CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Faster Breadth-First Search on Sparse Graphs

Implementing and Optimizing an Unordered Breadth-First Search Algorithm for Sparse High-Diameter Graphs Using a Relaxed FIFO Queue

Master's thesis in Computer science and engineering

SIMON HOLST

JOHAN SELIN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

A Faster Breadth-First Search on Sparse Graphs

Implementing and Optimizing an Unordered Breadth-First Search Algorithm for Sparse High-Diameter Graphs Using a Relaxed FIFO Queue

SIMON HOST
JOHAN SELIN

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

A Faster Breadth-First Search on Sparse Graphs
Implementing and Optimizing an Unordered Breadth-First Search Algorithm for Sparse High-Diameter Graphs Using a Relaxed FIFO Queue
SIMON HOLST JOHAN SELIN

Supervisor: Philippas Tsigas, Department of Computer Science and Engineering
Co-supervisor: Kåre von Geijer, Department of Computer Science and Engineering
Examiner: Johannes Åman Pohjola, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

A Faster Breadth-First Search on Sparse Graphs
Implementing and Optimizing an Unordered Breadth-First Search Algorithm for Sparse High-Diameter Graphs Using a Relaxed FIFO Queue
SIMON HOLST
JOHAN SELIN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Breadth-First Search (BFS) is a widely used graph processing algorithm with applications in many areas. State-of-the-art parallel BFS algorithms use a level-synchronous approach which explores the graph level by level, with a synchronization barrier between each level. The level-synchronous approach works well for dense graphs, where each level is relatively large, but its synchronization requirements limit performance on sparse graphs. Unordered BFS algorithms instead explore graphs without the level constraint, thus removing the synchronization barrier, at the cost of potentially doing significantly more work to complete the traversal. We present an efficient unordered BFS algorithm for sparse high-diameter graphs. The algorithm schedules the traversal using a relaxed FIFO queue, i.e., a queue that is allowed to dequeue items other than the head, to improve queue throughput. Additionally, we introduce two novel optimizations for unordered BFS. On a shared memory system, our algorithm achieves speedups of 1.6-2.4 compared to the state-of-the-art level-synchronous algorithm on four sparse real-world graphs.

Keywords: Concurrency, Algorithms, Unordered Breadth-First Search, Relaxed Data Structures, Scalability, Performance.

Acknowledgements

We would like to thank our supervisors, Kåre von Geijer and Philippas Tsigas. In particular, we express our gratitude to Kåre, who has provided excellent feedback, advice, and ideas.

We are also grateful to our examiner, Johannes Åman Pohjola, for his time and effort in evaluating our work.

Finally, we wish to thank Scott Beamer, author of the Direction-Optimizing BFS algorithm, for advice and guidance in analyzing Direction-Optimizing benchmark anomalies.

Simon Holst, Johan Selin, Gothenburg, 2025-06-12

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

2 Background **3**
2.1 Concurrent Programs . . . 3
2.2 Concurrent Data Structures . . . 4
2.2.1 Lock-free Data Structures . . . 5
2.2.2 Lock-free Queues . . . 5
2.3 Relaxed Data Structures . . . 6
2.4 d-Choice Balanced Operations Queue . . . 7
2.4.1 Sub-queue Implementation . . . 8
2.4.2 Ordering Errors . . . 9
2.5 Breadth-First Search . . . 9
2.5.1 Graph Definitions . . . 9
2.5.2 BFS Traversal . . . 9
2.6 The Parent-tracking BFS Problem . . . 10
2.7 Level-synchronous BFS . . . 10
2.8 Direction-Optimizing BFS . . . 11
2.9 Unordered BFS . . . 13
2.9.1 Example . . . 13
2.9.2 Correctness . . . 14
2.10 Related Work . . . 15

3 Algorithm **17**
3.1 Base Algorithm . . . 17
3.1.1 Termination Detection . . . 18
3.2 Optimizations . . . 19
3.2.1 Batching . . . 20
3.2.2 Dequeue-and-Decide . . . 20
3.2.3 Bounce-on-Depth . . . 22
3.3 Final Algorithm - DCBFS . . . 23

4 Experimental Evaluation **25**

4.1 Experiment Setup . . . 25
4.1.1 Evaluated Algorithms . . . 25
4.1.2 Graphs . . . 26
4.1.3 Machines and Experiment Setup . . . 26
4.1.4 Removal of Outliers . . . 27
4.2 Elapsed Time Comparisons . . . 27
4.3 Algorithm Scalability . . . 29
4.4 Elapsed Time Distributions . . . 31
4.5 Work Efficiency . . . 32
4.6 Queue Size . . . 36

5 Discussion **39**
5.1 Direction-Optimizing Outliers . . . 39
5.2 Omitted Optimizations . . . 40
5.2.1 Sequential Start . . . 41
5.2.2 Stickiness . . . 41
5.3 Future Work . . . 42
5.3.1 A Fast Bag . . . 42
5.3.2 Tuning Parameters . . . 42

6 Conclusion **43**

Bibliography **45**

List of Figures

2.1 Amdahl's Law showing speedups for a program where the parallelizable fraction is 100%, 95%, 90%, and 50%, using up to 128 processors. 4
2.2 Average throughput over 10 runs of 5 seconds of random enqueue/dequeue operations per queue on an Intel Xeon E5-2695 v4 processor. Between each operation is 100 cycles of "side work", as defined by [10]. The queues are pre-filled with 1024 elements. 6
2.3 Example state of a d-CBO queue with $n = 5$ sub-queues and $d = 2$ sampling size. In this visualization, elements do not move after being enqueued. The positions of the elements in green represent the number of enqueue operations performed on sub-queues 2 and 5. Positions of the elements marked with red represent the number of dequeue operations performed on sub-queues 1 and 4. In this example sub-queue 2 would be selected for enqueueing, and sub-queue 1 for dequeueing. 8
2.4 Comparison of level-synchronous BFS using top-down and bottom-up steps. Traversal has advanced to highlighted frontier. Explored and unexplored edges are shown as solid and dashed lines, respectively. . . 12
2.5 Example of a possible execution of BFS where redoing work is necessary, due to visiting nodes at different depths simultaneously. Solid and dashed lines represent explored and unexplored edges, respectively. 13

3.1 Dequeue-and-Decide decision process for choosing which batch to explore next. The entry step (consumer-batch loop) is the point where all of the nodes in the consumer-batch have been explored (Algorithm 7, lines 11–24). 22

4.1 Each algorithms' best performing configuration, with specifics detailed in Table 4.5. 28
4.2 Each algorithms' best performing configuration on a maximum number of threads, with specifics detailed in Table 4.6. 29
4.3 Scalability of algorithms using the same configuration as specified in Table 4.5, excluding the number of threads. Only DO and DO_TD have additional data points at 8 and 16 threads due to their performance peaking earlier. 30

4.4 Elapsed time distributions on road-networks and the hugebubbles-graph. All algorithms are chosen by their best performing number of threads and use $b = 32$, where applicable. Black bars show the mean elapsed times, and blue bars show the extreme elapsed times. Delta percentages (Δ) represent the difference in elapsed time extreme values. 31
4.5 Elapsed time distributions on road-networks and the hugebubbles-graph. All algorithms run with $t = 512$, and $b = 32$ where applicable. Black bars show the mean elapsed times, and blue bars show the extreme elapsed times. Delta percentages (Δ) represent the difference in elapsed time extreme values. 32
4.6 Work inefficiency of unordered algorithms. Two additional data points for algorithm DAD $b = 64$ are out of bounds. At 384 and 512 threads, its work inefficiency is 6.5× and 7.5×, respectively. 33
4.7 Distributions of work inefficiencies for unordered algorithms on the road-europe graph. 34
4.8 Work inefficiencies and processing rates for unordered algorithms, compared to a sequential algorithm on the road-europe graph using 512 threads. 35
4.9 Work inefficiencies and processing rates for unordered algorithms, compared to a sequential algorithm on the road-europe graph using 512 threads. 35
4.10 Queue sizes during traversals of road networks averaged over 32 traversals per algorithm. The results have been stretched to all have the same length. Opaque lines show smoothened results for readability, while the semi-transparent lines shows the exact values. The dotted lines (UBFS) are multiplied by a factor of 0.05. The data was gathered on Ithaca using 72 threads. 37

5.1 Distribution of time taken per traversal for the DO algorithm on Ithaca and Athena on the hugebubbles and road-asia graphs respectively. 64 traversals were performed per thread count. The runs on Athena were not using SMT. Black lines show the median elapsed time. 39
5.2 Time taken per top-down step (points) in the DO algorithm and the size of the frontier at each step (curves). The black color subplot shows an outlier run. The red subplot shows an artificially produced outlier run by oversubscribing cores. The green subplot shows a "normal" run, without any interference. The number of steps differ between runs due to the traversals beginning at different source nodes. All data was gathered from Ithaca using 72 threads. 40

List of Tables

4.1 Algorithms used in experiments. 26
4.2 Symbols for the parameters of our algorithms. 26
4.3 Graphs used in experiments. 27
4.4 Machines used in experiments. 27
4.5 Best performing configurations for the algorithms shown in Figure 4.1. 28
4.6 Best performing configurations on maximum number of threads ($t = 512$) for the algorithms shown in Figure 4.2. Algorithms whose only configurable parameter is thread count have been omitted from this table. 29

1

Introduction

Breadth-First Search (BFS) is a widely used graph-traversal kernel and is commonly considered one of the most important graph algorithms [1]. It has applications in many areas such as shortest path problems [2], cycle detection [3], web crawling [4], image processing [5], analyzing social networks [6], and more. As such, increasing the performance of the BFS algorithm is highly desirable. Moreover, with modern day processors' increasing core and thread count, finding highly parallel solutions is particularly useful.

State-of-the-art concurrent BFS methods use a *level-synchronous* approach [7]. A graph can be partitioned into *levels*, where each partition is a set of nodes with equal distance to the source node. As the name "level-synchronous" indicates, the approach synchronizes all threads between each level, i.e., each level must be fully explored before any thread can move on to the next. As levels grow larger, a smaller portion of time is spent synchronizing the threads, due to the work-to-synchronization ratio increasing. By partitioning each level, threads can each work on a partition of a level almost completely independently. The approach therefore works very well on dense graphs and scales well with high parallelism.

The level-synchronous approach does, however, not work well on all types of graphs and has two main drawbacks. First, if levels are very small, it might not be possible to partition the level and distribute the work effectively. Second, if the graph has many levels, there will be many synchronization points as well, which can significantly degrade performance. Level-synchronous traversals on sparse graphs with high diameters[1] are affected by both of the approach's disadvantages; a high diameter results in many synchronization points and the small size of the levels can potentially result in the hardware not being fully utilized. This suggests that diverging from a level-synchronous approach for this type of graph could be beneficial. However, working level-synchronously guarantees that the distance to each node is always correct when it is first explored. In contrast, an *unordered* approach removes the level-synchronous constraint, resulting in threads not having to synchronize between levels and work being more easily distributed [8]. However, it also introduces the possibility of exploring nodes in an incorrect order. In such cases, work will have to be redone in order to arrive at a correct solution.

[1]Informally, the diameter of a graph is the number of steps in the longest shortest path between any two nodes in the graph. For BFS, the number of levels is at least half the graph's diameter.

In 2011, Hassaan et al. [8] presented an unordered BFS algorithm where the traversal was scheduled using a first-in-first-out (FIFO) queue. By itself, scheduling the traversal using a FIFO queue does not make the traversal unordered, i.e., it does not introduce extra work. Exploring nodes in the wrong order is a consequence of using a FIFO queue together with concurrency. Due to concurrency, nodes may not be explored in the same order that they were originally enqueued, thus leading to an unordered traversal. Despite this, the algorithm presented by Hassaan et al. [8] was almost as fast as the corresponding level-synchronous implementation in their paper.

FIFO queues, particularly older designs such as the one likely[2] used by Hassaan et al. [8], do not scale well with high parallelism due to high memory contention of the head and tail [9]–[12]. More recent research suggests *relaxing* ordering semantics to improve parallel performance in concurrent data structures [9], [12], [13]. By relaxing the FIFO semantics of the queue, a dequeue operation is allowed to return an item which is not the oldest one in the queue. The *degree of relaxation* may be restricted to some bound, such as the queue only being allowed to dequeue any of the k oldest elements [9]. Using a relaxed FIFO queue can potentially improve the unordered BFS approach, but it also introduces another source of incorrect exploration order as a consequence of the relaxed semantics. Indeed, our results show speed ups of 1.6× to 2.4× on large real-world sparse graphs compared to the current state-of-the-art level-synchronous algorithm, known as *Direction-Optimizing* (DO) BFS [7].

In this thesis, we explore the optimization and usage of a relaxed FIFO queue in unordered BFS and compare the approach with the DO algorithm [7]. In Chapter 2 we explain the background discussed above in greater detail, such as concurrent and relaxed FIFO queues, unordered BFS, level-synchronous BFS and its state-of-the-art implementation, as well as related work in the area. Chapter 3 explains our unordered BFS algorithm and optimizations that were effective. Chapter 4 presents measurements and benchmarks of our algorithm and different configurations of it, and compares them to the state-of-the-art DO method. Chapter 5 discusses irregular behavior from the DO algorithm, optimization techniques that were implemented but omitted, as well as possibilities for future work. Finally, Chapter 6 concludes the thesis.

[2]The paper does not specify the queue implementation, but it is likely suboptimal since more performant queue designs have been published after this work.

2

Background

This chapter introduces the necessary background to understand and reason about utilizing a *relaxed* FIFO queue in unordered BFS traversals. Section 2.1 discusses the impact and difficulties of moving from a sequential program to a concurrent one. Followed by that, the differences between sequential and different types of concurrent data structures are discussed in Sections 2.2-2.4. This includes what *lock-free* and *relaxed* data structures are, how they typically work, and concrete implementations of said data structures. Sequential, parallel, and state-of-the-art versions of the BFS algorithm are then discussed in Sections 2.5-2.9. Additionally, the shortcomings of the current state-of-the-art method are highlighted, including when and how it might be improved on. Finally, Section 2.10 discusses related work in this area.

2.1 Concurrent Programs

Modern computer systems have since long relied on parallelism to improve performance [14]. Problematically, concurrent programming is fundamentally harder than its sequential counterpart and fully utilizing said parallelism is often difficult. In a sequential process, instructions are executed by a single execution context, i.e., a core or thread. The events in such a process form a total order, allowing for reasoning based on a single and predictable execution path. In concurrent systems however, this deterministic property is lost and the execution order of events are not guaranteed. As a consequence, every possible execution path must be considered by the programmer to ensure correctness. Further, many entirely new problems arise such as safely accessing shared memory, decomposing the task effectively, avoiding deadlocks, and managing synchronization costs.

In an ideally parallel system, the performance scales linearly with the number of execution contexts. Of course, this is seldom the case in real world applications and the aforementioned problems can still greatly limit the performance improvements. The maximum possible speedup of a parallel system can be described using Amdahl's Law [15]:

$$S = \frac{1}{1 - p + \frac{p}{n}} \tag{2.1}$$

where S is the speedup factor, p is the fraction of the program that can be parallelized, and n is the number of processors. Figure 2.1 visualizes the potential

speedups for $p \in \{1, 0.95, 0.9, 0.5\}$.

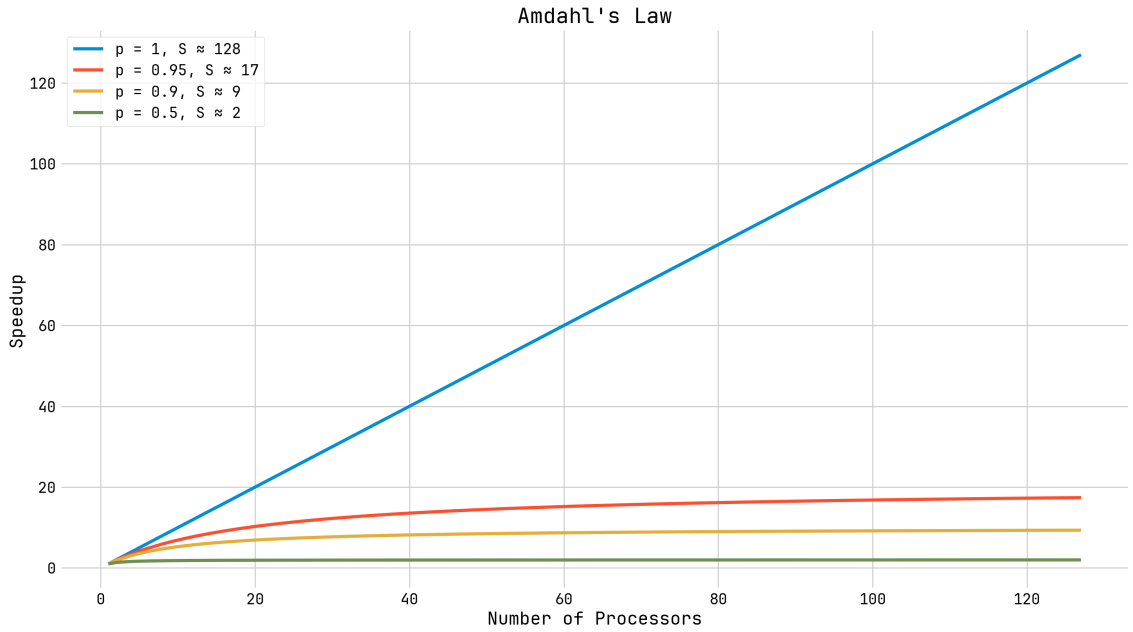


Figure 2.1: Amdahl's Law showing speedups for a program where the parallelizable fraction is 100%, 95%, 90%, and 50%, using up to 128 processors.

It is evident from the figure that even a small sequential portion severely inhibits the speedup gains in highly parallel systems. Time spent synchronizing access to shared memory, e.g., one thread waiting to acquire a lock from another, can be seen as a (mutually) sequential portion. The same applies to other synchronization mechanics as well, although in less obvious ways. Consequently, minimizing synchronization is vital to fully utilize parallel computers.

2.2 Concurrent Data Structures

In shared memory concurrent contexts, one cannot use the same data structures as in sequential contexts but must instead employ new strategies to ensure correctness. Perhaps the simplest strategy is to enforce mutual exclusion between threads, thus serializing access. This is commonly achieved by the use of a *lock*, where a thread must first acquire the lock before accessing the shared resource [16]. However, this simple approach comes with its costs, as threads can be withheld from accessing the shared resource. This can result in poor performance if a significant portion of time is spent waiting to receive access rather than performing work. In addition, managing the locks themselves also incurs overhead costs [16]. For FIFO queues, which are of great interest due to their use in BFS algorithms, this problem is particularly apparent due to the highly contended memory positions of its head and tail.

2.2.1 Lock-free Data Structures

Lock-free concurrent data structures guarantee system-wide progress, i.e., at any point in the program, at least one thread is making progress [17]. In practice, this generally results in the management of shared data without the use of mutual exclusion mechanisms [18], which can enable better scaling and performance compared to traditional synchronization techniques [19]. Instead, these data structures typically rely on atomic operations to achieve consistency. For example, fetch-and-add (FAA) or compare-and-swap (CAS) instructions. These are known as read-modify-write operations. As the name suggests, they read a memory location, update it in some way, and finally write the result to the same memory location, all in one atomic step. The FAA instruction simply increments the content of a memory location by some value. The CAS instruction is more complicated. It is defined as follows:

```
CAS(a,e,n) -> bool
```

where `a` is an address, `e` is the expected value at address `a`, and `n` a new value. If the value at `a` is equal to `e` then set the value at address `a` to `n` and return `true`. Otherwise, do nothing and return `false`.

The way in which lock-free data structures use these atomic operations to achieve consistency is not as straightforward in comparison to lock-based data structures. CAS-based algorithms typically synchronize using *CAS loops* [20]. In essence, a thread will read the shared data, then individually perform some computation, and finally update the shared data using a CAS call. This process is repeated until the CAS is successful, in which the process appears to have occurred atomically. The atomicity of the operation comes at a cost since each failed loop is work that was wasted by the CPU, which can become a significant bottleneck when contention is high [21].

2.2.2 Lock-free Queues

The Michael & Scott (MS) queue [10] is an influential lock-free concurrent FIFO queue introduced in 1996. The queue is implemented as a singly-linked list with each node containing a single value. To ensure correctness it uses CAS loops in its enqueue and dequeue operations. As mentioned, FIFO queues typically struggle with high contention of the head and tail positions. Consequently, the MS queue has suboptimal performance under high parallelism.

The design of the MS queue has since been improved upon by replacing each node with an array to reduce memory pressure. These designs keep track of where to enqueue and dequeue elements in the array by use of atomic counters that are incremented using FAA instructions. The LCRQ[1] algorithm by Morrison and Afak in 2013 [20] and the FAAArrayQueue (FAAAQ) by Ramalhete in 2016 [11] are two state-of-the-art examples. The effectiveness of this change is drastic, as seen in Figure 2.2, where each thread repeatedly flips a coin to decide whether to perform

[1]The acronym is not fully explained by the authors. However, CRQ stands for "Concurrent Ring Queue", and from context we assume that L stands for either "Linked" or "Linearizable". Thus, the acronym likely stands for Linked/Linearizable Concurrent Ring Queue.

an enqueue or dequeue operation. The throughput of the MS queue decreases as the number of threads increases, due to contention on the head and tail. In contrast, the throughput of the FAAArrayQueue and LCRQ algorithm strictly increase with an increased number of threads.

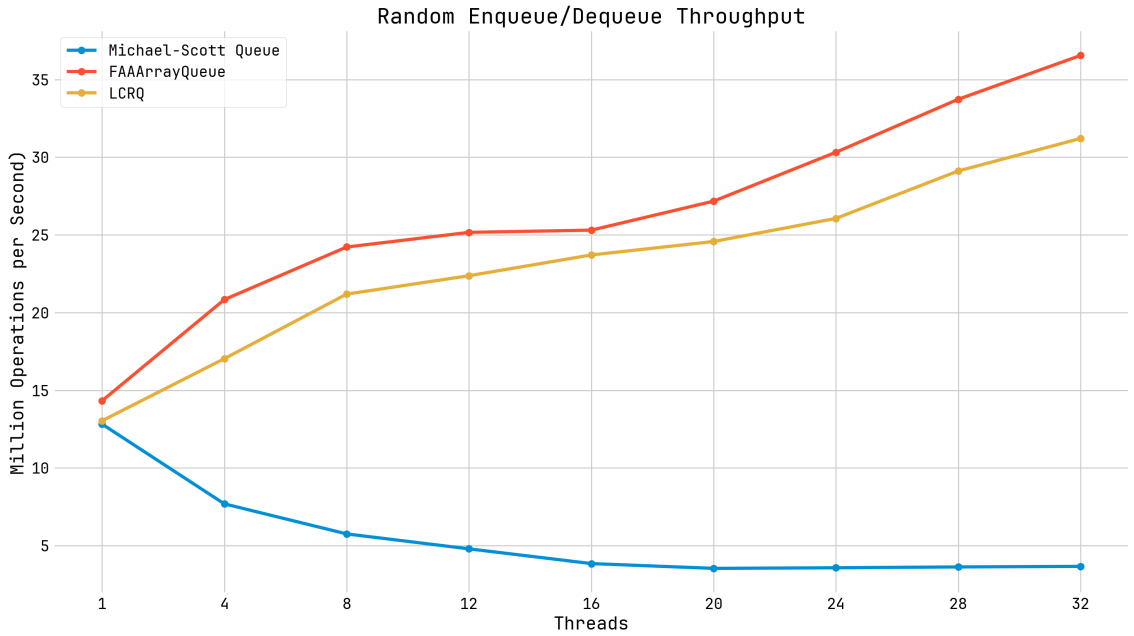


Figure 2.2: Average throughput over 10 runs of 5 seconds of random enqueue/dequeue operations per queue on an Intel Xeon E5-2695 v4 processor. Between each operation is 100 cycles of "side work", as defined by [10]. The queues are pre-filled with 1024 elements.

To our knowledge, since the introduction of the LCRQ in 2013 there have been no publications of concurrent FIFO queues with consistently better performance [12], [22]. Queues such as the FAAArrayQueue [11] or LPRQ[2] [23] by Romanov and Koval, improves the design in some aspects such as simplicity or portability but the performance remains largely the same. Indeed, the nature of queues such as those aforementioned have been proven inherently sequential by Ellen et al. [24]. They propose replacing strongly ordered data structures (such as queues) with *relaxed* ones for more efficient utilization of modern parallel hardware.

2.3 Relaxed Data Structures

Concurrent data structures are commonly implemented as simply variants of their sequential counterparts. As discussed in the previous section, some of these designs are inherently limited in their scalability. However, *relaxing* the semantics of a data structure can greatly increase the potential for concurrent scalability [13], [25], [26]. Typically, this improved performance is gained in exchange for correctness [9], such

[2]Similarly to LCRQ, the acronym is not explained. A possibility is "Linked/Linearizable Portable Ring Queue".

as relaxing the total order constraint of a FIFO queue. Fully relaxing a queue would be to replace it with a pool, which completely disregards ordering and would enable maximum performance gains. However, such a change might not be practical for applications which depend on, at least, some degree of ordering.

Instead, the FIFO queue might be relaxed to a k-out-of-order queue. Compared to a strict FIFO queue, which always dequeues the oldest element, a k-out-of-order queue can dequeue any one of the $k + 1$ oldest elements. The number of skipped elements is known as the *rank error*, shown in Definition 2.3.1. Another dimension of errors is the *delay* (Definition 2.3.2), which is the number of elements that have skipped past e before e is dequeued.

For BFS algorithms utilizing relaxed FIFO queues, both rank errors and delays can result in work needing to be redone. We discuss this in more detail in Section 2.9. However, whether work needed to be redone because of a rank error or delay is not important for the performance of the algorithm. We group these as *ordering errors*, see Definition 2.3.3.

Definition 2.3.1 (Rank error)**.** Consider a relaxed FIFO queue Q where an element e was enqueued at time t_e and dequeued at time t_d. The rank error of e is then given by the number of elements that were enqueued in Q before t_e and subsequently dequeued after t_d.

Definition 2.3.2 (Delay)**.** Consider a relaxed FIFO queue Q where an element e was enqueued at time t_e and dequeued at time t_d. The delay of e is then given by the number of elements that were enqueued in Q after t_e and subsequently dequeued before t_d.

Definition 2.3.3 (Ordering error)**.** Dequeueing an element e from a relaxed FIFO queue with rank error > 0 or delay > 0.

2.4 d-Choice Balanced Operations Queue

The d-Choice Balanced Operations Queue (d-CBO) by von Geijer et al. [12] is a highly performant relaxed FIFO queue. Internally, it consists of n independent concurrent sub-queues. To enqueue or dequeue elements, d random sub-queues are sampled in order to determine which of them to operate on. This is decided by keeping track of how many enqueue and dequeue operations have been performed on each sub-queue. Figure 2.3 shows an example representation of a d-CBO queue with $n = 5$ sub-queues and $d = 2$ sampling size. In the example, sub-queues 2 and 5 are sampled for an enqueue operation. These have enqueue-operation counts of two and four, respectively. Since sub-queue 2 has fewer enqueue-operations performed, it will be the queue that is enqueued to. Similarly, sub-queues 1 and 4 have been randomly sampled for a dequeue. Sub-queue 2 will then be selected as it has only one dequeue operation performed on it, compared to two for sub-queue 4.

The mechanics of the d-CBO queue allows it to scale very well with increased parallelism, as can be seen in the synthetic enqueue/dequeue benchmarks in [12]. When the thread count is high, the d-CBO queue with the FAAArrayQueue as sub-queue

displays throughput that is orders of magnitude greater compared to the FAAArrayQueue itself.

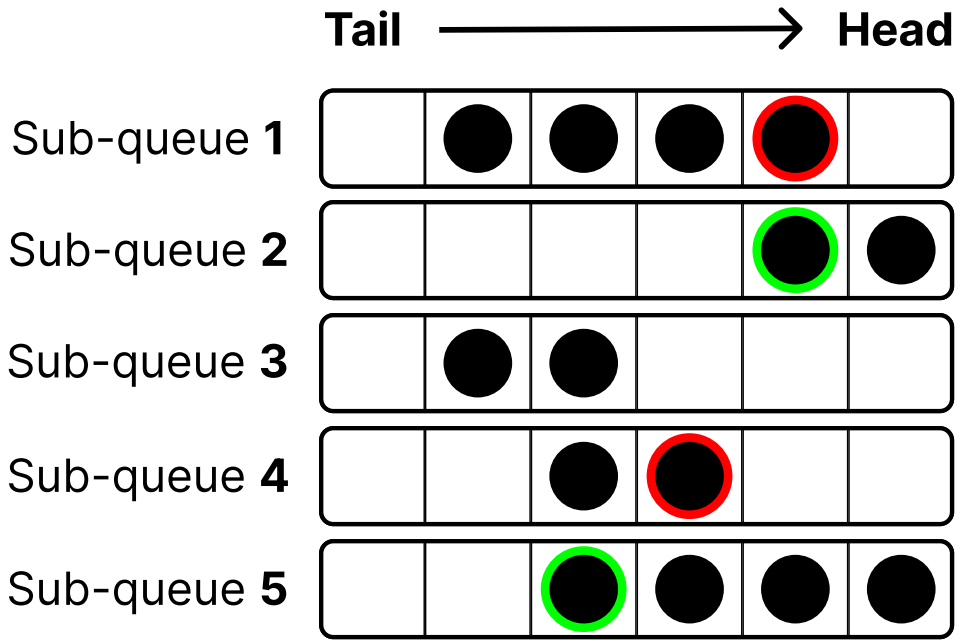


Figure 2.3: Example state of a d-CBO queue with $n = 5$ sub-queues and $d = 2$ sampling size. In this visualization, elements do not move after being enqueued. The positions of the elements in green represent the number of enqueue operations performed on sub-queues 2 and 5. Positions of the elements marked with red represent the number of dequeue operations performed on sub-queues 1 and 4. In this example sub-queue 2 would be selected for enqueueing, and sub-queue 1 for dequeueing.

2.4.1 Sub-queue Implementation

The d-CBO algorithm can be implemented with any sub-queue algorithm, as long as it adheres to the required interface. This allows for the d-CBO queue to be implemented using scalable sub-queues such as the FAAArrayQueue or LCRQ. The interface consists of three operations, apart from the obvious *Enqueue* and *Dequeue* operations:

- *EnqCount*. Returns the number of elements the sub-queue has enqueued.
- *DeqCount*. Returns the number of elements the sub-queue has dequeued.
- *EnqVersion*. Returns a number that is unique for each tail-item.

An *Enqueue* into the d-CBO queue will sample d sub-queues and enqueue the element into the one with the lowest *EnqCount*. Similarly, a *Dequeue* operation will dequeue from the sub-queue with the lowest *DeqCount*. The *EnqVersion* is used to determine if the d-CBO queue is empty. The exact details of how and why *EnqVersion* functions the way it does are outside of this thesis' scope. For our

purposes, queues such as the MS queue and FAAArrayQueue can be implemented with $EnqVersion = EnqCount$.

2.4.2 Ordering Errors

As mentioned in the previous section, the d-CBO queue's heuristic for deciding which sampled sub-queue to operate on is based on the number of enqueue / dequeue operations performed on each sub-queue. The operation-based heuristic results in a low degree of ordering errors in practice, and gives a probabilistic bound on both rank errors and delays of $O(\frac{n \log \log n}{\log d})$ with high probability. In practice, the queue has a mean rank error and delay approximately equal to the number of sub-queues [12].

2.5 Breadth-First Search

This section establishes a graph definition, which is used to formalize the BFS traversal order in terms of such a graph.

2.5.1 Graph Definitions

We define an undirected, unweighted graph $G = (V, E)$, where:

- $V = \{v_1, v_2, ..., v_n\}$ denotes the set of nodes in G
- $E = \{e_1, e_2, ..., e_m\}$ denotes the set of edges in G, where an $e \in E$ consists of an unordered pair of nodes $(v_i, v_j) \in V \times V$
- $|V|, |E|$ denotes the number of nodes and edges in G, respectively
- D denotes the diameter of the graph, which is defined as the shortest path between the two most distanced nodes in G

Despite our algorithm's (Algorithm 7) capability to process directed graphs, we restrict G to be undirected due to such graphs being easier to reason about.

2.5.2 BFS Traversal

Given a graph $G = (V, E)$ and source-node v_0, a BFS traversal is performed by visiting every node in the connected component of v_0 such that for any two nodes $v_i, v_j \in V$, v_i is visited before v_j if $depth(v_i) < depth(v_j)$, where:

- $v_0 \in V$ denotes the source-node
- $depth(v)$ denotes the function $depth : V \to \mathbb{N}$. Given a node $v \in V$, $depth(v)$ is the number of edges in the shortest path between node v and source-node v_0. If no path exists between v_0 and v we define the path as infinitely long: $\neg path(v_0, v) \to depth(v) = \infty$

In order to ease reasoning about BFS properties, we introduce the following definitions for two nodes, v and u:

Definition 2.5.1 (Neighbor)**.** u is a neighbor of v if there exists an edge $(v, u) \in E$.

Definition 2.5.2 (Parent)**.** v is a valid parent of u if u is a neighbor of v, and $depth(v) = depth(u) - 1$.

Definition 2.5.3 (Child)**.** u is a valid child of v if v is a neighbor of u, and $depth(v) = depth(u) - 1$.

2.6 The Parent-tracking BFS Problem

BFS by itself is only a specification of a traversal order, but with small modifications, the algorithm can be used to solve many problems. Examples include calculating reachability [27], detecting cycles [3], finding shortest paths [2], and many more [4]–[6]. In this thesis, BFS will be used to track the parent node for each visited node. For most graphs, there are multiple solutions to this problem since for a node $v \in V$, any neighbor v_n of v where $depth(v_n) = depth(v) - 1$ is a valid parent of v. Similar to how the Berkeley Graph Algorithm Platform (GAP) [28] defines the BFS problem, starting from a source-node v_0 we define a correct solution to BFS as a *parent array* p satisfying:

$\forall u, v \in V \times V$:

- $p(v_0) = v_0$
- $(p(v) = u \wedge v \neq v_0) \rightarrow (v, u) \in E$
- $(p(v) = u \wedge v \neq v_0) \rightarrow depth(v) = depth(u) + 1$
- $\neg path(v_0, v) \rightarrow p(v) = -1$

2.7 Level-synchronous BFS

Many modern parallel BFS implementations use an approach called *level-synchronous* (LS) [7], [29]–[31], and is formalized as pseudo-code in Algorithm 1. Specifically, it shows a parent-tracking version of LS BFS, but the LS approach is not limited to it and can solve other BFS problems with small modifications. The LS traversal begins at a node referred to as the *source-node*. It is the initial *frontier* to be explored in the graph, which can be seen at line 2. A frontier denotes a set of unexplored nodes with equal depth to the source-node, i.e., a *level*. The algorithm also maintains a set `next` which unexplored neighbors of the nodes in the frontier are added to. When the frontier has been fully explored, the next-set becomes the frontier, and a new next-set is initialized.

The *top-down* approach is used to assign possible children to each node in the frontier, and can be seen at line 10. If a child has not been assigned a parent, then the node in the frontier is set as parent and said child is added to the next frontier. Each node in the frontier can be completed in parallel, but exploration of the next frontier is continued only after every node in the frontier is completed; this is why it is called

level-synchronous. This step is repeated until the frontier is empty, which happens when the source-node's connected component is fully explored.

Algorithm 1 Level-synchronous BFS with a top-down approach

```
1: function BreadthFirstSearch(nodes, sourceNode)
2:     frontier ← {sourceNode}
3:     next ← ∅
4:     parents ← [-1,-1,...,-1]
5:     while frontier ≠ ∅ do
6:         TopDown(nodes, frontier, next, parents)
7:         frontier ← next
8:         next ← ∅
9:     return parents
10: function TopDown(nodes, frontier, next, parents)
11:     for node ∈ frontier do in parallel
12:         for neighbor ∈ neighbors[node] do
13:             parent ← parents[neighbor]
14:             if parent = -1 then
15:                 parents[neighbor] ← node
16:                 next ← next ∪ {neighbor}
```

Since there is a synchronization barrier between levels, a large frontier entails that more work can be performed before synchronization requirements arise. Consequently, the LS approach spends less time synchronizing threads on dense graphs, where the frontier grows fast. In contrast, it spends relatively more time synchronizing threads on sparse graphs, where each frontier remains small.

2.8 Direction-Optimizing BFS

Direction-Optimizing (DO) is a highly performant implementation of the LS approach [7]. DO builds upon the concepts of LS but adds various novel contributions. Essentially, the DO approach introduces another strategy called the *bottom-up* approach, which can be seen in Algorithm 2. The bottom-up approach examines the neighbors of every unexplored node, looking for *any* parent in the frontier. As a result, each node does not necessarily need to explore all of its neighbors, since the search can stop as soon as a valid parent has been found.

Algorithm 2 Bottom-up approach

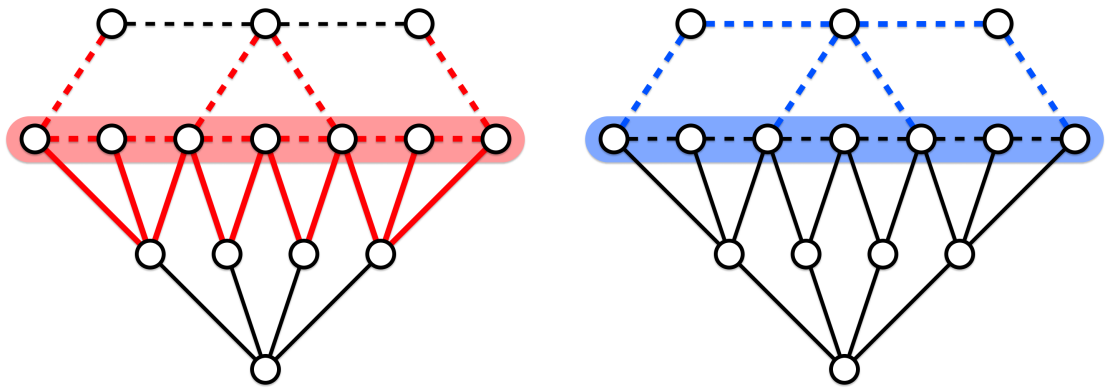
```
1: function BottomUp(nodes, frontier, next, parents)
2:     for node ∈ nodes do in parallel
3:         if parents[node] = -1 then
4:             for neighbor ∈ neighbors[node] do
5:                 if neighbor ∈ frontier then
6:                     parents[node] ← neighbor
7:                     next ← next ∪ {node}
8:                     break
```

Furthermore, DO alternates between the top-down and bottom-up approach during runtime by adopting a heuristic, which can be seen in Algorithm 3. This heuristic, denoted as **`select`** at line 6, typically selects the top-down approach when the frontier is small and the bottom-up approach when the frontier is large. This a simplification of the heuristic but represents the main idea; the exact details can be found in [7].

Algorithm 3 Direction-Optimizing BFS

```
1:  function BreadthFirstSearch(nodes, sourceNode)
2:      frontier ← {sourceNode}
3:      next ← ∅
4:      parents ← [-1,-1,...,-1]
5:      while frontier ≠ ∅ do
6:          Approach ← select TopDown | BottomUp
7:          Approach(nodes, frontier, next, parents)
8:          frontier ← next
9:          next ← ∅
10:     return parents
```

The benefit of shifting between the top-down and bottom-up approach is the reduction in number of edges looked at; this reduction can be seen in Figure 2.4. In the figure, the LS BFS algorithm has advanced to the frontier highlighted in red and blue. In the top-down approach (Figure 2.4a) it would have to look at 20 edges, whereas in the bottom-up approach (Figure 2.4b) it would have to look at 6 edges in the worst case. Possibly, it will look at less than 6 edges due to the bottom-up implementation (Algorithm 2, line 8), where an unexplored node can stop searching its neighbors when a suitable parent is found.



(a) Nodes in frontier highlighted in red. Edges to be explored in a top-down step colored red.

(b) Nodes in frontier highlighted in blue. Edges to be explored in a bottom-up step colored blue.

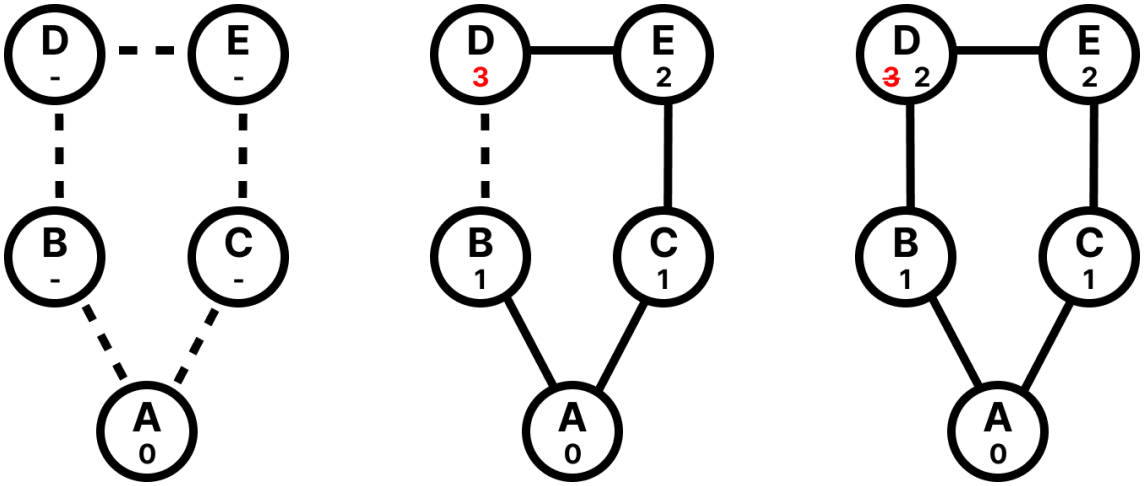
Figure 2.4: Comparison of level-synchronous BFS using top-down and bottom-up steps. Traversal has advanced to highlighted frontier. Explored and unexplored edges are shown as solid and dashed lines, respectively.

However, the DO approach still suffers from two issues. Firstly, it experiences high synchronization requirements on sparse graphs, due to previously mentioned small frontier sizes. Secondly, the heuristic picks the top-down approach over the bottom-up at small frontier sizes. Frontiers on sparse graphs are generally small which means the bottom-up approach will seldom be used.

2.9 Unordered BFS

Another concurrent approach, apart from LS discussed in Section 2.7, is *unordered* BFS. Similar to the LS approach, it starts traversal from the source-node, and continues by visiting the neighbors of its visited nodes. It differentiates itself by not restricting node traversal to levels, but rather an entirely unrestricted order. This alleviation eliminates the synchronization requirements between levels.

Typically, unordered BFS uses an underlying FIFO queue to manage the traversal order of the graph. However, in a concurrent setting it is possible for two nodes $v_i, v_j \in V$ that v_j is visited before v_i despite $depth(v_j) > depth(v_i)$. Consequently, work will have to be redone in the correct order to ensure a valid solution. Similarly, using an out-of-order queue (i.e., a relaxed FIFO queue) can result in extra work, since the out-of-order semantics allow for a node v_i to be dequeued and explored before a node v_j despite $depth(v_i) > depth(v_j)$.



(a) Untraversed state of G.

(b) G where D has been mislabeled due to an out-of-order traversal.

(c) Final state of G where D have been correctly labeled.

Figure 2.5: Example of a possible execution of BFS where redoing work is necessary, due to visiting nodes at different depths simultaneously. Solid and dashed lines represent explored and unexplored edges, respectively.

2.9.1 Example

Consider a BFS depth-tracking traversal of the graph G, as seen in Figure 2.5a, using a queue initially only containing the source-node A. The starting thread will

dequeue A and explore nodes B and C, setting their depths to 1 and subsequently enqueueing them. Suppose B is dequeued by thread t_1 who is then immediately suspended by the OS. Concurrently, another thread t_2 explores C, then E, then D and sets their corresponding depths. The resulting intermediate state can be seen in Figure 2.5b, where D has an incorrect depth of 3 rather than 2. When thread t_1 resumes execution, it will look at B's neighbor D and see that D can be assigned a lesser depth of 2. D will then be enqueued as there could be succeeding nodes that have also been mislabeled. However, in this scenario only D was mislabeled. Finally, we end up with a valid solution as seen in Figure 2.5c with node D having been explored twice.

2.9.2 Correctness

Unordered BFS will always produce a solution equivalent to that of ordered BFS, as stated by Theorem 1. The intuition for why can be explained in two steps. First, nodes will continue to be (re)explored until their correct depth is assigned. Lemma 2 and 3 prove that each node will be assigned the correct depth. Second, each node will only be explored a finite number of times, which is proven by Lemma 1. Together, this results in each node eventually being assigned its correct depth.

BFS, whether unordered or not, will only ever explore the nodes in the connected component which v_0 belongs to. Any node outside of said connected component is unreachable. Therefore, we only consider an unordered BFS traversal of a graph $G = (V, E)$ consisting of a single connected component. Suppose that the traversal begins at the source-node $v_0 \in V$ and that the queue used has no ordering semantics, since that covers traversal using both relaxed and non-relaxed queues. Property 1 describes the core mechanism of unordered BFS. Each node $v \in V$ is initialized with a tentative distance of $t = \infty$, with the exception of v_0 which is initialized to $t = 0$.

Property 1. By definition, every time a node v with tentative distance t is explored, all neighbors of v with tentative distances greater than $t+1$ will be lowered to $t+1$. In turn, each updated neighbor will be enqueued and explored at a later point in time.

We introduce two functions: d, which gives the "true" depth of a node, and d', which gives the tentative distance assigned to a node by the unordered algorithm. Let $d : V \to \mathbb{N}$ be a function such that, given a node $v \in V$, returns the length of the shortest path from v_0 to v. Let $d' : V \to \mathbb{N} \cup \{\infty\}$ be a function such that, given a node $v \in V$, returns the final tentative distance t assigned to v by the unordered traversal.

Lemma 1. Every node $v \in V$ will be explored and have $d'(v) \neq \infty$ in a finite number of steps.

Proof. Let $V_i = \{v \mid v \in V,\ d(v) = i\}$ be the set of all nodes with $d = i$. Thus, $V = \bigcup_{i=0}^{n} V_i$. By Property 1, exploring each node $v_i \in V_i$ results in each neighbor $v_{i+1} \in V_{i+1}, (v_i, v_{i+1}) \in E$ being assigned a non-infinite tentative distance. Each node is enqueued for exploration when assigned a tentative depth, resulting in each

node v_{i+1} being explored. Since $V_0 = \{v_0\}$, exploring v_0 results in each neighbor $v_1 \in V_1$ being explored and assigned a non-infinite tentative distance. Thus by induction, exploring V_0 results in all V_k with $k > 0$ eventually being explored as well.

By Property 1, a step is only taken when a node is enqueued and assigned a non-infinite tentative distance. A node is only assigned a tentative distance when said distance can be reduced, and this reduction is always an integer. Thus, there must be only a finite number of steps since the distance can only be lowered a finite number of times. As established, exploring v_0 results in each v being explored. Thus, each v will have $d'(v) \neq \infty$ in a finite number of steps. □

Lemma 2. For each node $v \in V$, $d'(v) \leq d(v)$.

Proof. Suppose that there exists a node a with a tentative distance $d'(a)$ greater than its real depth $d(a)$, i.e., $d'(a) > d(a)$. Then there must be a path from v_0 to a of length $d(a)$, containing two consecutive nodes $b,\ c$ (thereby neighbors), such that $d(b) + 1 = d(c)$ but $d'(b) + 1 < d'(c)$, since $d'(a) > d(a)$. However, from Property 1, $d'(c)$ must have been updated to at most $d'(b) + 1$ the last time b was explored. Therefore, no such $b,\ c$ can exist along the path from v_0 to a, and thus $d'(a) \leq d(a)$. □

Lemma 3. For each node $v \in V$, $d'(v) \geq d(v)$.

Proof. Property 1 gives that if a node $v \in V$ has $d'(v) \neq \infty$, then there exists a path from v_0 to v. Suppose then that there exists a v with $d'(v) < d(v)$. Since $d(v)$ is the length of shortest path by definition, this is a contradiction. Thus, $d'(v) \geq d(v)$. □

Theorem 1. Unordered BFS will always produce a solution equivalent to that of ordered BFS.

Proof. From Lemma 1 there is only a finite number of steps and thus the algorithm will terminate. After termination, each $v \in V$ will have $d'(v) \leq d(v)$ as proven by Lemma 2. Lemma 3 shows that each v must have $d'(v) \geq d(v)$. Thus, $d'(v) = d(v)$ since $d'(v) \geq d(v) \wedge d'(v) \leq d(v) \rightarrow d'(v) = d(v)$. Hence, the unordered BFS algorithm gives an equivalent solution to ordered BFS since each node is given the correct depth in a finite number of steps. □

2.10 Related Work

Unordered BFS is not a novel idea although it has not been thoroughly explored in the literature. In 2011, Hassaan et. al [8] published their work on multiple irregular problems, thereamong unordered BFS. On a graph comprised of approximately 8 million nodes and 74 million edges, their unordered and ordered (level-synchronous) algorithms demonstrated speedup factors of 22 and 27, respectively, compared to a sequential baseline. Although the ordered version performs better, this can possibly be attributed to the benchmarked graph being more suitable for a level-synchronous

workload. As discussed previously, level-synchronous approaches work very well on dense graphs such as the one used in their benchmarks. Additionally, the unordered algorithm is implemented using a concurrent FIFO queue, although it is not specified what queue algorithm is being used. Given the paper's date of release, it is likely that the queue algorithm is suboptimal by modern standards.

The authors of the *d*-CBO queue, von Geijer et al. [12], also implement an unordered BFS algorithm. Their benchmarks cover multiple sparse graphs of various sizes and compare the performance of their BFS algorithm for different queue implementations such as the FAAArrayQueue [11], LCRQ [20], *d*-RA [13], and *d*-CBO [12] variants. Generally the *d*-CBO queue performs the best at high thread counts, particularly using the FAAArrayQueue as the sub-queue. However, this benchmark is mostly a measure of the respective queues' performance in a real-world scenario. As such, it does not make any comparisons to state-of-the-art BFS methods or whether a queue-based approach is competitive at all. However, we can conclude that if a queue-based approach is to be used, then a relaxed queue is probably the best option.

Other relaxed queues such as the *d-RA* queue [13] or the 2D queue [32] are also possible candidates for usage in unordered BFS. However, the *d*-RA queue provides no bound on ordering errors and is susceptible to said errors increasing when the queue grows [12]. According to the synthetic benchmarks in [12], the *d*-CBO queue also displays about 8 times as high throughput compared to the *d*-RA queue in a highly concurrent context. The 2D queue provides a deterministic bound on rank errors which can be tuned but there is a trade-off between performance and rank errors. The 2D queue does not scale as well as the *d*-CBO queue at high thread counts [12]. However, on very sparse graphs it might be a viable choice, although it will not be investigated in this thesis.

Another implementation of an unordered BFS algorithm was made by Postkinova et al. in the paper *Multi-Queues Can Be State-of-the-Art Priority Schedulers* [33]. They present the *Stealing Multi-Queue* (SMQ) which is a derivation of the Multi-Queue (MQ) by Williams et al. [26], both of which are relaxed priority queues. Their BFS benchmarks only measure speedups relative to the MQ. Similar to the benchmarks in [12], it is hard to gauge the effectiveness of the approach compared to state-of-the-art methods.

In conclusion, the absolute performance of unordered BFS algorithms is unexplored in the literature, particularly ones utilizing relaxed data structures. Extrapolating from the research by Hasaan et al. [8]. and von Geijer et al. [12], it seems likely that combining an unordered BFS approach with a fast relaxed FIFO queue could perform well in specific scenarios.

3

Algorithm

In this chapter, we describe our unordered BFS algorithm and the optimizations we applied to improve its performance. This algorithm can be used with any concurrent queue, such as the MS queue, FAAArrayQueue, LCRQ, or the d-CBO queue with a valid sub-queue. The optimizations include a common strategy known as *batching*, as well as two novel strategies, which we introduce as *Dequeue-and-Decide* and *Bounce-on-Depth*.

3.1 Base Algorithm

The specific problem solved by the algorithm is the parent-tracking problem described in Section 2.6. In contrast to a level-synchronous algorithm like DO, the unordered algorithm needs to keep track of not only the parent of each node, but also its depth during the traversal. This is necessary since if a node v has been assigned a non-minimal depth, then a neighboring node u must be able to determine if u should become the parent of v instead. However, since the algorithm is lock-free, this means that we need to update two dependent values (parent and depth) synchronously. As a solution, we encode both the parent and the depth in a single 64-bit integer. This allows us to use a single CAS instruction to update both the parent and the depth at the same time. The 64-bit value can be represented as a struct of two 32-bit values, which can then be cast back into a single 64-bit value for the CAS instruction. We call such a struct `NodeData`.

Algorithm 4 is a pseudo-code implementation of the base algorithm. We refer to this algorithm as *UBFS*. During the traversal we keep track of nodes' parents and depths using the `parents` array (line 5) which maps a `NodeID` to its corresponding `NodeData`. The map behavior is achieved by storing the `Nodedata` for a `NodeID` at index = `NodeID`. Each `NodeData` is initialized with an unreachable parent (`NodeID == -1`) and the maximum representable value for the depth, as seen on line 5. The main loop (lines 7-18) continues until the traversal is finished and is executed in parallel. During traversal, we dequeue a `nodeId` (line 8), and then iterate over all its neighbors (line 11). If a shorter path to a neighbor is found (line 13) then we try to update the neighbor using a CAS operation (line 15). If it succeeds, we then enqueue the neighbor (line 16) and move on to the next neighbor. However, if the CAS operation fails, then some other thread has updated the data for that neighbor in the `parents` array between the time that we read it (either line 12 or 18) and the

CAS operation. In that scenario, the data from the `parents` array is reread and the condition on line 13 is reevaluated, since our path may still be shorter despite the update. This loop continues until the neighbor is successfully updated or we find that our path is no longer shorter than the existing one. When the main loop has finished, an array consisting only of the parents of each node is returned, i.e., the parent-array (line 19).

Algorithm 4 Unordered BFS - Base Implementation (UBFS)

```
 1: alias NodeID = int32
 2: struct NodeData = {parent: NodeID, depth: uint32}
 3: function BreadthFirstSearch(g: Graph, sourceId: NodeID)
 4:     queue ← {sourceId}
 5:     parents ← [NodeData(-1, MAX) | nodeId ∈ g]
 6:     parents[sourceId] ← NodeData(sourceId, 0)
 7:     while threads have work do in parallel
 8:         nodeId ← queue.Dequeue()
 9:         if nodeId = null then continue
10:         nodeDepth ← parents[nodeId].depth
11:         for neighborId ∈ g.neighbors(nodeId) do
12:             neighbor ← parents[neighborId]
13:             while nodeDepth + 1 < neighbor.depth do
14:                 updatedNeighbor ← NodeData(nodeId, nodeDepth + 1)
15:                 if CAS(parents[neighborId], neighbor, updatedNeighbor) then
16:                     queue.Enqueue(neighborId)
17:                     break
18:                 neighbor ← parents[neighborId]
19:     return [n.parent | n ∈ parents]
```

3.1.1 Termination Detection

Detecting termination for a level-synchronous approach is simple; when the next-set is empty the traversal is finished. For unordered BFS approaches it is more complicated since threads might observe an empty queue during the traversal. An obvious case of this is at the beginning of the traversal when some thread has dequeued the source node but has not yet enqueued any of its neighbors. During this time all other threads will observe an empty queue but should obviously not terminate yet. There are multiple ways to detect when termination should occur but the main condition is that all threads must observe an empty queue and none should be in the process of potentially producing more work.

Algorithm 5 shows how we implement termination detection. This algorithm is credited to Williams et al. [26][1]. The algorithm defines a function which returns whether the thread is finished with the traversal or not. There are two shared variables `noWorkCount` and `idleCount` which the threads increment (decrement)

[1]The actual paper does not provide the algorithm but an implementation can be found in Williams' source code: `https://github.com/marvinwilliams/multiqueue_experiments` (2025-04-08), which is subject to change.

using Fetch-and-Add (Fetch-and-Sub) instructions. If the dequeue is successful then the function immediately returns that we should not terminate (lines 4-5). If it fails then this thread has no work. We then keep trying to dequeue elements until we either succeed, in which case we return that we should not terminate, or all threads have no work (line 8 condition is true). If all threads have no work then we increment the `idleCount` and continually check that no threads have work (line 10). When all threads have become idle we can finally terminate. The loop on line 10 is necessary since a thread may enter the loop when all threads reported that they had no work, and subsequently exit the loop because another thread was successful in dequeueing an element (line 7) but had not yet decremented `noWorkCount` (line 14).

Algorithm 5 Termination Detection

```
1:  noWorkCount ← 0
2:  idleCount ← 0
3:  function ShouldTerminate(queue)
4:      if queue.Dequeue() ≠ null then
5:          return False
6:      FAA(noWorkCount)
7:      while queue.Dequeue() = null do
8:          if noWorkCount ≥ numThreads then
9:              FAA(idleCount)
10:             while noWorkCount ≥ numThreads do
11:                 if idleCount ≥ numThreads then
12:                     return True
13:             FAS(idleCount)
14:     FAS(noWorkCount)
15:     return False
```

Algorithm 5 exemplifies how to determine when threads terminate. However, actually using it in the BFS algorithm (Algorithm 4) would require a few changes to both algorithms, in particular to lines 7-9 in Algorithm 4 and not discarding the dequeued items in Algorithm 5. We omit this to simplify the structure of Algorithm 4.

3.2 Optimizations

We present three optimization techniques which can be applied to the UBFS algorithm (Algorithm 4), referred to as *batching*, *Dequeue-and-Decide* (DAD), and *Bounce-on-Depth* (BOD). The optimizations have a dependent order, i.e., BOD cannot be used without DAD, and DAD cannot be used without batching. Briefly, the intuition behind the optimizations are as follows: batching aims to enhance performance by improving memory efficiency and reducing interaction with the queue. DAD tries to improve raw processing speed of nodes by prohibiting threads from enqueueing certain batches and instead working on them themselves. BOD aims to reduce the amount of extra work performed by delaying the exploration of batches whose depth is considered too high. The efficiency of the optimizations depend on the graph topology, but generally provide significant speedups on sparse graphs.

3.2.1 Batching

Batching is the concept of enqueueing and dequeueing multiple elements from a queue at once, rather than just one. Batching is an established optimization technique and has been used in previous literature, both in BFS [8] and internally in data structures such as queues [26], [34]. Typically, BFS is a memory access-bound algorithm [35], emphasizing the importance of reducing memory accesses and increasing cache locality.

Intuitively, batching elements reduces the number of slow memory accesses due to locality and since fewer enqueue and dequeue operations have to be performed. Batching can therefore significantly improve performance, although there are several aspects to take into consideration. It is not necessarily the case that a larger batch size results in better performance. The input graph's topology can also influence the performance of a selected batch size. If exploring a batch does not produce more than one new batch of work, then parallelism can be limited. In a scenario where a batch never produces more than one new batch, the entire traversal will be performed sequentially, limiting performance. Larger batch sizes also generally increase the amount of extra work. We attribute this to the fact that exploration of nodes can be delayed as a consequence of not being enqueued before an entire batch is filled. Another potential cause is that the amount of work done locally by a thread is increased. If that work is built upon incorrect depths, then a larger amount of work is wasted.

To improve cache locality, we use an array to ensure that elements are placed contiguously in memory, potentially allowing the entire array to be loaded into the cache at once. Whether the entire array can be loaded into the cache at once depends on the *batch size*. It is therefore important to consider cache-line size, element size, and graph topology when choosing the batch size.

In our implementation of batching, we dequeue an array of `NodeID`:s (Algorithm 7, line 9), which is denoted as the *consumer-batch*. We iterate over each `NodeID` in the consumer-batch to check if they are suitable parents for any of their neighbors. If they are valid parents we do not directly enqueue the child `NodeID`, but instead add it to a new batch (Algorithm 7, line 19) called *producer-batch*. When the producer-batch is full it is enqueued into the queue, and the thread begins on an empty batch (Algorithm 7, lines 20–22). After a consumer-batch has been exhausted, the producer-batch is enqueued if there are any nodes in it, even if it is not full.

3.2.2 Dequeue-and-Decide

Dequeue-and-Decide (DAD) is a novel approach that builds upon batching. A problem with using the batching optimization naively is that when a consumer-batch has been fully explored, it is often the case that a partially full producer-batch remains, since batches are only enqueued when completely filled during the consumer-batch exploration. Thus, if the number of produced nodes is not a multiple of the batch size, then a partially full batch will remain, which will then be enqueued. These partially full batches can contain as little as a single element, which will once again

lead to more queue-accesses, while still having to maintain the overhead of using batching. To avoid this problem, DAD is used to ensure that all enqueued batches are full and allows threads to continue working locally on some batches while also trying to minimize the amount of extra work introduced as a consequence.

The idea behind the optimization is to dequeue a batch, referred to as the *backup-batch*, before the partially full producer-batch is enqueued. The depth of the nodes in the backup-batch can then be compared to those in the producer-batch. The batches will then be explored without ever enqueueing the producer-batch, beginning with the batch with the least depth. The pseudo-code for the optimization is described in Algorithm 6, disregarding the yellow colored parts.

Algorithm 6 Depth-Corrective Step

```
1:  function DepthCorrectiveStep(consumerBatch, producerBatch, backupBatch, queue)
2:      if backupBatch ≠ null then                          ▷ Explore backup
3:          consumerBatch ← backupBatch
4:          backupBatch ← null
5:          return
6:      if producerBatch.IsEmpty() then return    ▷ No producerBatch to compare with
7:      backupBatch ← queue.Dequeue()
8:      if backupBatch = null then                ▷ No backupBatch to compare with
9:          consumerBatch ← producerBatch
10:         producerBatch ← [ ]
11:         return
12:     difference ← Abs(Depth(backupBatch) - Depth(producerBatch))
13:     if Depth(backupBatch) ≥ Depth(producerBatch) then
14:         consumerBatch ← producerBatch
15:         if difference ≥ depthThreshold then                ▷ Bounce-On-Depth
16:             queue.Enqueue(backupBatch)
17:             backupBatch ← null
18:     else
19:         consumerBatch ← backupBatch
20:         backupBatch ← null
21:         if difference ≥ depthThreshold then                ▷ Bounce-On-Depth
22:             queue.Enqueue(producerBatch)
23:             producerBatch ← [ ]
```

In more detail, the DAD decision process begins with determining if the last producer-batch is empty (line 6). If it is empty this means the exploration of the consumer-batch produced perfectly filled producer-batches, and thus we continue from the beginning since we do not have a producer-batch to compare the backup-batch with. If the producer-batch is not empty we do not want to unconditionally enqueue it, but rather work on it locally. Subsequently, we try to dequeue a backup-batch (line 7). If the dequeue fails, we start exploring our partially filled producer-batch, i.e., the consumer-batch is assigned as the producer-batch. However if it succeeds, we can compare the depths of the producer-batch and the backup-batch, and explore the batch with the lowest depth (lines 13–14,18–19). Using the depth of the first node in each batch proved to work well as an approximation for the depth of the whole batch.

Assuming the backup-batch is picked, then we leave the producer-batch untouched and keep appending nodes to it. Otherwise, the producer-batch is explored and the backup-batch exploration is delayed until the producer-batch has been explored (line 2). A visualization of DAD's decision process can be seen in Figure 3.1.

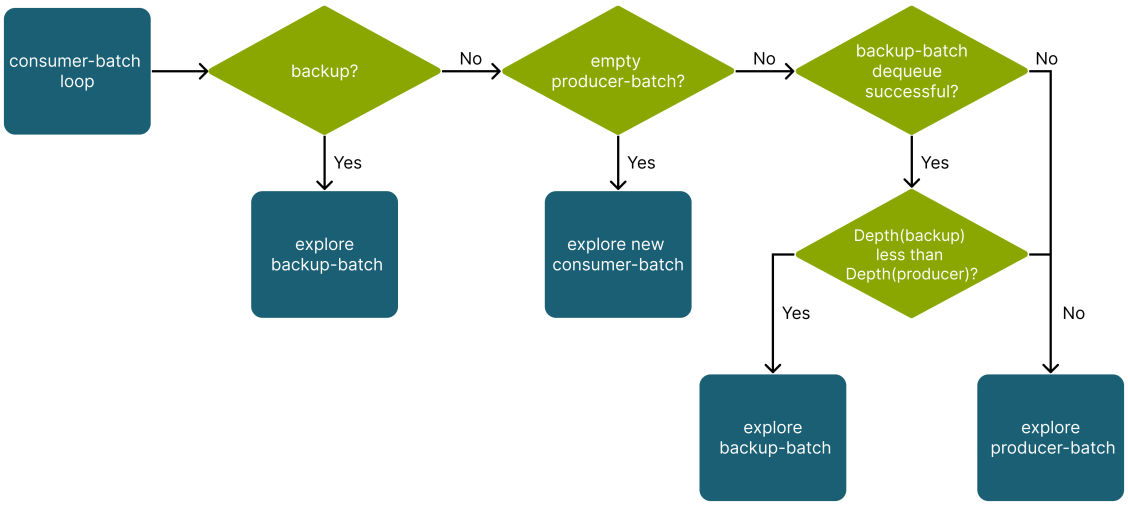


Figure 3.1: Dequeue-and-Decide decision process for choosing which batch to explore next. The entry step (consumer-batch loop) is the point where all of the nodes in the consumer-batch have been explored (Algorithm 7, lines 11–24).

By never enqueueing the remaining producer-batch after a consumer-batch has been explored, it is ensured that all enqueued batches are completely filled. This reduces the number of required queue accesses by enqueueing fewer times and providing threads with more elements to work on when dequeueing. Keeping the producer-batch also results in better cache efficiency, since the batch is likely already in the cache. Delaying a batch with greater depth reduces the probability of exploring a batch with non-minimal depths, further reducing the chance of having to redo work.

3.2.3 Bounce-on-Depth

Bounce-on-Depth (BOD) is an extension of the DAD approach. It introduces a *depth threshold* when comparing the producer-batch and backup-batch (lines 15, 21). If the difference in depth between the batches exceed the depth threshold, the batch with the higher depth is enqueued into the queue (lines 16, 22). As mentioned in Section 3.2.2, the delay in batch exploration can reduce the amount of work that has to be redone. While DAD delays this exploration by one iteration, BOD ensures that batches with depths further away than the depth threshold are evaluated much later instead. This corrective behavior has a trade-off between traversal speed (i.e., how many nodes per second are explored) and reduction in work.

Given a sufficiently high depth threshold and depending on the topology of the traversed graph, BOD's execution behavior is akin to DAD, since batches rarely are enqueued back into the queue. An extreme example of this would be a traversal where the depth threshold is larger than the graph's diameter. Since no node can be

at a greater depth from the source node than the diameter of the graph, the depth threshold will never be exceeded, resulting in behavior equivalent to DAD (albeit with some overhead). In contrast, if a relatively small depth threshold is chosen, the amount of queue interaction increases.

The BOD optimization has two notable implications. If the backup-batch is sent back into the queue, there is no need to explore it in the next iteration. However, if the producer-batch is sent to the queue, we reintroduce the occurrence of half-full batches in the queue, but for good reason this time. Again, the producer-batch is only enqueued when its depth exceeds the depth threshold. If the threshold is exceeded, then the likelihood of the depths being wrong is increased. Therefore, it is typically beneficial to delay the exploration, in order to potentially reduce the amount of extra work introduced.

3.3 Final Algorithm - DCBFS

The final algorithm uses all previously mentioned optimizations. We call this BFS algorithm *Depth-Corrective* (DC), due to the depth-corrective optimizations explained in Sections 3.2.2 and 3.2.3. The algorithm aims to balance the trade-off between work-speed and extra-work. Pseudo-code for the DC implementation can be seen in Algorithm 6. The BOD approach is highlighted in yellow, while the remainder of the code makes out the DAD approach.

The complete DC algorithm can be seen in Algorithm 7. As previously mentioned, the DC step occurs after the consumer-batch has been fully explored (line 26); the intention is to select the batch with the lower depth, in order to continue exploration while reducing the likelihood of wasting work.

Algorithm 7 Unordered BFS - Final Implementation (DC)

```
1: function BreadthFirstSearch(g: Graph, sourceId: NodeID)
2:     queue ← {[sourceId]}
3:     parents ← [NodeData(-1, MAX) | nodeId ∈ g]
4:     parents[sourceId] ← NodeData(sourceId, 0)
5:     producerBatch ← [ ]
6:     consumerBatch ← null
7:     backupBatch ← null
8:     while threads have work do in parallel
9:         if consumerBatch = null then consumerBatch ← queue.Dequeue()
10:        if consumerBatch = null then continue
11:        for nodeId ∈ consumerBatch do
12:            if nodeId = null then continue
13:            nodeDepth ← parents[nodeId].depth
14:            for neighborId ∈ g.neighbors(nodeId) do
15:                neighbor ← parents[neighborId]
16:                while nodeDepth + 1 < neighbor.depth do
17:                    updatedNeighbor ← NodeData(nodeId, nodeDepth + 1)
18:                    if CAS(parents[neighborId], neighbor, updatedNeighbor) then
19:                        producerBatch.Append(neighborId)
20:                        if producerBatch.IsFull() then
21:                            queue.Enqueue(producerBatch)
22:                            producerBatch ← [ ]
23:                        break
24:                    neighbor ← parents[neighborId]
25:        consumerBatch ← null
26:        DepthCorrectiveStep(consumerBatch, producerBatch, backupBatch, queue)
27:    return [n.parent | n ∈ parents]
```

4

Experimental Evaluation

This chapter presents benchmarks of the algorithms presented in Chapter 3, the DO algorithm [7] and a baseline sequential implementation. The experiments are run on a powerful benchmarking server. On road-network graphs and using the optimal number of threads for each algorithm, our algorithms are 3.4× to 8.0× faster than a sequential algorithm and 1.6× to 2.4× faster than DO. The performance of the DO algorithm peaks at fairly low thread counts on sparse graphs, and thus does not scale well into high threads counts. Using the maximal number of threads (512), our algorithms are 2.2× to 6.2× faster than sequential and 12.4× to 69.7× faster than DO.

4.1 Experiment Setup

To evaluate the algorithms presented in Chapter 3 and the DO algorithm, we use the Berkeley Graph Algorithm Platform Benchmark Suite (GAPBS) [28]. GAPBS is a C++ codebase that contains reference implementations of multiple state-of-the-art algorithms for different graph kernels, thereamong BFS. It also contains utilities for loading graphs and benchmarking. The specific algorithm for BFS included with GAPBS is the Direction-Optimizing (DO) algorithm [7]. We extend this codebase with our implementations, which allow us to objectively compare our algorithm to the DO algorithm. Furthermore, GAPBS is developed by the author (Scott Beamer) of the DO algorithm, and as such we expect the algorithm to be performantly implemented.

4.1.1 Evaluated Algorithms

Each algorithm is benchmarked using GAPBS and solving the parent-tracking BFS problem (Section 2.6). We perform BFS 32 times for each algorithm and average the result, unless otherwise mentioned. The different algorithms used in the experiments are shown in Table 4.1. Notably DC and BOD refer to the same algorithm. While DC is typically the most performant of our implementations we still include UBFS, Batching, and DAD in order to highlight their different characteristics. The DO_TD algorithm is a version of DO that we have scaled down to always use the top-down approach. This removes the overhead of the direction-optimizing heuristic, which is largely useless on very sparse graphs. The Sequential algorithm is a traditional BFS

algorithm, using a top-down approach and the `std::queue` queue from the C++ standard library.

Table 4.1: Algorithms used in experiments.

Abbreviation	Algorithm	Details
UBFS	Unordered BFS	Section 3.1
Batching	UBFS with batching	Section 3.2.1
DAD	UBFS with Dequeue-and-Decide	Section 3.2.2
BOD	UBFS with Bounce-on-Depth	Section 3.2.3
DC	Final algorithm (equivalent to BOD)	Section 3.3
DO	Direction-Optimizing BFS [7]	Section 2.8
DO_TD	Top-down only DO	Section 4.1.1
Sequential	Sequential BFS	Section 4.1.1

Furthermore, there are multiple parameters which can be tuned across the different algorithms and queues. We define a legend for these in Table 4.2. Unless otherwise specified, all experiments use the d-CBO queue with 64 FAAAQ [11] sub-queues and $d = 2$, i.e., 2-CBO queue. Specifically for DC and BOD, we use a depth threshold of 5. Of course, depending on the graph topology, a better threshold might exist. However, the algorithms presented make no assumptions regarding graph topology. Generally, a threshold of 5 performed well compared to both higher and lower values on sparse graphs.

Table 4.2: Symbols for the parameters of our algorithms.

Symbol	Meaning
b	Batch size
s	Number of sub-queues in d-CBO
t	Number of threads used
q	Queue algorithm used in the BFS traversal

4.1.2 Graphs

Table 4.3 shows the graphs used in our experiments, as well as their main characteristics. The graphs prefixed with "road" are real-world road networks. The hugebubbles graph is a frame of a 2D simulation. The Kronecker graph [36] is a dense, synthetic graph and was generated with 2^{24} nodes and the same generation parameters (A=0.57, B=C=0.19, D=0.05) as specified in GAP [28].

4.1.3 Machines and Experiment Setup

We run experiments on two different machines as seen in Table 4.4. The results presented are gathered from the Athena machine, unless otherwise specified. Both machines run the same environment using the OpenSUSE Tumbleweed operating

Table 4.3: Graphs used in experiments.

Graph	Nodes	Edges	Diameter	Max Degree
road-usa	24m	58m	8k	9
road-europe	51m	108m	28k	9
road-asia	12m	25m	45k	13
hugebubbles-00020	21m	64m	8k	3
Kronecker	17m	260m	6	406979

system with Linux kernel version 6.13.1. GAPBS, and therein our algorithms, are compiled with GCC version 14.2.1 and O3 optimization level. OpenMP is used for concurrency, since the DO algorithm in GAPBS is implemented with it. Hyper-threading (SMT) is used and threads are pinned using `numactl` with the following priority: core, core cluster, socket.

Table 4.4: Machines used in experiments.

Name	CPU	Sockets	Cores	RAM
Athena	AMD EPYC 9754	2	128 (2 threads each)	755 GB
Ithaca	Intel Xeon E5-2695 v4	2	18 (2 threads each)	64 GB

4.1.4 Removal of Outliers

In rare cases, the DO algorithm can have abnormally slow traversals. In the results shown in this section, if any such case occurred, those data points have simply been removed. This strictly benefits the DO algorithm in the comparisons. The reason behind the slow traversals is discussed in Section 5.1.

4.2 Elapsed Time Comparisons

The best performing configuration of each algorithm is presented in Figure 4.1, where the exact configurations are detailed in Table 4.5. Configurations with the lowest runtime are chosen, and vary in how many threads they utilize. The figure shows the relative speedup of each algorithm compared to the sequential BFS implementation. The DO_TD algorithm performs slightly better than DO on every graph. This is because the DO algorithm almost always uses the top-down approach due to the graphs' topologies, but it still evaluates the direction-optimizing heuristic to check if it should swap to a bottom-up approach. This extra work is wasted, and thus the performance difference between the two can be seen as the cost of evaluating the heuristic. Overall, the BOD algorithm outperforms every other algorithm on the presented graphs, thereamong DO and DO_TD. In the order the graphs are presented, BOD surpasses DO with speedups of 1.6×, 2.2×, 2.4×, and 1.8×, respectively.

The algorithm configurations chosen for Figure 4.2 instead shows the best performing combinations while utilizing all of the processor's threads ($t = 512$), and are

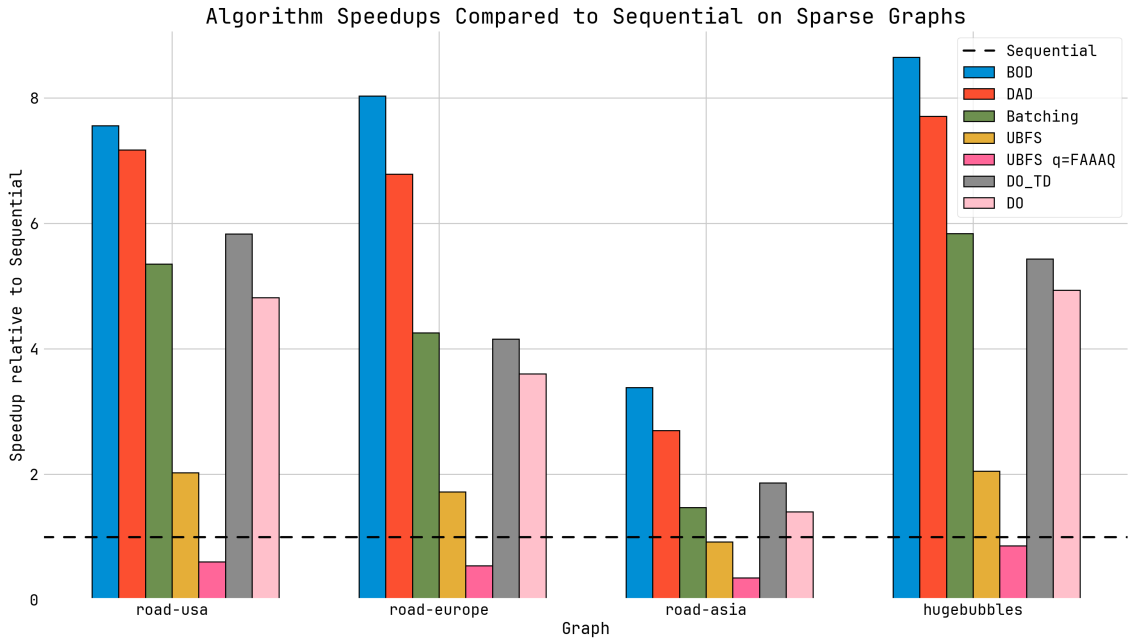


Figure 4.1: Each algorithms' best performing configuration, with specifics detailed in Table 4.5.

further detailed in Table 4.6. Notably, none of the algorithms perform the best on a maximum number of threads, and the relative speed compared to the sequential implementation is lesser. This graph also follows the trend where BOD outperforms the other algorithms, but this time with greater speedups compared to DO. In the order the graphs are presented, BOD exceeds DO with speedups of 12.4×, 22.9×, 69.8×, and 12.0×, respectively. These results make the drawbacks of using the DO algorithm on sparse graphs apparent, as the synchronization requirements inhibits processor utilization; this is especially evident for the road-asia graph, where the diameter is large (45k) and the speedup is great (69.8×).

Table 4.5: Best performing configurations for the algorithms shown in Figure 4.1.

	road-usa			**road-europe**			**road-asia**			**hugebubbles**		
Algorithm	t	s	b	t	s	b	t	s	b	t	s	b
Sequential	1	-	-	1	-	-	1	-	-	1	-	-
BOD	128	64	32	128	64	64	64	64	32	256	64	16
DAD	256	64	16	256	64	16	128	64	16	256	64	16
Batching	256	64	32	256	64	32	64	64	32	256	64	32
UBFS	256	128	-	256	128	-	128	64	-	256	128	-
UBFS q=FAAAQ	64	-	-	64	-	-	64	-	-	64	-	-
DO_TD	16	-	-	16	-	-	16	-	-	16	-	-
DO	16	-	-	16	-	-	16	-	-	16	-	-

Both Figure 4.1 and 4.2 display a behavior where the most performant algorithms are BOD, DAD, and batching, in that order. However, this is not the case for the

benchmarks on the dense Kronecker graph. Although BOD is the best performing algorithm among those three, it was still outperformed by DO with a 13.1× speedup, which is expected since BOD is optimized for sparse graphs. The Kronecker results were omitted from the figures due to scaling issues distorting the representation.

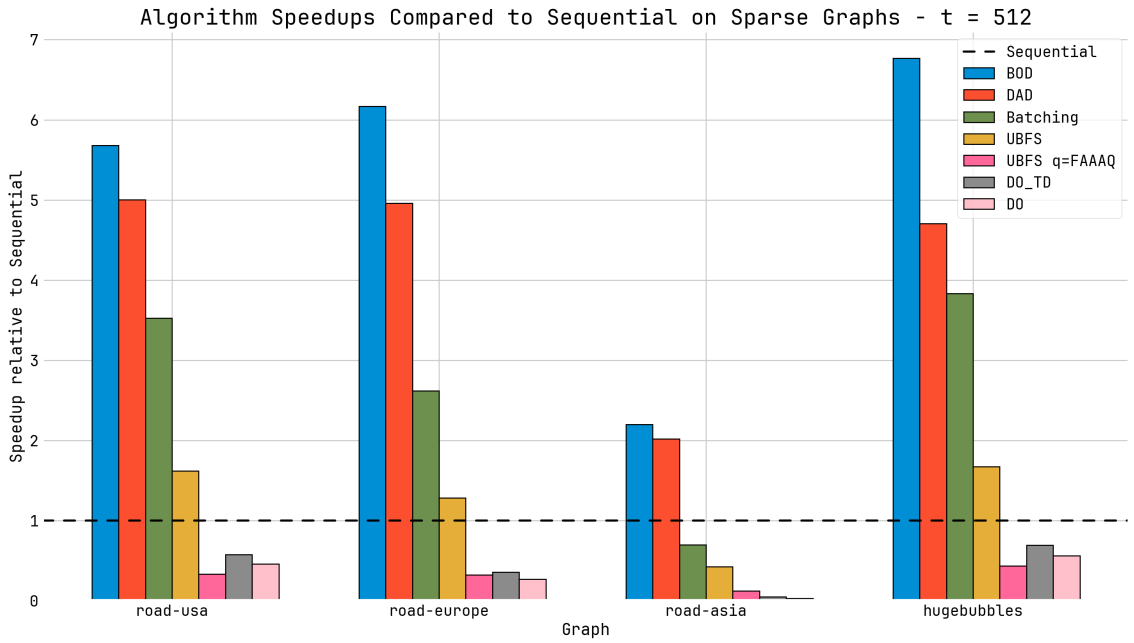


Figure 4.2: Each algorithms' best performing configuration on a maximum number of threads, with specifics detailed in Table 4.6.

Table 4.6: Best performing configurations on maximum number of threads ($t = 512$) for the algorithms shown in Figure 4.2. Algorithms whose only configurable parameter is thread count have been omitted from this table.

	road-usa		**road-europe**		**road-asia**		**hugebubbles**	
Algorithm	s	b	s	b	s	b	s	b
BOD	64	16	128	32	128	32	64	16
DAD	64	16	64	16	128	16	64	16
Batching	64	16	64	32	128	32	64	32
UBFS	128	-	128	-	64	-	128	-

4.3 Algorithm Scalability

Figure 4.3 shows the scalability of the algorithms using the same configuration as specified in Table 4.5, but over several thread configurations. For the unordered algorithms, the BOD algorithm performs best on all thread counts greater than one. The DO and DO_TD algorithms quickly reach their maximum performance at 16 threads. Beyond 16 threads their performance generally degrades, eventually

performing worse than a sequential algorithm. For the UBFS algorithm, using the d-CBO queue generally does not outperform the strict FAAAQ queue until 64 threads are used.

No algorithm scales positively over 256 threads and in some cases performs significantly worse. Only one socket is used at 256 threads or lower; after that point the second socket is used as well. It is not uncommon that introducing another socket degrades performance [37], [38]. However, performance typically peaks earlier than 256 threads, as seen for the road graphs. Although not visible in the figure, the work speed (i.e., number of nodes visited per second) strictly increases until 256 threads for the unordered algorithms using a relaxed queue. The reason why performance degrades regardless is due to extra work being introduced at a faster pace than work speed. For the UBFS q=FAAAQ algorithm, this is not the case due to the limited scalability of the queue, that is, the amount of extra work increases but the work speed does not.

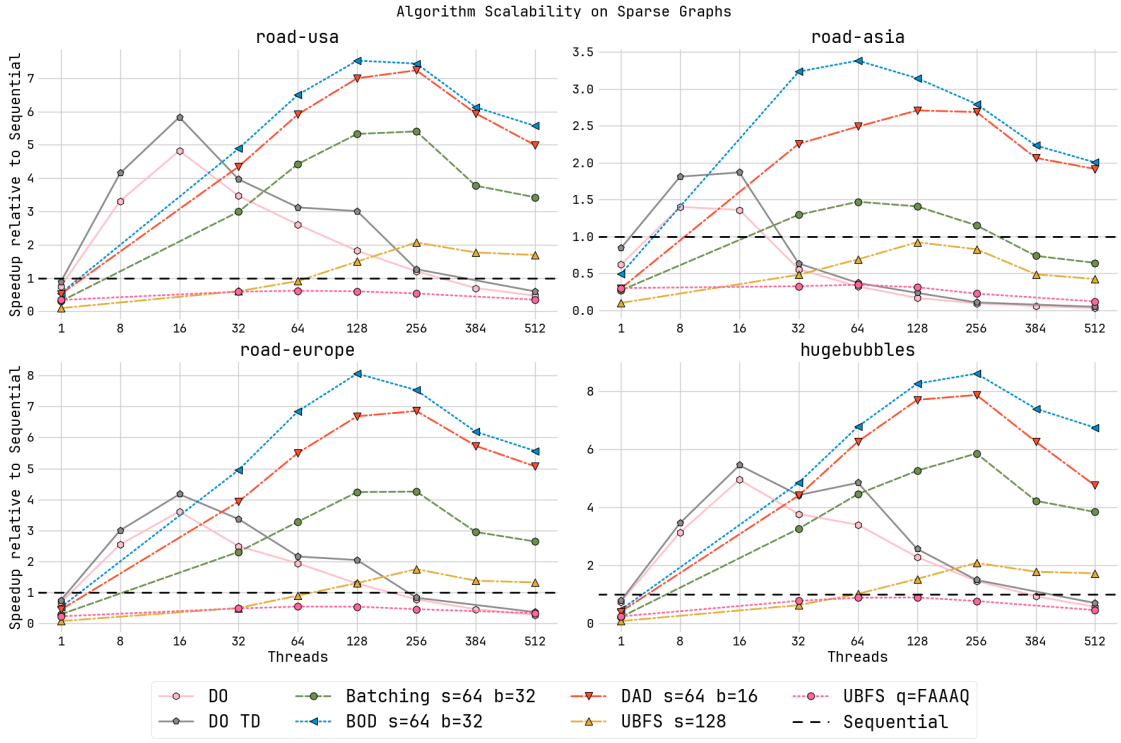


Figure 4.3: Scalability of algorithms using the same configuration as specified in Table 4.5, excluding the number of threads. Only DO and DO_TD have additional data points at 8 and 16 threads due to their performance peaking earlier.

4.4 Elapsed Time Distributions

The extra work that is introduced by unordered BFS or using an underlying relaxed queue can vary and influence the algorithms consistency. The following Figures displays the elapsed time distributions on sparse graphs. Figure 4.4 shows the best performing algorithm-combinations with $b = 32$, and Figure 4.5 shows equivalent combinations, but on a maximum number of threads.

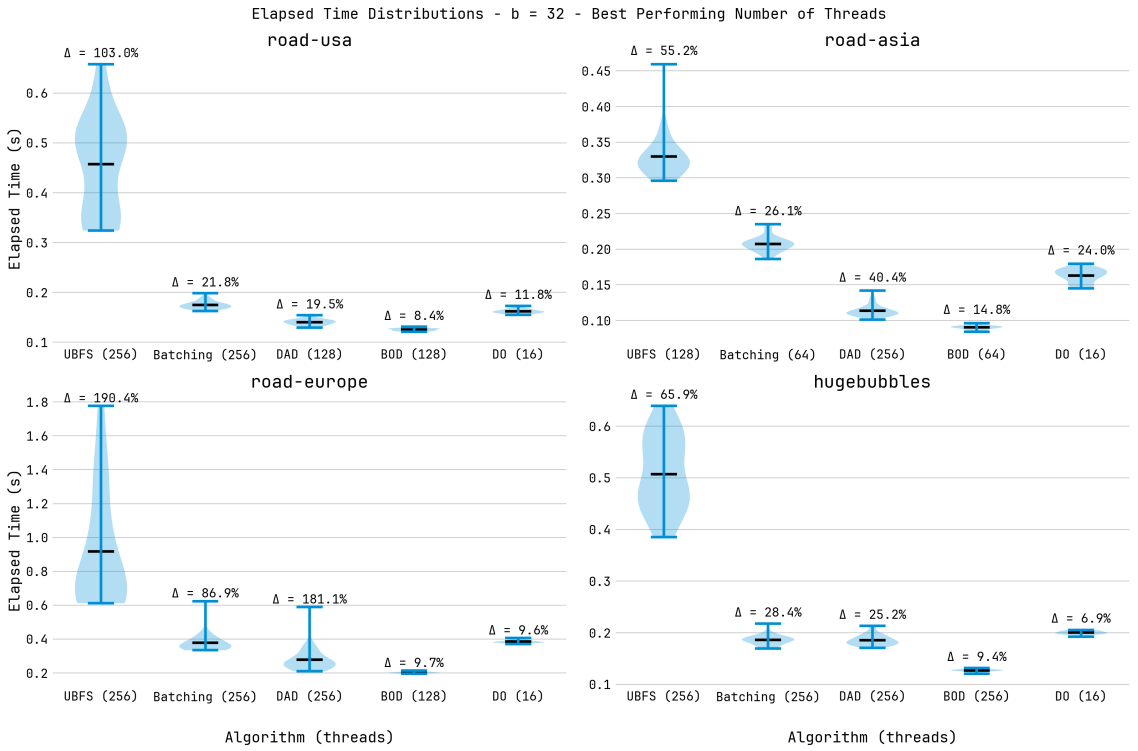


Figure 4.4: Elapsed time distributions on road-networks and the hugebubbles-graph. All algorithms are chosen by their best performing number of threads and use $b = 32$, where applicable. Black bars show the mean elapsed times, and blue bars show the extreme elapsed times. Delta percentages (Δ) represent the difference in elapsed time extreme values.

The reduction in difference between UBFS and batching can be explained by batching's implicit sequentiality (explained in Section 5.2.1). This affects DAD and BOD as well, since both of them use batching. Moreover, the final algorithm using BOD keeps a relatively small difference compared to DAD. This associates to the added corrective behavior BOD has over DAD. DAD increases throughput by interacting less with the queue; this greatly benefits runs with less extra-work, but conversely creates potential runs with greater extra-work, diverging the two and increasing the difference. BOD minimizes the difference by interacting with the queue to enqueue batches with a depth greater than the threshold, regardless if they had correct depths or not.

When comparing DO on its maximum number of threads (Figure 4.5) and best performing number of threads (Figure 4.4), it is apparent that an increase in thread

count increases elapsed time and difference. This can be attributed to one main reason. A higher thread count requires more threads to synchronize between each level. This also raises the probability that the operating system suspends any of the threads that are being used for another task, further slowing down the synchronization of all threads. And even though all of these threads require managing, most of the threads will not be utilized simultaneously; this is due to the frontiers of these sparse graphs containing fewer nodes than available threads for the most part.

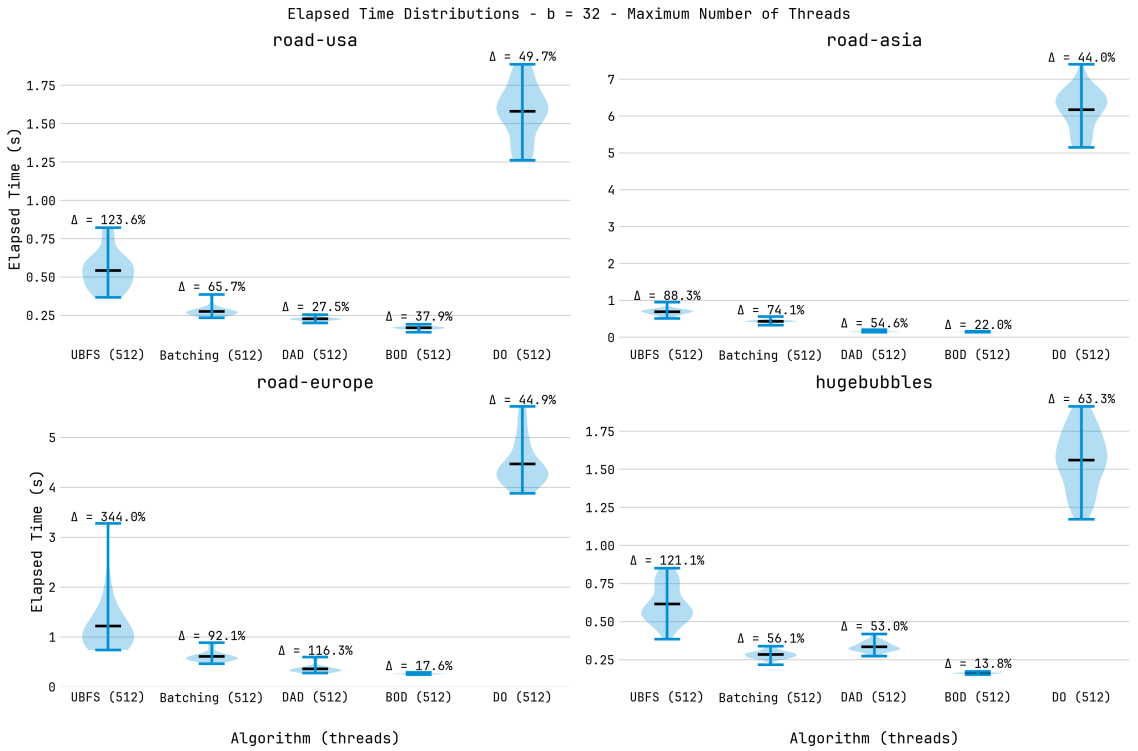


Figure 4.5: Elapsed time distributions on road-networks and the hugebubbles-graph. All algorithms run with $t = 512$, and $b = 32$ where applicable. Black bars show the mean elapsed times, and blue bars show the extreme elapsed times. Delta percentages (Δ) represent the difference in elapsed time extreme values.

4.5 Work Efficiency

The unordered approaches can vary greatly in how much extra work is produced. The amount of work an algorithm performs can be quantified by the number of nodes it visits. For a sequential or level-synchronous algorithm, the number of nodes visited will always be equal to the number of nodes in the connected component to which the source node belongs. Thus, the number of nodes visited is always the same for those algorithms. The size of the connected component defines a baseline, that is, the minimum number of nodes that must be visited to complete a BFS traversal. *Work inefficiency* is a factor indicating how much work an algorithm performs compared to baseline. A work inefficiency of $c\times$ means the algorithm performs c times as much work as baseline. For example, a work inefficiency of $1\times$ means that the

algorithm performs no more work than baseline, while a work inefficiency of 2× means it performs twice as much work as baseline.

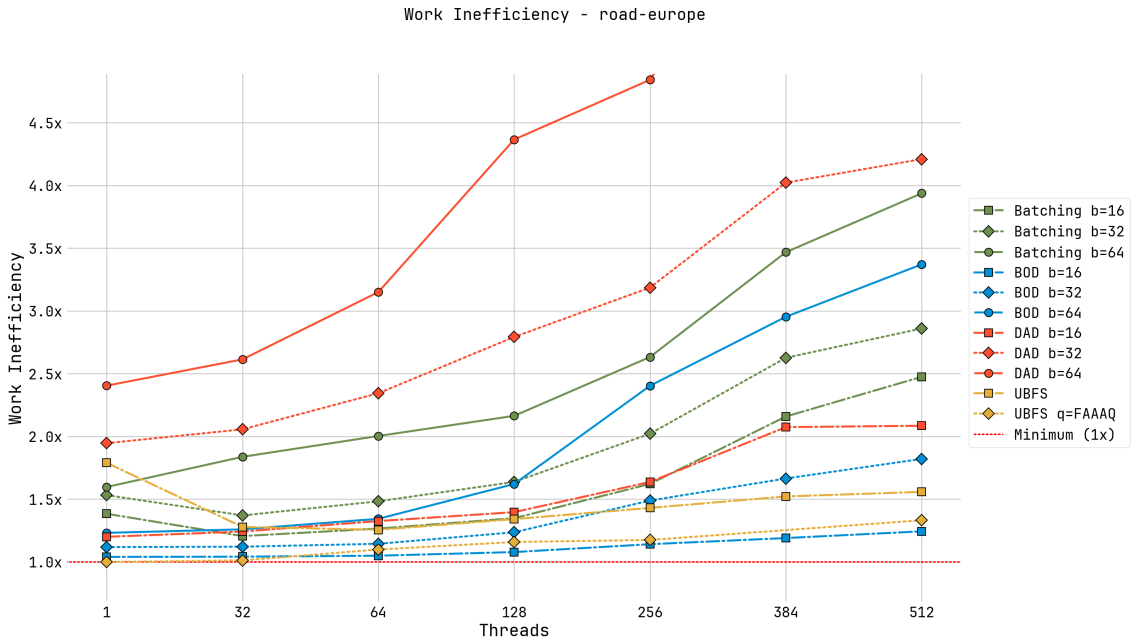


Figure 4.6: Work inefficiency of unordered algorithms. Two additional data points for algorithm DAD $b = 64$ are out of bounds. At 384 and 512 threads, its work inefficiency is 6.5× and 7.5×, respectively.

Figure 4.6 shows the work inefficiency of each algorithm on the road-europe graph. In almost all cases, the work inefficiency increases in correlation with the number of threads. A decrease in work inefficiency can be seen for some algorithms when using 32 threads instead of a single one. We suspect that this is due to multiple threads giving a load-balancing effect during the traversal, compared to when only a single thread is used. As seen in the figure, larger batches significantly increase work inefficiency. When comparing the two UBFS algorithms, there is a clear increase in work inefficiency when using the d-CBO queue instead of the FAAAQ. Notably, the BOD optimization achieves a lower work inefficiency than both of them, despite using both a relaxed queue and batching.

Figure 4.6 gives a clear overview of how the work inefficiency generally increases with the number of threads and batch size used. Figure 4.7 instead shows the distribution of the work inefficiency for each algorithm. Again, a higher thread count and a larger batch size typically results in greater work inefficiency. The DAD algorithm in particular has a very large range, ranging from almost baseline (1×) to 12×. In contrast, the BOD algorithm is typically very consistent. This mirrors the behavior of the elapsed time distributions presented in Section 4.4, to an extent. However, the despite DAD being less work efficient in most cases, it also has a higher *processing rate*, i.e., how many nodes it is able to visit per second.

Figure 4.8 shows the relationship between work inefficiency and processing rate, the result of which is the execution time. Processing rate is equivalent to the number

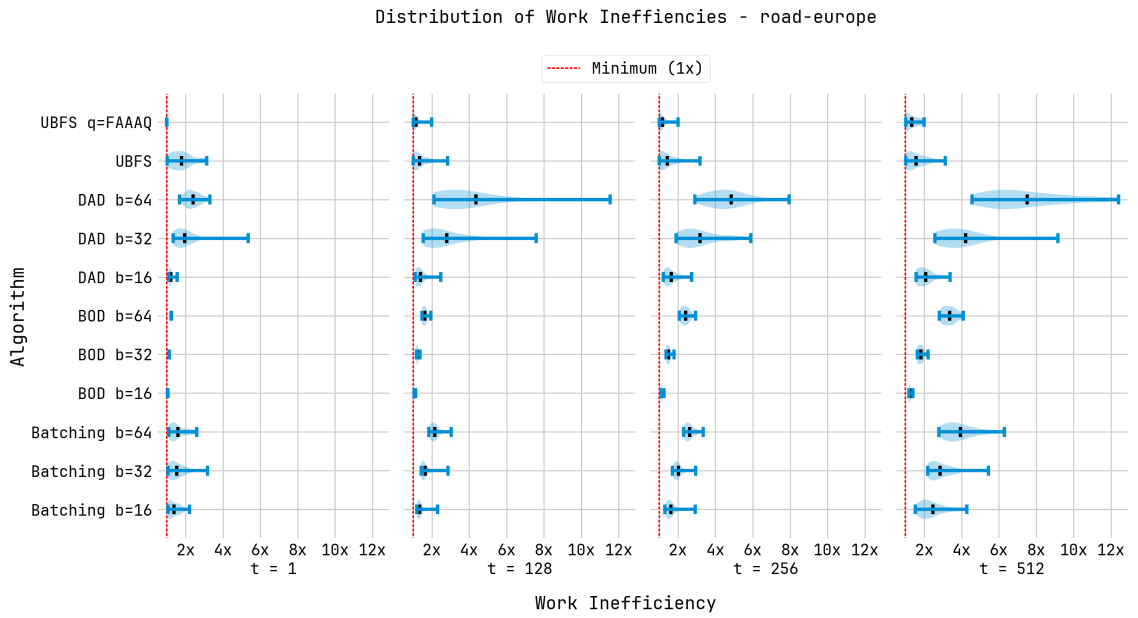


Figure 4.7: Distributions of work inefficiencies for unordered algorithms on the road-europe graph.

of nodes visited per time unit, e.g., the number of nodes visited per second. The processing rate is then described as a factor relative to the sequential algorithm's processing rate. This figure also includes configurations using 128 sub-queues for the d-CBO queue, as opposed to only 64. While the work inefficiency increases with a higher number of sub-queues, so does the processing rate. The same behavior applies to batch sizes. The fastest algorithm (H), has a processing rate that is 14 times faster than the sequential version, and performs about twice as much work, resulting in a speedup factor of 6.5. In contrast, the algorithm with the highest processing rate (O), has a processing rate that is 30 times faster than sequential, but performs 8 times as much work. As a result, it is almost twice as slow as H. Generally, the DAD algorithm processes nodes slightly below two times as fast as the equivalent BOD algorithm, but is slightly above two times as work inefficient. As a result, the BOD algorithm performs better. In addition, the BOD algorithm is more consistent.

The behavior is much different for dense graphs, as seen for the Kronecker graph in Figure 4.9. The behavior of the different algorithms is much more homogeneous compared to their behavior in Figure 4.8. Additionally, the work inefficiency and processing rate is much lower, compared to the sequential algorithm. The algorithms, disregarding U and V, form three clusters, all with roughly the same processing rate, judging by their color. The clusters correlate with the batch size used by the algorithms, where the work inefficiency increases when the batch size does. Since the processing rate is much the same, the performance of the algorithms degrades as the batch size increases.

The reason for this behavior is likely that the Kronecker graph has a much higher average degree compared to the road networks. The average degree for road net-

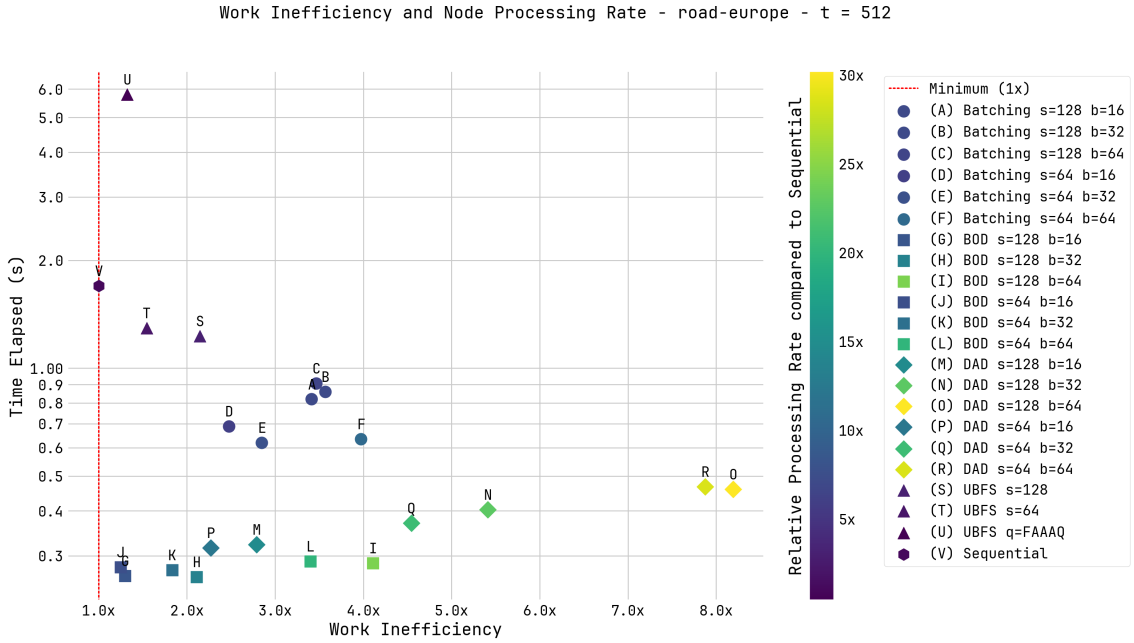


Figure 4.8: Work inefficiencies and processing rates for unordered algorithms, compared to a sequential algorithm on the road-europe graph using 512 threads.

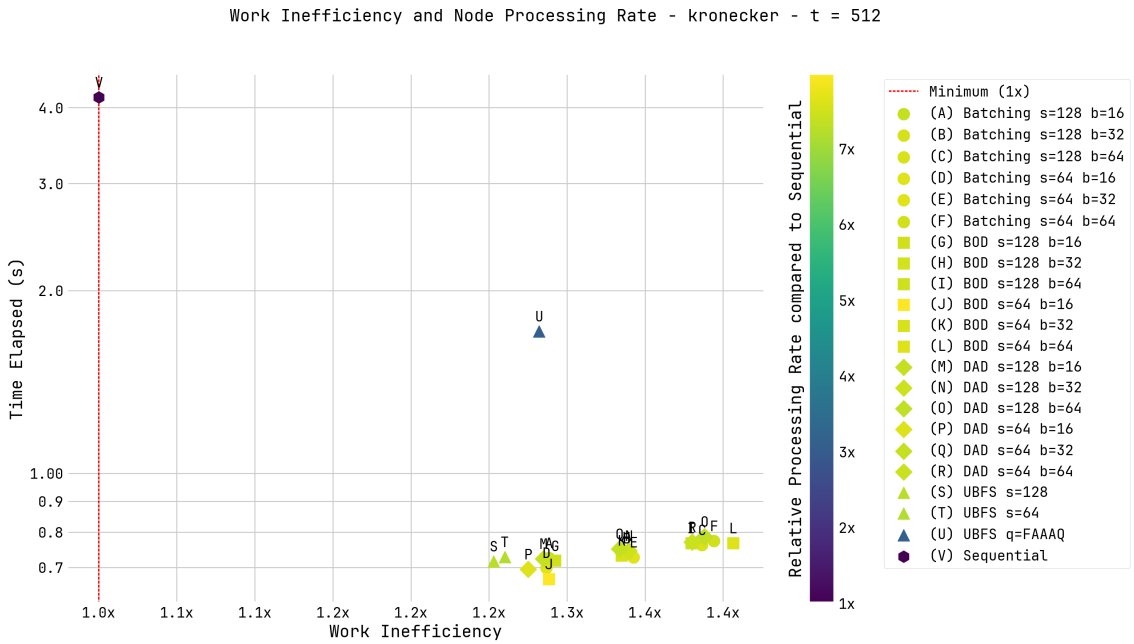


Figure 4.9: Work inefficiencies and processing rates for unordered algorithms, compared to a sequential algorithm on the road-europe graph using 512 threads.

works is typically around 2, rather than 16 for the Kronecker graph. Thus, on average, every dequeued node results in visiting 16 neighbors, rather than only 2. Intuitively, this produces an effect similar to batching, since for each dequeue, there is a significant amount of work available. This implicit batching behavior therefore nullifies the benefits of the batching optimization, to an extent. Yet, the cost of a higher work inefficiency must still be paid, resulting in the performance decreasing with larger batch sizes. This is evident from the figure, where the UBFS algorithms S and T perform very similarly to the other algorithms, with the exception of U, which uses the FAAAQ rather than the d-CBO queue.

The BOD optimization is also largely ineffective for this type of graph. Since the graph's diameter is roughly equal to the depth threshold, the depth-corrective behavior is very unlikely to be triggered. In fact, unlike on sparse graphs, the algorithm with the highest work inefficiency is one of the BOD configuration (L). The UBFS configuration using the FAAAQ (U) is also more than twice as fast as the sequential algorithm (V). This is a significant difference compared to its corresponding performance on the road-europe graph, where U is roughly three times slower than V, as seen in Figure 4.8. Again, this is likely due to the larger amount of work per node dequeued, resulting in less contention and better performance of the queue.

4.6 Queue Size

Figure 4.10 shows the size of the d-CBO queue during the BFS traversal. The algorithms used are the DC algorithm (solid curves) with different batch sizes and the UBFS algorithm (dashed curves). Both algorithms use 64 FAAAQs [11] as sub-queues. The data was gathered on the Ithaca machine using the maximum number of threads (72). As expected, the number of elements in the queue at any given moment scales inversely with an increased batch size. For the DC algorithm the number of elements per queue is very low, even falling below $s = 64$ for some configurations. In practice, few elements in the d-CBO queue result in the loss of its ordering semantics.

The small queue sizes raise the question whether any ordering semantics is necessary at all for the DC algorithm to perform well on sparse graphs. Despite the lack of ordering semantics, the depth-corrective property of the BOD optimization still provides significant performance increases, as seen in Section 4.2. We attribute this to the exploration of "bounced" batches still being meaningfully delayed both by the d-CBO queue and the BOD step. To begin exploring such a batch three conditions must be satisfied. First, the d-CBO queue must sample the sub-queue in which the bounced batch was added to. Second, the sampled queue with the bounced batch must have a lower *DeqCount* (Section 2.4) than the other sampled sub-queue. Third, the batch must now not exceed the threshold if dequeued as part of the BOD step, or it must be dequeued in the beginning of the loop on line 9 in Algorithm 7.

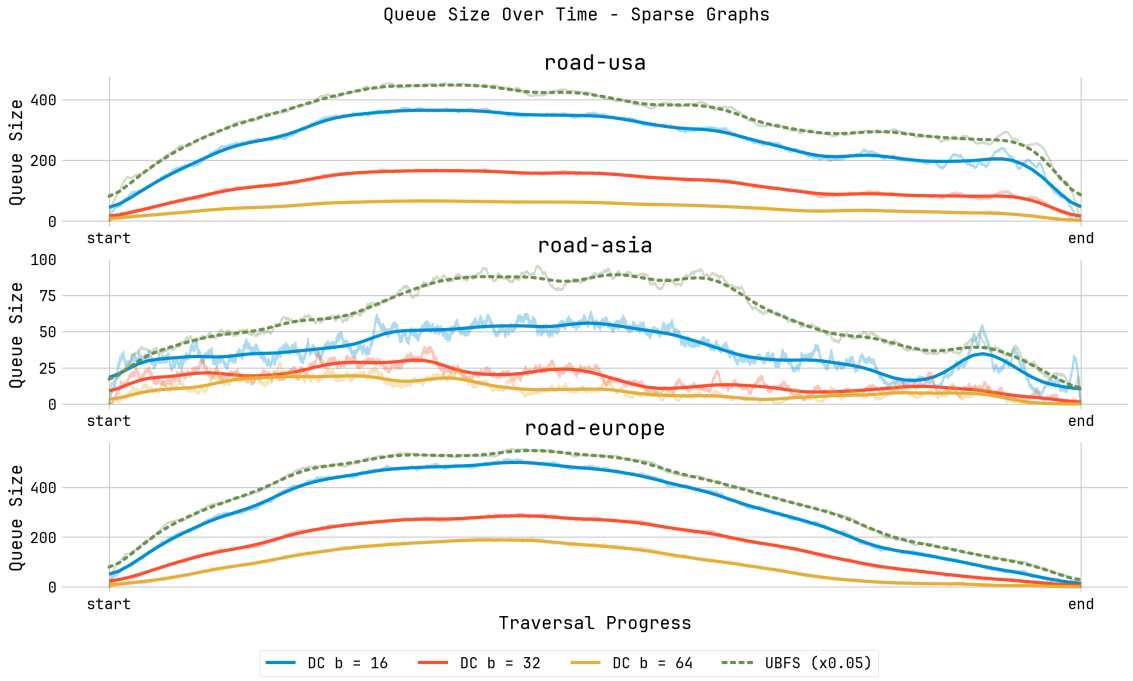


Figure 4.10: Queue sizes during traversals of road networks averaged over 32 traversals per algorithm. The results have been stretched to all have the same length. Opaque lines show smoothened results for readability, while the semi-transparent lines shows the exact values. The dotted lines (UBFS) are multiplied by a factor of 0.05. The data was gathered on Ithaca using 72 threads.

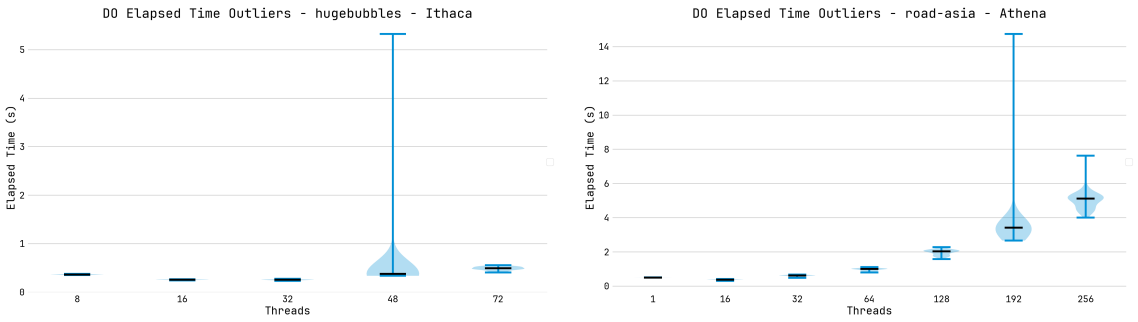
5

Discussion

This chapter discusses an oddity that occurs on rare occasions for the Direction-Optimizing algorithm, two attempted optimizations that were implemented but not used in the final algorithm, and finally future work that would be of interest to explore.

5.1 Direction-Optimizing Outliers

In rare cases, the Direction-Optimizing algorithm exhibited unexpected behavior throughout the experiments. Figure 5.1 shows the distribution of run-times for the DO algorithm on Ithaca and Athena, where two distinct outliers can be seen. As seen in Figure 5.1a there is a single run at 48 threads which was many times slower than the median. Similar behavior can be observed in Figure 5.1b at 192 threads.



(a) The DO algorithm has a significant outlier in the taken at the 48 thread count.

(b) The DO algorithm has a significant outlier at the 192 thread count.

Figure 5.1: Distribution of time taken per traversal for the DO algorithm on Ithaca and Athena on the hugebubbles and road-asia graphs respectively. 64 traversals were performed per thread count. The runs on Athena were not using SMT. Black lines show the median elapsed time.

After noticing this behavior, we investigated the issue using verbose logging to try to determine the cause. We noticed that all steps (i.e., a top-down step) the DO algorithm took periodically became very slow. Figure 5.2 visualizes the time taken for each top-down step for three separate traversals, a naturally occurring slow run, an artificially created slow run, and an unaffected run. Bottom-up steps are omitted since they very rarely occur and do not contribute to the slowdown. As seen in the

first half of the outlier run, the time taken for the steps seem to be grouped in three evenly spaced intervals. At around step 3000, the behavior of the DO goes back to normal. From this it is obvious that the size of the frontier, i.e., the amount of work to be done during a step, is not correlated with these slow step times.

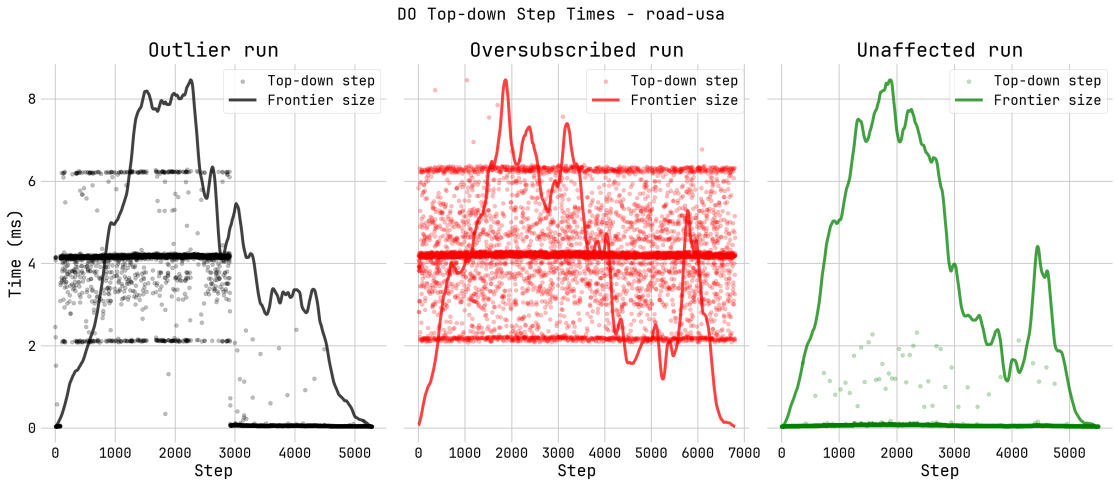


Figure 5.2: Time taken per top-down step (points) in the DO algorithm and the size of the frontier at each step (curves). The black color subplot shows an outlier run. The red subplot shows an artificially produced outlier run by oversubscribing cores. The green subplot shows a "normal" run, without any interference. The number of steps differ between runs due to the traversals beginning at different source nodes. All data was gathered from Ithaca using 72 threads.

The reason for this behavior is likely *interference*. As explained in Section 2.7, there is a synchronization barrier between each level (step). A single thread being delayed, e.g., due to the operating system unscheduling it, forces all other threads to wait. If this happens repeatedly for some reason, it greatly impacts the performance of the DO. To confirm this hypothesis we intentionally oversubscribed cores, the result of which can be seen in the *Oversubscribed run* in Figure 5.2. We did this by assigning OpenMP to use more threads than it was able to schedule at a single time. For example with the following environment variables:

`OMP_NUM_THREADS=72 GOMP_CPU_AFFINITY="0-64".`

Comparing the the outlier run and oversubscribed run, the characteristics of the slow steps are very similar, suggesting that interference is the issue. However, it is not obvious what causes the interference in the outlier run. Possible reasons are background processes or the benchmarking server being momentarily accessed.

5.2 Omitted Optimizations

This section is devoted to optimization techniques that were implemented, but then later discarded. There is also reasoning about the benefits and limitations of utilizing these optimizations.

5.2.1 Sequential Start

Sequential start is simply beginning the BFS traversal by exploring the first n nodes sequentially. This can significantly reduce the amount of extra work generated during a traversal, resulting in faster traversals. We suspect that this is due to extra work being generated immediately at the beginning of a traversal, which can then cascade during the run. Consider a source node n_{source} with two or more neighbors. The first explored neighbor n_{neigh} will then be enqueued and immediately dequeued by another thread. Thus, if n_{neigh} shares a neighbor n_{shared} with n_{source} and n_{shared} is explored from n_{neigh} before being explored from n_{source}, then extra work has been created. In addition, n_{shared} will be enqueued and continue to produce extra work until a correct path from n_{source} "catches up". By visiting the first n nodes sequentially, the amount of extra work is reduced since recently enqueued nodes will likely not be dequeued before its producer has been fully explored, due to intermediate nodes in the queue acting as a buffer.

The reason why this optimization is omitted from the final algorithm is its ineffectiveness when used with batching on sparse graphs, and batching is more important for the algorithm's performance. The reason why a sequential start does not improve the performance of the algorithm when used in conjunction with batching is likely that the same behavior is implicitly produced by batching, to some extent. With batching, the traversal will execute sequentially until exploration of a single dequeue-batch produces more than one enqueue-batch. Exactly how long the batching approach executes sequentially is determined by the graph's topology, where the duration decreases when the degree of the nodes increases.

It should be noted that the optimization can be effective on dense graphs where the frontier grows very quickly. Using a sequential start with $n = 1000$ we were able to achieve speedups of around 1.6× for some unordered algorithms compared to those presented in Figure 4.9. This is due to the extra work performed being significantly reduced. For some traversals, no extra work was performed at all. However, since the focus is on improving performance on sparse graphs, and this optimization was ineffective for those, we still choose to omit it.

5.2.2 Stickiness

Stickiness is an optimization technique that affects the random sampling of sub-queues in the d-CBO queue during enqueue and dequeue operations, discussed in Section 2.4. Instead of sampling new sub-queues for each operation, we "stick" to the same sample of sub-queues for a number of operations (stickiness period), before sampling new ones. The stickiness periods for enqueue and dequeue are independent, meaning that they can be set individually and do not affect one another. This approach has been used in previous literature [26], which showed the potential benefits of increasing cache locality and achieve higher throughput, with the drawback of greater rank errors.

Although the optimization showed potential, it was discarded because of the following reasons. Stickiness worked better on the BFS implementation with the d-CBO

queue without batching. By not using batching, each sub-queue in the d-CBO queue has a greater number of elements, making it possible for a thread to stick to a sub-queue for longer. The throughput was increased at the cost of rank errors, similar to the behavior described in [26]. However, the amount of work that had to be redone due to the increase in rank errors outweighed the gained throughput. When stickiness was combined with batching it had less of an effect, since batching reduces the total number of elements. When sticking to a sub-queue with few elements, we will quickly have to resample another sub-queue to keep providing the thread with elements, mitigating the purpose of stickiness. Stickiness was also hypothetically less performant due to the importance of providing threads with disjoint sub-queues, in order to efficiently work in parallel. It was found that using between 64-128 sub-queues provided the best performance for our other optimizations. This is too few sub-queues per thread for stickiness compared to what was used in [26], where they provide at least 3 sub-queues per thread.

5.3 Future Work

The results show the viability of sacrificing strictness in exchange for performance, both in terms of the level-synchronous constraint and the semantics of the FIFO queue. It seems likely that there are many algorithms which can be improved by relaxing, in particular graph algorithms. For BFS algorithms, we propose two areas which can be interesting to further investigate.

5.3.1 A Fast Bag

As mentioned in Section 4.6, the total size of the d-CBO queue can be very small on sparse graphs. As a result, the ordering semantics of the data structure is diminished. This presents an interesting question of whether a completely unordered data structure that was optimized for this use case could be more efficient. A data structure such as a *bag* is a potential candidate, if optimized for fast add and remove operations on a small collection, particularly for producer-consumer workloads. Since this is a very niche use case, we were unable to find any such algorithm in the literature that seemed to facilitate faster enqueue (add) and dequeue (remove) operations than the d-CBO queue.

5.3.2 Tuning Parameters

From the experiments of this thesis there was no clear configuration of algorithmic parameters that was generally optimal. These parameters include: number of sub-queues in the d-CBO queue, batch size, depth threshold, number of threads, and of course the graph itself. It could be interesting to more thoroughly investigate when and why these different parameters provide the best performance. With that analysis, it might be possible to create a preprocessing heuristic to select the best parameters for the traversal, or even decide to use the DO approach.

6

Conclusion

Our findings demonstrate that unordered BFS algorithms are able to outperform the Direction-Optimizing algorithm on sparse real-world graphs. Using a relaxed FIFO queue immediately improves performance compared to the same algorithm using a non-relaxed concurrent one. However, this improvement alone is insufficient to outperform level-synchronous methods. To further improve performance, three additional optimizations were introduced: batching, DAD, and BOD.

Batching is a common optimization technique to improve cache efficiency and reduce memory pressure on the underlying queue. By enqueueing and dequeueing batches of nodes rather than single nodes, the speedup is often doubled or more. DAD and BOD are two novel optimizations specific to unordered BFS. The DAD optimization improves work-speed significantly by guaranteeing that every enqueued batch is full and enabling threads to immediately work on specific batches without enqueueing them first. This comes at the cost of doing more extra work, but the increase in work-speed is typically greater, resulting in a performance increase. The BOD optimization balances the BFS traversal by delaying the exploration of batches with nodes that are deemed to have progressed too far. This reduces the amount of extra work on sparse graphs significantly but still allows for a high work-speed, resulting in further performance improvements.

With these optimizations, our unordered algorithm is able to outperform the state-of-the-art Direction-Optimizing algorithm on sparse real-world graphs. We achieve geometric mean speedups of 2.0× compared to the Direction-Optimizing algorithm and 6.9× compared to a sequential baseline. In conclusion, this demonstrates the viability of the unordered approach and provides a concrete example where relaxation is beneficial.

Bibliography

[1] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, IEEE, 2011, pp. 78–88.

[2] R. Zhou and E. A. Hansen, "Breadth-first heuristic search," *Artificial Intelligence*, vol. 170, no. 4-5, pp. 385–408, 2006.

[3] J. Barnat, L. Brim, and J. Chaloupka, "Parallel breadth-first search ltl model-checking," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, IEEE, 2003, pp. 106–115.

[4] S. Rawat and D. Patil, "Efficient focused crawling based on best first search," in *2013 3rd IEEE International Advance Computing Conference (IACC)*, IEEE, 2013, pp. 908–911.

[5] J. Silvela and J. Portillo, "Breadth-first search and its application to image processing problems," *IEEE Transactions on Image Processing*, vol. 10, no. 8, pp. 1194–1199, 2001.

[6] M. Kurant, A. Markopoulou, and P. Thiran, "On the bias of bfs (breadth first search)," in *2010 22nd International Teletraffic Congress (LTC 22)*, IEEE, 2010, pp. 1–8.

[7] S. Beamer, K. Asanovi, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.

[8] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms," *Acm Sigplan Notices*, vol. 46, no. 8, pp. 3–12, 2011.

[9] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova, "Quantitative relaxation of concurrent data structures," in *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2013, pp. 317–328.

[10] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996, pp. 267–275.

[11] P. Ramalhete, *FAAArrayQueue - MPMC lock-free queue*, Nov. 2016. [Online]. Available: `https://concurrencyfreaks.blogspot.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html`.

[12] K. von Geijer, P. Tsigas, E. Johansson, and S. Hermansson, "Balanced allocations over efficient queues: A fast relaxed fifo queue," in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2025, pp. 382–395.

[13] A. Haas, M. Lippautz, T. A. Henzinger, *et al.*, "Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2013, pp. 1–9.

[14] H. Sutter *et al.*, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobbs journal*, vol. 30, no. 3, pp. 202–210, 2005.

[15] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[16] P. E. McKenney, "Selecting locking primitives for parallel programming," *Communications of the ACM*, vol. 39, no. 10, pp. 75–82, 1996.

[17] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, IEEE, 2003, pp. 522–529.

[18] J. D. Valois, *Lock-free data structures*. Rensselaer Polytechnic Institute, 1995.

[19] N. Hunt, P. S. Sandhu, and L. Ceze, "Characterizing the performance and energy efficiency of lock-free data structures," in *2011 15th Workshop on Interaction between Compilers and Computer Architectures*, IEEE, 2011, pp. 63–70.

[20] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 103–112.

[21] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, IEEE, 2015, pp. 445–456.

[22] P. Fatourou, N. Giachoudis, and G. Mallis, "Highly-efficient persistent fifo queues," in *International Colloquium on Structural Information and Communication Complexity*, Springer, 2024, pp. 238–261.

[23] R. Romanov and N. Koval, "The state-of-the-art lcrq concurrent queue algorithm does not require cas2," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, pp. 14–26.

[24] F. Ellen, D. Hendler, and N. Shavit, "On the inherent sequentiality of concurrent objects," *SIAM Journal on Computing*, vol. 41, no. 3, pp. 519–536, 2012.

[25] N. Shavit, "Data structures in the multicore age," *Communications of the ACM*, vol. 54, no. 3, pp. 76–84, 2011.

[26] M. Williams, P. Sanders, and R. Dementiev, "Engineering multiqueues: Fast relaxed concurrent priority queues," *arXiv preprint arXiv:2107.01350*, 2021.

[27] S. Beamer, A. Buluc, K. Asanovic, and D. Patterson, "Distributed memory breadth-first search revisited: Enabling bottom-up search," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, IEEE, 2013, pp. 1618–1627.

[28] S. Beamer, K. Asanovi, and D. Patterson, "The GAP benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[29] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on bluegene/l," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005, pp. 25–25. DOI: 10.1109/SC.2005.4.

[30] R. Berrendorf and M. Makulla, "Level-synchronous parallel breadth-first search algorithms for multicore and multiprocessor systems," in *Proc. Sixth Intl. Conference on Future Computational Technologies and Applications (FUTURE COMPUTING 2014)*, 2014, pp. 26–31.

[31] M. Ryczkowska, M. Nowicki, and P. Baa, "Level-synchronous bfs algorithm implemented in java using pcj library," in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2016, pp. 596–601. DOI: 10.1109/CSCI.2016.0118.

[32] A. Rukundo, A. Atalar, and P. Tsigas, "Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework," in *33rd International Symposium on Distributed Computing (DISC 2019)*, J. Suomela, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 146, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 31:1–31:15, ISBN: 978-3-95977-126-9. DOI: 10.4230/LIPIcs.DISC.2019.31. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.DISC.2019.31.

[33] A. Postnikova, N. Koval, G. Nadiradze, and D. Alistarh, "Multi-queues can be state-of-the-art priority schedulers," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 353–367.

[34] G. Milman, A. Kogan, Y. Lev, V. Luchangco, and E. Petrank, "Bq: A lock-free queue with batching," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '18, Vienna, Austria: Association for Computing Machinery, 2018, pp. 99–109, ISBN: 9781450357999. DOI: 10.1145/3210377.3210388. [Online]. Available: https://doi.org/10.1145/3210377.3210388.

[35] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, Seattle, Washington: Association for Computing Machinery, 2011, ISBN: 9781450307710. DOI: 10.1145/2063384.2063471. [Online]. Available: https://doi.org/10.1145/2063384.2063471.

[36] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication," in *European conference on principles of data mining and knowledge discovery*, Springer, 2005, pp. 133–145.

[37] K. Z. Ibrahim, S. Hofmeyr, and C. Iancu, "Characterizing the performance of parallel applications on multi-socket virtual machines," in *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, IEEE, 2011, pp. 1–12.

[38] T. Li, Y. Ren, D. Yu, and S. Jin, "Analysis of numa effects in modern multicore systems for the design of high-performance data transfer applications," *Future Generation Computer Systems*, vol. 74, pp. 41–50, 2017.