UNIVERSITY OF CALIFORNIA
RIVERSIDE

Data Structures for Efficient Parallel Graph Processing

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Science

by

Xinjian Jiang

June 2026

Thesis Committee:
Professor Yihan Sun, Chairperson
Professor Yan Gu
Professor Zhijia Zhao

The Thesis of Xinjian Jiang is approved:

Committee Chairperson

University of California, Riverside

ACKNOWLEDGMENTS

I am sincerely grateful to my advisors, Yihan Sun and Yan Gu; without their support, I would not be where I am today.

ABSTRACT OF THE THESIS

Data Structures for Efficient Parallel Graph Processing

by

Xinjian Jiang

Master of Science, Graduate Program in Computer Science
University of California, Riverside, June 2026
Professor Yihan Sun, Chairperson

We present the Toggle Tree, a parallel data structure for frontier-based graph algorithms. A Toggle Tree is used to represent a subset of vertices using hierarchical bit vectors, enabling efficient parallel updates, traversal, and set-wise reductions without global packing. It provides an IndexSet interface that unifies a wide range of parallel graph algorithms under a common abstraction of frontiers and active sets, and an IndexMap interface that also incorporates priority-queue semantics. Specifically, the Toggle Tree aims to combine the advantages of traditional sparse and dense frontier representations, offering both provable theoretical guarantees and high efficiency in practice.

Using Toggle Trees, we implement five fundamental graph algorithms, including breadth-first search (BFS), k-core, degree-based graph coloring, single-source shortest paths using Bellman-Ford, and weighted-BFS (wBFS). Across a diverse set of large real-world graphs and the five graph problems, our implementations consistently outperform state-of-the-art baselines, including GBBS and PASGAL, achieving multi-fold speedups. Our code is open-sourced.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

With the rapid growth of graph data, analyzing and processing a graph in parallel has become increasingly important. In such tasks, a critical and unavoidable step is to efficiently maintain a subset of vertices. For instance, many algorithms maintain a *frontier* set containing the vertices to be processed in a given round. Depending on different algorithmic techniques, other dynamic vertex subsets may be needed, such as active/inactive vertices (vertices that have or have not been processed) or candidate sets (vertices ready for processing).

In many graph algorithms, efficiently maintaining such a vertex subset is critical to performance. The minimum requirements for these data structures usually include dynamic updates via insertions and deletions, and a *for-each* operation to process all vertices in it. Some algorithms require operations beyond basic set semantics, such as a priority queue interface. While maintaining a vertex subset seems simple, it is, however, highly non-trivial under the parallel setting. Due to inherent graph irregularity, a major challenge is

that these vertex subsets are highly dynamic, with their sizes fluctuating drastically during execution. For updates, the vertex subset must handle rapid, *concurrent* modifications, ideally close to constant time per operation. For the *for-each* operation, the vertex subset must remain efficient across all scales of the subset size—from a handful of vertices to nearly all vertices—ideally with access time proportional to the subset size s. Sequentially, simple array-based structures like FIFO queues (for vertex tracking) and binary heaps (for priority queries) suffice. However, directly using them in parallel incurs significant overhead due to the heavy contention (e.g., on queue ends or tree roots).

There have been several approaches in the literature, each of which has its limitations. They can be categorized as the *sparse* form or the *dense* form. The sparse representation still keeps an array of elements, each representing a vertex [37, 13, 14, 42, 15, 16], supporting straightforward *for-each* operation. However, for parallel updates, they may either use another round of edge scanning to insert elements synchronously [37, 13, 14] or leave empty slots to enable concurrent insertions [42, 15, 16], resulting in extra constant-factor overhead in performance and/or more space (we introduce existing techniques with more details in Section 4.5). An alternative is the *dense* representation, which simply stores n boolean flags, each representing whether vertex i is in the set. The maintenance overhead is seemingly minimal, but then a *for-each* needs $\Theta(n)$ work to loop over the entire array, disregarding the set size. As a result, the dense form is only used when the set size is $\Theta(n)$. Alongside the structural switching overhead, this limits the algorithm to leverage the benefit of dense mode for only a few rounds, or entirely precludes it if the frontier remains moderate throughout the execution.

In this paper, we overcome these challenges by proposing a simple yet highly efficient data structure: the *Toggle Tree*. It combines the advantages of both the sparse and dense modes. Each vertex only stores a bit representing its existence (as in the dense mode). Then, the *Toggle Tree* has hierarchical bitmasks built on top of the leaves. Each bit in the upper level represents the aggregation (logical OR) of w bits in the lower level, where w is the word width. When traversing the set (e.g., a *for-each*), empty subtrees can be skipped, with cost $\tilde{O}(s)$ for subset size s (close to the efficiency in the sparse mode). For updates, concurrent updates can be applied by simple atomic bit operations, enabling high space efficiency ($\Theta(n)$ bit or $\Theta(n/w)$ words in total for a Toggle Tree). As a stand-alone data structure, the Toggle Tree also eliminates the need of switching between the sparse and dense representations.

For generality, we refer to the abstract data type as an **IndexSet**, since its interface broadly applies to a subset of a universe of n indexes, denoted by integers from 0 to $n-1$. We can also augment the Toggle Tree to extend an IndexSet to an IndexMap, where each index is associated with a value, and to support a priority-queue-like interface. In particular, we introduce an auxiliary tournament tree that maintains subtree minima under concurrent monotone updates and supports batched extraction of the current minimum, providing a simple and efficient alternative to parallel priority queues for weighted graph traversals.

For the most typical usage of vertex subsets, where the algorithm processes a *frontier* of vertices in each round, we theoretically show that if a parallel graph algorithm finishes in ρ rounds, the amortized cost per insertion, deletion, or traversal step in Toggle Tree is bounded by $O(\frac{\log \rho}{\log \log n})$, assuming all vertices are touched. Even in the worst case

when $\rho = n$, the overhead is small; however, this indicates that only one vertex is processed per round with no parallelism. Usually, it is preferable to run these algorithms in parallel when $\rho = \text{polylog}(n)$, and in these cases, Toggle Tree operations are asymptotically optimal.

In practice, we implemented five fundamental parallel graph algorithms using *Toggle Trees* to demonstrate the efficiency and usability, including BFS, single-source shortest paths (SSSP) using Bellman-Ford, k-core, degree-based graph coloring, and weighted-BFS (wBFS). For all of them, we use existing algorithmic techniques and only replace the vertex subset with Toggle Trees. Compared to two graph libraries, GBBS [13] and PASGAL [15], our algorithms are on average 1.8× faster than the best baseline. Across 125 tested algorithm-graph instances, our solution outperforms the strongest baseline in 124 of them, achieving at least a 2× speedup in over a third of the test instances. Our code is publicly available at [38].

Chapter 2

Related Work

This chapter reviews prior work on shared-memory parallel graph processing, with an emphasis on the data-structural problem studied in this thesis: maintaining dynamic subsets of vertices under parallel updates and efficient traversal. The history is broad, spanning theoretical models, graph-processing systems, frontier representations, priority structures, and individual graph algorithms. Our goal is not to give a complete survey of every parallel graph algorithm, but to identify the line of work that leads to the vertex subset abstraction used throughout this thesis. In particular, many previous systems and algorithms already provide efficient ways to express graph computations, while the bottleneck addressed here is the low-level representation of active vertices and frontiers.

2.1 Shared-Memory Parallel Graph Algorithms

Parallel graph algorithms have long been studied under idealized shared-memory models. Early theoretical work often used variants of PRAM, where many processors access

a shared address space and the complexity of an algorithm is measured by parallel time and total operations. These models made it possible to state clean goals: achieve work close to the best sequential algorithm while reducing the critical path to polylogarithmic time. Although real machines differ substantially from PRAM, this theoretical viewpoint remains important because it separates the amount of total computation from the amount of available parallelism.

Modern multicore algorithm analysis commonly uses the work-span model [8, 7]. The work of a computation is the total number of operations, while the span is the length of the longest dependency chain. Together with work-stealing schedulers [1, 8], this model gives a useful bridge between theory and practice: a computation with work W and span S can run in roughly $W/P + S$ time on P processors, up to scheduler overheads. This formulation is especially natural for shared-memory graph algorithms implemented using nested parallelism, parallel loops, and divide-and-conquer primitives. The same model is used by systems and libraries such as ParlayLib [6], which provide high-level parallel primitives while retaining predictable cost bounds.

Graphs make this setting difficult. Unlike dense linear-algebra kernels or sorting routines, graph algorithms operate over irregular neighborhoods, skewed degree distributions, and data-dependent control flow. The amount of work generated by a vertex can vary from constant to millions of incident edges. Memory accesses are often indirect and hard to prefetch. On power-law graphs, a small number of high-degree vertices can dominate the work in a round, while on road networks and mesh-like graphs, the graph diameter can be large and the available parallelism per round can be limited. Thus, an algorithm that is

work-efficient in an abstract model may still perform poorly if the implementation creates excessive synchronization, poor locality, or too many small parallel tasks.

A recurring idea in shared-memory graph algorithms is to expose parallelism by processing sets of currently active vertices. Instead of assigning a static portion of the graph to each processor, many algorithms proceed in rounds. In each round they process a frontier, generate updates to neighboring vertices, and form the next frontier. This pattern appears in breadth-first search, Bellman-Ford style shortest paths, peeling algorithms, graph coloring, connectivity, matching, and many other problems. The abstraction is powerful because the frontier captures exactly the vertices whose state may change in the current step, allowing the algorithm to avoid scanning the entire graph on every round. At the same time, it introduces a basic data-structural requirement: the implementation must maintain and traverse changing vertex subsets efficiently.

This thesis follows this shared-memory, frontier-based line of work. The Toggle Tree is not a new parallel computation model, nor is it a replacement for existing graph algorithms. It is designed for the lower-level role that many graph algorithms implicitly require: representing a subset of vertex identifiers, supporting concurrent insertions and deletions, testing membership, reducing over active vertices, and traversing the active set with low overhead across a wide range of subset sizes. The rest of this chapter reviews the systems and algorithmic techniques that motivate this requirement.

2.2 Frontier-Based Graph Processing Frameworks

Ligra [37] was a major step in making shared-memory graph processing both simple and fast. Its programming model is built around a *vertexSubset* and two central operations, *vertexMap* and *edgeMap*. A vertexSubset represents the active vertices in a round, vertexMap applies a function to all active vertices, and edgeMap visits edges incident to active vertices to generate future work. Ligra also popularized a sparse-dense representation for frontiers: a small frontier can be stored sparsely as an array of vertex IDs, while a large frontier can be stored densely as a boolean array. This design captures a basic performance tradeoff that appears repeatedly in later systems. Sparse representations make traversal proportional to the number of active vertices, while dense representations make membership tests and bottom-up traversal efficient when a large fraction of vertices are active.

GBBS [13] continued this direction by providing a broad benchmark suite and implementations of many theoretically efficient graph algorithms. It showed that algorithms with good asymptotic guarantees can be competitive on real shared-memory machines when implemented carefully. GBBS relies on efficient parallel primitives, careful graph representations, and frontier-based processing patterns similar in spirit to Ligra. For this thesis, GBBS is important for two reasons. First, it represents a mature baseline for shared-memory graph processing. Second, it demonstrates that many high-performance graph algorithms can be expressed using a small number of common operations on vertex subsets. Improving the implementation of those operations can therefore affect many algorithms at once.

Galois [31] represents a different but related approach. Rather than focusing primarily on bulk-synchronous frontiers, Galois supports irregular parallelism through worklists, optimistic execution, and conflict detection. This style is useful for algorithms whose available work is dynamic and where strict round boundaries may be too restrictive. Nevertheless, Galois computations still rely on maintaining sets or queues of active items. The system perspective is different from Ligra and GBBS, but the underlying pressure is similar: the active workset must be represented in a way that allows many workers to obtain and generate work without excessive synchronization.

GraphIt [50, 49] approaches the problem from the perspective of a domain-specific language and compiler. It separates the algorithm from the scheduling strategy, allowing the programmer or compiler to choose traversal direction, frontier representation, and other optimizations. This separation is useful because many graph algorithms have a simple mathematical description but many possible implementation schedules. For example, the same traversal can be implemented in a top-down sparse style, a bottom-up dense style, or a hybrid style. GraphIt's schedule language makes these choices explicit, which reinforces the point that frontier representation is a central implementation decision rather than a minor detail.

Julienne [12] and later shared-memory graph libraries such as PASGAL [15] further specialize this frontier-based view. Julienne was designed for work-efficient parallel graph algorithms using bucketing and priority-based processing. PASGAL improves several graph algorithms through techniques such as vertical granularity control and hash-bag based frontier maintenance [42, 15]. These systems are especially close to the setting of

this thesis because they emphasize graph algorithms whose performance depends heavily on active-set maintenance. In our experiments, PASGAL provides strong baselines for several applications, and its hash bag is one of the most relevant existing data structures for comparison.

GPU graph frameworks also use frontier-based processing, although the hardware constraints are different. Merrill et al. developed scalable GPU graph traversal techniques [27, 28], Gunrock provides a high-performance GPU graph analytics library [43], and GraphBLAST uses a linear-algebraic GraphBLAS interface on GPUs [47]. These works are outside the main scope of this thesis because Toggle Tree targets CPU shared memory. However, they show that the frontier abstraction is not specific to CPU graph processing. Across architectures, high-performance graph algorithms repeatedly need a representation of the active vertices and operations to transform one frontier into the next.

The common lesson from these systems is that frontier-based graph processing is a successful abstraction, but it does not by itself solve the low-level representation problem. Systems must still decide how to store a frontier, how to insert into it concurrently, how to test membership, how to traverse it, and when to switch between representations. The Toggle Tree is complementary to these systems. It does not change the programming model; rather, it provides a data structure that can be used underneath frontier-based algorithms and libraries.

2.3 Representing and Maintaining Vertex Subsets

The representation of a vertex subset is one of the most persistent engineering choices in shared-memory graph processing. At first glance, a vertex subset seems simple: it is just a set of integers from 0 to $n - 1$. In parallel graph algorithms, however, this set is updated by many workers, traversed repeatedly, and used for membership tests and reductions. Its size may change by orders of magnitude across rounds. A representation that is ideal for a frontier with ten vertices may be inefficient for a frontier with ten million vertices, and a representation that supports cheap insertion may be expensive to traverse.

The most common sparse representation stores the active vertices in an array. This is simple and efficient for traversal: a parallel loop over an array of length s processes exactly the s active vertices. Sparse arrays also have good spatial locality and low memory overhead when the frontier is small. They are therefore a natural choice for top-down BFS, Bellman-Ford frontiers, coloring candidates, and many other algorithms. The difficulty is concurrent construction. If many vertices are discovered in parallel, inserting them into a compact array usually requires either a prefix-sum based packing step, per-worker buffers followed by concatenation, or atomic increments into a shared output region. The first option is bulk-synchronous and often requires an additional pass over candidate vertices. The second option adds buffering and merging overhead. The third option can create contention and may not preserve a compact representation without over-allocation.

Dense representations take the opposite approach. They allocate one flag per vertex, often as a byte array or bitmap, and mark whether each vertex is active. Insertions, deletions, and membership tests become direct indexed operations. Dense representations

are especially attractive when the frontier is large or when the algorithm must frequently test whether a vertex is in the frontier. For example, bottom-up BFS checks whether an unvisited vertex has a neighbor in the current frontier; dense membership makes this operation cheap. The drawback is traversal. To apply a function to all active vertices, a dense representation generally scans all n flags, even if only $s \ll n$ vertices are active. Thus, dense traversal has $\Theta(n)$ work independent of the actual frontier size.

Ligra's sparse-dense switching strategy [37] addresses this tradeoff by using sparse arrays for small frontiers and dense bitmaps or boolean arrays for large frontiers. GBBS and other systems inherit similar ideas [13]. This is often effective because many graph algorithms naturally have phases where the frontier is clearly small or clearly large. In BFS on small-world graphs, for instance, the frontier can expand rapidly from the source and later shrink. Switching allows the algorithm to use sparse traversal when the frontier is small and dense membership when the frontier is large. However, switching is not free. Converting between sparse and dense forms requires additional work, and the system must choose thresholds that depend on graph structure, machine characteristics, and the algorithm. Furthermore, many frontiers are neither tiny nor close to the full vertex set. For such middle-sized sets, sparse traversal is attractive, but dense membership and update semantics may also be useful.

Another strategy is to keep a sparse-like frontier but relax compactness. Instead of constructing a perfectly packed array, an algorithm can reserve more space than needed and allow workers to write into available slots. This reduces synchronization but introduces empty entries, wasted space, and extra traversal cost. The hash bag used in PASGAL and

related work is a representative example [42, 15]. It supports concurrent insertion into a bag-like structure and is useful for frontier maintenance in several algorithms. Hashing reduces contention compared with a single shared queue and avoids a global packing step, but it also introduces constant-factor overheads: hash computation, probing, empty slots, possible resizing, and less predictable memory access. These costs are often acceptable, and the hash bag is a strong practical baseline. Nevertheless, they motivate the search for a simpler structure when the universe is known to be the set of vertex IDs.

The known-universe property is important. In graph algorithms, the elements of a frontier are not arbitrary keys; they are integers in a fixed range. This makes bit-level representations possible. A bitmap uses one bit per vertex, giving high space efficiency and direct indexed updates. Atomic bit operations can support concurrent insertion and deletion with low overhead. The problem is that a flat bitmap alone does not provide sparse traversal. Scanning all words in the bitmap costs $\Theta(n/w)$ even if the frontier contains only a few elements. For this reason, flat bitmaps are usually treated as dense representations and are used mainly when the active set is large.

Hierarchical summaries address this limitation by placing additional levels above the bitmap. Each internal bit summarizes whether a block of lower-level bits is nonempty. Traversal can then skip entire empty subtrees and descend only into regions that contain active vertices. This idea is related in spirit to word-level bit tricks, hierarchical bitsets, and tree-based priority structures. It is also reminiscent of classical integer-set data structures such as van Emde Boas trees [40] and related variants [46], although the goal in this thesis is different. The Toggle Tree is not designed to improve the best known theoretical bound for general predecessor or priority operations. Instead, it exploits the fixed universe and the operations needed by graph algorithms to obtain a practical combination of dense-style updates and sparse-style traversal.

The IndexSet abstraction in this thesis is therefore best understood as a vertex-subset data type specialized for shared-memory graph processing. It supports Insert, Remove, Contains, ForEach, Reduce, and Clear. These operations match the needs of many frontier-based algorithms. The structure uses a bitmap at the leaves, so membership and updates are direct and compact. It also maintains upper-level summary bits, so traversal and reductions can skip empty regions. This differs from a sparse array because the set is not stored as a packed list of active vertices. It differs from a dense bitmap because traversal is not forced to scan the entire vertex universe. It differs from a hash bag because the position of an element is determined by its vertex ID rather than by hashing.

The distinction matters most when frontier sizes fluctuate. If a graph algorithm always maintained very small frontiers, a simple sparse array would be hard to beat. If it always maintained frontiers close to the full vertex set, a dense bitmap would be enough.

Real graph algorithms often move between these regimes. In BFS, the frontier may grow and shrink across levels. In k-core decomposition, a peeling round may expose a large or small set of newly peelable vertices depending on graph structure. In coloring, the set of ready vertices is determined by a priority DAG and may be irregular. In Bellman-Ford style relaxation, repeated distance improvements may generate frontiers with little predictable structure. The Toggle Tree is designed for this variability.

There is also a semantic difference between maintaining a frontier as a bag and maintaining it as a set. Many graph algorithms want set semantics: a vertex should appear at most once in the next frontier, even if many neighbors try to activate it. This can be enforced with atomic compare-and-swap on a visited flag, with duplicate filtering after construction, or with a data structure whose insert operation is naturally idempotent. Bit-based insertion has this idempotent property. If several processors set the same bit, the final state is still just one active vertex. This is convenient for graph algorithms where duplicate work can otherwise become expensive.

The cost of traversing a hierarchical bitmap depends on both the number of active elements and the distribution of those elements. If active vertices are clustered, large empty regions can be skipped. If active vertices are spread out, the traversal may visit more internal nodes, but it still avoids scanning blocks with no active bits. For graph applications, this is often a good tradeoff because the structure avoids a global representation switch while retaining simple atomic updates. The theoretical analysis in later chapters formalizes this cost in terms of the number of rounds and touched vertices. The related-work point is that Toggle Tree occupies a middle ground between the dominant sparse and dense representations used in prior graph systems.

2.4 Parallel Priority Structures for Ordered Graph Algorithms

Some graph algorithms require more than an unordered frontier. Shortest-path algorithms, weighted graph traversals, and priority-driven peeling algorithms often need to process vertices according to a key. Sequentially, this role is typically played by a priority queue. Dijkstra's algorithm, for example, repeatedly extracts the unsettled vertex with minimum tentative distance and relaxes its outgoing edges. In shared memory, however, a strict priority queue is difficult to parallelize. The minimum element is a global bottleneck, decrease-key operations can be frequent, and concurrent updates to a heap or tree can create synchronization hot spots.

Many parallel priority queues relax the exact sequential semantics in order to increase throughput. Randomized priority queues [35], relaxed concurrent priority queues, multi-queue designs, and other approaches reduce contention by allowing operations to access different parts of the data structure. These structures are useful in general concurrent settings, but graph algorithms often have more specific requirements. They may not need to extract exactly one minimum vertex at a time. Instead, they can process a batch of vertices with the same or similar priority. This observation underlies several shortest-path algorithms and motivates bucket-based designs.

Δ-stepping [30] is a classic example. It groups vertices by distance ranges and processes buckets rather than a single strict minimum at every step. This creates more parallelism than Dijkstra's algorithm while preserving correctness under appropriate relaxation rules. Later work developed efficient stepping algorithms and implementations for parallel

shortest paths [16], as well as other priority-based techniques for graph algorithms [41, 19]. The broad pattern is that ordered graph algorithms often become practical in parallel when the strict priority queue interface is replaced by a batched or relaxed interface.

The IndexMap abstraction in this thesis follows this spirit. It extends IndexSet by associating a value with each active index and supporting a batched ExtractMin operation that extracts all elements with the current minimum value. The implementation uses an auxiliary tournament-tree-like structure to maintain subtree minima. Tournament trees and related tree summaries have appeared in prior work on parallel priority structures [16, 41, 19]. The contribution here is not the general idea of a tournament tree, but its integration with the Toggle Tree representation and its use as a lightweight IndexMap for graph applications such as weighted BFS.

This distinction is important because a general-purpose concurrent priority queue must support many operation sequences and key patterns. A graph-specific IndexMap can exploit the structure of frontier-based algorithms: updates are often monotone, extraction is batched, and the universe of elements is fixed. These restrictions allow a simpler design than a fully general priority queue. They also fit naturally with shared-memory graph processing, where a round processes many active vertices in parallel before moving to the next set of candidates.

2.5 Parallel Graph Algorithms Using Frontiers

The applications in this thesis are standard graph problems. Our work does not claim new algorithms for these problems. Instead, it uses them to demonstrate that a better

vertex-subset representation can improve existing frontier-based algorithms. This section reviews the relevant algorithmic background and explains why each problem depends on maintaining active vertex sets.

2.5.1 Bellman-Ford

Single-source shortest paths is one of the central problems in graph algorithms and one of the most challenging to parallelize efficiently. Bellman-Ford is naturally parallel at the level of edge relaxations. In each round, vertices whose distances changed can relax their outgoing edges, generating a new frontier of vertices whose distances may improve further. This frontier-based version is simple and maps well to shared memory, but the number of rounds can be large. It is nevertheless a useful baseline and a common component in more advanced shortest-path methods.

Dijkstra's algorithm has better sequential behavior on graphs with nonnegative weights, but its strict extract-min order limits parallelism. Parallel shortest-path algorithms therefore often trade exact ordering for batched or relaxed processing. Δ-stepping [30] groups tentative distances into buckets, exposing parallelism within a bucket while controlling the amount of redundant relaxation. Efficient parallel implementations refine these ideas using careful bucket management, priority updates, and graph-specific optimizations [16, 11].

From the perspective of this thesis, SSSP algorithms show why both IndexSet and IndexMap are useful. Bellman-Ford style algorithms need an unordered frontier of vertices to relax. More ordered variants need a structure that can maintain tentative distances and extract a minimum or near-minimum set. Both cases rely on dynamically updated subsets

of vertices. The Toggle Tree does not replace shortest-path algorithm design; instead, it supplies a compact and parallel representation for the active sets that these algorithms already use.

2.5.2 k-core Decomposition

The k-core decomposition assigns each vertex a coreness value and is widely used to analyze the dense structure of large graphs. The classical sequential approach repeatedly removes vertices whose current degree is below a threshold, updating the induced degrees of their neighbors. This process is often described as peeling. It is linear-work sequentially, but parallelizing it efficiently is subtle because many degree decrements happen concurrently and newly peelable vertices must be discovered dynamically.

Parallel k-core algorithms commonly process a frontier of vertices that become peelable at the current threshold. When a vertex is peeled, the degrees of its active neighbors are decremented. If a neighbor's induced degree reaches the current threshold, it is inserted into the next frontier. The algorithm also maintains an active set of vertices that have not yet been peeled. Thus, k-core decomposition uses both a current frontier and a global active set, and both are updated across many rounds and subrounds.

Prior work has developed practical multicore k-core algorithms [23, 12, 26]. Julienne's bucketing framework [12] and later PASGAL implementations [15, 26] provide strong baselines. These algorithms focus on reducing redundant work, handling high-degree vertices, batching updates, and preserving work efficiency. The frontier representation is one of the implementation details that affects performance, especially because a vertex can become peelable due to updates from many neighbors.

The Toggle Tree version of k-core decomposition keeps the algorithmic structure of prior work. It still peels vertices in parallel, decrements induced degrees atomically, and forms new frontiers when vertices become ready. The difference is that active and frontier sets are represented by IndexSet instances. This allows idempotent insertion through bit updates, direct membership tests, and traversal that avoids scanning the entire graph when the active subset is sparse.

2.5.3 Breadth-First Search

Breadth-first search is the canonical frontier-based graph traversal. Starting from a source vertex, BFS processes vertices in nondecreasing hop distance. The current frontier contains all vertices at distance i, and the next frontier contains undiscovered neighbors at distance $i + 1$. This level-synchronous structure exposes parallelism because all vertices in the same frontier can be processed independently, apart from races when multiple parents discover the same child.

Parallel BFS has received extensive attention because it is both fundamental and a useful benchmark for graph-processing systems. In the top-down approach, the algorithm scans outgoing edges from vertices in the current frontier. This is efficient when the frontier is small, since work is proportional to the edges incident to active vertices. However, when the frontier becomes large, many edges may lead to already visited vertices, and the top-down method can waste work. Direction-optimizing BFS [3, 4] addresses this by switching to a bottom-up strategy. In bottom-up mode, each unvisited vertex checks whether any neighbor is in the current frontier. This can reduce edge examinations on small-world graphs with large frontiers.

The top-down and bottom-up modes require different frontier operations. Top-down BFS benefits from sparse traversal over the current frontier and efficient construction of the next frontier. Bottom-up BFS benefits from dense membership tests on the current frontier and traversal over the active set of unvisited vertices. Ligra [37], GBBS [13], and other systems implement BFS using sparse-dense switching to support these modes. GPU systems also rely heavily on frontier management for BFS [27, 28, 43].

The Toggle Tree is relevant to BFS because it can represent both the current frontier and the active set with the same data structure. Membership tests are direct bit checks, insertions into the next frontier are atomic bit updates, and traversal can skip empty regions. This does not change the BFS algorithm or the direction-optimization heuristic. It changes the cost of maintaining the vertex subsets used by the algorithm.

2.5.4 Graph Coloring

Graph coloring assigns colors to vertices so that adjacent vertices receive different colors. Many variants are NP-hard if the goal is to minimize the number of colors, so practical graph coloring algorithms often use heuristics. Sequential greedy coloring chooses an ordering of vertices and assigns each vertex the smallest available color. The quality and performance of the result depend heavily on the ordering. Classical heuristics include smallest-last and degree-based orderings [45].

Parallel graph coloring usually exposes concurrency by identifying vertices that can be colored simultaneously. Jones-Plassmann style algorithms [22] assign priorities to vertices and color local maxima in parallel. Other work studies ordering heuristics and practical improvements for parallel coloring [21]. Recent systems and graph-processing

libraries include coloring implementations as standard benchmarks [13], and specialized heuristic and local-search approaches continue to be studied [36, 17].

The coloring algorithm used in this thesis follows a priority-based frontier pattern. Each vertex maintains a count of higher-priority uncolored neighbors. Vertices whose count becomes zero are ready to be colored and form the current frontier. After a frontier is colored, the algorithm decrements the counters of lower-priority neighbors and inserts newly ready vertices into the next frontier. This is structurally similar to a topological traversal of a priority DAG. The active data structure must therefore support concurrent insertions into the next frontier, traversal over ready vertices, and clearing or swapping frontiers between rounds.

As with the other applications, the Toggle Tree does not introduce a new coloring heuristic. Its role is to reduce the overhead of maintaining the ready set. This is useful because coloring rounds can vary significantly in size depending on the graph and priority order. Some rounds contain many independent vertices, while others contain only a small number of newly ready vertices. A representation that handles both cases without switching is therefore attractive.

2.5.5 Weighted-BFS

Weighted BFS can be viewed as a bridge between ordinary BFS and general single-source shortest paths. For unweighted graphs, all edges advance the distance by one, so a simple FIFO level structure is enough. For graphs with nonnegative integer weights, the algorithm must process vertices in increasing tentative distance, but it may still be possible

to process many vertices with the same distance together. Thus, the frontier is no longer simply the next hop layer; it is a minimum-distance layer.

This batched interpretation is well matched to a priority frontier. Instead of extracting one vertex at a time, a weighted BFS implementation can repeatedly extract all active vertices with the minimum tentative distance, relax their outgoing edges, and update the priorities of affected neighbors. The active set contains unsettled vertices with finite tentative distances. The key operation is therefore not just traversal, but extraction of the current minimum layer.

Prior shortest-path work using buckets, stepping, and tournament-like structures provides the broader background for this view [30, 16, 41, 19]. In this thesis, weighted BFS is used to demonstrate the IndexMap version of Toggle Tree. The data structure maintains active vertices and their tentative distances, supports monotone decrease operations, and extracts all vertices at the current minimum distance. The algorithmic idea is standard; the contribution is that the same hierarchical representation used for unordered vertex subsets can be extended to ordered frontier extraction.

2.6 Positioning of This Thesis

The work reviewed above shows that shared-memory graph processing has matured along several dimensions. There are well-established parallel computation models and schedulers [8, 1, 7]. There are high-performance graph frameworks and libraries that expose frontier-based abstractions [37, 31, 50, 49, 13, 15]. There are strong algorithms for BFS, shortest paths, k-core decomposition, graph coloring, and related problems [3, 30, 12, 26, 22]. The remaining issue addressed in this thesis is narrower but pervasive: how to maintain dynamic vertex subsets efficiently under shared-memory parallelism.

Existing approaches each cover part of the design space. Sparse frontiers offer efficient traversal but require additional machinery for concurrent construction. Dense bitmaps offer simple updates and membership tests but require full scans for traversal. Sparse-dense switching works well in many systems but introduces conversion costs and threshold decisions. Hash bags support concurrent insertion but pay hashing, probing, and space overheads. General priority queues provide ordered operations but are often heavier than what frontier-based graph algorithms require.

The Toggle Tree is designed to fill the gap between these alternatives. For unordered vertex subsets, IndexSet combines bit-level updates with hierarchical summaries for traversal and reduction. For ordered active sets, IndexMap augments the structure with tournament-style minima and batched extraction. The result is a lightweight data structure aimed specifically at graph algorithms over a fixed vertex universe. It is complementary

to previous graph systems and algorithms: those systems define how computations are expressed and scheduled, while Toggle Tree improves the cost of a common data-structural component used inside them.

This positioning also explains the experimental methodology of the thesis. The applications are not chosen because they are new problems, but because they represent common frontier patterns: unweighted traversal, weighted traversal, repeated relaxation, peeling, and priority-based coloring. By applying Toggle Tree to these algorithms, we evaluate whether a single vertex-subset representation can improve performance across different graph workloads without changing the high-level algorithmic ideas.

Chapter 3

Preliminaries

Parallel Computational Model. We use the work-span model in the classical multi-threaded setting with binary fork-join [8, 7]. We assume a set of threads sharing a common memory. A process can `fork` two child threads to execute in parallel. When both children complete, the parent process continues. The ***work*** of an algorithm is the total number of instructions. The ***span*** of an algorithm is the length of the longest dependency chain among operations. A computation with work W and span S can be executed by a randomized work-stealing scheduler [1, 8, 20] in time $W/P + O(S)$ with high probability in W. We use hardware-supported atomic primitives, including atomic bitwise operations (`fetch_and_or` and `fetch_and_and`) for lock-free updates to the Toggle Tree (Section 4.2).

The analysis of the graph applications using the Toggle Tree (in Chapter 5) assumes additional unit-cost atomic operations, including `write_min` for distance relaxation, and atomic counter decrements in k-core and coloring. We make the same assumption when analyzing the corresponding algorithms, although the Toggle Tree itself does not use them.

Notations. We consider a graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, stored in compressed sparse row (CSR) format. For a vertex v, $N(v)$ denotes its neighbor set and $v) = |N(v)|$ its degree. We use w to denote the machine word size (typically 64). In asymptotic notation, we use $\log_a b$ as shorthand for $1 + \log_a (a + b)$ for simplicity.

Frontier-Based Graph Algorithms. Many parallel graph algorithms follow a *frontier-based* paradigm. The algorithm maintains an *active set* $\mathcal{A}$ of vertices that have not been finalized and a *frontier* $\mathcal{F}$ of vertices that are processed in the current round. In each round, the algorithm processes every vertex in $\mathcal{F}$ (e.g., relaxing outgoing edges), which may generate a new frontier $\mathcal{F}'$ for the next round and remove processed vertices from $\mathcal{A}$. The algorithm terminates when the frontier or the active set becomes empty. We use examples of BFS, Bellman-Ford, k-core, graph coloring, weighted-BFS, all discussed in Chapter 5.

Chapter 4

The Toggle Tree Design

In this section, we describe the design of the Toggle Tree, which implements the interfaces for IndexSet and IndexMap. IndexSet and IndexMap are similar to standard sets and maps, except that keys are integers from 0 to $n-1$. Well-known data structures exist for these data types, such as the van Emde Boas (vEB) tree [40, 19] and its variants [46]. In contrast to them, the primary goal of the Toggle Tree is to achieve exceptionally fast practical performance while preserving good theoretical bounds for graph applications. In the following, we review the interface of the Toggle Tree in Section 4.1, followed by its algorithms in Section 4.2 and analysis in Section 4.4. Finally, we compare our design with existing work in Section 4.5.

4.1 Interface

The IndexSet Interface. Here we consider the standard interface for sets, focusing primarily on the operations that are often used in graph applications, listed as follows.

- Init(n, b): initialize an empty ($b = 0$) or a full ($b = 1$) set over a universe of size n.
- Empty(): return whether $S = \emptyset$.
- Contains($index$): return whether $index \in S$.
- Insert($index$): (atomically) insert $index$ into S if $index \notin S$; return true when successful.
- Remove($index$): (atomically) delete $index$ from S if $index \in S$; return true when successful.
- ForEach$(f(\cdot))$: apply function $f(index)$ to each element $index \in S$.
- Reduce$(f(\cdot), \oplus, \perp)$: compute $\oplus_{index \in S} f(index)$ in parallel with associative operator $\oplus$.
- Clear(): delete all elements in the set.

These functions capture the common needs of many parallel graph algorithms. To help understand the interface, we show an example of parallel BFS in Algorithm 1. The algorithm maintains three sets: all active (unprocessed) vertices $\mathcal{A}$, the frontier $\mathcal{F}$, and the next frontier $\mathcal{F}'$. In each round, the algorithm applies ForEach on $\mathcal{F}$ to visit the neighbors of each $v \in \mathcal{F}$. If the neighbor u is in $\mathcal{A}$, meaning that u is visited for the first time, then we insert u into $\mathcal{F}'$ and Remove u from $\mathcal{A}$, both performed atomically.

The IndexMap Interface. A natural extension of the set interface is the "map" interface where each element is associated with a value. This is most common in distance-related graph applications, where each vertex maintains a tentative distance, and the algorithm

Algorithm 1: Parallel BFS using the IndexSet interface

```
1  Note: a detailed version with the directional optimization is shown in Algorithm 6
2  Function BFS(G = (V, E), source ∈ V):
3      A ← IndexSet.Init(G.n, 1), A.Remove(source)        // The active (unprocessed) vertices
4      F ← IndexSet.Init(G.n, 0), F.Insert(source)        // Initialize the current frontier
5      F' ← IndexSet.Init(G.n, 0)                         // Initialize the next frontier
6      while ¬ F.Empty() do
7          F.ForEach v ∈ F do
8              ParallelForEach u ∈ G.N(v) do
9                  if A.Contains(u) then                  // The first time to visit u
10                     F'.Insert(u)                       // Add to the next frontier
11                     A.Remove(u)                        // Remove from the active set
12         F.Clear()
13         swap F and F'
```

often needs to extract all vertices with the smallest value (distance). Consequently, IndexMap also incorporates priority-queue operations. Without loss of generality, we assume a min-priority queue throughout the paper based on a total ordering with the following interface.

- Init(): create a Toggle Tree, initialized as all elements with value $+\infty$.
- DecVal($index, val$): atomically update the value of element $index$ to val if val is smaller.
- ExtractMin($f(\cdot)$): find all elements with the minimum value, apply function $f(index)$ to them, and remove them, in parallel.

Parallelism and Concurrency. We consider the batch-concurrent setting: all batch operations (e.g., Init, ForEach, Reduce) are in parallel, and must run in isolation; all point updates (e.g., Insert, Remove, DecVal) are atomic, and can only be concurrent with the same type of operations. This is common for many parallel graph algorithms, which alternate between modification (e.g., Insert, Remove, DecVal) and traversal (e.g., ForEach, Reduce), following the frontier-based paradigm described in Chapter 3.

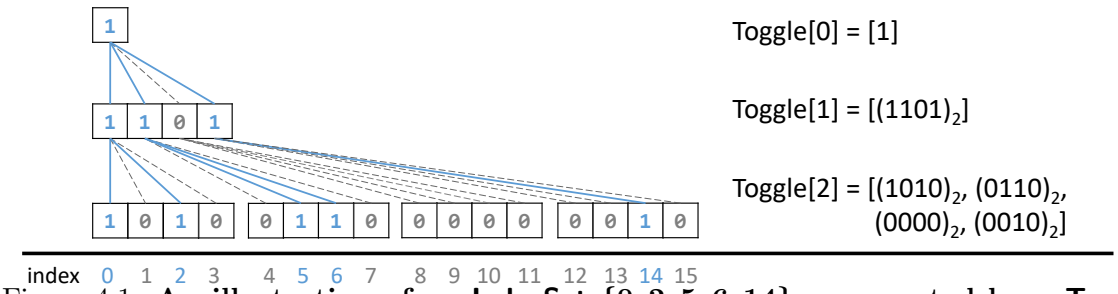

Figure 4.1: **An illustration of an IndexSet $\{0, 2, 5, 6, 14\}$ represented by a Toggle Tree.** Here we use $n = 16$ and $w = 4$.

4.2 Algorithms

The Toggle Tree Layout. The Toggle Tree data structure consists of $L + 1$ levels of bit vectors, where $L = \lfloor \log_w n \rfloor$. The bitvector array at the bottom level has the length of $\lceil n/w \rceil$. We refer to each bit as a *node* in the tree. For the ease of algorithm description, we use $toggle[i][j]$ to represent the j-th bit at the i-th level. At the lowest level L, each $toggle[L][j]$ directly denotes the existence of index j in the set. For the internal nodes $toggle[i][j]$ where $i < L$, it represents the aggregation of w bits at the next level (its children), namely $toggle[i+1][j \cdot w]$ to $toggle[i+1][(j+1) \cdot w - 1]$. If all of them are 0, $toggle[i][j] = 0$, otherwise $toggle[i][j] = 1$. In the code, every group of w bits are treated together as an integer. Reading or modifying each bit can be performed by simple bitwise operations. To do this atomically, we use hardware-supported `fetch_and_or` and `fetch_and_and`. The efficiency of this layout comes from built-in bit operations. In fact, checking the existence, modifying, or extracting a certain child of a node can be performed by $O(1)$ bitwise operations, instead of $O(w)$. For completeness, we list the built-in bitwise functions we use in Section 4.3.2.

In the following, we describe the algorithms in detail and analyze their work and span.

4.2.1 Algorithms for IndexSet

We now introduce algorithms for IndexSet. We present the pseudocode in Algorithm 2.

Empty and Contains($index$). By definition, Empty() simply checks the bit at the root level, while Contains($index$) simply checks the corresponding bit at the leaf level.

Insert($index$) and Remove($index$). Both of them start from $toggle[L][index]$, i.e., the leaf level. We use an atomic function to change the bit accordingly. If successful, this operation is also responsible for updating the ancestors accordingly, going upwards from $toggle[i][j]$ to its parent $toggle[i-1][j/w]$. In both cases, an early termination is possible. For Insert, if the current bit is already 1, upward propagation is unnecessary, as all the ancestors must also be 1. Similarly, for Remove, if a bit remains 1 after this removal, its ancestors will also remain 1, allowing the process to terminate before reaching the root.

ForEach(f). This operation enumerates all elements in the set and applies a user-provided function f to each of them. It will call ForEach$^*(i, j, f)$ as a helper function to process the subtree at $toggle[i][j]$, starting from the root $toggle[0][0]$. If $toggle[i][j]$ is a leaf ($i = L$), then it represents one active element j and the algorithm executes $f(j)$. Otherwise, the algorithm uses a parallel-for to visit all existing children of the current node, i.e., all the 1-bits of its children among the next level, and recursively applies ForEach* to them. Note

Algorithm 2: Algorithms for IndexSet

Data Structure:

$L+1$ levels of bit vectors, where $L = \lfloor \log_w n \rfloor$. For ease of algorithm description, we use $toggle[i][j]$ to represent the j-th bit at level i, where $0 \le i \le L$, and $0 \le j \le w^i$. In the actual implementation, every w bits are stored as an integer.

Notes:

`atomic_set`$(toggle[i][j], b)$ means to set the bit at $toggle[i][j]$ to $b = 0$ or 1 if its original value is not b, atomically. They will be implemented by using hardware-supported atomic operations `fetch_and_or` / `fetch_and_and` on the corresponding integer with a proper bitmask. It returns true if the operation is successful, and false otherwise.

```
1  Function Empty()                                              // checks whether the root is 0
2      return toggle[0][0] = 0
3  Function Contains(index)                                      // checks the corresponding bit at the leaf level
4      return toggle[L][index] = 1
5  Function Insert(index)
6      i ← L, j ← index                                          // start from the index at the leaf level
7      if atomic_set(toggle[i][j], 1) then
8          repeat
9              i ← i − 1, j ← j/w                                // get the parent of toggle[i][j]
10             if toggle[i][j] = 0 then atomic_set(toggle[i][j], 1) else break
11         until i = 0
12         return true
13     else return false                                         // no change made; the element already exists
14 Function Remove(index)
15     i ← L, j ← index                                          // start from the index at the leaf level
16     if atomic_set(toggle[i][j], 0) then
17         repeat
18             i ← i − 1, j ← j/w                                // get the parent of toggle[i][j]
19             if toggle[i][j] has no child with value 1 then atomic_set(toggle[i][j], 0) else break
20         until i = 0
21         return true
22     else return false                                         // no change made; the element did not exist
23 Function ForEach(f)                                           // Apply function f(index) to all indexes in the set
24     ForEach*(0, 0, f) // Use helper function ForEach*
25 Function ForEach*(i, j, f)                                    // Apply function f(index) to toggle[i][j] and all its descendants
26     if i = L then f(j)
27     else                                                      // Recursively process all existing children of toggle[i][j] at the next level
28         ParallelForEach j* s.t. w · j ≤ j* < w · (j + 1), toggle[i + 1][j*] = 1 do
29             ForEach*(i + 1, j*, f)
   // Reduce φ(index) in the current set using associative ⊕ and identity ⊥
30 Function Reduce(φ, ⊕, ⊥)
31     Reduce*(0, 0, φ, ⊕, ⊥) // Use helper function Reduce*
32 Function Reduce*(i, j, φ, ⊕, ⊥)                               // apply Reduce(φ, ⊕, ⊥) to the subtree at toggle[i][j]
33     if i = L then return φ(j)
34     else                                                      // Recursively process all existing children of toggle[i][j] at the next level
35         ParallelForEach j* s.t. w · j ≤ j* < w · (j + 1), toggle[i + 1][j*] = 1 do
36             t[j* − w · j] ← Reduce*(i + 1, j*, φ, ⊕, ⊥)
37         s ← ⊥
38         foreach t[j*] do s ← s ⊕ t[j*]                        // In theory this can be performed by a parallel reduce
39         return s
40 Function Clear()                                              // remove all indexes
41     ForEach*(0, 0, index ↦ Remove(index))
```

that all its children by definition are stored in a word with width w, and we use bitwise operations such as pop-count to enumerate all 1-bits.

Reduce($\phi, \oplus, \perp$). This operation aggregates information over all elements in the set, with an identity value $\perp$, a mapping function ϕ, and an associative operator $\oplus$. Similar to ForEach, it uses a helper function $\mathsf{Reduce}^*(i, j, \dots)$ applied to the subtree at $toggle[i][j]$. For a leaf node, the reduction result is simply $\phi(j)$, in which j is the index of the element. For an internal node, the algorithm first gathers the list of children and allocates a partial array indexed by these children. It then recursively computes the reduction of every child in parallel, and finally combine them using $\oplus$. This abstraction is general enough to facilitate many set-level queries, such as counting elements, computing sums, finding minimum/maximum values, or combining user-defined summaries.

Clear(). This function removes all elements by using $\mathsf{ForEach}^*(0, 0, f)$ with f deleting the element itself from the set.

Taken together, these operations define a minimal yet expressive interface for an IndexSet. In Chapter 5, we will see that they efficiently support a wide range of graph algorithms.

4.2.2 Algorithms for IndexMap

Recall that an IndexMap associates a value for each index in an IndexSet, and supports a priority-queue interface. The implementation of the IndexMap interface maintains three fields for each tree node: a *flag* bit, a *dirty* bit, and a value *value*. For a leaf, *value* is simply the current value of the corresponding element. For an internal node, *value* is the minimum *value* among its active children, or $+\infty$ if no child is active. At a high level, it

is similar to a parallel tournament tree with larger fan-outs. We present the algorithms in Algorithm 3.

DecVal($index, val$). The operation updates the value of element *index* to *val*. It first applies a concurrent `write_min` to the target leaf value. If the value decreases, the leaf becomes active and all ancestors on the leaf-to-root path are marked dirty, which can stop early when the *dirty* bit is already set on the path. Importantly, the update does not immediately recompute all subtree minima. Instead, it only records that the corresponding ancestors may need a repair. This lazy strategy keeps concurrent updates simple and cheap.

ExtractMin(f). This function identifies all elements in the set with the minimum value, applies function f to them, and deletes them from the set. Before extraction, the operation ExtractMin checks whether the root is dirty. If so, it invokes $\mathsf{Repair}^*(0, 0)$, which recursively traverses only dirty branches and restores the correct subtree minima. Once repaired, the root value is the current global minimum *minval*. The extraction procedure then calls $\mathsf{ExtractMin}^*(i, j, minval, f)$ with this value. The recursion descends only into children whose value is at most the threshold, and when it reaches a leaf with *minval*, it applies f to it and marks the leaf inactive. After the recursive calls return, the node recomputes its value from its remaining children.

Algorithm 3: Algorithms for IndexMap

Data Structure:
$(L+1)$ levels of toggles, where $L = \lfloor \log_w n \rfloor$. For ease of algorithm description, we use $toggle[i][j]$ to represent the j-th node at level i, where $0 \le i \le L$, and $0 \le j \le w^i$. Each toggle has a *flag* bit to indicate its existence, a *dirty* bit to indicate the staleness of the value, and a *value* as its value.

// Decrease the value of an element index and mark affected ancestors dirty
Function DecVal($index, val$)
  if `write_min`($toggle[L][index].value, val$) **then**
    $i \leftarrow L, j \leftarrow index$
    while $toggle[i][j].dirty = 0$ **do**
      `atomic_set`($toggle[i][j].dirty, 1$)
      if $i = 0$ **then break**
      $i \leftarrow i - 1, j \leftarrow j/w$

// find all active elements with the current minimum value, apply f to them, and remove them
Function ExtractMin(f)
  if $toggle[0][0].dirty = 1$ **then** Repair$^*(0, 0)$
  $minval \leftarrow toggle[0][0].value$
  ExtractMin$^*(0, 0, minval, f)$

// Restore subtree minimal only along dirty branches in the subtree $toggle[i][j]$
Function Repair$^*(i, j)$
  $toggle[i][j].dirty \leftarrow 0$
  $toggle[i][j].flag \leftarrow 1$
  if $i = L$ **then return**
  Let $C \leftarrow \{j^* : j \cdot w \le j^* < w \cdot (j+1), toggle[i+1][j^*].dirty = 1\}$
  ParallelForEach $j^* \in C$ **do** Repair$^*(i+1, j^*)$
  $toggle[i][j].value \leftarrow \min(toggle[i][j].value, \min_{j^* \in C} toggle[i+1][j^*].value)$

// find all active elements in subtree $toggle[i][j]$ with minimum value $minval$, apply f to them, and remove them
Function ExtractMin$^*(i, j, minval, f)$
  if $i = L$ **then** *// reaching the leaf level*
    if $toggle[i][j].value = minval$ **then**
      $f(j)$ *// apply f to it*
      $toggle[i][j].flag \leftarrow 0$ *// remove it*
      $toggle[i][j].value \leftarrow +\infty$
  // enumerate all children with minval
  ParallelForEach $j^* : w \cdot j \le j^* < w \cdot (j+1), toggle[i+1][j^*].value = minval$ **do**
    ExtractMin$^*(i+1, j^*, minval, f)$
  // recalculate the value as the minimum value in its subtree
  $toggle[i][j].value \leftarrow \min_{j^* \in [w \cdot j, w \cdot (j+1))} toggle[i+1][j^*].value$
  if *none of $toggle[i][j]$'s children has flag* $= 1$ **then** $toggle[i][j].flag \leftarrow 0$

4.3 Implementation Details

4.3.1 Granularity Control

Another important detail for implementing parallel data structures is the granularity control. Instead of generating parallel tasks recursively to the last level, usually

a moderate-sized base case is used to hide parallel overhead. In the Toggle Tree, we use a simple heuristic, where granularity control is only applied to the last level. Among the $w(= 64)$ elements per word, we set the granularity to 8 *actual active elements*, i.e., rather than assigning a dedicated parallel task to each element, we group every 8 *active* elements into a single task. In our implementation, such a simple strategy performs sufficiently well on almost all graphs.

4.3.2 Bitwise Operations

To ensure that word-level operations require only $O(1)$ work, we employ three compiler built-ins provided by GCC, named `__builtin_popcountll`, `__builtin_ctzll`, and `__builtin_ia32_pdep_di`, introduced below.

- `__builtin_popcountll`: This function returns the number of 1-bits in a 64-bit word.
- `__builtin_ctzll`: This function returns the index of the lowest 1-bit in a nonzero 64-bit word.
- `__builtin_ia32_pdep_di`: This function is an x86-64-specific built-in associated with the BMI2 instruction set. In our work, we only take advantage of its following property: suppose the i-th 1-bit of a word is located at x-th bit, then

  ```
  __builtin_ia32_pdep_di(1<<i,word) = 1<<x
  ```

  and therefore `__builtin_ctzll(__builtin_ia32_pdep_di(1<<i,word))=x`.

In other words, for a given word, `__builtin_popcountll` can be used to determine the iteration range of a parallel loop, and each task can use its own index i together with

`__builtin_ctzll` and `__builtin_ia32_pdep_di` to obtain, in $O(1)$ work, the offset of the set bit it is responsible for processing.

4.4 Theoretical Analysis

Intuitively, maintaining the IndexSet using a tree instead of an array may incur a logarithmic overhead due to its tree height. While this applies to trees with small fan-outs, our analysis demonstrates that by setting the fan-out w to the word size $\Theta(\log n)$, this overhead becomes small, or negligible in most cases. As a result, the Toggle Tree can consistently deliver fast performance in graph applications due to structural simplicity, as discussed in more details in Section 4.5. Our analysis is rather simple and based on the following fact.

Fact 1 *Given a complete w-ary tree with n leaves, the total number of tree nodes of k leaves and their ancestors is $O\left(k \log_w \frac{n}{k}\right)$.*

This fact is commonly used in analyzing tree algorithms. We can pessimistically assume that the top $\lceil \log_w k \rceil$ levels are all selected with $O(k)$ nodes. At most k nodes can be selected for the rest of $\lceil \log_w n \rceil - \lceil \log_w k \rceil = O(\log_w(n/k))$ levels, leading to the stated bound.

We can now show the bounds for Toggle Tree operations. Since some algorithms are parameterized with a function $f(\cdot)$ (ForEach, Reduce, or Clear), the actual complexity relies on the cost of $f(\cdot)$. To provide a general analysis, we define the ***accessing work/span*** as the cost to traverse relevant nodes in the tree excluding the cost of these functions.

Theorem 2 (IndexSet Operations) *Given a Toggle Tree of index size n and fan-out w, k concurrent Insert or Remove operations take $O\left(k \log_w \frac{n}{k}\right)$ work, each with $O(\log_w n)$ span. ForEach, Reduce, or Clear has $O\left(k \log_w \frac{n}{k}\right)$ accessing work and $O(\log n)$ accessing span when the current set contains k elements.*

Proof. According to Theorem 1, since these operations requires to access k leaves, they will visit/modify $O\left(k \log_w \frac{n}{k}\right)$ tree nodes. For Insert, Remove, and Clear, all modifications to the tree nodes are by `fetch_and_or` and `fetch_and_and`, which takes constant work per bit and gives the stated work bound in the theorem. ForEach and Reduce are read-only, so the work is naturally $O\left(k \log_w \frac{n}{k}\right)$. For ForEach, Reduce, and Clear, the traversal consists of at most $\lceil \log_w n \rceil$ levels, each of $O(\log w)$ span for the parallel for-loop. In total, the span for these operations are $O(\log n)$ by multiplying the two terms. ■

The cost for IndexMap is slightly higher due to atomic update to the values.

Theorem 3 (IndexMap Operations) *Given a Toggle Tree of index size n and fan-out w, k concurrent DecVal operations take $O\left(k \log_w \frac{n}{k}\right)$ work; ExtractMin has $O\left(kw \log_w \frac{n}{k}\right)$ accessing work and $O(\log n)$ accessing span when the output has k elements.*

Proof. The analysis of the IndexMap operations is almost identical to that of IndexSet, with only one minor difference. In ExtractMin, since we need to check all w subtrees in each relevant tree node (line 25 in Algorithm 3), the work bound is increased by a factor of w, making it slightly less efficient than other Toggle Tree operations. ■

The bound indicates that all operations on the Toggle Tree is roughly $O\left(\log_w \frac{n}{k}\right)$ per element, which can be $O(\log n / \log \log n)$ (i.e., the tree depth) when plugging in $k = O(1)$ and $w = \Theta(\log n)$. We note that, however, for tasks suitable for parallel graph processing,

the set sizes are usually reasonably large to enable meaningful parallelism. Motivated by this observation, we provide a more detailed analysis on the cost of using Toggle Trees for graph algorithms, which can be directly plugged in the analysis for the applications using the Toggle Tree in Chapter 5.

Theorem 4 *To process s elements in a Toggle Tree in a total of ρ rounds, the accessing work is $O\left(s \log_w \frac{n\rho}{s}\right)$ work, or $O\left(s \cdot \frac{\log(n\rho/s)}{\log\log n}\right)$ when $w = \Theta(\log n)$.*

Proof. Let $s_1, s_2, \ldots, s_\rho$ be the number of elements accessed in each of the ρ rounds, where $\sum_{i=1}^{\rho} s_i = s$. The accessing work is bounded by $O\left(\sum_{i=1}^{\rho} s_i \log_w \frac{n}{s_i}\right)$. Because the function $f(x) = x \log_w(n/x)$ is concave for $x > 0$, by Jensen's inequality, this total cost is maximized in the worst case when the elements are evenly distributed, i.e., $s_1 = s_2 = \cdots = s_\rho = s/\rho$. Plugging in these values yields the stated bound. ■

With Theorem 4, we can derive tighter bounds for graph applications. For instance, in breadth-first search (BFS), every vertex is visited once (via an insertion and visited once in looping over a frontier). Consequently, we have $s = \Theta(n)$, and the total work is $O\left(n \cdot \frac{\log \rho}{\log\log n}\right)$, where ρ is the number of BFS rounds. For graph processing tasks that may benefit from parallelism, the number of rounds is usually small. For example, if ρ is polylogarithmic, the work becomes $O\left(n \cdot \frac{\log \mathrm{polylog}(n)}{\log\log n}\right) = O(n)$, which is optimal.

4.5 Comparisons to Existing Approaches

In this section, we briefly review typical frontier representations in existing graph algorithms, especially the baselines in our experiments, and discuss how Toggle Tree is designed to address the challenges in them. We still use the example of BFS algorithm

in Algorithm 1. As mentioned, the most widely-used form is to materialize all vertices in a contiguous array, as is used in many graph libraries [37, 13, 14, 12]. This streamlines ForEach traversals, but using a compact array prevents concurrent insertions to the next frontier $\mathcal{F}'$ while processing the current frontier $\mathcal{F}$. This can be addressed by a two-pass traversal of $\mathcal{F}$: first to mark the vertices to be inserted, and the second time to pack them into $\mathcal{F}'$. Another solution, represented by the *hash bag* [42], uses a hash-table-like strategy: it leaves sufficient empty slots, and a concurrent insert puts an element to a random slot in the array. Both of them incur undesired overhead in time and/or space in practice.

Beyond the sparse representation, almost all existing parallel BFS implementations incorporate a dense mode, where the presence of a vertex in the set is indicated by a bit flag. Extensive experimental studies [3, 37, 50, 16] have demonstrated the effectiveness of the dense mode in accelerating parallel graph processing. The advantage stems in part from its simplicity, and additionally from its facilitation of the *directional optimization* [3] (see Algorithm 6). However, the ForEach operations in the dense mode require checking all vertices and processing the flagged ones. This restricts the applicability of the dense representation only to a few rounds with large frontiers.

The underlying representation of Toggle Tree is also based on bit flags, making its update cost close to the dense mode. Given $w = 64$ on modern machines, the tree height is at most 6 for any graph that fits in memory (up to $64^6 \approx 6.9 \times 10^{10}$ vertices). Therefore each single update modifies at most 6 bits, and can be much fewer with early termination (see Section 4.4). For traversing the set, it does not exhaustively check all bits, but use the hierarchy to skip subtrees with all 0s. This makes the traversal performance competitive

to the sparse mode, with a minor overhead factor of $O(\log_w n/k)$ for subset size k. The value $\log_w n/k$ is also bounded by 6, and can be smaller for larger k. Altogether, based on Theorem 4, when a graph algorithm finishes in polylog(n) rounds, the overhead is a constant factor.

Using Toggle Tree, an algorithm does not need to distinguish sparse and dense modes. Beyond BFS, for problems such as graph coloring and k-core, many state-of-the-art implementations never use a dense mode, because the frontiers rarely grow sufficiently large. Using Toggle Tree, then can also leverage fast updates as if in dense modes, without much penalty in the theoretical efficiency to traverse the frontiers. In our experiments, our implementation for these two algorithms using the Toggle Tree indeed improves state-of-the-art implementations.

Chapter 5

Graph Applications using the Toggle Tree

In this section, we discuss how to apply Toggle Tree to five of the most well-known parallel graph algorithms, as examples to show its usability. Integrating Toggle Tree into these algorithms is straightforward: we keep the algorithms themselves unchanged and maintain vertex subsets using Toggle Trees. As we will show in Chapter 6, Toggle Tree significantly improves the performance of these algorithms compared to existing baselines.

5.1 Bellman-Ford

The Bellman-Ford algorithm computes single-source shortest paths from a *source* vertex. The algorithm maintains a frontier (subset of vertices) in each round and updates their neighbors. If the distance of a vertex is updated, it is added to the next frontier. The

Table 5.1: **Work and span of graph algorithms using Toggle Tree.** m: number of edges; n: number of vertices; ρ: number of rounds. We assume the machine word width $w = \Theta(\log n)$.

Algorithm	**Rounds (ρ)**	**Work**	**Span**
Bellman-Ford	shortest-path tree depth	$O(\rho(m+n))$	$O(\rho \log n)$
BFS	diameter	$O\left(m + n \cdot \frac{\log \rho}{\log \log n}\right)$	$O(\rho \log n)$
Coloring	priority DAG depth	$O\left(m + n \cdot \frac{\log \rho}{\log \log n}\right)$	$O(\rho \log n)$
KCore	peeling rounds	$O\left((m+n)\frac{\log \rho}{\log \log n}\right)$	$O(\rho \log n)$
Weighted-BFS	weighted diameter	$O\left((m + n\log n)\frac{\log \rho}{\log \log n}\right)$	$O(\rho \log n)$

Algorithm 4: Parallel Bellman-Ford using Toggle Trees

```
1  Function Bellman-Ford(G = (V, E), source ∈ V):
2      dist ← ARRAY.Init(G.n, ∞), dist[source] ← 0
3      F ← IndexSet.Init(G.n, 0), F.Insert(source)          // the current frontier
4      F' ← IndexSet.Init(G.n, 0)                           // the next frontier
5      for round ← 0, 1, ..., n − 1 do
6          if F.Empty() then break
7          F.ForEach v ∈ F do
8              ParallelForEach (u, weight) ∈ G.N(v) do
9                  if write_min(dist[u], dist[v]+weight) then
10                     F'.Insert(u)
11         F.Clear()                                        // remove all vertices from the current frontier
12         swap F and F'                                    // F gets vertices in the next frontier; F' becomes empty
13     return dist[·]
```

pseudocode is shown in Algorithm 4. We use two alternating Toggle Trees to represent the current and next frontiers, denoted as $\mathcal{F}$ and $\mathcal{F}'$. The frontier size can be n in the worst case, leading to the following bounds (as good as any known Bellman-Ford implementation).

Corollary 5 *In graph with $n = |V|$ vertices and $m = |E|$ edges, Algorithm 4 (Bellman-Ford) has $O(\rho(m+n))$ work and $O(\rho \log n)$ span, where ρ is the number of executed rounds.*

Algorithm 5: Parallel k-core using Toggle Trees

```
1  Function k-core(G = (V, E)):
2      coreness ← ARRAY.Init(G.n, 0)
3      induced ← ARRAY.Init(G.n, v ↦ |G.N(v)|)        // induced degree; initialized as the original degree
4      A ← IndexSet.Init(G.n, 1)                      // active set: all unpeeled vertices; initialized as V
5      F ← IndexSet.Init(G.n, 0)                      // the current frontier; initialized as empty
6      F' ← IndexSet.Init(G.n, 0)                     // next frontier; initialized as empty
7      while ¬A.Empty() do
8          k ← A.Reduce(v ↦ induced[v], min, +∞)
9          A.ForEach v ∈ A do                         // extract all vertices with induced degree k to the current frontier
10             if induced[v] = k then F.Insert(v)
11         while ¬F.Empty() do
12             F.ForEach v ∈ F do
13                 coreness[v] ← k                    // assign coreness k to vertex v
14                 A.Remove(v)                        // peel v (remove it from the active set)
15                 ParallelForEach u ∈ G.N(v) do
16                     if A.Contains(u) and atomic_decrement(induced[u]) = k+1 then
17                         F'.Insert(u)
18             F.Clear()                              // remove all vertices from the current frontier
19             swap F and F'                          // F gets vertices in the next frontier; F' becomes empty
20     return coreness[·]
```

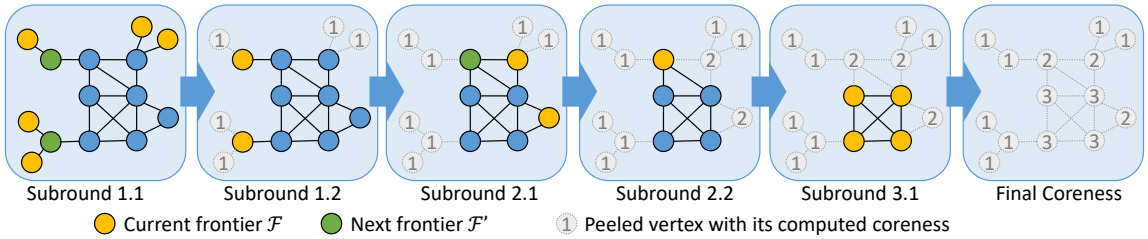

Figure 5.1: An illustration of the peeling process in the k-core algorithm.

5.2 k-core Decomposition

The k-core of a graph is the maximal induced graph where all vertices have degree k or more. The output of k-core usually computes the *coreness* of each vertex, where coreness[v] is the maximum k such that v is in the k-core of the graph. An illustration is presented in Fig. 5.1.

The k-core algorithm is a typical *peeling* algorithm, and we outline the parallel algorithm [12, 26] in Algorithm 5. The algorithm starts with k as the minimum degree in the graph. In each *round*, the algorithm first *peels* all vertices with degree k in parallel.

This may cause the degrees of remaining vertices to drop to k, and the process repeats until all vertices have degree more than k, forming several *subrounds*. All vertices peeled in this round have coreness of k. The algorithm then sets $k \leftarrow k + 1$ and continues to the next round. To distinguish from the vertex degree in the original graph, we refer to the modified degree during the algorithm as the *induced* degree.

We present the k-core algorithm using Toggle Trees in Algorithm 5, originally from [12, 26]. The algorithm maintains three IndexSets: an active set $\mathcal{A}$ for all vertices not peeled so far, a frontier $\mathcal{F}$ for all vertices in the current subround, and the next frontier $\mathcal{F}'$. At the beginning, $\mathcal{A}$ contains all vertices. In each round, vertices with degree k will be extracted from $\mathcal{A}$ and inserted into the initial frontier of the round $\mathcal{F}$. To process each $v \in \mathcal{F}$, it will be peeled from the graph, causing all its neighbors' degree to decrease by 1. If this decrement caused a neighbor u's degree to drop to k, u is added to the next frontier $\mathcal{F}'$ and removed from the active set $\mathcal{A}$. Finally, we swap $\mathcal{F}$ and $\mathcal{F}'$, clear $\mathcal{F}'$ and move to the next subround.

Liu et al. [26] proved that, assuming extracting vertices from $\mathcal{A}$ takes $O(|\mathcal{A}|)$ work, the algorithm described above has $O(m + n)$ work. This can be achieved using a flat array, and each extraction is implemented by a parallel *pack/filter*. Using Toggle Trees, however, the work is slightly higher due to the hierarchical structure. By using the Toggle Tree, we can achieve the following bound according to Theorem 4, which is work-efficient when $\rho = \text{polylog}(n)$.

Corollary 6 *Given a graph with n vertices and m edges, let ρ be the number of rounds, Algorithm 5 (k-core decomposition) has $O\left((m+n)\frac{\log\rho}{\log\log n}\right)$ work and $O(\rho\log n)$ span.*

Proof. In the k-core algorithm, s_i in IndexSet.Reduce is the number of vertices not yet peeled and remaining in the active set. A vertex v remains in the $\mathcal{A}$ set only as long as its coreness has not yet been determined. Specifically, a vertex will be removed from the $\mathcal{A}$ set in the round where the minimum induced degree k equals $coreness[v]$. Since k is strictly monotonically increasing, a vertex v can stay in the $\mathcal{A}$ set for at most $(coreness[v]+1)$ rounds.

Summing over all vertices, we have

$$s=\sum_{i=1}^{\rho}s_i=\sum_{v\in V}(\text{number of rounds } v \text{ stays in } \mathcal{A})\le\sum_{v\in V}(coreness[v]+1).$$

By the definition of k-core, the coreness of a vertex cannot exceed its initial degree, $coreness[v]\le degree[v]$. Therefore, $s\le\sum_{v\in V}d(v)+n=2m+n$. Substituting $s\le 2m+n$ into the work bound in Theorem 2, we obtain the work of Reduce and collecting minimum elements in Algorithm 5 bounded by

$$O\left((m+n)\left(1+\frac{\log(n\rho/(m+n))}{\log\log n}\right)\right)\le O\left((m+n)\left(1+\frac{\log\rho}{\log\log n}\right)\right).$$

Meanwhile, edge updates happen at most m times, and each vertex appears in the frontier at most once, so this part of the work is $O\left(m+n+n\cdot\frac{\log\rho}{\log\log n}\right)$. Hence, the total work of Algorithm 5 is $O\left((m+n)(1+\frac{\log\rho}{\log\log n})\right)$, which gives the stated bound. ■

Although the algorithm has an $O(1+\frac{\log\rho}{\log\log n})$-factor overhead in work, this value is small in practice. For $\rho=polylog(n)$, this overhead is just a constant. Benefiting from its structural simplicity, our algorithm actually outperforms the implementation of Liu et al. using a flat array in the experimental evaluation.

Algorithm 6: Parallel BFS using Toggle Trees

```
1  Function BFS(G = (V, E), source ∈ V):
2      dist ← ARRAY.Init(G.n, ∞)
3      mode ← FORWARD
4      A ← IndexSet.Init(G.n, 1), A.Remove(source)
5      F ← IndexSet.Init(G.n, 0), F.Insert(source)
6      F' ← IndexSet.Init(G.n, 0)
7      for round ← 0, 1, ... do
8          if mode = FORWARD then                                  // Forward is the default mode
9              if F.Empty() then break
10             if F.Reduce(v ↦ |N(v)|, +, 0) > α · G.m then        // Switch mode when frontier is large
11                 F.ForEach v ∈ F do dist[v] ← round
12                 F.Clear()
13                 mode ← BACKWARD
14                 continue
15             F.ForEach v ∈ F do
16                 dist[v] ← round
17                 ParallelForEach u ∈ N(v) do
18                     if A.Contains(u) then A.Remove(u), F'.Insert(u)
19             F.Clear()
20             swap F and F'
21         else                                                    // Backward Mode: directional optimization
22             A.ForEach v ∈ A do
23                 if ∃ u ∈ N(v) : ¬ A.Contains(u) then F.Insert(v)
24             if F.Empty() then break
25             if F.Reduce(v ↦ 1, +, 0) < β · G.n then             // Switch mode when frontier is sparse
26                 F.ForEach v ∈ F do A.Remove(v)
27                 mode ← FORWARD, round ← round − 1
28                 continue
29             F.ForEach v ∈ F do
30                 A.Remove(v), dist[v] ← round
31             F.Clear()
32     return dist[·]
```

5.3 Breadth-First Search (BFS)

BFS computes single-source shortest paths on unweighted graphs from a source vertex *source*. We have already shown how a simple BFS using Toggle Trees in Algorithm 1 in Section 4.1. The advanced version of parallel BFS also includes a technique called direction optimization [3], which is used in existing implementations [37, 13, 15, 5].

This algorithm also keeps two alternating frontiers $\mathcal{F}$ and $\mathcal{F}'$, and an active set $\mathcal{A}$ maintaining unvisited vertices, each implemented by a Toggle Tree. The only difference

from Algorithm 1 is that, when a frontier has sufficient size, a *backward* strategy is used, where each active vertex $v \in \mathcal{A}$ checks their neighbors, and if a neighbor is in the current frontier $\mathcal{F}$, v adds itself to $\mathcal{F}'$. Here, switching between the two modes is by the Reduce function, on lines 10 and 25, using the same heuristics as in the existing algorithms.

Based on Theorem 4, the work and span are stated below. Here the work is optimal when $\rho = \text{polylog}(n)$ or $m = \Omega(n \log_{\log n} \rho)$.

Corollary 7 *Given a graph with n vertices, m edges, and the diameter ρ, Algorithm 6 (BFS) has $O\left(m + n \cdot \frac{\log \rho}{\log \log n}\right)$ work and $O(\rho \log n)$ span.*

Proof. In top-down mode, there are at most m edge visits, and the total number of vertex accesses is at most n, so the work bound is $m + n + n \cdot \log \rho / \log \log n$. In bottom-up mode, although each single round could visit up to m edges and n insertions into the frontier, the number of rounds the algorithm remains in this mode is less than a constant $1/\beta$, so it does not change the total work bound. ■

5.4 Graph Coloring

The graph coloring problem asks to assign a color to every vertex such that no two adjacent vertices share the same color. While assigning a different color to every vertex satisfies the constraint, practically, we use heuristics that minimize the number of used colors. Here we consider the commonly used parallel algorithm by the largest-degree-first heuristics [13, 21], where vertices with the largest degree are colored first, using the smallest possible color that is not used by their colored neighbors. This iterative process finishes until

Algorithm 7: Parallel Largest-First (LF) Graph Coloring Algorithm using Toggle Trees

```
1  Function Coloring(G = (V, E)):
2      color ← ARRAY.Init(G.n, ∞)
3      perm ← an array of random permutation of [0, 1, ..., G.n]
4      priority ← ARRAY.Init(G.n, ∞)
5      F ← IndexSet.Init(G.n, 0)
6      F' ← IndexSet.Init(G.n, 0)
7      ParallelForEach v ∈ V do
8          priority[v] ←
             |{u ∈ G.N(v) : (|G.N(u)| > |G.N(v)|) ∨ (|G.N(u)| = |G.N(v)| ∧ perm[u] < perm[v])}|
9          if priority[v] = 0 then F.Insert(v)
10     while ¬ F.Empty() do
11         F.ForEach v ∈ F do
12             used ← {color[u] : u ∈ N(v), color[u] ≠ ∞}
13             color[v] ← min{c ≥ 0 : c ∉ used}
14             ParallelForEach u ∈ N(v) with color[u] = ∞ do
15                 if atomic_decrement(priority[u]) = 1 then F'.Insert(u)
16         F.Clear()
17         swap F and F'
18     return color[·]
```

all vertices are colored. The algorithm based on the Toggle Tree is given in Algorithm 7, where we also keep two alternating frontiers $\mathcal{F}$ and $\mathcal{F}'$ for the current vertices being colored, and those in the next round. The algorithm keeps $priority[v]$ for each vertex v, indicating the number of uncolored neighbors that have higher degrees (breaking ties based on a random permutation). Each vertex in the current frontier decrements their neighbors' counters, and add them to the next frontier when the counters become 0.

Based on Theorem 4, we can derive the following work and span bounds, which is optimal when $\rho = \text{polylog}(n)$ or $m = \Omega(n \cdot \frac{\log \rho}{\log \log n})$.

Corollary 8 *Given a graph with n vertices and m edges, Algorithm 7 (degree-based coloring) has $O\left(m + n \cdot \frac{\log \rho}{\log \log n}\right)$ work and $O(\rho \log n)$ span, where ρ is the executed rounds of this algorithm (or the priority DAG depth).*

Algorithm 8: Parallel Weighted-BFS (wBFS) using Toggle Tree with IndexSet interface

```
1  Function wBFS(G = (V, E), source ∈ V):
2      dist ← ARRAY.Init(G.n, +∞), dist[source] ← 0
3      A ← IndexSet.Init(G.n, 0), A.Insert(source)
4      F ← IndexSet.Init(G.n, 0)
5      round ← 0
6      while ¬ A.Empty do                          // Processing vertices with distance = round
7          A.ForEach v ∈ A do
8              if dist[v] = round then F.Insert(v)          // vertices with distance = round are settled
9          F.ForEach v ∈ F do A.Remove(v)                   // Remove settled vertices from A
10         F.ForEach v ∈ F do
11             ParallelForEach (u, w) ∈ G.N(v) do           // Relax their neighbors
                   // If the relaxation is successful, add u to the active set
12                 if write_min(dist[u], dist[v] + w) then A.Insert(u)
13         F.Clear()
14         round ← round + 1
15     return dist[·]
```

Algorithm 9: Parallel WeightedBFS using Toggle Tree with IndexMap interface

```
1  Function wBFS(G = (V, E), source ∈ V):
2      dist ← ARRAY.Init(G.n, +∞), dist[source] ← 0
3      T ← IndexMap.Init(n), T.DecVal(source, 0)
4      F ← IndexSet.Init(G.n, 0)
5      while ¬ T.Empty() do
6          T.ExtractMin v ∈ T do                    // extract settled vertices to the frontier
7              dist[v] ← the value of v in T
8              F.Insert(v)
9          F.ForEach v ∈ F do
10             ParallelForEach (u, w) ∈ G.N(v) do
11                 T.DecVal(u, dist[v] + w)
12         F.Clear()
13     return dist[·]
```

5.5 Weighted-BFS

Weighted-BFS (wBFS) computes single-source shortest paths on graphs, and extends with nonnegative integer weights. To deal with the edge weights, we show the algorithm using IndexSet in Algorithm 8, and IndexMap in Algorithm 9. Both are quite simple. For the IndexSet version, instead of using the current frontier to explore the next one, we again keep the active set $\mathcal{A}$ for unsettled vertices. In each round, we loop over all active vertices in $\mathcal{A}$, move those with the smallest tentative distance to $\mathcal{F}$, mark them as settled,

and relax their neighbors (inserting them back to $\mathcal{A}$ if successful). The IndexMap (Algorithm 9) is even simpler. As an IndexMap, we directly call ExtractMin to find all vertices with smallest tentative distance from $\mathcal{A}$. When relaxing neighbors, we directly update their distances using DecVal.

Corollary 9 *Given a graph with n vertices and m edges, Algorithm 8 (weighted BFS) based on IndexSet has $O(m + n\rho)$ work and $O(\rho \log n)$ span; Algorithm 9 based on IndexMap has $O\left((m + n\log n)\frac{\log\rho}{\log\log n}\right)$ work and $O(\rho \log n)$ span.*

Proof. For Algorithm 8, since each vertex appears in the frontier at most once, the work for edge relaxations is $O\left(m + n + n \cdot \frac{\log\rho}{\log\log n}\right)$. Another part of the work is extracting the vertices with minimum distance, which depends on how many vertices are active across all rounds. This quantity cannot be bounded more tightly in general, so we use the upper bound $s = n\rho$. Thus, the work bound of Algorithm 8 is
$O\left(m + n + n \cdot \frac{\log\rho}{\log\log n} + n\rho + n\rho \cdot \frac{\log(\frac{n\rho}{n\rho})}{\log\log n}\right) = O(m + n\rho)$.

As for Algorithm 9, the work for edge relaxations is now $O\left((m+n)\frac{\log\rho}{\log\log n}\right)$. For extracting the vertices with minimum distance, we have $s = n$, and thus the work of this part is $O\left(n\log n(1 + \frac{\log\rho}{\log\log n})\right)$. In all, the work bound of Algorithm 9 is

$$O\left(m \cdot \frac{\log\rho}{\log\log n} + n\log n\left(1 + \frac{\log\rho}{\log\log n}\right)\right),$$

with $O\left((m + n\log n)\frac{\log\rho}{\log\log n}\right)$ as shorthand. ■

Chapter 6

Experiments

Setup. Our experiments were conducted on a 96-core machine (192 hyper-threads) equipped with four 2.1 GHz Intel Xeon Gold 6252 CPUs, each with a 36MB L3 cache and 1.5TB of main memory. For all parallel tests, we used `numactl -i all` to interleave memory across all CPUs. Our code is in C++ and uses ParlayLib [6] for a parallel scheduler and simple primitives (e.g., sorting and random permutation). For algorithms with sources, we choose five random sources but avoid vertices in small connected components. We run the algorithms on each source twice (the first time for warm-up) and report the average of the second runs. For all other algorithms, we run them six times and report the average of the last five.

Datasets. We evaluate our algorithms on a diverse collection of graphs, including social networks, web graphs, road networks, k-NN graphs, and synthetic mesh-like graphs. Since k-core and coloring are defined on undirected graphs, we symmetrize all graphs to make them consistent across applications. The graph statistics, together with their references

Table 6.1: **Information of all tested graphs.** n: number of vertices. m: number of edges in the undirected or symmetrized graph.

Category	Graph	n	m	Notes
Social	FS	65.6M	3.6B	Friendster [48]
	TW	41.7M	2.4B	Twitter [24]
	OK	3.1M	234.4M	com-orkut [48]
	FK	59.2M	185.0M	socfb-konect [34, 39, 33]
	FU	58.8M	184.4M	socfb-uci-uni [34]
	LJ	4.8M	85.7M	soc-LiveJournal1 [2]
Web	CW	978.4M	74.7B	ClueWeb [29]
	SD	89.2M	3.9B	sd-arc [29]
	AR	22.7M	1.1B	arabic [34]
	UK	18.5M	523.6M	uk-2002 [29]
	EH	11.3M	521.9M	eu-2015-host [29]
	EW	6.6M	300.3M	enwiki-2023 [10, 9]
	IC	7.4M	302.0M	indochina [29]
Road	PL	372.4M	953.6M	Planet road network [32]
	EU	130.7M	332.6M	OpenStreetMap Europe [32]
	AS	95.7M	243.6M	OpenStreetMap Asia [32]
	NA	87.0M	220.3M	OpenStreetMap North America [32]
	AF	33.5M	88.9M	OpenStreetMap Africa [32]
	RU	23.9M	57.7M	USA road network [32]
	DE	12.3M	32.3M	Germany road network [32]
k-NN & Synthetic	COS5	321.1M	1.96B	Cosmo50 [25, 44]
	GL10	24.9M	309.7M	GeoLife [51, 44]
	BBL	21.2M	63.6M	hugebubbles-00020 [34]
	TRCE	16.0M	48.0M	hugetrace-00020 [34]
	CH5	4.2M	29.7M	Chem [18, 44]

and acronyms, are listed in Table 6.1. Social and web graphs are typically dense with low diameters, while the others are sparse with large diameters. For parallel graph algorithms, a lower diameter usually means fewer rounds in the algorithm, and thus better parallelism.

6.1 Overall Performance

Settings and Baselines. We implement the graph algorithms in Chapter 5, integrating Toggle Tree into state-of-the-art approaches. To isolate performance gains from the data

structure, we carefully align our algorithmic techniques with the baselines. We then compare the end-to-end performance of our implementation against these baselines. We note that all studied applications are fundamental graph problems with extensive existing work. For implementation and comparison, our techniques and baselines mainly come from two graph libraries: GBBS [13] and PASGAL [15], with their algorithms proposed in several relevant papers [13, 12, 15, 42, 26, 37].

For all five algorithms, we include the implementations from GBBS [13] as baselines, with the relevant techniques proposed in [12, 37]. GBBS is a well-known award-winning graph processing library, providing state-of-the-art performance for a wide range of graph algorithms. It uses a two-pass traversal approach (see Section 4.5) to maintain vertex sets.

We also include k-core and BFS from PASGAL [15], a more recent library that improved upon GBBS in these two problems [15, 26]. PASGAL uses hash bags (see Section 4.5) to maintain frontiers, and vertical granularity control (VGC) to optimize performance on large-diameter graphs [42]. Its k-core algorithm includes sampling for large-degree nodes [26]. We adopt these optimizations in our implementations.

All implementations, including ours, use directional optimization [3] (see Chapter 5) for BFS, in which case GBBS and PASGAL need to switch to the dense mode (a bit flag array).

Performance Analysis. We now present an end-to-end performance comparison of our implementation with baselines from existing work. For clarity, we present a summary of running times for each algorithm in Table 6.2, which reports the geometric mean for all

graphs in each category. The raw running times are provided in Table A.1 in the appendix. We also present the relative performance of all baselines on each problem in Fig. 6.1.

Across all tests, our implementations based on Toggle Trees achieve a significant advantage over the baselines. Averaging over all tested graphs, our solution is at least *1.8× faster than all the baselines for any of the tested graph algorithms.* For each individual input instance, our solution is only slower than a baseline in one case (CH5 for wBFS). This is an instance with extremely small frontiers, limiting potential parallelism. Over 50% of the frontiers have sizes under 5; over 99% are under 160. However, with such small frontiers, all parallel approaches are slower than a simple sequential weighted BFS, indicating this is not a real use case for parallel data structures. We still include it as a stress test for the Toggle Tree.

Among the remaining 124 test instances of graph-problem combinations, our solution is significantly faster than all baselines. Our solution is at least 2× faster than the *strongest baseline* in more than one third of the test instances (43/125), and is at least 1.5× faster in more than two thirds of them (83/125). These results provide strong evidence for the robust and superior performance of the Toggle Tree, indicating that using it can lead to improved performance on these parallel graph problems over the state of the art.

6.2 In-Depth Performance Study

In this section, we conduct several in-depth performance study of Toggle Tree.

Comparison to the Hash Bag. We compare Toggle Tree to its counterpart, the hash bag [42], which provides a similar interface for frontier processing. We substitute Toggle Tree with the PASGAL hash bag in four applications: Bellman-Ford, k-core, BFS, and coloring. To isolate data structure differences, we disable most optimizations (e.g., VGC) in both versions. Table 6.3 summarizes the results, with raw data deferred to Table A.3 in the appendix.

By replacing only the frontier data structure, Toggle Trees outperform hash bags by 1.56× in k-core, 1.48× in coloring, 2.75× in BFS, and 1.75× in Bellman-Ford. Across all problem-graph combinations, Toggle Trees consistently prove superior to hash bags.

To understand the performance gains, we select 6 representative graph-problem combinations. Fig. 6.4 plots the performance of Toggle Trees and hash bags across varying frontier sizes. Each point (x, y) represents a single round processing a frontier of size x with running time y in this round. Solid lines indicate the corresponding linear fits.

Across all graphs, the fitted lines for Toggle Trees exhibit smaller slopes than hash bags, indicating lower constant overhead in practice. While Toggle Trees is slower than hash bags on small frontiers for certain graphs, this regime contributes negligibly to the total end-to-end runtime. Conversely, large frontiers dominate the total execution time, a regime where the Toggle Tree maintains a clear advantage. Thus, for large-scale graph processing, the Toggle Tree is overall preferable to the hash bag.

IndexSet vs. IndexMap for wBFS. We tested two wBFS variants using IndexSet and IndexMap. In almost all cases, both outperform the baseline, with IndexMap exhibiting a

slight advantage. This aligns with our theoretical analysis (Theorem 9). While IndexMap reduces vertex extraction costs, its updates become more expensive. Hence, IndexMap remains preferable in this use case, despite the small performance gap.

Self-relative Speedup. Fig. 6.5 presents the self-relative speedups of all baselines. Each algorithm's speedup is measured relative to its own single-core execution. GBBS shows relatively low BFS speedups on road networks due to their inherent sparsity, whereas our solution and PASGAL using the VGC optimization are more resistant to this. Otherwise, the self-relative speedups of our approach and the baselines remain consistent. This demonstrates that our scalability matches state-of-the-art parallel graph libraries, and our primary performance gains are from lower total work (or smaller constant factors).

Table 6.2: **End-to-end running time comparison on the graph applications with state-of-the-art implementations. Lower is better.** Reported numbers are summary of running times (in seconds) for each graph category. Baselines are introduced in Section 6.1. Raw data of per-graph running time are in Table A.1. Each entry is the geometric mean of the graphs within the category, in seconds. Geomean in the last row is the geometric mean among all the tested graphs. IS: IndexSet; IM: IndexMap.

	BFS			Weighted-BFS		
	GBBS	PASGAL	Ours	GBBS	Ours (IS)	Ours (IM)
Social	0.046	0.067	0.033	0.399	0.237	0.220
Web	0.149	0.117	0.060	0.533	0.304	0.299
Road	2.610	0.546	0.259	21.1	5.45	4.37
k-NN & Synthetic	0.889	0.308	0.191	3.21	2.47	2.20
Geomean	0.358	0.191	0.099	2.00	0.977	0.877

	k-Core			Coloring		Bellman-Ford	
	GBBS	PASGAL	Ours	GBBS	Ours	GBBS	Ours
Social	1.17	0.722	0.377	0.921	0.453	0.465	0.307
Web	2.56	1.89	1.11	2.18	0.988	0.479	0.347
Road	0.460	0.305	0.173	0.498	0.334	15.2	8.37
k-NN & Synthetic	0.677	0.128	0.0683	0.258	0.160	15.1	6.38
Geomean	1.01	0.526	0.291	0.765	0.420	2.50	1.47

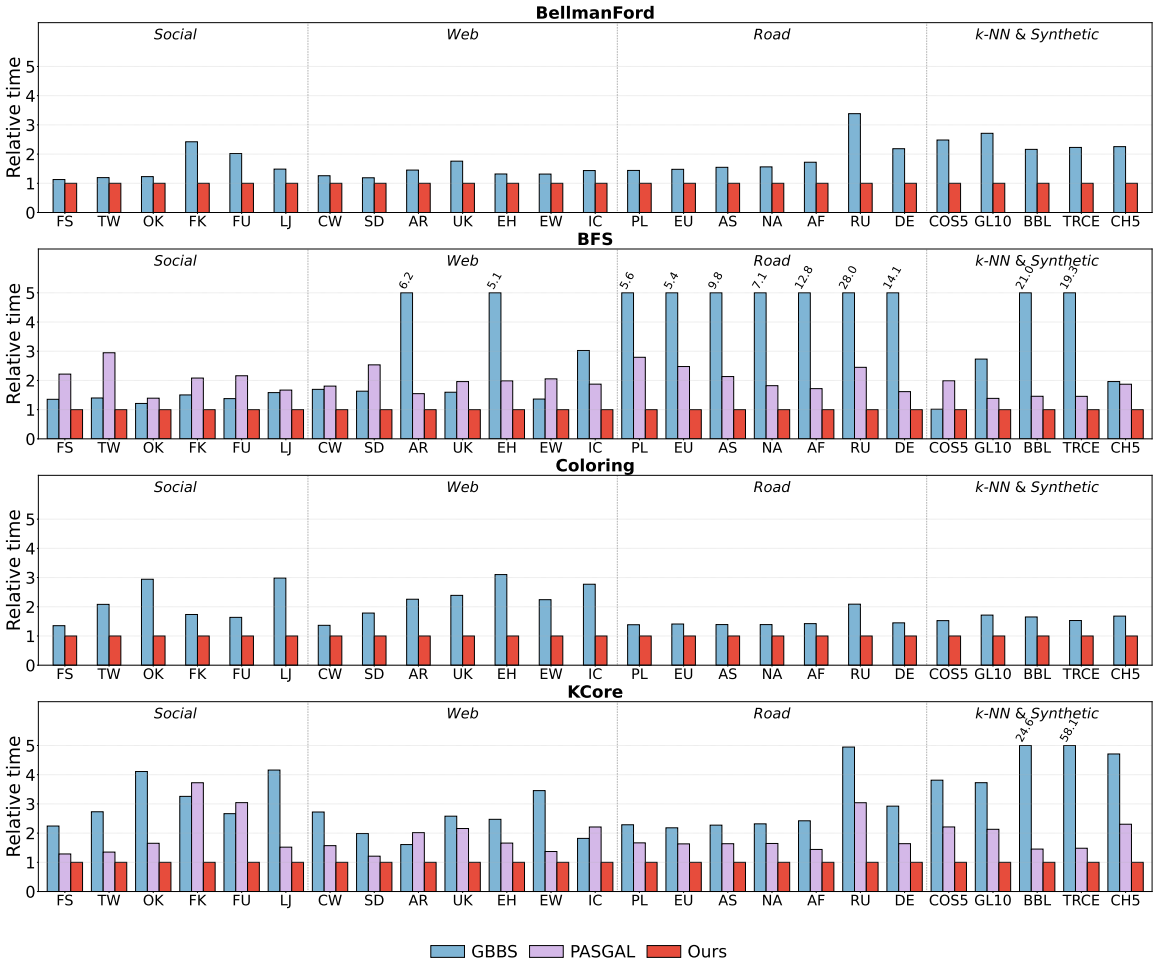

Figure 6.1: **Relative running time for Bellman-Ford, BFS, graph coloring, and k-core of Toggle Tree (Ours), GBBS [13] and PASGAL [15]. Lower is better.** Bars show running time normalized to ours (red bars = 1). Values above 5 are clipped with actual number shown above the bar.

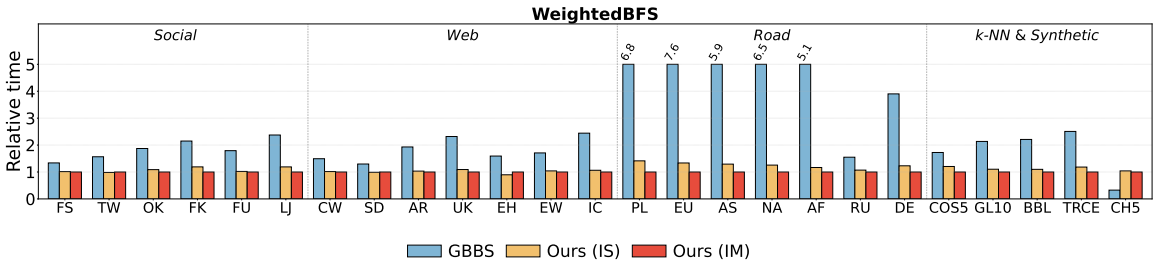

Figure 6.2: **Relative running time for wBFS of Toggle Tree (Ours) and GBBS [13]. Lower is better.** Bars show running time normalized to ours (IM). Ours (IS) and Ours (IM) are our implementations for IndexSet and IndexMap, respectively.

Table 6.3: **Running time comparison with the *hash bag*, which means to take the hash bag data structure in PASGAL and use it to replace Toggle Tree in our implementations. Lower is better.** Reported numbers are running times in seconds. Lower is better. Rawdata of per-graph running time is at Table A.3. Each entry is the geometric mean of the graphs within the category, in seconds. Geomean in the last row is the geometric mean among all the tested graphs.

	BFS		k-Core		Coloring		Bellman-Ford	
	Hash Bag	Ours	Hash Bag	Ours	Hash Bag	Ours	Hash Bag	Ours
Social	0.101	0.0327	0.748	0.477	0.658	0.453	0.544	0.307
Web	0.272	0.0604	1.80	1.58	1.38	0.988	0.692	0.347
Road	1.38	0.750	0.351	0.215	0.511	0.334	11.7	8.37
k-NN & Synthetic	0.876	0.422	0.349	0.153	0.252	0.160	12.5	6.38
Geomean	0.428	0.156	0.665	0.425	0.623	0.420	2.57	1.47

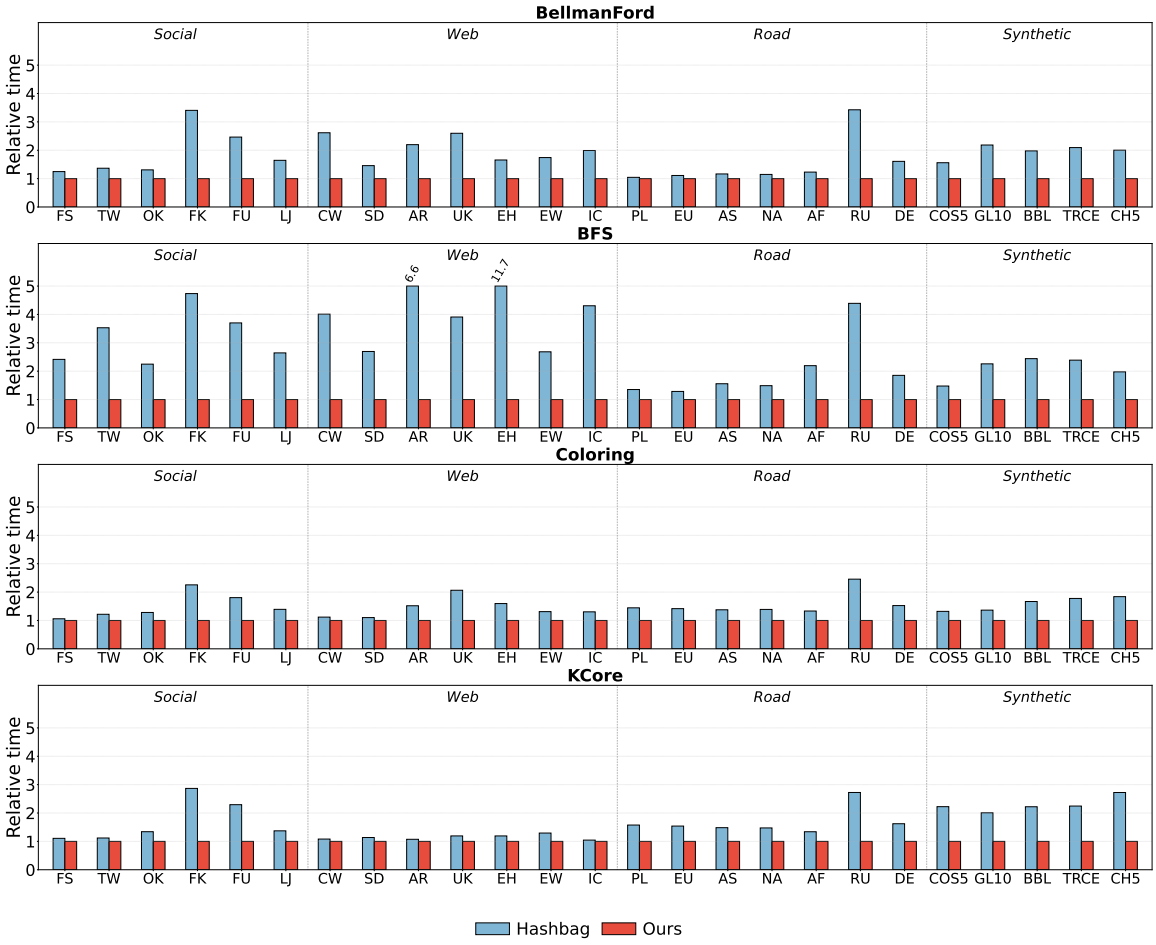

Figure 6.3: Comparison between Hashbag and our approach (Ours) for Bellman-Ford, BFS, coloring, and k-core. Blue bars show Hashbag running time normalized to ours (red bars = 1). Higher bar means Hashbag is slower relative to ours. Values above 5 are clipped with actual number shown above the bar.

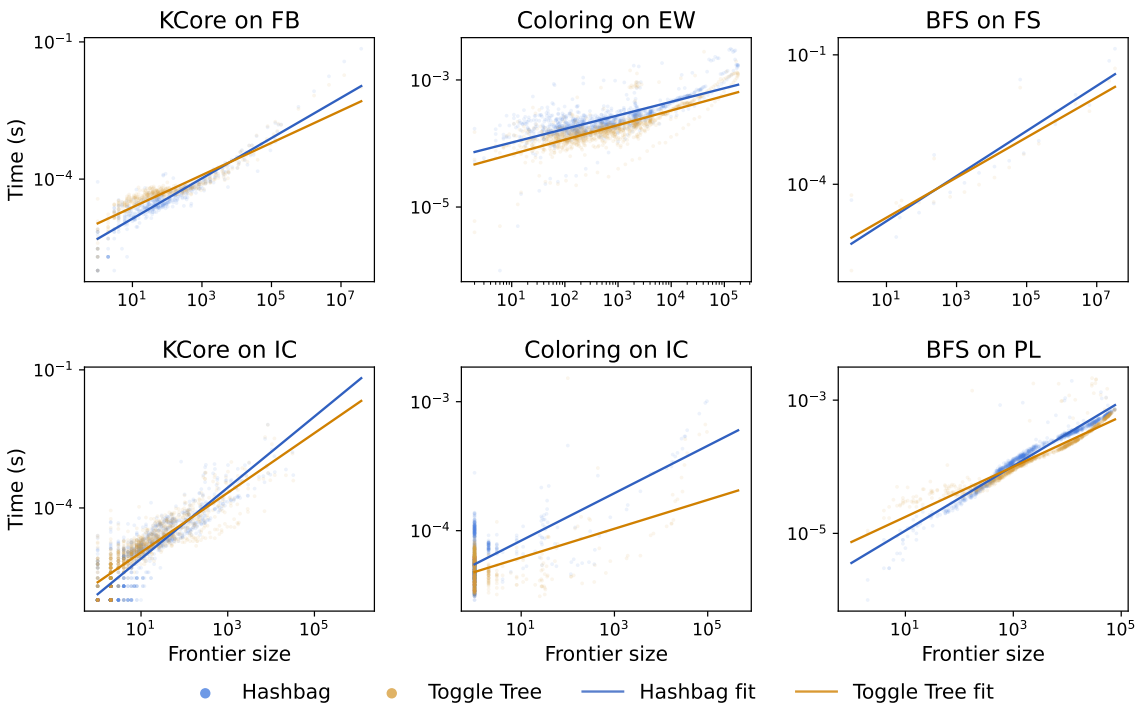

Figure 6.4: **Per-round time comparison of hash bag and Toggle Tree on k-core, graph coloring, and BFS on selected graphs**. Each point represents one round, with the x-axis showing the frontier size and the y-axis showing the time, in seconds. Lines show log–log linear regression fits. We randomly sample 1/10 points if the number of rounds exceeds 5000 for better visualization.

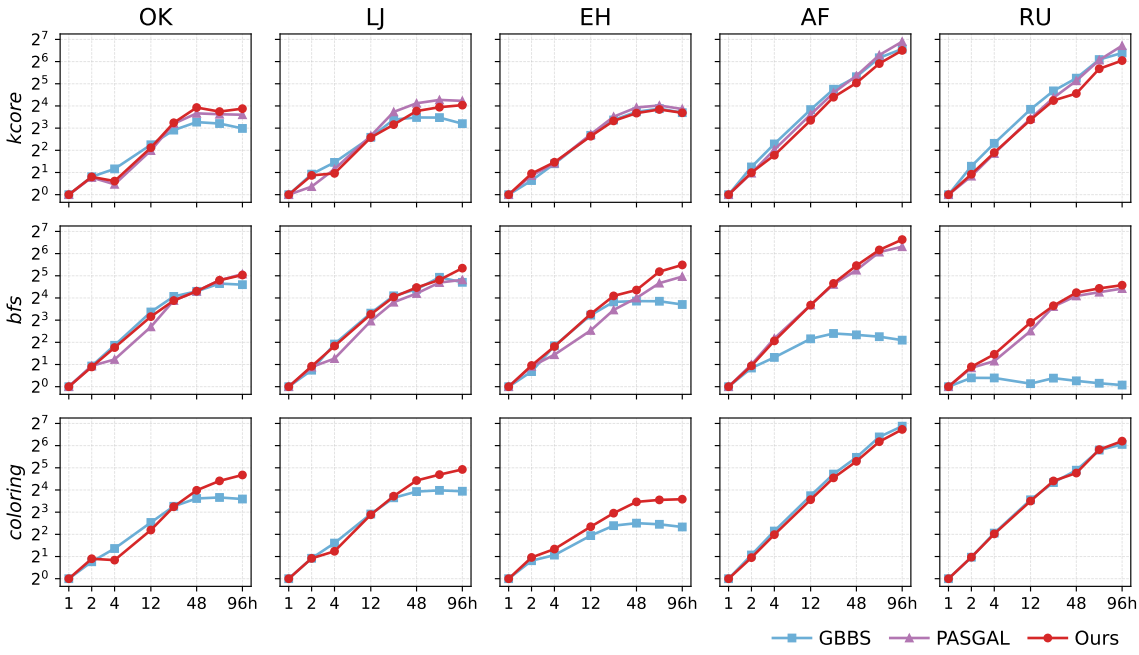

Figure 6.5: Self-relative speedup of our implementation and baselines. Log-log plot of each implementation's speedup as a function of the number of cores, relative to its own single-core performance. "96h" means 96 cores with hyperthreading. Baselines include GBBS [13] and PASGAL [15].

Chapter 7

Conclusion

We present Toggle Tree, a lightweight parallel data structure for maintaining vertex subsets in frontier-based graph algorithms. It aims to combine the advantages of traditional sparse and dense representations. The design uses bit vectors to provide easy dynamic updates (similar to the dense mode), and a tree-like hierarchy to enable efficient traversal (similar to the sparse mode). Toggle Trees provide interfaces for IndexSet and IndexMap, which are readily applicable to various parallel algorithms.

As examples, we apply Toggle Trees to five fundamental graph problems: BFS, k-core, graph coloring, SSSP using Bellman-Ford, and weighted-BFS (wBFS), and analyze their resulting cost bounds. All implementations achieve the same or nearly the same (up to a logarithmic overhead) work bounds as their counterparts in existing work.

Although conceptually simple, our experiments show that Toggle Trees are highly efficient in practice. Across a broad set of real-world graphs, our approach consistently outperforms baselines in GBBS [13] and PASGAL [15], achieving an average speedup of

at least 1.8× over the best baseline across all tested problems. These results demonstrate the effectiveness of Toggle Trees for implementing high-performance parallel algorithms. We believe that Toggle Trees will serve as a highly promising foundation for future graph algorithms and frameworks.

Bibliography

[1] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)*, 34(2):115–144, 2001.

[2] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 44–54, 2006.

[3] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, pages 1–10, 2012.

[4] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 21(3-4):137–148, 2013.

[5] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[6] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. Parlaylib — a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 507–509, 2020.

[7] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 89–102, 2020.

[8] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. on Computing*, 27(1), 1998.

[9] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *www*, pages 587–596, 2011.

[10] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *www*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[11] Xuhao Chen, Pingfan Li, Jianbin Fang, Tao Tang, Zhiying Wang, and Canqun Yang. Efficient and high-quality sparse graph coloring on gpus. *Concurrency and Computation: Practice and Experience*, 29(10):e4064, 2017.

[12] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 293–304, 2017.

[13] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)*, 8(1):1–70, 2021.

[14] Laxman Dhulipala, Charlie McGuffey, Hongbo Kang, Yan Gu, Guy E Blelloch, Phillip B Gibbons, and Julian Shun. Semi-asymmetric parallel graph algorithms for NVRAMs. *Proceedings of the VLDB Endowment (PVLDB)*, 13(9), 2020.

[15] Xiaojun Dong, Yan Gu, Yihan Sun, and Letong Wang. Brief announcement: Pasgal: Parallel and scalable graph algorithm library. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, page 439–441, 2024.

[16] Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. Efficient stepping algorithms and implementations for parallel shortest paths. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 184–197, 2021.

[17] Antoine El-Hayek, Kathrin Hanauer, and Monika Henzinger. On b-matching and fully-dynamic maximum k-edge coloring. *arXiv preprint arXiv:2310.01149*, 2023.

[18] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical*, 215:618–629, 2015.

[19] Yan Gu, Ziyang Men, Zheqi Shen, Yihan Sun, and Zijin Wan. Parallel longest increasing subsequence and van emde boas trees. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2023.

[20] Yan Gu, Zachary Napier, and Yihan Sun. Analysis of work-stealing and parallel cache complexity. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 46–60. SIAM, 2022.

[21] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. Ordering heuristics for parallel graph coloring. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 166–177, 2014.

[22] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM J. on Computing*, 14(3):654–669, 1993.

[23] Humayun Kabir and Kamesh Madduri. Parallel k-core decomposition on multicore platforms. In *2017 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 1482–1491. IEEE, 2017.

[24] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *International World Wide Web Conference (WWW)*, pages 591–600, 2010.

[25] YongChul Kwon, Dylan Nunley, Jeffrey P Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. Scalable clustering algorithm for n-body simulations in a shared-nothing cluster. In *International Conference on Scientific and Statistical Database Management*, pages 132–150. Springer, 2010.

[26] Youzhe Liu, Xiaojun Dong, Yan Gu, and Yihan Sun. Parallel k-core decomposition: Theory and practice. *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 3(3), 2025.

[27] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *ACM Sigplan Notices*, 47(8):117–128, 2012.

[28] Duane Merrill, Michael Garland, and Andrew Grimshaw. High-performance and scalable gpu graph traversal. *ACM Transactions on Parallel Computing (TOPC)*, 1(2):1–30, 2015.

[29] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. Web data commons — hyperlink graphs. `http://webdatacommons.org/hyperlinkgraph`, 2014.

[30] Ulrich Meyer and Peter Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[31] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. Deterministic galois: On-demand, portable and parameterless. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[32] OpenStreetMap contributors. OpenStreetMap. `https://www.openstreetmap.org/`, 2010.

[33] Veronica Red, Eric D Kelsic, Peter J Mucha, and Mason A Porter. Comparing community structure to characteristics in online collegiate social networks. *SIAM review*, 53(3):526–543, 2011.

[34] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI Conference on Artificial Intelligence*, 2015.

[35] Peter Sanders. Randomized priority queues for fast parallel access. *J. Parallel Distrib. Comput.*, 49(1):86–97, 1998.

[36] André Schidler and Stefan Szeider. Sat-boosted tabu search for coloring massive graphs. *ACM Journal of Experimental Algorithmics*, 28:1–19, 2023.

[37] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 135–146, 2013.

[38] Toggle tree implementation. `https://github.com/ucrparlay/toggle-tree`, 2026.

[39] Amanda L Traud, Peter J Mucha, and Mason A Porter. Social structure of facebook networks. *sma*, 391(16):4165–4180, 2012.

[40] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[41] Letong Wang, Xiangyun Ding, Yan Gu, and Yihan Sun. Fast and space-efficient parallel algorithms for influence maximization. *Proceedings of the VLDB Endowment (PVLDB)*, 17(3), 2023.

[42] Letong Wang, Xiaojun Dong, Yan Gu, and Yihan Sun. Parallel strong connectivity based on faster reachability. *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1(2):1–29, 2023.

[43] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 1–12, 2016.

[44] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. Geograph: A framework for graph processing on geometric data. *ACM SIGOPS Operating Systems Review*, 55(1):38–46, 2021.

[45] Dominic JA Welsh and Martin B Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 1967.

[46] Dan E Willard. Log-logarithmic worst-case range queries are possible in space θ(n). *Information Processing Letters*, 17(2):81–84, 1983.

[47] Carl Yang, Aydın Buluç, and John D Owens. Graphblast: A high-performance linear algebra-based graph framework on the gpu. *ACM Transactions on Mathematical Software (TOMS)*, 48(1):1–51, 2022.

[48] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[49] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. Optimizing ordered graph algorithms with graphit. In *International Symposium on Code Generation and Optimization (CGO)*, pages 158–170, 2020.

[50] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.

[51] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. Learning transportation mode from raw gps data for geographic applications on the web. In *International World Wide Web Conference (WWW)*, pages 247–256, 2008.

Appendix A

Raw Running time in Experiments

In the main body, we present figures and summary numbers of the running time in the experiments. Here we present the original running time of our algorithm and all baselines in Tables 6.3 and A.1. For each problem and each graph instance, the fastest time is marked bold.

Table A.1: Raw data for Table 6.2: BFS and weighted BFS. Each entry is the arithmetic mean of 5 runs, in seconds. Baselines are introduced in Section 6.1. IS: IndexSet; IM: IndexMap.

		BFS			WeightedBFS		
		GBBS	PASGAL	Ours	Julienne	Ours(IS)	Ours(IM)
Social	FS	0.164	0.269	**0.121**	1.88	1.42	**1.40**
	TW	0.0750	0.158	**0.0536**	0.974	**0.611**	0.622
	OK	0.0113	0.0130	**0.00932**	0.0914	0.0531	**0.0488**
	FK	0.0653	0.0904	**0.0434**	0.482	0.267	**0.224**
	FU	0.0712	0.112	**0.0517**	0.514	0.293	**0.287**
	LJ	0.0147	0.0156	**0.00932**	0.0981	0.0492	**0.0413**
Web	CW	2.67	2.85	**1.58**	16.3	11.1	**11.0**
	SD	0.261	0.406	**0.160**	1.58	**1.20**	1.22
	AR	0.385	0.0967	**0.0625**	0.377	0.202	**0.196**
	UK	0.0518	0.0636	**0.0324**	0.294	0.139	**0.127**
	EH	0.121	0.0473	**0.0238**	0.187	**0.105**	0.117
	EW	0.0154	0.0232	**0.0113**	0.120	0.0729	**0.0700**
	IC	0.0635	0.0393	**0.0210**	0.191	0.0832	**0.0782**
Road	PL	5.80	2.89	**1.03**	72.2	15.0	**10.6**
	EU	3.10	1.41	**0.571**	45.1	7.88	**5.90**
	AS	4.19	0.915	**0.429**	41.1	9.03	**6.98**
	NA	2.61	0.674	**0.370**	31.3	6.02	**4.79**
	AF	1.91	0.258	**0.150**	13.8	3.17	**2.71**
	RU	2.22	0.194	**0.0794**	6.14	4.24	**3.96**
	DE	0.999	0.115	**0.0709**	5.29	1.67	**1.36**
k-NN & Synthetic	COS5	1.75	3.42	**1.72**	9.05	6.32	**5.25**
	GL10	0.381	0.193	**0.140**	2.31	1.19	**1.08**
	BBL	2.44	0.169	**0.116**	7.58	3.77	**3.43**
	TRCE	1.62	0.122	**0.0841**	5.57	2.63	**2.22**
	CH5	0.210	0.201	**0.107**	**0.386**	1.23	1.18

Table A.2: Raw data for Table 6.2: k-core, coloring, and Bellman-Ford. Each entry is the arithmetic mean of 5 runs, in seconds. Baselines are introduced in Section 6.1.

		k-Core			Coloring		Bellman-Ford	
		GBBS	PASGAL	Ours	GBBS	Ours	GBBS	Ours
Social	FS	6.08	3.49	**2.71**	5.61	**4.15**	3.08	**2.73**
	TW	4.93	2.44	**1.81**	4.67	**2.24**	1.05	**0.883**
	OK	1.18	0.475	**0.288**	0.750	**0.255**	0.113	**0.0917**
	FK	0.346	0.396	**0.106**	0.250	**0.144**	0.518	**0.214**
	FU	0.360	0.411	**0.135**	0.315	**0.193**	0.641	**0.318**
	LJ	0.591	0.216	**0.142**	0.392	**0.132**	0.0830	**0.0559**
Web	CW	52.4	30.3	**19.3**	33.9	**24.8**	20.3	**16.1**
	SD	6.27	3.82	**3.16**	7.81	**4.37**	2.16	**1.82**
	AR	2.08	2.61	**1.29**	1.08	**0.477**	0.304	**0.209**
	UK	0.998	0.835	**0.387**	0.444	**0.186**	0.214	**0.122**
	EH	1.22	0.816	**0.492**	2.95	**0.950**	0.171	**0.130**
	EW	0.853	0.338	**0.247**	0.539	**0.240**	0.116	**0.0882**
	IC	1.01	1.22	**0.553**	1.16	**0.417**	0.102	**0.0712**
Road	PL	2.77	2.02	**1.21**	3.37	**2.43**	81.8	**56.8**
	EU	0.892	0.667	**0.409**	1.12	**0.796**	28.7	**19.4**
	AS	0.668	0.481	**0.294**	0.805	**0.579**	26.8	**17.3**
	NA	0.622	0.441	**0.268**	0.726	**0.522**	20.4	**13.1**
	AF	0.257	0.153	**0.106**	0.287	**0.202**	7.48	**4.34**
	RU	0.170	0.104	**0.0344**	0.133	**0.0635**	8.42	**2.49**
	DE	0.0969	0.0543	**0.0332**	0.0898	**0.0619**	2.33	**1.07**
k-NN & Synthetic	COS5	3.51	2.04	**0.922**	4.48	**2.94**	122	**49.2**
	GL10	0.275	0.157	**0.0738**	0.406	**0.237**	4.78	**1.76**
	BBL	1.30	0.0769	**0.0529**	0.154	**0.0934**	17.6	**8.14**
	TRCE	2.62	0.0669	**0.0451**	0.104	**0.0677**	9.57	**4.30**
	CH5	0.0430	0.0210	**0.00912**	0.0397	**0.0236**	7.86	**3.49**

Table A.3: Raw data for Table 6.3. Each entry is the arithmetic mean of 5 runs, in seconds. Hashbag: this implementation takes the hash bag data structure in PASGAL and use it to replace Toggle Tree in our own implementations.

		BFS		k-Core		Coloring		Bellman-Ford	
		Hashbag	Ours	Hashbag	Ours	Hashbag	Ours	Hashbag	Ours
Social	FS	0.296	**0.123**	3.18	**2.87**	4.39	**4.15**	3.41	**2.73**
	TW	0.188	**0.0533**	4.43	**3.96**	2.73	**2.24**	1.21	**0.883**
	OK	0.0204	**0.00907**	0.480	**0.358**	0.327	**0.255**	0.120	**0.0917**
	FK	0.199	**0.0421**	0.330	**0.115**	0.325	**0.144**	0.729	**0.214**
	FU	0.200	**0.0539**	0.327	**0.143**	0.347	**0.193**	0.783	**0.318**
	LJ	0.0242	**0.00916**	0.241	**0.176**	0.183	**0.132**	0.0919	**0.0559**
Web	CW	6.35	**1.58**	62.3	**57.6**	27.8	**24.8**	42.2	**16.1**
	SD	0.433	**0.161**	4.13	**3.63**	4.82	**4.37**	2.65	**1.82**
	AR	0.405	**0.0617**	1.86	**1.74**	0.724	**0.477**	0.460	**0.209**
	UK	0.122	**0.0313**	0.652	**0.547**	0.384	**0.186**	0.316	**0.122**
	EH	0.264	**0.0225**	0.674	**0.565**	1.52	**0.950**	0.215	**0.130**
	EW	0.0329	**0.0122**	0.374	**0.289**	0.315	**0.240**	0.154	**0.0882**
	IC	0.0936	**0.0217**	0.778	**0.744**	0.544	**0.417**	0.142	**0.0712**
Road	PL	3.19	**2.36**	2.13	**1.35**	3.51	**2.43**	59.4	**56.8**
	EU	1.78	**1.38**	0.709	**0.460**	1.13	**0.796**	21.6	**19.4**
	AS	1.95	**1.25**	0.520	**0.351**	0.797	**0.579**	20.1	**17.3**
	NA	1.43	**0.960**	0.480	**0.326**	0.725	**0.522**	15.0	**13.1**
	AF	0.980	**0.447**	0.193	**0.144**	0.269	**0.202**	5.34	**4.34**
	RU	1.35	**0.309**	0.130	**0.0477**	0.156	**0.0635**	8.53	**2.49**
	DE	0.459	**0.248**	0.0697	**0.0430**	0.0944	**0.0619**	1.72	**1.07**
k-NN & Synthetic	COS5	2.53	**1.71**	2.27	**1.02**	3.88	**2.94**	76.8	**49.2**
	GL10	0.467	**0.207**	0.198	**0.0988**	0.323	**0.237**	3.85	**1.76**
	BBL	1.25	**0.513**	0.662	**0.298**	0.156	**0.0934**	16.1	**8.14**
	TRCE	0.902	**0.378**	0.533	**0.237**	0.120	**0.0677**	9.00	**4.30**
	CH5	0.387	**0.196**	0.0324	**0.0119**	0.0434	**0.0236**	6.98	**3.49**