is called *core maintenance* (Sarıyüce et al., 2016; Zhang et al., 2017; Wu et al., 2015; Saríyüce et al., 2013).

There are two main proposed algorithms Traversal (Sarıyüce et al., 2016) and Order (Zhang et al., 2017). For edge insertion, by efficiently maintaining the order of all vertices in a graph, the Order algorithm has a significantly smaller searching range and thus has a better performance compared with the Traversal algorithm.

1.1.1 Application Examples

Ecological Networks. As an example, in Figure 1.1, we show an ecological network where the vertices are plants and pollinators and the edges are plant-pollinator interactions (Burleson-Lesser et al., 2020). After k-core decomposition, there are 1-shell to 5-shell. and the maximum k-core is 5-core. The maximum k-core is 5-core, so we find the densest subgraph that is 5-core. For different k-core, a larger k provides more stability and thus a more robust network. Here is the quotation for this ecological network analytics after k-core decomposition in (Burleson-Lesser et al., 2020):

> "For ecological networks, the extinction of species in k-shells[2] increasingly close to the tipping point changes the mutualistic structure of the network. As more and more species are removed and the outermost shells collapse, species closer and closer to the core of the network become commensalists (which only receive a benefit from other species) rather than symbionts (which both give and receive a benefit). At some point, the core species are providing too much for the rest of the network and not receiving enough of a benefit in return; as this continues and there cease to be any

[2]The *k-shell* is the set of vertices that are part of the k-core but not part of the $(k+1)$-core.

species providing benefits to the others, the network is rendered unsustainable and totally collapses."

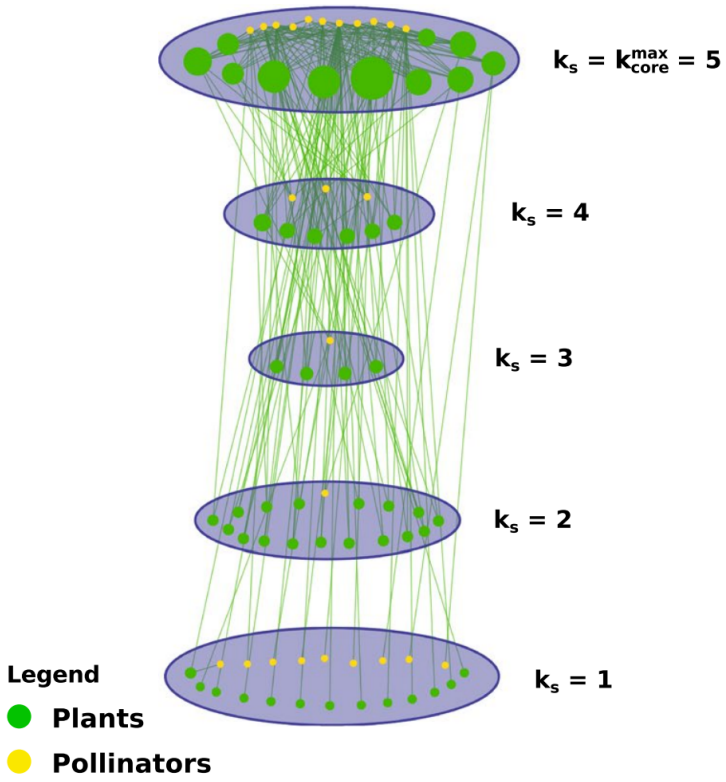


Figure 1.1: The k-core decomposition for an ecological network showing the interaction between plants and pollinators, where the k_s is the k-shell and k_{core}^{max} is the maximum k-core(Burleson-Lesser et al., 2020).

Social Networks. As an example, In Figure 1.2, we show the influential spreaders by tracking the actual spreading dynamics in social networks, including Live Journal, APS Journals, Facebook, and Twitter (Pei et al., 2014; Kong et al., 2019). Pei et al. (Pei et al., 2014) found that the widely used Degree and *PageRank* (an algorithm used by Google Search to measure the importance of website pages) could not rank the impact of users, and the spreading is larger for vertices with higher k-shell. They discover that the best communicators have been on k-core on different social platforms, such as Facebook and Twitter. In Figure 1.2, we observe that for a vertex a larger k-shell and larger in-degree (or PageRank) always indicates higher influence over many social networks.

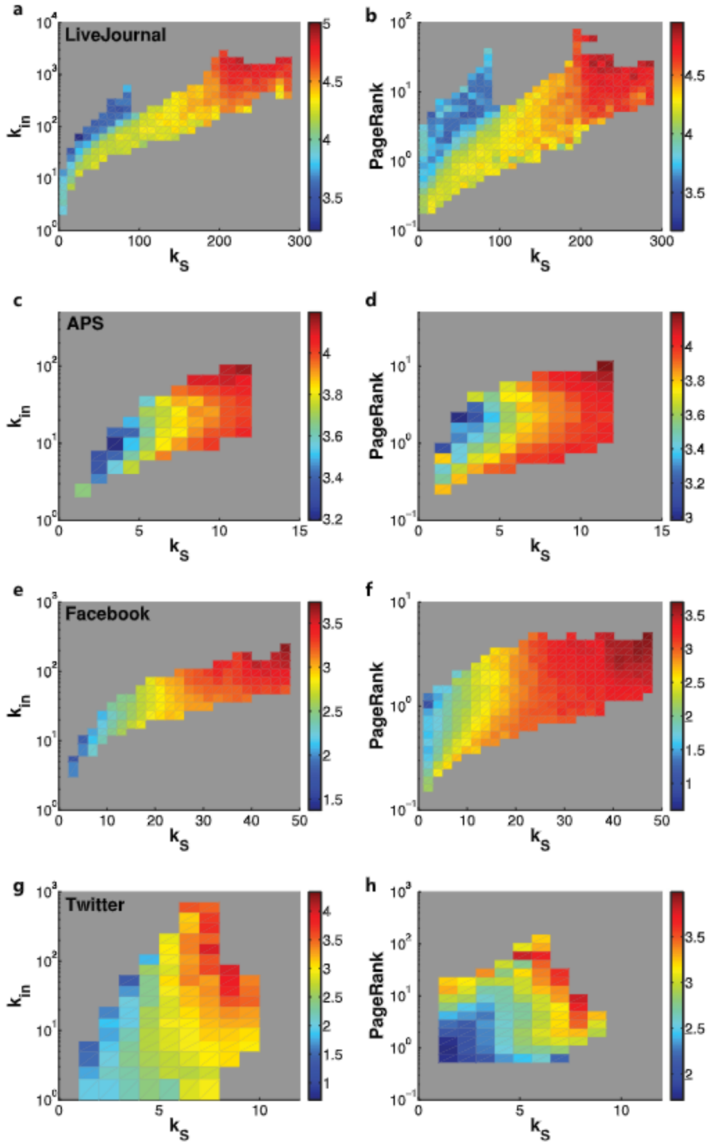


Figure 1.2: The k-shell index can predict the average influence of spreading, where k_s is the k-shell for the x-axis and the k_{in} is the in-degree of a vertex for the y-axis (Pei et al., 2014).

1.2 Summary of Contributions

In this section, we give a high-level summary of the contribution. In Parts II and IV, we provide the technical components. The main contributions are summarized as follows:

- In Part II, Chapter 3 discusses three graph trimming algorithms. Inspired by three Arc-Consistency algorithms, AC-3, AC-4, and AC-6, we classify two existing graph trimming algorithms as AC-3-based and AC-4-based trimming algorithms. We also propose an AC-6-based trimming algorithm. Furthermore, we parallelize these three trimming algorithms.

- Part IV discusses the core maintenance algorithms. First, Chapter 4 proposes a parallel OM data structure, which supports inserting, removing, and comparing the order of two items in parallel. Second, based on the sequential OM data structure, we can efficiently maintain the order of vertices in a graph and then propose a simplified order-based core maintenance algorithm in Chapter 5, which has an improved running time compared with the order-based algorithm. Finally, based on the work of Chapter 4 and Chapter 5, we propose a parallel core maintenance algorithm in Chapter 6.

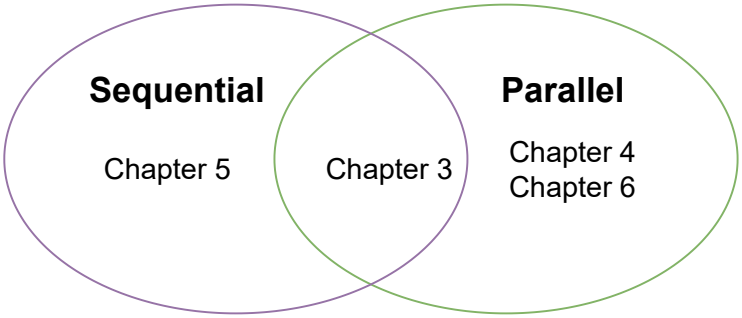


Figure 1.3: A pictorial representation of each chapter for the sequential and parallel algorithms.

For each part, we first discuss the sequential graph algorithm and parallelize them. Figure 1.3 illustrates each chapter focuses on sequential or parallel algorithms.

1.3 Research Work Presented in This Thesis

This thesis contains the following research work (in the order they are presented in this thesis):

- (Guo and Sekerinski, 2022a) Bin Guo, Emil Sekerinski. Efficient Parallel Graph Trimming by Arc-Consistency. (Chapter 3)

- (Guo and Sekerinski, 2022b) Bin Guo, Emil Sekerinski. New Parallel Order Maintenance Data Structure. (Chapter 4)

- (Guo and Sekerinski, 2022c) Bin Guo, Emil Sekerinski. Simplified Algorithms for Order-Based Core Maintenance. (Chapter 5)
- Bin Guo, Emil Sekerinski. Parallel Order-Based Core Maintenance in Dynamic Graphs. (Chapter 6)

Chapter 2

Preliminaries and Notation

This chapter presents the definitions and notation that will be used throughout the thesis. Individual chapters have additional definitions and notations that are specific to the chapter.

2.1 Graph Definitions and Notations

In this thesis, we consider simple, finite graphs with no self-loops or multi-edges. Given a directed graph $G = (V, E)$, let $n = |V|$ and $m = |E|$ be the numbers of vertices and edges, respectively. A vertex v in graph G is also denoted as $v(G)$. Of course, for a directed graph, $(v, w) \in E$ does not imply that $(w, v) \in E$. The *post* of vertex v in G is the set of all the successors (outgoing edges) of v, defined by $v.post = \{w \mid (v, w) \in E\}$; when the context is clear, we use $v.post$ instead of $v(G).post$. The *pre* of vertex v is the set of all the predecessors (ingoing edges) of v, defined by $v.pre = \{w \mid (w, v) \in E\}$. For each vertex $v \in V$, its *out-degree* is the number of successors $|v.post|$ and its *in-degree* is the number of predecessors $|v.pre|$. We define the set of neighbors of a vertex $u \in V$ as $u.adj$, formally $u.adj = \{v \in V : (u, v) \in E\}$. We denote the degree of u in G as $u.deg = |u.adj|$.

A transposed graph $G^T = (V, E^T)$ is equivalent to the graph $G = (V, E)$ with all its edges reversed, $E^T = \{(w, v) \mid (v, w) \in E\}$. It is easy to see that $v(G).post = v(G^T).pre$ and $v(G).pre = v(G^T).post$ for each $v \in V$. A transposed graph G^T can be generated in order to efficiently obtain $v(G).pre$ without traversing the whole original graph G.

For an undirected graph, an edge $(v, w) \in E$ is equivalent to an edge $(w, v) \in E$. Therefore, for a vertex $v \in V$, it only has neighbors $v.adj$, but no successors $v.post$ and predecessors $v.pre$.

We define a graph G' to be a subgraph of G, denoted as $G' \subseteq G$, if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. Given a subset $V' \subseteq V$, the subgraph induced by V', denoted as $G(V')$, is defined as $G(V') = (V', E')$ where $E' = \{(u, v) \in E :$

$u, v \in V'\}$.

In this thesis, we use the above G as the definition of graphs. Other special cases of graphs, e.g. bipartite graphs in which a set of vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent, are still covered by the definition G.

2.2 Shared-Memory Work-Depth Model

We analyze our parallel algorithms in the *work-depth* model (Cormen et al., 2009; Shun, 2017b). An algorithm in the work-depth model is characterized by two complexity measures, *work* and *depth*, which are standard measures for analyzing shared-memory algorithms. The work, denoted as $\mathcal{W}$, is the total number of operations that are used by the algorithm. It is impossible that the parallel algorithm can perform less work than the best-known sequential algorithm since any parallel algorithm can be made sequential by performing each parallel step sequentially. Thus, the gold standard for parallel algorithms are algorithms whose work matches the running time of the best-known sequential algorithm. We call such algorithms *work-efficient* algorithms. The depth, denoted as $\mathcal{D}$, is the longest chain of sequential operations (JéJé, 1992). For sequential algorithms, the depth is equal to the work. For parallel algorithms, the depth of the algorithm is often much smaller than the work of the algorithm. An efficient parallel graph algorithm in this model should be work-efficient and has polynomial $\log n$ depth.

This model is particularly convenient for analyzing nested parallel algorithms. Assuming that a scheduler dynamically load-balances a parallel computation across all available workers, the expected running time is $\mathcal{O}(\mathcal{W}/\mathcal{P}+\mathcal{D})$ when using $\mathcal{P}$ workers. For the multi-core architecture, a worker is a working process corresponding to a physical core. In particular, for sequential algorithms, the work and the depth terms are equivalent. A parallel algorithm is *work-efficient* if its work is asymptotically equal to the work of the fastest sequential algorithm for the same problem (Blelloch and Maggs, 2010).

2.3 Graph Storage

In this work, explicit graphs and implicit graphs are discussed. Explicit graphs are typically stored in the *compressed sparse row* (CSR) format (Hong et al., 2012, 2013). This format uses two arrays to represent the graph: an $O(n)$-sized array stores an index to the beginning of each vertex's adjacency list and an $O(m)$-sized array stores each vertex's adjacency list. The CSR representation is compact, memory bandwidth-friendly, and thus suitable for efficient graph

traversals. It is easy to see that successors of each vertex $v \in V$ are ordered and thus can be traversed one by one in order.

On modern computers, registers can move data around in single clock cycles. However, registers are very expensive since a CPU typically has a limited number of registers available, e.g. the x86 architecture has 8 General-Purpose Registers (GPR) and 6 Segment Registers. The dynamic random access memory is very cheap but takes hundreds of cycles after a request to receive the data. To bridge this gap between them are the cache memories, named L1, L2, L3 in decreasing speed and cost. If the data is stored in memory sequentially, the CPU can prefetch the data into the cache for fast accessing, which is cache-friendly. For a graph stored in CSR format, we can see that sequentially traversing all edges is cache-friendly as the cache hit rate is high, but randomly traversing all edges is not cache-friendly as the cache hit rate is low.

Implicit graphs are defined as $G = (v_0, \texttt{POST})$ assuming that all the vertices in G are reachable from vertex v_0, where v_0 is the *initial vertex* and $\texttt{POST}(v)$ is a function that returns all of the *successors* of vertex v, that is, $\texttt{POST}(v) = v.post$. One kind of implicit graphs are model checking graphs (Pelánek, 2007) such that for each vertex v in a graph G, all the edges are calculated online by $\texttt{POST}(v)$. Another kind of implicit graphs are external graphs such that all the edges are stored on disks sequentially; once a vertex v is traversed, the edges of v are loaded into memory. The advantage of implicit graphs is that they allow handling large graphs with limited memory usage. However, much running time is spent on generating the edges via $\texttt{POST}(v)$. If an algorithm can run on implicit graphs without loading the whole graphs into memory, we say this algorithm has the *on-the-fly property*

2.4 Parallel Programming

2.4.1 OpenMP

OpenMP[1] (Open Multi-Processing) (Dagum and Menon, 1998) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran, on many platforms, instruction-set architectures and operating systems. In this paper, OpenMP (version 4.5) is used as the threading library to implement the parallel algorithms. The task-level parallelism is implemented by using the clause "`#pragma omp parallel for`" (C++ code). Given an input graph, this implementation statically assigns the same number of vertices to each worker p. For data-level parallelism, however, it is critical to handle a potential workload imbalance problem. Note that real-world graphs can be highly

[1] https://www.openmp.org/

irregular because of their scale-free property, e.g., a few vertices can have a huge number of successors while many vertices have only several successors. Therefore, statically assigning the same number of vertices to each worker naturally induces workload imbalance since the work of each vertex involves immediate propagation.

There is a better strategy. All of the vertices in the graph can be dynamically assigned to each worker p by the clause "`#pragma omp for schedule(dynamic, s)`". That means each worker executes a chunk of iterations with size s and then requests another chunk until no chunks remain to distribute. If one of the workers finishes processing a chunk of vertices early, it applies to the next chunk of vertices at once without waiting for other workers. In this way, we realize a relatively balanced load for each worker without difficulties. Note that the chunk size cannot be either much large or small; the too large chunk size may cause work-load imbalance for multiple workers; the too small chunk size may cause much running time spent on scheduling.

2.4.2 Atomic Primitives

Our algorithms in this thesis make use of the atomic to reduce the synchronization overhead. The compare&swap (`CAS`) and fetch&add (`FAA`) operations are universal atomic primitives that are supported on the majority of current parallel architectures (Valois, 1995; Michael, 2002; Milman et al., 2018).

As shown in Algorithm 4, the `CAS` atomic primitive takes three arguments, a variable (location) x, an old value a and a new value b. It checks the value of the variable x, and if it equals to the old value a, it updates the pointer to the new value b and then returns *true*; otherwise, it returns *false* to indicate that the updating fails. Here, we use a pair of *angular brackets*, $\langle ... \rangle$, to indicate that the operations in between are executed atomically.

Algorithm 1: CAS(x, a, b)

```
1 ⟨ if x = a then
2 |   x ← b; return true
3 else return false ⟩                    /* ⟨...⟩ atomic */
```

The `FAA` atomic primitive is shown in Algorithm 2. The old value of x is fetched and added by a. The new value of x is returned. For instance, there is a race condition when one worker is executing "$x = x + a$" and the other worker is executing "$x = x + b$" concurrently. Using `FAA` can efficiently get the correct result without workers affecting each other.

Algorithm 2: FAA(x, a)

1 $\langle x \leftarrow x + a \rangle$ /* $\langle ... \rangle$ atomic */

2.4.3 Lock Implementation

OpenMP (Open Multi-Processing) (Chandra et al., 2001) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran, on many platforms, instruction-set architectures, and operating systems. This paper uses OpenMP (version 4.5) as the threading library to implement the parallel algorithms. In this work, the key issue is how to implement the locks for synchronization. One solution is to use the OpenMP lock, "`omp_set_lock`" and "`omp_unset_lock`". Each worker will suspend the working task until the specified lock is available. The OpenMP lock will be efficient if a lot of work is within the locked region.

The other solution is the spin lock, which can be implemented by the atomic primitive `CAS`. Given a variable x as a lock, the `CAS` will repeatedly check x, and set x from `false` to `true` if x is `false`. In other words, one worker will busy-wait the lock x until it is released by other workers without suspension. The spin lock will be efficient if significantly little work within the locked region exists. In this case, suspending has a higher cost than busy waiting for multiple workers.

Algorithm 3: Lock(x)

```
1 i ← 1
2 while true do
3 |   if x.lock = false ∧ CAS(x.lock, false, true) then return
4 |   j ← i
5 |   while j > 0 do j ← j − 1
6 |   i ← 2 × i
```

Algorithm 3 shows an implementation of the spin lock. To reduce the bus traffic, $x.lock$ is tested before using `CAS` to set $x.lock$ from `false` to `true` (line 3). Additionally, it is more effective for other workers to *back off* for some duration, giving competing workers a chance to acquire the lock. Typically, especially for our use cases, the large number of unsuccessful tries indicates the longer the worker should back off. Here, we use a simple strategy that exponentially increases the back off time for each try (lines 1 and 4 - 6), where i and j are local variables without increasing the bus traffic (Herlihy et al., 2020).

We implement a condition-lock as in Algorithm 5. The condition c is checked before and after the `CAS` lock (lines 1 and 3). It is possible that other

workers may update the condition c simultaneously. If c is changed to `false` after locking x, x will be unlocked and then return `false` immediately (line 4). Such a conditional `Lock` can atomically lock x by satisfying c and thus can avoid blocking on a locked x that does not satisfy the condition c.

Algorithm 4: Lock(x) with c

```
1 while c do
2 |   if CAS(x, false, true) then
3 |   |   if c then return true
4 |   |   else x ← false
5 return false
```

2.5 C++

In the experiments, all tested algorithms are implemented in C++. The most important reason is that it is fair to compare the running time of all tested algorithms by implementing them with the same programming language like C++.

There are four other reasons. First, C++ is a compiled language that produces highly optimized machine code, making it well-suited for algorithms that require high performance. We compare the running time of different algorithms and C++ is the best choice. Second, C++ provides low-level control over atomic primitives, such as `CAS`; also, C++ supports the parallel programming interface OpenMP. Third, C++ supports object-oriented programming (OOP) concepts, such as classes, inheritance, and polymorphism. OOP can help organize code and make it easier to maintain and extend over time Finally, C++ has a Standard Template Library (STL), which is a collection of generic algorithms, data structures, and containers that are part of the C++ Standard Library.

2.6 Random Selection

In the experiment, we always randomly select a bunch of items for testing, e.g. we randomly select 100,000 edges from a graph for insertion and removal operations. In this case, we require a large size of random numbers. Our implementation is based on the pseudo-random number generator, e.g. the `rand`() function supported by C++ after including the head file "stdlib.h", which can produce numbers that are uniformly distributed in a certain range. With these sources of pseudo-random numbers, we can randomly select items for testing our algorithms.

Part II

Graph Trimming

Chapter 3

Graph Trimming

Given a large data graph, trimming techniques can reduce the search space by removing vertices without outgoing edges. One application is to speed up the parallel decomposition of graphs into strongly connected components (SCC decomposition), which is a fundamental step for analyzing graphs. We observe that graph trimming is essentially a kind of arc-consistency problem, and AC-3, AC-4, and AC-6 are the most relevant arc-consistency algorithms for application to graph trimming. The existing parallel graph trimming methods require worst-case $\mathcal{O}(nm)$ time and worst-case $\mathcal{O}(n)$ space for graphs with n vertices and m edges. We call these parallel AC-3-based as they are much like the AC-3 algorithm. In this chapter, we propose AC-4-based and AC-6-based trimming methods. That is, AC-4-based trimming has an improved worst-case time of $\mathcal{O}(n+m)$ but requires worst-case space of $\mathcal{O}(n+m)$; compared with AC-4-based trimming, AC-6-based has the same worst-case time of $\mathcal{O}(n+m)$ but an improved worst-case space of $\mathcal{O}(n)$. We parallelize the AC-4-based and AC-6-based algorithms to be suitable for shared-memory multi-core machines. The algorithms are designed to minimize synchronization overhead. For these algorithms, we also prove the correctness and analyze time complexities with the work-depth model.

In experiments, we compare these three parallel trimming algorithms over a variety of real and synthetic graphs on a multi-core machine, where each core corresponds to a worker. Specifically, for the maximum number of traversed edges per worker by using 16 workers, AC-3-based traverses up to 58.3 and 36.5 times more edges than AC-6-based trimming and AC-4-based trimming, respectively. That is, AC-6-based trimming traverses much fewer edges than other methods, which is meaningful especially for implicit graphs. In particular, for the practical running time, AC-6-based trimming achieves high speedups over graphs with a large portion of trimable vertices.

3.1 Introduction

Given a large data graph, the graph trimming is to remove vertices without outgoing edges. One issue is that trimming such unqualified vertices may cause other vertices to become useless. Naively repeating the trimming process may lead to a quadratic worst-case time complexity. Thus, linear time bounded graph trimming methods are desired. Additionally, the availability of multi-core processors motivates efficient parallelization of such graph trimming methods. Here, a *worker* is a working process corresponding to a physical core for a multi-core processor.

To the best of our knowledge, there exists little work on parallel trimming over large data graphs, except for (III et al., 2005; Hong et al., 2013; Slota et al., 2014; Ji et al., 2018; Chen et al., 2018). In these studies, the graph trimming is adopted to quickly remove the vertices without out-going edges so that can speed up the strongly connected component (SCC) decomposition. In (III et al., 2005), McLendon et al. first apply a linear time graph trimming method to remove *size*-1 SCCs; however, a parallel version is not provided. In (Hong et al., 2013), Hong et al. propose a quadratic time graph trimming technique by "peeling" *size*-1 and *size*-2 SCCs, i.e. SCCs with only 1 or 2 vertices. The "peeling" step is straightforward: (1) all vertices are checked in parallel and the *trimmable* ones are removed, which may cause other vertices to become trimmable; (2) this process is repeated until no vertex can be removed from the graph. The advantage of this graph trimming technique is that it can be highly parallelized without difficulties. However, it has a quadratic worst-case time complexity of $O(nm/\mathcal{P} + \alpha)$, where n is the number of vertices, m is the number of edges, $\mathcal{P}$ is the number of workers, and α is the depth of the algorithm (explained in the next section). This parallel trimming technique is widely used in later SCC decomposition methods (Slota et al., 2014; Ji et al., 2018; Chen et al., 2018).

In this work, we apply the well-known *arc-consistency* (AC) algorithms to graph trimming. Based on that, we not only classify existing graph trimming algorithms but also propose a new graph trimming algorithm that improves the time and space complexities by an order of magnitude. Before discussing these contributions, we first show an application of graph trimming, the SCC decomposition in large graphs (III et al., 2005; Hong et al., 2013; Slota et al., 2014; Ji et al., 2018; Chen et al., 2018).

3.1.1 An Application of Graph Trimming

Detecting the strongly connected components in directed graphs, the so-called SCC decomposition, is one of the fundamental analysis steps in many applications such as social networks (Kumar et al., 2010), communication networks (Sun et al., 2016), knowledge networks (Auer et al., 2007), and model checking graphs (Hojati et al., 1993). Given a directed graph $G = (V, E)$, a *strongly connected component* of G is a maximal set of vertices $C \subseteq V$ such that every two vertices u and v in C are reachable from each other. The early SCC algorithms are based on depth-first search (DFS) (Tarjan, 1972; Cormen et al., 2009). However, lexicographical-first DFS is P-complete and even the random DFS is hard to parallelize (Reif, 1985; Aggarwal and Anderson, 1988). The breadth-first search (BFS) based Forward-Backward (FW-BW) algorithm has been proposed. Unlike DFS, BFS can be parallelized without difficulty. Starting from a selected pivot vertex, FW-BW performs a forward BFS to identify the vertex set FW that the pivot can reach, followed by a backward BFS to identify the set BW that can reach the pivot. The intersection between FW and BW is an SCC that contains the pivot (Fleischer et al., 2000). In the worst case, each vertex can be selected as a pivot to travel the whole graph in $O(m)$, which yields a quadratic time complexity of $O(mn)$ (Fleischer et al., 2000). In (Coppersmith et al., 2003; Fleischer et al., 2007), the worst-case time complexity is improved to $O(m \log n)$ by using a divide-and-conquer approach.

Interestingly, real-world graphs demonstrate SCC features that follow the *power-law property* (Hong et al., 2013), that is, several large SCCs take the majority of vertices and the rest of them are trivial SCCs. More importantly, most of the trivial SCCs are size-1 SCCs. The key observation is that a size-1 SCC is easy to identify: it has zero incoming edges or zero outgoing edges. Therefore, graph trimming can be used to remove such size-1 SCCs in parallel with less computational effort than FW-BW and thus in practice can speed up FW-BW. Analogously to size-1 SCCs, size-2 (Hong et al., 2013) and size-3 (Ji et al., 2018) SCCs also can be trimmed but with more computational effort.

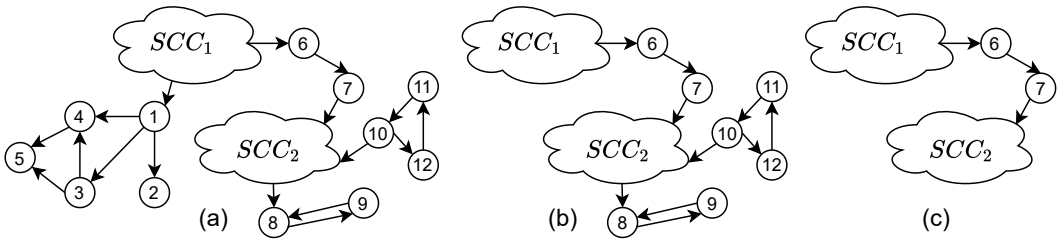


Figure 3.1: A graph that can use graph trimming to remove size-1, size-2 and size-3 SCCs.

Figure 3.1 illustrates the FW-BW algorithm with graph trimming. Figure 3.1(a) shows that there are altogether two large SCCs, SCC_1 and SCC_2 (whose sizes are greatly larger than 3) and the other trivial SCCs. It is easy to see that vertices v_1 to v_7 are size-1 SCCs, vertices v_8 and v_9 compose one size-2 SCC, and vertices v_{10} to v_{12} compose one size-3 SCC. In Figure 3.1(b), we first try to trim all size-1 SCCs: (1) in the first repetition, vertices v_5 and v_2 are removed since they have no outgoing edge, which causes vertex v_4 to have no outgoing edges; (2) in the second repetition, vertex v_4 is removed, which causes vertex v_3 to have no outgoing edges; (3) in the third repetition, vertex v_3 is removed, which causes vertex v_1 to have no outgoing edges; (4) in the final repetition, vertex v_1 is removed. Similarly, in Figure 3.1(c), size-2 and size-3 SCCs can also be removed. Note that vertices v_6 and v_7, located between two large SCCs, are size-1 SCCs, but they can not be directly trimmed. After the first round of graph trimming, the FW-BW algorithm can identify two large SCCs, SCC_1 and SCC_2, which also can be deleted from the graph. After removing the two large SCCs, the second round of trimming can remove vertices v_6 and v_7 with two iterations.

The naive trimming method as used in FW-BW (Fleischer et al., 2007) has a quadratic time complexity of $O(mn)$ in the worst case. The drawback of such trimming is that it sacrifices the better worst-case time complexity of FW-BW, $O(m \log n)$ (Fleischer et al., 2007). From our experiments, we noticed that the running time of such trimming will dramatically increase with the number of peeling steps α because of the increasing number of repetitions. This is why FW-BW with trimming as in (Hong et al., 2013) is only efficient for *small-word graphs*. The small-world property states that the *diameters* (greatest shortest-path distance between any pair of vertices) of graphs are very small even for very large graph instances (Watts and Strogatz, 1998), which always implies a small number of peeling steps. The focus of this paper is to improve traditional graph trimming so that algorithms based on FW-BW with trimming (III et al., 2005; Hong et al., 2013; Slota et al., 2014; Ji et al., 2018; Chen et al., 2018) can be more efficient, especially for non-small-world graphs.

For instance, in (Ji et al., 2018), the parallel SCC decomposition algorithm ISPAN is proposed. It combines the power of graph trimming and FW-BW, both of which can be efficiently parallelized. In particular, graph trimming is used at two places; before large SCC detection, trimming is used to remove the size-1 SCCs; after the large SCC is detected, trimming is again used to remove size-1, size-2, and size-3 SCCs. The evaluation uses 56 workers over 16 graphs and shows that ISPAN achieves a significant speedup of 171 - 6591 times over the sequential DFS-based Tarjan's algorithm (Cormen et al., 2009) and of 85 - 1475 times over the parallel DFS-based UFSCC algorithm (Bloemen et al., 2016).

3.1.2 The New Method

Essentially, graph trimming is a kind of *Constrain Satisfaction Problem* (CSP), that is, a set of vertices must satisfy a number of constraints or limitations, e.g. each vertex needs at least one outgoing edge or it will be removed as a size-1 SCC. Many filtering algorithms (Dib et al., 2010) have been proposed to remove values that obviously do not belong to the solution of a CSP and thus reduce the search space. The closest related filtering algorithms to graph trimming are *Arc-Consistency* (AC) algorithms for binary CSPs, in particular AC-3 (Mackworth and Freuder, 1985), AC-4 (Mohr and Henderson, 1986), and AC-6 (Freuder and Régin, 1999).

The key observation is that the graph trimming technique in (III et al., 2005) is like AC-4 (*AC-4-based*). Also, the other widely used graph trimming technique in (Hong et al., 2013; Slota et al., 2014; Ji et al., 2018; Chen et al., 2018) is like AC-3 (*AC-3-based*). Stimulated by AC-6, we design a novel graph trimming algorithm (*AC-6-based*). Compared to AC-3-based and AC-4-based trimming, our new AC-6-based trimming is more complicated and not easy to parallelize. To the best of our knowledge, there exists little work on parallel AC algorithms (Cooper and Swain, 1992; Kirousis, 1993). In this work, we design efficient sequential and parallel versions of AC-6-based trimming for a multi-threaded shared memory architecture.

Table 3.1 summarizes the complexities of different parallel graph trimming algorithms in the *work-depth* model, where the *work* is the number of operations used by the algorithm and the *depth* is the length of the longest sequential dependence in the computation. We can see that all three trimming algorithms have the different parallel depth, and AC-3-based trimming has a smallest depth. AC-3-based trimming has larger worst-case work and time complexities than the other two algorithms. The AC-6-based and AC-4-based algorithms have the same worst-case work and time complexities. We show that AC-6-based trimming traverses fewer edges and uses less space. For example, over all tested graphs in our first experiments, AC-6-based trimming reduces the number of traversed edges 3.3 - 192.5 times compared with AC-4-based trimming and 1.5 - 44 times compared with AC-3-based trimming.

To parallelize the AC-4-based and AC-6-based algorithms, the conventional way is with mutual exclusion by using `Lock` and `Unlock` operations that guarantee exclusive access to data structures shared by multiple workers. In this work, however, we use atomic primitives to minimize the synchronization overhead.

Trimming	On-The-Fly	Worst-Case (O) $\mathcal{W}$	$\mathcal{D}$	Space
AC-3-based	✓	$\alpha(n+m)$	αDeg_{out}	n
AC-4-based	✗	$n+m$	$\|Q_p\| Deg_{in} Deg_{out}$	$n+m$
AC-6-based	✓	$n+m$	$\|Q_p\| {Deg_{in}}^2$	n

Table 3.1: The worst-case work, depth, and space complexities of parallel graph trimming algorithms, where n is the number of vertices, m is the number of edges, $\mathcal{P}$ is the number of total workers, α is the number of peeling steps, Deg_{out} is the maximal out-degree for all vertices, Deg_{in} is the maximal in-degree for all vertices, $|Q_p|$ is the upper-bound size of waiting sets among $\mathcal{P}$ workers such that sometimes $|Q_p| \geq \alpha$.

3.1.3 On-the-fly Property

The *on-the-fly* property (Bloemen et al., 2016) means an algorithm can run on an implicit graph defined as $G = (v_0, \texttt{POST})$, where v_0 is the *initial vertex* and $\texttt{POST}(v)$ is a function that returns all of the *successors* of vertex v. One drawback of the FW-BW method is that the backward search requires reverse edges, which means all edges have to be loaded into memory; storing the graph as an adjacent list with only outgoing edges is not sufficient. The on-the-fly property is necessary when handling large graphs that occur in e.g. verification (Merz, 2001), as it may allow the algorithm to terminate early after processing only a fraction of the graph without needing memory space for loading the whole graph. It also benefits algorithms that rely on implicit graphs (Pelánek, 2007), in which the edges are calculated online by function $\texttt{POST}(v)$.

The on-the-fly properties of three graph trimming algorithms are summarized in Table 3.1. It is easy to see that the AC-4-based trimming cannot run on-the-fly as it requires reverse edges and thus the whole graph must be loaded into the memory. AC-3-based and AC-6-based trimming can run on-the-fly as they only rely on the post of vertex v when traversing each vertex $v \in V$ and their space usage is bounded by $O(n)$. However, compared with AC-3-based trimming, AC-6-based trimming needs much less work. Note that on implicit graphs, all the edges are computed online by the function $\texttt{POST}(v)$, which typically costs more running time than directly loading edges from memory like with explicit graphs. The proposed AC-6-based trimming traverses fewer edges than AC-3-based trimming and thus performs better on implicit graphs.

3.1.4 Contribution

The contributions of this work are summarized below:

- We provide a formal definition of graph trimming based on the Constraint Satisfaction Problem (CSP) and Arc-Consistency (AC). Following three well-known arc-consistency algorithms, that is, AC-3, AC-4, and AC-6, we categorize the existing graph trimming algorithms as AC-3-based (Hong et al., 2013; Slota et al., 2014; Ji et al., 2018; Chen et al., 2018) and AC-4-based algorithms (III et al., 2005).

- We revisit the existing parallel AC-3-based algorithm. We give the detailed steps of the AC-4-based algorithm and parallelize it using atomic primitives.

- We propose a novel AC-6-based algorithm that has optimized time and space complexities. We further parallelize the AC-6-based algorithm using atomic primitives. These are the main contributions of this work.

- For all three graph trimming algorithms, we formally discuss their correctness, time complexity, and space complexity. The time complexities for parallel algorithms are analyzed in the work-depth model.

- Finally, for all three parallel trimming algorithms, our experiments compare the number of traversed edges and practical running time with 1 to 16 workers over a variety of real and synthetic graphs.

3.2 Related Work

Parallel DFS-based SCC Decomposition. In Section 1, several methods for BFS-based SCC decomposition were introduced. Although DFS is inherently sequential (Reif, 1985), there is a lot of work based on Tarjan's algorithm. In (Lowe, 2016), Lowe proposed a synchronized Tarjan's algorithm, that is, multiple instances of Tarjan's algorithm run without overlapping stacks. To do this, a worker is suspended on a vertex which is located in another worker's stack and then both workers' stacks can be merged if necessary. The drawback is that this stack merging leads to a worst-case quadratic time complexity of $O(n^2)$. Lowe's experiments show decent speedups on model checking graphs with trivial SCCs, but not for graphs with large SCCs. In (Renault et al., 2015), Renault et al. present a novel algorithm without sacrificing the linear time complexity, $O(n + m)$, and the on-the-fly property. Multiple instances of Tarjan's algorithm run and communicate completely explored SCCs via a shared union-find structure. Bolomen et al. (Bloemen et al., 2016; Bloemen, 2015) proposed an improved UFSCC algorithm which communicates partially found SCCs by using a modified union-find data structure. In their experiments, UFSCC shows a significant speedup compared to Renault's algorithm (Renault et al., 2015) on implicit model checking inputs and synthetic

graphs. One notable property of these algorithms is that they can run on-the-fly on implicit graphs.

However, above DFS-based SCC algorithms do not utilize graph trimming techniques to remove trivial SCCs. A possible reason is that the traditional graph trimming technique has quadratic worst-case time complexity and, more importantly, it is hard to run on-the-fly. The proposed parallel AC-6-based graph trimming algorithm has linear running time and has the on-the-fly property, so it can be used to quickly trimming a high ratio of size-1 SCCs for above DFS-based SCC algorithms.

Graph Trimming. Generally, the term "graph trimming" is widely used in graph algorithms to minimizing the search space and the trimming rules may be different for different problems. For example, in (Heule, 2019), "graph trimming" computes a smaller and smaller unsatisfiable core for a propositional formula; in (Gao et al., 2015), "graph trimming" is used to minimize the number of vertices as monitors to identify all interesting links; in (Erlebach et al., 2010), given a graph in which each vertex has a nonnegative weight, "graph trimming" deletes vertices with a small total weight such that the remaining graph does not contain any long simple paths. Note that, in the current work, given a directed graph, the terminology "graph trimming" is specifically confined as each vertex has at least one outgoing edge.

Another related term is "graph pruning". For example, in (Harabor and Grastien, 2011), given a geographical graph, "graph pruning" can dynamically jump over some searching branches by some simple rules for finding the path between two nodes.

3.3 Preliminary

A *constraint satisfaction problem* (CSP) (Russell and Norvig, 2009; Dib et al., 2010) can be defined as a triple $P = (X, D, C)$, where $X = \{X_1, ..., X_n\}$ is a set of n variables, $D = \{D(X_1), ..., D(X_n)\}$ is the set of n domains such that $D(X_i)$ is a set of possible values of variable X_i, and C is a set of constraints that specify allowable combinations of values. A *solution* of a constraint set C is an instantiation of the variables such that all constraints are satisfied. Here, we restrict to *binary constraints* C_{ij} between pairs (X_i, X_j) of variables, i.e. $C = \{C_{ij} \mid i, j \in 1...n\}$.

A value $v_i \in D_i$ is *binary consistent* with a constraint C_{ij} if there exists $v_j \in D_j$ such that (v_i, v_j) satisfies C_{ij}. Then $v_j \in D_j$ is called a *support* of $v_i \in D_j$ over C_{ij}. A value $v_i \in D_i$ is *viable* if it has supports for every D_j such that each $C_{ij} \in C$ is satisfied. A variable in a CSP is *arc-consistent* (AC) if every value in its domain satisfies each binary constraint $C_{ij} \in C$.

Several AC algorithms have been proposed for removing values that are not viable. AC-1 (Mackworth, 1977) revisits all the binary arcs that have to be revisited once some domains are reduced. Improving on AC-1, algorithm AC-2 (Mackworth, 1977) only revisits the arcs that are affected by reducing some domains. Algorithm AC-3 (Mackworth, 1977; Mackworth and Freuder, 1985) generalizes and simplifies AC-2.

Algorithm 5 shows the detailed steps of AC-3. Initially, the global set Q includes all constraint arcs $C_{ij} \in C$ (line 1). Each constraint arc C_{ij} is picked and then removed from the set Q (line 3), and each pair of values in the domains $D(X_i)$ and $D(X_j)$ are checked by the procedure `Revise` (line 4), that is, for each value $v_i \in D(X_i)$, if $D(X_j)$ does not contain a value v_j such that (v_i, v_j) satisfies the constraint C_{ij}, the value v_i is repeatedly removed from $D(X_i)$ (lines 7 - 13). If $D(X_i)$ is changed, the associated constraints C_{ij} are placed into Q again (lines 5 and 6). This process is repeated until the set Q becomes empty (line 2). It is easy to see that AC-3 is not efficient since the revision of any domain will force neighbor constraints to be revisited again.

Algorithm 5: AC-3

input : An arc-consistency problem $P = (X, D, C)$
output: A filtered domain set D

```
1  Q ← C
2  while Q ≠ ∅ do
3  |  remove a constraint C_ij from Q
4  |  if Revise(X_i, X_j, D) then
5  |  |  for X_k ∈ {X_k' : C_k'i ∈ C} \ {X_j} do
6  |  |  |  Q ← Q ∪ {C_ki}
7  procedure Revise(X_i, X_j, D)
8  |  revised ← false
9  |  for v_i ∈ D(X_i) do
10 |  |  if no value v_j in D(X_j) satisfies C_ij then
11 |  |  |  delete v_i from D(X_i)
12 |  |  |  revised ← true
13 |  return revised
```

AC-4 (Mohr and Henderson, 1986) improves the worst-case time complexity of AC-3 by using auxiliary data structures, *supports* and *counters*, but its average running time is close to the worst-case time complexity. However, AC-3 has better average running time and space usage than AC-4 and thus AC-3 is always preferred to AC-4 (Russell and Norvig, 2009) in practice. Algorithm AC-6 (Bessière, 1994) combines AC-3 and AC-4. It only records one support for each value, unlike AC-4 which records all supports, since a single support

is enough to prove that a value is viable. Because of this, AC-6 has the same worst-case time complexity as AC-4 but averagely performs much better than AC-4 in many applications. Further, AC-6 needs less space than AC-4 since for each value only a single support is recorded.

Table 3.2 summarizes the time and space complexities of these three algorithms. AC-6 has the best worst-case time and space complexities. AC-4 and AC-6 have the same time complexity. However, in reality, AC-3 and AC-6 perform sometimes better than AC-4 due to AC-4 always running close to its worst-case time. The details are explained in the next section.

Algorithm	Time (O)	Space (O)
AC-3	ed^3	$e + kd$
AC-4	ed^2	ed^2
AC-6	ed^2	ed

Table 3.2: The worst-case time and space complexities of three arc-consistency algorithms, where e is the number of arcs, k is the number of variables, and d is the size of the largest variable's domain.

3.4 Graph Trimming

Definition 3.4.1 (Trimmed Graph)**.** Given a directed graph $G = (V, E)$, the trimmed graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$ is a maximal subgraph of G, where each vertex has at least one outgoing edge, formally $\forall v \in V' : v.post \neq \emptyset$.

The graph trimming is to obtain trimmed graphs according to Definition 3.4.1. Without changing the original graph $G = (V, E)$, each vertex $v \in V$ is assigned a status, denoted as $v.status$, with values `LIVE` and `DEAD`, which indicates if vertex v is located in the graph (live) or removed (dead), respectively.

3.5 Graph Trimming as Arc Consistency

Intuitively, we can regard graph trimming as a graph with a constraint that each vertex has at least one outgoing edge. Based on this observation, we define graph trimming as an arc-consistency problem with one single variable, viz. the set of all vertices, and a single binary constraint, viz. each vertex must have at least one outgoing edge as one support. Then, trimming a graph means determining the domain of available vertices.

More formally, given a directed graph $G = (V, E)$, graph trimming can be defined as an arc-consistency problem (X, D, C) with variables $X = \{X_1\}$, domains $D = \{D(X_1)\}$ and constraint $C = \{C_{11}\}$. Here, we assume that $X_1 = V$ and $C_{11} = E$, that is, each vertex $v_1 \in D(V)$ must has at least one support vertex $v_1' \in D(V)$ in the same domain, where $(v_1, v_1') \in E$.

Consequently, three important AC algorithms, AC-3, AC-4, and AC-6, can be applied to graph trimming. Interestingly, we find that one widely used graph trimming method (Hong et al., 2013; Slota et al., 2014; Ji et al., 2018; Chen et al., 2018) is analogous to AC-3 (we call it AC-3-based). The other (III et al., 2005) is analogous to AC-4 (we call it AC-4 based); however, the detailed steps are not discussed, and a parallel version is not provided. As a contribution, we design a novel graph trimming algorithm based on AC-6 (we call it AC-6-based).

3.6 AC-3-Based Graph Trimming

In the graph trimming problem, there exists only a single variable and a single constraint. Therefore, AC-3, as shown in Algorithm 5, can be simplified when applied to graph trimming. The idea is straightforward: (1) for all vertices in a graph, the vertices with zero out-degrees are removed; (2) this process is repeated until the graph does not change. This naive trimming method is widely used (Hong et al., 2013; Slota et al., 2014; Ji et al., 2018; Chen et al., 2018) for quickly removing the size-1 SCCs, but the correctness and complexities are not formally discussed. In this section, we revisit the existing parallel AC-3-based algorithm for graph trimming and formally discuss the correctness and complexities. The sequential AC-3-based algorithm is immediate and not discussed further.

3.6.1 The Parallel AC-3-Based Algorithm

Algorithm 6 shows the detailed steps of the parallel AC-3-based algorithm for graph trimming. The procedure `ZeroOutDegree`(v) (lines 11 - 14) returns `TRUE` if vertex v has at least one available outgoing edge and `FALSE` otherwise. All vertices in V are initialized as `LIVE`. After partitioning V into $V_1 \dots V_{\mathcal{P}}$, we have $\mathcal{P}$ workers execute the procedure $\texttt{Trim}_p(V_p)$ in parallel (lines 4 and 5). The main procedure $\texttt{Trim}_p(V_p)$ (lines 7 - 10) removes the vertices in V_p that have an out-degree of zero. The removing process repeats until the graph does not change (lines 2 - 6). One advantage of this algorithm is that it is easy to parallelized: each copy of procedure $\texttt{Trim}_p(V_p)$ for a worker p (line 5) can run in parallel with only *change* as the sole shared variable.

For the specific implementations in (Hong et al., 2013; Ji et al., 2018; Chen et al., 2018), there are two strategies to improve the AC-3-based algorithm for

Algorithm 6: Parallel AC-3-based Graph Trimming

```
   input  : Graph G = (V, E)
   output: Trimmed Graph G
 1 for v ∈ V do v.status ← LIVE
 2 repeat
 3 |   change ← false
 4 |   partition V into V_1 ... V_P
 5 |   Trim_1(V_1) || ... || Trim_P(V_P)
 6 until ¬ change
 7 procedure Trim_p(V_p)
 8 |   for v ∈ V_p : v.status = LIVE do
 9 |   |   if ZeroOutDegree(v) then
10 |   |   |   v.status ← DEAD; change ← true
11 procedure ZeroOutDegree(v)
12 |   for w ∈ v.post with w.status = LIVE do
13 |   |   return false
14 |   return true
```

graph trimming.

- If the transposed graph G^T is loaded in memory, another constraint can be considered, that each vertex $v \in V$ must have at least one available incoming edge. That means the in-degree also be checked (line 9). In this case, more size-1 SCCs can be quickly trimmed. The problem is that the transposed graph is required, which costs $O(n + m)$ memory space.

- The number of repetitions can be limited to a constant number like 3 or the repetitions stop when the number of removed vertices is less than a threshold like 100 (line 6). The problem is that some of the trimable vertices may not be removed. This strategy is sometimes effective at reducing the computational time but sometimes not, since the worst-case time complexity is not improved.

Correctness. For the correctness, trimming has to be sound and complete. Soundness means that all removed vertices, which are assigned a status of `DEAD`, must have no outgoing edges or only edges to removed vertices:

$$\begin{aligned} sound(V) \;\equiv\; \forall v \in V :\; & v.status = \texttt{DEAD} \implies \\ & (\forall w \in v.post : w.status = \texttt{DEAD}) \end{aligned} \tag{3.6.1}$$

Completeness means that all vertices that have no outgoing edges or have only outgoing edges to removed vertices are removed:

$$\begin{aligned} \mathit{complete}(V) \;\equiv\; \forall v \in V :\; & (\forall w \in v.\mathit{post} : w.\mathit{status} = \texttt{DEAD}) \implies \\ & v.\mathit{status} = \texttt{DEAD} \end{aligned} \tag{3.6.2}$$

The algorithm has to ensure both soundness and completeness for all vertices in the graph:

$$\mathit{sound}(V) \;\wedge\; \mathit{complete}(V) \tag{3.6.3}$$

which is equivalent to:

$$\forall v \in V : v.\mathit{status} = \texttt{DEAD} \;\equiv\; (\forall w \in v.\mathit{post} : w.\mathit{status} = \texttt{DEAD}) \tag{3.6.4}$$

For arguing about the correctness of for-loops, we use following rule: consider the loop **for** $x \in X$ **do** S and let $P(X')$ be a predicate. If (1) initially $P(\emptyset)$ holds and (2) under precondition $P(X')$ the body S establishes postcondition $P(X' \cup \{x\})$ for any $X' \subset X$ and $x \in X \setminus X'$, then finally $P(X)$ holds; X' is the set of visited elements and $P(X')$ is the loop invariant.

Theorem 3.6.1 (Soundness)**.** *For any $G = (V, E)$ Algorithm 6 terminates with $\mathit{sound}(V)$.*

Proof. The invariant of the for-loop of $\texttt{Trim}_p$ (lines 8-10) is $\mathit{sound}(V')$: initially that holds as the universal quantification in $\mathit{sound}(V')$ is empty. The invariant is preserved as $v.\mathit{status}$ is only set to DEAD if the status of all $w \in v.\mathit{post}$ is DEAD (lines 9 and 10). We use the fact that ZeroOutDegree(v) returns $(\forall w \in v.\mathit{post} : w.\mathit{status} = \texttt{DEAD})$. The postcondition of $\texttt{Trim}_p(V_p)$ is therefore $\mathit{sound}(V_p)$. The postcondition (line 5) is then $\mathit{sound}(V_1) \wedge \ldots \wedge \mathit{sound}(V_{\mathcal{P}})$, which is equivalent to $\mathit{sound}(V)$. Thus $\mathit{sound}(V)$ is the invariant of the repeat-until loop (lines 2 - 6) and therefore holds on termination. □

Theorem 3.6.2 (Completeness)**.** *For any $G = (V, E)$ Algorithm 6 terminates with $\mathit{complete}(V)$.*

Proof. The invariant of the for-loop of $\texttt{Trim}_p$ (lines 8-10) is $\neg \mathit{change} \implies \mathit{complete}(V')$, where V' is the set of visited vertices. If *change* is TRUE, the invariant is obviously preserved as *change* is not set to FALSE in this or any other parallel copy of $\texttt{Trim}_p$. Suppose *change* is FALSE and $\mathit{complete}(V')$ holds. For $v \in V \setminus V'$ that remains LIVE, the procedure ZeroOutDegree(v) (line 9), which computes $(\forall w \in v.\mathit{post} : w.\mathit{status} = \texttt{DEAD})$, must return false. Since setting $v.\mathit{status}$ to DEAD may invalidate $\mathit{complete}(V' \cup \{v\})$ for this or some other parallel copy of $\texttt{Trim}_p$, variable *change* is set to TRUE, which re-establishes the invariant for this and all other parallel copies of $\texttt{Trim}_p$. □

Complexities. The complexity of parallel AC-3-based graph trimming has been discussed in existing work (Hong et al., 2013; Slota et al., 2014; Ji et al., 2018; Chen et al., 2018), but not with the work-depth model. We adopt the work-depth model to analyze the time complexity.

Theorem 3.6.3. *Algorithm 6 requires $O(\alpha(n+m))$ expected work, $O(\alpha Deg_{out})$ depth, and thus $O(\alpha(n+m)/\mathcal{P} + \alpha Deg_{out})$ time complexity.*

Proof. For the inner for-loop (lines 8 - 10), checking the out-degree of each vertex $v \in V$ requires $O(n+m)$ work in the worst case since all edges may need to be traversed in case of some vertices are removed. For the outer repeat-loop (lines 2 - 6), all vertices must be checked again once at least one vertex is removed. The repetition is carried out α times. Therefore, the expected work is $O(\alpha(n+m))$.

We analyze the working depth. For the procedure $\texttt{Trim}_p$, the inner for-loop (lines 12 and 13) in procedure `ZeroOutDegree` run in sequential with depth $O(Deg_{out})$. The repetition (lines 2 - 6) is carried out α times. Therefore, the theoretical working depth is $O(\alpha Deg_{out})$, and thus the theoretical time complexity is $O(\alpha(n+m)/\mathcal{P} + \alpha Deg_{out})$. □

Theorem 3.6.4. *The space complexity of Algorithm 6 is $O(n)$.*

Proof. Each vertex $v \in V$ requires *status* in memory to record if vertex v is `LIVE` or `DEAD`. Besides *status*, no other auxiliary data structures are utilized. Therefore, the space complexity is $O(n)$. □

3.7 AC-4-Based Graph Trimming

AC-4 improves the worst-case time complexity of AC-3 by using auxiliary data structures, *supports* and *counters*. Specifically, for each value in its domain, its supports are recorded, and its total number of supports is recorded with a counter. When removing one value, the corresponding counters located by supports are decreased by one. The values whose counters are reduced to zero must be removed, which may cause other values to be removed. In a word, the supports and counters are used for efficient propagation after unqualified values are removed.

The AC-4 algorithm can be applied to graph trimming, which we call AC-4-based trimming. For a directed graph $G = (V, E)$, the supports can be simplified as the transposed graph $G^T = (V, E^T)$, and the counters can be implemented by out-degree counters for all vertices $v \in V$, denoted as $v.deg_{out}$. AC-4-based graph trimming is used in (III et al., 2005) for quickly removing size-1 SCCs to speed up SCC decomposition; nevertheless, the details of this algorithm are not discussed, and its parallel version is not given. In this section, we provide both sequential and parallel AC-4-based algorithms.

3.7.1 The Sequential AC-4-Based Algorithm

Algorithm 7 shows the detailed steps of the sequential AC-4-based algorithm. Compared to the AC-3-based algorithm, there are two new data structures: 1) the transposed graph $G^T = (V, E^T)$ is required for accessing the predecessors of a vertex $v \in V$ (line 6); 2) a waiting set Q is required for propagation when processing removed vertices (line 3). The procedure `DoDegree(`v, Q`)` (lines 9 - 11) removes the vertex v and puts it into Q for propagation if v is `LIVE` and its out-degree counter $v.deg_{out}$ is zero. That means all vertices in the waiting set Q are dead.

Now we explain Algorithm 7. Initially, for all vertices, their status and out-degree counters are correctly initialized (line 1). For each vertex $v \in V$, the out-degree counter is checked by calling procedure `DoDegree(`v, Q`)` (line 3). The removed vertices are added into the wait set Q and then propagated to update the out-degree counters of other vertices (lines 4 - 8). That is, for each vertex $w \in Q$, all its predecessors' out-degree counters are off by 1 and then checked by the procedure `DoDegree(`v, Q`)` (lines 6 - 8). During this process, new vertices may be removed and added into the waiting set Q so that the algorithm does not terminate until Q becomes empty (line 4).

Algorithm 7: Sequential AC-4-based Graph Trimming

```
   input  : Graph G = (V, E) and its transposed graph G^T = (V, E^T)
   output: Trimmed graph G
 1 for v ∈ V do v.status, v.deg_out ← LIVE, |v(G).post|
 2 for v ∈ V with v.status = LIVE do
 3 |   Q ← ∅; DoDegree(v, Q)
 4 |   while Q ≠ ∅ do
 5 |   |   remove a vertex w from Q
 6 |   |   for v' ∈ w(G^T).post do
 7 |   |   |   v'.deg_out ← v'.deg_out − 1
 8 |   |   |   DoDegree(v', Q)
 9 procedure DoDegree(v, Q)
10 |   if v.deg_out = 0 ∧ v.status = LIVE then
11 |   |   v.status ← DEAD; Q ← Q ∪ {v}
```

Correctness. We show soundness and completeness together.

Theorem 3.7.1 (Soundness and Completeness)**.** *For any $G = (V, E)$ Algorithm 7 terminates with sound(V) and complete(V).*

Proof. Let V' be the set of vertices visited by the outer for-loop (lines 2 - 8). The invariant of the outer for-loop (lines 2 - 8) is that all vertices are sound,

all visited vertices are complete, and that for each vertex $v \in V'$ the counter $v.deg_{out}$ is the number of live vertices of outgoing edges:

$$\begin{aligned} &sound(V) \ \wedge \ complete(V') \\ &\wedge \ (\forall v \in V : v.deg_{out} = |\{w \in v.post \mid w.status = \texttt{LIVE}\}|) \end{aligned}$$

The invariant holds initially as setting all vertices to `LIVE` makes them sound and V' is initially empty.

The invariant of the while-loop (lines 4 - 8) is that all states are sound, but setting a vertex to `DEAD` may lead to its predecessors being incomplete; also, all vertices in Q have been set to `DEAD` and all $v'.deg_{out}$ are off by one where v' are all vertices with a successor in Q,

$$\begin{aligned} &sound(V) \ \wedge \ complete(V' \setminus Q.pre) \ \wedge \ (\forall v \in Q : v.status = \texttt{DEAD}) \\ &\wedge \ (\forall v \in V : v.deg_{out} = |\{u \in v.post \mid u.status = \texttt{LIVE} \vee u \in Q\}|) \end{aligned}$$

where $Q.pre = (\cup\, q \in Q : q.pre)$. Since the while-loop terminates only when $Q = \emptyset$, it follows that the invariant of the outer for-loop is preserved. We now argue that the while-loop preserves this invariant:

- $sound(V)$ is preserved as $v \in V$ is set to `DEAD` only if $v.deg_{out} = 0$, which implies that there cannot be `LIVE` vertices in $v.post$.
- $complete(V' \setminus Q.pre))$ is preserved as $v \in V' \setminus Q.pre$ is completed vertices and v is indeed set to `DEAD` if $v.deg_{out} = 0$, which implies that there cannot be `LIVE` vertices in $v.post$.
- $\forall v \in Q : v.status = \texttt{DEAD}$ is preserved as v is added to Q only after v is set to `DEAD`.
- $\forall v \in V : v.deg_{out} = |\{u \in v.post \mid u.status = \texttt{LIVE} \vee u \in Q\}|$ is preserved as $v.deg_{out}$ is initialized as the number of available out-going edges and decremented only when removed successors propagated.

At the termination of the outer for-loop (lines 2 - 8), we get $Q = \emptyset$ and $V' = V$. The postcondition of the outer for-loop is $sound(V) \wedge complete(V)$. □

Complexities.

Theorem 3.7.2. *The worst-case time complexity of Algorithm 7 is $O(n+m)$.*

Proof. The out-degree counter $v.deg_{out}$ for all vertices can be initially calculated within $O(n+m)$ time (line 1) as each edge is traversed once. Each vertex $v \in V$ can be removed and then added into the waiting set Q at most once (lines 10 and 11); each reversed edge in $v(G^T).post$ is traversed at most once (lines 6 - 8). In this case, in lines 2 - 8, we get a running time of $O(n + m)$. Therefore, the total worst-case running time is $O(n + m)$. □

Theorem 3.7.3. *The space complexity of Algorithm 7 is $O(n+m)$.*

Proof. In line 7, the transposed graph $G^T = (V, E^T)$ is used. In this case, in order to generate G^T, the whole graph $G = (V, E)$ must be stored in memory, which requires $O(n+m)$ space. For all vertices $v \in V$, storing $v.deg_{out}$ and $v.status$ uses $O(n)$ space. Therefore, the total used space is $O(n+m)$. □

3.7.2 The Parallel AC-4-Based Algorithm

Algorithm 8 shows the detailed steps of the parallel AC-4-based algorithm. All vertices in V are partitioned into $V_1 \dots V_{\mathcal{P}}$ (line 2) so that $\mathcal{P}$ workers can execute the procedure $\texttt{Trim}_p(V_p)$ in parallel (line 3). Compared with Algorithm 7, there are three refinements. First, each worker $p \in [1...\mathcal{P}]$ has its private waiting set Q_p for propagation (line 6) so that the operations on Q_p do not require to be synchronized. Secondly, the out-degree counter deg_{out} has to be updated by the atomic primitive fetch&add `FAA` since multiple workers may decrease such a counter (line 11). Thirdly, it is possible that $v.deg_{out} = 0$ (in line 13) is detected by multiple workers; we use the atomic primitive `CAS` to set the $v.status$ from `LIVE` to `DEAD` (line 13) and return `TRUE` if successful, which ensures that v is added into a single one waiting set Q_p (line 14).

Algorithm 8: Parallel AC-4-based Graph Trimming

```
   input  : Graph G = (V, E) and its transposed graph G^T = (V, E^T)
   output: Trimmed graph G
 1 for v ∈ V do v.status, v.deg_out ← LIVE, |v.post|
 2 partition V into V_1, ..., V_P
 3 Trim_1(V_1) ‖ ... ‖ Trim_P(V_P)
 4 procedure Trim_p(V_p)
 5 |   for v ∈ V_p with v.status = LIVE do
 6 |   |   Q_p ← ∅; DoDegree_p(v)
 7 |   |   while Q_p ≠ ∅ do
 8 |   |   |   remove a vertex w from Q_p
 9 |   |   |   for v' ∈ w(G^T).post do
10 |   |   |   |   FAA(v'.deg_out, −1)
11 |   |   |   |   DoDegree_p(v', Q_p)
12 procedure DoDegree_p(v, Q_p)
13 |   if v.deg_out = 0 ∧ CAS(v.status, LIVE, DEAD) then
14 |   |   Q_p ← Q_p ∪ {v}
```

Correctness. We show the soundness and completeness together.

Theorem 3.7.4 (Soundness and Completeness)**.** *For any $G = (V, E)$ Algorithm 8 terminates with sound(V) and complete(V).*

Proof. The invariant of the while-loop (lines 4 - 8) in procedure $\texttt{Trim}_p(V_p)$ is the same as that in Algorithm 7 except that it adds one more conjunct. That is, a removed vertex can only be added into a single one Q_p for propagation.

$$\begin{aligned} &sound(V_p) \wedge complete(V_p' \setminus Q_p.pre) \wedge (\forall v \in Q_p : v.status = \texttt{DEAD}) \\ &\quad \wedge (\forall v \in V : v.deg_{out} = |\{u \in v.post \mid u.status = \texttt{LIVE} \vee u \in \cup Q_{1..\mathcal{P}}\}|) \\ &\quad \wedge (\forall i, j \in \{1..\mathcal{P}\} : i \neq j \implies Q_i \cap Q_j = \emptyset) \end{aligned}$$

We now argue that the while-loop preserves this invariant:

- $(\forall v \in V : v.deg_{out} = |\{u \in v.post \mid u.status = \texttt{LIVE} \vee u \in \cup Q_{1..\mathcal{P}}\}|)$ is preserved as $v.deg_{out}$ is off by one atomically when a worker is decreasing.
- $(\forall i, j \in \{1..\mathcal{P}\} : i \neq j \wedge Q_i \cap Q_j = \emptyset)$ is preserved as $v.status$ is set from `LIVE` to `DEAD` by the atomic primitive `CAS` and only when successful, v is added to one Q_p.

The postcondition of line 3 is then $sound(V_1) \wedge complete(V_1) \ldots sound(V_{\mathcal{P}}) \wedge complete(V_{\mathcal{P}})$, which is equivalent to $sound(V) \wedge complete(V)$. □

Complexities.

Theorem 3.7.5. *Algorithm 8 requires $O(n + m)$ expected work, $O(|Q_p|Deg_{in}Deg_{out})$ depth, and thus $O((n + m)/\mathcal{P} + |Q_p|Deg_{in}Deg_{out})$ time complexity.*

Proof. This algorithm has the same framework as Algorithm 7, so the total expected work equals the running time of Algorithm 7, that is $O(n + m)$. The initial for-loop (lines 1) can easily run in parallel within expected depth $O(Deg_{out})$.

We analyze the working depth for the procedure $\texttt{Trim}_p$. For each round of the outer while-loop (lines 7 - 11), it runs with depth $|Q_p|$ which is the upper-bound size of waiting sets among $\mathcal{P}$ workers. As Q_p is private for worker p without synchronization, it is possible that $|Q_p| \geq \alpha$. The most inner for-loop (line 9) runs sequentially with depth $O(Deg_{in})$, and the out-degree counters have to concurrently update with depth Deg_{out}. Therefore, the total working depth is $O(\alpha Deg_{in}Deg_{out})$ and thus the worst-case time complexity is $O((n + m)/\mathcal{P} + \alpha Deg_{in}Deg_{out})$. □

Theorem 3.7.6. *The space complexity of Algorithm 8 is $O(n + m)$.*

Proof. By using the atomic primitive `CAS` in line 12, each vertex $v \in V$ may be removed at most once and then put into at most single one waiting set Q_p, so all waiting set for $\mathcal{P}$ workers require $O(n)$ space. Similar to Algorithm 7, storing deg_{out} and *status* requires $O(n)$ space and the reverse edges require $O(n+m)$ space (line 8). Therefore, the total used space is $O(n+m)$. □

3.8 AC-6-Based Graph Trimming

As mentioned, AC-4 has better worst-case time complexity than AC-3, but AC-4 always has a worse average running time than AC-3. Additionally, AC-4 does not have the on-the-fly property. AC-6 improves AC-4 by only recording one support for each value since one support is enough to guarantee that a value is viable. In this case, compared with AC-4, AC-6 performs better in many applications, requires less space usage, and has the on-the-fly property.

To the best of our knowledge, we are the first to introduce AC-6 to graph trimming and call it the AC-6-based algorithm. The idea is novel: 1) each vertex v maintains a set of vertices $v.S$ that choose v as an available outgoing edge; 2) when removing v as it has no outgoing edges, each vertex $w \in v.S$ has to find another available outgoing edge to replace v; otherwise, w has to be removed; 3) this process repeats until no vertices can be removed. In this section, we propose new sequential and parallel AC-6-based algorithms for graph trimming, which is the main contribution of this work.

3.8.1 The Sequential AC-6-Based Algorithm

Analogous to AC-6, the AC-6-based trimming algorithm is based on the concept of *support*. That is, for each vertex v in a given directed graph G, the support of v is one of v's available outgoing edges and v cannot be removed if v's support exists. One auxiliary data structure, the supporting set, is needed to store all the supports for propagation, which is formally defined below.

Definition 3.8.1 (Supporting Set)**.** Given a directed graph $G = (V, E)$, for a vertex $v \in V$, the supporting set $v.S$ of v is the set of predecessors that choose v as their single one support: $(\forall v \in V : v.S \subseteq \{u \in v.pre \mid u.status = \texttt{LIVE}\}) \wedge (\forall v \in V : v.S \neq \emptyset \implies v.status = \texttt{LIVE}) \wedge (\forall u, v \in V : u \neq v \implies u.S \cap v.S = \emptyset)$.

In other words, $v.S$ records all `LIVE` vertices that have v as their support. Absolutely, v must be `LIVE` if the vertices in $v.S$ choose v as a support as an existing support has to be an available outgoing edge; a vertex can be added into at most one supporting set as each vertex only needs to maintain single one support.

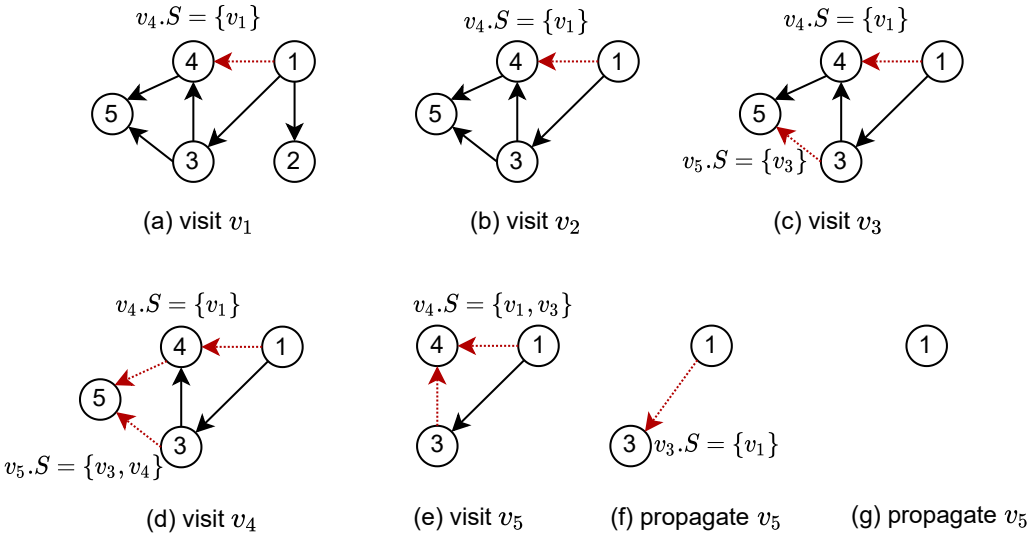


Figure 3.2: Steps of the sequential AC-6-based trimming algorithm based on part of the graph in Figure 3.1.

Figure 3.2 illustrates the AC-6-based algorithm based on the part of the example graph in Figure 3.1. The dashed red arrows are the edges visited to find available supports and then added to the corresponding supporting set $v.S$. Each vertex is successively visited from v_1 to v_5 as shown in Figure 3.2 (a) to (e). In Figure 3.2 (a), v_1 is visited and its first support v_4 is found with vertex v_1 added into $v_4.S$. In Figure 3.2 (b), v_2 is removed since v_2 has no outgoing edges; no propagation happens as $v_2.S$ is empty. In Figure 3.2 (c), v_3 is visited and its first support v_5 is found with v_3 added into $v_5.S$. In Figure 3.2 (d), v_4 is visited and its first support v_5 is found with v_3 added into $v_4.S$. In Figure 3.2 (e), v_5 is removed as it cannot find any support; since $v_5.S$ includes v_3 and v_4 the propagation happens as follow: v_3 finds a next available support v_4 with v_3 added into $v_4.S$, and the v_4 is failed to find a next available support so that v_4 should be removed in the next step. In Figure 3.2 (f), v_4 is removed; since the supporting set $v_4.S$ includes vertices v_1 and v_3 the propagation happens as follow: v_1 finds a next available support, v_3, which is added into $v_4.S$, and v_3 fails to find a next available support so that v_3 should be removed in the next step. In Figure 3.2 (g), the vertex v_3 is removed and $v_1 \in v_3.S$ should be further propagated. Finally, v_1 should also be removed as it has no outgoing edges. As we can see, AC-6-based trimming can remove some of the vertices without propagation, e.g. v_2.

Algorithm 9 shows the detailed steps of the sequential AC-6-based algorithm. For each vertex v in the graph, a supporting set $v.S$ is required for recording the vertices that choose v as an available support. We first consider

Algorithm 9: Sequential AC-6-based Graph Trimming

```
   input  : Graph G = (V, E)
   output: Trimmed graph G
 1 for v ∈ V do v.status, v.S ← LIVE, ∅
 2 for v ∈ V do
 3 |   Q ← ∅; DoPost(v)
 4 |   while Q ≠ ∅ do
 5 |   |   remove a vertex w from Q
 6 |   |   for v' ∈ w.S do
 7 |   |   |   w.S ← w.S \ {v'}
 8 |   |   |   DoPost(v')
 9 procedure DoPost(v)
10 |   for w ∈ v.post with w.status = LIVE do
11 |   |   w.S ← w.S ∪ {v}; v.post ← v.post \ {w}; return
12 |   v.status ← DEAD; Q ← Q ∪ {v}
```

the procedure `DoPost`(v) (lines 9 - 12). If v successfully finds a live successor w, then w is added to $v.S$ and the procedure finishes (lines 10 and 11). Otherwise, v has to be removed from the graph as v has no available outgoing edges, and v is set to `DEAD` and then put into the waiting set Q (line 12). Note that, the visited vertex w is removed from $v.post$ to avoid redundant checking (line 11), which can ensure that each edge is visited at most once. Now we explain the main algorithm (lines 1 - 8). Initially, all vertices are `LIVE` and their supporting sets are empty (line 1). For each vertex $v \in V$, the support $v.s$ is checked by the procedure `DoPost`(v) (line 3) and the removed vertices are added into Q for propagation (lines 4 - 8). That is, a vertex $w \in Q$ is removed from Q (line 5) and for all the vertices in $w.S$ are checked by the procedure `DoPost`(v') (lines 6 - 8). This propagation is repeated until Q is empty (line 4), as vertices may be removed and added into Q by the procedure `DoPost`(v') (line 8).

Correctness. We show the soundness and completeness together.

Theorem 3.8.1 (Soundness and Completeness)**.** *For any $G = (V, E)$ Algorithm 9 terminates with sound(V) and complete(V).*

Proof. Let V' be the set of vertices visited by the outer for-loop (lines 2 - 8). The invariant of the outer for-loop is that all vertices are sound, all visited vertices are complete, all visited `LIVE` vertices must have a support, and that for each vertex $v \in V$ the supporting set $v.S$ includes all visited vertices that

choose v as their single one support:

$$\begin{aligned} & \mathit{sound}(V) \ \wedge \ \mathit{complete}(V') \ \wedge \ (\forall v \in V' : v.\mathit{status} = \texttt{LIVE} \implies v \in S) \\ & \wedge \ (\forall v \in V : v.S \subseteq \{u \in v.\mathit{pre} \mid u.\mathit{status} = \texttt{LIVE} \wedge u \in V'\}) \\ & \wedge \ (\forall v \in V : v.S \neq \emptyset \implies v.\mathit{status} = \texttt{LIVE}) \\ & \wedge \ (\forall u, v \in V : u \neq v \implies u.S \cap v.S = \emptyset) \end{aligned}$$

where $S = (\cup v \in V : v.S)$. The invariant holds initially as setting all vertices to `LIVE` and V' is empty.

The invariant of the while-loop (lines 4 - 8) is that all states are sound but setting a vertex to `DEAD` may lead to its predecessors to be incomplete; also, all vertices $w \in Q$ are set to `DEAD` and all vertices in $w.S$ have to update their support.

$$\begin{aligned} & \mathit{sound}(V) \ \wedge \ \mathit{complete}(V' \setminus Q.S) \ \wedge \ (\forall v \in Q : v.\mathit{status} = \texttt{DEAD}) \\ & \wedge \ (\forall v \in V' : v.\mathit{status} = \texttt{LIVE} \implies v \in S) \\ & \wedge \ (\forall v \in V : v.S \subseteq \{u \in v.\mathit{pre} \mid u.\mathit{status} = \texttt{LIVE} \wedge u \in V'\}) \\ & \wedge \ (\forall v \in V : v.S \neq \emptyset \implies v.\mathit{status} = \texttt{LIVE} \vee v \in Q) \\ & \wedge \ (\forall u, v \in V : u \neq v \implies u.S \cap v.S = \emptyset) \end{aligned}$$

where $S = (\cup v \in V : v.S)$ and $Q.S = (\cup q \in Q : q.S)$. Since the while-loop terminates only when $Q = \emptyset$, it follows that the invariant of the outer for-loop is preserved.

We now argue that the while-loop preserves this invariant:

- $\mathit{sound}(v)$ is preserved as $v \in V$ is set to `DEAD` if w cannot find a support in $v.\mathit{post}$, which implies that there cannot be `LIVE` vertices in $v.\mathit{post}$.
- $\mathit{complete}(V' \setminus Q.S)$ is preserved as $v \in V' \setminus Q.S$ is indeed set to `DEAD` if the support of v not exists, which implies that there cannot be `LIVE` vertices in $v'.\mathit{post}$.
- $\forall v \in Q : v.\mathit{status} = \texttt{DEAD}$ is preserved as v is added to Q only after v is set to `DEAD`.
- $\forall v \in V' : v.\mathit{status} = \texttt{LIVE} \implies v \in S$ is preserved as v has to find a support after being visited if v is `LIVE`.
- $\forall v \in V : v.S \subseteq \{u \in v.\mathit{pre} \mid u.\mathit{status} = \texttt{LIVE} \wedge u \in V'\}$ is preserved as the visited vertices $u \in V'$ is `LIVE` when choosing v as a support.
- $\forall v \in V : v.S \neq \emptyset \implies v.\mathit{status} = \texttt{LIVE} \vee \wedge v \in Q$ is preserved as new vertices can be added into $v.S$ if v is `LIVE`, and after setting v to `DEAD` and adding into Q all vertices in $v.S$ will find next available support.

– $\forall u, v \in V : u \neq v \implies u.S \cap v.S = \emptyset$ is preserved as each vertex maintains at most single one support.

At termination of the outer for-loop (lines 2 - 8), we get $Q = \emptyset$ and $V' = V$. The postcondition of the outer for-loop is $sound(V) \wedge complete(V)$. □

Complexities.

Theorem 3.8.2. *The time complexity of the Algorithm 9 is $O(n + m)$.*

Proof. Each vertex $v \in V$ can be removed and then added into the waiting set Q at most once. In the procedure `DoPost(`v`)`, each outgoing edge of v is traversed at most once to find the support (lines 10 and 11) as the visited vertex w is removed from $v.post$ to avoid repetitive visiting. In this case, The most inner for-loop (lines 6 - 8) calls procedure `DoPost(`v`)` to find a support for vertex v. Therefore, with this assumption, the worst-case time complexity is $O(n + m)$. □

Theorem 3.8.3. *The space complexity of the Algorithm 9 is $O(n)$.*

Proof. The global waiting set Q has a maximum size of $O(n)$ as each vertex $v \in V$ can be set to `DEAD` and added into Q at most once. The supporting sets have the total size at most $O(n)$ as each vertex $v \in V$ has at most one support recorded in a corresponding supporting set. Obviously, *status* requires $O(n)$ space. Therefore, the worst-case space complexity is $O(n)$. □

3.8.2 The Parallel AC-6-Based Algorithm

Algorithm 10 shows the detailed steps of the parallel AC-6-based trimming algorithm. Compared with the sequential AC-6-based trimming in Algorithm 9, there are two refinements. First, each worker $p \in [1 \ldots \mathcal{P}]$ has its private waiting set Q_p for propagation so that the synchronization on Q_p is unnecessary (lines 6, 8, and 19). Secondly, the supporting set $w.S$ for each vertex $w \in V$ can concurrently have new vertices added by multiple workers synchronized by locking (lines 14 - 17). When adding vertices to $w.S$, vertex w has to be `LIVE` (line 15) as no vertices can be added to $w.S$ after setting w to `DEAD`. When setting vertex v to `DEAD`, we have to lock v to ensure that no other workers are adding vertices to $v.S$; otherwise, after v is added into Q and propagated (lines 19 and 8 - 11), other workers still have possibility to add vertices into $v.S$ which can never be propagated. In other words, we lock $v.S$ when setting v from `LIVE` to `DEAD` to ensure that all vertices in $v.S$ are propagated together. Note that, when removing vertices from $w.S$ (line 10), it is unnecessary to lock $w.S$ as currently w is `DEAD` so that no workers can add vertices into $w.S$ and w is only accessed by a single worker, p.

We implement the lock by the `CAS` primitive with busy waiting (lines 20 and 21). Here, the busy waiting is suitable as there are at most two operations within locking (lines 15 and 16) so that the expected waiting time is really short.

Algorithm 10: Parallel AC-6-based Graph Trimming

```
   input  : G = (V, E)
   output: Trimmed graph G
 1 for v ∈ V do v.S, v.status ← ∅, LIVE; Unlock(v)
 2 partition V into V_1, ..., V_P
 3 Trim_1(V_1) ‖ ... ‖ Trim_P(V_P)
 4 procedure Trim_p(V_p)
 5 |  for v ∈ V_p do
 6 |  |  Q_p = ∅; DoPost_p(v, Q_p)
 7 |  |  while Q_p ≠ ∅ do
 8 |  |  |  remove a vertex w from Q_p
 9 |  |  |  for v' ∈ w.S do
10 |  |  |  |  w.S ← w.S \ {v'}
11 |  |  |  |  DoPost_p(v', Q_p)
12 procedure DoPost_p(v, Q_p)
13 |  for w ∈ v.post with w.status = LIVE do
14 |  |  Lock(w)
15 |  |  if w.status = LIVE then
16 |  |  |  w.S ← w.S ∪ {v}; Unlock(w); return
17 |  |  Unlock(w)
18 |  Lock(v); v.status ← DEAD; Unlock(v)
19 |  Q_p ← Q_p ∪ {v}
20 procedure Lock(w)
21 |  while ¬ CAS(w.lock, FALSE, TRUE) do skip
22 procedure Unlock(w)
23 |  w.lock ← FALSE
```

Correctness.

Theorem 3.8.4 (Soundness and Completeness of Parallel AC-6-based Trimming)**.** *For any $G = (V, E)$ Algorithm 10 terminates with sound(V) and complete(V).*

Proof. The invariant of the while-loop (lines 4 - 8) in procedure $\texttt{Trim}_p(V_p)$ is the same as that in Algorithm 7 except that it adds one more conjunct. That

is, a removed vertex can only be added into a single one Q_p for propagation.

$$\begin{aligned} & \mathit{sound}(V_p) \ \wedge \ \mathit{complete}(V'_p \setminus Q_p.S) \ \wedge \ (\forall v \in Q_p : v.\mathit{status} = \texttt{DEAD}) \\ & \wedge \ (\forall v \in V'_p : v.\mathit{status} = \texttt{LIVE} \implies v \in S) \\ & \wedge \ (\forall v \in V_p : v.S \subseteq \{u \in v.\mathit{pre} \mid u.\mathit{status} = \texttt{LIVE} \wedge u \in V'\}) \\ & \wedge \ (\forall v \in V_p : v.S \neq \emptyset \implies v.\mathit{status} = \texttt{LIVE} \vee v \in Q_p) \\ & \wedge \ (\forall u, v \in V_p : u \neq v \implies u.S \cap v.S = \emptyset) \\ & \wedge \ (\forall i, j \in \{1..\mathcal{P}\} : i \neq j \implies Q_i \cap Q_j = \emptyset) \end{aligned}$$

where $S = (\cup v \in V : v.S)$, $Q_p.S = (\cup q \in Q_p : q.S)$, and $V' = \cup V'_{1..\mathcal{P}}$. In the algorithm, for each vertex v, multiple workers add new vertices into $v.S$ concurrently, and during this time v cannot be set to `DEAD`. We now argue that the while-loop preserves this invariant:

- $\forall v \in V_p : v.S \subseteq \{u \in v.\mathit{pre} \mid u.\mathit{status} = \texttt{LIVE} \wedge u \in V'\}$ is preserved as v is locked when a worker is adding new vertices to $v.S$.
- $\forall v \in V_p : v.S \neq \emptyset \implies v.\mathit{status} = \texttt{LIVE} \vee (v.\mathit{status} = \texttt{DEAD} \wedge v \in Q_p)$ is preserved as 1) v is locked when the worker p setting v to `DEAD` to ensure that after setting v to `DEAD` no vertices can be added into $v.S$ by other workers, and 2) only the current worker p can set v to `DEAD` and add v to the private set Q_p.
- $\forall i, j \in \{1..\mathcal{P}\} : i \neq j \ \wedge \ Q_i \cap Q_j = \emptyset$ is preserved as only single one worker p can add v to Q_p after setting v to `DEAD`.

At the termination of outer for-loop (lines 2 - 8), we get $Q_p = \emptyset$ and $V'_p = V_p$. The postcondition of line 3 is then $\mathit{sound}(V_1) \wedge \mathit{complete}(V_1) \wedge \ldots \wedge \mathit{sound}(V_\mathcal{P}) \wedge \mathit{complete}(V_\mathcal{P})$, which is equivalent to $\mathit{sound}(V) \wedge \mathit{complete}(V)$. □

Complexities.

Theorem 3.8.5. *The Algorithm 10 requires $O(n + m)$ expected work, $O(|Q_p|{Deg_{in}}^2)$ depth, and $O((n + m)/\mathcal{P} + |Q_p|{Deg_{in}}^2)$ time complexity.*

Proof. This algorithm has the same framework as Algorithm 9, so the total expected work equals the running time of Algorithm 9, that is $O(n+m)$. The initial for-loop (lines 1) can run in parallel within expected depth $O(1)$.

We analyze the working depth for procedure $\texttt{Trim}_p$. For each round of the outer while-loop (lines 7 - 11), it runs with depth $|Q_p|$ which is the upper-bound size of waiting sets among $\mathcal{P}$ workers. As Q_p is private for a worker p without synchronization, it is possible that $|Q_p| \geq \alpha$. The most-inner for-loop (line 9) runs sequentially with depth Deg_{in} as Deg_{in} is the upper-bound size for a supporting set. The supporting sets concurrently add new vertices

with depth Deg_{in} (line 16). The locking operation (lines 14 and 18) needs to busy-check by the `CAS` primitive only a few times with high probability as there are at most two operations within the lock. Therefore, the total working depth is $O(|Q|{Deg_{in}}^2)$ with high probability and the worst-case running time is $O((n+m)/\mathcal{P} + |Q|{Deg_{in}}^2)$ with high probability. □

Theorem 3.8.6. *The space complexity of Algorithm 10 is $O(n)$, which equals to its sequential version Algorithm 9.*

Proof. Each vertex $v \in V$ may be removed at most once and then put into at most one waiting set Q_p, so all waiting sets require $O(n)$ space. Each vertex has at most one support which is stored into the corresponding supporting set and thus the total size of all supporting sets is $O(n)$. The status for each vertex $v \in V$ has the size of $O(n)$. Therefore, the total space complexity is $O(n)$. □

3.9 Implementation

Parallelism. For simplicity, our parallel trimming algorithms sacrifice some parallelism. That is, the most inner for-loop can run in parallel (lines 12 - 13 in Algorithm 5, lines 9 - 11 in Algorithm 8, and lines 9 -11 in Algorithm 10); the private waiting set Q_p for a worker p can be balanced in Algorithm 8 and Algorithm 10) so that Q_p has at most α vertices. As shown in Table 3.3, the working depth can be improved if we achieve full parallelism. However, the scheduler will be challenged to parallel inside each worker p efficiently. One solution is to maintain a *frontier* (subset) of all affected vertices, and in each step all vertices in a frontier can be processed in parallel (Defo et al., 2019).

Trimming	Worst-Case (O) Work	Depth	Space
AC-3-based	$\alpha(n+m)$	α	n
AC-4-based	$n+m$	αDeg_{out}	$n+m$
AC-6-based	$n+m$	αDeg_{in}	n

Table 3.3: The worst-case work, depth, time, and space complexities of full parallelized graph trimming algorithms.

In practice, our algorithms can be highly parallelized. There are two reasons. First, most real graphs always have millions of edges, and $|Q|$, Deg_{in}, and Deg_{out} are relatively much smaller than $n+m$. Secondly, multi-core machines always have a limited number of workers, e.g. $\mathcal{P} = 32$. Therefore, our trimming algorithms can achieve a load balance among multiple workers with high probability.

Traverse Edges. Since the edges are linearly stored in memory, we can optimize the implementation of trimming algorithms. In the procedure `ZeroOutDegree` of Algorithm 6, for vertex v only the first `LIVE` edge needs to be found. In this case, each vertex can maintain an index *edge_index* to record the position of visited edges. In the next round, we can "jump" over the edges that have already visited. By doing this, we can reduce the number of traversed edges to a certain degree. Similarly, we can apply this strategy to the procedure `DoPost` of Algorithm 9 and Algorithm 10; by doing this, each edge can be traversed at most once.

Cache-Friendliness. For multi-core architectures, contiguous memory accessing is much faster than random memory accessing because of the possibility of pre-fetching by L1, L2, and L3 caches. Of course, accessing cache is faster than accessing the memory by an order of magnitude. For explicit graphs stored in memory, the cache can affect the running time by an order of magnitude. A *cache-friendly* program has a large portion of contiguous memory accessing that can fully utilize the cache to obtain speedup. In contrast, a *cache-unfriendly* program has a large portion of random memory accessing that can not efficiently utilize the cache.

Since all edges are stored in contiguous memory as CSR format for a tested graph, we compare the cache property of three different graph trimming methods together as follow:

- AC-3-based Graph Trimming is cache-friendly as all edges are stored in an array and can be traversed sequentially with a high cache hit rate.
- AC-4-based Graph Trimming is less cache-friendly as each vertex v are traversed almost randomly, but v's edges are traversed sequentially with a medium cache hit rate.
- AC-6-based Graph Trimming is least cache-friendly as for each vertex v, both v and v's edges are traversed almost randomly with low cache hit rate.

Memory Usage. We compare the practical memory usage in Table 3.4. Assume that storing a vertex or an integer takes H bits. All three algorithms require 1 bit for the status of each vertex, in total n bits. For both *AC4Trim* and *AC6Trim*, there are $\mathcal{P}$ waiting sets $Q_1 \dots Q_{\mathcal{P}}$, in total nH bits, since each vertex can be put into Q_p at most once. For *AC4Trim*, a reversed graph has to be loaded into memory, in total $(n+m)H$ bits; each vertex maintains an out-degree counter deg_{out}, in total nH bits. For *AC6Trim*, each vertex has a supporting set S, in total nH bits, since each vertex can be put into a set S

at most once. For *AC3Trim* and *AC6Trim*, each vertex maintains an index *edge_index* to "jump" over the visited edges, in total nH bits.

AC3-based	*AC4-based*	*AC6-based*
bit$[n]$: $\forall v.status$	bit$[n]$: $\forall v.status$	bit$[n]$: $\forall v.status$
bit$[nH]$:$\forall v.edge_index$	bit$[nH]$: $\forall v.deg_{out}$	bit$[n]$: $\forall v.lock$
	bit$[nH]$: $Q_1 \dots Q_{\mathcal{P}}$	bit$[nH]$:$\forall v.edge_index$
	bit$[(n+m)H]$: G^T	bit$[nH]$: $\forall v.S$
		bit$[nH]$: $Q_1 \dots Q_{\mathcal{P}}$

Table 3.4: Compare the memory usage for *AC3Trim*, *AC4Trim* and *AC6Trim*, where storing a vertex takes H bits.

3.10 Experiments

In this section, we evaluate three different parallel algorithms for graph trimming:

- the AC-3-based trimming algorithm (Hong et al., 2013; Ji et al., 2018) (*AC3Trim* for short),
- the AC-4-based trimming algorithm (III et al., 2005) (*AC4Trim* for short),
- the AC-6-based trimming algorithm (*AC6Trim* for short).

The experiments are performed on a server with an AMD CPU (16 cores, 32 hyper-threads, 32 MB of last-level cache) and 96 GB main memory. The server runs the Ubuntu Linux (18.04) operating system. All tested algorithms are implemented in C++ and compiled with g++ version 7.3.0 with the -O3 option [1]. OpenMP (Dagum and Menon, 1998) version 4.5 is used as the threading library. We perform every experiment at least 50 times (at least 10 times for time-consuming experiments) and calculate their means with 95% confidence intervals.

We first give in total 15 real and synthetic benchmark graphs. Before the evaluation, we discuss the workload balance. Then, over these tested graphs, we evaluate the number of traversed edges and then compare the real running times by varying the workers from 1 to 32. We also evaluate the stability and scalability by using 16 workers.

3.10.1 Graph Benchmarks

We evaluate the performance of our method on a variety of model checking, real-world, and synthetic graphs shown in Table **??**.

- The *cambridge.6*, *bakery.6* and *leader-filters.7* graphs come from the model checking problems in the BEEM database (Pelánek, 2007), which are implicit and can be generated on-the-fly. For convenience, these graphs are converted to explicit graphs (Bloemen et al., 2016) and stored in files.
- The *livej*, *patent*, and *wikitalk* graphs are obtained from SNAP (**?**) [2]; they represent the Live-Journal social network (**?**), the U.S patent dataset is maintained by the National Bureau of Economic Research (**?**),

[1] All our implementations, benchmarks, and results are available at `https://github.com/Itisben/graph-trimming.git`

[2] `https://snap.stanford.edu`

and Wikipedia Talk (communication) network (**?**), respectively. The *dbpedia*, *baidu*, and *wiki-talk-en* graphs are collected from the University of Koblenz-Landau (**?**); they represent the DBpedia network (Auer et al., 2007), the hyperlink network between the articles of the Baidu (**?**) encyclopedia, and the communication network of the English Wikipedia (Sun et al., 2016), respectively.

- The *com-friendster*, *twitter* and *twitter-mpi* are three super large graphs with billions of edges obtained from the Network Repository (**?**) [3]; they represent the online gaming social network (**?**), the follower network from Twitter (**?**), the Twitter follow data collected in 2009 (**?**), respectively.

- The *ER*, *BA*, and *RMAT* graphs are synthetic graphs; they are generated by the SNAP (**?**) system using the Erdös-Rényi graph model (which generates a random graph), the Barabasi-Albert graph model (which generates a graph with power-law degree distribution), and the R-MAT graph model (which generates large-scale realistic graph similar to social networks), respectively; for these generated graphs, the average degree is fixed to 8 by choosing $1,000,000$ vertices and $8,000,000$ edges.

All these graphs are stored in the Compressed Sparse Row (CSR) binary format (Hong et al., 2012, 2013), which is compact and memory bandwidth-friendly. Taking the super large graph *twitter* for example, the text file that includes all edges requires 30 GB while the CSR binary format only requires 6 GB.

Table **??** provides an overview of the 15 tested graphs. For some graphs, e.g. *cambridege.6* and *ER*, less than 1% of vertices can be trimmed. However, for most of the other graphs, a high ratio of vertices can be trimmed, especially for *leader-filters.7*, *BA*, and *com-friendster*, whose vertices can be trimmed altogether. More importantly, for most graphs, the trimming steps α, maximum in-degree Deg_{in}, and maximum out-degree Deg_{out} are always small. When analyzing the parallel time complexity, these three values are associated with the parallel depths. The small values of the depths indicate that the execution can be highly parallelized (Blelloch and Maggs, 2010).

3.10.2 Workload Balance

Given a tested graph, all vertices are partitioned into multiple chunks, which can be dynamically assigned to workers for workload balance. One issue is how to determine the size of chunks. A large size of chunks may lead to workload imbalance, while a small size of chunks may lead to a high cost of scheduling. Since the trend is similar for all tested graphs, we select three typical graphs

[3] `http://networkrepository.com`

Name	$\|V\|$	$\|E\|$	Deg_{in}	Deg_{out}	α	%Trim
cambridge.6	3.3M	9.5M	15	6	65	0.25%
bakery.6	11.8M	40.4M	24	4	47	22.26%
leader-filters.7	26.3M	91.7M	12	6	73	100.00%
dbpedia	4.0M	13.8M	473.0K	1.0K	116	36.23%
baidu	2.1M	17.8M	98.0K	2.6	9	27.97%
livej	4.8M	69.0M	14.0K	20.3K	8	12.23%
patent	6.0M	16.5M	779	770	5	100.00%
wiki-talk-en	3.0M	25.0M	121.3K	488.2K	7	87.42%
wikitalk	2.4M	5.0M	3.3K	100.0K	5	94.49%
com-friendster	125M	1.8B	4.2K	3.6K	11.7K	100.00%
twitter	41.4B	1.4B	770.2K	3.9M	6	10.05%
twitter-mpi	52.6B	2.0B	3.5M	780.0K	7	17.52%
ER	1.0M	8.0M	25	24	3	0.03%
BA	1.0M	8.0M	8	5.2K	122	100%
RMAT	1.0M	8.0M	335	1.9K	7.0K	99.98%

Table 3.5: The characteristics for model checking, real-world, and synthetic graphs. Here, columns denote the number of vertices n, the number of edges m, the maximum in-degree, the maximum out-degree, the number of peeling steps, and the percentage of trimmable vertices, respectively.

with a variety of Deg_{in}, Deg_{out}, and α for the evaluation. In Figure **??**, we test three trimming algorithms over three selected graphs, *leader-filters.7*, *livej*, and *wiki-talk-en*, that have millions of vertices, by using 16 workers and varying the chunk size from 1 to 2^{20}. All three trimming algorithms tend to be efficient when choosing a chunk size between 2^{10} and 2^{16}. Therefore, in our experiments, we fix the chunk size to $2^{12} = 4096$ for both workload balance and efficient scheduling.

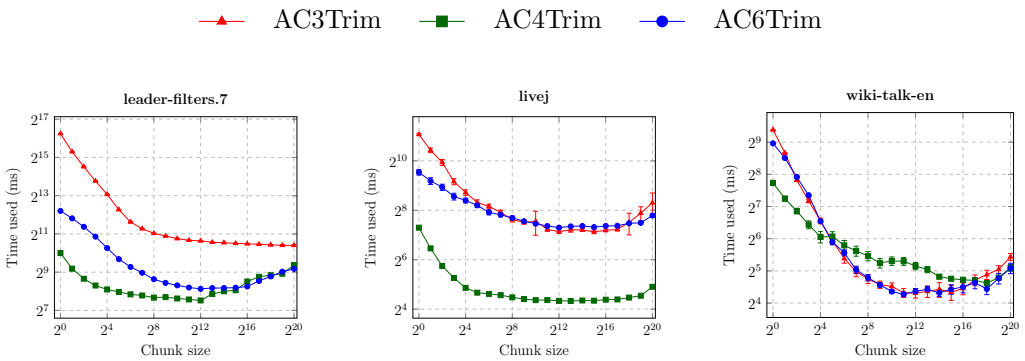


Figure 3.3: The practical running time for *AC3Trim, AC4Trim* and *AC6Trim* with 16 workers by varying the chunk size.

The other issue is the upper-bound size of the waiting set Q_p for each worker p in AC4Trim and *AC6Trim*. Here, Q_p is private to worker p, and

thus the vertices in Q_p are processed sequentially by worker p without synchronization. A large size of Q_p may lead to workload imbalance. In Table **??**, over all tested graphs we show the upper-bound size of Q_p, denoted as $|Q_p|$, for *AC4Trim* and *AC6Trim* by using 16 workers. We can see that $|Q_p|$ is relatively small compared with millions of vertices. Further, over all tested graphs *AC6Trim* has $|Q_p|$ bounded by 900, while *AC4Trim* has $|Q_p|$ up to 20761. Especially, *AC6Trim* has much smaller values of $|Q_p|$ than *AC4Trim* for graphs like *dbpedia* and *twitter-mpi*. That means that *AC6Trim* on average has better workload balance than *AC4Trim*.

Name	*AC4Trim* $\|Q_p\|$	*AC6Trim* $\|Q_p\|$
cambridge.6	17	6
bakery.6	21	52
leader-filters.7	16	103
dbpedia	**20761**	**852**
baidu	439	108
livej	274	8
patent	95	55
wiki-talk-en	84	5
wikitalk	33	7
com-friendster	646	677
twitter	293	287
twitter-mpi	15217	327
ER	**1**	**1**
BA	66	27
RMAT	299	411

Table 3.6: The upper-bound size of Q_p for *AC4Trim* and *AC6Trim* by using 16 workers. The best and worst cases are in **bold** for each column.

3.10.3 Evaluating the Number of Traversed Edges

To evaluate the Arc-Consistency algorithms, the traditional approach is to count the total number of checked constraints. For each constraint check, a pair of values in the domain $D(X_i)$ and $D(X_j)$ is checked. Such an evaluation is reasonable because 1) most of the running time is spent on checking numerous constraints, 2) the time used for checking each constraint significantly varies for different kinds of arc-consistency problems. Analogous to evaluating Arc-Consistency algorithms, we compare the total number of traversed edges of three trimming algorithms. This is especially meaningful for the implicit graphs since their edges are generated on-the-fly, costing most of the running time.

In this experiment, we exponentially increase the number of workers from 1 to 32 and count the largest number of traversed edges per worker over graphs

in Table **??**. The plots in Figure **??** depict the maximum number of traversed edges per worker for the three compared methods. The x-axis is the number of workers and the y-axis is the number of traversed edges. Also, we choose the total number of edges in a graph as a baseline (denoted as m). Note that, for the AC-4-based trimming algorithm, the out-degree counter of each vertex v in graph is initialized as $v.deg_{out} = |v.post|$. To calculate $v.deg_{out}$, there are two cases: 1) we can traverse all v's successors one by one to count the total numbers of successors (denoted as *AC4Trim*), which means all v's edges of are traversed once; 2) if v' successors are stored successively, we can take the difference between the index of v's first successor and v's last successor without traversing the edges (denoted as *AC4Trim**), which means only v is traversed once and all v's edges are not traversed. Absolutely, *AC4Trim* traverses a higher number of edges than *AC4Trim**.

In Figure **??**, a first look over nearly all testing graphs reveals that the number of traversed edges of all three algorithms is linearly decreasing with an increasing number of workers, which achieves a good load balance. Over most of the testing graphs, *AC6Trim* traverses fewer edges compared with *AC4Trim* and *AC3Trim*. *AC3Trim* sometimes traverses more edges than the baseline m for some graphs with large α. Specifically, we make four observations:

- Over graphs with a higher value of α, e.g. *cambridge.6, bakery.6, leader-filter.7, dbpedia, com-friendster* and *RMAT*, *AC3Trim* always traverses much more edges than both *AC4Trim* and *AC6Trim* and even more than the baseline m. This is because *AC3Trim* has the worst work complexity $\mathcal{O}(\alpha(n+m))$ which requires α number of repetitions, but *AC4Trim* and *AC6Trim* have a linear work complexity of $\mathcal{O}(n+m)$.

- Over the graphs with a lower value of α, e.g. *wiki-talk-en* and *wikitalk*, *AC3Trim* traverses fewer edges than *AC4Trim*. This is because *AC3Trim* executes always close to the best-case time complexity, but *AC4* executes always close to the worst-case time complexity. This is why *AC3Trim* is sometimes more powerful than *AC4Trim* in real-world graphs with a relatively low value of α.

- Over all graphs, *AC6Trim* always traverses much fewer edges than *AC4Trim* even if they have nearly the same time complexity. The reason is that *AC6Trim* can traverse only part of the edges of removed vertices, which is close to the best-case time complexity. However, *AC4Trim* has to traverse all edges to initialize the counters $\forall v \in V : v.deg_{out}$ and all ingoing edges of removed vertices, which is close to the worst-case time complexities. Therefore, *AC6Trim* certainly traverses fewer edges than *AC4Trim*.

- Over all graphs, for all three methods, the number of traversed edges is well bounded without obvious variation even these three methods are non-deterministic. That is, for the number of traversed edges, the affect of non-determinism can be omitted.

Name	1-worker vs 16-worker AC3Trim	AC4Trim	AC6Trim	AC3Trim vs AC6Trim	AC4Trim vs AC6Trim
cambridge.6	13.04	15.97	11.74	**58.29**	2.08
bakery.6	12.90	15.61	12.58	25.44	2.22
leader-filters.7	13.22	15.15	13.23	4.71	1.75
dbpedia	14.13	5.94	10.26	42.43	6.69
baidu	13.49	13.07	11.19	5.79	9.35
livej	13.69	15.90	13.38	7.58	13.51
patent	14.63	8.09	15.07	1.04	3.72
wiki-talk-en	13.33	15.76	13.17	4.87	35.33
wikitalk	10.89	15.55	11.05	4.31	**36.51**
com-friendster	16.78	**2.00**	1.07	5.57	1.00
twitter	14.54	15.92	**14.95**	5.45	33.31
twitter-mpi	13.66	15.87	13.56	5.77	32.00
ER	**20.00**	**16.00**	12.47	1.87	6.24
BA	**5.90**	4.84	1.33	**0.43**	**0.55**
RMAT	10.23	2.07	**1.05**	10.39	1.02

Table 3.7: Compare the ratio for the maximum number of traversed edges per worker. The best and worst cases are in **bold** for each column.

In Table **??**, columns 2 - 4 compare the ratio of the maximum number of traversed edges per worker between using a single worker and using 16 workers for *AC3Trim*, *AC4Trim* and *AC6Trim*, respectively. We can see that for *AC3Trim*, the ratio is at least 5.9 as *AC3Trim* is easy to be parallelized without using locks or atomic primitives. We also can see that for *AC3Trim* the ratio is larger than 16 in some graphs, e.g. *com-friendster* and *ER*. The reason is that parallel *AC3Trim* is non-deterministic; that is, different trimming orders lead to different numbers of traversed edges; if numerous vertices are early determined as `DEAD`, the time complexity is close to the best case. For *AC4Trim* and *AC6Trim*, the ratio is relatively low in some graphs with large α, e.g. *RMAT* and *com-friendster*, as large α always leads to high working depths.

In columns 5 and 6 of Table **??**, we fix using 16 workers and compare the ratio of traversed edge numbers between *AC3Trim* and *AC6Trim* and between *AC4Trim* and *AC6Trim*. We can see that *AC6Trim* traverses much fewer edges than *AC3Trim*, up to 58 times over the graph *cambridge.6*; *AC6Trim* traverses much fewer edges then *AC4Trim*, up to 36 times over the graph *wikitalk*. *AC3Trim* traverses the fewest edges in some graphs, e.g. *BA*, as the time complexity is close to the best case.

3.10.4 Evaluating the Real Running Time

In this experiment, we exponentially increase the number of workers from 1 to 32 and evaluate the real running time over graphs in Table **??**. The plots in Figure **??** depict the performance of the three compared methods. The x-axis is the number of workers and the y-axis is the execution time (millisecond). The first look over all testing graphs reveals that the trends for the running time are much different from the trends for the number of traversed edges shown in Figure **??**. This can be explained as follow:

- Although *AC6Trim* always traverses the fewest numbers of edges, *AC6Trim* is slower than *AC4Trim* in some graphs, e.g. *cambridge.6, livej, pokec* and *ER*, and even *AC3Trim* is the fastest in some graphs, e.g. *BA*. The main reason is that *AC6Trim* is cache-unfriendly while *AC3Trim* and *AC4Trim* are cache-friendly. That means *AC6Trim* cannot fully use caches to archive the best performance even if *AC6Trim* traverse the least number of edges. The other reason is that maintaining the supporting sets in *AC6Trim* costs much more computational time than maintaining the out-degree counters in *AC4Trim*; there is no auxiliary data structure in *AC3Trim* so that no computational time is spent on this part.

- In *AC4Trim*, the running times have a wide variation in certain graphs, e.g. *bakery.6*, *leader-filter.7*, *livej*. The reason is that *AC4Trim* is sometimes less cache-friendly. The unexpected missing cache leads to the performance decreased. However, *AC3Trim* is always cache-friendly and *AC6Trim* is always cache-unfriendly so that their performance is more stable than *AC3Trim*.

- In *AC6Trim*, the running times begin to increase when using more than 4 workers in certain graphs, e.g. *dbpedia* and *baidu*. The reason is that the supporting set shared by multi-worker is synchronized by busy waiting, which leads to contention. At the same time, *AC4Trim* still has not obvious speedup as there is less contention to use atomic primitive updating the out-degree counters. However, *AC3Trim* always has a speedup by multiple workers and even has the best performance with 16 workers in some graphs, e.g. *BA*, as *AC3Trim* has no shared data structures and thus no contention.

In columns 2 - 4 of Table **??** we compare the running time speedup between using one worker and 16 workers for *AC3Trim*, *AC4Trim* and *AC6Trim*, respectively. It is clear that *AC3Trim* achieves the best speedup and *AC4* achieves the worst speedup. This is because of the contention on shared data structures with multiple workers. In columns 5 and 6 of Table **??** we fix using 16 workers and compare the speedups for the running time between *AC6Trim* and *AC3Trim* and between *AC6Trim* and *AC4Trim*. We can see that our *AC6Trim* is up to 24 times faster than *AC3Trim* over *RMAT* and up to 7.8 times faster than *AC4Trim* over *leader-filters.7*. However, in some graphs, *AC4Trim* and *AC3Trim* have better performances than *AC6Trim*.

Name	16-workers speedup vs 1-worker			AC6Trim speedup vs	
	AC3Trim	AC4Trim	AC6Trim	AC3Trim	AC4Trim
cambridge.6	2.42	2.72	2.37	7.15	0.13
bakery.6	2.54	**771.96**	2.87	7.39	0.33
leader-filters.7	**2.04**	58.42	2.32	6.27	0.71
dbpedia	3.31	2.67	**0.84**	4.71	0.34
baidu	3.34	11.38	1.14	0.58	0.33
livej	3.07	3.11	2.36	1.00	0.30
patent	4.71	44.02	**7.11**	1.03	**4.65**
wiki-talk-en	3.45	54.85	3.37	1.02	2.10
wikitalk	3.20	2.28	3.21	0.82	2.12
com-friendster	8.09	1.13	1.28	19.62	1.00
twitter	6.42	4.32	5.80	0.70	0.39
twitter-mpi	5.85	5.79	2.99	0.62	0.18
ER	3.68	3.79	1.86	0.19	**0.12**
BA	5.32	1.94	1.92	**0.18**	0.36
RMAT	**6.62**	**1.09**	1.18	**20.97**	0.88

Table 3.8: Compare the speedups for running times between using 1-worker and 16-worker for *AC3Trim*, *AC4Trim*, and *AC6Trim*, respectively; by fixing with 16 workers, compare the running time speedup between *AC6Trim* and *AC3Trim* and between *AC6Trim* and *AC4Trim*. The best and worst cases are in **bold** for each column.

3.10.5 Evaluating Stability

One issue is the stability of the trimming algorithms when executing the same algorithm multiple times. In this experiment, we compare 50 testing result over three chosen graphs, *leader-filters.7*,*livej*, and *wiki-talk-en*. In Figure **??**, the x-axis of plots is the index of the repeating times. The upper three plots in Figure **??** depict the number of traversed edges for three trimming methods, in which the y-axis is the number of traversed edges. We observe that the number of traversed edges is well bounded for all three trimming methods. The lower three plots in Figure **??** depict the running time for three trimming methods, in which the y-axis is the running time. We observe that *AC4Trim* always has a wider variation than other methods. The reason is that parallel *AC4Trim* is non-deterministic, which means each time the order of removed vertices is different; *AC4Trim* is not always cache-friendly as vertices are not sequentially traversed; there is a high probability that the performance decreases due to the unexpected missing cache. Even *AC3Trim* and *AC6Trim* are also non-deterministic, *AC3Trim* is cache-friendly and *AC6Trim* is cache-unfriendly; cache-friendliness does not always lead to a wide performance variation.

3.10.6 Evaluating Scalability

An issue is the scalability of the trimming algorithms when the size of graphs is varied. In this experiment, we test the scalability of the trimming algorithms over the three largest graphs, i.e., *com-friendster, twitter*, and *twitter-mpi*. Using 16 workers, we vary the number of edges and vertices by randomly sampling at a ratio from 10% to 100%, respectively. By sampling the edges, we simply remove the unsampled edges. By sampling the vertices, we simply set the unsampled vertices to `DEAD`. As shown in Figure **??**, we can see that the smaller ratio of sampling edges or vertices always leads to the higher ratio of trimable vertices, e.g. *twitter* and *twitter-mpi*; but for *com-friendster* all vertices are always trimmable with any ratio of sampling. Especially when sampling 10% edges or vertices, nearly 60% of vertices can be trimmed for *twitter* and *twitter-mpi*, and without sampling less than 20% of vertices can be trimmed. This is result is reasonable as more unsampled edges or vertices will lead to more vertices without out-going edges and thus can be trimmed.

We show the result of sampling edges in Figure **??**, in which the x-axis of plots is the ratio of sampled edges. The upper three plots in Figure **??** depict the maximum number of traversed edges per worker. We observe that the number of traversed edges is generally increasing with the ratio of the sampled edges. Not surprisingly, *AC6Trim* traverses the least number of edges, and *AC3Trim* traverses the highest number of edges. But for *AC3Trim* the number of traversed edges fluctuates when increasing the sampling ratio of edges as the number of peeling steps α may fluctuate with a different sampling ratios of edges. The lower three plots in Figure **??** depict the real running time. We make three observations.

- Over *com-friendster*, *AC6Trim* has the best performance. The reason is that 100% of vertices can be trimmed so that *AC4Trim* accesses all vertices almost randomly. In this case, *AC4Trim* is likely too cache-unfriendly, and the cache can not provide an obvious speedup.

- Over *twitter* and *twitter-mpi*, *AC4Trim* has a wide variation and *AC4Trim* performs worse than *AC6Trim* in most of cases. The reason is that for *AC4Trim* more trimmable vertices lead to the cache being less effective, and sometimes the cache can provide a speedup but sometimes not; but *AC6Trim* is cache-unfriendly, and the cache cannot affect the running time.

- Over *twitter* and *twitter-mpi*, *AC3Trim* always has as good performance as *AC6Trim* even if *AC3Trim* traverse much more edges than *AC6Trim*. The reason is that *AC3Trim* is cache-friendly and achieve a high speedup with caching.

Analogously, we show the result of sampling vertices in Figure **??**, in which the x-axis of plots is the ratio of sampling vertices. There are almost the same trends as shown in Figure **??**. One difference is in upper three plots; that is, over *twitter* and *twitter* we can see *AC4Trim* traverse more edges than

AC6Trim as the the unsampled vertices are set to `DEAD` and their out-degree counters are still calculated. The other difference is in lower three plots; that is, over *twitter* and *twitter* we can see *AC6Trim* performs much better than *AC4Trim*, except when vertices are 100% sampled. In this experiment, for implicit graphs loaded into memory, we can see that *AC6Trim* is most scalable no matter how many vertices are trimmed and how many edges or vertices are sampled.

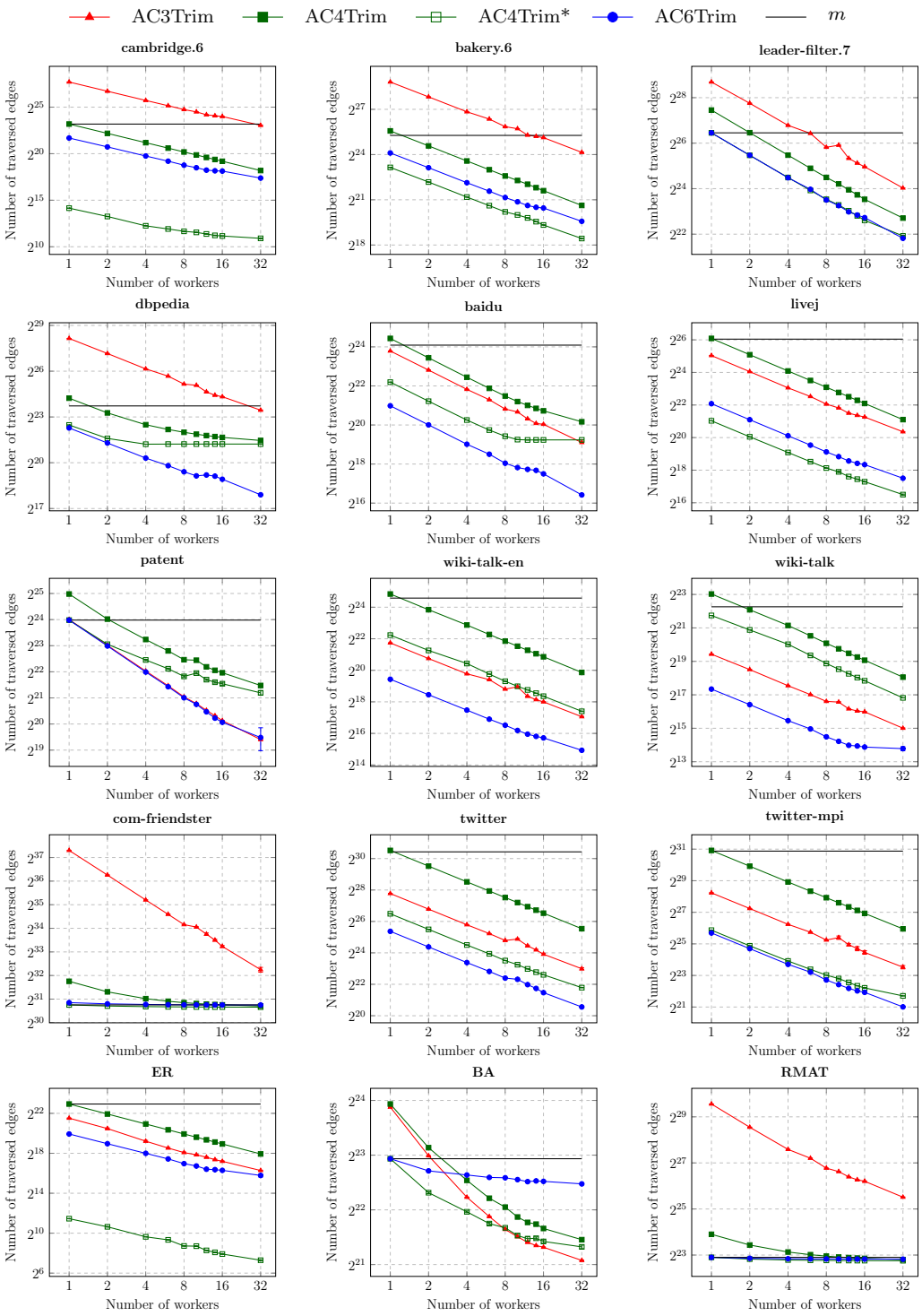


Figure 3.4: The maximum number of traversed edges per worker for *AC3Trim*, *AC4Trim*, *AC4Trim** and *AC6Trim* by varying the number of workers. The number of edges m in a graph is chosen as a baseline.

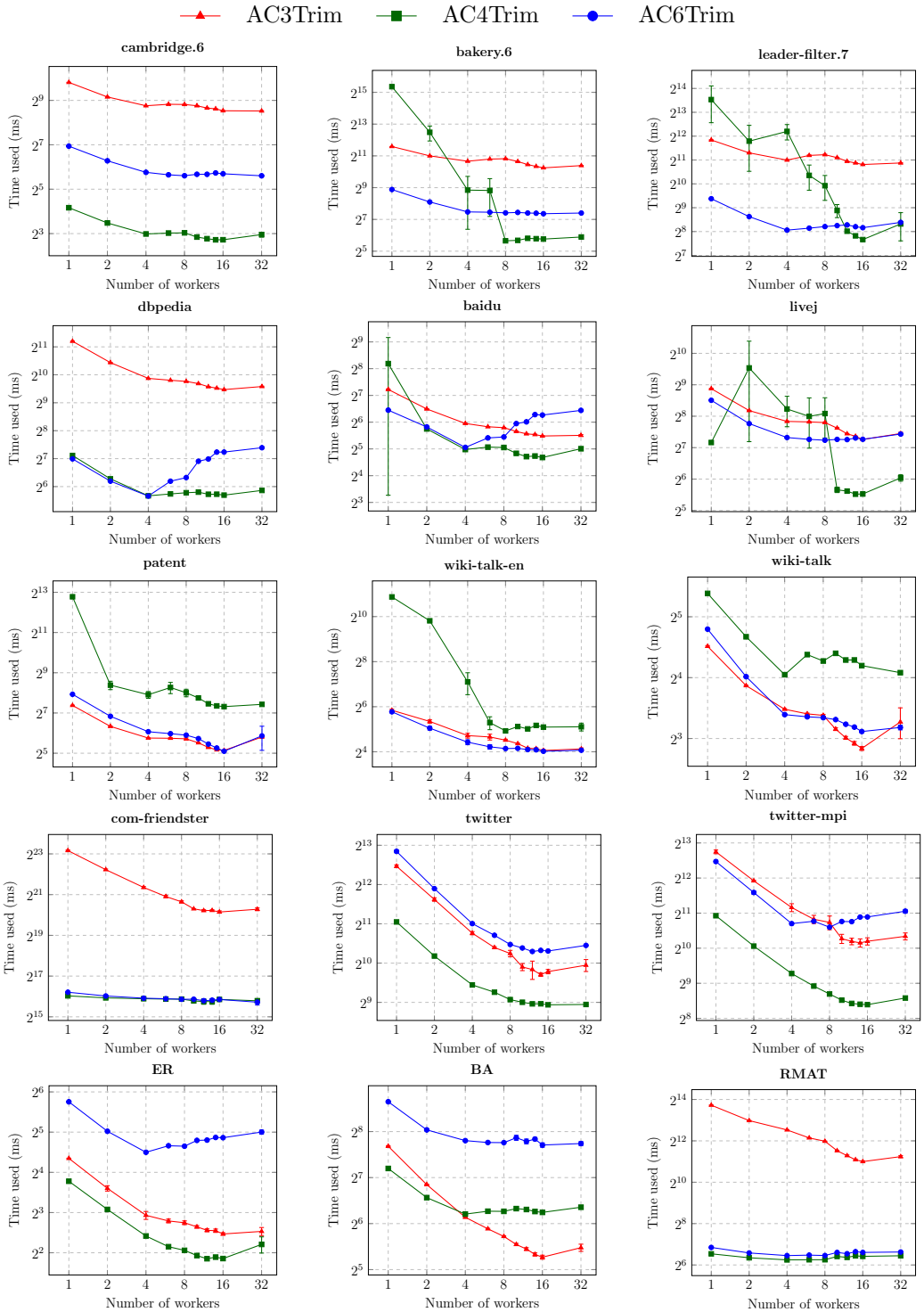


Figure 3.5: The real running time for *AC3Trim, AC4Trim* and *AC6Trim* by varying the number workers.

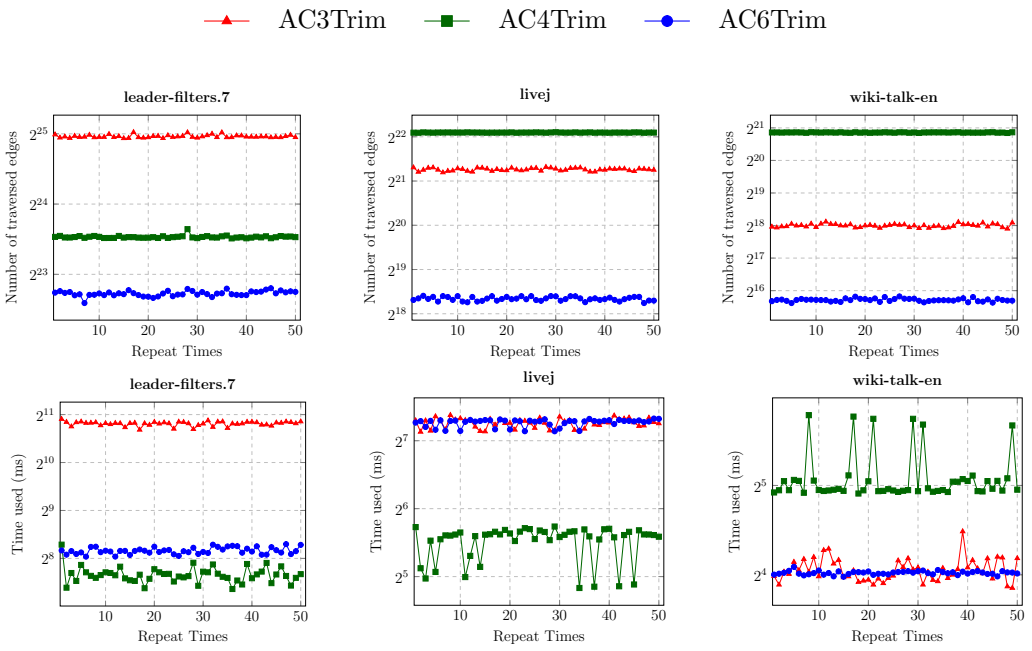


Figure 3.6: The stability of the traversed edge number and the running time for *AC3Trim*, *AC4Trim* and *AC6Trim*.

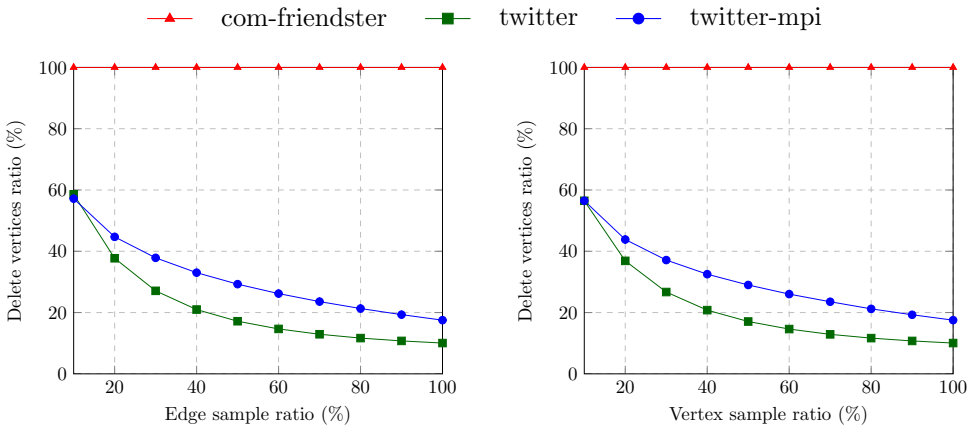


Figure 3.7: The ratio of trimmable vertices.

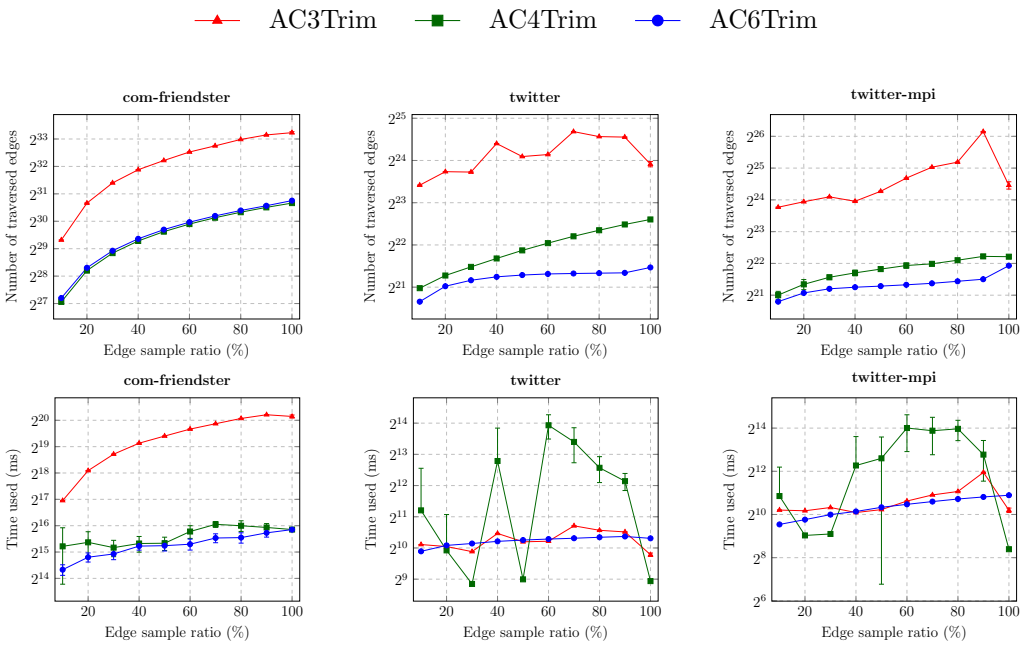


Figure 3.8: The scalability of *AC3Trim, AC4Trim* and *AC6Trim* by using 16 workers. The number of edges is varied by randomly sampling from 10% to 100%

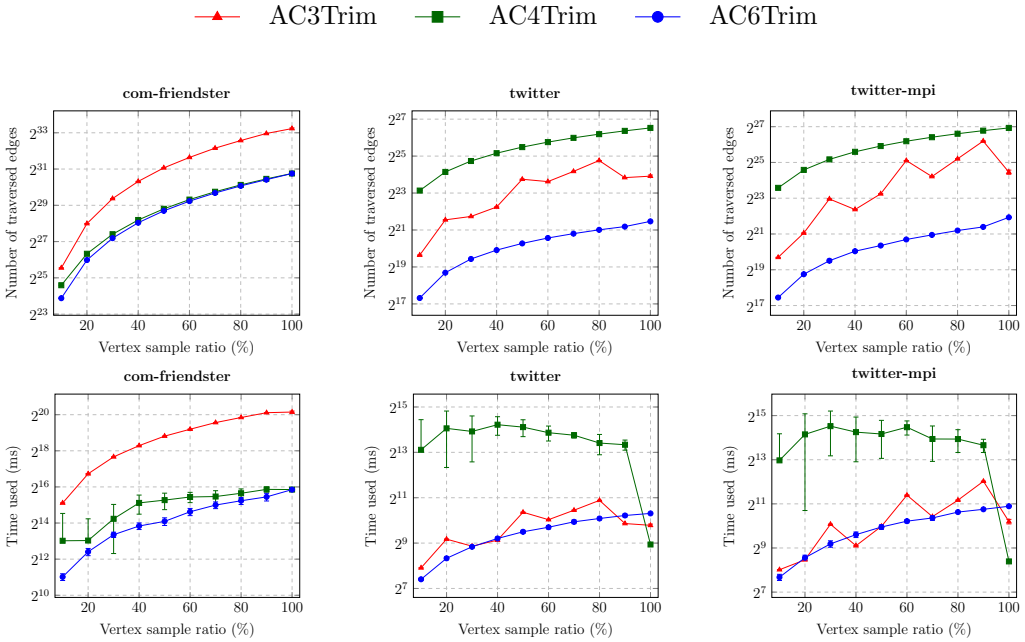


Figure 3.9: The scalability of *AC3Trim, AC4Trim* and *AC6Trim* by using 16 workers. The vertices are varied by randomly sampling at radio from 10% to 100%

Part III

Core Maintenance

Chapter 4

Parallel Order Maintenance

The *Order-Maintenance* (OM) data structure maintains a total order list of items for insertions, deletions, and comparisons. As a basic data structure, OM has many applications, such as maintaining the topological order, core numbers, and truss in graphs, and maintaining ordered sets in Unified Modeling Language (UML) Specification. The prevalence of multicore machines suggests parallelizing such a basic data structure. This chapter proposes a new parallel OM data structure that supports insertions, deletions, and comparisons in parallel. Specifically, parallel insertions and deletions are synchronized by using locks efficiently, which achieve up to 7x and 5.6x speedups with 64 workers. One big advantage is that the comparisons are lock-free so that they can execute highly in parallel with other insertions and deletions, which achieve up to 34.4x speedups with 64 workers. Typical real applications maintain order lists that always have a much larger portion of comparisons than insertions and deletions. For example, in core maintenance, the number of comparisons is up to 297 times larger compared with insertions and deletions in certain graphs. This is why the lock-free order comparison is a breakthrough in practice.

4.1 Introduction

The well-known *Order-Maintenance* (OM) data structure (Dietz and Sleator, 1987; Bender et al., 2002; Utterback et al., 2016) maintains a total order of unique items in an order list, denoted as $\mathbb{O}$, by following three operations:

- `Order`$(\mathbb{O}, x, y)$: determine if x precedes y in the ordered list $\mathbb{O}$, supposing both x and y are in $\mathbb{O}$.
- `Insert`$(\mathbb{O}, x, y)$: insert a new item y after x in the ordered list $\mathbb{O}$, supposing x is in $\mathbb{O}$ and y is not in $\mathbb{O}$.
- `Delete`$(\mathbb{O}, x)$: delete x from the ordered list $\mathbb{O}$, supposing x is in $\mathbb{O}$.

In the sequential case, the OM data structure has been well studied. The naive idea is to use a balanced binary search tree (Cormen et al., 2022), all three operations can be performed in $O(\log N)$ time, where there are at most N items in the ordered list $\mathbb{O}$. In (Dietz and Sleator, 1987; Bender et al., 2002), the authors propose an OM data structure that supports all three operations in $O(1)$ time. The idea is that all items in $\mathbb{O}$ are linked as a double-linked list. Each item is assigned a label to indicate its order. We can perform the `Order` operation by comparing the labels of two items by $O(1)$ time. Also, the `Delete` operation also costs $O(1)$ time without changing other labels. For the `Insert`(x, y) operation, y can be directly inserted after x with $O(1)$ time, if there exists label space between x and x's successors; otherwise, a ***relabel*** procedure is triggered to rebalance the labels, which costs amortized $O(\log N)$ time per insertion. After introducing the list of sublists structure, the amortized running time of the relabel procedure can be optimized to $O(1)$ per insertion. Thus, the `Insert` operation has $O(1)$ amortized time.

In the parallel or concurrent case, however, there exists little work (Gilbert et al., 2003; Utterback et al., 2016) to the best of our knowledge. In this thesis, we present a new parallelized OM data structure that supports `Insert`, `Order`, and `Delete` operations executing in parallel. In terms of parallel `Insert` and `Delete` operations, we use locks for synchronization without interleaving with each other. In the average case, it is a high probability that the multiple `Insert` or `Delete` operations occur in different positions of $\mathbb{O}$ so that both operations can execute highly in parallel. Especially for the `Order` operations, we adopt a lock-free mechanism, which can always execute highly in parallel for any pair of items in $\mathbb{O}$. To implement the lock-free `Order`, we devise a new algorithm for `Insert` operation that can always maintain the *Order Invariant* for all items, even if many relable procedures are triggered. Here, the Order Invariant means the labels of items indicate their order correctly. When `Insert` operations always maintain the Order Invariant, we do not need to lock a pair of items when comparing their labels in parallel. In other words, lock-free `Order` operations are based on `Insert` operations that preserve the Order Invariant.

Our new parallel lock-free `Order` operation is a breakthrough for real applications. Typically, for the OM data structure, a large portion of operations are comparing the order of two items. For example, in Figure 4.1, we show the number of OM operations for the core maintenance in (Guo and Sekerinski, 2022c) by randomly inserting $100,000$ edges over 12 tested data graphs. We observe that the number of `Order` operations is much larger than the number of `Insert` and `Delete` operations, e.g., for the RMAT graph, the number of `Order` operations is about 287 times of the number of other operations. The reason is that graphs always have a much larger number of edges than vertices. The big advantage of our parallel `Order` operation is that it can execute highly

Figure 4.1: The number of OM operations for core maintenance by inserting 100,000 random edges into each graph.

in parallel without locking items, which is essential when trying to parallel the core maintenance algorithms.

Parallel operation	Worst-case (O) $\mathcal{W}$	$\mathcal{D}$	time	Best-case (O) $\mathcal{W}$	$\mathcal{D}$	time
`Insert`	$m^\dagger$	$m^\dagger$	$\frac{m}{\mathcal{P}} + m^\dagger$	$m^\dagger$	$1^\dagger$	$\frac{m}{\mathcal{P}}^\dagger$
`Delete`	m	m	$\frac{m}{\mathcal{P}} + m$	m	1	$\frac{m}{\mathcal{P}}$
`Order`	m	1	$\frac{m}{\mathcal{P}}$	m	1	$\frac{m}{\mathcal{P}}$

Table 4.1: The worst-case and best-case work, depth complexities of parallel OM operations, where m is the number of operations executed in parallel, $\mathcal{P}$ is the total number of workers, and † is the amortized complexity.

Table 6.1 compares the worst-case and best-case work and depth complexities for three OM operations, given m operations in parallel. In the best case, all three operations have $O(m)$ work and $O(1)$ depth. However, the parallel `Insert` has worst-case $O(m)$ work and $O(m)$ depth; such a worst-case is easy to construct, e.g. m items are inserted into the same position of $\mathbb{O}$ and thus all insertions are reduced to sequential. The parallel `Delete` also has worst-case $O(m)$ work and $O(1)$ depth; but such a worst case only happens when all deletions cause a blocking chain, which has a very low probability. Especially,

since the parallel `Order` is lock-free, it always has $O(m)$ work and $O(1)$ depth whatever in the worst or best case. This is why the parallel `Order` always has great speedups for multicore machines. The lock-free parallel `Order` is an important contribution in this work.

Additionally, we conduct extensive experiments on a 64-core machine over a variety of test cases to evaluate the parallelism of the new parallel OM data structure.

4.2 Related Work

In (Dietz, 1982), Dietz proposes the first order data structure, which has `Insert` and `Delete` in $O(\log n)$ amortized time and `Order` in $O(1)$ time. In (Tsakalidis, 1984), Tsakalidis improves the update bound. In (Dietz and Sleator, 1987), Dietz et al. propose a fastest order data structure, which has `Insert` in $O(1)$ amortized time, `Delete` in $O(1)$ time, and `Order` in $O(1)$ time. In (Bender et al., 2002), Bender et al. propose significantly simplified algorithms that match the bounds in (Dietz and Sleator, 1987).

The special case of OM is the *file maintenance* problem. The file maintenance is to store n items in an array of size $O(n)$. It supports four operations, i.e., insert, delete, scan-right (scan next k items starting from e), and scan-left (analogous to scan-right).

In (Utterback et al., 2016), Utterback et al. propose a parallel OM data structure specifically used for *series-parallel (SP)* maintenance, which identifies whether two accesses are logically parallel. Several parallelism strategies are present for the OM data structure combined with SP maintenance. We apply the strategy of splitting a full group into our new parallel OM data structure.

4.3 Preliminary

The well-know *Order-Maintenance* (OM) data structure (Dietz and Sleator, 1987; Bender et al., 2002; Utterback et al., 2016) maintains a total order of unique items in an order list, denoted as $\mathbb{O}$, by following three operations:

- `Order`$(\mathbb{O}, x, y)$: determine if x precedes y in the ordered list $\mathbb{O}$, supposing both x and y are in $\mathbb{O}$.
- `Insert`$(\mathbb{O}, x, y)$: insert a new item y after x in the ordered list $\mathbb{O}$, supposing x is in $\mathbb{O}$ and y is not in $\mathbb{O}$.
- `Delete`$(\mathbb{O}, x)$: delete x from the ordered list $\mathbb{O}$, supposing x is in $\mathbb{O}$.

The idea is that items in the total order are assigned labels to indicate the order. Typically, each label can be stored as an $O(\log N)$ bits integer, where N is the maximal number of items in $\mathbb{O}$. Assume it takes $O(1)$ time to compare two integers. The `Order` operation requires $O(1)$ time by comparing labels; also, the `Delete` operation requires $O(1)$ time since all other labels are not affected.

In terms of the `Insert` operation, efficient implementations provide $O(1)$ amortized time. First, a *two-level* data structure (Utterback et al., 2016) is used. That is, each item is stored in the bottom-list, which contains a *group* of consecutive elements; each group is stored in top-list, which can contain $\Omega(\log N)$ items. Both the top-list and the bottom-list are organized as double-linked lists, and we use $x.pre$ and $x.next$ to denote the predecessor and successor of x, respectively. Second, each item x has a top-label $L^t(x)$, which equals to x's group label denoted as $L^t(x) = L(x.group)$, and bottom-label $L_b(x)$, which is x's label. The L^t is in the range $[0, N^2]$ and L_b in the range $[0, N]$.

Initially, there are N' items contained in N' groups ($N' \leq N$), separately, where each group can have a top-label L with a N gap between neighbors and each item has a bottom-label L_b as $\lfloor N/2 \rfloor$.

Definition 4.3.1 (Order Invariant)**.** The OM data structure maintains the Order Invariant for x precedes y in the total order, denoted as

$$x \preceq y \equiv L^t(x) < L^t(y) \ \vee \ (L^t(x) = L^t(y) \wedge L_b(x) < L_b(y))$$

The OM data structure maintains the Order Invariant defined in Definition 4.3.1. In order words, to determine the order of x and y, we first compare their top-labels (group labels) of x and y; if they are the same, we continually compare their bottom labels.

4.3.1 Sequential Order Insert

An `Insert`$(\mathbb{O}, x, y)$ is implemented by inserting y after x in x's bottom-list, assigning y an label $L_b(y) = \lfloor (L_b(x.next) - L_b(x))/2 \rfloor$, and set y in the same group as x with $y.group = x.group$ such that $L^t(y) = L^t(x)$. If y can successfully obtain a new label, then the insertion is complete in $O(1)$ time. Otherwise, e.g. $x.next$ has label $L_b(x) + 1$, the group is *full*, which triggers a *relabel* operation. Specifically, the relabel operations have two steps:

– *Rebalance*: if there has no label space after x's group g, we have to rebalance the top-labels of groups. From g, we continuously traverse the successors g' until $L(g') - L(g) > j^2$, where j is the number of traversed

groups. Then, new group labels can be assigned with the gap j, in which newly created groups can be inserted. Finally, a new group can be inserted after g.

– *Split*: when x' group g is full, g is split out one new group, which contains at most $\frac{\log N}{2}$ items and new bottom-labels L_b are uniformly assigned for items in new groups. Newly created groups are inserted after g, where we can create the label space by the above rebalance operation.

Such rebalance and split operations will continue until less than $\frac{\log N}{2}$ items are left in g. Also, new bottom-labels L_b are uniformly assigned for items in g.

For the implementation of the `Insert` operation, there are three important features: (1) each group, stored in the top-list, contains $\Omega(\log N)$ items, so that the total number of insertions is $O(N/\log N)$; (2) the amortized cost of splitting groups is $O(1)$ per insert; (3) the amortized cost of inserting a new group into top-list is $O(\log N)$ per insert. Thus, each `Insert` operation only costs amortized $O(1)$ time.

Example 4.3.1 (Insert)**.** In Figure 4.2, we show a simple example of OM data structure. The squares are groups located in a single double-linked top-list with a head h^t and a tail t_t. The cycles are items with pointers to their own groups, located in a single double-linked bottom-list. For simplicity, we choose $N = 2^4 = 16$, so that for items the top-labels L^t are 8-bit integers (above groups as group labels), and the bottom labels are 4-bit integers (below items).

Figure 4.2(a) shows an initial state of the two-level lists and labels. The top-list has head h^t and tail t^t labeled by 0 and $16^2 - 1$, respectively, and includes four groups g_1 to g_4 labeled with gap 16. The bottom-list has head h_b and tail t_b without labels, and includes four items v_1 to v_4 located in four groups g_1 to g_4 with same labels 7.

Figure 4.2(b) shows an intermediate state after a lot of `Insert` and `Delete` operations. We can see that there not exist label space between v_1 and v_2. Both v_1 and v_2 are located in the group g_1. We get that g_1 is full when inserting a new item after v_1.

In Figure 4.2(c), a new item u will insert after v_1. But the group g_1 is full (no label space after v_1), which will trigger a *relabel* process. That is, the group g_1 is split into two groups, g_1 and g; the old group g_1 only has v_1; the new created group g contains v_2 and v_3; also, v_1 to v_3 are average assigned L_b within their own group. However, there is no label space between g_1 and g_2 to insert the new group g, which will trigger a *rebalance* process. That is, we traverse groups from g_1 to g_4, where g_4 is the first that satisfies $L(g_4) - L(g_1) = 15 > j^2$ $(j = 3)$. Then, both g_2 and g_3 are assigned new

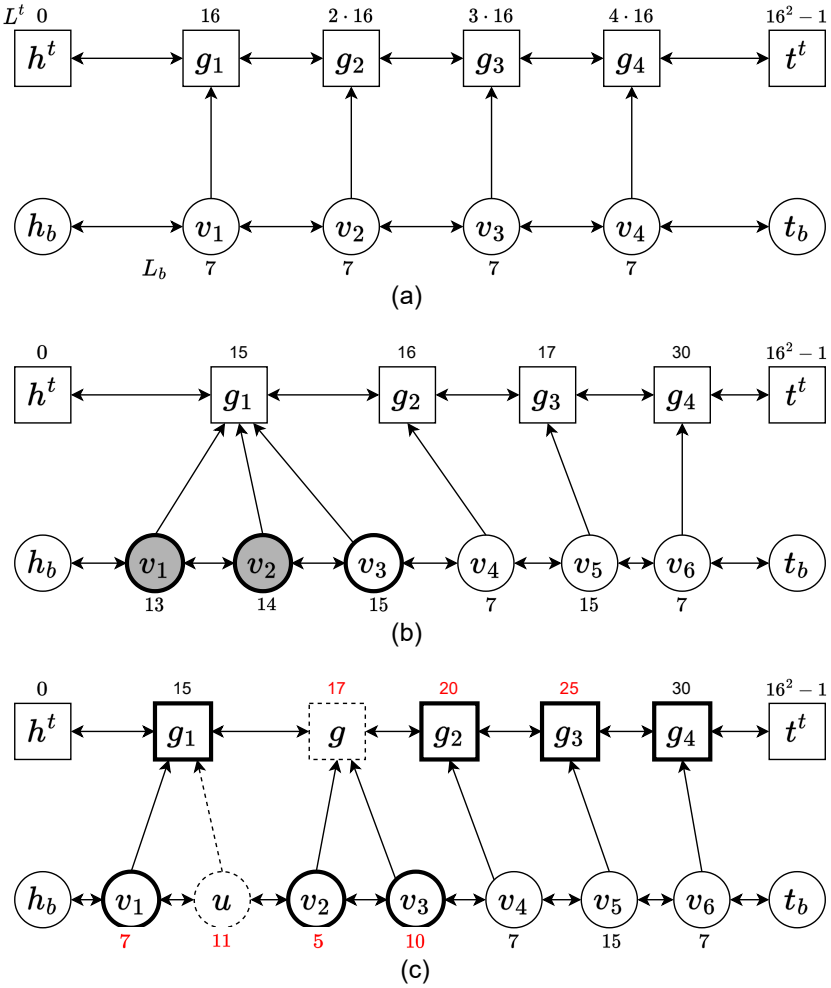


Figure 4.2: A example of the OM data structure with $N = 16$.

top-labels as 20 and 25, respectively, which have the gap as 5. Now, g can be inserted after g_1 with $L(g) = L(g_2) + (L(g_2) - L(g_1))/2 = 17$. Finally, we can insert u successfully after v_1 in g_1, with $L_b(u) = L_b(v_1) + (15 - L_b(v_1))/2 = 11$.

4.4 Parallel OM Data Structure

In this section, we present the parallel version of the OM data structure. We start from the parallel `Delete` operations. Then, we discuss the parallel `Insert` operation and show that the Order Invariant is preserved at any steps including the relable process, which is the main contribution of this work. Finally, we present the parallel `Order` operation, which is lock-free and thus can be executed highly in parallel.

4.4.1 Parallel Delete

The detailed steps of parallel `Delete` operations are shown in Algorithm 11. For each item x in $\mathbb{O}$, we use a status $x.live$ to indicate x is in $\mathbb{O}$ or removed. Initially $x.live$ is `true`. We use the atomic primitive `CAS` to set $x.live$ from `true` to `false`, and the repeated deleting x will return `error` (line 1). In lines 2 - 6 and 13, we remove x from the bottom-list in $\mathbb{O}$. To do this, we first lock $y = x.pre$, x, and $x.next$ in order to avoid deadlock (lines 2 - 4). Here, after locking y, we have to check that y still equals $x.pre$ in case $x.pre$ is changed by other workers (line 3). Then, we can safely delete x from the the bottom-list, and set x's pre, , L_b, and $group$ to empty (line 6). Finally, we unlock all locked items in reverse order (line 13). Do not forget that, for the group $g = x.group$, we need to delete g when it is empty, which is analogous to deleting x (lines 7 - 12).

Algorithm 11: Parallel-Delete($\mathbb{O}, x$)

```
1  if not CAS(x.live, true, false) then return error
2  y ← x.pre; Lock(y)
3  if y ≠ x.pre then Unlock(y); goto line 1
4  Lock(x); Lock(x.next)
5  g ← x.group
6  delete x from bottom-list; set x.pre, x.next, L_b(x), and x.group to ∅
7  if |g|= 0 ∧ CAS(g.live, true, false) then
8  |   g' ← g.pre ; Lock(g')
9  |   if g' ≠ g.pre then Unlock(g'); goto line 8
10 |   Lock(g); Lock(g.next)
11 |   delete g from top-list; set g.pre, g.next, and L(g) to ∅
12 |   Unlock(g.next); Unlock(g); Unlock(g')
13 Unlock(x.next); Unlock(x); Unlock(y);
```

We *logically* delete items by setting their flags (line 1). One benefit of such a method is that we can delay the *physical* deleting of items (lines 2 - 13). The physical deleting can be batched and performed lazily at a convenient time, reducing the overhead of synchronization. Typically, this trick is always used in linked lists for delete operations (Herlihy et al., 2020). In his paper, we will not test the delayed physical delete operations.

Obviously, during the parallel `Delete` operations, the labels of other items are not affected and the order invariant is maintained.

Correctness. For deleting x, we always lock three items, $x.pre$, x, and $x.next$ in order. Therefore, there are no blocking cycles, and thus it is deadlock-free.

Complexities. Suppose there are m items to delete in the OM data structure. The total work is $O(m)$. In the best case, m items can be deleted in parallel by $\mathcal{P}$ workers with $O(1)$ depth, so that the total running time is $O(m/\mathcal{P})$. In the worst-case, n items have to be deleted one by one, e.g. $\mathcal{P}$ workers are blocked as a chain, with $O(m)$ depth, so that the total running time is $O(m/\mathcal{P} + m)$.

However, when deleting multiple items in parallel, the blocking chain is unlikely to appear, and thus the worst-case has a low probability to happen.

4.4.2 Parallel Insert

The detailed steps of parallel `Insert` are shown in Algorithm 12 for inserting y after x. Within this operation, we should lock x and its successor $z = x.next$ in order (lines 1 and 7). For obtaining a new bottom-label for y, if x and z is in the same group, $L_b(z)$ is the right bound; otherwise, N is the right bound supposing L_b is a $(\log N)$-bit integer (line 2). If there does not exist a label gap in the bottom-list between x and $x.next$, we know that $x.group$ is full, and thus the `Relabel` procedure is triggered to make label space for y (line 3). Then, y is inserted into the bottom-list between x and $x.next$ (line 6), in the same group as x (line 4), by assigning a new bottom-label (line 5).

In the `Relabel`(x) procedure, we will split the full group of x. We lock x's group g_0 and g_0's successor $g.next$ (line 9). We also lock all items $y \in g_0$ except x, as x is already locked in line 1 (line 10). To split the group g_0 into multiple new smaller groups, we traverse the items $y \in g_0$ in reverse order by three steps (lines 11 - 15). First, if there does not exist a label gap in the top-list between g_0 and $g_0.next$, the `Rebalance` procedure is triggered to make label space for inserting a new group with assigned labels (lines 12 and 13). Second, we split $\frac{logN}{2}$ items y from g_0 in reverse order to the new group g, which maintains the Order Invariant (line 14). Third, we assign new L_b to all items in the new group g by using the `AssignLable` procedure(line 15), which also maintains the Order Invariant. The for-loop (lines 11 - 15) stops if less than $\frac{logN}{2}$ items are left in g_0. We assign new L_b to all left items in g_0 by using the `AssignLable` procedure(line 16). Finally, do not forget to unlock all locked groups and items (line 17).

In the `Rebalance`(g) procedure, we make label space after g to insert new groups. Starting from $g.next$, we traverse groups g' in order until $w > j^2$ by locking g' if necessary (g and $g.next$ are already locked in line 9), where j is the number of visited groups and w is the label gap $L(g') - L(g)$ (lines 19 - 22). That means j items will totally share $w > j^2$ label space. All groups, whose labels should be updated, are added to the set A (line 21). We assign new labels to all groups in A by using the `AssignLabel` procedure (line 23), which maintains the Order Invariant. Finally, do not forget to unlock groups

locked in line 21.

Especially, in the `AssignLable`$(A, \mathcal{L}, l_0, w)$ procedure, we assign labels without affecting the Order Invariant, where the set A includes all elements whose labels need to update, $\mathcal{L}$ is the label function, l_0 is the starting label, and w is the label space. Note that, $\mathcal{L}$ can correctly return the bounded labels, e.g., $L_b(x.next) = N$ when x is at the tail of its group $x.group$. For each $z \in A$ in order, we first correctly assign a temporary label $\overline{\mathcal{L}}(z)$ (line 27), which can replace its real label $\mathcal{L}(z)$ at the right time by using stack S (lines 28 - 32). Specifically, for each $z \in A$ in order, if its temporary label $\overline{\mathcal{L}}(z)$ is between $\mathcal{L}(z.pre)$ and $\mathcal{L}(z.next)$, we can safely *replace* its label by updating $\mathcal{L}(z)$ as $\overline{\mathcal{L}}(z)$ (lines 29 and 30), which maintains the Order Invariant; otherwise, z is added into the stack S for further propagation (line 32). For the propagation, when one element z replaces the labels (line 30), which means all elements in stack S can find enough label space, each $x \in S$ can be popped out by replacing its label (line 31). This propagation still maintains the order Invariant.

Example 4.4.1 (Parallel Insert)**.** Continually, we use Figure 4.2 to show an example for the parallel `Insert`. In Figure 4.2(b), we lock v_1 and lock $v2$ in order when inserting u after v_1. However, there is no label space, and the group g_1 is full, which triggers the `Relabel` procedure. For the first step of relabel, the other item v_3 in group g_1 is locked to split the group g_1.

In Figure 4.2(c), we lock g_1 and g_2 in order when inserting a new group g after g_1, which triggers the `Rebalance` procedure on the top-list. For rebalance, g_3 and g_4 are locked in order. The new temporal labels $\overline{L^t}$ of g_2 and g_3 are generated as $\overline{20}$ and $\overline{25}$. To replace real labels with temporal ones, we traverse g_2 and g_3 in order. First, we find that $L(g_1) < \overline{L}(g_2) < L(g_3)$ as $15 < \overline{20} < 17$ is false, so that g_2 is added to the stack such that $S = \{g_2\}$. Second, when traversing g_3, we find that $L(g_2) < \overline{L}(g_3) < L(g_4)$ as $16 < \overline{25} < 30$ is true, so that $L(g_3)$ is replaced as 25. In this case, the propagation of S begins and g_2 is popped out with $L(g_2)$ replace as 20. Finally, g_3 and g_4 are unlocked and the `Rebalance` procedure finishes.

After rebalance, the new group g can be inserted after g_1 with $L(g_1) = 17$. The relabeling continue. The item v_3 is spitted out to g with $L_b(v_3) = 15 \wedge L^t(v_3) = 17$ maintaining the Order Invariant; similarly, v_2 is also spitted out to g. Now, both v_2 and v_3 require to assign new L_b by the `AssignLable` procedure. The new temporal label $\overline{L_b}$ of v_2 and v_3 are generated as $\overline{5}$ and $\overline{10}$, respectively. For replacing, we traverse v_2 and v_3 in order by two steps. First, for v_2, we find that $\overline{L_b}(v_2) < L_b(v_3)$ is true, so that $L_b(v_2)$ is replaced as 5. Second, for v_3, we find that $L_b(v_2) < \overline{L_b}(v_3)$ is true, so that $L_b(v_3)$ is replaced as 10 It has no propagation since the stack S is empty.

Algorithm 12: Parellel-Insert($\mathbb{O}, x, y$)

```
1  Lock(x); z ← x.next; Lock(z)
2  if x.group = z.group then b ← L_b(z) else b ← N
3  if b − L_b(x) < 2 then
4  Relabel(x);
5
6  insert y into bottom-list between x and x.next
7  L_b(y) = L_b(x) + ⌊(b − L_b(x))/2⌋
8  y.group ← x.group
9  Unlock(x); Unlock(z)
10 procedure Relabel(x)
11     g_0 ← x.group; Lock(g_0); Lock(g_0.next);
12     Lock all items y ∈ g_0 with y ≠ x in order
13     for y ∈ g_0 in reverse order until less than (log N)/2 items left in g_0 do
14         if L(g_0.next) − L(g_0) < 2 then Rebalance(g_0)
15         insert a new group g into the top-list after g_0 with
             L(g) = (L(g_0.next) − L(g_0))/2
16         split out (log N)/2 items y into g
17         AssignLabel(g, L_b, 0, N)
18     AssignLabel(g_0, L_b, 0, N)
19     Unlock g_0.next, g_0, and all items y ∈ g_0 with y ≠ x
20 procedure Rebalance(g)
21     g' ← g.next; j ← 1; w ← L(g') − L(g); A ← ∅
22     while w ≤ j^2 do
23         A ← A ∪ {g'}; g' ← g'.next; Lock(g')
24         j ← j + 1; w ← L(g') − L(g)
25     AssignLabel(A, L^t, L^t(g), w)
26     Unlock all locked groups in line 23.
27 procedure AssignLabel(A, L, l_0, w)
28     S ← empty stack; k ← 1; j ← |A|+1
29     for z ∈ A in order do L̄(z) = l + k · w/j; k ← k + 1
30     for z ∈ A in order do
31         if L(z.pre) < L̄(z) < L(z.next) then
32             L(z) ← L̄(z)
33             while S ≠ ∅ do  x ← S.pop(); L(x) ← L̄(x)
34         else  S.push(z)
```

Correctness. We prove that the Order Invariant is preserved during parallel `Insert` operations. In Algorithm 12, there are two cases where the labels are updated, splitting groups (lines 11 - 15) and assigning labels (lines 15, 16, and 23) by using the `AssignLabel` procedure (lines 25 - 32).

Example 4.4.2 (Assign Label)**.** In Figure 4.3, we show an example that how the `AssignLabel` procedure preserves the Order Invariant. The label space is from 0 to 15 shown as indices. There are four items v_1, v_2, v_3, and v_4 with initial old labels 1, 2, 3, and 14, respectively; also, four temporal labels, $3, 6, 9$, and 12, are averagely assigned for them. We traverse items from v_1 to v_4 in order. First, v_1 and $v2$ are added into the stack S. Second, v_3 can safely replace its old label with its new temporal label 9, which makes space for $v2$ that is at the top of S. So, we pop out v_2 from S and v_2 gets a new label 6, which makes space for v_1 that is at the top of S. So, we pop out v_1 from S and v_1 gets a new label 3. Finally, v_4 can safely get its new label 12. In a word, updating the v_3's label will repeatedly make space for v_2 and v_1 in the stack. During such a process, we observe that when each time one old label updates as a new temporal label, the labels always correctly indicate the order. Therefore, the Order Invariant is always preserved.

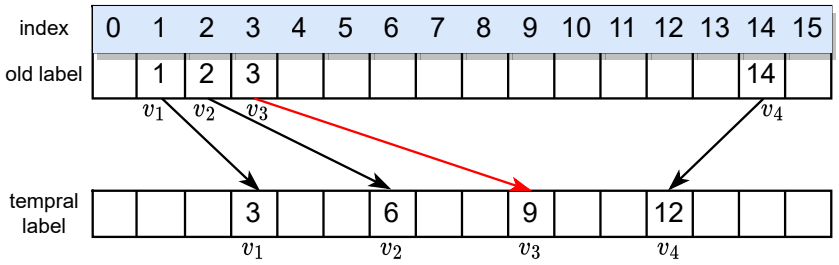


Figure 4.3: A example of the `AssignLable` procedure.

Theorem 4.4.1. *When splitting full groups (line 14), the Order Invariant is preserved.*

Proof. The algorithm splits $\frac{\log N}{2}$ items y out from g_0 into the new group g (line 14), where each $y \in g_0$ is traversed in reverse order within the for-loop (lines 11 - 15). For this, the invariant of the for-loop is that y has largest L_b within g_0; the new group g has $L(g) > L(g_0)$; also, y satisfies the Order Invariant:

$$(\forall x \in g_0 : x \neq y \implies L_b(y) > L_b(x)) \\ \wedge \ (L(g_0) < L(g)) \ \wedge \ (y.pre \preceq y \preceq y.next)$$

We now argue the for-loop preserve this invariant:

- $\forall x \in g_0 : x \neq y \implies L_b(y) > L_b(x)$ is preserved as y is traversed in reserve order within g_0 and all other items y' that have $L_b(y) < L_b(y')$ are already spitted out from g_0.
- $L(g_0) < L(g)$ is preserved as g is new inserted into top-list after g_0.
- $y.pre \preceq y$ is preserved as we have $y \in g \wedge y.pre \in g_0$ and $L(g_0) < L(g)$.

– $y \preceq y.next$ is preserved as if y and $y.next$ all in the same group g, we have $L_b(y) < L_b(y.next)$; also, if y and $y.next$ in different groups, we have y is the first item moved to g or y is still located in g_0, which their groups indicates the correct order.

At the termination of the for-loop, the group g is split into multiple groups preserving the Order Invariant. □

Theorem 4.4.2. *When assigning labels by using the* `AssignLable` *procedure (lines 25 - 32), the Order Invariant is preserved.*

Proof. The `AssignLabel` procedure (lines 25 - 32) assigns labels for all items $z \in A$. The temporal labels are first generated in advance (line 27). Then, the for-loop replaces the old label with new temporal labels (lines 28 - 32). The key issue is to argue the correctness of the inner while-loop (line 31). The invariant of this inner while-loop is that the top item in S has a temporal label that satisfies the Order Invariant:

$$\begin{aligned}&(\forall y \in S : (y \neq S.top \implies y \preceq S.top) \ \wedge \ y \preceq z)\\ &\wedge \ x = S.top \implies \mathcal{L}(x.pre) < \overline{\mathcal{L}}(x) < \mathcal{L}(x.next)\end{aligned}$$

The invariant initially holds as $\mathcal{L}(z)$ is correctly replaced by the temporal label $\overline{\mathcal{L}}(z)$ in line 30 and z is $x.next$, so that $\overline{\mathcal{L}}(x) < \mathcal{L}(z)$; also, we have $(x.pre) < \overline{\mathcal{L}}(x)$ as if it is not satisfied, x should not be added into S, which causes a contradiction. We now argue the while-loop (line 31) preserves this invariant:

– $\forall y \in S : (y \neq S.top \implies y \preceq S.top) \ \wedge \ y \preceq z$ is preserved as all items in S are added in order, so the top item always has the largest order; also, since all item in A are traversed in order, so z has the larger order than all item in S.

– $x = S.top \implies \overline{\mathcal{L}}(x) < \mathcal{L}(x.next)$ is preserved as $\mathcal{L}(x.next)$ is already replace by the temporal label $\overline{\mathcal{L}}(x.next)$ and x is precede $x.next$ by using temporal labels.

– $x = S.top \implies \mathcal{L}(x.pre) < \overline{\mathcal{L}}(x)$ is preserved as if such invariant is not satisfied, x should not be added into S and can safely replace its label with its temporal label, which causes a contradiction.

At the termination of the inner while-loop, we get $S = \emptyset$, so that all items that precede z have replaced new labels maintaining the Order Invariant. At the termination of the for-loop (lines 28 -32), all items in A have been replaced with new labels. □

Complexities. For the sequential version, it is proven that the amortized time is $O(1)$. The parallel version has some refinement. That is, the `AssignLable` procedure traverses the locked items two times for generating temporal labels and replacing the labels, which cost amortized time $O(1)$. Thus, supposing m items are inserted in parallel, the total amortized work is $O(m)$. In the best case, m items can be inserted in parallel by $\mathcal{P}$ workers with amortized depth $O(1)$, so that the amortized running time is $O(m/\mathcal{P})$. In the worst-case, m items have to be inserted one-by-by, e.g. $\mathcal{P}$ workers simultaneously insert items at the head of $\mathbb{O}$ with amortized depth $O(m)$, and thus the amortized running time is $O(m/\mathcal{P} + m)$.

The worst-case is easy to happen when all insertions accrued in the same position of $\mathbb{O}$. Such worst-case can be improved by batch insertion. The idea is that we first allocate enough label space for $m/\mathcal{P}$ items per worker, then $\mathcal{P}$ workers can insert items in parallel. However, this simple strategy requires pre-processing of $\mathbb{O}$ and does not change the worst-case time complexity.

4.4.3 Parallel Order

The detailed steps of the parallel `Order` are shown in Algorithm 13. When comparing the order of x and y, they can not be deleted (line 1). We first compare the top-labels of x and y (lines 2 - 5). Two variables, t and t', obtain the values of $L^t(x)$ and $L^t(y)$ for comparison (line 2) and the result is stored as r. After that, we have to check $L^t(x)$ or $L^t(y)$ has been updated or not; if that is the case, we have to redo the whole procedure (line 5). Second, we compare the bottom-labels of x and y, if their top-lables are equal (lines 6 - 9). Similarly, two variables, b and b', obtain the value of $L_b(x)$ and $L_b(t)$ for comparison (line 7) and the result is stored as r. After that, we have to check whether four labels are updated or not; if anyone label is the case, we have to ***redo*** the whole procedure (line 9). We can see our parallel `Order` is lock-free so that it can execute highly in parallel. We return the result at line 11. During the order comparison, x or y can not be deleted (line 19).

Example 4.4.3 (Order)**.** In Figure 4.3, we show an example to determine the order of v_2 and v_3 by comparing their labels. Initially, both v_2 and v_3 have old labels, 2 and 3. After the `Relabel` procedure is triggered, both v_2 and v_3 have new labels, 6 and 9, in which the Order Invariant is preserved. However, it is possible that `Relabel` procedures are triggered in parallel. We first get $t = \mathcal{L}(v_3) = 3$ (old label) and second get $t' = \mathcal{L}(v_2) = 6$ (new label), so that it is incorrect for $\mathcal{L}(v_2) > \mathcal{L}(v_3)$. After we get $t' = \mathcal{L}(v_2) = 6$, the value of $\mathcal{L}(v_2)$ have to be updated to 9 since the Order Invariant is maintained. In this case, we redo the whole parallel `Order` until t and t' are not updating during comparison and thus get the correct result.

Algorithm 13: Parallel-Order($\mathbb{O}, x, y$)

```
1  if x.live = false ∨ y.live = false then return fail
2  t, t', r ← L^t(x), L^t(y), ∅
3  if t ≠ t' then
4  |   r ← t < t'
5  |   if t ≠ L^t(x) ∨ t' ≠ L^t(y) then goto line 1
6  else
7  |   b, b' ← L_b(x), L_b(y); r ← b < b'
8  |   if t ≠ L^t(x) ∨ t' ≠ L^t(y) ∨ b ≠ L_b(x) ∨ b' ≠ L_b(y) then
9  |   |   goto line 1
10 if x.live = false ∨ y.live = false then return fail
11 return r
```

Correctness. We have proven that the parallel `Insert` preserves the Order Invariant even though relabel procedures are triggered, by which labels correctly indicate the order. In this case, it is safe to determine the order for x and y in parallel. We first argue the top-labels (lines 2 - 5). The problem is that we first get $t \leftarrow L^t(x)$ and second get $t' \leftarrow L^t(y)$ successively (line 2), by which l and l' may be inconsistent, due to a `Relabel` procedure may be triggered. To argue the consistency of labels, there are two cases: 1) both t and t' obtain old labels or new labels, which can correctly indicate the order; 2) the t first obtains an old label and t' second obtains a new label, which may not correctly indicate the order as x may already update with a new label, and vice versa; if that is the case, we redo the whole process.

On the termination of the parallel `Order`, the invariant is that t and t' are consistent and thus correctly indicate the order. The bottom-labels are analogous (lines 6 - 9).

Complexities For the sequential version, the running time is $O(1)$. For the parallel version, we have to consider the frequency of redo. It has a significantly low probability that the redo will be triggered. This is because the labels are changed by the `Relabel` procedure, which is triggered when inserting $\Omega(\log N)$ items. Even if the labels of x and y are updating when comparing their order, it still has a small probability that such label updating happens during the comparison of labels (lines 3 - 4 and 7 - 8).

Thus, supposing m items are comparing orders in parallel, the total work is $O(m)$, and the depth is $O(1)$ with a high probability. So that the running time is $O(m/\mathcal{P})$ with high probability.

4.5 Experiments

We report on experimental studies for our three parallel order maintenance operations, `Order`, `Insert`, and `Delete`. We generate four different test cases to evaluate their parallelized performance. All the source code is available on GitHub[1].

Experiment Setup The experiments are performed on a server with an AMD CPU (64 cores, 128 hyperthreads, 256 MB of last-level shared cache) and 256 GB of main memory. The server runs the Ubuntu Linux (22.04) operating system. All tested algorithms are implemented in C++ and compiled with g++ version 11.2.0 with the -O3 option. OpenMP [2] version 4.5 is used as the threading library. We choose the number of workers exponentially increasing as $1, 2, 4, 8, 16, 32$, and 64 to evaluate the parallelism. With different numbers of workers, we perform every experiment at least 100 times and calculate the mean with 95% confidence intervals. Note that, the error bars are too small to see in our experiments.

In our experiment, for easy implementation, we choose $N = 2^{32}$, a 32-bit integer, as the capacity of $\mathbb{O}$. In this case, the bottom-lables L_b are 32-bit integers, and the top-lables L^t are 64-bit integers. One advantage is that reading and writing such 32-bit or 64-bit integers are atomic operations in modern machines. There are initially 10 million items in the order list $\mathbb{O}$. To test our parallel OM data structure, we do four experiments:

- *Insert*: we insert 10 million items into $\mathbb{O}$.
- *Order*: for each inserted item, we compare its order with its successive item, so that it has 10 million `Order` operations.
- *Delete*: we delete all inserted items, a total of 10 million times.
- *Mixed*: again, we insert 10 million items, mixed with 100 million `Order` operations. For each inserted item, we compare its order with its ten successive items, a total of 100 million times order comparison. This experiment is to test how often "redo" occurs in the `Order` operations when there are parallel `Insert` operations. The reason for this experiment is that many `Order` operations are mixed with few `Insert` and `Delete` operations in real applications, as shown in Figure 4.1.

For each experiment, we have four test cases by choosing different numbers of positions for inserting:

[1] `https://github.com/Itisben/Parallel-OM.git`
[2] `https://www.openmp.org/`

– *No* case: we have 10 million positions, the total number of initial items in $\mathbb{O}$, so that each position averagely has 1 inserted item. Thus, it almost has *no* `Relabel` procedures triggered when inserting.

– *Few* case: we randomly choose 1 million positions from 10 million items in $\mathbb{O}$, so that each position averagely has 10 inserted items. Thus, it is possible that a *few* `Relabel` procedures are triggered when inserting.

– *Many* case: we randomly choose 1,000 positions from 10 million items in $\mathbb{O}$, so that each position averagely has 10,000 inserted items. Thus, it is possible that *many* `Relabel` procedures are triggered when inserting.

– *Max* case: we only choose a single position (at the middle of $\mathbb{O}$) to insert $10,000,000$ items. In this way, we obtain a *maximum* number of triggered relabel procedures.

All items are inserted on-the-fly without preprocessing. In other words, 10 million items are randomly assigned to multiple workers, e.g 32 workers, even if in the *Max* case all insertions are reduced to sequential execution.

4.5.1 Evaluating Relabelling

In this test, we evaluate the `Relabel` procedures that are triggered by `Insert` operations over four test cases, *No*, *Few*, *Many*, and *Max*. For this, different numbers of workers will have the same trend, so we choose 32 workers for this evaluation.

In Table 4.2, columns 2 - 4 show the details in the *Insert* experiment, where *Relabel#* is the times of triggered `Relabel` procedures, $L_b\#$ is the number of updated bottom-labels for items, $L^t\#$ is the number of updated top-labels for items, and *AvgLabel#* is the average number of updated labels for each inserted items when inserting 10 million items. We can see that, for four cases, the amortized numbers of updated labels increase slowly, where the average numbers of inserted items for each position increase by 1, 10, 10 million, and 10 billion. This is because our parallel `Insert` operations have $O(1)$ amortized work.

	Insert				*Mixed*
Case	Relabel#	$L_b\#$	$L^t\#$	AvgLabel#	OrderRedo#
No	0	10,000,000	0	1	0
Few	2,483	10,069,551	4,967	1	0
Many	356,624	19,985,472	5,754,501	2.6	0
Max	357,142	19,999,976	99,024,410	**11.8**	0

Table 4.2: The detailed numbers of the relabel procedure.

In Table 4.2, the last column shows the numbers of *redo* for `Order` operations in the *Mixed* experiment, which are all zero. Since the *Mixed* has mixed `Order` and `Insert` operations, we may redo the `Order` operation if the corresponding labels are being updated. However, `Relabel` happens with a low probability; also, it is a low probability that related labels are changed when comparing the order of two items. This is why the numbers of *redo* are zero, leading to high parallel performance.

4.5.2 Evaluating the Running Time

In this experiment, we exponentially increase the number of workers from 1 to 64 and evaluate the real running time. We perform *Insert*, *Order*, *Delete*, and *Mixed* over four test cases, *No*, *Few*, *Many* and *Max*.

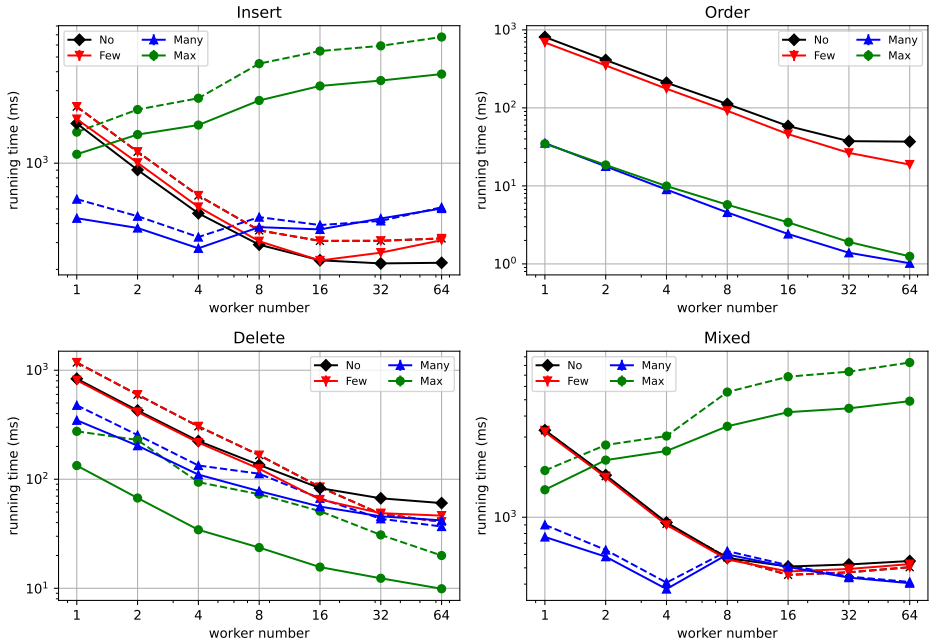


Figure 4.4: Evaluate the running times by varying the number of workers.

The plots in Figure 4.4 depict the performance. The x-axis is the number of workers and the y-axis is the execution time (millisecond). Note that, we compare the performance by using two kinds of lock: the OpenMP lock (denoted by solid lines) and the spin lock (denoted by dash lines). A first look reveals that the running times normally decrease with increasing numbers of workers, except for the *Max* case over *Insert* and *Mixed* experiments. Specifically, we make several observations:

– Three experiments, *Insert*, *Delete*, and *Mixed*, that use the spin lock are much faster than using the OpenMP lock. This is because the lock regions

always have few operations, and busy waiting (spin lock) is much faster than suspension waiting (OpenMP lock). Unlike the above three experiments, the *Order* experiment does not show any differences since `Order` operations are lock-free without using locks for synchronization.

– For the *Max* case of *Insert* and *Mixed*, abnormally, the running time is increasing with an increasing number of workers. The reason is that the `Insert` operations are reduced to sequential in the *Max Case* since all items are inserted into the same position. Thus it has the highest contention on shared positions where multiple workers access at the same time, especially for 64 workers.
– For the *Many* case of *Insert* and *Mixed*, the running times are decreasing until using 4 workers. From 8 workers, however, the running times begin to increase. This is because the `Insert` operations have only 1,000 positions in the *Many* case, and thus it may have high contention on shared positions when using more than 4 workers.
– Over the *Order* and *Delete* experiments, we can see the *Many* and *Max* cases are always faster than the *Few* and *No* cases. This is because the *Few* and *No* cases have $1,000$ and 1 operating positions, respectively; all of these positions can fit into the CPU cache with high probability, and accessing the cache is much faster than accessing the memory.

4.5.3 Evaluating the Speedups

The plots in Figure 4.5 depict the speedups. The x-axis is the number of workers, and the y-axis is the speedups, which are the ratio of running times (by using spin locks) between the sequential version and using multiple workers. The dotted lines show the perfect speedups as a baseline. The numbers beside the lines indicate the maximal speedups. A first look reveals that all experiments achieve speedups when using multiple cores, except for the *Max* case over *insert* and *Mixed* experiments.

Specifically, we make several observations:

– For all experiments, we observe that the speedups are around 1/4 to 1 when using 1 worker in all cases. This is because, for all operations of OM, the sequential version has the same work as the parallel version. Especially, for *Delete*, such speedups are low as 1/2 - 1/4, as locking items for deleting costs much running time.

– For *Insert* and *Mixed*, we achieve around 7x speedups using 32 workers in *No* and *Few* cases, and around 2x speedups using 4 workers in *Many* cases. This is because all CPU cores have to access the shared memory by the bus, which connects memory and cores, and the atomic `CAS` operations will lock the bus. Each `Insert` operation may have many atomic

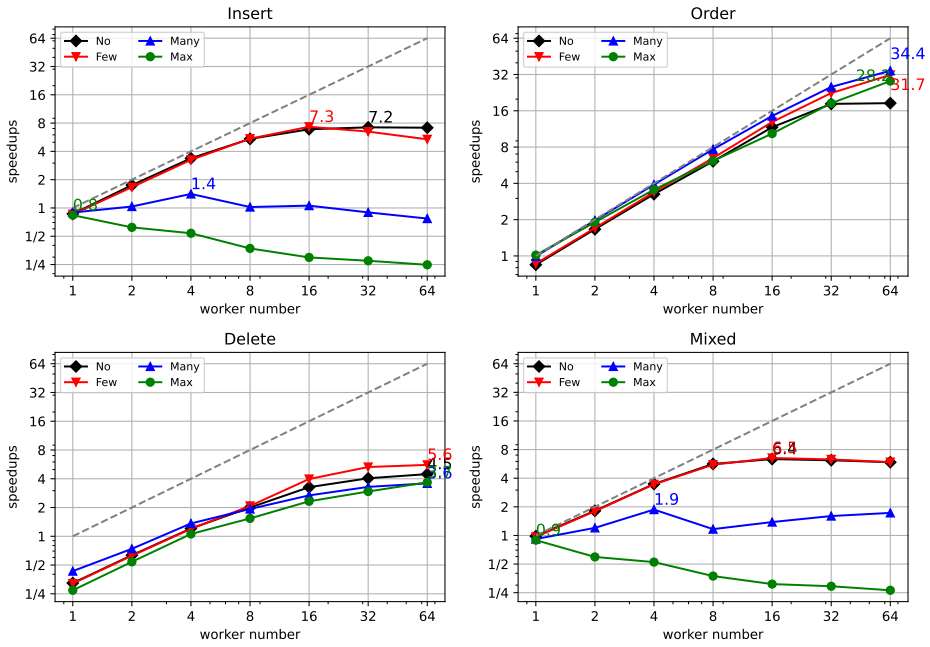


Figure 4.5: Evaluate the speedups by increasing the number of workers from 1 to 64.

`CAS` operations for spin lock and many atomic read and write operations for updating labels and lists. In this case, the bus traffic is high, which is the performance bottleneck for *Insert* operations.

– For *Order*, all four cases achieve almost perfect speedups from using 1 to 32 workers, as `Order` operations are lock-free.

– For *Delete*, it achieves around 4x speedups using 64 workers in four cases. This is because, for parallel `Delete` operations, the worst case, which is all operations are blocking as a chain, is almost impossible to happen.

4.5.4 Evaluating the Scalability

In this experiment, we increase the scale of the initial order list from 10 million to 100 million and evaluate running times with fixed 32 workers. We test three cases, *No*, *Few*, and *Many*, by fixing the average number of items per insert position. For example, given an initial order list with 20 million items, the *No* case has 20 million insert positions, the *Few* case has 2 million positions, and the *Many* case has 2,000 insert positions. Since the *Max* case is reduced to sequential and can be optimized by using a single worker, we omit it in this test.

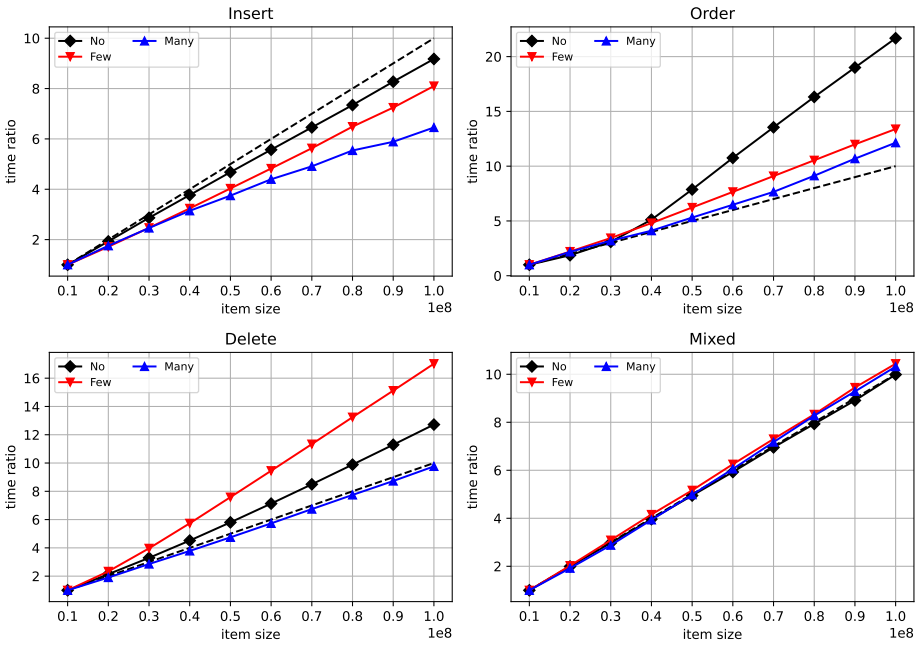


Figure 4.6: Evaluate the scalability by using 32 workers

The plots in Figure 4.6 depict the performance. The x-axis is the initial size of the order list, and the y-axis is the time ratio of the current running time to the "10 million" running time. The dotted lines show the perfect time ratio as a baseline. Obviously, the beginning time ratio is one. We observe that the time ratios are roughly close to linearly increasing with the scales of the order list. This is because all parallel `Insert`, `Delete`, and `Order` have best-case time complexity $O(\frac{m}{p})$ and their running times are always close to the best case.

Specifically, for *Order*, we can see the time ratio is up to 20x with a scale of 100 million in *No* case. This is because *No* case has 100 million positions for random `Order` operations, which is not cache friendly; also, by increasing the scale of data, the cache hit rate decreases, so the performance is affected.

4.5.5 Evaluating the Stability

In this experiment, by using 32 workers, we repeatedly test *Insert*, *Order*, and *Delete* operations 100 times, to evaluate the stability. Each time, we randomly choose positions and randomly insert items for the *NO*, *Few*, and *Many* cases, so that the test is different. However, it is always the same for the *Max* case, since there is only one position to insert all items. The plots in Figure 6.6 depict the running time by performing the experiments 100 times. The x-axis is the index of repeating times and the y-axis is the running times

(milliseconds). We observe that the performance of *Insert*, *Order*, *Delete*, and *Mixed* are all well-bounded for all four cases. We observe that the *Max* case has wider variation than other cases over *Insert* and *Mixed*. This is because the parallel `Insert` operations always have contention on shared data in memory. Such contention causes the running times to variate within a certain range.

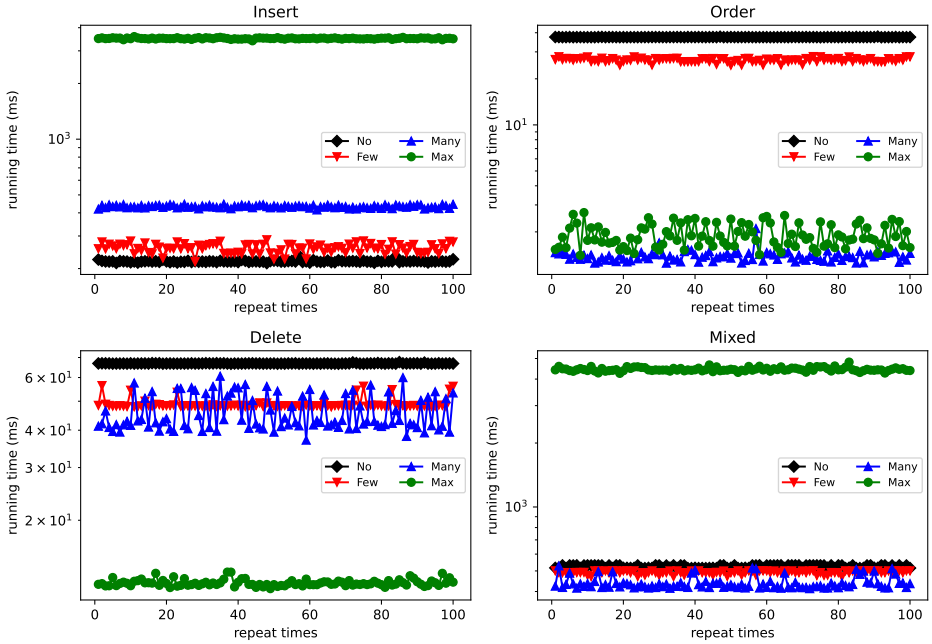


Figure 4.7: Evaluate the stability of running times over 32 works by repeatedly testing all operations 100 times.

Chapter 5

Sequential Core Maintenance

Graph analytics attract much attention from both research and industrial communities. Due to its linear time complexity, the k-core decomposition is widely used in many real-world applications such as biology, social networks, community detection, ecology, and information spreading. In many such applications, the data graphs continuously change over time. The changes correspond to edge insertion and removal. Instead of recomputing the k-core, which is time-consuming, we study how to maintain the k-core efficiently. That is, when inserting or deleting an edge, we need to identify the affected vertices by searching for more vertices. The state-of-the-art order-based method maintains an order, the so-called k-order, among all vertices, which can significantly reduce the searching space. However, this order-based method is complicated for understanding and implementing, and its correctness is not formally discussed.

In this chapter, we propose a simplified order-based approach by introducing the classical Order Data Structure to maintain the k-order, which significantly improves the worst-case time complexity for both edge insertion and removal algorithms. Also, our simplified method is intuitive to understand and implement; it is easy to argue the correctness formally. Additionally, we discuss a simplified batch insertion approach. The experiments evaluate our simplified method over 12 real and synthetic graphs with billions of vertices. Compared with the existing method, our simplified approach achieves high speedups up to 7.7x and 9.7x for edge insertion and removal, respectively.

5.1 Introduction

Given an undirected dynamic graph $G = (V, E)$, after inserting an edge into or removing an edge from G, the problem is how to efficiently update the core number for the affected vertices. To do this, we first need to identify a set of vertices whose core numbers need to be updated (V^*) by traversing a possibly

larger set of vertices (V^+). Then it is easy to re-compute the new core numbers of vertices in V^*. Clearly, an efficient edge insertion algorithm should have a small cost for identifying V^*, which means a small ratio $|V^+|/|V^*|$. In this work, we mainly discuss the edge insertion algorithms for core maintenance.

In (Sarıyüce et al., 2016), Sariyüce et al. propose a Traversal algorithm. The Traversal insertion algorithm searches V^* only in a local region near the edge that is inserted, which can be much faster than recomputing the core numbers for the whole graph. However, this insertion algorithm has a high variation in terms of performance due to the high variation of the ratio $|V^+|/|V^*|$. In (Zhang et al., 2017), Zhang et al. propose an Order algorithm, which is the state-of-the-art method for core maintenance.

However, this Order approach has two drawbacks. First, the Order insertion algorithm is so complicated that it is not intuitive for easy understanding. This complexity further brings difficulties to the correctness and implementation; actually, the proof of correctness for the edge-insert algorithm is not formally discussed in (Zhang et al., 2017). Second, the k-order of the vertices in a graph is maintained by two specific data structures: 1) $\mathcal{A}$ (double linked lists combined with balanced binary search trees) for operations like inserting, deleting, comparing the order of two vertices, all of which requires worst-case $O(\log|V|)$ time; and 2) $\mathcal{B}$ (double linked lists combined with heaps) for searching the ordered vertices by jumping unnecessary ones, which requires worst-case $O(\log|V|)$ time; both data structures are complicated to implement.

We try to overcome the above drawbacks in (Zhang et al., 2017) by proposing our Simplified-Order approach. The idea behind our new approach is that we introduce the well-known Order Maintenance Data Structure (Dietz and Sleator, 1987; Bender et al., 2002) to maintain the k-order of vertices in a graph G. By doing this, there are several benefits. First, this classical OM data structure only requires amortized $O(1)$ time for order operations, including inserting, deleting, and comparing the order of two vertices; this is faster than the $\mathcal{A}$ data structure in (Zhang et al., 2017) especially when $|V|$ is large. Also, the original order-based insertion algorithm can be introduced to maintain each affected vertex in k-order in worst-case $O(log|E^+|)$ time ($|E^+|$ is the number of edges adjacent to vertices in V^+); this is also faster than the $\mathcal{B}$ data structure in (Zhang et al., 2017) since normally we have $|E^+| \ll |V|$. Second, compared with the method in (Zhang et al., 2017), when introducing the OM data structures and priority queues, the $\mathcal{A}$ and $\mathcal{B}$ data structures can be abandoned and so that the Order approach can be significantly simplified; also, our new approach simplifies the proof of correctness. Finally, our simplified order-based insertion algorithm can be easily extended to handle a batch of insertion edges without difficulties since it is common that a great number of edges are inserted or removed simultaneously; by doing this, the

vertices in $V^+ \setminus V^*$ are possibly avoided to be repeatedly traversed so that the total size of V^+ is smaller compared to unit insertion. The main contributions are summarized below:

- We investigate the drawbacks of the state-of-the-art ORDER core maintenance algorithms in (Zhang et al., 2017).
- Based on (Zhang et al., 2017), by introducing the OM data structure (Dietz and Sleator, 1987; Bender et al., 2002), we propose a simplified order-based insertion algorithm. Not only can the worst-case time complexity be improved, but also the proof of correctness is simplified.
- We extend our simplified core insertion algorithm to handle a batch of edges, with a smaller size of V^* compared to unit insertion.
- Finally, we conduct extensive experiments with different kinds of real data graphs to evaluate different algorithms.

5.2 Related Work

In (Cheng et al., 2011), Cheng et al. propose an external memory algorithm, so-called EMcore, which runs in a top-down manner such that the whole graph does not have to be loaded into memory. In (Wen et al., 2016), Wen et al. provide a semi-external algorithm, which requires $O(n)$ size memory to maintain the information of vertices. In (Khaouid et al., 2015), Khaouid et al. investigate the core decomposition in a single PC over large graphs by using `GraphChi` and `WebGraph` models. In (Montresor et al., 2012), Montresoret et al. consider the core decomposition in a distributed system. In addition, Parallel computation of core decomposition in multi-core processors is first investigated in (Dasari et al., 2014), where the ParK algorithm was proposed. Based on the main idea of ParK, a more scalable PKC algorithm has been reported in (Kabir and Madduri, 2017).

In (Li et al., 2013), an algorithm that is similar to TRAVERSAL algorithm (Sarıyüce et al., 2016) is given, but this solution has quadratic time complexity. In (Wen et al., 2016), a semi-external algorithm for core maintenance is proposed in order to reduce the I/O cost, but this method is not optimized for CUP time. In (Wang et al., 2017; Hua et al., 2019; Jin et al., 2018; Wang et al., 2017), parallel approaches for core maintenance are proposed for both edge insertion and removal. There exists some research based on core maintenance. In (Yu et al., 2021), the authors study computing all k-cores in the graph snapshot over the time window. In (Lin et al., 2021), the authors explore the hierarchical core maintenance.

5.3 Preliminary

Definition 5.3.1 (k-Core)**.** Given an undirected graph $G = (V, E)$ and an integer k, a subgraph G_k of G is called a k-core if it satisfies the following conditions: (1) for $\forall u \in V(G_k)$, $u.deg \geq k$; (2) G_k is maximal. Moreover, $G_{k+1} \subseteq G_k$, for all $k \geq 0$, and G_0 is just G.

Definition 5.3.2 (Core Number)**.** Given an undirected graph $G = (V, E)$, the core number of a vertex $u \in G(V)$, denoted as $u.core$, is defined as $u.core = max\{k : u \in V(G_k)\}$. That means $u.core$ is the largest k such that there exists a k-core containing u.

Definition 5.3.3 (Subcore)**.** Given an undirected graph $G = (V, E)$, a maximal set of vertices $S \subseteq V$ is called a k-subcore if (1) $\forall u \in S, u.core = k$; (2) the induced subgraph $G(S)$ is connected. The subcore that contains vertex u is denoted as $\texttt{sc}(u)$.

Definition 5.3.4 (k-Core Decomposition)**.** Given a graph $G = (V, E)$, the problem of computing the core number for each $u \in V(G)$ is called core decomposition.

In (Batagelj and Zaversnik, 2003), Batagelj and Zaversnik propose an algorithm with a linear running time of $O(m + n)$, the so-called BZ algorithm. The general idea is the *peeling process*. That is, to compute the k-core G_k of G, the vertices (and their adjacent edges) whose degrees are less than k are repeatedly removed. When there are no more vertices to remove, the resulting graph is the k-core of G.

Algorithm 14: BZ algorithm for core decomposition

input : an undirected graph $G = (V, E)$
output: the core number $u.core$ for each $u \in V$

```
1 for u ∈ V do u.d ← |u.adj|; u.core = ∅
2 Q ← a min-priority queue by u.d for all u ∈ V
3 while Q ≠ ∅ do
4 |   u ← Q.dequeue()
5 |   u.core ← u.d; remove u from G
6 |   for v ∈ u.adj do
7 |   |   if u.d < v.d then v.d ← v.d − 1
8 |   update Q
```

Algorithm 14 shows the steps of the BZ algorithm. In initialization, for each vertex $u \in V$, the auxiliary degree $u.d$ is set to $|u.adj|$ and the core number $u.core$ is not identified (line 1). The postcondition is that for each vertex $u \in V$, the $u.d$ equals to the core number, formally $u.d = u.core$. We

state informally lines 3 - 8 as a loop invariant: (1) the vertex u always has the minimum degree $u.d$ since u is removed from the min-priority queue Q (line 4); and (2) if u obtains its core number, $u.core$ equals to $u.d$ (line 5). The key step is updating $v.d$ for all $v \in u.adj$. That is, $v.d$ are decremented by 1 if $u.d$ is smaller than $v.d$ (lines 6 and 7). In this algorithm, the min-priority queue Q can be efficiently implemented by bucket sorting (Batagelj and Zaversnik, 2003), by which the total running time is optimized to linear $O(m + n)$.

Definition 5.3.5 (Core Maintenance for Edge Insertion)**.** Given an undirected graph $G = (V, E)$, the candidate set V^* is identified after an edge is inserted. The core numbers of vertices in V^* are increased.

Definition 5.3.6 (Core Maintenance for Edge Deletion)**.** Given an undirected graph $G = (V, E)$, the candidate set V^* is identified after an edge is removed. The core numbers of vertices in V^* are decreased.

Definition 5.3.7 (Candidate Set V^* and Searching Set V^+)**.** Given an undirected graph $G = (V, E)$, when an edge is inserted or removed, a candidate set of vertices, denoted as V^*, have to be computed so that the core numbers of all vertices in V^* must be updated. In order to identify V^*, a minimal set of searching vertices, denoted as V^+, is traversed by repeatedly accessing their adjacent edges.

Definition 5.3.7 says that V^* is identified by traversing a minimum number of vertices in V^+, so that V^* has to belong to V^+, denoted as $V^* \subseteq V^+$. Further, the vertices $v \in V^+ \setminus V^*$, are searching vertices but not candidate vertices. Efficient core maintenance algorithms should have a small ratio of $|V^+|/|V^*|$ in order to minimize the cost of computing V^*. After V^* is identified, the core number of vertices in V^* can be updated accordingly. In (Zhang and Yu, 2019b), Zhang et al. prove that the core maintenance is asymmetric: the edge removal is bounded for $V^* = V^+$, but the edge insertion is unbounded for $V^* \subseteq V^+$. In other words, to identify V^*, the edge removal only needs to traverse V^*; however, the edge insertion may traverse a much larger set of vertices than V^*.

Example 5.3.1. Consider the graph G in Figure 5.1. The numbers inside the vertices are the core numbers. Three vertices, v_1 to v_3, have same core numbers of 2; the other vertices, u_1 to u_{1000}, have same core numbers of 1. The whole graph G is the 1-core of since each vertex has a degree of at least 1; the subgraph induced by $\{v_1, v_2, v_3\}$ is the 2-core since each vertex in this subgraph has a degree of at least 2. After inserting an edge, for example (u_1, u_{500}), we observe that the core numbers of all vertices are not changed according to the peeling process. In this case, the candidate set $V^* = \emptyset$. However, the searching set V^+ is different for different edge insertion algorithms, e.g., the

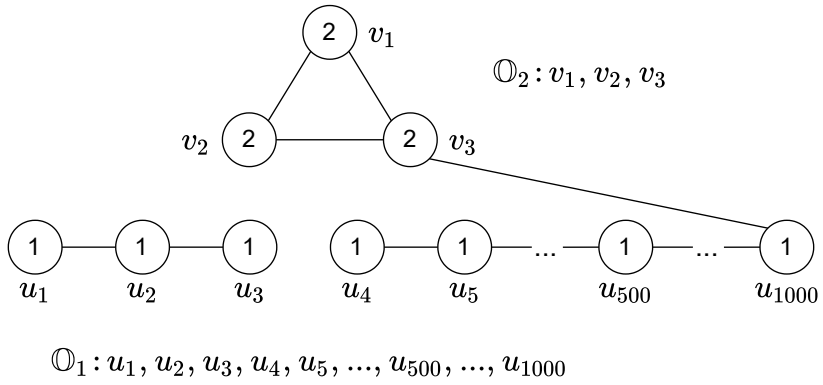


Figure 5.1: A sample graph G with $\mathbb{O} = \mathbb{O}_1\mathbb{O}_2$ in k-order.

order-based algorithm may have $V^+ = \{u_1, u_2, u_3\}$ and the traversal algorithm may traverse all vertices in $\mathtt{sc}(u_1)$ and $\mathtt{sc}(u_{500})$ with $V^+ = \{u_1, u_2, \ldots, u_{1000}\}$.

We present two theorems given in (Li et al., 2013; Sarıyüce et al., 2016; Zhang et al., 2017).

Theorem 5.3.1. (Li et al., 2013; Sarıyüce et al., 2016; Zhang et al., 2017) After inserting an edge in or removing an edge from $G = (V, E)$, the core number of a vertex $u \in V^*$ increases or decreases by at most 1, respectively.

Theorem 5.3.2. (Li et al., 2013; Sarıyüce et al., 2016; Zhang et al., 2017) Suppose an edge (u, v) with $K = u.core \leq v.core$ is inserted to (resp. removed from) G. Suppose V^* is non-empty. We have the following: (1) if $u.core < v.core$, then $u \in V^*$ and $V^* \subseteq \mathtt{sc}(u)$ (as in Definition 5.3.3); (2) if $u.core = v.core$, then both vertices u and v are in V^* (resp. at least one of u and v is in V^*) and $V^* \subseteq \mathtt{sc}(u) \cup \mathtt{sc}(v)$; (3) the induced subgraph of $V^* \in G \cup \{(u, v)\}$ is connected.

Theorem 5.3.2 suggests that: (1) V^* only includes the vertices $u \in V$ with $u.core = K$; (2) V^* can be searched in a small local region near the inserted or removed edge rather than in a whole graph. That is, to identify V^*, all vertices in V^+ are located in the subcores containing u and v.

5.3.1 The Order-Based Core Maintenance

In this section, we discuss the state-of-the-art sequential order-based core maintenance approach in (Zhang et al., 2017), so-called the Order algorithm. It is based on the k-order, which can be generated by the BZ algorithm for core decomposition (Batagelj and Zaversnik, 2003) as in Algorithm 14. The k-order is defined as follows.

Definition 5.3.8 (k-Order $\preceq$)**.** (Zhang et al., 2017) Given a graph G, the k-order $\preceq$ is defined for any pairs of vertices u and v over the graph G as follows: (1) when $u.core < v.core$, $u \preceq v$; (2) when $u.core = v.core$, $u \preceq v$ if u's core number is determined before v's by the BZ algorithm (Algorithm 14, line 5).

A k-order $\preceq$ is an instance of all the possible vertex sequences produced by Algorithm 14. For the k-order, transitivity holds, that is, $u \preceq v$ if $u \preceq w \wedge w \preceq v$. For each edge insertion and removal, the k-order will be maintained.

Here, $\mathbb{O}_k$ denotes the sequence of vertices in k-order whose core numbers are k. A sequence $\mathbb{O} = \mathbb{O}_0\mathbb{O}_1\mathbb{O}_2\cdots$ over $V(G)$ can be obtained, where $\mathbb{O}_i \preceq \mathbb{O}_j$ if $i < j$. It is clear that $\preceq$ is defined over the sequence of $\mathbb{O} = \mathbb{O}_0\mathbb{O}_1\mathbb{O}_2\cdots$. In other words, for all vertices in graph, the sequence $\mathbb{O}$ indicates the k-order $\preceq$.

Example 5.3.2. Continually consider the graph G in Figure 5.1. The numbers inside the vertices are the core numbers. The k-order of G is shown by $\mathbb{O}_1$ and $\mathbb{O}_2$, which is the order of core numbers determined by the BZ algorithm (Algorithm 14 line 5); also, $\mathbb{O}_1$ is determined before $\mathbb{O}_2$, so that we have $\mathbb{O}_1 \preceq \mathbb{O}_2$.

Edge Insertion. The key step for the insertion algorithm is to determine V^*. To do this, two degrees, $u.d^+$ and $u.d^*$, for each vertex $u \in V(G)$ are maintained in order to identify whether u can be added into V^* or not:

- remaining degree $u.d^+$: the number of the neighbors after vertex u in $\mathbb{O}$ that can potentially support the increment of the current core number.
- candidate degree $u.d^*$: the number of the neighbors before vertex u in $\mathbb{O}$ that can potentially have their core number increased.

Assume that an edge (u, v) is inserted with $K = u.core \leq v.core$. The intuition behind the order-based insertion algorithm is as follows. Starting from u, all affected vertices with the same core number K (Theorem 5.3.2) are traversed in $\mathbb{O}$. For each visited vertex $w \in V^+$, the value of $w.d^* + w.d^+$ is maximal as w is visited by k-order. In this case, w will be added into V^* if $w.d^*+w.d^+ > K$; otherwise, w is impossibly in V^*, which may repeatedly cause other vertices to be removed from V^*. When all vertices with core number K are traversed, this process terminates and V^* is identified. Finally, the core numbers for all vertices in V^* are updated by increasing by 1 (Theorem 5.3.1). Obviously, for all vertices $u \in V$, the order $\mathbb{O}$ along with $u.d^+$ and $u.d^*$ must be maintained accordingly.

Compared with the Traversal insertion algorithm (Sarıyüce et al., 2016), the benefit of traversing with k-order is that a large number of unnecessary

vertices in $V^+ \setminus V^*$ can be avoided. This is why the ORDER insertion algorithm is generally more efficient.

The ORDER insertion algorithm is not easy to implement as it needs to traverse the vertices in $\mathbb{O}$ efficiently. There are three cases. First, given a pair of vertices $u, v \in \mathbb{O}_k$, the order-based insertion algorithm needs to efficiently test whether $u \preceq v$ or not. For this, $\mathbb{O}_k$ is implemented as a double linked list associated with a data structure $\mathcal{A}_k$ which is a binary search tree and each tree node holds one vertex. For all $u, v \in \mathbb{O}_k$, we can test the order $u \preceq v$ in $O(\log|\mathbb{O}_k|)$ time by using $\mathcal{A}_k$. Second, the order-based insertion algorithm needs to efficiently "jump" over a large number of non-affected vertices that have $u.d^* = 0$. To do this, $\mathbb{O}_k$ is also associated with a data structure $\mathcal{B}$, which is a min-heap. Here, $\mathcal{B}$ supports finding a affected vertex u with $u.d^* > 0$ sequentially in $\mathbb{O}_k$ with $O(1)$ time; but it requires $O(\log|\mathbb{O}_k|)$ time to maintain the min-heap. Therefore, when maintaining $\mathbb{O}$, both $\mathcal{A}$ and $\mathcal{B}$ requires to updated accordingly, which requires worst-case $O(|V^+|\cdot \log|\mathbb{O}_k| + O(|V^*|)\log|\mathbb{O}_{k+1}|)$ time for removing $v \in V^*$ from $\mathbb{O}_k$ and then inserting $v \in V^*$ at the head of $\mathbb{O}_{k+1}$.

As we can see, the $\mathcal{A}$ and $\mathcal{B}$ data structures are complicated, which complicates understanding and implementation. Additionally, the operations on $\mathcal{A}$ and $\mathcal{B}$ are time-consuming, especially when handling a data graph with a large sizes of $\mathbb{O}_k$ or $\mathbb{O}_{k+1}$.

Edge Removal. The order-based removal algorithm adopts the same routine used in the traversal removal algorithm (Sarıyüce et al., 2016) to compute V^*. This order-based removal algorithm is based on the *max-core degree*.

Definition 5.3.9 (max-core degree *mcd*)**.** (Sarıyüce et al., 2016; Zhang et al., 2017) Given a graph $G = (V, E)$, for each vertex $v \in V$, the max-core degree, $v.mcd$, is the number of v's neighbors w such that $w.core \geq v.core$, defined as $v.mcd = |\{w \in v.adj : w.core \geq v.core\}|$.

As discussed, the edge removal is much simpler than the edge insertion since edge removal is bounded for $V^* = V^+$. Assuming an edge (u, v) is removed from the graph, both $u.mcd$ and $v.mcd$ are updated accordingly. This may repeatedly affect other adjacent vertices' *mcd*. When the process terminates, all affected vertices u that have $u.mcd < u.core$ can be added into V^* and then their core numbers are off by 1. Obviously, for all vertices $u \in V^*$, the sequence $\mathbb{O}$ along with $u.mcd$ must be maintained accordingly.

Compared with the TRAVERSAL removal algorithm (Sarıyüce et al., 2016), the difference is that the ORDER removal algorithm needs to maintain the k-order $\mathbb{O}$ for all vertices in V^*. That is, all vertices in V^* with core number k are deleted from $\mathbb{O}_k$ and then appended to $\mathbb{O}_{k-1}$ in the corresponding k-order. Recall that two associated data structures, $\mathcal{A}$ and $\mathcal{B}$, are used for the order-based insertion algorithm. Both $\mathcal{A}$ and $\mathcal{B}$ must be updated accordingly, which

requires worst-case $O(|V^*|\cdot(\log|O_k|+\log|O_{k-1}|))$ time for removing $v \in V^*$ from $\mathbb{O}_k$ and appending $v \in V^*$ at the tail of $\mathbb{O}_{k-1}$. Analogously to the order-based insertion, the operations on $\mathcal{A}$ and $\mathcal{B}$ are time-consuming when handling a data graph with a large size of $\mathbb{O}_k$ or $\mathbb{O}_{k-1}$.

The ORDER removal algorithm is presented in Algorithm 15. After an edge is removed, the affected vertices, u and v, have to be put into V^* if their *mcd* less than *core* (lines 2 to 4), which may repeatedly cause the other vertices' *mcd* decrease and then added to V^* (lines 5 to 9). The queue R is used to propagate the vertices added to V^* whose *mcd* are less than their core numbers (lines 5 and 6). The k-order is maintained for inserting an edge next time (line 11). Also, all vertices' *mcd* have to be updated for removing an edge next time (line 12).

Algorithm 15: RemoveEdge$(G, \mathbb{O}, (u, v))$

```
1  R, K, V* ← an empty queue, Min(u.core, v.core), ∅
2  remove (u, v) from G⃗ with updating u.mcd and v.mcd
3  if u.mcd < K then V* ← V* ∪ {u}; R.enqueue(u)
4  if v.mcd < K then V* ← V* ∪ {v}; R.enqueue(v)
5  while R ≠ ∅ do
6  |   w ← R.dequeue()
7  |   for w' ∈ w.adj with w'.core = K ∧ w ∉ V* do
8  |   |   w'.mcd ← w'.mcd − 1
9  |   |   if w'.mcd < K then V* ← V* ∪ {w'}; R.enqueue(w')
10 for w ∈ V* do w.core ← w.core − 1
11 Remove all w ∈ V* from O_K and append to O_{K−1} in k-order
12 update mcd for all related vertices accordingly
```

5.4 The Simplified Order-Based Algorithm

The main reason for the order-based algorithm being complicated and inefficient is that two data structures, $\mathcal{A}$ and $\mathcal{B}$, are used to maintain $\mathbb{O}$ in k-order for all vertices in a graph. In this section, we adopt the Order Maintenance (OM) data structure (Dietz and Sleator, 1987; Bender et al., 2002) to maintain the k-order for all vertices. There are two benefits: one is that the k-order operations, such as inserting, deleting, and comparing the order of two vertices, can be optimized to $O(1)$ amortized running time; the other is that the original order-based method (Zhang et al., 2017) can be simplified, which makes it easier to implement and to discuss the correctness.

Before introducing the new method, we propose a *constructed Directed Acyclic Graph (DAG)* to simplify the statement of our algorithms. Given an

undirected graph $G = (V, E)$ with $\mathbb{O}$ in k-order, each edge $(u, v) \in E(G)$ can be assigned a direction such that $u \preceq v$. By doing this, a *direct acyclic graph* (DAG) $\vec{G} = (V, \vec{E})$ can be constructed where each edge $u \mapsto v \in \vec{E}(\vec{G})$ satisfies $u \preceq v$. Of course, the k-order of G is the *topological order* of $\vec{G}$. The *post* of a vertex v in $\vec{G}(V, \vec{E})$ is all its successors (outgoing edges), defined by $u(\vec{G}).post = \{v \mid u \mapsto v \in \vec{E}(\vec{G})\}$; the *pre* of a vertex v in $\vec{G}(V, \vec{E})$ is all its predecessors (incoming edges), defined by $u(\vec{G}).pre = \{v \mid v \mapsto u \in \vec{E}(\vec{G})\}$. When the context is clear, we use $u.post$ instead of $u(\vec{G}).post$ and $u.pre$ instead of $u(\vec{G}).pre$.

In other words, the constructed DAG $\vec{G} = (V, \vec{E})$ is equivalent to the undirected graph $G(V, E)$ by associating the direction for each edge in k-order. This newly defined constructed DAG $\vec{G}$ is convenient for describing our simplified order-based insertion algorithm.

Lemma 5.4.1. *Given a constructed DAG $\vec{G} = (V, \vec{E})$, for each vertex $v \in V$, the out-degree $|v.post|$ is not greater than the core number, $|v.post| \leq v.core$.*

Proof. Since the topological order of $\vec{G}$ is the k-core of G, when removing the vertex v by executing the BZ algorithm (Algorithm 14, line 5) all the vertices in $v.pre$ are already removed. In such a case, the out-degree of v is its current degree. If there exist $|v.post| > v.core$ the value $v.core$ should equal to $|v.post|$, which leads to a contradiction. □

If inserting an edge into a constructed DAG $\vec{G}$ does not violate Lemma 5.4.1, no maintenance operations are required. Otherwise, $\vec{G}$ has to be maintained to re-establish Lemma 5.4.1.

5.4.1 The Simplified Order-Based Insertion

Theory Background. With the concept of the constructed DAG $\vec{G}$, we can introduce our simplified insertion algorithm to maintain the core numbers after an edge is inserted to $\vec{G}$. For convenience, based on the constructed DAG $\vec{G}$, we first redefine the two concepts of candidate degree and remaining degree as in (Zhang et al., 2017).

Definition 5.4.1 (candidate in-degree)**.** Given a constructed DAG $\vec{G}(V, \vec{E})$, the candidate in-degree $v.d^*_{in}$ is the total number of its predecessors located in V^*, denoted as

$$v.d^*_{in} = |\{w \in v.pre : w \in V^*\}|$$

Definition 5.4.2 (remaining out-degree)**.** Given a constructed DAG $\vec{G}(V, \vec{E})$, the remaining out-degree $v.d^+_{out}$ is the total number of its successors without the ones that are confirmed not in V^*, denoted as

$$v.d^+_{out} = |\{w \in v.post : w \notin V^+ \setminus V^*\}|$$

In other words, assuming that $K = v.core$, the candidate in-degree $v.d^*_{in}$ counts the number of predecessors that are already in the new $(K+1)$-core; $v.d^+_{out}$ counts the number of successors that can be in the new $(K+1)$-core. Therefore, $v.d^*_{in} + v.d^+_{out}$ upper bounds the number of v's neighbors in the new $(K+1)$-core.

Theorem 5.4.2. *Given a constructed DAG $\vec{G} = (V, \vec{E})$ by inserting an edge $u \mapsto v$ with $K = u.core \leq v.core$, the candidate set V^* includes all possible vertices that satisfy: 1) their core numbers equal to K, and 2) their total numbers of candidate in-degree and remaining out-degree are greater than K, denoted as*

$$\forall w \in V : w \in V^* \equiv (w.core = K \ \wedge \ w.d^*_{in} + w.d^+_{out} > K)$$

Proof. According to Theorem 5.3.1 and Theorem 5.3.2, for all vertices in V^*, we have 1) their core numbers equal to K, and 2) their core numbers will increase to $K+1$ and they can be added to new $(K+1)$-core. By the definition of k-core, for a vertex $v \in V^*$, v must have at least $K+1$ adjacent vertices that can be in the new $(K+1)$-core. As $v.d^*_{in} + v.d^+_{out}$ is the number of v's adjacent vertices that can be in the new $(K+1)$-core, we get $v.d^*_{in} + v.d^+_{out} > K$ for all vertices $v \in V^*$. □

Theorem 5.4.3. *Given a constructed DAG $\vec{G} = (V, \vec{E})$ by inserting an edge $u \mapsto v$ with u in $\mathbb{O}_K$, all affected vertices w are after u in $\mathbb{O}_K$. Starting from u, when w is traversed in $\mathbb{O}_K$ and the V^+, V^*, $w.d^*_{in}$, $w.d^+_{out}$ are updated accordingly, each time the value of $w.d^*_{in} + w.d^+_{out}$ is maximal.*

Proof. For all the vertices in the constructed DAG $\vec{G}$, $\mathbb{O}$ is the topological order in $\vec{G}$ according to the definition of $\vec{G}$. When traversing affected vertices w in G in such topological order, each time for w all the affected predecessors must have been traversed, so that we get the value of $w.d^*_{in}$ is maximal; also, all the related successors are not yet traversed, so that the value of $w.d^+_{out}$ is also maximal. Therefore, the total value of $w.d^*_{in} + w.d^+_{out}$ is maximal. □

In other words, when traversing the affected vertices w in $\mathbb{O}$, $w.d^*_{in} + w.d^+_{out}$ is the upper bound. That means, when traversing the vertices after w in $\mathbb{O}$, $w.d^*_{in} + w.d^+_{out}$ only can be decreased as some vertices can be removed from V^*. In this case, we can safely remove w from V^* if $w.d^*_{in} + w.d^+_{out} \leq K$, since w is impossibly in V^* according to Theorem 5.4.2. This is the key idea behind the order-based insertion algorithm.

The Algorithm. Algorithm 16 shows the detailed steps when inserting an edge $u \mapsto v$. One issue is the implementation of traversing the vertices in $\mathbb{O}_k$. We propose to use a Min-Priority Queue combined with the OM data structure

(line 4). The idea is as follows: 1) $\mathbb{O}_k$ is maintained by the OM data structure (Dietz and Sleator, 1987; Bender et al., 2002), by which each vertex is assigned a label (an integer number) to indicate the order, and 2) all adjacent vertices are added into a Min-Priority Queue by using such labels as their keys. By doing this, each time we can dequeue a vertex from the Min-Priority Queue to "jump" over not-affected vertices efficiently. Further, three colors are used to indicate the different status for each vertex v in a graph:

- `white`: v has initial status, $v \notin V^* \wedge v \notin V^+$.
- `black`: v is traversed and identified as a candidate vertex, $v \in V^* \wedge v \in V^+$.
- `gray`: v is traversed and identified impossibly as a candidate vertex, $v \notin V^* \wedge v \in V^+ \equiv v \in V^+ \setminus V^*$.

Before executing, we assume that for all vertices $v \in V(\vec{G})$ their d^+_{out} and d^*_{in} are correctly maintained, that is $v.d^+_{out} = |v.post| \ \wedge \ v.d^*_{in} = 0$. Initially, both V^* and V^+ are empty (all vertices are `white`) and K is initialized to $u.core$ since $u \preceq v$ for $u \mapsto v$ (line 1). After inserting an edge $u \mapsto v$ with $u \preceq v$ in $\mathbb{O}$, we have $u.d^+_{out}$ increase by one (line 2). The algorithm will terminate if $u.d^+_{out} \leq u.core$ as Lemma 5.4.1 is satisfied (line 3). Otherwise, u is added into the Min-Priority Queue Q (line 4) for propagation (line 8 to 13). For each w removed from Q (line 6), we check the value of $w.d^*_{in} + w.d^+_{out}$. That is, if $w.d^*_{in} + w.d^+_{out} > K$, vertex w can be added to V^* and may cause other vertices added in V^*, which is processed by the `Forward` procedure (line 7). Otherwise, w cannot be added to V^*, which may cause some vertices to be removed from V^* processed by the `Backward` procedure (line 8). Here, $w.d^*_{in} > 0$ means w is affected, or else w can be omitted since w has no predecessors in V^* (line 8). When Q is empty, this process terminates and V^* is obtained (line 5). At the ending phase, for all vertices V^*, their core numbers are increased by one (by Theorem 5.3.1) and their d^*_{in} are reset (line 9). Finally, the $\mathbb{O}$ is maintained (line 10).

The detailed steps of the `Forward` procedure are shown in Algorithm 17. At first, u is added to V^* and V^+ (set from `white` to `black`) since u has $u.d^*_{in} + u.d^+_{out} > K$ (line 1). Then, for each u's successors v whose core numbers are equal to K (by Theorem 5.3.2), $v.d^*_{in}$ is increased by one (lines 2 and 3). In this case, v is affected and has to be added into Q for subsequent propagation (line 4).

The detail steps of the `Backward` procedure are shown in Algorithm 18. In the `DoPre`(u) procedure, for all u's predecessors v that are located in V^* (line 11), $v.d^+_{out}$ is decreased by one since u is set to `gray` and cannot be added into V^* any more (line 12); in this case, v has to be added into R for propagation if $v.d^*_{in} + v.d^+_{out} \leq K$(line 13). Similarly, in the `DoPost`(u) procedure, for all

Algorithm 16: InsertEdge($\vec{G}, \mathbb{O}, u \mapsto v$)

input : A DAG $\vec{G}(V, \vec{E})$; the corresponding $\mathbb{O}$; an edge $u \mapsto v$ to be inserted.
output: An updated DAG $\vec{G}(V, \vec{E})$; the updated $\mathbb{O}$.

1 $V^*, V^+, K \leftarrow \emptyset, \emptyset, u.core$ // all vertices are white
2 insert $u \mapsto v$ into $\vec{G}$ with $u.d^+_{out} \leftarrow u.d^+_{out} + 1$
3 **if** $u.d^+_{out} \leq K$ **then return**
4 $Q \leftarrow$ a min-priority queue by $\mathbb{O}$; $Q.enqueue(u)$
5 **while** $Q \neq \emptyset$ **do**
6 $\quad w \leftarrow Q.dequeue()$
7 $\quad$ **if** $w.d^*_{in} + w.d^+_{out} > K$ **then** `Forward`(w, Q, K)
8 $\quad$ **else if** $w.d^*_{in} > 0$ **then** `Backward`$(w, \mathbb{O}, K)$
// Ending Phase
9 **for** $w \in V^*$ **do** $w.core \leftarrow K + 1$; $w.d^*_{in} \leftarrow 0$
10 **for** $w \in V^*$ **do** remove w from $\mathbb{O}_K$ and insert w at the beginning of $\mathbb{O}_{K+1}$ in k-order (the order w added into V^*)

Algorithm 17: Forward(u, Q, K)

1 $V^* \leftarrow V^* \cup \{u\}$; $V^+ \leftarrow V^+ \cup \{u\}$ // u is white to black
2 **for** $v \in u.post$ **:** $v.core = K$ **do**
3 $\quad v.d^*_{in} \leftarrow v.d^*_{in} + 1$
4 $\quad$ **if** $v \notin Q$ **then** $Q.enqueue(v)$

u's successors v that have $v.d^*_{in} > 0$ (line 15), $v.d^*_{in}$ is decreased by one (line 16) and added into R for propagation if $v.d^*_{in} + v.d^+_{out} \leq K$ (lines 17 and 18).

The detailed steps of the `Backward` procedure are shown in Algorithm 18. The queue R is used for propagation (line 2). The `DoPre`(u) procedure updates the graph when setting u from `white` to `gray` or from `black` to `gray`, that is, for all u's predecessors in V^*, all d^+_{out} are off by 1 and then added to R for propagation, if its $d^*_{in} + d^+_{out} \leq K$ since they cannot be in V^* any more (lines 10 - 13). Similarly, the `DoPost` procedure updates the graph when setting u from `black` to `gray`, that is, for all u's successors with $d^*_{in} > 0$, all d^*_{in} are off by 1 and then added to R for propagation if it is in V^* and its $d^*_{in} + degout \leq K$ (lines 14 - 18). Now, we explain the algorithm step by step. At first, w is just added to V^+ (set from `white` to `gray`) since w has $w.d^*_{in} + w.d^+_{out} \leq K$ (line 1). The queue R is initialized as empty for propagation (line 2) and w is propagated by the `DoPre` procedure. Of course, w's d^+_{out} and d^*_{in} are updated (line 3) since all `black` vertices causing $w.d^*_{in}$ increased will be moved after w in $\mathbb{O}$ eventually. All the vertices in R are `black` waiting to be propagated (lines 26 to 31). For each $u \in R$, vertex u is removed from R (line 5) and removed from V^*, which sets u from `black` to `gray` (line 6). This may require d^*_{in} and

d^+_{out} of adjacent vertices to be updated, which is done by the procedures `DoPre` and `DoPost`, respectively (line 7). To maintain $\mathbb{O}_K$, u is first removed from $\mathbb{O}_K$ and then inserted after p in $\mathbb{O}_K$, where p initially is w or the previous moved vertices in $\mathbb{O}_K$ (line 8). Of course, u's d^+_{out} and d^*_{in} are updated (line 3) since all `black` vertices causing $u.d^*_{in}$ increased will be moved after w in $\mathbb{O}$ eventually. This process is repeated until R is empty (lines 26 to 31).

Algorithm 18: Backward$(w, \mathbb{O}, K)$

```
1  V+ ← V+ ∪ {w}; p ← w                                   // w is white to gray
2  R ← an empty queue; DoPre(w)
3  w.d+out ← w.d+out + w.d*in; w.d*in ← 0
4  while R ≠ ∅ do
5  |  u ← R.dequeue()
6  |  V* ← V* \ {u}                                      // u is black to gray
7  |  DoPre(u); DoPost(u)
8  |  DELETE (O_K, u); INSERT(O_K, p, u); p ← u
9  |  u.d+out ← u.d+out + u.d*in; u.d*in ← 0
10 procedure DoPre(u)
11 |  for v ∈ u.pre : v ∈ V* do
12 |  |  v.d+out ← v.d+out − 1
13 |  |  if v.d*in + v.d+out ≤ K ∧ v ∉ R then R.enqueue(v)
14 procedure DoPost(u)
15 |  for v ∈ u.post : v.d*in > 0 do
16 |  |  v.d*in ← v.d*in − 1
17 |  |  if v ∈ V* ∧ v.d*in + v.d+out ≤ K ∧ v ∉ R then
18 |  |  |  R.enqueue(v)
```

Example 5.4.1. Consider inserting an edge to a constructed graph in Figure 6.1 obtained from Figure 5.1. The numbers inside the vertices are the core numbers, and the two numbers beside the vertices u_1, u_2, u_3 and u_{500} are their $d^*_{in} + d^+_{out}$. Initially, we have the min-priority queue $Q = \emptyset$ and $K = 1$. In Figure 6.1(a), after inserting an edge $u_1 \mapsto u_{500}$, we get $u_1.d^+_{out} = 2 > K$ and therefore u_1 is added to Q as $Q = \{u_1\}$. We begin to propagate Q. First, in Figure 6.1(a), u_1 is removed from Q to do the `Forward` procedure since $u_1.d^+_{out} + u_1.d^*_{in} = 0 + 2 > K$, by which u_1 is colored by `black`, all $u_1.post$'s d^*_{in} add by 1, and all $u_1.post$ are put into Q as $Q = \{u_2, u_{500}\}$. Second, in Figure 6.1(b), u_2 is removed from Q to do the `Forward` procedure since $u_2.d^+_{out} + u_2.d^*_{in} = 1 + 1 > K$, by which u_2 is colored by `black`, all $u_2.post$'s d^*_{in} add by 1, and all $u_1.post$ are added into Q as $Q = \{u_3, u_{500}\}$. Third, in Figure 6.1(c), however, u_3 is removed from Q to do the `Backward` procedure since $u_3.d^*_{in} + u_3.d^+_{out} = 1 + 0 \leq K$, by which u_3 is colored by `gray` and we have $Q = \{u_{500}\}$.

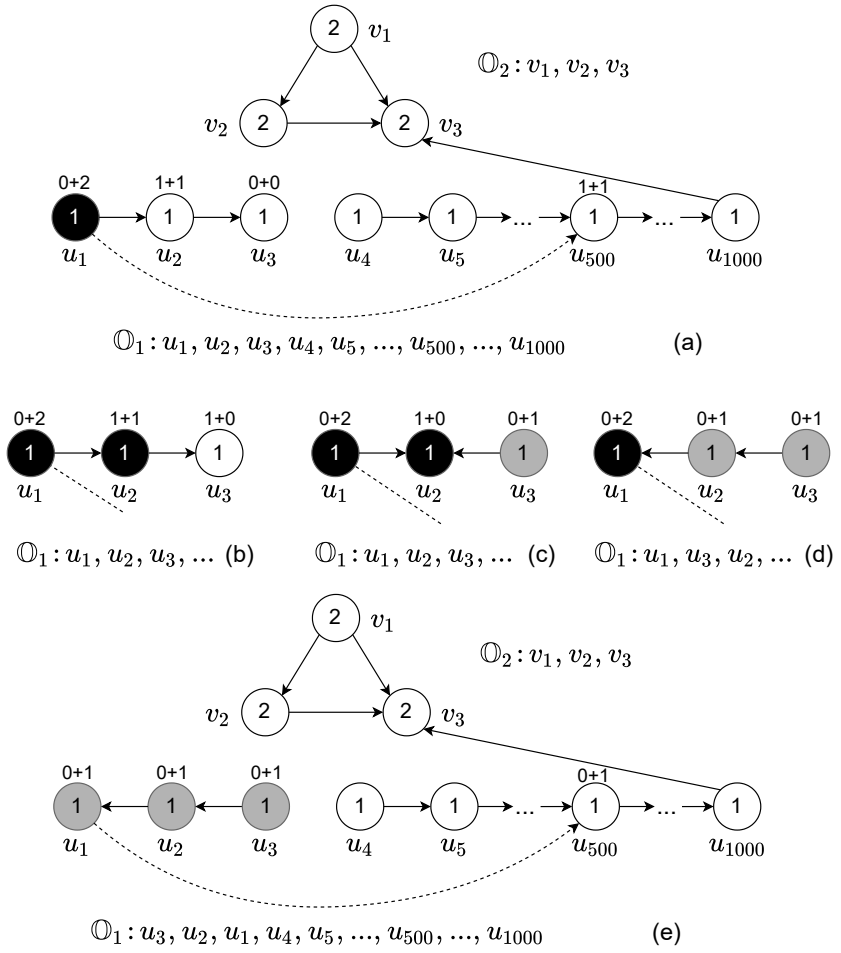


Figure 5.2: Insert one edge $u_1 \mapsto u_{500}$ to a constructed graph $\vec{G}$ obtained from Figure 5.1.

The `Backward` procedure continues. In Figure 6.1(d), we get $u_2.d^+_{out}$ off by 1 and $u_2.d^*_{in} + u_2.d^+_{out} = 1 + 0 \leq K$, so that u_2 is set to `gray`, by which u_2 is moved after u_3 in $\mathbb{O}_1$. In Figure 6.1(e), we get $u_1.d^+_{out}$ off by 1 and $u_1.d^*_{in} + u_1.d^+_{out} = 0 + 1 \leq K$, so that u_1 is also set to `gray`, by which u_1 is moved after u_3 in $\mathbb{O}_1$; also, we get $u_{500}.d^*_{in}$ off by 1 and the `Backward` procedure terminate. Finally, we still need to check the last u_{500} in Q, which can be safely omitted since its d^*_{in} is 0. In this simple example, we have $V^* = \emptyset \wedge V^+ = \{u_1, u_2, u3\}$ and only 4 vertices added to Q. A large number of vertices in $\mathbb{O}_1$, e.g. $u_4 \ldots u_{1000}$, are avoided being traversed.

Correctness. The key issue of the Algorithm 16 is to identify the candidate set V^*. For correctness, the algorithm has to be sound and complete. The

soundness implies that all vertices in V^* are correctly identified,

$$sound(V^*) \equiv \forall v \in V : v \in V^* \Rightarrow v.d^*_{in} + v.d^+_{out} > K \ \wedge \ v.core = K$$

The completeness implies that all possible candidate vertices are added into V^*,

$$complete(V^*) \equiv \forall v \in V : v.d^*_{in} + v.d^+_{out} > K \ \wedge \ v.core = K \ \Rightarrow \ v \in V^*$$

The algorithm has to ensure both soundness and completeness

$$sound(V^*) \ \wedge complete(V^*),$$

which is equivalent to

$$\forall v \in V : v \in V^* \ \equiv \ v.d^*_{in} + v.d^+_{out} > K \ \wedge \ v.core = K$$

To argue the soundness and completeness, we first define the vertices in $V(G)$ to have correct candidate in-degrees and remaining out-degrees as

$$in^*(V) \equiv \forall v \in V : v.d^*_{in} = |\{w \in v.pre : w \in V^*\}|$$
$$out^+(V) \equiv \forall v \in V : v.d^+_{out} = |\{w \in v.post : w \notin V^+ \setminus V^*\}|$$

We also define the sequence $\mathbb{O}$ for all vertices in V are in k-order as

$$\forall v_i \in V : \mathbb{O}(V) = (v_1, v_2, \ldots, v_i) \Rightarrow v_1 \preceq v_2 \preceq \ldots \preceq v_i$$

Theorem 5.4.4 (soundness and completeness)**.** *For any constructed graph $\vec{G}(V, \vec{E})$, The while-loop in Algorithm 16 (lines 8 to 13) terminates with $sound(V^*)$ and $complete(V^*)$.*

Proof. The invariant of the outer while-loop (lines 8 to 13 in Algorithm 16) is that all vertices in V^* are sound, but adding a vertex to V^* (`white` to `black`) may lead to its successors to be incomplete; for all vertices, their d^*_{in} and d^+_{out} counts are correctly maintained. All vertices in Q have their core numbers as K and their d^*_{in} must be greater or equal to 0, and all vertices $v \in V$ must be greater than 0 if v is located in Q; also, the k-order for all the vertices not in V^* is correctly maintained:

$$\begin{aligned} & sound(V^*) \ \wedge \ complete(V^* \setminus Q) \ \wedge \ in^*(V) \ \wedge \ out^+(V) \\ & \wedge \ (\forall v \in Q : v.core = K \wedge v.d^*_{in} \geq 0) \\ & \wedge \ (\forall v \in V : v.d^*_{in} > 0 \Rightarrow v \in Q) \\ & \wedge \ \mathbb{O}(V \setminus V^*) \end{aligned}$$

The invariant initially holds as $V^* = \emptyset$ and for all vertices their d^*_{in}, d^+_{out} and k-order are correctly initialized; also u is first add to Q for propagation only when $u.core = K \wedge u.d^+_{out} > K \wedge u.d^*_{in} = 0$. We now argue that the while-loop preserves this invariant:

- $sound(V^*)$ is preserved as $v \in V$ is added to V^* only if $v.d^*_{in} + v.d^+_{out} > K$ by the `Forward` procedure; also, v is safely removed from V^* if $v.d^*_{in} + v.d^+_{out} \leq K$ by the `Backward` procedure according to Theorem 5.4.3.
- $complete(V^* \setminus Q)$ is preserved as all the affected vertices v, which may have $v.d^*_{in} + v.d^+_{out} > K$, are added to Q by the `Forward` procedure for propagation.
- $in^*(V)$ is preserved as each time when a vertex v is added to V^*, all its successors' d^*_{in} are increased by 1 in the `Forward` procedure; also each time when a vertex v cannot be added to V^*, the $\mathbb{O}$ may change `Backward` procedure.
- $out^*(V)$ is preserved as each time when a vertex v cannot be added to V^*, the $\mathbb{O}$ may change and the corresponding d^+_{out} are correctly maintained by the `Backward` procedure.
- $(\forall v \in Q : v.core = K \wedge v.d^*_{in} \geq 0)$ is preserved as in the `Forward` procedure, the vertices v are added in Q only if $v.core = K$ with $v.d^*_{in}$ add by 1; but $v.d^*_{in}$ may be reduced to 0 in the `Backward` procedure when some vertices cannot in V^*.
- $(\forall v \in V : v.d^*_{in} > 0 \Rightarrow v \in Q)$ is preserved as v can be added in Q only after adding $v.d^*_{in}$ by 1 in the `Forward` procedure.
- $\mathbb{O}(V \setminus V^*)$ is preserved as the k-order of all vertices $v \in V^+ \setminus V^*$ is correctly maintained by the `Backward` procedure and the k-order of all the other vertices $v \in V \setminus V^+$ is not affected.

We also have to argue the invariant of the inner while-loop in the `Backward` procedure (lines 26 to 31 in Algorithm 18). The additional invariant is that all vertices in R has to be located in V^* but not sound as their $d^*_{in} + d^+_{out} \leq K$:

$$
\begin{aligned}
&sound(V^* \setminus R) \ \wedge \ complete(V^* \setminus Q) \ \wedge \ in^*(V) \ \wedge \ out^+(V) \\
&\wedge \ (\forall v \in R : v.core = K \wedge v \in V^* \wedge v.d^*_{in} + v.d^+_{out} \leq K) \\
&\wedge \ (\forall v \in Q : v.core = K \wedge v.d^*_{in} \geq 0) \\
&\wedge \ (\forall v \in V : v.d^*_{in} > 0 \Rightarrow v \in Q) \\
&\wedge \ \mathbb{O}(V \setminus V^*)
\end{aligned}
$$

The invariant initially holds as for w, all its predecessors' d^+_{out} are off by 1 and added in R if their $d^*_{in} + d^+_{out} \leq K$ since w is identified in $V^+ \setminus V^*$ (`gray`). We have $w \preceq$ all vertices in V^* in $\mathbb{O}$, denoted as $w \preceq V^*$, as 1) v can be moved to the head of $\mathbb{O}_{K+1}$ and $v.core$ is add by 1 if v is still in V^* when the outer while-loop terminated, and 2) v is removed from V^* and moved after w in O_K.

In this case, $w.d^+_{out}$ and $w.d^*_{in}$ are can be correctly updated to $(w.d^+_{out} + w.d^*_{in})$ and 0, respectively. We now argue that the while-loop preserves this invariant:

- $sound(V^* \setminus R)$ is preserved as all $v \in V^*$ are added to R if $v.d^*_{in} + v.d^+_{out} \leq K$.
- $in^*(V)$ is preserved as each time for a vertex $u \in R$ setting from `black` to `gray`, for all its affected successor, which have $d^*_{in} > 0$, their d^*_{in} are off by 1; also, $u.d^*_{in}$ is set to 0 when setting from `black` to `gray` since $u \preceq$ all vertices in V^* in the changed $\mathbb{O}$.
- $out^*(V)$ is preserved as each time for a vertex $u \in R$ setting from `black` to `gray`, for all its affected predecessor, which are in V^*, their d^+_{out} are off by 1; also, $u.d^+_{out}$ is set to $u.d^+_{out} + u.d^*_{in}$ since $u \preceq$ all vertices in V^* in the changed $\mathbb{O}$.
- $(\forall v \in R : v.core = K \wedge v \in V^* \wedge v.d^*_{in} + v.d^+_{out} \leq K)$ is preserved as each time for a vertex $v \in V^*$, v is checked when $v.d^*_{in}$ or $v.d^+_{out}$ is off by 1, and v is added to R if $v.d^*_{in} + v.d^+_{out} \leq K$.
- $\mathbb{O}(V \setminus V^*)$ is preserved as each vertex v that removed from V^* by pealing are moved following p in O_K, where p is w or the previous vertex removed from V^*.

At the termination of the inner while-loop, we get $R = \emptyset$. At the termination of the outer while-loop, we get $Q = \emptyset$. The postcondition of the outer while-loop is $sound(V^*) \wedge complete(V^*)$. □

At the ending phase of Algorithm 16, the core numbers of all vertices in V^* are increased by 1, and $\mathbb{O}$ in k-order is maintained. On the termination of Algorithm 16, all core numbers are correct and $\mathbb{O}(V) \wedge in^*(V) \wedge out^+(V)$ are preserved, which provides a correct initial state for the next edge insertion.

Complexity.

Theorem 5.4.5. *The time complexity of the simplified order-based insertion algorithm is $O(|E^+| \cdot \log|E^+|)$ in the worst case, where $|E^+|$ is the number of adjacent edges for all vertices in V^+ defined as $|E^+| = \sum_{v \in V^+} v.deg$.*

Proof. As the definition of V^+, it includes all traversed vertices to identify V^*. In the `Forward` procedure, the vertices in V^+ are traversed at most once, so do in the `Backward` procedure, which requires worst-case $O(|E^+|)$ time. In the while-loop (Algorithm 16 lines 5 - 10), the min-priority queue Q includes at most $|E^+|$ vertices since each related edge of vertices in V^+ is added into Q at most once. The min-priority queue can be implemented by min-heap, which

requires worst-case $O(|E^+|\cdot\log|E^+|)$ time to dequeue all the values. All the vertices in $\mathbb{O}$ are maintained with OM data structure so that manipulating the order of one vertex requires amortized $O(1)$ time; there are totally at most $|V^+|$ vertices whose order are manipulated, which requires worst-case $O(|V^+|)$ amortized time. Therefore, the total worst-case time complexity is $O(|E^+|+|E^+|\cdot\log|E^+|+|V^+|) = O(|E^+|\cdot\log|E^+|)$. □

Theorem 5.4.6. *The space complexity of the simplified order-based insertion algorithm is $O(n)$ in the worst-case.*

Proof. Each vertex v is assigned three counters that are $v.core$, $v.d^*_{in}$ and $v.d^+_{out}$, which requires $O(3n)$ space. Both Q and R have at most n vertices, respectively, which require worst-case $O(2n)$ space together. Two arrays are required for V^+ and V^*, which requires worst-case $O(2n)$ space. All vertices in $\mathbb{O}$ are maintained by OM data structure. For this, all vertices are linked by double linked lists, which requires $O(2n)$ space; also, vertices are assigned labels (typically 64 bits integer) to indicate the order, which requires $O(2n)$ space. Therefore, the total worst-case space complexity is $O(3n + 2n + 2n + 2n + 2n) = O(n)$. □

5.4.2 The Simplified Order-Based Removal

Our simplified order-based removal Algorithm is mostly the same as the original order-based removal Algorithm in (Zhang et al., 2017; Sarıyüce et al., 2016), so that the details are omitted in this section. The only difference is that our simplified order-based removal algorithm adopts the OM data structure to maintain $\mathbb{O}$, instead of the complicated $\mathcal{A}$ and $\mathcal{B}$ data structures (Zhang et al., 2017). In this case, the worst-case time complexity can be improved as the OM data structure only requires amortized $O(1)$ time for each order operation.

Complexities.

Theorem 5.4.7. *The time complexity of the simplified order-based removal algorithm is $O(Deg(G) + |E^*|)$ in the worst case, where $|E^*|= \sum_{w\in V^*} w.deg$.*

Proof. Typically, the data graph G is stored by adjacent lists. For removing an edge (u, v), all edges of the vertex u and v are sequentially traversed, which requires at most $O(Deg(G))$ time. We know that V^+ includes all traversed vertices to identify the candidate set V^* and $V^* = V^+$ in this algorithm. The vertices in V^* are traversed at most once, which requires worst-case $O(|E^*|)$ time. All vertices in V^* are removed from the $\mathbb{O}_K$ and appended to $\mathbb{O}_{K-1}$ in k-order, which requires $O(|V^*|)$ time as each insert or remove operation only requires amortized $O(1)$ time by the OM data structure. Since it is possible

that $Deg(G) > |E^*|$ in some cases like $V^* = \emptyset$, the total worst-case time complexity is $O(Deg(G) + |E^*|+|V^*|) = O(Deg(G) + |E^*|)$. □

Theorem 5.4.8. *The space complexity of the simplified order-based removal algorithm is $O(n)$ in the worst-case.*

Proof. For each vertex v in the graph, $v.mcd$ is used to identify the V^*, which requires $O(n)$ space. All vertices in $\mathbb{O}$ are maintained by OM data structure, which requires $O(4n)$ space. A queue is used for the propagation, which requires worst-case $O(n)$ space. One array is required for V^*, which requires $O(n)$ space. Therefore, the total worst-case space is $O(n + 4n + n + n) = O(n)$. □

5.5 The Simplified Order-Based Batch Insertion

In practice, it is common that a great number of edges are inserted into a graph together. If multiple edges are inserted one by one, the vertices in $V^+ \setminus V^*$ may be repeatedly traversed. Instead of inserting one by one, we can handle the edge insertion in batch. In this section, we extend our simplified order-based unit insertion algorithm to batch insertion.

Let $\Delta G = (V, \Delta E)$ be an inserted graph to a constructed DAG $\vec{G}$. That is, $\Delta E(\Delta G)$ contains a batch of edges that will be inserted to $\vec{G}$. Each edge $u \mapsto v \in \Delta E$ satisfies $u \preceq v$ in the k-order of $\vec{G}$.

Theorem 5.5.1. *After inserted a graph $\Delta G = (V, \Delta E)$ to constructed DAG $\vec{G} = (V, \vec{E})$, the core number of a vertex $v \in V(\vec{G})$ increases by at most 1 if v satisfies $|v.post| \leq v.core + 1$.*

Proof. For each $v \in V(\vec{G})$, Lemma 5.4.1 proves that the out-degree of v satisfies $|v.post| \leq v.core$. Analogies, when inserting ΔG into $\vec{G}$ with $|v.post| \leq v.core + 1$, the core number can be increased by at most 1, as after inserting the new graph has to satisfy $v.d_{out} \leq v.core$ for all vertices $v \in V(\vec{G})$. □

Theorem 5.5.1 suggests that in each round we can insert multiples edges $u \mapsto v \in \Delta E$ into $\vec{G}$ only if $|u(\vec{G}).post| \leq u(\vec{G}).core + 1$; otherwise, $u \mapsto v$ has to be inserted in next round until all edges are inserted. In the worst-case, there are $Deg(\Delta G)$ round required if each edges $u \mapsto v \in \Delta E$ satisfy $u(\vec{G}).d^+_{out} = u(\vec{G}).core$.

The Algorithm. Algorithm 19 shows the detailed steps. A batch of edges $u \mapsto v \in \Delta E$ can be inserted into $\vec{G}$ only if $u.d^+_{out} \leq u.core$ (lines 3 and 4). When $u.d^+_{out} = u.core + 1$, we can put u into the Min-Priority Queue Q for propagation (line 5). Of course, the inserted edges are removed from ΔG (line 6). After all possible edges are inserted, the propagation is the same as in lines 5 - 10 of Algorithm 16 (line 7), where K is the core numbers of local k-subcore with $K = u.core \leq v.core$ for an inserted edge $u \mapsto v$. This process repeatedly continues until the ΔG becomes empty (line 1).

Algorithm 19: BatchInsertEdge($\vec{G}, \mathbb{O}, \Delta G$)

input : A constructed DAG $\vec{G} = (V, \vec{E})$; the corresponding $\mathbb{O}$; An inserted graph $\Delta G = (V, \Delta E)$.
output: A updated DAG $\vec{G}(V, \vec{E})$; A updated $\mathbb{O}$.

```
1 while ΔG ≠ ∅ do
2     V*, V+, Q ← ∅, ∅, a min-priority queue by 𝕆
3     for u ↦ v ∈ ΔE(ΔG) : u.d+_out ≤ u.core do
4         insert u ↦ v into G⃗ with u.d+_out ← u.d+_out + 1
5         if u.d+_out = u.core + 1 then Q.enqueu(u)
6         remove u ↦ v from ΔG
7     same code as lines 5 - 10 in Algorithm 16 with K as the core number of
        local subcore
```

Example 5.5.1. Consider inserting two edges in the constructed graph in Figure 5.3. Initially, the Min-Priority Queue Q is empty and K is the core number of the corresponding k-subcore. In Figure 5.3(a), after inserting two edges, $u_1 \mapsto v_2$ and $u_2 \mapsto v_2$, we get $u_1.d^+_{out} = K + 1 = 2$ and $u_2.d^+_{out} = K + 1 = 2$, so that these two edges can be inserted in batch and we put u_1 and u_2 in Q as $Q = \{u_1, u_2\}$. We begin to propagate Q. First, in Figure 5.3(a), u_1 is removed from Q to do the `Forward` procedure since $u_1.d^*_{in} + u_1.d^+_{out} = 0 + 2 > K = 1$, by which u_1 is colored by `black`; within subcore `sc`(u_1), all $u_1.post$'s d^*_{in} are added by 1, and all $u_1.post$ are put in Q as $Q = \{u_2\}$. Second, in Figure 5.3(b), u_2 is removed from Q to do the `Forward` procedure since $u_2.d^*_{in} + u_2.d^+_{out} = 0 + 2 > K = 1$, by which u_2 is colored by `black`; within subcore `sc`(u_2), all $u_2.post$'s d^*_{in} are added by 1, and all $u_2.post$ are put in Q as $Q = \{u_3\}$. Third, in Figure 5.3(c), u_3 is removed from Q to do the `Backward` procedure since $u_3.d^*_{in} + u_3.d^+_{out} = 1 + 0 \leq K = 1$, by which u_3 is colored by `black` and $u_2.d^+_{out}$ off by 1; however, since $u_2.d^*_{in} + u_2.d^+_{out} = 1 + 1 > K = 1$, we have u_2 still `black` and the `Backward` procedure terminates. Finally, in Figure 5.3(d), two `black` vertices, u_1 and u_2, have increased core numbers as 2; then, they are removed from $\mathbb{O}_1$ and inserted before the head of $\mathbb{O}_2$ to maintain the k-order.

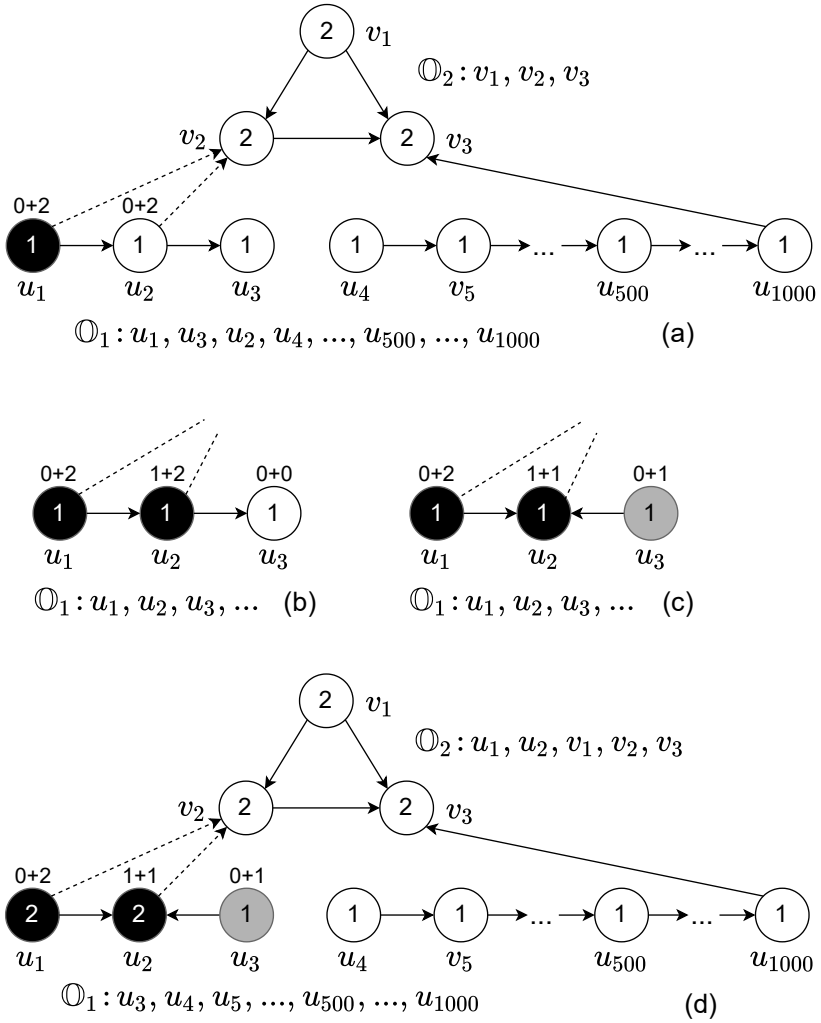


Figure 5.3: Insert a batch of two edges $u_1 \mapsto v_2$ and $u_2 \mapsto v_2$ to a constructed graph $\vec{G}$ obtained from Figure 5.1.

In this example, we have $V^* = \{u_1, u_2\} \wedge V^+ = \{u_1, u_2, u_3\}$ by batch inserting two edges together. If we insert $u_1 \mapsto v_2$ first and then insert $u_2 \mapsto v_2$ second, the final V^* is same; but V^+ is $\{u_1, u_2, u_3\}$ and $\{u_2, u_3\}$ for two inserted edges, respectively. In this case, both u_2 and u_3 are repeatedly traversed, which can be avoided by batch insertion.

Correctness. For each round of the while-loop (lines 2 - 7), the correctness argument is totally the same as the single edge insertion in Algorithm 16.

Complexities. The total worst-case running time of line 7 is the same as in Algorithm 16, which is $O(|E^+| \cdot \log|E^+|)$. Typically, for Q, the running time of enqueue and dequeue are larger than in Algorithm 16 since each time numerous vertices can be initially added into Q for propagation (line 5). The outer while-loop (line 1) runs at most ΔE rounds, so that ΔE is checked at most $O(Deg(\Delta G) \cdot |\Delta E|)$ round as $\mathbb{O}$ can be changed and thus the directions of edges in ΔE can be changed. Typically, the majority of edges can be inserted in the first round of the while-loop. Therefore, the time complexity of Algorithm 19 is $O(|E^+| \cdot \log|E^+| + Deg(\Delta G) \cdot |\Delta E|)$ in the worst case.

The space complexity of Algorithm 19 is the same as Algorithm 16.

5.6 Experiments

In this section, we conduct experimental studies using 12 real and synthetic graphs and report the performance of our algorithm by comparing it with the original order-based method:

– The order-based algorithm (Zhang et al., 2017) with unit edge insertion (`I`) and edge removal (`R`); Before running, we execute the initialization (`Init`) step.

– Our simplified order-based with unit edge insertion (`OurI`) and edge removal (`OurR`); Before running, we execute the initialization (`OurInit`) step.

– Our simplified order-based batch edge insertion (`OurBI`).

The experiments are performed on a desktop computer with an Intel CPU (4 cores, 8 hyperthreads, 8 MB of last-level cache) and 16 GB of main memory. The machine runs the Ubuntu Linux (18.04) operating system. All tested algorithms are implemented in C++ and compiled with g++ version 7.3.0 with the -O3 option. All implementations and results are available at github[1].

[1] `https://github.com/Itisben/SimplifiedCoreMaint.git`

Tested Graphs. We evaluate the performance of different methods over a variety of real-world and synthetic graphs, which are shown in Table 6.2. For simplicity, directed graphs are converted to undirected ones in our testing; all of the self-loops and repeated edges are removed. That is, a vertex cannot connect to itself, and each pair of vertices can connect with at most one edge. The *livej*, *patent*, *wiki-talk*, and *roadNet-CA* graphs are obtained from SNAP[2]. The *dbpedia*, *baidu*, *pokec* and *wiki-talk-en wiki-links-en* graphs are collected from the KONECT[3] project. The *ER*, *BA*, and *RMAT* graphs are synthetic graphs; they are generated by the SNAP[4] system using Erdös-Rényi, Barabasi-Albert, and the R-MAT graph models, respectively. For these generated graphs, the average degree is fixed to 8 by choosing 1,000,000 vertices and 8,000,000 edges.

Graph	n	m	AvgDeg	Max k
livej	4,847,571	68,993,773	14.23	372
patent	6,009,555	16,518,948	2.75	64
wikitalk	2,394,385	5,021,410	2.10	131
roadNet-CA	1,971,281	5,533,214	2.81	3
dbpedia	3,966,925	13,820,853	3.48	20
baidu	2,141,301	17,794,839	8.31	78
pokec	1,632,804	30,622,564	18.75	47
wiki-talk-en	2,987,536	24,981,163	8.36	210
wiki-links-en	5,710,993	130,160,392	22.79	821
ER	1,000,000	8,000,000	8.00	11
BA	1,000,000	8,000,000	8.00	8
RMAT	1,000,000	8,000,000	8.00	237

Table 5.1: Tested real and synthetic graphs.

In Table 6.2, we can see all graphs have millions of edges, their average degrees are ranged from 2.1 to 22.8, and their maximal core numbers range from 3 to 821. For each tested graph, the distribution of core numbers for all the vertices is shown in Figure 5.4, where the x-axis is the core numbers, and the y-axis is the size of the vertices. For most graphs, many vertices have small core numbers, and few have large core numbers. Specifically, *wiki-link-en* has the maximum core numbers up to 821, so that for most of its $\mathbb{O}_k$, the sizes are around 1000; *BA* has a single core number as 8 so that all vertices are in the single $\mathbb{O}_k$. Since all vertices with core number k in $\mathbb{O}_k$ are maintained in k-order, the size of $\mathbb{O}_k$ is related to the performance of different methods.

[2] `http://snap.stanford.edu/data/index.html`

[3] `http://konect.cc/networks/`

[4] `http://snap.stanford.edu/snappy/doc/reference/generators.html`

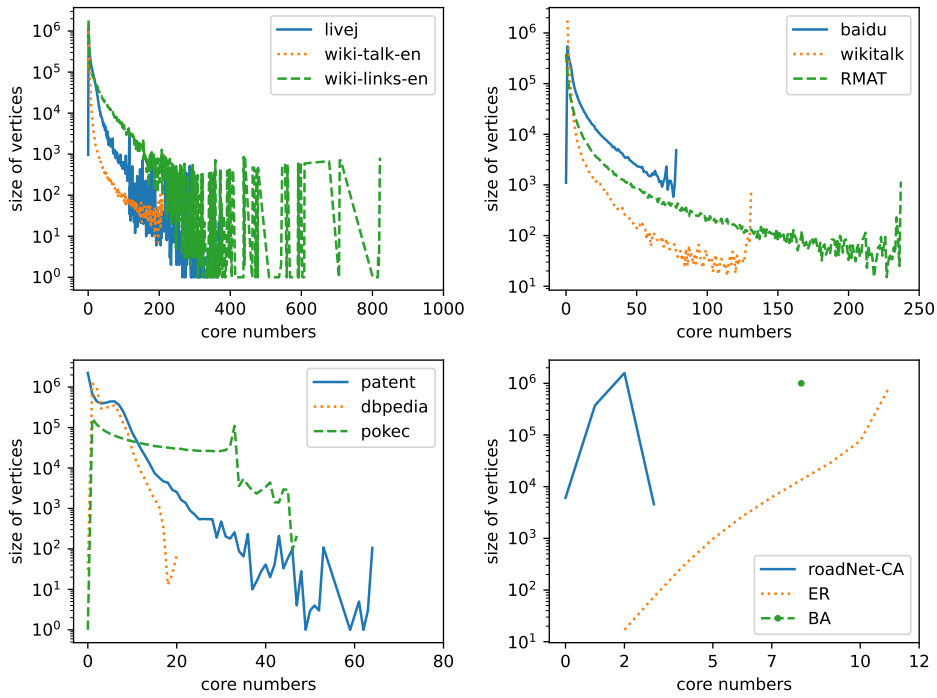


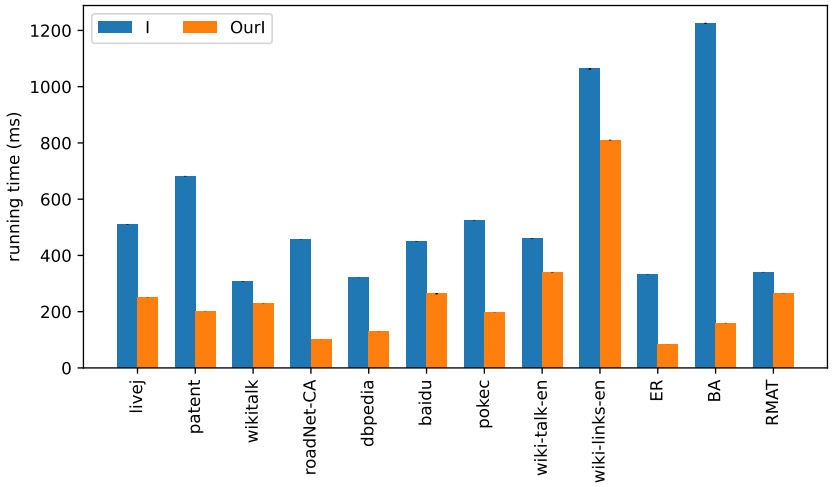
Figure 5.4: The distribution of core numbers.

5.6.1 Running Time Evaluation

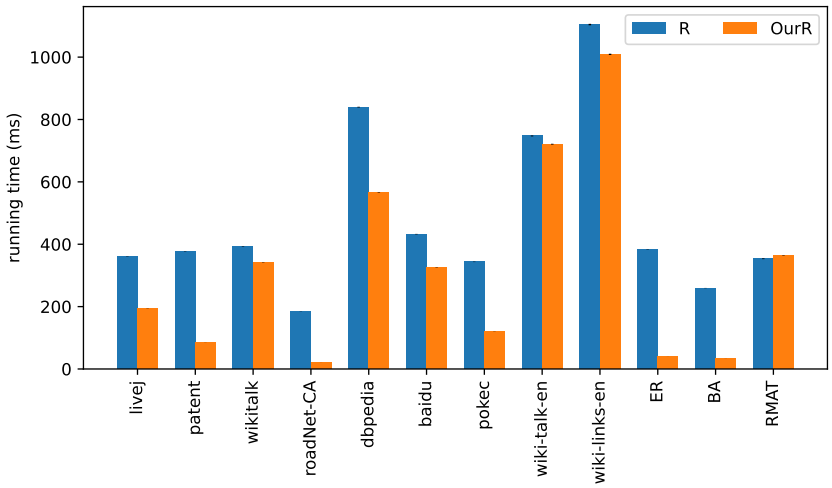
In this experiment, we compare the performance of our simplified order-based method (`OurI` and `OurR`) with the original order-based method (`I` and `R`). For each tested graph, we first randomly select 100,000 edges out of each tested graph. For each graph, we measure the accumulated time for inserting or removing these 100,000 edges. Each test runs at least 50 times, and we calculate the means with 95% confidence intervals. Note that, the error bars are too small to see in our experiments.

The results for edge insertion are shown in Figure 6.4(a). We can see `OurI` outperforms `I` over all tested graphs. Specifically, Table 5.2 shows the speedups of `OurI` vs. `I`, which ranges from 1.29 to 7.69. The reason is that the sequence $\mathbb{O}_k$ in k-order is maintained separately for each core number k. Each time insert v into or remove v from $\mathbb{O}_k$, `OurI` only requires worst-case $O(1)$ amortized time while `I` requires worst-case $O(\log|\mathbb{O}_k|)$ time. Therefore, over *BA* we can see `OurI` gains the largest speedup as 7.69 since all vertices have single one core number with $|\mathbb{O}_8| = 8,000,000$; over *wiki-links-en* we can see `OurI` gains the smallest speedup as 1.29 since vertices have core numbers ranging from 0 to 821 such that a large portion of order lists has $|\mathbb{O}_k|$ around 1000.

Similarly, we can see that the edge removal has almost the same trend of



(a) Edge Insertion



(b) Edge Removal

Figure 5.5: Compare the running times of two methods.

speedups in Figure 6.4(b), which ranges from 1.16 to 5.26 in Table 5.2. However, we observe that the speedups of removal may be less than the insertion over most graphs. The reason is that edge removal requires fewer order operations compared with edge insertion. That is, unlike the edge insertion, it is not necessary to compare the k-order for two vertices by $\texttt{Order}(\mathbb{O}, x, y)$ when reversing the vertices. The main order operations are $\texttt{Remove}(\mathbb{O}, x)$ and then $\texttt{Insert}(\mathbb{O}, x, y)$, when the core numbers of vertices $x \in V^*$ are off by 1.

In Table 5.2, for the batch insertion, we can see the speedups of `OurBI` vs. `I` are much less than the speedups of `OurI` vs. `I`, although `OurBI` may have smaller size of V^+ than `OurI`. One reason is that `OurBI` has to traverse

inserted graph ΔG at most $Deg(\Delta G)$ round, which is the maximum degree of the inserted graph ΔG. The other reason is that compared with `OurI`, `OurBI` has a larger size of priority queue Q, which `OurBI` requires more running time on enqueue and dequeue operations.

In Table 5.2, we also observe that the speedups of `OurInit` vs. `Init` is a little larger than 1. The reason is that for initialization, most of the running time is spent on computing the core number for all vertices by the BZ algorithm. After running the BZ algorithm, `OurInit` assigns labels for all vertices to construct $\mathbb{O}$ in k-order, which requires worst-case $O(n)$ time. However, `Init` has to add all vertices to binary search trees, which requires worst-case $O(n \log n)$ time.

Table 5.2: Compare the speedups of our method for all graphs.

Graph	`OurI` vs `I`	`OurBI` vs `I`	`OurR` vs `R`	`OurInit` vs `Init`
livej	2.04	1.66	1.87	1.02
patent	3.37	2.68	4.41	1.04
wikitalk	1.34	1.63	1.15	1.26
roadNet-CA	4.51	2.95	8.56	1.17
dbpedia	2.49	2.14	1.49	1.08
baidu	1.70	1.68	1.33	1.04
pokec	2.67	2.37	2.87	1.03
wiki-talk-en	1.36	1.45	1.04	1.20
wiki-links-en	1.31	1.16	1.09	1.02
ER	3.97	2.76	9.72	1.08
BA	7.69	5.26	7.42	1.15
RMAT	1.29	1.31	0.97	1.09

5.6.2 Stability Evaluation

We test the stability of different methods over two selected graphs, i.e., *wikitalk* and *dbpedia* as follows. First, we randomly sample $5,000,000$ edges and partition them into 50 groups, where each group has totally different $100,000$ edges. Second, for each group, we measure the accumulated running time of different methods. That is, the experiments run 50 times, and each time has totally different inserted or removed edges.

Figure 6.6 shows the results over two selected graphs. We can see that `OurI` and `OurR` outperform `I` and `R`, respectively. More important, the performance of `OurI` and `OurR` is as well-bounded as `I` and `R`, respectively. The reason is that `I` and `R` are well-bounded as the variation of V^+ is small for different

inserting or removal edges; also, `OurI` and `OurR` have the same size of traversed vertices V^+ and thus have similar well-bounded performance.

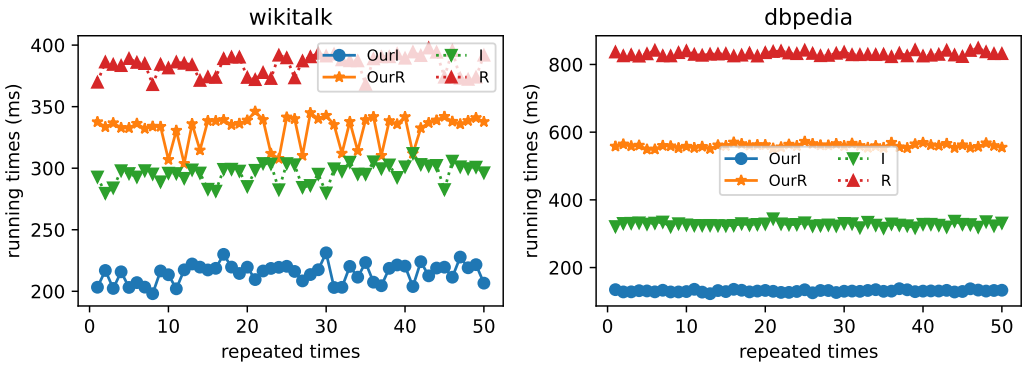


Figure 5.6: The stability of all methods over selected graphs.

5.6.3 Scalability Evaluation

We test the scalability of different methods over two selected graphs, i.e., *wikitalk* and *dbpedia*. We vary the number of edges exponentially by randomly sampling from 100,000 to 200,000, 400,000, 800,000, 1,600,000, etc. We keep incident vertices of edges for each sampling to generate the induced subgraphs. Over each subgraph, we further randomly selected 100,000 edges for insertion or removal. For example, over *wikitalk*, the first subgraph has 100,000 edges, all of which can be inserted or removed; the last subgraph has 3,200,000 edges, only 100,000 of which can be inserted or removed. Over each subgraph, we measure the accumulated time for the insertion or removal of these 100,000 edges. Each test runs at least 50 times, and we calculate the average running time.

We show the result in Figure 5.7, where the x-axis is the number of sampled edges in subgraphs increasing exponentially, and the y-axis is the running times (ms) for different methods by inserting or removing 100,000 edges. Table 5.3 shows the details of scalability evaluation, where m' is the number of edges in subgraphs, $\#lb$ is the number of updated labels used by the OM data structures of our methods, and $\#rp$ is the number of outer while-loop repeated rounds for `OurBI`. We make several observations as follows:

- In Figure 5.7, a first look reveals that the running time of `OurI` grow more slowly compared with `I`. The reason is that `OurI` improves the worst-case running time of each order operation of $\mathbb{O}_k$ from $O(\log|\mathbb{O}_k|)$ to $O(1)$. In this case, the larger sampled graphs have a larger size of $\mathbb{O}_k$, which can lead to higher speedups. However, we can see that the running time of `OurR` always grows with a similar trend compared with `R`. The reason is that a large

percentage of the running time is spent on removing edges from the adjacent lists of vertices, which requires traversing all the corresponding edges. Because of this, even `OurR` has more efficient order operations for $\mathbb{O}$ than `R`, the speedups are not obvious.

– In Figure 5.7, we observe that `OurBI` sometimes runs faster than `OurI`. The reason is as follows. From Table 5.3, compared with `OurI`, `OurBI` has less traversed vertices (V^+), as some repeated traversed vertices can be avoided; also, `OurBI` has less number of updated labels ($\#lb$), as the number of relabel process can be reduced. However, compared with `OurI`, `OurBI` may add much more vertices into priority queue Q, which costs more the running time of enqueue and dequeue. Also, `OurBI` may require several times of repeated rounds ($\#rp$), which may cost extra running time. Typically, this extra running time is acceptable as most of the edges can be inserted in the first round, e.g., for *dbpedia* with 6.4M sampled edges, the number of batch inserted edges is 100000, 1555, 12, and 0 in four rounds, respectively. This is why `OurBI` sometimes runs faster but sometimes slower compared with `OurI`.

– In Figure 5.7, we observe that `OurR` is always faster than `OurI`. The reason is as follows. From Table 5.3, compared with `OurI`, `OurR` has less number of traversed vertices (V^*), as `OurR` has $V^* = V^+$; `OurR` has less number of updated labels ($\#lb$), as vertices are removed from O_K and then appended after O_{K-1} and thus the relabel process is not always triggered.

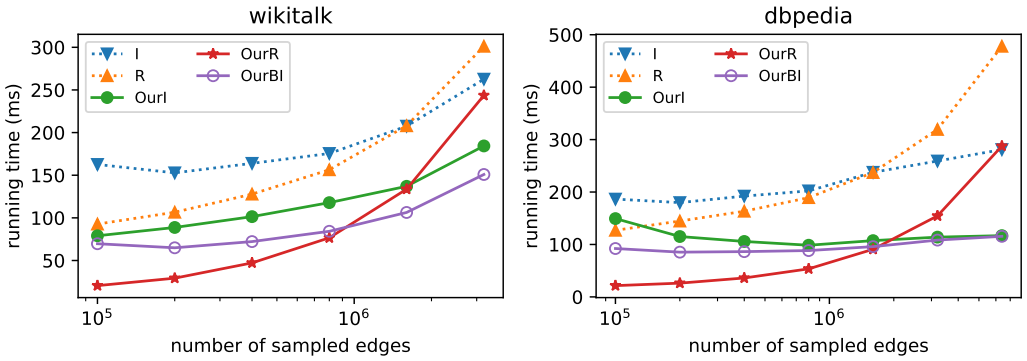


Figure 5.7: The scalability of all methods over selected graphs.

	OurI			OurBI				OurR	
m'	$\|V^*\|$	$\|V^+\|$	$\#lb$	$\|V^*\|$	$\|V^+\|$	$\#lb$	$\#rp$	$\|V^*\|$	$\#lb$
0.1M	107K	131K	1.5M	107K	123K	116K	10	107K	107K
0.2M	101K	118K	1.4M	101K	109K	111K	11	101K	101K
0.4M	101K	116K	1.3M	101K	107K	107K	9	101K	101K
0.8M	101K	113K	1.3M	101K	106K	104K	10	101K	101K
1.6M	100K	110K	1.2M	100K	106K	103K	11	100K	100K
3.2M	101K	110K	1.1M	101K	106K	103K	9	101K	101K
0.1M	146K	149K	2.6M	146K	151K	149K	4	146K	146K
0.2M	129K	136K	2.1M	129K	136K	132K	3	129K	129K
0.4M	117K	131K	1.8M	117K	130K	124K	3	117K	117K
0.8M	109K	127K	1.6M	109K	126K	118K	3	109K	109K
1.6M	105K	127K	1.4M	105K	125K	116K	4	105K	105K
3.2M	102K	124K	1.3M	102K	122K	113K	4	102K	102K
6.4M	100K	124K	1.2M	100K	122K	112K	4	100K	100K

Table 5.3: The details of scalability evaluation by varying the number of sampled edges over *wikitalk* and *dbpedia*.

Chapter 6

Parallel Core Maintenance

The core numbers of vertices in a graph are one of the most well-studied cohesive subgraph models because of the linear running time. In practice, many data graphs are dynamic graphs that are continuously changing by inserting or removing edges. The core numbers are updated in dynamic graphs with edge insertions and deletions, which is called core maintenance. When a burst of a large number of inserted or removed edges come in, we have to handle these edges on time to keep up with the data stream. There are two main sequential algorithms for core maintenance, Traversal and Order. The experiments show that the Order algorithm significantly outperforms the Traversal algorithm over a variety of real graphs.

To the best of our knowledge, all existing parallel approaches are based on the Traversal algorithm. These algorithms exploit parallelism only for vertices with different core numbers; they reduce to sequential algorithms when all vertices have the same core numbers. In this chapter, we propose a new parallel core maintenance algorithm based on the Order algorithm. More importantly, our new approach always has parallelism, even for graphs where all vertices have the same core numbers. Extensive experiments are conducted over real-world, temporal, and synthetic graphs on a multicore machine. The results show that for inserting and removing a batch of edges using 16 workers, our method achieves up to 289x and 10x times speedups compared with the most efficient existing method, respectively.

6.1 Introduction

Many sequential algorithms are devised on core maintenance in dynamic graphs (Guo and Sekerinski, 2022c; Zhang et al., 2017; Sarıyüce et al., 2016; Wu et al., 2015; Saríyüce et al., 2013; Li et al., 2013). The main idea for core maintenance is that we first need to identify a set of vertices whose core numbers need to be updated (denoted as V^*) by traversing a possibly larger cope

of vertices (denoted as V^+). There are two main algorithms, Order (Zhang et al., 2017) and Traversal (Sarıyüce et al., 2016). Given an inserted edge, the Order algorithm has to traverse much fewer vertices than the Traversal algorithm by maintaining the order for all vertices. That is why the Order algorithm has significantly improved running time. In (Guo and Sekerinski, 2022c), a Simplified-Order algorithm is proposed for easy understanding and implementation based on the Order algorithm.

All the above methods are sequential for maintaining core numbers over dynamic graphs, which means each time only one insert or removal edge is handled. The problem is that when a burst of a large number of inserted or removed edges come in, these edges may not be handled on time to keep up with the data stream (Gabert et al., 2021). The prevalence of multi-core machines suggests parallelizing the core maintenance algorithms. Many multi-core parallel batch algorithms for core maintenance have been proposed in (Hua et al., 2019; Jin et al., 2018; Wang et al., 2017). All above methods have similar ideas: 1) they use an available structure, e.g. *Join Edge Set* (Hua et al., 2019) or *Matching Edge Set* (Jin et al., 2018), to preprocess a batch of inserted or removed edges avoiding repeated computations, and 2) each worker performs the Traversal algorithm. There are two drawbacks to these approaches. First, they are based on the sequential Traversal algorithm (Saríyüce et al., 2013; Li et al., 2013), which is much less efficient than the Order algorithm (Hua et al., 2019; Guo and Sekerinski, 2022c). Second, they exploit the parallelism only for vertices with different core numbers, that is, they reduce to sequential algorithms when all affected vertices have the same core numbers.

To overcome the above drawbacks, inspired by the Simplified-Order algorithm (Guo and Sekerinski, 2022c), we propose a new parallel algorithm to maintain core numbers for dynamic graphs, so-called the Parallel-Order algorithm. That is, each worker handles one inserted or removed edge at a time and propagates the affected vertices in order, and we lock vertices for synchronization. The parallel order maintenance data structure (Guo and Sekerinski, 2022b) is adopted to maintain the order for all vertices. For edge insertion and removal, our parallel approach has the same work as the sequential Simplified-Order algorithm (Guo and Sekerinski, 2022c). The main contributions of our work are summarized below:

- For edge insertion and removal, we design novel mechanisms for synchronization by only locking vertices in V^+ instead of locking all accessed edges. In other words, all the neighbors of vertices in V^+ are not necessarily locked. This is meaningful considering real graphs always have a much larger number of edges than vertices, let alone dense graphs. Additionally, for each inserted or removed edge, the size of V^+ is typically less than 10. Thus, it has a low probability that multiple workers block as a chain and then reduce to sequential execution. Fewer locked vertices

will lead to higher parallelism.

- For edge insertion, we lock all vertices in order to avoid deadlocks. For edge removal, we design a conditional lock mechanism to avoid deadlocks. We prove that deadlocks will never happen.

- We conduct extensive experiments on a multicore machine over various graphs. Our method achieves significant up to 5x times speedups by using 16 workers for edge insertion and removal. Compared with the existing most efficient parallel approach in (Hua et al., 2019), our method achieves up to 289x and 10x times speedups for edge insertion and removal, respectively.

	Worst-case (O)		Best-case (O)	
Parallel	$\mathcal{W}$	$\mathcal{D}$	$\mathcal{W}$	$\mathcal{D}$
Insert	$m'\|E^+\|\log\|E^+\|$	$m'\|E^+\|\log\|E^+\|$	$m'\|E^+\|\log\|E^+\|$	$\|E^+\|\log\|E^+\|+m'\|V^*\|$
Remove	$m'\|E^*\|$	$m'\|E^*\|$	$m'\|E^*\|$	$\|E^*\|+m'\|V^*\|$

Table 6.1: The worst-case and best-case work, depth complexities of our parallel core maintenance operations for inserting and removing a batch of edges, where m' is the total number of edges that are inserted or removed in parallel, E^+ is adjacent edges for all vertices in V^+, and E^* is adjacent edges for all vertices in V^*.

Table 6.1 shows the work and depth of our Parallel-Order algorithm for inserting or removing m' edges in parallel. For both edge insertion and removal, one big issue is the depth $\mathcal{D}$ equal to the work $\mathcal{W}$ in the worst case; that is, all workers execute as one blocking chain such that only one worker is active. However, with a high probability, such a worst-case does not happen as the number of locked vertices is always small for each insertion or removal. For our method, all vertices in V^+ or V^* are locked together for each insertion or removal.

6.2 Related Work

In (Saríyüce et al., 2013; Li et al., 2013), an algorithm that is similar to the Traversal algorithm is given, but this solution has quadratic time complexity. In (Sun et al., 2020), Sun et al. design algorithms to maintain approximate cores in dynamic *hypergraphs* in which a *hyteredge* may contain one or more participating vertices compared with exactly two in graphs. In (Gabert et al., 2021), Gabert et al. propose parallel core maintenance algorithms for maintaining cores over hypergraphs. There exists some research based on core

maintenance. In (Yu et al., 2021), the authors study computing all k-cores in the graph snapshot over the time window. In (Lin et al., 2021), the authors explore the hierarchy core maintenance. In (Weng et al., 2021), the distributed approaches to core maintenance are explored.

All the above work focus on unweighted graphs, but graphs are weighted in a lot of realistic applications. For an edge-weighted graph, the degree of a vertex is the sum of the weights of all its incident edges. But it has a large search range to maintain the core numbers after the change by using the traditional core maintenance algorithms directly, as the degree of a related vertex may change widely. In (Zhou et al., 2021), Zhou et al. extend the coreness to weighted graphs and devise weighted core decomposition algorithms; also they devise weighted core maintenance based on the k-order (Zhang et al., 2017; Guo and Sekerinski, 2022c). In (Liu and Zhang, 2020), Liu et al. improve the core decomposition and incremental maintenance algorithm to suit edge-weighted graphs.

6.3 Parallel Core Maintenance

The existing parallel core maintenance algorithms are based on the sequential Traversal algorithm which is experimentally shown much less efficient than the sequential Order algorithm. In this section, based on the Order algorithm, we propose a new parallel core maintenance algorithm, so-called Parallel-Order, for both edge insertion and removal.

The main steps for parallel edges in parallel are shown in Algorithm 20. Given an undirected graph G, the core number and k-order can be initialized by the BZ algorithm (Batagelj and Zaversnik, 2003) in linear time. A batch ΔE of edges will insert into G. We split these edges ΔE into $\mathcal{P}$ parts, $\Delta E_1 \dots \Delta E_{\mathcal{P}}$, where $\mathcal{P}$ is the total number of workers (line 1). Each worker p inserts multiple inserted edges of ΔE_p in parallel with other workers (line 2). One by one, a worker p deals with a single edge in $\texttt{InsertEdge}_p$ (line 4). The key issue is how to implement $\texttt{InsertEdge}_p$ executed by a worker p in parallel with other workers.

Algorithm 20: Parallel-InsertEdges$(G, \mathbb{O}, \Delta E)$

```
1 partition ΔE into ΔE_1, ..., ΔE_P
2 DoInsert_1(ΔE_1) ‖ ... ‖ DoInsert_P(ΔE)
3 procedure DoInsert_p(ΔE_p)
4 |   for (u, v) ∈ ΔE_p do InsertEdge_p(G, 𝕆, (u, v))
```

For removing edges in parallel, it is analogous to Algorithm 20, and the key issue is $\texttt{RemoveEdge}_p$. Note that, the insertion and removal can not run

in parallel simultaneously, which can greatly simplify the synchronization of algorithms.

One benefit of our method is that, unlike the existing parallel core maintenance methods (Hua et al., 2019; Jin et al., 2018; Wang et al., 2017), the prepossessing of ΔE_p is not required so that edges can be inserted or removed on-the-fly.

6.3.1 Parallel Edge Insertion

Algorithm. The detailed steps of $\texttt{InsertEdge}_p$ are shown in Algorithm 22, which is analogous to Algorithm 16. We introduce several new data structures. First, the min-priority queue Q_p, the queue R_p, the candidate set V_p^*, and the searching set V_p^+ are all private to each worker p, so cannot be accessed by other workers and synchronization is not necessary (lines 3, 7). Second, for each vertex $u \in V$, we introduce a status $u.s$, initialized as 0, and atomically incremented by 1 before and after the k-order operation (lines 16 and 30). In other words, when $u.s$ is an odd number, the k-order of u is being maintained. By using such a status of each vertex, we obtain $v \in u.post$ ($u \preceq v$) or $v \in u.pre$ ($v \preceq u$) by the parallel $\texttt{Order}(u, v)$ operation. As shown in Algorithm 21, when comparing the order of u and v, we ensure that u and v are not updating their k-order.

Algorithm 21: Parallel-Order($\mathbb{O}, u, v$)

```
1 do
2 |   do s ← u.s; s' ← v.s while s mod 2 = 0 ∨ s' mod 2 = 0
3 |   r ← u ⪯ v
4 while s = u.s ∧ s' = v.s
5 return r
```

Given an inserted edge $u \mapsto v$ where $u \preceq v$, we lock both u and v together when both are not locked (line 1). We redo the lock of u and v if they are updated by other workers as $v \preceq u$ (line 2). After locking, K is initialized to the smaller core number of u and v. After inserting the edge $u \mapsto v$ into the graph G (line 4), v can be unlocked (line 5). If $u.d_{out}^+ \leq K$, we unlock u and terminate (line 6); otherwise, we set w as u for propagation (line 7). In the do-while-loop (lines 8 - 13), initially, w equals u, which was already locked in line 1 (line 7). We calculate $w.d_{in}^*$ by counting the number of $w.pre$ located in V_p^* (line 9). If $w.d_{in}^* + w.d_{out}^+ > K$, vertex w does $\texttt{Forward}_p$ (line 10). If $w.d_{in}^* + w.d_{out}^+ \leq K \wedge w.d_{in}^* > 0$, vertex w does $\texttt{Backward}_p$ (line 11). If $w.d_{in}^* + w.d_{out}^+ \leq K \wedge w.d_{in}^* = 0$, we skip w and unlock w since w cannot be in V^+ (line 11). Successively, we dequeue a vertex w from Q_p with $w.core = K$ and lock w at the same time (line 12). The do-while-loop terminates when no

more vertices can be dequeued from Q_p (line 13). All vertices $w \in V_p^*$ have their core numbers increased by 1 and their $w.d_{in}^*$ is reset to 0 (line 15); also, all w are removed from $\mathbb{O}_K$ and inserted at the head of $\mathbb{O}_{K+1}$ to maintain the k-order by using the parallel OM data structure, where all $w.s$ are atomically increased by 1 before and after this process (line 16). Before termination, we unlock all locked vertices w (line 17).

The `Forward`(u) and `Backward`(w) procedures are almost the same as their sequential version since all vertices in V^+ are locked. There are only several differences. In `Forward`$_p(u)$, for each v in $u.post$ whose core numbers equal to K, we add v into the priority queue Q_p (line 21); but $v.d_{in}^*$ is not maintained by adding 1 since it will be calculated in line 9 when it is used. In the `Backward`$_p(w)$ procedure, w is removed from $\mathbb{O}_K$ and appended after pre to maintain the k-order by using the parallel OM data structure, where $w.s$ are atomically increased by 1 before and after this process (line 30).

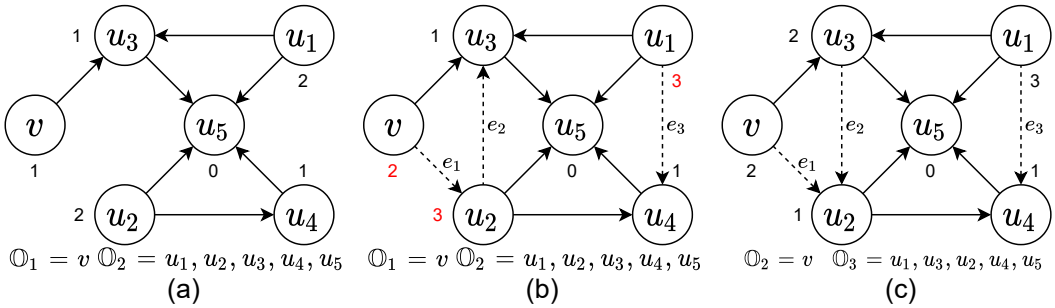


Figure 6.1: An example graph maintains the core numbers after inserting three edges, e_1, e_2, and e_3. The letters inside the circles are vertices' IDs and the $\mathbb{O}_k$ is the k-order of vertices with core numbers k. The beside numbers are corresponding remaining out-degrees d_{out}^+. The direction for each edge indicates the k-order of two vertices, which is constructed as a DAG. (a) an initial example graph. (b) insert 3 edges. (c) the core numbers and k-orders update.

Example 6.3.1. *Sequential.* In Figure 6.1, we show an example graph that maintains the core numbers of vertices after inserting edges, e_1 to e_3, successively. Figure 6.1(a) shows an example graph constructed as a DAG where the direction of edges indicates the k-order. After initialization, v has a core number 1 with k-order $\mathbb{O}_1$ and u_1 to u_5 have a core number 2 with k-order $\mathbb{O}_2$.

Figure 6.1(b) shows edges, e_1 to e_3, are inserted. (1) For e_1, we increase $v.d_{out}^+$ to 2 so that $v.d_{out}^+ > v.core$ and $V^* = \{v\}$. Then, we stop since all $v.post$ have core numbers larger than $v.core$. Finally, we increase $v.core$ from 1 to 2. (2) For e_2, we increase $v.d_{out}^+$ to 3 so that $v.d_{out}^+ > v.core$ and $V^* = \{u_2\}$. Then, we traverse u_3 in k-order and find that $u_3.d_{in}^* + u_3.d_{out}^+ = 1+1 = 2 \leq K = 2$, so

Algorithm 22: InsertEdge$_p(\vec{G}, \mathbb{O}, u \mapsto v)$

```
1  Lock u and v together if both are not locked
2  if v ⪯ u then Unlock u and v; goto line 1
3  V_p^*, V_p^+, K, ← ∅, ∅, min(u.core, v.core)
4  insert u ↦ v into G⃗ with u.d_out^+ ← u.d_out^+ + 1
5  Unlock(v)
6  if u.d_out^+ ≤ K then Unlock(u); return
7  Q_p, w ← a min-priority queue by 𝕆, u
8  do
9      w.d_in^* ← |{w' ∈ w.pre : w' ∈ V_p^*}|                    // calculate d_in^*
10     if w.d_in^* + w.d_out^+ > K then Forward_p(w)
11     else if w.d_in^* > 0 then Backward_p(w) else Unlock(w)
12     w ← Q_p.dequeue() with w.core = K and Lock(w)
13 while w ≠ ∅
14 for w ∈ V_p^* do
15     w.core ← K + 1; w.d_in^* ← 0
       // atomically add w.s
16     ⟨w.s++⟩; Delete(𝕆_K, w); Insert(𝕆_{K+1}, head, w); ⟨w.s++⟩
17 Unlock all locked vertices

18 procedure Forward_p(u)
19     V_p^* ← V_p^* ∪ {u}; V_p^+ ← V_p^+ ∪ {u}                   // u is locked
20     for v ∈ u.post : v.core = K do
21         if v ∉ Q_p then Q_p.enqueue(v)

22 procedure Backward_p(w)
23     V_p^+ ← V_p^+ ∪ {w}; pre ← w                               // w is locked
24     R_p ← an empty queue; DoPre_p(w, R_p)
25     w.d_out^+ ← w.d_out^+ + w.d_in^*; w.d_in^* ← 0
26     while R_p ≠ ∅ do
27         u ← R_p.dequeue()
28         V_p^* ← V_p^* \ {u}
29         DoPre_p(u, R_p); DoPost_p(u, R_p)
           // atomically add w.s
30         ⟨w.s++⟩; Delete (𝕆_K, u); Insert(𝕆_K, pre, u); ⟨w.s++⟩
31         pre ← u; u.d_out^+ ← u.d_out^+ + u.d_in^*; u.d_in^* ← 0

32 procedure DoPre_p(u, R_p)
33     for v ∈ u.pre : v ∈ V_p^* do
34         v.d_out^+ ← v.d_out^+ − 1
35         if v.d_in^* + v.d_out^+ ≤ K ∧ v ∉ R_p then R_p.enqueue(v)

36 procedure DoPost_p(u, R_p)
37     for v ∈ u.post do
38         if v ∈ V_p^* ∧ v.d_in^* > 0 then
39             v.d_in^* ← v.d_in^* − 1
40             if v.d_in^* + v.d_out^+ ≤ K ∧ v ∉ R_p then R_p.enqueue(v)
```

that u_3 cannot add to V^* which cause u_2 to be removed from V^* (by `Backward`). In this case, u_2 is moved after u_1 in the k-order as $\mathbb{O}_2 = u_1, u_3, u_2, u_4, u_5$. (3) For e_3, we increase $u_1.d^+_{out}$ to 3 so that $u_1.d^+_{out} > K = 2$ and $V^* = \{u_2\}$. Then, we traverse u_3, u_2, u_4 and u_5 in k-order, all of which are added into V^* since their $d^*_{in} + d^+_{out} > K = 2$. Finally, we increase the core numbers of u_2 to u_5 from 2 to 3.

Figure 6.1(c) shows the result after inserting edges. We can see all vertices have their core numbers increased by 1. Orders $\mathbb{O}_2$ and $\mathbb{O}_3$ are updated accordingly. All vertices' d^+_{out} are updated accordingly.

Parallel. Continuing with Figure 6.1, we show an example of maintaining the core numbers of vertices in parallel after inserting three edges. Figure 6.1(b) shows three edges, e_1, e_2 and e_3, being inserted in parallel by three workers, p_1, p_2, and p_3, respectively. (1) For e_1, the worker p_1 will first locks v and u_2 for inserting the edge. But if u_2 is already locked by p_2, worker p_1 has to wait for p_2 to finish and unlock u_2. (2) For e_2, worker p_2 first locks u_2 and u_3 for inserting the edge, after which u_3 is unlocked. Then, u_3, u_4, and u_5 are added to its priority queue Q_2 for propagation. That is, u_3 is locked and dequeued from Q_2 with $u_3.d^*_{in} = 1$ (assuming that p_2 locks u_3 before P_3 lock u_3). After propagation, we get that V^* is empty. Subsequently, u_4 and u_5 are locked and dequeued from Q_2, which are unlocked and skipped since their $d^*_{in} = 0$. The k-order $\mathbb{O}_2$ is updated to u_1, u_3, u_2, u_4, and u_5. (3) For e_3, the worker p_3 will first lock u_1 and u_4 for inserting the edge, after which u_4 is unlocked. Then, u_3, u_4 and u_5 are added to Q_3 for propagation. That is, u_3 is locked and dequeued from Q_2 (assuming that p_3 waits for u_3 to be unlocked by p_2) with $u_3.d^*_{in} = 1$, by which u_3 is added to V^* and u_2 is added to Q_3 for propagation. Subsequently, u_3, u_2, u_4, and u_5 are locked and dequeued from Q_3 for propagation, which are all added to V^* (assuming that p_3 waits for u_2 to be unlocked by p_2).

We can see three vertices, u_3, u_4 and u_5, can be added in Q_2 and Q_3 at the same time. That is, when p_3 removes u_3 from Q_2, it is possible that u_3 has already been accessed by p_2. In this case, we have to update Q_3 before dequeuing if we find that u_3 is accessed by p_2, in case the k-order of u_3 in Q_3 is changed by p_2.

Correctness. We only argue the correctness of Algorithm 16 related to the concurrent part. There are no deadlocks since both u and v are locked together for an inserted edge $u \mapsto v$ (line 1) and the propagated vertices are locked in k-order (line 12).

For each worker p, the accessed vertices are synchronized by locking. The key issue is to ensure that a vertex w is locked and then dequeued from Q_p in k-order in the do-while-loop (lines 8 - 12). The invariant is that w has a

minimal k-order in Q_p:

$$\forall v \in Q_p : w \notin Q_p \wedge w \preceq v$$

Initially, the invariant is preserved as $w = u$ and $Q_p = \emptyset$. When dequeuing w from Q, the worker p will first lock w that has the minimum k-order and then remove w. In this case, other vertices $v \in Q$ can be accessed by other workers q. For this, there are two cases. 1) Other vertices v may have increased core numbers, which will be removed from Q and skipped. 2) Other vertices v may have $v.d^*_{in} + v.d^+_{out} \leq K$ and cannot be added to V^*_q, which may cause other vertices v' to be removed from V^*_q by the `Backward` procedure; also, all v' are moved after v in k-order, and all v' cannot possibly be moved before v. In a word, all vertices in Q_p cannot have a smaller k-order than w when w is locked.

The worker p traverses $u.post$ in the for-loop (lines 20 - 21, 37 - 40), where u is locked by p; but, all $u.post$ are not locked by p and may be locked by other workers for updating, so do $u.pre$ in the for-loop (lines 33 - 35). The invariant is that all $u.post$ have k-order greater than u and all $u.pre$ have k-order less than u:

$$(v \in u.post \implies u \preceq v) \wedge (v' \in u.pre \implies v' \preceq u)$$

- $v \in u.post \implies u \preceq v$ is preserved as all vertices $u.post$ may have increased core numbers, but v will never be moved before u in k-order by other workers q by the `Backward` procedure, which has been proved before.
- $v' \in u.pre \implies v' \preceq u$ is preserved as u is already locked by worker p so that no other workers can access u and move v' after u in k-order by the `Backward` procedure.

In other words, the sets $u.post$ and $u.pre$ will not change until u is unlocked, even when other workers access the vertices in $u.post$ and $u.pre$.

Time Complexity. When m' edges are inserted into the graphs, the total work is the same as the sequential version in Algorithm 16, which is $O(m'|E^+|\log|E^+|)$, where E^+ is the largest number of adjacent edges for all vertices in V^+ among each inserted edge, defined as $E^+ = \sum_{v \in V^+_p} v.deg$. In the best case, m' edges can be inserted in parallel by $\mathcal{P}$ workers with a depth $O(|E^+|\log|E^+| + m'|V^*|)$ as each worker will not be blocked by other workers; but, all vertices in $|V^*|$ are removed from $\mathbb{O}_K$ and inserted sequentially at the head of O_{K+1}. Therefore, The best-case running time is $O(m'|E^+|\log|E^+|/\mathcal{P} + |E^+|\log|E^+| + m'|V^*|)$. In the worst case, m' edges have to be inserted one by one, which is the same as the sequential execution, since $\mathcal{P}$ workers make a blocking chain. Therefore, the worst-case running time is $O(m'|E^+|\log|E^+|)$.

However, in practice, such a worst-case is unlikely to happen. The reason

is that, given a large number of inserted edges, they have a low probability of connecting with the same vertex; also each inserted edge has a small size of V^+ (e.g. 0 or 1) with a high probability.

Space Complexity. For each vertex $v \in V$, it takes $O(3)$ space to store $v.d^*_{in}$, $v.d^+_{out}$, $v.s$, and locks, which makes $O(3n)$ space in total. Each worker p maintains their private V^*_p, V^+_p, which takes $O(2|V^+|\mathcal{P})$ space in total. Similarly, each worker p maintains Q_p and R_p, which take $O(|E^+|\mathcal{P})$ space in total since at most $O(2|E^+|)$ vertices can be added to Q_p and R_p for each inserted edge. The OM data structure is used to maintain the k-order for all vertices in the graph, which takes $O(n)$ space. Therefore, the total space complexity is $O(n + |V^+|+|E^+|\mathcal{P}) = O(n + |E^+|\mathcal{P})$.

6.3.2 Parallel Edge Removal

Algorithm. The detailed steps of $\texttt{RemoveEdge}_p$ are shown in Algorithm 23. We introduce several new data structures. First, the queue R_p is privately used by worker p and cannot be accessed by other workers without synchronization (line 2). Second, each worker p adopts a set A_p to record all the visited vertices $w' \in w.adj$ to avoid repeatedly revisiting $w' \in w.adj$ again. Third, each vertex $v \in V$ has a status $v.t$ with four possible values:

- $v.t = 2$ means v is ready to be propagated (line 22).
- $v.t = 1$ means v is been propagated by the inner for-loop (lines 11 - 14).
- $v.t = 3$ means v has to be propagated again by the inner for-loop (lines 11 - 14), as some vertices $v.adj$ have core numbers decreased by other workers.
- $v.t = 0$ means v is just initialized or already propagated (line 33).

Given a removed edge (u, v), we lock both u and v together when both are not locked (line 1). After locking, K is initialized as the smaller core number of u and v (line 2). We execute the procedure $\texttt{CheckMCD}_p$ for u or v to make $u.mcd$ and $v.mcd$ non-empty (line 3). We remove the edge (u, v) safely from the graph G (line 4). For u or v, if their core number is greater or equal to K, we execute the procedure $\texttt{DoMCD}_p$ (lines 5 and 6), by which u and v may be added in R_p for propagation. If u or v is not in R_p, we immediately unlock u or v (line 7). The while-loop (lines 8 - 16) propagates all vertices in R_p. A vertex w is removed from R_p and an empty set A_p is initialized (line 9). In the inner for-loop (lines 11 - 14), the adjacent vertices $w' \in w.adj$ are condition-locked with $w'.core = K$ (lines 11 and 12), as $w'.core$ can be decreased from K to $K - 1$ by other workers. For each locked $w' \in w.adj$, we first execute the $\texttt{CheckMCD}_p$ procedure in case $w'.mcd$ is empty and then execute the $\texttt{DoMCD}_p$ procedure (line 13). The visited w' are added into A_p to avoid visiting them repeatedly (line 14). We atomically decrease $w.t$ by 1 before and after such an inner for-loop since other workers can access $w.t$ in line 32 (lines 10 and 15).

After that, if $w.t > 0$, we have to propagate w again as other vertices in $w.adj$ have core numbers decreased from $K+1$ to K by other workers (line 16). The while-loop will not terminate until R_p becomes empty (line 8). Finally, all vertices in V^* are appended to $\mathbb{O}_{K-1}$ to maintain the k-order (line 17). We must not forget to unlock all locked vertices before termination (line 18).

In procedure $\texttt{DoMCD}_p(u)$, vertex u has already been locked by worker p (line 19). We decrease $u.mcd$ by 1 as $u.mcd$ cannot be empty (line 20). If it still has $u.mcd \geq u.core$, we finally unlock u and terminate (line 21 and 25). Otherwise, we first decrease $u.core$ by 1 and set $u.t$ as 2 together, which has to be an atomic operation since $v.t$ indicates v's status for other workers (line 22). Then, we add u to R_p for propagation (line 23); also, we set $u.mcd$ to empty since the value is out of date, which can be calculated later if needed (line 24).

In the procedure $\texttt{CheckMCD}(u)$, we recalculate $u.mcd$ if it is empty (line 27). We initially set temporarily mcd as 0 (line 28), and then we count $u.mcd$ (lines 29 - 33). Here, $u.mcd$ is the number of $v \in u.adj$ for two cases: 1) $v.core \geq u.core$, or 2) $v.core = u.core - 1$ and $v.t > 0$ (line 29); if either one is satisfied,we add the temporal mcd by 1 (line 30). When $v.core = K - 1$, it is possible that $v.t$ is been updated by other workers. If $v.t$ equals 1, we know that v is been propagating. In this case, we have to set $v.s$ from 1 to 3 by the atomic primitive $\texttt{CAS}$, which leads to v redo the propagation in line 16 by other workers (line 32). Here, we skip executing $\texttt{CAS}$ when $v = w$ (line 32) to avoid many useless redo processes in line 13. If $v.t$ is reduced to 0, the propagation of v is finished so that v cannot be counted as $u.mcd$ and the temporary mcd is off by 1 (line 33). Finally, we set $u.mcd$ as the temporary mcd and terminate (line 34). The big advantage is that we calculate $u.mcd$ without locking all neighbors $u.adj$ of v.

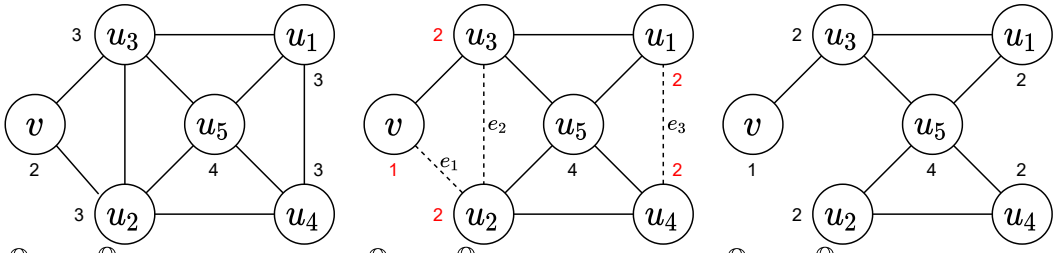


Figure 6.2: An example graph maintains the core numbers after removing 3 edges, e_1, e_2, and e_3. The letters inside the cycles are vertices' IDs and the $\mathbb{O}_k$ is the k-order of vertices with core numbers k. The beside numbers are corresponding mcd. (a) an initial example graph. (b) remove three edges. (c) the core numbers and $\mathbb{O}_k$ update.

Algorithm 23: RemoveEdge$_p(G, \mathbb{O}, (u, v))$

```
1  Lock u and v together if both are not locked
2  K, R_p, V_p* ← Min(u.core, v.core), an empty queue, ∅
3  CheckMCD_p(u, ∅); CheckMCD_p(v, ∅)
4  remove (u, v) from G
5  if v.core ≥ K then DoMCD_p(u)
6  if u.core ≥ K then DoMCD_p(v)
7  Unlock u if u ∉ R_p; Unlock v if v ∉ R_p
8  while R_p ≠ ∅ do
9      w, A_p ← R_p.dequeue(), ∅
10     ⟨w.t ← w.t − 1⟩                                   // atomically sub
11     for w' ∈ w.adj : w' ∉ A_p ∧ w'.core = K do
12         if Lock(w') with w'.core = K then
13             CheckMCD_p(w', w); DoMCD_p(w')
14             A_p ← A_p ∪ {w'}
15     ⟨w.t ← w.t − 1⟩                                   // atomically sub
16     if w.t > 0 then goto line 10
17 Append all u ∈ V_p* at the tail of O_{K−1} in k-order
18 Unlock all locked vertices

19 procedure DoMCD_p(u)
20     u.mcd ← u.mcd − 1                                 // u is locked
21     if u.mcd < K then
22         ⟨u.core ← K − 1; u.t = 2⟩                     // atomic operation
23         R_p.enqueue(u); u.mcd ← ∅
24         V_p* ← V_p* ∪ {u}; Delete(O, u)
25     else Unlock(u)

26 procedure CheckMCD_p(u, w)
27     if u.mcd ≠ ∅ then return
28     mcd ← 0
29     for v ∈ u.adj : v.core ≥ K ∨ (v.core = K − 1 ∧ v.t > 0) do
30         mcd ← mcd + 1
31         if v.core = K − 1 then
32             if v ≠ w ∧ v.t = 1 then CAS(v.t, 1, 3)
33             if v.t = 0 then mcd ← mcd − 1
34     u.mcd ← mcd
```

Example 6.3.2. *Sequential.* In Figure 6.2, we show an example graph that maintains the core numbers of vertices after removing three edges, e_1 to e_3, successively. Figure 6.2(a) shows that v has a core number of 2 with k-order $\mathbb{O}_2$ and all u_1 to u_5 have core numbers of 3 with k-order $\mathbb{O}_3$. We can see that for all vertices the core numbers are less or equal to mcd.

Figure 6.2(b) shows edges, e_1, e_2 and e_3, removed. (1) For e_1, $v.mcd$ is off by 1 so that we have $v.mcd < K = 2$ and $V^* = \{v\}$, but $u_2.mcd$ is not affected. Then, there is no propagation since all $v.adj$ have core numbers larger than $K = 2$. Finally, we decrease $v.core$ from 2 to 1. (2) For e_2, both $u_2.mcd$ and $u_3.mcd$ are off by 1 and less than $K = 3$, so that $V^* = \{u_2, u_3\}$. Then, both u_2 and u_3 are added into R for propagation, and u_1, u_4 and u_5 are consecutively added into V^* with $V^* = \{u_2, u_3, u_1, u_4, u5\}$. Finally, we decrease the core numbers of u_1 to u_5 from 3 to 2; also, the mcd of both u_2 and u_3 are updated to 2, and the mcd of u_1, u_4 and u_5 are updated to 3. (3) For e_3, both $u_2.mcd$ and $u_3.mcd$ are off by 1. But their mcd are still not less than $K = 2$ so that $V^* = \emptyset$. The propagation stop. Finally, the mcd of both u_1 and u_4 are updated to 2.

Figure 6.2(c) shows the result after removing edges. We can see that all vertices have their core numbers decreased by 1. Orders $\mathbb{O}_1$ and $\mathbb{O}_2$ are updated accordingly. Also, all vertices' mcd are updated accordingly.

Parallel. Continuing with Figure 6.2, we show an example of maintaining the core numbers of vertices in parallel when removing three edges. Figure 6.2(b) shows three edges, e_1, e_2 and e_3, being removed in parallel by three workers, p_1, p_2, and p_3, respectively. (1) For e_1, worker p_1 will lock v and u_2 together for removing the edge. But u_2 is already locked by p_2, so p_1 has to wait for p_2 to unlock u_2. Then, u_2 is unlocked without changing $u_2.mcd$, and the core number of v is off by 1 added to R_1 for propagation. Since only one $u_3 \in v.adj$ has a core number greater than v, the propagation of v terminates. Finally, v is unlocked. (2) For e_2, the worker p_2 first locks u_2 and u_3 together for removing the edge. Then, both $u_2.core$ and $u_3.core$ are off by 1, and u_2 and u_3 are added to R_2 for propagation. For propagating u_2, we traverse all $u_2.adj$; the vertex u_4 is locked by the worker p_3. At the same time, $u_4.core$ is decreased from 2 to 1 and p_1 will skip locking u_4 since the condition is not satisfied for the conditional lock. Vertex u_5 is locked by p_2 and has $u_5.mcd$ off by 1. Similarly, for propagating u_3, we traverse all $u_3.adj$ by skipping u_1 and decreasing $u_5.mcd$. Now, we have $u_5.mcd = 2 < u_5.core = 3$, so $u_5.core$ is off by 1. Finally, we unlock u_2, u_3, and u_5; all their core numbers are 2 now. (3) For e_3, worker p_3 will first lock u_1 and u_4 together for removing the edge. Then both $u_1.core$ and $u_4.core$ are off by 1. Vertices u_1 and u_4 are added to R_3 for propagation. The propagation will stop since the neighbors of u_1 and u_4 (u_3, u_2, and u_5) are locked by p_2 and have decreased core numbers. Finally, we unlock u_1 and u_4; all their core numbers are 2 now. We can see p_2 and p_3 execute without blocking each other, and only vertices in V^* are locked.

The above example assumes that the mcd of all vertices is initially generated. Suppose $u_3.mcd = \varnothing$ before removing e_2, we have to calculate $u_3.mcd$ by `CheckMCD`. At this time, u_2 and u_5 are counted as $u_3.mcd$ since they are not locked by p_3, but u_1 is locked by p_3 for propagation. The key issue is whether

u_1 is counted as $u_3.mcd$ or not. There are two cases. (1) If $u_1.core = 3$, we increment $u_3.mcd$ by 1. (2) If $u_1.core$ is decreased to 2 and u_1 is propagating, we also increment $u_3.mcd$ by 1. Since it is possible that u_1 has already propagated u_3, we force u_1 to redo the propagation by setting $u_1.t$ from 1 to 3 atomically.

Correctness. Algorithm 23 has no deadlocks. First, both u and v are locked together for a removed edge (u, v) (line 1). Second, for all vertices $w \in R_p$, we have w locked by the worker p and $w.core = K - 1$; also, worker p will lock all $w' \in w.adj$ with $w.core = K$ for propagation (lines 11 and 12). There are four cases:

- if all w' are not locked, there are no deadlocks;
- if w' is locked by another worker q but $w'.core$ is not decreased, there are no deadlocks as w' has no propagation and worker p will wait until w' is unlocked by q;
- if w' is locked by another worker q and w' always has $w'.core > K$, there are no deadlocks as w' is skipped for traversing.
- importantly, if w' is locked by the other worker q and $w'.core$ is decreased from K to $K - 1$, there are no deadlocks as w has propagation stopped on w' for $w'.core = K - 1$ and w' has propagation stopped on w for $w.core = K - 1$. We use "`Lock with`" to conditionally lock w' with $w'.core = K$, which ensure to stop busy-waiting when $w'.core$ decreases from K to $K - 1$.

The key issue of Algorithm 23 is to correctly maintain the mcd of all vertices in the graph:

$$\forall v \in V : v.mcd = |\{w \in v.adj : w.core \geq v.core\}| \tag{6.3.1}$$

All vertices v in the graph satisfy $v.mcd \geq v.core$; when removing an edge, v with $v.mcd < v.core$ are repeatedly added to V_p^* and have their core numbers decreased by 1 in order to make $v.mcd \geq v.core$. After deleting one edge, the vertices with decreased core numbers are added into R_p for propagation. The key issue is to argue the correctness of the while-loop (lines 8 to 16) for propagation.

We first define some useful notations. For all vertices $v \in V$, we use $v.lock$, a boolean value, to denote v is locked and $\neg v.lock$ to denote v is unlocked. We use R to denote the union of all propagation queues R_p, denoted as $R = \cup_{p=1}^{\mathcal{P}} R_p$, and a vertex $v \in R$ indicates v can be in one of the R_p for worker p.

The invariant of the outer while-loop (lines 8 - 16) is that all vertices $w \in V$ maintain a status $w.t$ indicating w in R or not; for all vertices $w \in R_p$, which

are locked and added into V^*, their core numbers are off by 1 and *mcd* set as empty (waiting to be recalculated); also, for all vertices $w \in V$, if $w.mcd$ is not empty, $w.mcd$ is the number of neighbors u that have core numbers that are 1) greater or equal $w.core$, or 2) equal to $w.core - 1$ with u in R waiting to be propagated:

$$\begin{aligned}
&(\forall w \in V : (w.t > 0 \equiv w \in R) \wedge (w.t = 0 \equiv w \notin R)) \\
&\wedge\ (\forall w \in R_p : w.core = K - 1 \wedge w.mcd = \varnothing \wedge w \in V_p^* \\
&\quad \wedge w.lock \wedge w.t > 0) \\
&\wedge\ (\forall u \in V : u.mcd \neq \varnothing \implies u.mcd = |\{v \in u.adj : v.core \geq \\
&\quad u.core\ \vee\ (v.core = u.core - 1 \wedge v \in R)\}|)
\end{aligned}$$

The invariant initially holds as vertices u and v may be added to R_p due to deleting an edge (u, v) and u or v is locked if added into R_p. We now argue the while-loop preserve the invariant:

- $\forall w \in V : (w.s > 0 \equiv w \in R) \wedge (w.s = 0 \equiv w \notin R)$ is preserved as $w.t$ is set to 2 and w is added to R at the same time by the atomic operation in line 22; also, $w.s$ is off to 0 when w is removed from R_p for propagation.
- $\forall w \in R_p : w.core = K - 1 \wedge w.mcd = \varnothing \wedge w \in V_p^* \wedge w.lock \wedge w.s > 0$ is preserved as when adding w to R_p, $w.core$ is off by 1, $w.mcd$ is set to empty, $w.t$ is set to 2, and w is added to V^*; also, w is locked before added into R.
- $\forall u \in V : u.mcd \neq \varnothing \implies u.mcd = |\{v \in u.adj : v.core \geq u.core \vee (v.core = u.core - 1 \wedge v \in R)\}|$ is preserved as when $u.mcd$ are calculated by the `CheckMCD` procedure, u may have neighbors $v \in u.adj$ whose core numbers are off by 1 and added to R by other workers and the propagation has not yet happened. Note that, the atomic operation in line 22 ensures that u has the core number off by 1 and added to R at the same time.

At the termination of the while-loop, the propagation queue $R = \emptyset$, so that all vertices $w \in V$ have $w.mcd$ correctly maintained.

We now argue the correctness of the inner for-loop (lines 11 - 14), which is important to parallelism. There are two more issues with the inner for-loop. One is that $w'.core$ may be decreased from $K + 1$ to K concurrently by other workers after visiting w' (line 11), which may lead to some w' that have $w'.core$ decreased to K to be skipped. The other is that $v.core$ and $v.s$ may be updated concurrently by other workers (line 29).

We first define useful notations as follow. For the inner for-loop (lines 11 - 14), we denote the set of visited neighbors of w as $w.V$, so that $w.V = \emptyset$ before the for-loop, $w.V \subseteq w.adj$ when executing the for-loop, and $w.V = w.adj$ after the for-loop; also, we denote the set of A_p as $w.A_p$. Note that, we redo the

for-loop if $w.t > 0$ by resetting $w.V$ to empty (line 16). We use V^* to denote the union of all V_p^*, denoted as $V^* = \cup_{p=1}^{\mathcal{P}} V_p^*$, and vertex $v \in V^*$ indicates v can be in one of the V_p^* for worker p.

Of course, the outer while-loop (lines 8 - 16) invariant is preserved. The additional invariant of the inner for-loop is that for all vertices $u \in V$, if $u.mcd$ is not empty, $u.mcd$ is the number of neighbors v that have core numbers that are 1) greater or equal to $u.core$, 2) equal to $u.core - 1$ with $u \in R$, or 3) $w.core - 1$ which has u removed from R for propagation but v is not yet propagated by u; also, a status of $v.t = 1$ indicates that v is doing the propagation and $v.t = 0$ indicates that v has finished the propagation:

$$\begin{aligned}
&\forall w \in V : (w.t = 1 \vee w.t = 3 \equiv w.V \subseteq w.adj) \\
&\wedge\ (\forall u \in V : u.mcd \neq \varnothing \implies u.mcd = |\{v \in u.adj : \\
&\quad v.core \geq u.core \ \vee\ (v.core = u.core - 1 \wedge v \in R) \vee \\
&\quad (v.core = u.core - 1 \wedge v \notin R \wedge v \in V^* \wedge v \notin u.V \wedge v \notin u.A_p)\}|)
\end{aligned}$$

The invariant initially holds as we have $w.t = 1 \wedge w.V = \emptyset \wedge w.A_p = \emptyset$. We now argue that the inner for-loop preserves the invariant:

- $\forall w \in V : (w.t = 1 \vee w.t = 3 \equiv w.V \subseteq w.adj)$ is preserved as $w.t$ is set to 2 when w is added to R_p and $w.s$ is atomically off by 1 before and after the for-loop; also, $w.t$ may be atomically added by 2 by `CAS` when a neighbor w' in $w.adj$ is calculating its mcd.

- $\forall u \in V : u.mcd \neq \varnothing \implies u.mcd = |\{v \in u.adj : v.core \geq u.core \ \vee\ (v.core = u.core - 1 \wedge v \in R) \ \vee (v.core = u.core - 1 \wedge v \notin R \wedge v \in V^* \wedge v \notin u.V \wedge v \notin A_p)$ is preserved as when $u.mcd$ is calculated by the `CheckMCD` procedure, u may have neighbors $v \in u.adj$ whose core numbers are off by 1 and added to R for further propagation; also, it is possible that v is added to V^* and already been removed from R before the inner for-loop (line 9) for propagation, which has three cases:

  1. if v not yet traversed u such that $v.core = u.core - 1$ for propagation as $u \notin v.A_p$, u should count v as $u.mcd$.
  2. if v has already traversed u such that $v.core = u.core - 1$ for propagation as $u \in v.A_p$, u should not count v as $u.mcd$.
  3. if v has already traversed u such that $v.core \neq u.core - 1$ for propagation, but after that $u.core$ has been updated to $v.core = u.core - 1$, u should count v as $u.mcd$.

  The third case requires the repeated traversing of u. We use $v.t = 1$ to let u know that v is executing the inner for-loop (lines 11 - 14) for propagating all vertices in $v.adj$. When $v.t = 1$, u will atomically add $v.t$

by 1 (line 32) and the propagation of v can run again with $v.A_p$ avoiding repeated propagation (line 16).

At the termination of the inner for-loop by $w.t = 0$, we have $w.V = w.adj$ so that the invariant of the while-loop holds. At the termination of the outer while-loop, the propagation queue $R = \emptyset$, so that all vertices $v \in V$ have $v.mcd$ correctly maintained as in Equation 6.3.1.

Time Complexity. When m' edges are removed from the graph, the total work is the same as the sequential version in Algorithm 15, which is $O(m'|E^*|)$ where E^* is the largest number of adjacent edges for all vertices in V^* among each removed edge, defined as $E^* = \sum_{v \in V_p^*} v.deg$. Analogies to edge insertion, the best-case running time is $O(m'|E^*|/\mathcal{P} + |E^*| + m'|V^*|)$; the worst-case running time is $O(m'|E^*|)$. In practice, such a worst-case is unlikely to happen.

Space Complexity. For each vertex $v \in V$, it takes $O(1)$ space to store $v.mcd$ and locks, which makes $O(n)$ space in total. Each worker p maintains a private V_p^*, which takes $O(|V^*|\mathcal{P})$ space in total. Each worker p maintains a private R_p, which takes $O(|E^*|\mathcal{P})$ space in total since at most $O(|E^*|)$ vertices can be added to R_p for each removed edge. The OM data structure is used to maintain the k-order for all vertices in the graph, which takes $O(n)$ space. Therefore, the total space complexity is $O(n + |V^*|\mathcal{P} + |E^*|\mathcal{P}) = O(n + |E^*|\mathcal{P})$.

6.4 Implementation

The min-priority queue Q is used in core maintenance for edge insertion to efficiently obtain a vertex $v \in Q$ with a minimum k-order $\mathbb{O}$, where $\mathbb{O}$ is maintained by the parallel OM data structure. The Q is implemented with min-heap by comparing the labels maintained by the parallel OM data structure, which supports enqueue and dequeue in $O(\log|Q|)$ time.

The key issue is how to efficiently implement the enqueue and dequeue operations, when the labels of vertices in $\mathbb{O}_k$ are updated by the relable procedure (including rebalance and split) in OM data structure. The solution is to re-make the min-heap of Q when each time the relable procedure is triggered in $\mathbb{O}_k$. For the implementation, we require the following structures:

– Each k-order $\mathbb{O}_k$ maintains a version number $\mathbb{O}_k.ver$, which is atomically added by 1 before and after one triggered relable procedure.

– Each k-order $\mathbb{O}_k$ maintains a counter $\mathbb{O}_k.cnt$, which can atomically record how many workers are executing the triggered relable procedure.

– The vertex v is added to Q along with the current top-label (group label), bottom-label, status, and version number, denoted as $[L_b(v), L^t(v), v.s, ver]$. These two labels are used for min-priority in Q.

– All vertices $v \in Q$ should have the same version number, which equals to Q's version number $Q.ver$.

Definition 6.4.1 (Version Invariant)**.** All vertices v in Q maintain the invariant that all $v.ver$ are the same version numbers as $Q.ver$, denoted as $\forall u, v \in Q : u.ver = v.ver = Q.ver$.

All vertices v in Q preserve the above Version Invariant for the `dequeue` operation, as the inconsistent version numbers of vertices may lead to wrong results. The detailed steps of updating $Q.ver$ are shown in Algorithm 24. Initially, we set ver' as the current version of $\mathbb{O}_K$ (line 1). If $ver' \neq Q.ver$, all vertices v in Q will update their $[L_b(v), L^t(v), v.s, ver']$ to the current new values, with ver' as their version numbers (lines 4 - 7). We have to ensure that $\mathbb{O}_k.cnt = 0$ and $ver' = \mathbb{O}_k.ver$ during such updating; otherwise, we will redo the updating (lines 2 and 8). We also have to ensure that $v.s$ is an even number and not changed during updating; otherwise, we have to redo the updating (lines 5 and 7) since other workers have accessed the vertices in Q and their k-order may be changed. In other words, no other workers can be executed during the updating. Finally, we set $Q.ver$ to ver' since all versions in Q have the same version as ver' (line 6).

Algorithm 24: $Q.update\_version(\mathbb{O}_k)$

```
1 ver' ← O_K.ver
2 if O_k.cnt ≠ 0 ∨ ver' ≠ O_K.ver then goto line 1
3 if ver' ≠ Q.ver then
4 |   for v ∈ Q do
5 |   |   s' ← v.s
6 |   |   Update v with current [L_b(v), L^t(v), s', ver']
7 |   |   if ¬Even(s') ∨ s' ≠ v.s then goto line 5
8 if O_K.cnt ≠ 0 ∨ ver' ≠ O_K.ver then goto line 1
9 Q.ver ← ver'
```

Enqueue. The details steps of enqueue operation are shown in Algorithm 25. We set ver' as current version of $\mathbb{O}_K$ (line 1). For the vertex v, we add the values of $[L_b(v), L^t(v), v.s, ver']$ to Q, with ver' as their version numbers (line 2), and then update the Q. If ver' is not consistent with $\mathbb{O}_k.ver$ or $Q_p.ver$, we set $Q_p.ver$ as $\varnothing$, which indicates the delayed version updating when executing `dequeue` operations.

Algorithm 25: $Q.enqueue(\mathbb{O}_k, v)$

```
1 ver', s' ← 𝕆_K.ver, v.s
2 Add v into Q with [L_b(v), L^t(v), v.s, ver'] and then update Q
3 if ver' ≠ 𝕆_K.ver ∨ ver' ≠ Q.ver ∨ s' ≠ s ∨ ¬Even(s) then
4 |   Q.ver ← ∅
```

Dequeue. Algorithm 26 shows the detailed steps of dequeue operations. If $Q.ver$ is empty, we will update the version of Q so that all vertices in Q have consistent labels (line 2). In this case, we obtain v as $Q.front()$ which has the lowest k-order by comparing the labels(line 3). We conditionally lock v with $v.core = k$ as we will skip v when it has $v.core \neq k$ for the core maintenance (lines 4 and 5), since v can be accessed by other workers and has an increased core number. After locking v, it is necessary to check v's current status value $v.s$ with v's status value in Q (lines 6 and 7). If they are not equal, we know that v has been accessed by other workers, $v.core = k$, and v's k-order may be changed; then $Q.ver$ is set to empty to update the version in the next round (line 7). We remove v from Q and then return v, which is locked with the smallest k-order with $v.core = k$ (line 11). The whole process continues until we successfully obtain v from Q or Q is empty (lines 1 and 8). If no qualified v exist in Q, it will return empty (line 9).

Algorithm 26: $Q.dequeue(\mathbb{O}_k)$

```
1 while Q ≠ ∅ do
2 |   if Q.ver = ∅ then Q_p.update_version(𝕆_k)
3 |   v ← Q.front()
4 |   if ¬(Lock v with v.core = k) then
5 |   |   Remove v from Q; continue
6 |   if v.s ≠ [v.s]_Q then
7 |   |   Unlock(v); Q_p.ver ← ∅; continue
8 |   Remove v from Q_p; return v
9 return ∅
```

Running Time. The priority queue can be implemented by min-heap, which requires worst-case $O(\log|Q|)$ time for both enqueuing and dequeuing one item. For our implementation with updating versions, it requires worst-case $O(\log|Q)$ time for enqueuing and $O(|Q|\log|Q|)$ time for dequeuing, as we may rebuild the min-heap when removing a vertex. However, such a worst-case can happen with a low probability, as vertices are always inserted into the different positions of $\mathbb{O}_K$ and a limited number of relable procedures can be triggered. Also, when inserting a batch of edges, the sizes of Q are always

small, e.g. less than 10, so that the process of updating versions tends to not affect the dequeue performance.

6.5 Experiments

In this section, we experimentally compare the following core maintenance approaches:
- The *Join Edge Set* based parallel edge insertion algorithm (`JEI` for short) and removal algorithm (`JER` for short) (Hua et al., 2019)
- The *Matching Edge Set* based parallel edge insertion (`MI` for short) and removal algorithm (`MR` for short) (Jin et al., 2018)
- Our parallel edge insertion algorithm (`OurI` for short) and removal algorithm (`OurR` for short)
- As baselines, the sequential Simplified-Order edge insertion algorithm (`OI` for short) and removal algorithm (`OR` for short) (Guo and Sekerinski, 2022c)
- As baselines, the sequential Traveral edge insertion algorithm (`TI` for short) and removal algorithm (`TR` for short) (Sarıyüce et al., 2016)

The source code is available on GitHub[1].

Experiment Setup. The experiments are performed on a server with an AMD CPU (64 cores, 128 hyperthreads, 256 MB of last-level shared cache) and 256 GB of main memory. Each core corresponds to a worker. The server runs the Ubuntu Linux (22.04) operating system. All tested algorithms are implemented in C++ and compiled with g++ version 11.2.0 with the -O3 option. OpenMP[2] version 4.5 is used as the threading library. We perform every experiment at least 50 times and calculate their means with 95% confidence intervals. Note that, the error bars are too small to see in our experiments.

Tested Graphs. We evaluate the performance of different methods over a variety of real-world and synthetic graphs shown in Table 6.2. For simplicity, directed graphs are converted to undirected ones; all of the self-loops and repeated edges are removed. That is, a vertex cannot connect to itself, and each pair of vertices can connect with at most one edge. The *livej*, *patent*, *wiki-talk*, and *roadNet-CA* graphs are obtained from SNAP[3]. The *dbpedia*, *baidu*, *pokec* and *wiki-talk-en wiki-links-en* graphs are collected from the KONECT[4] project. The *ER*, *BA*, and *RMAT* graphs are synthetic graphs generated by

[1] `https://github.com/Itisben/Parallel-CoreMaint.git`
[2] `https://www.openmp.org/`
[3] `http://snap.stanford.edu/data/index.html`
[4] `http://konect.cc/networks/`

the SNAP[5] system using Erdös-Rényi, Barabasi-Albert, and the R-MAT graph models, respectively; the average degree is fixed to 8 by choosing 1,000,000 vertices and 8,000,000 edges. All the above twelve graphs are static graphs. We randomly sample 100,000 edges for insertion and removal.

We also select four real temporal graphs, *DBLP*, *Flickr*, *StackOverflow*, and *wiki-edits-sh* from KONECT; each edge has a timestamp recording the time of this edge inserted into the graph. We select 100,000 edges within a continuous time range for insertion and removal.

Graph	$n = \|V\|$	$m = \|E\|$	AvgDeg	Max k
livej	4,847,571	68,993,773	14.23	372
patent	6,009,555	16,518,948	2.75	64
wikitalk	2,394,385	5,021,410	**2.10**	131
roadNet-CA	1,971,281	5,533,214	2.81	**3**
dbpedia	3,966,925	13,820,853	3.48	20
baidu	2,141,301	17,794,839	8.31	78
pokec	1,632,804	30,622,564	18.75	47
wiki-talk-en	2,987,536	24,981,163	8.36	210
wiki-links-en	5,710,993	130,160,392	22.79	**821**
ER	1,000,000	8,000,000	8.00	11
BA	1,000,000	8,000,000	8.00	8
RMAT	1,000,000	8,000,000	8.00	237
DBLP	1,824,701	29,487,744	16.17	286
Flickr	2,302,926	33,140,017	14.41	600
StackOverflow	2,601,977	63,497,050	**24.41**	198
wiki-edits-sh	4,589,850	40,578,944	8.84	47

Table 6.2: Tested real and synthetic graphs.

In Table 6.2, we can see all graphs have millions of edges. Their average degrees range from 2.1 to 24.4, and their maximal core numbers range from 3 to 821. In Figure 6.3, we can see that the core numbers of vertices are not uniformly distributed in all tested graphs, where the x-axis is core numbers and the y-axis is the number of vertices. That is, a great portion of vertices have small core numbers, and few have large core numbers. For example, *wikitalk* has 1.7 million vertices with a core number of 1; *roadNet-CA* has four core numbers from 0 to 3; *BA* only has a single core number of 8. For `JEI`, `JER`, `MI` and `MR`, such core number distribution is an important property since the vertices with the same core number can only be processed by a single worker at the same time, while `OurI` and `OurR` do not have this limitation.

[5] `http://snap.stanford.edu/snappy/doc/reference/generators.html`

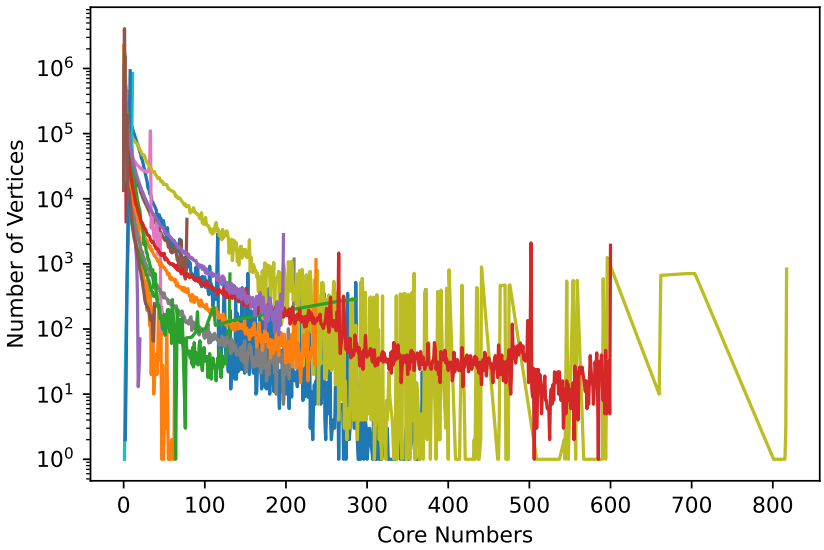


Figure 6.3: The distributions of vertices' core numbers.

6.5.1 Running Time Evaluation

In this experiment, we exponentially increase the number of workers from 1 to 64 to evaluate the real running time over graphs in Table 6.2. For the twelve static graphs, we randomly sample 100,000 edges. For the four temporal graphs, we select the latest period of 100,000 edges. These edges are first removed and then inserted. The accumulated running times are measured.

The plots in Figure 6.4 depict the performance of four compared algorithms, where the running times above 3600 seconds are not depicted. Comparing three parallel methods, the first look reveals that `OurI` and `OurR` always have the best performance and `MI` and `MR` always have the worst performance, respectively. Compared with the two baseline methods, we find that `OI` and `OR` are much more efficient than `TI` and `TR`, respectively. Specifically, we make several observations:

- By using one worker, `Our` and `OurR` have the same running time as the baselines of `OI` and `OR`, respectively. This is because `OurI` and `OurR` are based on `OI` and `OR` and have the same work complexities, respectively.

- By using one worker, `JEI` and `JER` are always faster than `TI` and `TR`, respectively. This is because although `JEI` and `JER` are based on `TI` and `TR`, a batch of insertions or removals are processed together and thus repeated computations can be avoided. Also, `MI` and `MR` have the same trend.

- By using one worker, all algorithms are reduced to sequential, and `OurI`

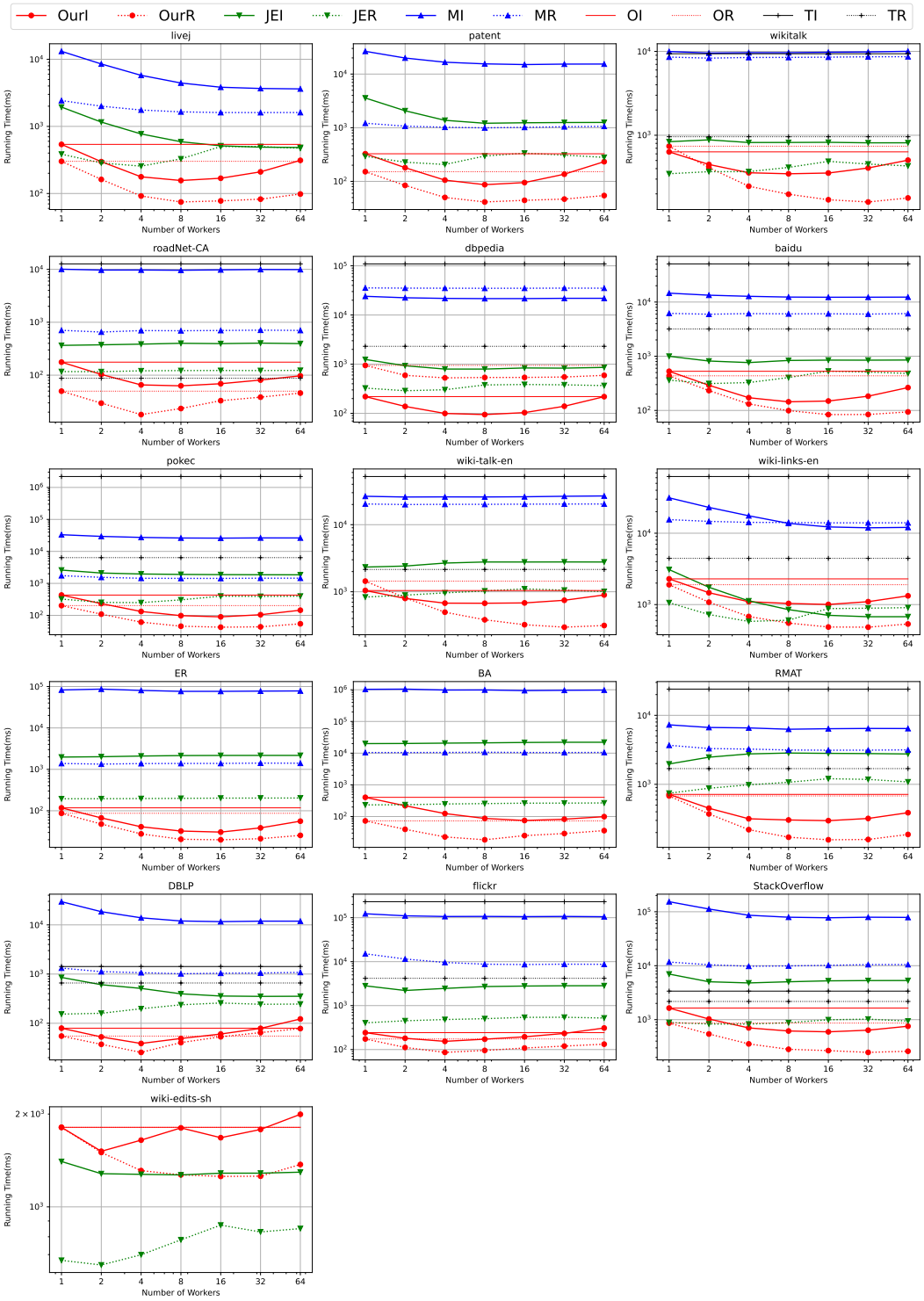


Figure 6.4: The real running time by varying the number of workers.

performs much faster than JEI. This is because for edge insertion, OurI is based on the OI, while JEI is based on TI. OI is much faster than TI. Also, MI and MR have the same trend.

– By using one worker, OurR does not always perform better than JER. This is because our method uses arrays to store edges, which can save space, while the join-edge-set-based method uses binary search trees to store edges. When deleting an edge (u, v), OurR has to traverse all vertices of $u.adj$ and $v.adj$, while JER only need to traverse $\log|u.adj|$ and $\log|v.adj|$ vertices. That means OurR costs more running time than JER for deleting an edge from the graph.

– By using multiple workers, OurI and OurR can always achieve better speedups compared with other parallel methods, but JEI and JER have no speedups over some graphs. This is because JEI and JER have limited parallelism, as affected vertices with different core numbers cannot perform in parallel, while OurI and OurR do not have such a limitation. Also, MI and MR have the same trend.

– By using multiple workers, the running time of OurI and OurR may begin to increase when using more than 8 or 16 workers, e.g. *livej*, *patent*, and *dbpedia*. This is because of the contention on shared data structures with multiple workers, and more workers may lead to higher contention. In addition, for JEI and JER, when the core numbers of vertices in graphs are not well distributed, some workers are wasted, which results in extra overheads.

Graph	1-worker vs 16-worker						1-worker OurI vs		1-worker OurR vs		16-worker OurI vs		16-worker OurR vs	
	OurI	OurR	JEI	JER	MI	MR	JEI	MI	JER	MR	JEI	MI	JER	MR
livej	3.2	3.9	3.8	0.8	**3.4**	1.5	3.6	24.4	0.7	4.5	3.0	22.7	3.0	**9.5**
patent	3.4	3.4	2.9	0.9	1.8	1.2	11.0	81.2	0.9	**3.7**	13.0	158.3	3.5	10.7
wikitalk	1.8	4.4	1.0	0.7	**1.0**	**1.0**	1.3	15.7	0.5	13.5	2.3	27.6	1.4	24.1
roadNet-CA	2.6	1.5	0.9	1.0	1.0	1.0	2.1	57.1	0.7	4.0	5.7	141.9	1.8	10.1
dbpedia	2.1	1.8	1.5	0.8	1.1	1.0	5.7	109.4	1.5	**162.0**	8.1	208.1	3.7	**337.9**
baidu	3.5	**5.2**	1.2	0.7	1.2	1.0	1.9	27.6	0.7	11.7	5.7	82.1	3.6	40.7
pokec	4.8	4.7	1.4	0.8	1.3	1.2	6.0	76.9	0.7	4.0	20.9	288.8	4.4	15.9
wiki-talk-en	1.5	4.5	0.8	0.8	1.0	1.0	2.2	25.4	0.8	19.5	4.1	38.2	1.6	29.7
wiki-links-en	2.3	3.9	**4.4**	**1.2**	2.6	1.1	1.3	13.7	0.5	6.8	**0.7**	**12.2**	0.9	13.9
ER	3.8	4.4	0.9	1.0	1.1	1.0	16.8	700.3	1.7	11.8	70.9	2500.7	**6.6**	45.5
BA	**5.4**	2.9	0.9	0.9	1.1	1.0	**49.6**	**2555.3**	0.6	25.9	**289.1**	**12552.9**	3.5	139.7
RMAT	2.4	4.3	**0.7**	**0.6**	1.1	1.2	2.8	**10.2**	1.0	5.2	9.5	21.5	4.1	10.5
DBLP	1.3	**1.0**	2.4	0.6	2.5	1.3	10.8	371.7	**1.9**	16.7	5.9	192.4	4.3	17.2
flickr	1.2	1.6	1.0	0.7	1.2	**1.8**	11.6	506.7	1.7	62.3	14.3	542.3	2.8	44.1
StackOverflow	2.8	3.2	1.3	0.9	2.0	1.2	4.3	93.5	0.5	7.1	8.8	130.0	1.7	17.1
wiki-edits-sh	**1.1**	1.4	1.1	0.8	-	-	**0.8**	-	**0.4**	-	0.8	-	**0.5**	-

Table 6.3: Compare the speedups.

In Table 6.3, columns 2 to 7 compare the running time speedups between using 1 worker and 16 workers for all tested algorithms. It is clear that `OurI` and `OurR` always achieve better speedups up to 5x, compared with other parallel methods. Columns 8 to 11 compare the running time speedups between our method and the other parallel methods by using 1 worker; columns 12 to 15 compare the running time speedups between our method and the other parallel methods by using 16 workers. We first can see that compared with `MI` and `MR`, `OurI` and `OurR` always achieve the highest speedups both using 1 worker and 16 workers. Compared with `JEI`, `OurI` achieves up to 50x speedups even using 1 worker and achieves up to 289x speedups when using 16 workers. We observe that compared with `JER`, `OurR` does not always achieve speedups when using a single worker, but achieves up to 10x speedups when using 16 workers. Especially, over *wiki-edits-sh*, `OurI` and `OurR` run slightly slower than `JEI` and `JER` when using 1 worker and 16 workers, respectively. The reason is that the special properties of graphs may affect the performance of different algorithms.

6.5.2 Scalability Evaluation

In this experiment, we test the scalability over four selected graphs, e.g., *livej*, *baidu*, *dbpedia*, *roadNet-CA*. For each graph, we first randomly select from 100,000 to 1 million edges. By using 16 workers, we measure the accumulated running time and evaluate the ratio of real running time between the corresponding size of edges and 100,000 edges. The plots in Figure 6.5 depict the performance of four compared algorithms. The x-axis is the size of inserted or removed edges, and the y-axis is the time ratio. Ideally, 1 million edges should have a ratio of 10 since the edge size is 10 times of 100,000. We observe that over *livej*, four algorithms always have similar time ratios with increased edge size. Over other graphs, `OurI` and `OurR` always have larger time ratios compared with `JEI` and `JER`, respectively. Further, `OurI` has a time ratio of up to 20 when applying 1 million edges. This is because `JEI` or `JER` adopts the joint edge set structure to preprocess a batch of updated edges; if there are more updated edges, they can process more edges in each iteration and avoid unnecessary access. However, `OurI` and `OurR` do not preprocess a batch of updated edges so more updated edges require more accumulated running time.

We also observe that even with 1 million update edges, `OurI` and `OurR` still have better performance than `JEI` and `JER`, respectively. Over four tested graphs, `OurI` still has 2.6x, 1.9x, 3.8x and 3.0x speedups compared with `JEI`, and `OuR` also has 7.8x, 5.5x, 0.9x and 3.3x speedups compared with `JER`, respectively. The reason is that `OurI` and `OurR` (based on the Order algorithm) have less work than `JEI` and `JER` (based on the Traversal algorithm; also,

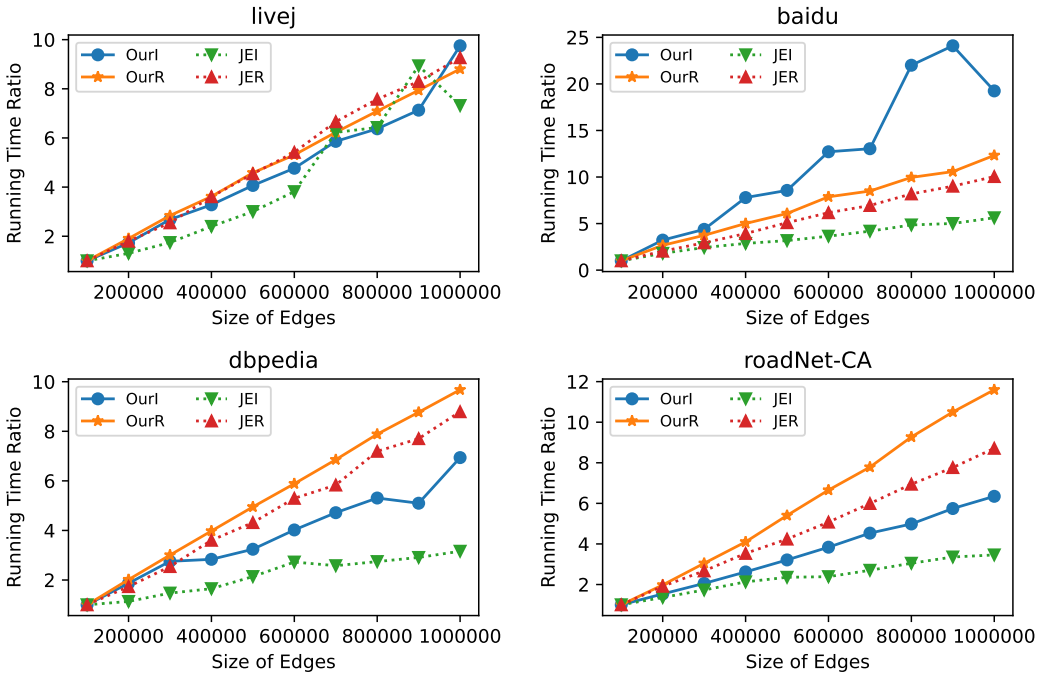


Figure 6.5: The running time ratio with 16-worker by varying the size of inserted or removed edges.

unlike `OurI` and `OurR`, `JEI` and `JER` have an extra cost to preprocess the edges.

6.5.3 Stability Evaluation

In this experiment, we test the stability over four selected graphs, e.g., *livej*, *baidu*, *dbpedia*, *roadNet-CA*, by using 16 workers. First, we randomly sample $5,000,000$ edges and partition them into 50 groups, where each group has totally different $100,000$ edges. Second, for each group, we measure the accumulated running time of different methods. That is, the experiments run 50 times so each time it has totally different inserted or removed edges.

The plots in Figure 6.6 depict the result. The x-axis is the repeated times, and the y-axis is the running times. We observe that the performance of `OurI`, `OurR`, and `JER` are always well-bounded, but the performance of `JEI` always has large fluctuations. The reason is that `JEI` is based on the Traversal algorithm and `OurI` is based on the Order algorithm. It is proved that for the edge insertion, the Traversal algorithm has large fluctuations for the ratio of $|V^+|/|V^*|$ for different edges with high probability, while the Order algorithm does not have this problem. For the edge removal, both `Our` and `JER` have $V^+ = V^*$, so their running times remain stable for different batches of edges.

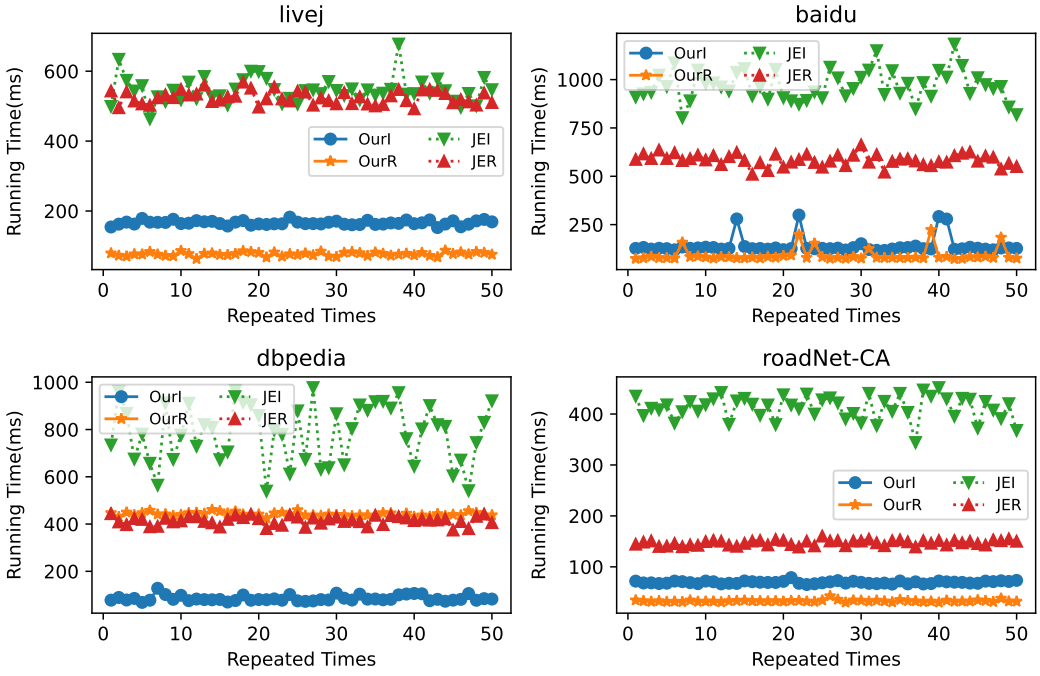


Figure 6.6: The running time with 16-worker by varying a batch of inserted or removed edges for each time.

Part IV

Conclusion

Chapter 7

Conclusion

7.1 Summary

This thesis presents a set of parallel graph algorithms in the shared-memory work-depth model. In Part II, Chapter 3 studies graph trimming algorithms for removing vertices without outgoing edges. The arc-consistency algorithms, in particular AC-3, AC-4, and AC-6, can be applied to graph trimming, leading to the so-called AC-3-based, AC-4-based, and AC-6-based trimming algorithms, respectively. Based on that, we propose parallel AC-4-based and AC-6-based trimming algorithms that have better worst-case time complexities than AC-3-based. The common existing graph trimming method is actually parallel AC-3-based, which has the worst time complexity. Although AC-4-based and AC-6-based algorithms have similar worst-case time complexities, the AC-6-based algorithm traverses fewer edges per worker and requires less memory usage than the AC-4-based one. For implicit graphs in which edges are generated on-the-fly, the AC6-based algorithm does not rely on the reversed graphs, unlike the AC4-based algorithm, and thus is more suitable for trimming implicit graphs. For explicit graphs in which all edges are linearly stored in memory, our AC-6-based algorithm does not always outperform the other methods, but always traverses the least number of edges and has the best stability and scalability.

In Part IV, we study the parallel core maintenance algorithms. In Chapter 4, we present a new parallel order maintenance (OM) data structure. The parallel `Insert` and `Delete` are synchronized with locks efficiently. Notably, the parallel `Order` is lock-free and can execute highly in parallel. Experiments demonstrate significant speedups (for 64 workers) over the sequential version on a variety of test cases. In Chapter 5, we simplify the state-of-the-art core maintenance algorithm and also improve its worst-case time complexity by introducing the sequential Order Data Structure (in Chapter 4) to maintain the k-order of all vertices in the graph. Our simplified approach is easy to

understand, implement, and argue the correctness. The experiments show that our approach outperforms the existing methods over a variety of data graphs. In Chapter 6, we parallelize our simplified core maintenance algorithm (in Chapter 5) to handle a batch of inserted or removed edges in parallel. We adopt the parallel OM data structure (in Chapter 4) to maintain the k-order of all vertices in the graph. We also propose novel mechanisms to avoid deadlocks. The experiments show that our parallel edge insertion is much faster than the existing approaches even using a single worker over various data graphs; also, for both edge insertion and removal, our parallel approach achieves high speedups compared with the existing approach when using 16 workers. More importantly, our parallel approach achieves high parallelism even when the core numbers of vertices are not well-distributed, e.g. all vertices have the same core numbers in a graph. However, in this case, existing approaches have limited parallelism.

7.2 Future Work

There are a number of open questions that remain as a result of this thesis. We can apply graph trimming to Strong Connected Components decomposition as a great percentage of size-1 SCCs can be trimmed in parallel. We also can apply graph trimming to cycle directions as the trimmable vertices cannot be in cycles and can be trimmed in parallel. Both applications depend on Depth First Search (DFS), which is hard to parallelize. However, our trimming techniques can efficiently trim graphs in parallel if there is a large portion of trimmable vertices in graphs. In particular, we can apply the AC-6-based algorithm to trim the model-checking graphs in which edges are expensively calculated on-the-fly; fewer traversed edges will likely save running time.

For both parallel Core Maintenance, we will investigate the insertions or deletions in batches by preprocessing the edges to avoid unnecessary access. In other words, we can process more edges in each iteration and each edge costs less running time on average. Similarly, such batch insertion and removal can also be applied to the parallel Order Maintenance data structure.

Our parallel methodology to core maintenance can be applied to *truss maintenance* (Zhang and Yu, 2019a) since their definitions are analogous. Specifically, given a graph $G = (V, E)$, the *k-truss* G_k is defined as the maximal subgraph in which each edge is contained in at least $k - 2$ triangles. For each edge $(u, v) \in E$, the *truss decomposition* is to compute the *trussness*, which is defined as the maximum k such that (u, v) resides in the k-truss but not in the $(k + 1)$-truss of G. Here, the truss maintenance is to maintain the trussnesses when edges are inserted or removed. We can see all edges maintain the order defined by the truss decomposition. When inserting an edge, such an order for all edges can be used to reduce the searching range to obtain the

edges whose trussnesses are increased. We can parallelize such a truss maintenance approach based on maintaining the order of edges by our parallel OM data structure.

Bibliography

Alok Aggarwal and Richard J Anderson. 1988. A random NC algorithm for depth first search. *Combinatorica* 8, 1 (1988), 1–12.

Gregory R Andrews. 1991. *Concurrent programming: principles and practice.* Benjamin-Cummings Publishing Co., Inc.

Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. DBpedia: A Nucleus for a Web of Open Data. In *The Semantic Web*. Springer Berlin Heidelberg, 722–735. https://doi.org/10.1007/978-3-540-76298-0_52

Vladimir Batagelj and Matjaz Zaversnik. 2003. An $O(m)$ Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003). http://arxiv.org/abs/cs/0310049

Michael A Bender, Richard Cole, Erik D Demaine, Martin Farach-Colton, and Jack Zito. 2002. Two simplified algorithms for maintaining order in a list. In *European Symposium on Algorithms*. Springer, 152–164.

Christian Bessière. 1994. Arc-consistency and arc-consistency again. *Artificial Intelligence* 65, 1 (Jan. 1994), 179–190. https://doi.org/10.1016/0004-3702(94)90041-8

Guy E Blelloch and Bruce M Maggs. 2010. Parallel algorithms. In *Algorithms and theory of computation handbook: special topics and techniques*. 25–25.

Vincent Bloemen. 2015. *On-The-Fly parallel decomposition of strongly connected components*. Master's thesis. University of Twente.

Vincent Bloemen, Alfons Laarman, and Jaco van de Pol. 2016. Multi-core on-the-fly SCC decomposition. *ACM SIGPLAN Notices* 51, 8 (Feb. 2016), 1–12. https://doi.org/10.1145/3016078.2851161

Kate Burleson-Lesser, Flaviano Morone, Maria S Tomassone, and Hernán A Makse. 2020. K-core robustness in ecological and financial networks. *Scientific reports* 10, 1 (2020), 1–14.

Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP*. Morgan kaufmann.

Xuhao Chen, Cheng Chen, Jie Shen, Jianbin Fang, Tao Tang, Canqun Yang, and Zhiying Wang. 2018. Orchestrating parallel detection of strongly connected components on GPUs. *Parallel Comput.* 78 (Oct. 2018), 101–114. https://doi.org/10.1016/j.parco.2017.11.001

Yangjun Chen, Bin Guo, and Xingyue Huang. 2019. Δ-Transitive closures and triangle consistency checking: a new way to evaluate graph pattern queries in large graph databases. *The Journal of Supercomputing* (Feb. 2019). https://doi.org/10.1007/s11227-019-02762-4

James Cheng, Yiping Ke, Shumo Chu, and M Tamer Özsu. 2011. Efficient core decomposition in massive networks. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 51–62.

Paul R. Cooper and Michael J. Swain. 1992. Arc consistency: parallelism and domain dependence. *Artificial Intelligence* 58, 1-3 (Dec. 1992), 207–235. https://doi.org/10.1016/0004-3702(92)90008-l

Don Coppersmith, Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. 2003. *A divide-and-conquer algorithm for identifying strongly connected components*. Technical Report. Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US). https://doi.org/10.2172/889876

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press. 603–611 pages.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55. https://doi.org/10.1109/99.660313

Naga Shailaja Dasari, Ranjan Desh, and Mohammad Zubair. 2014. ParK: An efficient algorithm for k-core decomposition on multicore processors. In *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 9–16.

Rodrick Kuate Defo, Richard Wang, and M Manjunathaiah. 2019. Parallel BFS implementing optimized decomposition of space and kMC simulations for diffusion of vacancies for quantum storage. *Journal of Computational Science* 36 (2019), 101018.

Mohammad Dib, Rouwaida Abdallah, and Alexandre Caminada. 2010. Arc-Consistency in Constraint Satisfaction Problems: A Survey. In *2010 Second International Conference on Computational Intelligence, Modelling and Simulation*. IEEE. https://doi.org/10.1109/cimsim.2010.18

Paul Dietz and Daniel Sleator. 1987. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 365–372.

Paul F Dietz. 1982. Maintaining order in a linked list. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*. 122–127.

Thomas Erlebach, Torben Hagerup, Klaus Jansen, Moritz Minzlaff, and Alexander Wolff. 2010. Trimming of graphs, with application to point labeling. *Theory of Computing Systems* 47, 3 (2010), 613–636.

Lisa K Fleischer, Bruce Hendrickson, and Ali Pınar. 2000. On identifying strongly connected components in parallel. In *International Parallel and Distributed Processing Symposium*. Springer, 505–511. https://doi.org/10.1007/3-540-45591-4_68

Lisa K. Fleischer, Bruce Hendrickson, and Ali Pınar. 2007. On Identifying Strongly Connected Components in Parallel. November 2014 (2007), 505–511. https://doi.org/10.1007/3-540-45591-4_68

E Freuder and J-Ch Régin. 1999. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence* 107, 1 (1999), 125–148. https://doi.org/10.1016/s0004-3702(98)00105-2

Kasimir Gabert, Ali Pinar, and Ümit V Çatalyürek. 2021. Shared-Memory Scalable k-Core Maintenance on Dynamic Graphs and Hypergraphs. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 998–1007.

Yi Gao, Wei Dong, Wenbin Wu, Chun Chen, Xiang-Yang Li, and Jiajun Bu. 2015. Scalpel: Scalable preferential link tomography based on graph trimming. *IEEE/ACM Transactions on Networking* 24, 3 (2015), 1392–1403.

Seth Gilbert, Jeremy Fineman, and Michael Bender. 2003. Concurrent Order Maintenance. *unpublished* (2003). https://ocw.mit.edu/courses/6-895-theory-of-parallel-systems-sma-5509-fall-2003/b5e012c63722c74d9d6504fef3caba00_gilbert.pdf

Bin Guo and Emil Sekerinski. 2022a. Efficient parallel graph trimming by arc-consistency. *The Journal of Supercomputing* (2022), 1–45.

Bin Guo and Emil Sekerinski. 2022b. New Parallel Order Maintenance Data Structure. *arXiv preprint arXiv:2208.07800* (2022).

Bin Guo and Emil Sekerinski. 2022c. Simplified Algorithms for Order-Based Core Maintenance. *arXiv preprint arXiv:2201.07103* (2022).

Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert E Tarjan. 2012. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Transactions on Algorithms (TALG)* 8, 1 (2012), 1–33.

Daniel Harabor and Alban Grastien. 2011. Online graph pruning for pathfinding on grid maps. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 25.

Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The art of multiprocessor programming*. Newnes.

Marijn JH Heule. 2019. Trimming graphs using clausal proof optimization. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 251–267.

Ramin Hojati, Robert K Brayton, and Robert P Kurshan. 1993. BDD-based debugging of designs using language containment and fair CTL. In *International Conference on Computer Aided Verification*. Springer, 41–58. https://doi.org/10.1007/3-540-56922-7_5

Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: a DSL for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. 349–362. https://doi.org/10.1145/2248487.2151013

Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. 2013. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*. ACM Press. https://doi.org/10.1145%2F2503210.2503246

Qiang-Sheng Hua, Yuliang Shi, Dongxiao Yu, Hai Jin, Jiguo Yu, Zhipen Cai, Xiuzhen Cheng, and Hanhua Chen. 2019. Faster parallel core maintenance algorithms in dynamic graphs. *IEEE Transactions on Parallel and Distributed Systems* 31, 6 (2019), 1287–1300.

William McLendon III, Bruce Hendrickson, Steven J. Plimpton, and Lawrence Rauchwerger. 2005. Finding strongly connected components in distributed graphs. *J. Parallel and Distrib. Comput.* 65, 8 (Aug. 2005), 901–910. https://doi.org/10.1016/j.jpdc.2005.03.007

Joseph JéJé. 1992. *An introduction to parallel algorithms*. Reading, MA: Addison-Wesley.

Yuede Ji, Hang Liu, and H. Howie Huang. 2018. iSpan: Parallel Identification of Strongly Connected Components with Spanning Trees. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. https://doi.org/10.1109/sc.2018.00061

Hai Jin, Na Wang, Dongxiao Yu, Qiang Sheng Hua, Xuanhua Shi, and Xia Xie. 2018. Core Maintenance in Dynamic Graphs: A Parallel Approach Based on Matching. *IEEE Transactions on Parallel and Distributed Systems* 29, 11 (nov 2018), 2416–2428. https://doi.org/10.1109/TPDS.2018.2835441 arXiv:1703.03900

Humayun Kabir and Kamesh Madduri. 2017. Parallel k-core decomposition on multicore platforms. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1482–1491.

Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. 2015. K-core decomposition of large networks on a single PC. *Proceedings of the VLDB Endowment* 9, 1 (2015), 13–23.

Lefteris M. Kirousis. 1993. *Fast parallel constraint satisfaction*. Technical Report 1. 147–160 pages. https://doi.org/10.1016/0004-3702(93)90063-h

Yi-Xiu Kong, Gui-Yuan Shi, Rui-Jie Wu, and Yi-Cheng Zhang. 2019. *k-core: Theories and applications*. Technical Report. 1–32 pages. https://doi.org/10.1016/j.physrep.2019.10.004

Ravi Kumar, Jasmine Novak, and Andrew Tomkins. 2010. Structure and Evolution of Online Social Networks. In *Link Mining: Models, Algorithms, and Applications*. Springer New York, 337–357. https://doi.org/10.1007/978-1-4419-6515-8_13

Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2013. Efficient core maintenance in large dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering* 26, 10 (2013), 2453–2465.

Zhe Lin, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Zhihong Tian. 2021. Hierarchical core maintenance on large dynamic graphs. *Proceedings of the VLDB Endowment* 14, 5 (2021), 757–770.

Bin Liu and Feiteng Zhang. 2020. Incremental Algorithms of the Core Maintenance Problem on Edge-Weighted Graphs. *IEEE Access* PP (04 2020), 1–1. https://doi.org/10.1109/ACCESS.2020.2985327

Gavin Lowe. 2016. Concurrent depth-first search algorithms based on Tarjan's Algorithm. *International Journal on Software Tools for Technology Transfer* 18, 2 (apr 2016), 129–147. https://doi.org/10.1007/s10009-015-0382-1

Alan K. Mackworth. 1977. Consistency in networks of relations. *Artificial Intelligence* 8, 1 (Feb. 1977), 99–118. https://doi.org/10.1016/0004-3702(77)90007-8

Alan K Mackworth and Eugene C Freuder. 1985. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial intelligence* 25, 1 (1985), 65–74. https://doi.org/10.1016/0004-3702(85)90035-9

Fragkiskos D Malliaros, Christos Giatsidis, Apostolos N Papadopoulos, and Michalis Vazirgiannis. 2020. The core decomposition of networks: Theory, algorithms and applications. *The VLDB Journal* 29, 1 (2020), 61–92. https://doi.org/10.1007/s00778-019-00587-4

Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert. 1996. Maintaining a topological order under edge insertions. *Inform. Process. Lett.* 59, 1 (1996), 53–58.

Robert C Martin, James Newkirk, and Robert S Koss. 2003. *Agile software development: principles, patterns, and practices*. Vol. 2. Prentice Hall Upper Saddle River, NJ.

Stephan Merz. 2001. Model Checking: A Tutorial Overview. In *Modeling and Verification of Parallel Processes*. Springer Berlin Heidelberg, 3–38. https://doi.org/10.1007/3-540-45510-8_1

Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel Algorithms and Architectures - SPAA '02*. ACM Press. https://doi.org/10.1145/564870.564881

Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. 2018. Bq: A lock-free queue with batching. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures - SPAA '18*. ACM Press. https://doi.org/10.1145/3210377.3210388

Daniele Miorandi and Francesco De Pellegrini. 2010. K-shell decomposition for dynamic complex networks. In *8th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*. IEEE, 488–496.

Roger Mohr and Thomas C. Henderson. 1986. Arc and path consistency revisited. *Artificial Intelligence* 28, 2 (March 1986), 225–233. https://doi.org/10.1016/0004-3702(86)90083-4

Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2012. Distributed k-core decomposition. *IEEE Transactions on parallel and distributed systems* 24, 2 (2012), 288–300.

Gordon E Moore. 1998. Cramming more components onto integrated circuits. *Proc. IEEE* 86, 1 (1998), 82–85.

Sen Pei, Lev Muchnik, José S Andrade, Jr, Zhiming Zheng, and Hernán A Makse. 2014. Searching for superspreaders of information in real-world social media. *Scientific reports* 4, 1 (2014), 5547.

Radek Pelánek. 2007. BEEM: Benchmarks for Explicit Model Checkers. In *Model Checking Software*. Springer Berlin Heidelberg, 263–267. https://doi.org/10.1007/978-3-540-73370-6_17

John H. Reif. 1985. Depth-first search is inherently sequential. *Inform. Process. Lett.* 20, 5 (June 1985), 229–234. https://doi.org/10.1016/0020-0190(85)90024-9

Etienne Renault, Alexandre Duret-Lutz, Fabrice Kordon, and Denis Poitrenaud. 2015. Parallel explicit model checking for generalized Büchi automata. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 9035. Springer Verlag, 613–627. https://doi.org/10.1007/978-3-662-46681-0_56

Stuart Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall Press, USA. 202–233 pages.

Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2013. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment* 6, 6 (2013), 433–444.

Ahmet Erdem Sarıyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2016. Incremental k-core decomposition: algorithms and evaluation. *The VLDB Journal* 25, 3 (2016), 425–447.

Julian Shun. 2017a. *Shared-memory parallelism can be simple, fast, and scalable*. Morgan & Claypool.

Julian Shun. 2017b. *Shared-memory parallelism can be simple, fast, and scalable*. PUB7255 Association for Computing Machinery and Morgan & Claypool.

George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS*. IEEE Computer Society, 550–559. https://doi.org/10.1109/IPDPS.2014.64

Bintao Sun, T-H Hubert Chan, and Mauro Sozio. 2020. Fully dynamic approximate k-core decomposition in hypergraphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 14, 4 (2020), 1–21.

Jun Sun, Jerome Kunegis, and Steffen Staab. 2016. Predicting User Roles in Social Networks Using Transfer Learning with Feature Transformation. In *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*. IEEE, IEEE, 128–135. https://doi.org/10.1109/icdmw.2016.0026

Lubos Takac and Michal Zabovsky. 2012. Data analysis in public social networks. In *International Scientific Conference and International Workshop Present Day Trends of Innovations*, Vol. 1.

Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (June 1972), 146–160. https://doi.org/10.1137/0201010

Athanasios K Tsakalidis. 1984. Maintaining order in a generalized linked list. *Acta informatica* 21, 1 (1984), 101–112.

Robert Utterback, Kunal Agrawal, Jeremy T Fineman, and I-Ting Angelina Lee. 2016. Provably good and practically efficient parallel race detection for fork-join programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. 83–94.

John D. Valois. 1995. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing - PODC '95*. ACM Press, 214–222. https://doi.org/10.1145/224964.224988

Na Wang, Dongxiao Yu, Hai Jin, Chen Qian, Xia Xie, and Qiang-Sheng Hua. 2017. Parallel algorithm for core maintenance in dynamic graphs. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2366–2371.

Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393, 6684 (1998), 440. https://doi.org/10.1515/9781400841356.301

Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2016. I/O efficient core graph decomposition at web scale. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 133–144.

Tongfeng Weng, Xu Zhou, Kenli Li, Peng Peng, and Keqin Li. 2021. Efficient distributed approaches to core maintenance on large dynamic graphs. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2021), 129–143.

Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. 2015. Core decomposition in large temporal graphs. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 649–658.

Gai Xiaoping, Ruan Mengyu, Zhang Hong, Wu Ping, Ren Ruijun, and Gao Feng. 2021. Construction Technology of Knowledge Graph and its Application in Power Grid. In *E3S Web of Conferences*, Vol. 256. EDP Sciences, 01039.

Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2021. On querying historical k-cores. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2033–2045.

Yikai Zhang and Jeffrey Xu Yu. 2019a. Unboundedness and efficiency of truss maintenance in evolving graphs. In *Proceedings of the 2019 International Conference on Management of Data*. 1024–1041.

Yikai Zhang and Jeffrey Xu Yu. 2019b. Unboundedness and efficiency of truss maintenance in evolving graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 1024–1041. https://doi.org/10.1145/3299869.3300082

Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2017. A fast order-based approach for core maintenance. In *Proceedings - International Conference on Data Engineering*. 337–348. https://doi.org/10.1109/ICDE.2017.93 arXiv:1606.00200

Wei Zhou, Hong Huang, Qiang-Sheng Hua, Dongxiao Yu, Hai Jin, and Xiaoming Fu. 2021. Core decomposition and maintenance in weighted graph. *World Wide Web* 24, 2 (2021), 541–561.