PARKore: Parallel, Asynchronous, Relaxed K-Core Decomposition

by

Stephen Yang

A thesis submitted in conformity with the requirements
for the degree of Bachelor of Applied Science

Supervisor: Mark Christopher Jeffrey
University of Toronto

PARKore: Parallel, Asynchronous, Relaxed K-Core Decomposition

Stephen Yang
Bachelor of Applied Science

Supervisor: Mark Christopher Jeffrey
University of Toronto
2024

Abstract

k-core is one of many metric used in graph analysis. Across various scientific research, k-core is applied as a descriptor of the importance of certain nodes in a network. Fast algorithms to generate a k-core decomposition of a graph exist, but are difficult to parallelize. Historically, k-core is thought to require a strict ordering of tasks for correctness. As such, methods that rearrange task orderings are considered but not performant in prior work. Most parallel k-core algorithms simply rely on bulk synchronous parallelization, where work is divided between processors and synchronized at common barriers. However, these types of algorithms can suffer penalties if work is not divided evenly, or if the total amount of work is small.

We present PARKore, the first asynchronous, relaxed k-core decomposition. PARKore enables relaxed scheduling of k-core tasks by building on top of the optimal peeling algorithm, adding new state variables in order to track dependencies. We show that PARKore handles priority inversions and order relaxation without issue. As a result, PARKore allows for high work-efficiency when parallelized, as all threads working on the algorithm can operate asynchronously from each other. We find that across six benchmarks, PARKore has near state of the art performance with some improvements for certain graphs.

Acknowledgements

I would first like to thank my supervisor, Mark Jeffrey, for his advice and guidance. Prior to this thesis, I was predominantly interested in computer hardware and RTL design. Mark introduced me to the field of parallel computing, where the interplay of hardware and software is critical for optimizing performance. His endless patience and productivity tips were integral to the completion of this thesis. I would also like to thank Gil, who continues to astonish me with his ideas and advice. Gil introduced me to k-core decomposition, provided the initial groundwork for this thesis, and was always instrumental in guiding me back in the right direction.

Additionally, thank you to my girlfriend Jessica, who persisted with me through all the late nights and work sessions. Finally, a big thank you to my mom for always being there when I needed it.

Contents

1 Introduction **1**

1.1 k-core Motivation . . . 2

1.2 Performance . . . 3

1.3 Contributions . . . 3

1.4 Thesis Organization . . . 4

2 Background **5**

2.1 k-core . . . 5

2.1.1 Peeling Algorithm . . . 6

2.2 Parallel k-core Algorithms . . . 9

2.3 Parallelization Approaches and Relaxation . . . 12

2.3.1 Speculation . . . 12

2.3.2 Bulk Synchrony . . . 13

2.3.3 Relaxation . . . 13

2.4 Concurrent Priority Schedulers . . . 14

3 PARKore **15**

3.1 The PARKore Algorithm . . . 15

3.2 Optimizations and Multithreading . . . 20

3.3 MultiQueues as Priority Schedulers . . . 21

4 Evaluation and Results **23**

4.1 Methodology . . . 23

4.2 Performance . . . 24

4.3 Cache Performance . . . 27

4.4 Priority Scheduler Overheads . . . 30

4.5 Sensitivity to MultiQueue Parameters . . . 31

5 Conclusion and Future Work **36**

5.1 Future Work 36

A Appendix **38**

A.1 PARKore Code 39

A.2 Sequential BZ Implementation 53

A.3 MPKI for LLC Load and Stores 58

List of Figures

2.1 The k-core of an example graph. 6
2.2 An example of k-core decomposition. 8
2.3 A correct k-core decomposition solution. 8
2.4 An incorrectly solved k-core decomposition due to order relaxation. . 9

3.1 A snapshot of a k-core decomposition example after a yellow vertex updates the central vertex at T_0. 18

4.1 PARKore and Julienne **speedup** against work-efficient sequential implementation. 25
4.2 Running time of PARKore and Julienne. 48h refers to 48 hyper-threads 26
4.3 Instruction count of all applications. 27
4.4 **MPKI**: LLC Load and Store misses across all graphs and applications. 28
4.5 Cache accesses and misses across all graphs and applications, normalized to graph size n. 29
4.6 Time breakdown for PARKore using BMQ and MQIO. 31
4.7 **Speedup**: MQIO relative to best sequential. 33
4.8 **Speedup**: Bucket MQ (64 buckets) relative to best sequential. 34
4.9 **Speedup**: Bucket MQ (256 buckets) relative to best sequential. . . . 35

A.1 **MPKI**: LLC Load Misses. 58
A.2 **MPKI**: LLC Store Misses. 59

List of Tables

2.1 Work-inefficient parallel k-core algorithm runtime (s) results [17] . . . 11
2.2 Work-efficient parallel k-core algorithm results [39] 12
2.3 Published parallelization approaches for k-core 12

3.1 Data structures in PARKore 16
3.2 State variables for vertex in example. Left: `update_hist`. Right: Other. 19
3.3 Default MQ Parameters. 22

4.1 Overview of graph datasets 24

A.1 State variable mappings 39

1. Introduction

Network structures pervade all aspects of society, from the structure of energy grids, to cell organization in organisms, to social media networks. Insights into the characteristics of these networks have enamored researchers for decades. For example, graph analysis of social networks could inform how information (or disease) travels. To study these networks, many metrics have been proposed, including graph density, shortest paths, graph centrality [12], community detection [11], and more. One such method is the k-core decomposition of a graph. The k-core of a graph is often employed as a proxy for the most important set of nodes in the graph. Analysis using the k-core of networks has various applications, such as explaining sudden state transitions in statistical mechanics [25], extracting graph characteristics using graph mining [34], modelling of disease transmission in societies [18], and revealing organization in the brain [20].

However, as networks of interest grow, subsequent analysis calls for hugely increased computational demand. On modern computers, high performance of programs is achieved by exploiting parallelism across various levels. Processors extract Instruction-Level Parallelism by executing independent instructions concurrently, while programs must be written by programmers to execute well on these highly multithreaded processors. For k-core decomposition, fast algorithms do exist; however, they do not parallelize well. Programs that parallelize well characteristically contain lots of separable data and independent instructions, which can be easily divided into smaller tasks. Unfortunately for k-core decomposition, this is not the case as the best sequential algorithms contain significant dependencies within the algorithm itself. These dependencies also present themselves during runtime, varying greatly with the input graph. Consequently, predictions to the dynamic schedule are futile. In fact, k-core decomposition is currently known to be *strictly scheduled* [39, 28], meaning a global scheduling order is required for the algorithm to complete correctly.

1.1 k-core Motivation

Various research fields have applied k-core as a metric of node importance or interconnectedness. k-core as a graph metric appears in seemingly unrelated fields, where graph nodes range from physical atoms [25, 6], to human cells [7, 13], or individual people [37, 19]. Historically, k-core materializes in surprising ways across these applications: For example, in statistical mechanics, the sudden emergence (percolation) of k-cores in a network of atoms can indicate sudden state transitions occurring only for certain materials at certain densities [25].

Recently, breakthroughs in neuroscience have enabled modelling of the human cerebral cortex as a graph with unprecedented fidelity. In one case, a selection of 66 designated anatomical subregions were taken, resulting in a graph of the human brain with 998 regions of interest (ROI's) [13]. Graph edges were experimentally discovered using magnetic resonance imaging (MRI) and computer-based diffusion MRI of a human brain. Using k-core on this neuronal graph, researchers found that structural connections and functional interactions between cortex regions were significantly correlated. In addition to mapping the cortex, connectivity mapping using k-core has also been applied to Alzheimer's disease research [7]. In patients with Alzheimer's, the k-core model of the brain loses nodes rapidly as the disease progresses, signifying a loss of neural interconnections.

Social graphs are another domain where k-core appears. Modern media networks, such as the Facebook social graph [37], Twitter (now X) [19] and Orkut, are frequently modelled as graphs where edges correspond to relationships between between people (nodes). In these social graphs, a node in a k-core suggests a high degree of importance. Conventional wisdom suggests that the most efficient spreaders of disease or information in a society should be those who are most well connected (highest degree in a graph) or most central; however, empirical evidence suggests that these optimal spreaders are those located within a k-core [18].

As such, performance improvements for k-core decomposition is of interest not only to parallel programming research, but to scientists across various fields. However, due to the nature of k-core decomposition, current software and hardware struggles to effectively parallelize this algorithm. We will discuss the reasons for this in the following sections.

1.2 Performance

The peeling algorithm for k-core decomposition involves peeling away lower degree vertices until only the kth core remains [4]. As a result, a strict ordering is required for correctness in serial execution of k-core decomposition. This requirement extends to parallel k-core implementations, and constitutes a large challenge in parallelizing k-core [28]. Existing work towards this end groups into three main methodologies: bulk-synchronous execution, speculative execution, and order relaxation [28]. Most k-core literature focus on bulk-synchronous approaches, where work is partitioned into private queues, and separate compute threads must synchronize with a barrier to make global updates [24, 8, 9, 39]. However, the use of barriers can drastically decrease performance: in some algorithms, as much as 90% of program run time is spent on synchronization related overhead [30]. Additionally, barriers can induce highly uneven work distribution. Even for state of the art k-core decomposition algorithms, a significant portion of threads process as few as one vertex between barriers [28]. Second, a major challenge for parallelizing k-core decomposition has been increasing the ratio of useful work done across all threads relative to the total work done by a sequential algorithm. Termed as *work efficiency* [5], this ratio is a crucial performance metric that has only recently been optimized for k-core [9]. Traditional implementations of parallel k-core require roughly $O(m + k_{max}n)$ work, while sequential implementations take $O(m + n)$ work [9].

In recent years, dependency analysis research has yielded algorithms with relaxed schedulers which are now being applied to problems with strict ordering requirements [3]. These algorithms exploit dependencies between tasks to enable out-of-order execution. However, k-core is thought to require explicit global synchronization [39] alongside strict ordering, thus there are no current implementations using relaxed scheduling.

1.3 Contributions

We think that through scheduler dependency analysis [28] of the k-core algorithm, a novel asynchronous parallel k-core algorithm (which forgoes barriers) can be implemented in software. This thesis introduces PARKore: Parallel, Asynchronous, Relaxed k-core decomposition. PARKore is a highly parallelizable, work-efficient, k-core decomposition algorithm which challenges traditional notions on the ordering requirements for k-core. The main contributions of this work include the formulation of PARKore and evaluation of PARKore's software implementation.

1.4 Thesis Organization

This thesis is organized as follows: Chapter 2 provides background on k-core, the k-core decomposition problem, and motivation for k-core. Additionally, a review of prior work in extracting parallelism from k-core is presented. A section on prior work on concurrent priority schedulers is included. In Chapter 3 we present PARKore, the first relaxed-scheduling, work efficient implementation of k-core. In Chapter 4, we describe evaluation methods, and benchmark PARKore's performance across 6 different graph datasets. Finally, in Chapter 5, we conclude and propose future work.

2. Background

To begin, we introduce the notion of a k-core and its related definitions, such as the *coreness* (the maximum core) of a vertex in section 2.1. Define a graph as $G = (V, E)$, where V refers to the set of it's vertices, and E the set of edges connecting vertices. For a graph G, the challenge of finding the coreness numbers of all vertices is called the k-core decomposition of a graph. The Peeling algorithm is an optimal work-efficient algorithm for k-core, and is described in 2.1.1. Yet, this algorithm is challenging to parallelize, with previous attempts described in section 2.2. Lastly, a short background on concurrent priority schedulers is presented in 2.4.

2.1 k-core

The term k-core was initially conceptualized in 1983 by two independent authors. The first reference to k-cores was from Seidman as a metric to categorize the "knittedness" or cohesion of a social network [33]. Simultaneously, Matula and Beck independently introduced the notion of k-linkages in a paper about smallest last ordering, and described an $O(V + E)$ algorithm to find k-linkages [22]. Although Matula and Beck's algorithm was for smallest last vertex ordering, their algorithm computationally yields the k-core upon completion. Formally, we define the k-core of a graph $G = (V, E)$ to be:

Definition 1 (k-Core) *For a graph $G = (V, E)$, let H be a subgraph of G (a portion of G). Let $\delta(v)$ be the degree of node v in H. Then, the k-core of $H \subseteq G$ is the maximal subgraph such that $\forall v \in H, \delta(v) \geq k$.*

Alternatively, picture a k-core as nested subgraphs H_k of G, where all vertices in H_k have at least k neighbors. When $k = 1$, the 1-core is trivially every connected set of vertices, or equivalently the exclusion of all isolated vertices from G. Note that if G is connected, then $H_1 \equiv G$. Additionally, from the definition of k-core, any vertex u in a k-core has at least k neighbors. As such, we can show that k-cores are nested: vertex $u \in H_k$ has $\delta(u) = k > k - 1$, thus $u \in H_{k-1}$. A example of k-core on a graph with $k = 1, 2, 3$ is shown in Figure 2.1.

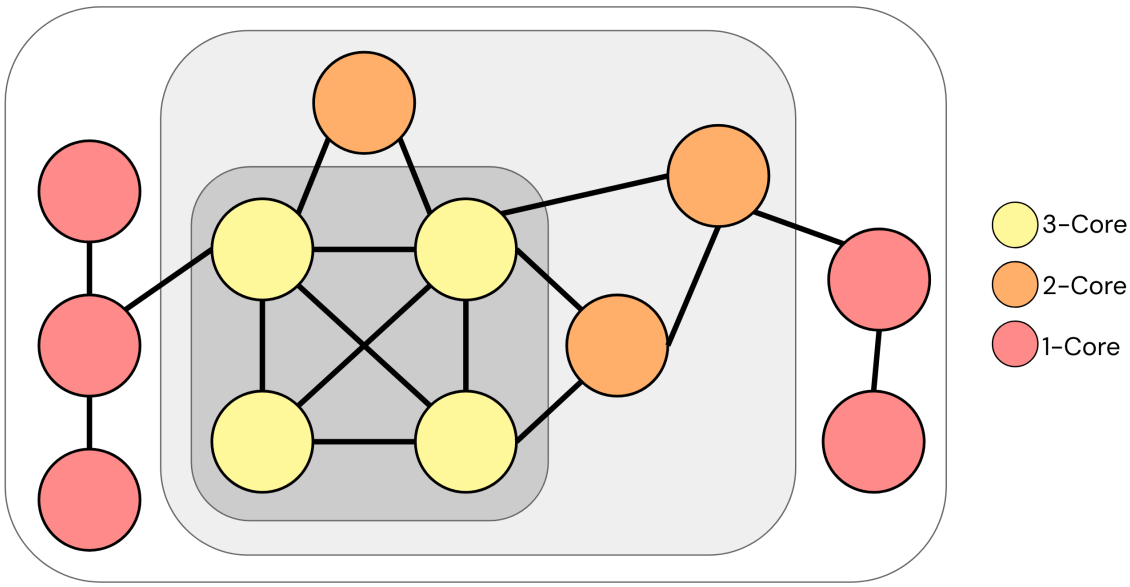

Figure 2.1: The k-core of an example graph.

Lets begin by defining the *coreness* of a vertex:

Definition 2 *The coreness of vertex $v \in V$ is the largest k for which $v \in H_k$.*

Returning to figure 2.1, note the four yellow vertices in the middle. Despite the top left yellow vertex having 5 total neighbors, it only has a coreness of 3, as two of its neighbors are not part of a 3-core. In the left most column of red vertices, the middle vertex has degree 3, but only has coreness 1 since not all of its neighbors have 3 cores. Because of the nesting property, it is sufficient for k-core decomposition algorithms to compute the coreness numbers of each vertex $v \in V$. Any k-core of graph $G = (V, E)$ can be simply reproduced from the coreness numbers by passing over each vertex in G and checking if all v with coreness $> k$ belong in H_k. We investigate k-core decomposition algorithms in the following sections.

2.1.1 Peeling Algorithm

Building on the work of Matula-Beck, Batagelj and Zaversnik implement their k-core decomposition algorithm, using priority queues, which runs in the same time bounds $O(V+E)$ [4]. The BZ algorithm, shown in Algorithm 1, repeatedly "peels" the lowest degree vertices, until no vertices remain.

Algorithm 1 BZ k-core Decomposition

1: **function** BZQ($cores$)
2: Compute degrees of vertices
3: Order vertices V in increasing order of degree
4: **for** $v \in V$ in order **do**
5: core[v] := degree[v]
6: **for** $u \in v.neighbors$ **do**
7: **if** degree[u] $>$ degree[v] **then**
8: degree[u]$-=1$
9: Reduce priority of u by 1
10: **end if**
11: **end for**
12: **end for**
13: **end function**

In the BZ algorithm, an initialization is performed where vertices are ordered in ascending order by degree (lines 2-3) into a set V. Then, each iteration of the loop (line 4), the lowest degree vertex currently in V is popped (line 5). Then, a peeling is performed for all of v's neighbors (line 6). If any neighbor u of v's has a higher degree, u's degree is decremented and it's priority reduced by 1 (lines 7-9). Examining the BZ algorithm, we note that the algorithm maintains two invariants:

1. Whenever an vertex v is popped (line 5 in Algorithm 1), v necessarily has the lowest degree in the set V.

2. At any iteration, the degree of an *unpopped* node in the graph is an upper bound on it's coreness. Accordingly, when a vertex u is removed (i.e. it's degree drops below the current lowest degree vertex v), the degree of u at removal time is u's coreness.

Invariant 1 is held when V is a **priority queue**, where a `dequeueMin()` function would be used to pop a lowest degree vertex. Invariant 2 can be shown given a priority queue is used: If a vertex u has a lower priority than the lowest vertex $v \in V$, then no vertex $w \in V$ can cause a coreness update, as all coreness updates are induced by lower degree vertices. As such, k-core decomposition requires a strict ordering where the scheduling of vertices in the algorithm is fixed for correctness. As an example, consider the graph in figure 2.2. The correct ordering is to pop red vertices first, which update the center blue vertex and top right orange vertex. Then, the top right vertex has a new degree of 1, so it is popped and consequently updates the center orange vertex.

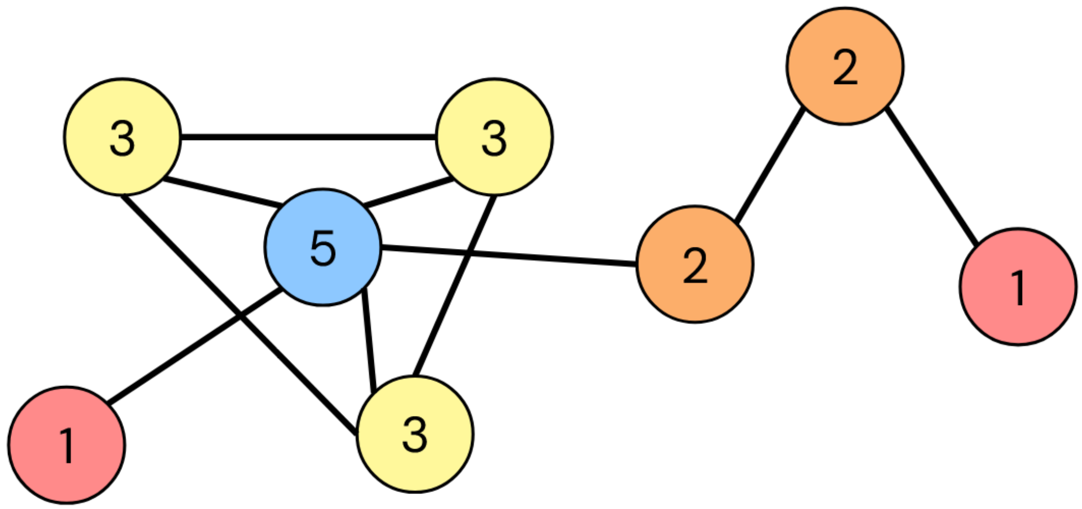

Figure 2.2: An example of k-core decomposition.

Ultimately, the graph example in figure 2.2 produces the result shown in figure 2.3. The series of updates resulting in this solution are depicted using arrows. However, what happens if instead of an algorithm accidentally started by popping not either of the red vertices, but one of the yellow (degree 3) vertices?

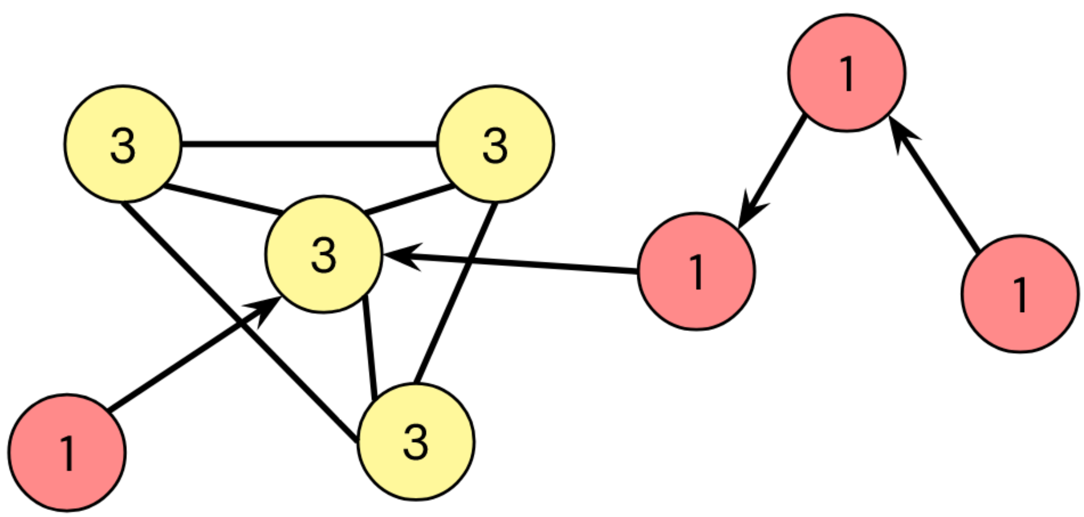

Figure 2.3: A correct k-core decomposition solution.

Take that the top right yellow vertex is decremented at the beginning of the algorithm. In this case, the yellow vertex (degree 3) updates the original blue vertex (degree 5), as indicated by the red arrow in figure 2.4. Next, the algorithm behaves as before, yielding two more decrements to the previously blue vertex for a total of 3 updates as indicated by black arrows. In this scenario, the originally blue vertex's degree has dropped so significantly that it now updates its yellow neighbors (orange arrows), causing the coreness numbers for every vertex in all collapse to 1. A representation of this incorrect result is shown in figure 2.4, where every vertex now has coreness 1 due to execution with an improper priority ordering.

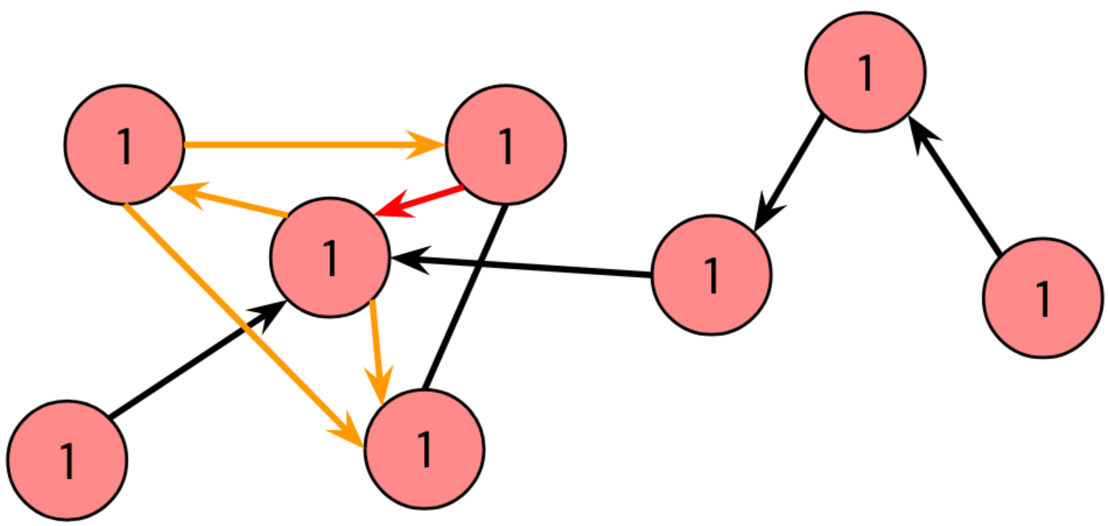

Figure 2.4: An incorrectly solved k-core decomposition due to order relaxation.

As evident from this example, the scheduling order is crucial to ensure the correctness of the decomposition algorithm. Current software and hardware struggle to parallelize for this reason, as the scheduling order dictates a global structure between threads that would prefer to work alone. Specific challenges in parallelizing k-core manifest in existing work, which we will explore in the next section.

2.2 Parallel k-core Algorithms

Returning briefly to the priority queue based BZ algorithm, observe that every vertex is popped once and only once: each edge $e \in E$ from u to v is traversed at most twice (once if u is popped, once if v updates u). This is a consequence of using priority updates (`dequeueMin`) as there is no `push` to add elements to the priority queue. In the BZ algorithm, $|V|$ passes are required, with up to $O(E)$ edge processed in total.

As such, the BZ algorithm requires $O(m + n)$ work, where $m = |V|$ and $n = |E|$. This result forms a baseline which all parallel implementations are measured from.

Montresor et al. implemented a parallel k-core algorithm by partitioning a graph into memory onto hosts in a distributed compute systems [24]. Each processor operates on its local subset of G. During an iteration of the distributed algorithm, each node produces an estimate of its coreness, which is message-passed to relevant neighbor processors. The neighbor processors utilize updated coreness estimates, repeating until convergence across all systems. In theory, this algorithm requires $O(k_{max} \cdot n)$ work [24]. However, for certain graphs such as one with a long "chain" of equal degree nodes, the practical worst case work could be as high as $O(m \times n)$, since n iterations are needed, and each iteration could take $O(m)$ time [17].

ParK, implemented by Dasari et al., is the first parallel algorithm to run in $O(k_{max} \cdot n + m)$ work [8]. In ParK, the k-core update is split into two parts: a scan phase, followed by `processSubLevel()`. In the scan phase, the graph G is traversed, and all nodes with degree equal to k are added to the current level. In `processSubLevel()`, every vertex in the current level k is peeled, with an added optimization to add $k + 1$ degree nodes to the next level. The work for scan phase is $O(k_{max} \cdot n)$, since there are k_{max} levels with n checks per level. `processSubLevel()` additionally takes $O(m)$ work, since each vertex is processed exactly once (when its degree is equal to the current level k). Combined, the work of the ParK algorithm is $O(k_{max} \cdot n + m)$. When parallelizing, the scan phase is trivially parallelized: n nodes are simply distributed amongst t threads. Similarly, parallelizing `processSubLevel()` involves distributing the work equally among the t threads. Critically, between the scan and process phases, barriers are constructed which all threads must synchronize to in order to progress. ParK, therefore, represents an example of bulk-synchronous parallelization.

Building off of ParK, Kabir and Madduri attempted to reduce overheads stemming from barrier and synchronization overhead in their PKC algorithm [17]. In essence, PKC removes the barrier between scan and process phases by instead instantiating thread-level buffers, which each hold n/t nodes of G. Since coreness updates are atomic decrements, it is sufficient for each thread in PKC to only scan cores which are within its own thread-level buffer. The authors note that the sequential work for PKC is the same as ParK ($O(k_{max} \cdot n + m)$). Additionally, a synchronization barrier between process and scan (at the end of each iteration) is still required. A comparison of the following work-inefficient algorithm runtimes for an selection of graphs is shown in Table 2.1. These results are reported by [17] as runtime in seconds for the three implementations, on a quad-socket 32 thread Intel Xeon system with

512GB of memory.

	com-orkut	soc-LiveJournal1	soc-friendster	indochina-2004	webbase-2001
PKC	2.38	0.87	31.32	1.77	5.47
ParK	3.49	1.44	35.51	10.45	52.70
MPM	9.22	4.20	386.74	3.47	46.95

Table 2.1: Work-inefficient parallel k-core algorithm runtime (s) results [17]

On average, PKC outperforms both ParK and MPM on most graphs. Additionally, the authors find that BZ outperforms both ParK and MPM when run serially, but PKC has better single threaded performance than BZ [17].

In 2017, a major shift occured in the landsdcape of parallel algorithms research. Julienne was able to achieve a work-efficient, parallelized implementation of k-core [9]. The Julienne implementation of k-core utilizes buckets for vertex degrees, removing the need to scan G or create a thread-local portion of G. Consequently, there is also no need to synchronize all threads at the end of each iteration, like in PKC, since elements can be independently (atomically) inserted or removed from buckets. The Julienne k-core requires $O(m+n)$ expected work with $O(\rho \log n)$ depth, where ρ is the peeling-complexity, or number of steps required to peel the graph completely [9].

Following Julienne, Ordered GraphIt bolstered performance further by replacing the priority update function with a histogram (vector) based update [39]. In Julienne, after each update to a specific vertex v's bucket, any consecutive accesses to v requires a function call and further computations, which may also suffer from contention with other threads if v has a high degree. Ordered GraphIt subverts these overheads using lazy bucketing with a fixed priority decrement, suitable for k-core as updates can only apply constant unity decrements to coreness. Consequently, contention is also avoided on vertices that have high degree [39].

A comparison between GraphIt, Julienne, and prior work Ligra, is shown in Table 2.2. Note that while Ligra does not constitute a work-efficient implementation, the GraphIt and Julienne authors also benchmark their code against Ligra as a baseline, with significant speedups. Runtimes are computed on a dual-socket, 24 core (48 hyper-thread) Intel Xeon system with 127GB of DDR3-1600 memory [39].

As shown in Table 2.2, Ordered GraphIt outperformed Julienne in all graphs for k-core, although for many graphs the speedup was marginal. Both Ordered GraphIt and Julienne outperformed the work-inefficient Ligra by a significant margin. Additionally, when compared to the results in Table 2.1, the work-efficient implementations outperform prior methods for parallel k-core algorithms.

	com-orkut	LiveJournal1	Friendster	Twitter	RoadUSA
Ordered GraphIt	1.634	0.745	14.423	10.294	0.305
Julienne	1.62	0.752	14.6	10.5	0.327
Ligra	8.09	5.99	324	225.102	1.76

Table 2.2: Work-efficient parallel k-core algorithm results [39]

Despite their performance, Ordered GraphIt and Julienne still remain rooted in the class of bulk-synchronous algorithms: Bucketing implementations still require threads to arrive at barriers between executing work at different priority levels. Is it then possible to extract more parallelism in k-core by doing work *across* priority levels?

2.3 Parallelization Approaches and Relaxation

Even with the advent of work-efficient, parallelizable k-core decomposition algorithms, we believe there is still significant room for unlocking parallelism in k-core. Currently, there are three main approaches for parallel execution (scheduling) of sequential tasks while maintaining priority ordering: *speculation*, *bulk-synchrony*, and *relaxation*. A grid of existing ideologies for applying parallelism to priority scheduled algorithms is shown in table 2.3.

	Speculative	Bulk-Synchronous	Asynchronous
Software	Infeasible	[8, 17, 24, 35, 9, 39]	**Unexplored**
Hardware	[15]	[2, 23]	[28]

Table 2.3: Published parallelization approaches for k-core

2.3.1 Speculation

Speculative execution can enable execution of ordered algorithms with strict priority [14, 26], by speculating when a task can be executed out of order, and rolling back on incorrect speculations. In practice, software speculation incurs significant overhead penalties which greatly outweight the potential benefits [39]. For speculation to work, proper rollback mechanisms supporting precise state recovery need to be implemented, which are simply not performant in software. Hardware solutions have been proposed to unlock parallelism by creating the aforementioned rollback mechanisms and precise state structures, using custom hardware [15, 28]. However, due to costs associated with hardware manufacturing, these solutions have yet to be built.

2.3.2 Bulk Synchrony

Bulk-synchronous execution involves bucketing equal priority tasks, and executing them in parallel. Since all items share the same priority, they can be executed in any order. Additionally, these tasks are required to synchronize before any items with different priority can be processed.

As discussed in section 2.2, software bulk-synchronous approaches are common [8, 17, 24, 35], but suffer from significant drawbacks. Namely, the use of barriers can exacerbate workload discrepancies between threads, resulting in highly uneven work distribution. For example, as much as one third of all barriers process a single vertex in certain k-core applications [28]. Further, in shared memory systems, synchronization overheads emerge when multiple threads write to the same memory address. Even when atomics are used (or alternative lockless methods), this can result in undesirable overhead [17]. Hardware bulk-synchrony is also possible, such as the distributed computing approach taken by [24]. Recently, the use of GPU's in vectorizing k-core has enabled performant hardware bulk-synchronous implementations to expose SIMD (Single Instruction, Multiple Data) parallelism [23, 2]. However, these suffer from the same restrictions as software bulk-synchronous algorithms.

2.3.3 Relaxation

Relaxed approaches allow schedulers to relax the priority order, distributing tasks with varying priorities across processors. However, relaxation forfeits guarantees that task ordering will match work-efficient task orderings. As such, relaxed programs represent a compromise between parallelism and work-efficiency. Recently, Alistarh et al. applied relaxed scheduling to iterative algorithms such as maximal independent set (MIS) [3]. Using a relaxed scheduler, Alistarh et al. was able to achieve $O(n + \text{poly}(k))$ work when compared to exact schedulers, where k is a relaxation factor. Somewhat unintuitively, the extra $\text{poly}(k)$ work is not proportional to size of the graph, suggesting a possibility for amortization of the scheduler cost with sufficient graph size or parallelization. Experimental results validated this, with better performance due to the scalability of the relaxed scheduler despite $\text{poly}(k)$ additional work.

We believe that through scheduler dependence analysis, similar relaxation techniques can be brought into a k-core algorithm, while maintaining work efficiency. This thesis explores a software asynchronous (relaxed) approach, appearing as the previously untouched territory in table 2.3.

2.4 Concurrent Priority Schedulers

Priority scheduling and associated data structures (e.g. a priority queue) are indispensable for algorithms across various domains. Beyond k-core, other algorithms such as Set Cover [16], Greedy Maximal Independend Set (MIS) [3], and Residual Belief Propagation [10] also benefit greatly from priority scheduling. However, priority queues are less efficient at higher core counts due to contention between threads [21]. Prior work in Concurrent Priority Schedulers (CPS) have produced the MultiQueue [31, 29, 38], which is an array of sequential priority queues. Each MultiQueue (MQ) contains $c \cdot p$ priority queues, where c is a tunable parameter, and p is the number of threads. Each of the $c \cdot p$ queues is protected by lock access, meaning at most p queues can be locked at once. Given $c > 1$, finding an unlocked queue is guaranteed.

The key insight behind MQ's is that compared to a single PQ, the MQ enables $O(\log n)$ insertion with minimal overheads for queue selection (if $c > 1$) and locking in a multithreaded use case. As such, insertions are incredibly efficient as threads do not need to contend with each other. On the other hand, pops from the MQ are not guaranteed to return the global highest priority, but instead approximates on average a high priority element by selecting a maximal priority element from a small number of queues. In section 3.3, we apply a variant of the MultiQueue CPS to the PARKore algorithm.

3. PARKore

First, we introduce PARKore in section 3.1, motivated by the example given in section 2.1. Moreover, we provide insight into the key invariant of the PARKore algorithm and its implication on relaxation. Then, optimizations and multithreading techniques used in the software implementation of PARKore are provided in section 3.2. Lastly, in section 3.3 we describe the CPS used in PARKore, which is a variant of the MultiQueue which builds on top of the Bucket Queue structure.

3.1 The PARKore Algorithm

To motivate the PARKore algorithm, we first define key structures that allow PARKore to operate with any relaxation in the schedule. These key structures act as state variables for every vertex in the input graph G. Table 3.1 shows the state variables for a vertex v. The size column indicates the size overhead of the state variable relative to the underlying node representation. For example, if graph nodes are represented using 32b of data, then `estimated_core` would also require 32b of data. Note that $v.e$ refers to the set of neighbors of v.

Name	Size	Description
`current_core`	1	Current coreness of vertex v
`update_hist`	$\lvert v.e \rvert$	`update_hist` is initialized as an array of size $v.\text{deg}+1$, and represents a histogram of incoming edge `visible_core` from v's neighbors. At termination, the coreness of v is equal to the H-index of `update_hist`.
`visible_core`	1	The coreness state of v visible to v's neighbors. Alternatively, `visible_core` can be interpreted as the current lowest estimated coreness of v. Initialized to a large number (i.e. n).
`ecp`	1	The number of *equal core predecessors*. An equal core predecessor is an update from neighbor u to v, occuring when u had v's `current_core`, which speculatively causes v's coreness to be reduced. `ecp`> 0 suggests a priority inversion occured, and a subsequent decrement to v's coreness is ignored until `ecp`$= 0$.

Table 3.1: Data structures in PARKore

Pseudocode for PARKore is provided in algorithm 2. Initially, `current_core`, `ecp`, `update_hist` and `visible_core` are initialized according to their descriptions in Table 3.1 on a per-vertex basis, as seen in lines 2-6. Initially, all elements in G are inserted into the PQ on line 7. Then, the algorithm loops indefinitely (lines 41-43), exiting when the PQ is empty (lines 12-14). For each loop, a vertex v is dequeued from the PQ. Lines 16-18 check if the vertex v has an update, and proceeds to the next dequeue if not. Crucially, the if statement at line 18 also evaluates to true if vertex v has never been popped before. If v has been updated (or is being visited for the first time), also update the `visible_core` of v. Next, for every neighbor u of v, we first check if updates are possible (if v's coreness is less than u's degree). If so, lines 22-23 update the `update_hist` arrays of u, representing an update from v to u. Lines 24-31 handle coreness updates (decrement to `current_core`), updates to `ecp`, and priority updates via `decrementMin`. The final parallel `C++` PARKore code can be found in appendix A.1.

Algorithm 2 PARKore

```
function INIT(P, G)
    for v ∈ G.V do
        current_core[v] = v.deg
        ecp[v] = 0
        update_hist[v] = [0]*v.deg + [v.deg]
        visible_core[v] = n
        P.PUSH(v)
    end for
end function

function PARKORE_RUN(P, G)
    if P.EMPTY( )then
        return
    end if
    v = P.DEQUEUEMIN( )
    old_core = visible_core[v]
    est_core = current_core[v]
    if old_core != est_core then
        visible_core[v] = est_core
        for u ∈ v.neighbors do
            if est_core < u.deg then
                update_hist[u][old_core]−−          ▷ unseen bounds check on old_core here
                update_hist[u][est_core]++
                if old_core >= current_core[u] and est_core < current_core[u] then
                    if ecp[u] > 0 then
                        ecp[u]−−
                    else
                        current_core[u]−−
                        ecp[u] = update_hist[u][current_core[u]]
                        P.DECREMENTMIN(u)
                    end if
                end if
            end if
        end for
    end if
end function

function PARKORE(G)
    P = PriorityQueue
    INIT(G)
    while true do
        PARKORE_RUN(P, G)
    end while
end function
```

To demonstrate PARKore in action, lets revisit the k-core decomposition example in figure 2.2. The incorrect solution shown in figure 2.4 arose when a yellow vertex was prematurely dequeued, and subsequently dropped the blue vertex's coreness. Consider a point T_0 in the algorithm just after the yellow vertex dequeues and updates the blue vertex, but no other vertex has dequeued. A snapshot of the graph at T_0 is depicted in figure 3.1.

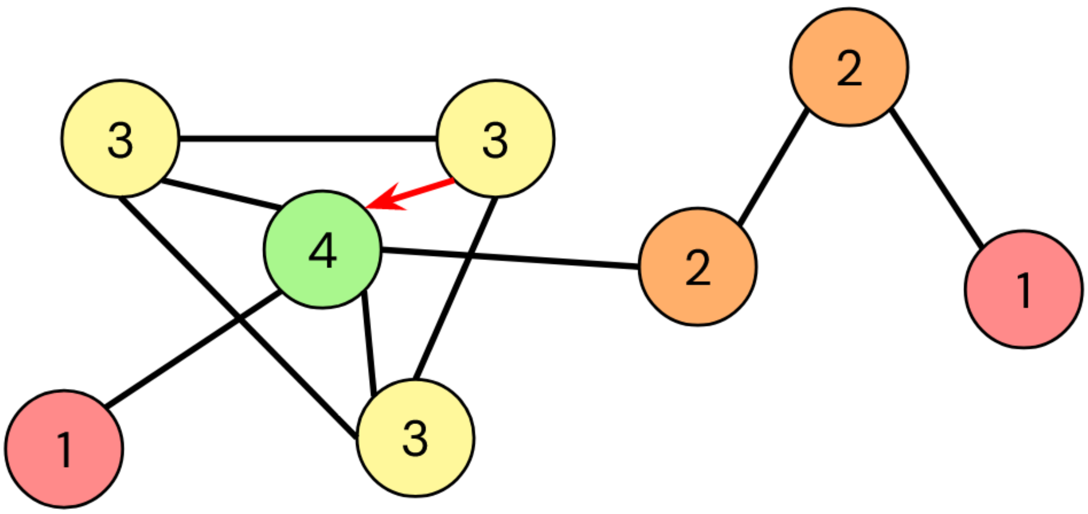

Figure 3.1: A snapshot of a k-core decomposition example after a yellow vertex updates the central vertex at T_0.

As the example progresses, we can track the state variables for the central (previously blue) vertex in table 3.2. Start and end indicate the begin and completion of the main loop (lines 41-43 in algorithm 2). The red text indicates the index corresponding to the central vertex's current coreness, as a position within the `update_hist` array. Alternatively, the red element in the table is also the number of equal core predecessors at the `current_core` of the central vertex. Finally, note that `visible_core` is initialized to a large number, defaulted to n, since the max degree of any node cannot be larger than n.

The first row of table 3.2 shows initialization of state variables. Then, at T_0 the yellow vertex dequeues and causes an update to the central vertex, represented as an increment in index 3 of `update_hist`, as the update source was a core 3 vertex (yellow vertex). Consequently, the `current_core` is also decremented at this time. T_1 represents the cycle after the bottom left red vertex decrements and causes an update to the central vertex. As a result, an increment in index 1 of `update_hist`

Time	0	1	2	3	4	5
start	0	0	0	0	0	0
T_0	0	0	0	1	0	0
	⋮	⋮	⋮	⋮	⋮	⋮
T_1	0	1	0	1	0	0
	⋮	⋮	⋮	⋮	⋮	⋮
T_2	0	2	0	1	0	0
T_{dq}	0	2	0	1	0	0
end	0	2	0	1	0	0

Time	`current_core`	`visible_core`	`ecp`
start	5	8	0
T_0	4	8	0
	⋮	⋮	⋮
T_1	3	8	**1**
	⋮	⋮	⋮
T_2	3	8	**0**
T_{dq}	3	3	0
end	3	3	0

Table 3.2: State variables for vertex in example. Left: `update_hist`. Right: Other.

is applied, and `current_core` is decremented again. At time T_2, the (previously) orange vertex to the right of the central vertex dequeues, and applies its update. This vertex applies an increment in index 1 of `update_hist`, but when it goes to decrement the `current_core` of the central vertex, it finds it cannot. This is because `ecp` was previously set to 1 at time T_1, as indicated by the red 1 in `update_hist`. As such, `ecp` is decremented *instead of* the coreness value. Lastly, the central vertex dequeues at T_{dq}, and updates it's own `visible_core`, wrapping up the algorithm loop. At this point, the PQ is also empty, and we see that the coreness of the central vertex is three - matching the correct solution shown in figure 2.3.

We have thus shown that PARKore can handle the case when higher priority elements (yellow vertex with priority=3) are processed before lower priority elements. The key insight that allows PARKore to handle such inversions is the `ecp` state variable, which was decremented at time T_2 instead of the `current_core`. Specifically, PARKore maintains the following invariant in each iteration:

$$\texttt{current\_core}[v] = v.\text{deg} - \sum_{i=0}^{\texttt{current\_core}[v]-1} \texttt{update\_hist}[v] - \texttt{ecp}$$

Equivalently, this invariant can be rearranged as follows:

$$v.\text{deg} - \texttt{current\_core}[v] = \sum_{i=0}^{\texttt{current\_core}[v]-1} \texttt{update\_hist}[v] + \texttt{ecp}$$

Observe that the left hand side constitutes the sum of updates to a vertex v's coreness. Then, the invariant maintains that all updates which have cause a vertex's decrement originates from one of two sources. First, updates from a lower degree vertex tracked in `update_hist`. Or second, updates from vertices with equal coreness that popped before v but now could be reordered after v without loss of correctness. Furthermore, since v.deg is constant, PARKore's invariant ensures that any contributions to decrementing v's coreness are matched by `ecp`, effectively blocking decrements to v's coreness until there are no more equal core predecessors.

As a result, PARKore is resilient to dequeue's in any order, enabling threads to work without needing barrier based synchronization or equal priority tasks. In essence, PARKore enables threads to work asynchronously on tasks which need not have equal priority to tasks in other threads.

However, PARKore's state variables are shared between threads, which could also apply asynchronous contention as multiple threads attempt to load data from a shared memory location. Also, PARKore has a large critical section (lines 21-33 in algorithm 2) which requires locking. We describe optimizations to reduce atomic and lock overheads, as well as enable multithreading in section 3.2.

3.2 Optimizations and Multithreading

First, we pack all state variables into 64b structs, aligning with 64b cache lines common in modern CPU's to guarantee only the first memory miss per access to more than one state variables. We also reduce unnecessary pointer dereferences in `update_hist`, by compressing `update_hist` into Compressed Sparse Row (CSR) format [32]. As such, only one array index is required, as opposed to two dereferences using the 2D arrays method.

Although previously unmentioned, the comparison and subsequent write on lines 18-19 is implemented as a Compare and Swap (CAS) using `C++20` offerings, to ensure that threads operating on a vertex v correctly write to shared memory (and avoid missing updates to v from other threads). Additionally, to lock the critical section in lines 21-33 of algorithm 2, we avoid mutex locks and instead opt for atomic based locks. We implement both Test and Test and Set (TTAS) locks and Reader/Writer (RW) locks, with the reader locks securing the main critical section, and updates accuumulated and deferred to a writer locked section occuring after a vertex v is dequeued. We find that the RW lock outperformed TTAS by a factor of 5x across graphs, and is therefore chosen as the default locking structure in PARKore.

To run PARKore, we utilize `std::thread`, with each thread sharing a common priority scheduler, graph memory and state variable arrays. Lastly, note that the initialization and run functions in algorithm 2 share the same inputs. We parallelize both portions by dividing initialization amongst all threads, which share a single global barrier to ensure proper synchronization of the state variables. No other barriers are used in PARKore.

3.3 MultiQueues as Priority Schedulers

Recall that the original insight of the BZ algorithm [4] is that utilizing a priority queue limits the total number of pops to $O(V)$, and updates to $O(2E)$. Despite PARKore's resilience to priority inversions and ability to schedule in a relaxed manner, applying an optimal ordering still results in lower overall dequeues and updates, and higher work-efficiency. To this end, PARKore utilizes a Concurrent Priority Scheduler (CPS) which enables PARKore to have work-efficiency, as the priority scheduler approximates a global priority ordering. Explicitly, we apply a Bucket MultiQueue as well as a base MultiQueue to PARKore, benchmarking them against each other and choosing the more performant option. The MultiQueue (MQ) utilizes heaps for the underlying priority queue, with an interface for push/pop (no priority updates). We refer to this MQ as the MQIO (MultiQueue Input/Output).

The Bucket MultiQueue (BMQ) is a novel idea, which combines the array of PQ structure of a MQ, but switches out the underlying datatype. Instead of using a heap (priority queue), the BMQ utilizes a bucketing structure, akin to the buckets found in Julienne [9]. This reduces queue overheads by cheapening access to the underlying data. The BMQ shares the same interface as the MQIO.

Both the MQIO and the BMQ have tunable parameters which can effect dynamic runtime performance. Both MQ types have tunable number of queues, batch enqueue size, and batch dequeue size. Batching involves reserving two thread-local buffers (one for push, one for pop) and writing to these buffers in push or pop calls. When one of the thread-local buffers fills up, all of its tasks get sent to the MQ in bulk. Batching has previously been shown as an optimization for MultiQueue and its variants [29]. We investigate the dynamic effects of batch sizes for various graphs, and report these results in section 4.5

Additionally, the BMQ has two additional parameters over the MQIO. These are the number of buckets, and a delta parameter. The delta parameter dictates how a priority id is shifted into bucked id, commonly expressed as

$$\mathrm{ID}_{\mathrm{bucket}} = \mathrm{priority} >> \mathrm{delta}$$

Where >> is the bitwise right shift operator. For example, delta = 0 implies that priority id is directly used to index into buckets. A list of default MultiQueue parameters is shown in table 3.3.

MQ Parameter	BMQ Default	MQIO Default
c	4	4
Num Queues	192	192
Batch Deqeue	Varying (See section 4.5)	Varying (See section 4.5)
Num Buckets	64	N/A
Delta	0	N/A

Table 3.3: Default MQ Parameters.

4. Evaluation and Results

We evaluate PARKore on six distinct benchmark graphs. First, we report the hardware platform and software framework used to evaluate PARKore. We find that PARKore matches state of the art performance on certain benchmarks, and investigate PARKore's performance on other benchmarks. Some suggestions for future work and optimizations are proposed from analysis of cache performance and application parameters. Finally, we characterize PARKore's performance sensitivity to MultiQueue parameters and other performance optimizations.

4.1 Methodology

All implementations of PARKore are written in `C++20` and compiled with `gcc` version 11.4.0, with `-O3` optimization. Threaded implementations are written using `std::thread` and compiled with `-pthread`. We use the Ligra [35] source code for loading adjacency graphs formatted in PBBS style [36]. For compiling Ligra and Julienne applications, we use OpenCILK 2.1 with `clang` version 16.06. Note that Ligra is also compiled with `C++14` and `-O3` optimization. Since Ligra was originally written for use with Cilk Plus, certain modifications were required to the original Ligra source code to facilitate the switch to OpenCILK 2.1. Internally, Ligra stores loaded graphs in Compressed Sparse Row (CSR) format [32], giving a base space usage of $O(E)$ for all applications.

Experiments are conducted on a 24-core (48-thread) shared workstations (ug253) hosted by the University of Toronto. This workstations contains two 2.1GHz Intel 12-core Silver 4310 Xeon processors in a two socket configuration, with 256GB of main RAM and Intel two-way hyper-threading. Cumulatively, the workstation has 1.8MB of L1 cache, 30MB of L2 cache, and 36MB of L3 cache. All experiments were run on 48 threads unless otherwise specified.

Graphs are chosen based on size and availability and are described in table 4.1. Predominantly, we chose road graphs and social network graphs. The social network graphs we used follow a power-law distribution, where most vertices have a moderate number of neighbors and a few vertices have many neighbors. Both graph data size and characteristics impact algorithm performance. Certain large graphs, such as Hyperlink, are massive enough to eclipse the available capacity in Last Level Cache (LLC), requiring cache misses and retrievels from lower level cache or memory during dynamic execution. On the other hand, the USA roads has a maximum degree of nine, with the majority of nodes having only degree four (representing intersections which usually intersect four roads). As a result, many vertices bin into the same priority level and could implicate performance trends for bulk-synchronous programs. For k-core, we additionally ensure all graphs are symmetric. Note that in table 4.1, n denotes number of nodes and m number of edges.

Graph	n	m	highest degree
Youtube	1157827	5975248	28754
Orkut	3072626	234370166	33313
USA Roads [1]	23947347	58333344	9
RMat	33554432	398555100	16417
Twitter [19]	41652230	2405026390	2997487
Hyperlink2012 [27]	101717775	3880015728	3032590

Table 4.1: Overview of graph datasets

Default BMQ and MQIO parameters are given in table 3.3. Based on the sensitivity studies (section 4.5), we chose MQIO and BMQ parameters for each graph independently, and utilize those parameters for the results in section 4.2.

4.2 Performance

Fig 4.1 compares the performance of PARKore and Julienne for both types of Multi-Queues (BMQ and MQIO). Results are shown as speedup (see Def.3) versus the best performing sequential BZ peeling algorithm. Our implementation of the BZ algorithm can be found in section A.2.

Definition 3 (Speedup)

$$\text{Speedup} = \frac{\text{Sequential Execution Time}}{\text{Parallel Execution Time}}$$

We find that in general, PARKore using the BMQ outperforms the MQIO. This indicates that the bucket structure performs well in k-core, and is able to reduce enqueue/dequeue overheads as compared to a min-heap or alternative priority queue datatype. Additionally, both versions of PARKore outperform Julienne on Orkut and Youtube, with as much as 1.8x speedup on Orkut. For the USA roads graph, Julienne is almost 2x faster than PARKore. We attribute this behaviour to the low number of buckets for roads graphs, where almost every vertex fits into a single bucket, and the Julienne's relatively low cost when processing tasks in a single bucket.

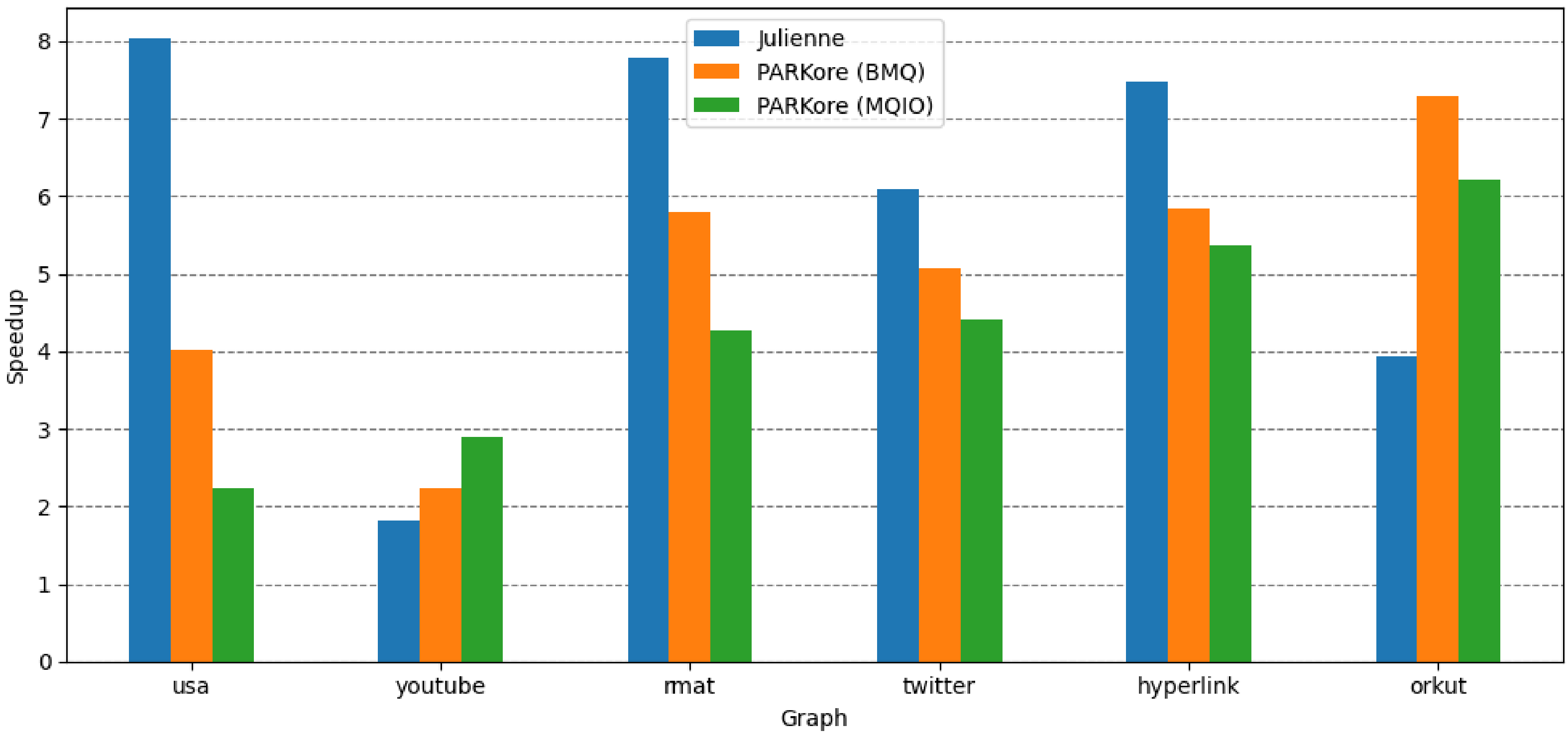

Figure 4.1: PARKore and Julienne **speedup** against work-efficient sequential implementation.

We find that Julienne outperforms PARKore on larger graphs, such as Twitter and Hyperlink; however, examining the scaling plots yield interesting features. figure 4.2 compares the performance of PARKore (with both MQ types) and Julienne as the system scales from 1 to 48 threads. At high thread counts (where hyper-threading is used extensively), Julienne's performance degrades relative to peak core counts, whereas PARKore maintains scaling. We suspect that this is due to compounded synchronization effects suffered by bulk-synchronous applications when CPU pipelines are completely multithreaded. figure 4.2 shows that PARKore's scaling bottoms out only on the youtube graph, suggesting further analysis is required. As such, one topic for future work is to examine PARKore's scaling performance on higher core counts.

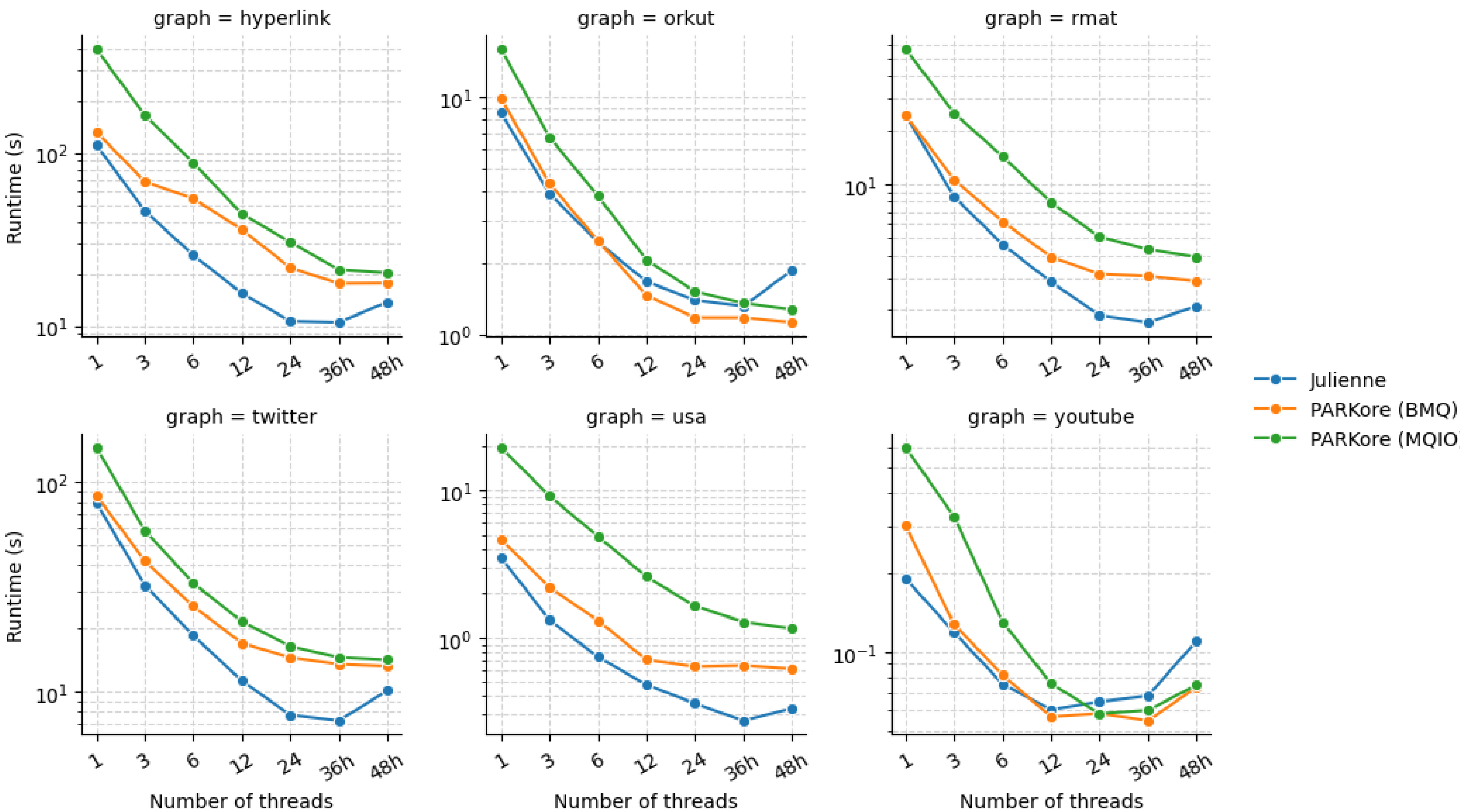

Figure 4.2: Running time of PARKore and Julienne. 48h refers to 48 hyper-threads

figure 4.3 shows the instruction count (IC) for all applications across all graph benchmarks. Even though PARKCore outperformed Julienne in 2 benchmarks, we note that Julienne's IC is actually higher than PARKore for most graphs. One apparent source of error is in the differences between build systems for Julienne (`clang`) and PARKcore (`g++`).

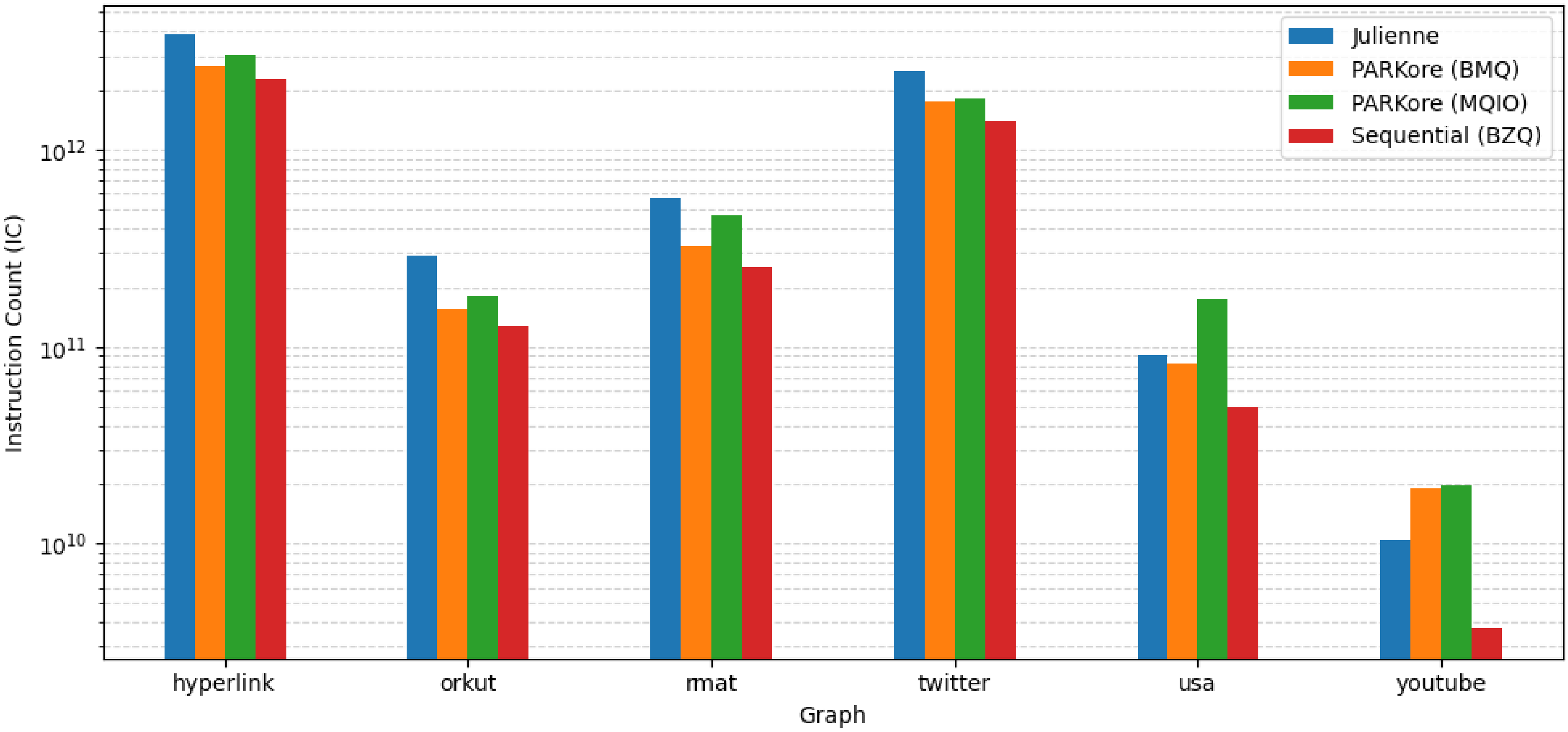

Figure 4.3: Instruction count of all applications.

However, IC alone is insufficient in informing performance differences between Julienne and PARKore. As a further analysis, we look at the cache performance of PARKore, as the state variabels in the PARKore algorithm are suspected to contribute a significant memory overhead that has implications for performance.

4.3 Cache Performance

We examine the cache performance of PARKore and evaluate it against the best sequential and Julienne programs. Cache misses, accesses, and dynamic instruction count are collected using Intel performance counters and reported with `perf stat`. Using `perf stat`, the following event counters were recorded for the running process (and all child threads): `instructions`, `LLC-loads`, `LLC-load-misses`, `LLC-stores`, and `LLC-store-misses`. Figure 4.4 reports the load and store misses in the last level cache (LLC, in this case, L3 cache) for all programs and graphs. We report load and store MPKI together, with the store MPKI stacked vertically on top of the load MPKI.

From figure 4.4, we see that across most graphs, Julienne has significantly lower MPKI compared to PARKore with either BMQ or MQIO. For Hyperlink, Orkut,

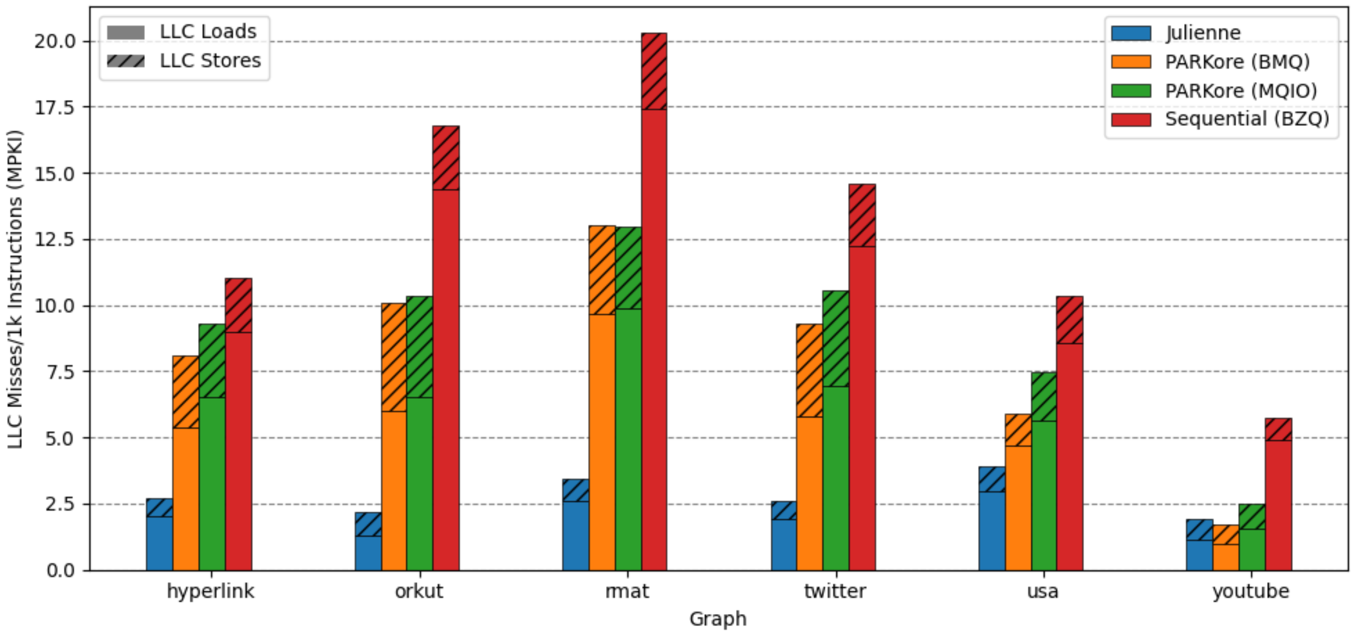

Figure 4.4: **MPKI**: LLC Load and Store misses across all graphs and applications.

RMat, and Twitter, PARKore has a 3-4x increase in the MPKI relative to Julienne. While Julienne has a base $O(E)$ space requirement, it does not have any other overhead to contribute compulsory misses (whereas PARKore does: the state variables). PARKore also exhibits a greater number of store misses compared to Julienne, which we attribute to the interplay between relaxed scheduling and the CPS. For both the BMQ and MQIO, no update operation is supported (only push), with each batched push writing to the underlying MQ structures.

Interestingly, PARKore has significantly lower MPKI compared to the sequential BZ implementation. We reason that this is partially due to lower dynamic instruction count for the sequential implementation, as well the existence of the BZQ structures. In the BZ algorithm, $O(2V + d_{max})$ space is required to instantiate the necessary memory structures for the BZ priority queue (d_{max} is the maximal degree in the graph). Additionally, these priority structures exhibit poor memory spatial locality, since priority decrements are modelled in the BZ algorithm as shifts of priority to a lower degree bin. Even though vertices are allocated in a contiguous array, the size of a single bin is bounded only by the number elements with the degree equal to bin id. As such, when the BZ algorithm is run on large graphs, it is likely for array indexing to cause cache when shifting between priority levels.

In figure 4.5, we examine the cache misses without factoring in instruction count (which varies drastically with program). Instead, in figure 4.5, cache accesses are normalized relative to the number of nodes in the graph (a proxy for graph size). We again show misses and accesses for both loads and stores stacked upon one another.

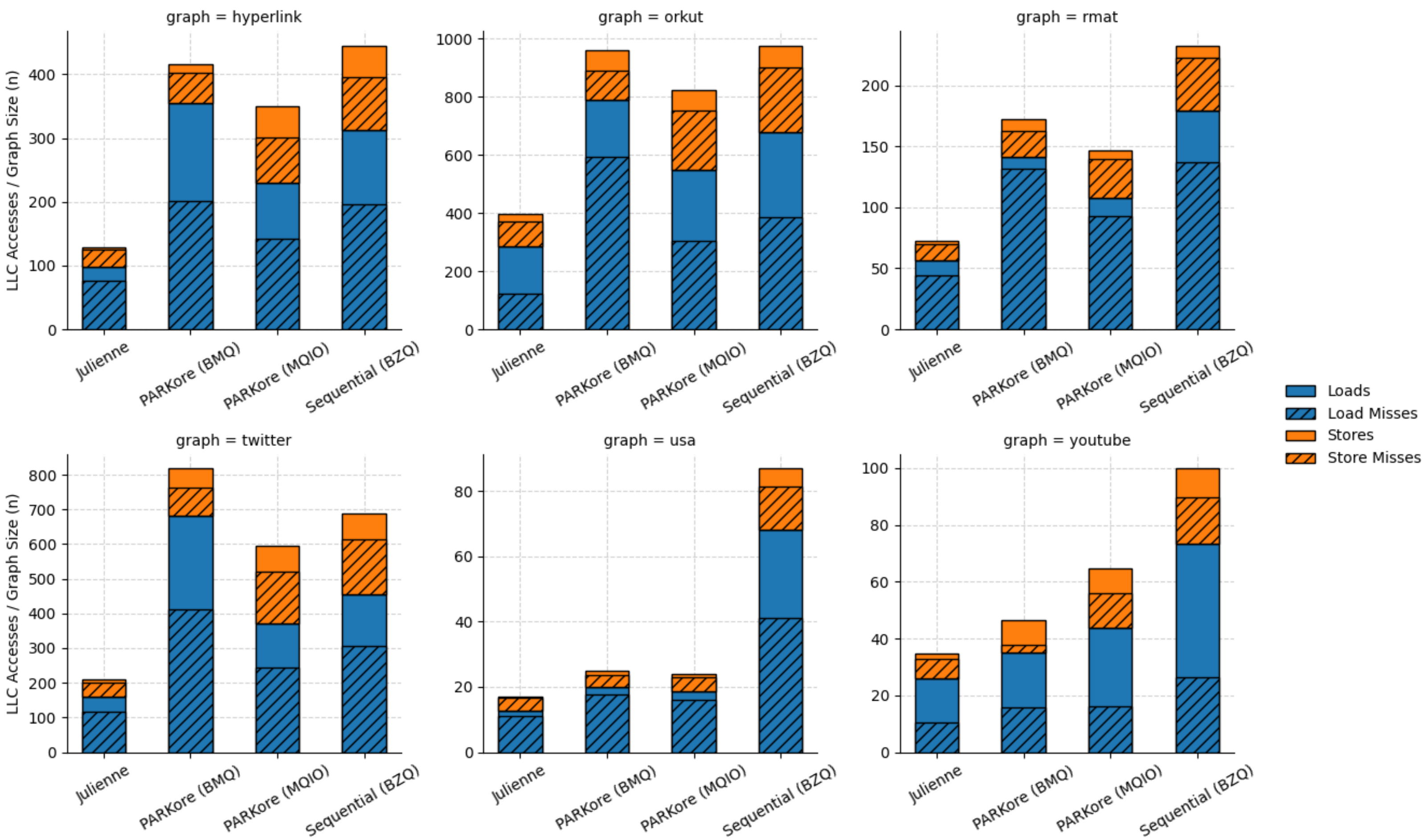

Figure 4.5: Cache accesses and misses across all graphs and applications, normalized to graph size n.

Consider the base cost for loading a graph G using the ligra framework [35]. Since graphs are stored in adjacency list format, there are $|E|$/cache line size compulsory cache misses for every application in figure 4.5. If we consider that compulsory misses for loading a graph G using the ligra framework are shared between programs, we can reason that a significant fraction of cache misses stem from PARKore's additional state variables, which require $O(3V + E)$ more space than the BZ algorithm [4]. This is evidenced by the data structure sizes provided in figure 3.1. In practice, the PARKore implementation uses a `64b` struct which packs the state variables `current_core`, `visible_core`, and `ecp`, as well as an array (in CSR format) for `update_hist`. Additionally, on the graphs where Julienne performs significantly better than PARKore (namely, USA and RMat), the cache miss rate for PARKore is extremely high (0.857 for PARKore on RMat, 0.844 for PARKore on USA). As such, we recommend that further analysis seek to bring down the memory usage for PARKore in attempts to improve cache hit rate.

4.4 Priority Scheduler Overheads

We investigate the difference between MQIO and BMQ by using time breakdowns of both applications. Specifically, we look for overheads contributed by the priority schedulers themselves. In figure 4.6, we plot the time breakdown for PARKore on all benchmarks, with the percent of execution time taken up by enqueue and dequeue operations. Across all benchmarks, enqueue costs are fairly minimal - pops from any MultiQueue are constant time given $c > 1$. However, the BMQ does not suffer from dequeue costs that the MQIO incurs, owing to the performance improvement of using a bucket structure as opposed to a heap in the MQIO. For USA roads, dequeue can take as much as 66% of the execution time when using a MQIO.

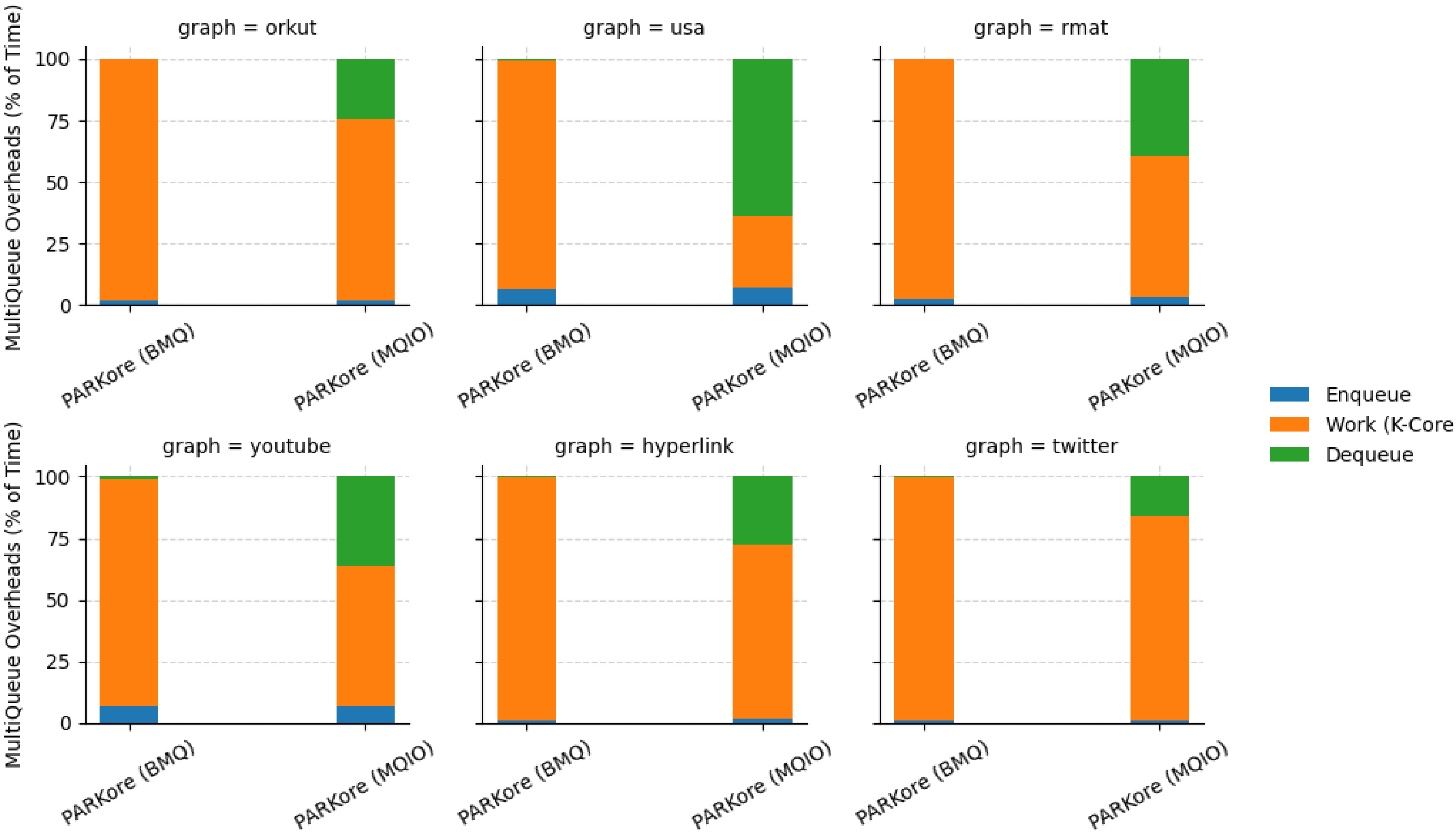

Figure 4.6: Time breakdown for PARKore using BMQ and MQIO.

4.5 Sensitivity to MultiQueue Parameters

In our experience, MultiQueue parameters can have significant impacts on performance. As such, we optimize results for MultiQueue parameters such as batched enqueue size and batched dequeue size by sweeping over possible values for each graph input. For results presented in previous sections, we use the following parameters on a per-graph basis for each MultiQueue configuration.

For MQIO, sweeps can be found in figure 4.7, where a sweet spot for most graphs occurs at enqueue batch sizes of $[16, 256]$ and dequeue batch sizes of $[4, 16]$. In these experiments, the graphs utilized prefer larger enqueue sizes relative to largeer dequeue sizes, which aligns with the notion that updates are more abundant than dequeues in k-core. One outlier of interest is USA roads, where dequeue batch size is directly correlated with speedup. The best performing batch settings (dequeue size $= 16384$, enqueue size $= 256$) on USA roads yield almost a 3x speedup compared to no batching. This trend can further be explained with the graph characteristics: road graphs contain lots of equal degree vertices, with only select few higher degree vertices. As such, many vertices do not contribute updates to their equal core neighbors, which suits a MultiQueue that can pop many vertices in constant time.

Bucket MQ sweeps are shown below for bucket sizes of 64 and 256 in figures 4.8 and 4.9 respectively. We note that beyond a certain point, bucket size has marginal effect on the performance of the BMQ. This is suggested by the similarity in performance numbers between figure 4.8 and figure 4.9. Again, best performing parameters were taken on a per-graph basis, with general trends for BMQ being that enqueue batch sizes around 256 being the most performant, whereas dequeue batch sizes have a bit more variance per graph. On Youtube, we find that the parameter sweep yields significant noise, as indicated by the variance in both the speedup of the best performing configuration, as well as the variance in parameter settings for this best performing config.

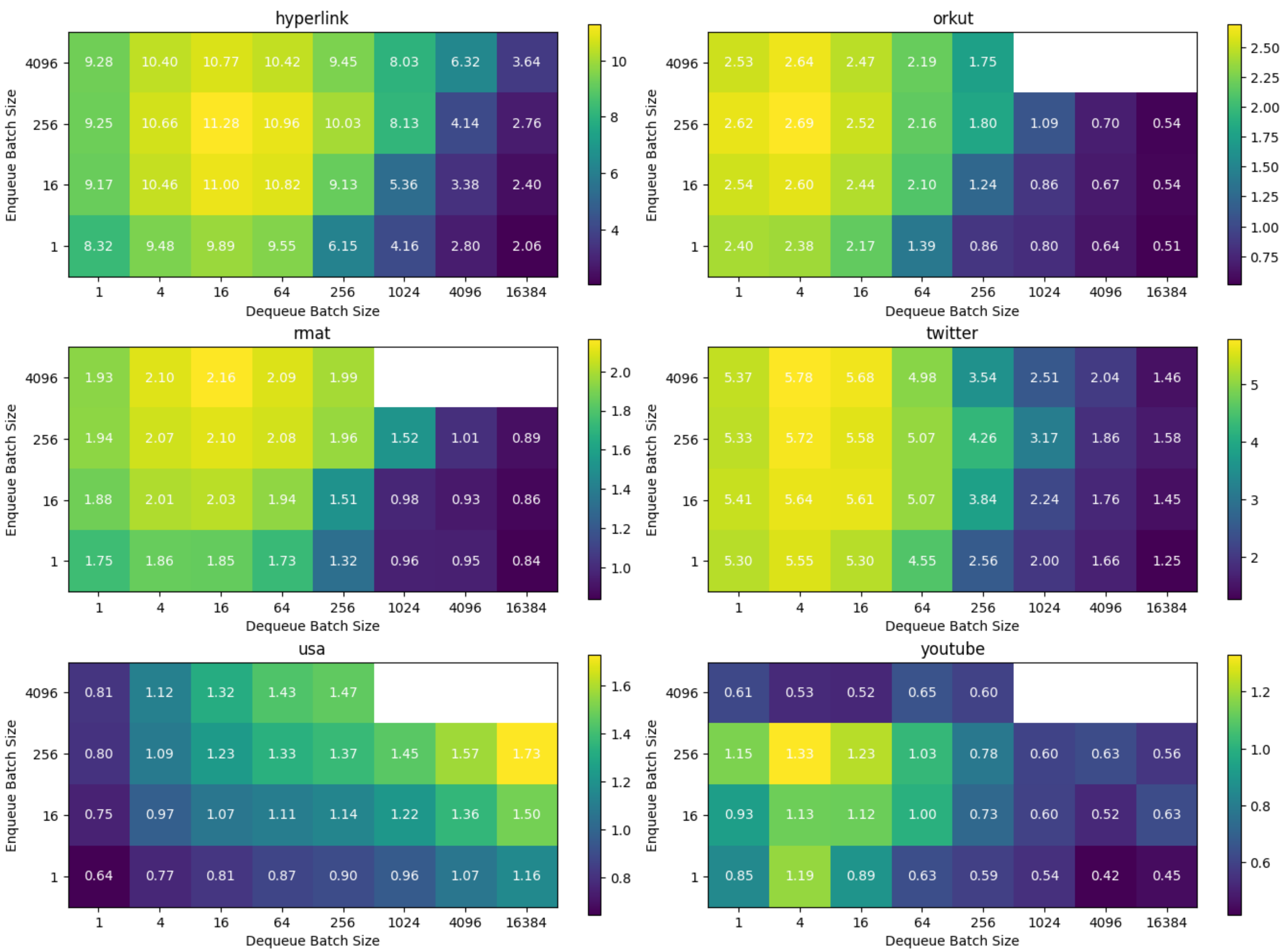

Figure 4.7: **Speedup**: MQIO relative to best sequential.

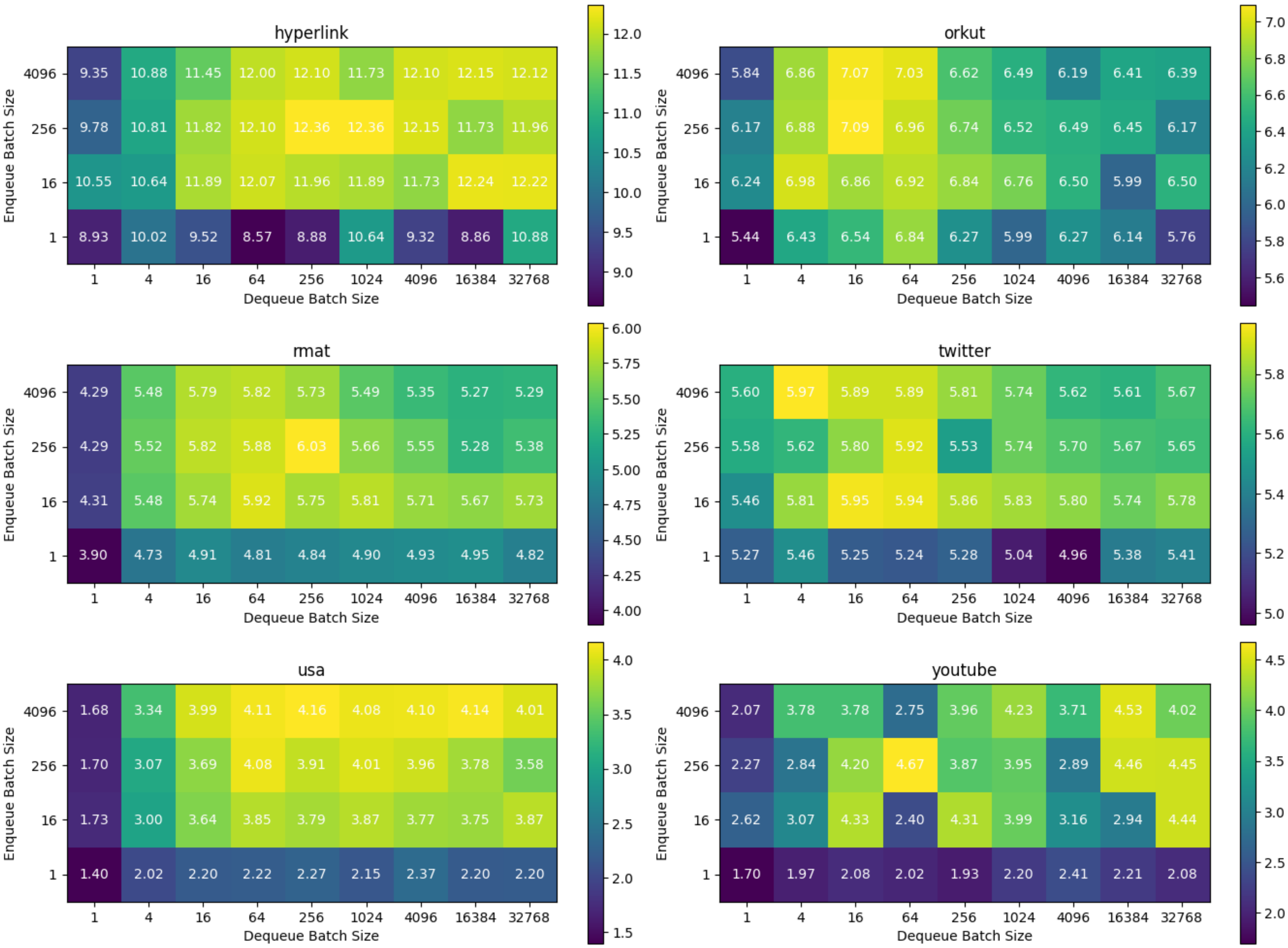

Figure 4.8: **Speedup**: Bucket MQ (64 buckets) relative to best sequential.

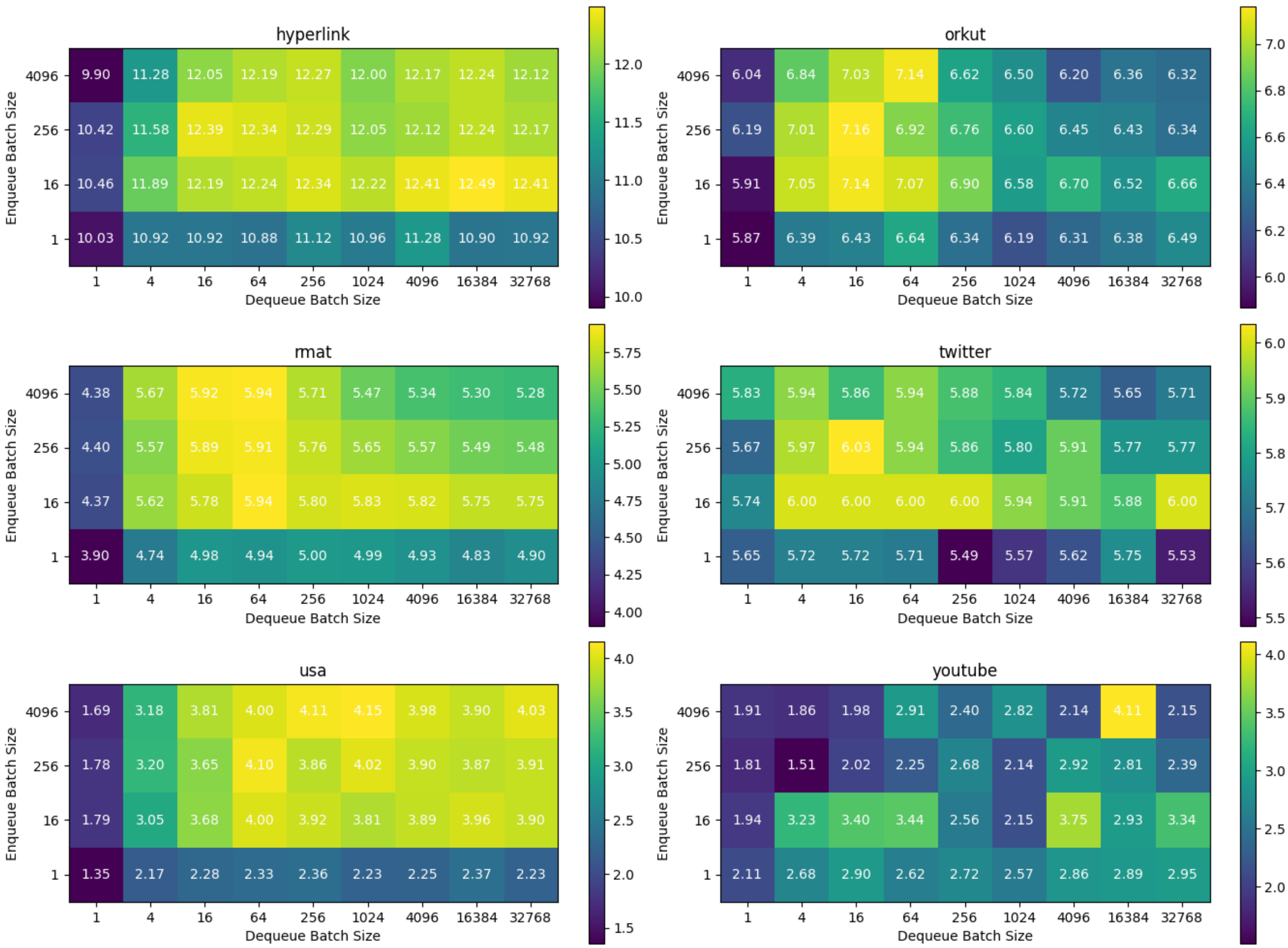

Figure 4.9: **Speedup**: Bucket MQ (256 buckets) relative to best sequential.

5. Conclusion and Future Work

Algorithms like k-core decomposition require a strict task ordering for correctness and work-efficiency. Additionally, task orderings are not only dependent on the graph input, but also hidden until runtime. When attempting to parallelize k-core, these requirement makes extracting parallelism difficult. Hardware centric versions for k-core decomposition either theorize speculative approaches using custom hardware structures, or are implemented bulk-synchronously using GPU's. Software solutions forego speculation as the performance consequences are severe on misspeculations. Asynchronous algorithms for k-core have previously not been thought possible. Thus, prior work in parallelizing k-core decomposition predominantly focus on bulk-synchronous approaches.

We introduced the first asynchronous, relaxed algorithm for k-core that is tolerant to relaxation in the scheduling order, as well as priority inversions from potential misspeculations on priority. Our results show that an algorithm of this class can be competitive with state of the art work, but require additional research and optimization.

5.1 Future Work

Moving forward, there are immediate directions for optimization. Firstly, the build system needs to be synchronized across programs. In our experiments, certain programs relied on `clang`, while others on `g++`. This has impacts on dynamic instruction count and performance that may result in discrepancies in our results. Additionally, we suffered difficulties during the evaluation of our results, namely due to usage of shared workstations. In the future, experimentation should strive to be conducted on high core count servers with low to no base load.

Additionally, during analysis of cache performance, we found that PARKore initiates many more calls to the cache than other programs, due to the additional memory cost from PARKore's state variables. By optimizing memory cost (and memory locality), it may be possible to improve the performance of PARKore even further.

One optimization is to reduce the `update_hist` array from $O(E)$ to $O(V)$ by creating equal size buckets for each vertex.

Lastly, future research outside of k-core could benefit from algorithms such as PARKore. Could other algorithms that are thought to have relaxed schedules also benefit from a relaxation algorithm of this type? Perhaps more generally, we wonder if it is possible to build a software framework that can generate relaxed algorithms given proper understanding of the algorithm invariants.

A. Appendix

A.1 PARKore Code

The following code uses different naming for state variables as compared to table 3.1. A mapping between state variables is provided below in table

State Variable	Name in Code
`current_core`	core
`update_hist`	histories
`visible_core`	activity
`ecp`	excess

Table A.1: State variable mappings

```
#include "ligra.h"
#include "utils.h"
#include <cassert>
#include <cstdlib>
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>
#include <thread>
#include <atomic>
#include <tuple>
#include <iostream>
#include <latch>

#include <boost/heap/d_ary_heap.hpp>

#include "BucketStructs.h"
#include "MultiQueue.h"
#include "MultiQueueUpdate.h"
#include "MultiBucketQueue.h"
#include "MultiBucketQueueUpdate.h"

#include "src/logger.h"
#include "src/utils.h"


// #define VALIDATE
#define VERBOSE

#define QUEUES_PER_THREAD 4

// QUEUE TYPES
using PQElement = std::tuple<uint32_t, uint32_t>;

struct queue_params
{
  size_t numBuckets = 0;
  size_t numQueues = 0;
  size_t delta = 0;
```

```
  size_t batchPushSize = 0;
  size_t batchPopSize = 0;
};

// GLOBALS
std::mutex cout_mutex;

/*
 * Contents:
 *  core: Current estimated coreness, equal to priority in the pq
 *  excess: number of scheduler dependences within this core, must reach 0
 *              to reduce core, and is decremented before core.
 *  activity: Previous estimated coreness used to adjust the coreness estimates
 *              for neighbours.  Becomes equal to core when the vertex is
 *              dequeued.
 *  pending: number of pending decrements to core/excess.  Accumulates as
 *              neighbours are dequeued, then applied to core/excess and
 *              set to 0.
 *  lock: a Reader/Writer lock for synchronization.  Pending may be incremented
 *              in Reader mode, core/excess require Writer to modify.  Activity
 *              is lock-free, as is enqueues.
 *  enqueues: Count of how many times this vertex is enqueued to the pq.  May
 *              become inaccurate if it saturates, which is fine.  The point is
 *              to make sure that there is always at least one entry in the pq
 *              if pending > 0, so we don't lose updates during termination
 *                  (note that I never saw this happen, but it theoretically
 *                  could)
 *                  */
struct vertex_data_t
{
  std::atomic_int32_t core;
  int32_t excess;
  std::atomic_int32_t activity;
  std::atomic_uint16_t pending;  // 16-bits to fit in 4 words still
  std::atomic_uint8_t enqueues;
  std::atomic_int8_t lock;
};

static_assert(sizeof(vertex_data_t) == sizeof(uint32_t) * 4);
static_assert(std::alignment_of_v<vertex_data_t> == std::alignment_of_v<uint32_t>);


inline bool attemptUpdate(vertex_data_t & v, std::atomic_int32_t * histories)
{
  bool update_occurred = false;
  if (v.pending.load(std::memory_order_relaxed) == 0) {return false;}
  {
    writer_guard w(v.lock);
    int32_t pending = v.pending.load(std::memory_order_relaxed);
    int32_t core = v.core.load(std::memory_order_relaxed);
    if (pending == 0) {return false;}
    v.pending.store(0, std::memory_order_relaxed);
    do {
      pending -= v.excess;
      if (pending > 0) {
```

```
        pending--;
        core--;
        v.excess = histories[core].load(std::memory_order_relaxed);
        update_occurred = true;
        if (v.excess < 0) {
          pending -= v.excess;
          v.excess = 0;
        }
      } else {
        v.excess = -pending;
      }
    } while (pending > 0);
    if (update_occurred) {v.core.store(core, std::memory_order_relaxed);}
  }
  return update_occurred;
}

template<bool USE_TTAS, bool USE_BMQ, bool USE_UPDATE, typename MQType>
void thread_run(
  const uint32_t t_id,
  graph<symmetricVertex> & G,
  const size_t n,
  const size_t m,
  std::vector<vertex_data_t> & vertex_data,
  std::vector<std::atomic_int32_t> & histories,
  MQType & mq)
{
  thread_local uint64_t ucoremin = 0;
  thread_local uint64_t ucoremax = 0;
  thread_local uint64_t ucoremiddle = 0;
  thread_local uint64_t excesseqz = 0;
  thread_local uint64_t pastupdatemin = 0;
  thread_local uint64_t update = 0;
  ucoremin = 0;
  ucoremax = 0;
  ucoremiddle = 0;
  excesseqz = 0;
  pastupdatemin = 0;
  update = 0;

  logging<log_msg_t> logger(m / 1024, ".", "pkcps_thread_" + std::to_string(t_id) + "
      .data");

  auto arr_offset_of_index = [&V = G.V,
      ZeroDeg = G.V[0].getOutNeighbors()](long v) ->  uint32_t {
      return V[v].getOutNeighbors() - ZeroDeg;
    };

  while (true) {
    uint32_t v;
    if constexpr(USE_UPDATE){
      auto item = mq.tryPop();
      if (item) {
        std::tie(std::ignore, v) = item.get();
      } else {
```

```
            break;
        }
    } else {
        auto item = mq.pop();
        if (item) {
            std::tie(std::ignore, v) = item.get();
        } else {
            break;
        }
    }


    #ifdef VALIDATE
    assert(v < n);
    #endif

    vertex_data_t & vdv = vertex_data[v];

    if constexpr (!USE_TTAS) {
        saturatingDecr(&vdv.enqueues);
        update += attemptUpdate(vdv, &histories[arr_offset_of_index(v)]);
    }

    int32_t new_act = vdv.core.load(std::memory_order_relaxed);
    int32_t old_act = vdv.activity.load(std::memory_order_relaxed);

    if (!updateMin(&vertex_data[v].activity, new_act, old_act)) {continue;}
    pastupdatemin++;

    #ifdef VALIDATE
    assert(new_act < old_act);
    assert(old_act > 0);
    #endif

    logger.log({0, v, new_act, old_act});

    uint32_t * const begin = &G.V[v].getOutNeighbors()[0];
    uint32_t * const end = begin + G.V[v].getOutDegree();
    for (uint32_t * it = begin; it != end; it++) {
        const uint32_t u = *it;
        vertex_data_t & vdu = vertex_data[u];
        if (vdu.core.load() <= new_act) {
            // Necessarily:
            //      new_act < old_act
            //      and
            //      vdu.core <= G.V[u].getOutDegree()
            // making a comparison of new_act with old_act and degree moot
            ucoremin++;
            continue;
        }

        const uint32_t hist_offset = arr_offset_of_index(u);
        bool update_occurred = false;
        bool attempt_update = false;
        int32_t ucore, pending;
```

```
    {  // acquire
      std::conditional_t<USE_TTAS,
        ttas_guard<int8_t>,
        reader_guard<int8_t>
      > g(vdu.lock);

      ucore = vdu.core.load(std::memory_order_relaxed);
      if (old_act < ucore) {
        // It's possible for old_act to exceed u's degree, so we need to test
        // some equivalent condition. u's core is a lower bound on its degree
        // and exhibits better cache locality
        histories[hist_offset + old_act].fetch_sub(1, std::memory_order_relaxed);
        ucoremax++;
      }
      histories[hist_offset + new_act].fetch_add(1, std::memory_order_relaxed);
      if (new_act < ucore && ucore <= old_act) {
        ucoremiddle++;
        if constexpr(USE_TTAS) {
          for (int32_t num_decs = 1; num_decs > 0; num_decs--) {
            if (vdu.excess == 0) {
              ucore--;
              int32_t h = histories[hist_offset + ucore];
              if (h < 0) {num_decs -= h;}
              vdu.core.store(ucore, std::memory_order_relaxed);
              vdu.excess = max(0, h);
              update_occurred = true;
            } else {
              vdu.excess--;
            }
          }
        } else {
          pending = vdu.pending.fetch_add(1, std::memory_order_acquire) + 1;
          if (pending > (1 << 12) || pending >= (ucore - new_act) >> 1) {
            attempt_update = true;
          } else if (vdu.enqueues.load(std::memory_order_relaxed) == 0) {
            // If we added pending, make sure that we
            // update the vertex at some point in the future
            // by enqueueing it if it isn't already enqueued
            // and we don't update it right now
            update_occurred = true;
          }
        }
      }
    }  // release
    if (attempt_update) {
      update_occurred = attemptUpdate(vdu, &histories[hist_offset]);
      if (update_occurred) {update++;}
    }
    if (update_occurred) {
      excesseqz++;
      if constexpr(!USE_TTAS) {saturatingIncr(&vdu.enqueues);}
      if constexpr(USE_UPDATE) {
        mq.updateMin(vdu.core.load(std::memory_order_relaxed), u);
      }
      else
```

```
          {
            mq.push(vdu.core.load(std::memory_order_relaxed), u);
          }
        }
        logger.log({1, u, ucore, vdu.excess});
      }
    }
    #ifdef VERBOSE
    {
      std::lock_guard<std::mutex> lock(cout_mutex);
      std::cout << "Thread " << t_id << " found: " << std::endl
                << "\tit passed the updateMin    " << std::setw(10) << pastupdatemin << " times" <<
        std::endl
                << "\tucore <= new_act < old_act " << std::setw(10) << ucoremin << " times" <<
        std::endl
                << "\tnew_act < ucore <= old_act " << std::setw(10) << ucoremiddle << " times " <<
        "(with excess==0 " << excesseqz << " times)" << std::endl
                << "\tnew_act < old_act < ucore  " << std::setw(10) << ucoremax << " times" <<
        std::endl
                << "\tand performed updates      " << std::setw(10) << update << " times" <<
        std::endl;
    }
    #endif
}

template<typename T>
inline std::tuple<T, T> get_range(size_t seg_id, size_t num_segs, size_t n_)
{
  assert(seg_id < num_segs);
  if (num_segs == 0 || num_segs == 1) {
    return std::make_tuple<T, T>(0, n_);
  }
  size_t range_size = n_ / num_segs;
  return std::make_tuple<T, T>(
    seg_id * range_size,
    ((seg_id < num_segs - 1) ? (seg_id + 1) * range_size : n_) - 1);
}

template<bool USE_TTAS, typename MQType>
void initialize(
  const uint32_t t_id,
  const uint32_t num_threads,
  graph<symmetricVertex> & G,
  const size_t n,
  const size_t m,
  std::vector<vertex_data_t> & vertex_data,
  std::vector<std::atomic_int32_t> & histories,
  MQType & mq
)
{
```

```
  auto mrange = get_range<uint32_t>(t_id, num_threads, m);
  auto nrange = get_range<uint32_t>(t_id, num_threads, n);

  #ifdef VALIDATE
  {
    std::lock_guard<std::mutex> lg(cout_mutex);
    std::cout << "thread " << t_id << " initializing with ranges "
              << "n=(" << std::get<0>(nrange) << "," << std::get<1>(nrange)
              << "), m=(" << std::get<0>(mrange) << "," << std::get<1>(mrange) << ")"
                  << std::endl;
  }
  #endif

  for (size_t i = std::get<0>(mrange); i <= std::get<1>(mrange); i++) {
    std::atomic_init(&histories[i], static_cast<int32_t>(0));
  }

  for (size_t i = std::get<0>(nrange); i < std::get<1>(nrange); i++) {
    vertex_data[i].core = G.V[i].getOutDegree();
    vertex_data[i].excess = 0;
    std::atomic_init(&vertex_data[i].activity, static_cast<uint32_t>(n));
    std::atomic_init(&vertex_data[i].lock, 0);
    if constexpr (USE_TTAS) {std::atomic_init(&vertex_data[i].enqueues, 1);}
    mq.push(G.V[i].getOutDegree(), i);
  }
}

template<bool USE_TTAS, bool USE_BMQ, bool USE_UPDATE, typename MQType>
void thread_task(
  const uint32_t t_id,
  const uint32_t num_threads,
  graph<symmetricVertex> & G,
  const size_t n,
  const size_t m,
  std::vector<vertex_data_t> & vertex_data,
  std::vector<std::atomic_int32_t> & histories,
  MQType & mq,
  std::latch & bar)
{
  if constexpr(!USE_UPDATE){
    mq.initTID();
  }
  initialize<USE_TTAS, MQType>(t_id, num_threads, G, n, m, vertex_data, histories, mq
    );
  bar.arrive_and_wait();
  thread_run<USE_TTAS, USE_BMQ, USE_UPDATE, MQType>(t_id, G, n, m, vertex_data,
    histories, mq);
}

template<bool USE_TTAS, bool USE_BMQ, bool USE_UPDATE, class vertex>
struct kcore
{
  graph<vertex> & G;
  size_t num_threads;
  queue_params & qparams;
```

```
  std::vector<uint32_t> operator()()
  {
    std::cerr << "Only symmetric vertex is supported (-s)\n";
    std::abort();
  }
};


template<bool USE_TTAS>
struct kcore<USE_TTAS, true, true, symmetricVertex>
{
  graph<symmetricVertex> & G;
  size_t num_threads;
  queue_params & qparams;

  std::vector<uint32_t> operator()()
  {
    size_t n = G.n;
    size_t m = G.m;

    std::vector<vertex_data_t> vertex_data(n);
    std::vector<std::atomic_int32_t> histories(m + 1);

    // threads
    std::vector<std::thread *> workers;

    // initialization latch
    std::latch bar{(ptrdiff_t)(num_threads)};

    using MQ_Bucket_Update = BucketMultiQueue<std::greater<uint32_t>, uint32_t,
        uint32_t, false>;

    MQ_Bucket_Update mq(n,
      qparams.numQueues,
      num_threads,
      qparams.delta,
      qparams.numBuckets,
      qparams.batchPopSize,
      qparams.batchPushSize,
      increasing
    );

    for (size_t t = 1; t < num_threads; t++) {
    #ifdef VALIDATE
      std::cout << "spawning worker " << t << "\n";
    #endif
      std::thread * worker = new std::thread(
        thread_task<USE_TTAS,true,true,MQ_Bucket_Update>,
        t,
        num_threads,
        std::ref(G),
        n,
        m,
        std::ref(vertex_data),
```

```
          std::ref(histories),
          std::ref(mq),
          std::ref(bar));
        workers.push_back(worker);
      }

      // spawn on thread 0
      thread_task<USE_TTAS,true,true,MQ_Bucket_Update>(0, num_threads, G, n, m, vertex_data, histories, mq, bar);

      // wait for thread exit
      for (std::thread * worker : workers) {
        worker->join();
        delete worker;
      }

      // print mq stats
      #ifdef VERBOSE
      mq.stat();
      #endif


      int32_t largestCore = 0;
      std::vector<uint32_t> cores(n);
      for (size_t i = 0; i < n; i++) {
        auto c = vertex_data[i].core.load(std::memory_order_relaxed);
        cores[i] = c;
        largestCore = std::max(largestCore, c);
      }
      cout << "largestCore was " << largestCore << endl;
      if (n == 3072626 && m == 234370166) {assert(largestCore == 253);}

      return cores;
    }
};

template<bool USE_TTAS>
struct kcore<USE_TTAS, true, false, symmetricVertex>
{
  graph<symmetricVertex> & G;
  size_t num_threads;
  queue_params & qparams;

  std::vector<uint32_t> operator()()
  {
    size_t n = G.n;
    size_t m = G.m;

    std::vector<vertex_data_t> vertex_data(n);
    std::vector<std::atomic_int32_t> histories(m + 1);

    // threads
    std::vector<std::thread *> workers;

    auto prefetcher = [](uint32_t v) -> void {};
```

```
    // initialization latch
    std::latch bar{(ptrdiff_t)(num_threads)};

    auto getBucketID = [&](uint32_t v) -> bucket_id {
        return bucket_id(vertex_data[v].core) >> qparams.delta;
      };

    using MQ_Bucket = MultiBucketQueue<decltype(getBucketID), decltype(prefetcher),
        std::greater<bucket_id>, uint32_t, uint32_t, false>;

    MQ_Bucket mq(getBucketID,
      prefetcher,
      qparams.numQueues,
      num_threads,
      qparams.delta,
      qparams.numBuckets,
      qparams.batchPopSize,
      qparams.batchPushSize,
      increasing
    );

    for (size_t t = 1; t < num_threads; t++) {
    #ifdef VALIDATE
      std::cout << "spawning worker " << t << "\n";
    #endif
      std::thread * worker = new std::thread(
        thread_task<USE_TTAS,true,false,MQ_Bucket>,
        t,
        num_threads,
        std::ref(G),
        n,
        m,
        std::ref(vertex_data),
        std::ref(histories),
        std::ref(mq),
        std::ref(bar));
      workers.push_back(worker);
    }

    // spawn on thread 0
    thread_task<USE_TTAS,true,false,MQ_Bucket>(0, num_threads, G, n, m, vertex_data,
        histories, mq, bar);

    // wait for thread exit
    for (std::thread * worker : workers) {
      worker->join();
      delete worker;
    }

    // print mq stats
    #ifdef VERBOSE
    mq.stat();
    #endif
```

```

    int32_t largestCore = 0;
    std::vector<uint32_t> cores(n);
    for (size_t i = 0; i < n; i++) {
      auto c = vertex_data[i].core.load(std::memory_order_relaxed);
      cores[i] = c;
      largestCore = std::max(largestCore, c);
    }
    cout << "largestCore was " << largestCore << endl;
    if (n == 3072626 && m == 234370166) {assert(largestCore == 253);}

    return cores;
  }
};


template<bool USE_TTAS>
struct kcore<USE_TTAS, false, true, symmetricVertex>
{
  graph<symmetricVertex> & G;
  size_t num_threads;
  queue_params & qparams;

  std::vector<uint32_t> operator()()
  {
    size_t n = G.n;
    size_t m = G.m;

    std::vector<vertex_data_t> vertex_data(n);
    std::vector<std::atomic_int32_t> histories(m + 1);

    // threads
    std::vector<std::thread *> workers;

    // initialization latch
    std::latch bar{(ptrdiff_t)(num_threads)};

    using UpdateQueueType = boost::heap::d_ary_heap<
    PQElement,
    boost::heap::compare<std::greater<PQElement>>,
    boost::heap::arity<4>,
    boost::heap::mutable_<true>
    >;
    using MQ_UpdateMin = MultiQueueUpdateMin<
        UpdateQueueType, UpdateQueueType::handle_type,
        std::greater<PQElement>, uint32_t, uint32_t
    >;

    MQ_UpdateMin mq(n, qparams.numQueues, num_threads);

    for (size_t t = 1; t < num_threads; t++) {
    #ifdef VALIDATE
      std::cout << "spawning worker " << t << "\n";
    #endif
      std::thread * worker = new std::thread(
```

```
        thread_task<USE_TTAS, false , true , MQ_UpdateMin>,
        t ,
        num_threads ,
        std :: ref (G) ,
        n ,
        m,
        std :: ref ( vertex_data ) ,
        std :: ref ( histories ) ,
        std :: ref (mq) ,
        std :: ref ( bar ) ) ;
      workers . push_back ( worker ) ;
    }

    // spawn on thread 0
    thread_task<USE_TTAS, false , true , MQ_UpdateMin>(0, num_threads , G, n , m,
        vertex_data , histories , mq, bar ) ;

    // wait for thread exit
    for ( std :: thread * worker : workers) {
      worker−>join () ;
      delete worker ;
    }

    // print mq stats
    #ifdef VERBOSE
    mq. stat () ;
    #endif

    int32_t largestCore = 0;
    std :: vector<uint32_t> cores (n) ;
    for ( size_t i = 0; i < n; i++) {
      auto c = vertex_data [ i ] . core . load ( std :: memory_order_relaxed ) ;
      cores [ i ] = c ;
      largestCore = std :: max( largestCore , c ) ;
    }
    cout << "largestCore was " << largestCore << endl ;
    if (n == 3072626 && m == 234370166) { assert ( largestCore == 253) ;}

    return cores ;

  }
};

template<bool USE_TTAS>
struct kcore<USE_TTAS, false , false , symmetricVertex>
{
  graph<symmetricVertex> & G;
  size_t num_threads ;
  queue_params & qparams ;

  std :: vector<uint32_t> operator () ()
  {
    size_t n = G.n;
    size_t m = G.m;

```

```
    std::vector<vertex_data_t> vertex_data(n);
    std::vector<std::atomic_int32_t> histories(m + 1);

    // threads
    std::vector<std::thread *> workers;

    auto prefetcher = [](uint32_t v) -> void {};

    // initialization latch
    std::latch bar{(ptrdiff_t)(num_threads)};

    using MQ_IO = MultiQueue<decltype(prefetcher), std::greater<PQElement>, uint32_t,
        uint32_t,
      false>;

    MQ_IO mq(prefetcher,
      qparams.numQueues,
      num_threads,
      qparams.batchPopSize,
      qparams.batchPushSize
    );

    for (size_t t = 1; t < num_threads; t++) {
    #ifdef VALIDATE
      std::cout << "spawning worker " << t << "\n";
    #endif
      std::thread * worker = new std::thread(
        thread_task<USE_TTAS,false,false,MQ_IO>,
        t,
        num_threads,
        std::ref(G),
        n,
        m,
        std::ref(vertex_data),
        std::ref(histories),
        std::ref(mq),
        std::ref(bar));
      workers.push_back(worker);
    }

    // spawn on thread 0
    thread_task<USE_TTAS,false,false,MQ_IO>(0, num_threads, G, n, m, vertex_data,
        histories, mq, bar);

    // wait for thread exit
    for (std::thread * worker : workers) {
      worker->join();
      delete worker;
    }

    // print mq stats
    #ifdef VERBOSE
    mq.stat();
    #endif
```

```
    int32_t largestCore = 0;
    std::vector<uint32_t> cores(n);
    for (size_t i = 0; i < n; i++) {
      auto c = vertex_data[i].core.load(std::memory_order_relaxed);
      cores[i] = c;
      largestCore = std::max(largestCore, c);
    }
    cout << "largestCore was " << largestCore << endl;
    if (n == 3072626 && m == 234370166) {assert(largestCore == 253);}

    return cores;
  }
};

template<class vertex>
void Compute(graph<vertex> & GA, commandLine P)
{
  bool printCores = P.getOption("-p");
  bool TTAS = P.getOption("-ttas");
  bool BQ = P.getOption("-bq");
  bool U = P.getOption("-u");
  size_t numWorkers = P.getOptionLongValue("-n", 1);
  size_t numBuckets = P.getOptionLongValue("-nb", 64);
  size_t numQueues = P.getOptionLongValue("-nq", QUEUES_PER_THREAD * numWorkers);
  size_t delta = P.getOptionLongValue("-d", 0);
  size_t batchPushSize = P.getOptionLongValue("-pushs", 1);  // enqueue batch size
  size_t batchPopSize = P.getOptionLongValue("-pops", 1);  // dequeue batch size

  queue_params qp = {
    .numBuckets = numBuckets,
    .numQueues = numQueues,
    .delta = delta,
    .batchPushSize = batchPushSize,
    .batchPopSize = batchPopSize
  };

  cout << "### application: parkcorecps";
  if (TTAS) {cout << " with ttas";} else {cout << " with rw lock";}
  if (BQ) {cout << " using bucket queue";} else {cout << " using push/pop queue";}
  if (U) {cout << " with update()";} else {cout << " using push()";}
  cout << endl;
  cout << "### graph: " << P.getArgument(0) << endl;
  cout << "### workers: " << numWorkers << endl;
  cout << "### n: " << GA.n << endl;
  cout << "### m: " << GA.m << endl;
  cout << "###" << endl;
  cout << "### queues: " << qp.numQueues << endl;
  if (BQ) {
    cout << "### numBuckets: " << qp.numBuckets << endl;
    cout << "### delta: " << qp.delta << endl;
  }
  cout << "### batchPushSize: " << qp.batchPushSize << endl;
  cout << "### batchPopSize: " << qp.batchPopSize << endl;

  std::vector<uint32_t> cores;
```

```

  if(TTAS && BQ && U){
    kcore<true, true, true, vertex> k{GA, numWorkers, qp};
    cores = k();
  } else if(TTAS && BQ && !U){
    kcore<true, true, false, vertex> k{GA, numWorkers, qp};
    cores = k();
  } else if(TTAS && !BQ && U){
    kcore<true, false, true, vertex> k{GA, numWorkers, qp};
    cores = k();
  } else if (TTAS && !BQ && !U){
    kcore<true, false, false, vertex> k{GA, numWorkers, qp};
    cores = k();
  } else if(!TTAS && BQ && U){
    kcore<false, true, true, vertex> k{GA, numWorkers, qp};
    cores = k();
  } else if(!TTAS && BQ && !U){
    kcore<false, true, false, vertex> k{GA, numWorkers, qp};
    cores = k();
  } else if(!TTAS && !BQ && U){
    kcore<false, false, true, vertex> k{GA, numWorkers, qp};
    cores = k();
  } else if (!TTAS && !BQ && !U){
    kcore<false, false, false, vertex> k{GA, numWorkers, qp};
    cores = k();
  }

  if (printCores) {
    cout << "cores: " << endl;
    for (int i = 0; i < GA.n; i++) {
      cout << i << " " << cores[i] << endl;
    }
  }
}
```

A.2 Sequential BZ Implementation

Below is a code listing for the sequential BZ algorithm implementation that all multithreaded programs were benchmarked to. Note that we additionally used this implementation to prove the correctness of the PARKore algorithm in single threaded execution. As such, this code shares the same variable mapping as seen in section A.1.

```
#include "ligra.h"
#include "utils.h"
#include <cassert>
#include <cstdlib>
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>

```

```
// #define VALIDATE

typedef struct vertex_data_t
{
  uint32_t cores;
  uint32_t excess;
  uint32_t activities;
  uint32_t pos;
} vertex_data_t;


void validate_vertex(
  uint32_t core_id,
  uint32_t degree,
  vertex_data_t & vertex_data,
  std::vector<int32_t> histories,
  uint32_t hist_offset)
{
  cout << "------ validate -----" << endl;
  cout << "core_id: " << core_id << endl;
  cout << "cores: " << vertex_data.cores << endl;
  cout << "degree: " << degree << endl;
  cout << "excess: " << vertex_data.excess << endl;
  cout << "hist_offset: " << hist_offset << endl;
  cout << "histories: \n";

  uint32_t sum_hist = 0;
  for (size_t i = 0; i < vertex_data.cores; i++) {
    cout << histories[i] << endl;
    sum_hist += histories[hist_offset + i];
  }
  cout << "sum_hist: " << sum_hist << endl;

  assert(vertex_data.cores == degree - sum_hist - vertex_data.excess);
}

template<class vertex>
std::vector<uint32_t> KCore(graph<vertex> & G, bool printCores = false)
{
  std::abort();
}

template<>
std::vector<uint32_t> KCore<symmetricVertex>(graph<symmetricVertex> & G, bool printCores)
{

  // init
  size_t largestCore = 0;
  size_t n = G.n;
  size_t m = G.m;

  std::vector<uint32_t> vert(n, 0);
  std::vector<vertex_data_t> vertex_data(n, {0, 0, static_cast<uint32_t>(n), 0});

```

```
  for (size_t i = 0; i < n; i++) {
    vertex_data[i].cores = G.V[i].getOutDegree();
  }

  std::vector<int32_t> histories(m + 1, 0);

  uint32_t md = std::max_element(
    vertex_data.begin(), vertex_data.end(),
    [](const vertex_data_t & a, const vertex_data_t & b)
    {return a.cores < b.cores;})->cores;
  std::vector<uint32_t> bin(md + 2, 0);

  for (size_t v = 0; v < n; v++) {
    bin[vertex_data[v].cores + 1]++;
  }

  std::partial_sum(bin.begin(), std::prev(bin.end()), bin.begin());

  std::vector<uint32_t> incr(md + 1, 0);
  for (size_t v = 0; v < n; v++) {
    vertex_data[v].pos = bin[vertex_data[v].cores] + incr[vertex_data[v].cores];
    vert[vertex_data[v].pos] = v;
    incr[vertex_data[v].cores]++;
  }

  auto arr_offset_of_index = [&V = G.V,
      ZeroDeg = G.V[0].getOutNeighbors()](long v) -> uint32_t {
    return V[v].getOutNeighbors() - ZeroDeg;
  };

  // begin main loop
  for (size_t i = 0; i < n; i++) {
    uint32_t v = vert[i];

    const uint32_t old_act = vertex_data[v].activities;
    const uint32_t new_act = vertex_data[v].cores;

    if (old_act == new_act) {
      continue;
    }

    vertex_data[v].activities = new_act;

    uint32_t * const begin = &G.V[v].getOutNeighbors()[0];
    uint32_t * const end = begin + G.V[v].getOutDegree();
    for (uint32_t * it = begin; it != end; it++) {
      const uint32_t u = *it;

      #ifdef VALIDATE
      std::cout << "i: " << i << " j: " << j << " u: " << u << " v: " << v << std::endl;
      std::cout << "deg(v): " << G.V[v].getOutDegree() << " deg(u): " << G.V[u].getOutDegree() <<
        std::endl;
      std::cout << "old_act: " << old_act << " new_act: " << new_act << std::endl;
```

```

    assert(new_act < old_act);
    assert(old_act > 0);
      #endif

    const uint32_t hist_offset = arr_offset_of_index(u);

      #ifdef VALIDATE
    int deg_cumulative = u;
    for (int z = 0; z < u; z++) {
      deg_cumulative += G.V[z].getOutDegree();
    }
    assert(deg_cumulative == hist_offset);
      #endif

    if (new_act >= G.V[u].getOutDegree()) {
      continue;
    }

    #ifndef VALIDATE
    if (old_act < vertex_data[u].cores) {
      assert(hist_offset + old_act < m);
      (histories[hist_offset + old_act])--;
    }
      #else
    (histories[hist_offset + old_act])--;
      #endif

    (histories[hist_offset + new_act])++;

    // v.activities >= u.cores && v.cores < u.cores
    if (old_act >= vertex_data[u].cores && new_act < vertex_data[u].cores) {
      // vertex_data[u] -= (vertex_data[u] > 0);
      // vertex_data[u].excess -= (vertex_data[u].excess > 0);
      if (vertex_data[u].excess > 0) { // ternary or satdec?
        vertex_data[u].excess--;
        std::abort();   // should never be reached in seq implementation
      } else {
        assert(vertex_data[u].cores >= 0);

        // update prio
        uint32_t du = vertex_data[u].cores;
        uint32_t pu = vertex_data[u].pos;
        uint32_t pw = bin[du];    // bin of v
        uint32_t w = vert[pw];    // first elem in bin

        if (u != w) {   // swap
          vertex_data[u].pos = pw;
          vert[pu] = w;
          vertex_data[w].pos = pu;
          vert[pw] = u;
        }
        bin[du]++;
        vertex_data[u].cores--;
        vertex_data[u].excess = histories[hist_offset + vertex_data[u].cores];
```

```
          #ifdef VALIDATE
          std::cout << "update! cores[" << u << "]: " << vertex_data[u].cores << std
              ::endl;
          #endif
        }
      }

      #ifdef VALIDATE
      validate_vertex(
        u,
        G.V[u].getOutDegree(),
        vertex_data[u],
        histories,
        arr_offset_of_index(u) + u);
      #endif
    }
  }

// end main loop
  for (size_t i = 0; i < n; i++) {
    if (vertex_data[i].cores > largestCore) {
      largestCore = vertex_data[i].cores;
    }
  }
  cout << "largestCore was " << largestCore << endl;

  std::vector<uint32_t> cores;

  std::transform(
    vertex_data.begin(),
    vertex_data.end(),
    std::back_inserter(cores),
    std::mem_fn(&vertex_data_t::cores));

  return cores;
}

template<class vertex>
void Compute(graph<vertex> & GA, commandLine P)
{
  bool printCores = P.getOptionValue("-p");
  cout << "### application: kcore-seq-bzq-aofs" << endl;
  cout << "### graph: " << P.getArgument(0) << endl;
  cout << "### workers: " << getWorkers() << endl;
  cout << "### n: " << GA.n << endl;
  cout << "### m: " << GA.m << endl;
  auto cores = KCore(GA, printCores);
  if (printCores) {
    cout << "cores: " << endl;
    for (int i = 0; i < GA.n; i++) {
      cout << i << " " << cores[i] << endl;
    }
  }
}
```

A.3 MPKI for LLC Load and Stores

Figures A.1 and A.2 show load misses and store misses respectively. Misses are measured using Intel hardware performance counters and reported by using the linux tool `perf stat`. We report MPKI as the number of misses per 1000 program instructions.

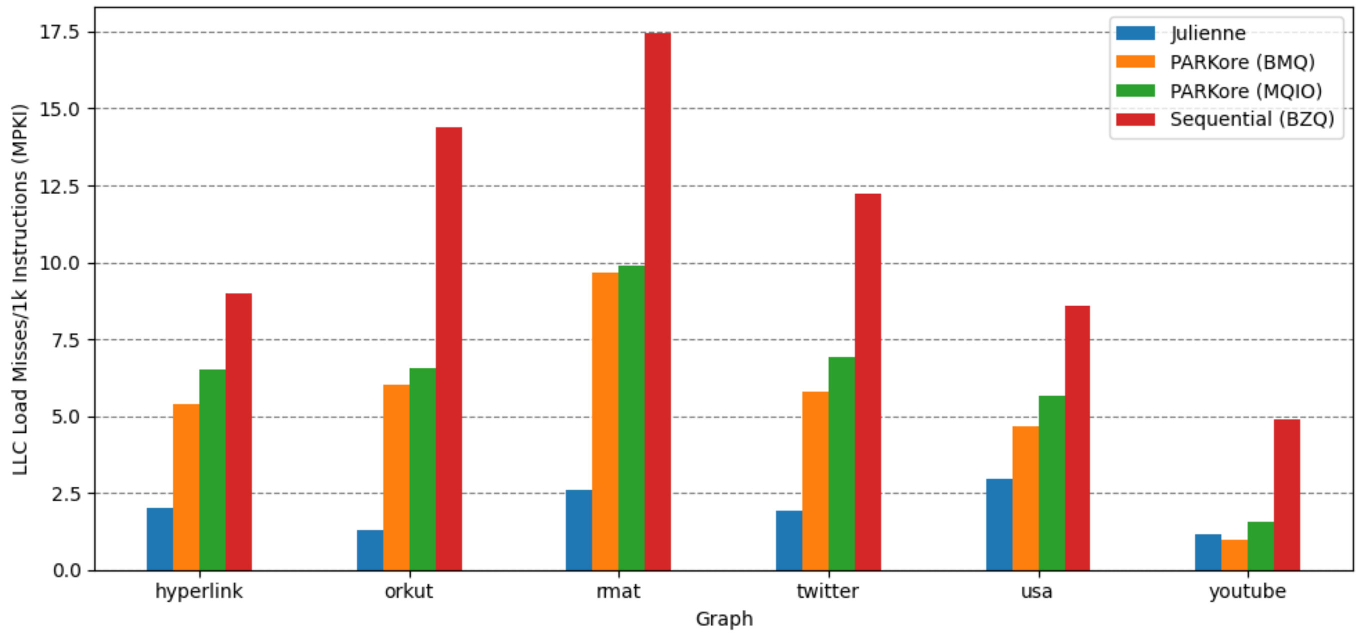

Figure A.1: **MPKI**: LLC Load Misses.

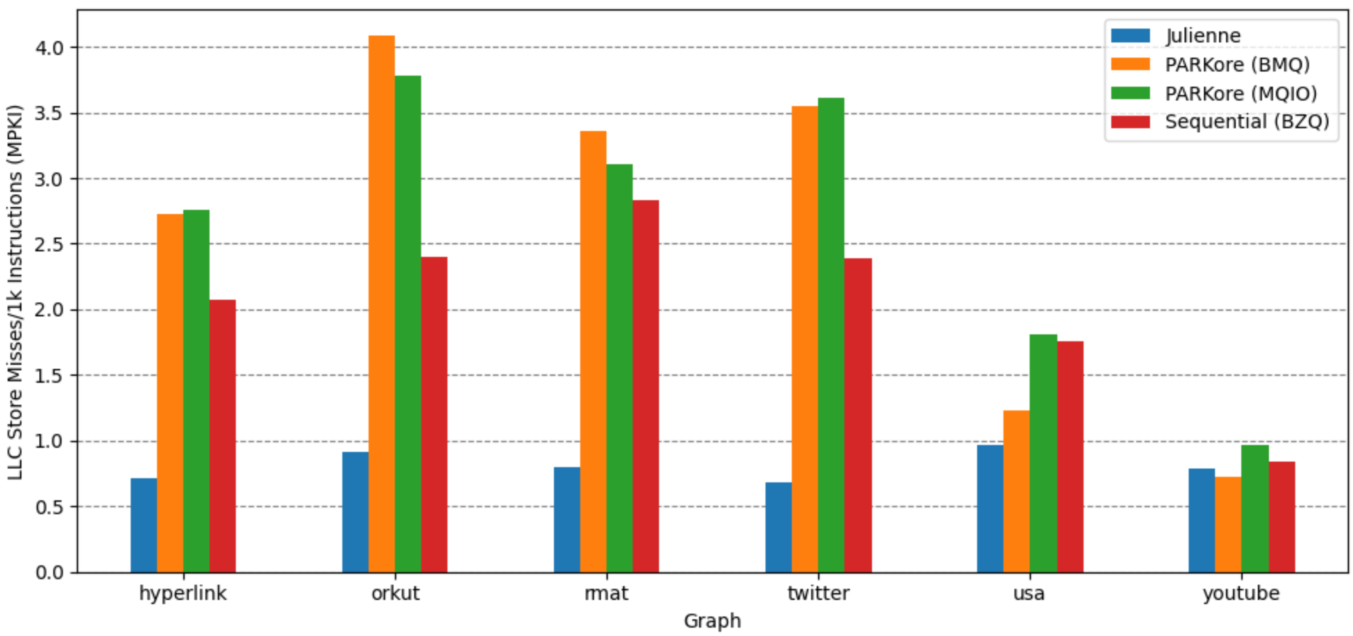

Figure A.2: **MPKI**: LLC Store Misses.

Bibliography

[1] 9th dimacs implementation challenge: Shortest paths, 2009.

[2] Akhlaque Ahmad, Lyuheng Yuan, Da Yan, Guimu Guo, Jieyang Chen, and Chengcui Zhang. Accelerating k-core decomposition by a gpu. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 1818–1831, 2023.

[3] Dan Alistarh, Trevor Brown, Justin Kopinsky, and Giorgi Nadiradze. Relaxed schedulers can efficiently parallelize iterative algorithms. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. ACM, July 2018.

[4] Vladimir Batagelj and Matjaz Zaversnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[5] Guy E Blelloch and Bruce M Maggs. Parallel algorithms. *ACM Computing Surveys (CSUR)*, 28(1):51–54, 1996.

[6] Davide Cellai, Aonghus Lawlor, Kenneth A. Dawson, and James P. Gleeson. Tricritical point in heterogeneous k-core percolation. *Physical Review Letters*, 107(17), October 2011.

[7] Madelaine Daianu, Neda Jahanshad, Talia M Nir, Arthur W Toga, Clifford R Jack, Jr, Michael W Weiner, Paul M Thompson, and Alzheimer's Disease Neuroimaging Initiative. Breakdown of brain connectivity between normal aging and alzheimer's disease: a structural k-core network analysis. *Brain Connect.*, 3(4):407–422, 2013.

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

[9] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, page 293–304, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Gal Elidan, Ian McGraw, and Daphne Koller. Residual belief propagation: Informed scheduling for asynchronous message passing. *CoRR*, abs/1206.6837, 2012.

[11] Santo Fortunato and Darko Hric. Community detection in networks: A user guide. *Physics Reports*, 659:1–44, 2016. Community detection in networks: A user guide.

[12] Linton C. Freeman. Centrality in social networks conceptual clarification. *Social Networks*, 1(3):215–239, 1978.

[13] Patric Hagmann, Leila Cammoun, Xavier Gigandet, Reto Meuli, Christopher J Honey, Van J Wedeen, and Olaf Sporns. Mapping the structural core of human cerebral cortex. *PLoS Biol.*, 6(7):e159, July 2008.

[14] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 2011.

[15] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A scalable architecture for ordered parallelism. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 228–241, 2015.

[16] David S. Johnson. Approximation algorithms for combinatorial problems. *Proceedings of the fifth annual ACM symposium on Theory of computing*, 1973.

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

[18] Maksim Kitsak, Lazaros K. Gallos, Shlomo Havlin, Fredrik Liljeros, Lev Muchnik, H. Eugene Stanley, and Hernán A. Makse. Identification of influential spreaders in complex networks. *Nature Physics*, 6(11):888–893, Nov 2010.

[19] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 591–600, New York, NY, USA, 2010. Association for Computing Machinery.

[20] Nir Lahav, Baruch Ksherim, Eti Ben-Simon, Adi Maron-Katz, Reuven Cohen, and Shlomo Havlin. K-shell decomposition reveals hierarchical cortical organization of the human brain. *New Journal of Physics*, 18(8):083013, aug 2016.

[21] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Priority queues are not good concurrent priority schedulers. In Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, volume 9233 of *Lecture Notes in Computer Science*, pages 209–221. Springer, 2015.

[22] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, jul 1983.

[23] Amir Mehrafsa, Sean Chester, and Alex Thomo. Vectorising k-core decomposition for gpu acceleration. In *Proceedings of the 32nd International Conference on Scientific and Statistical Database Management*, SSDBM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. *CoRR*, abs/1103.5320, 2011.

[25] Flaviano Morone, Kate Burleson-Lesser, H.A. Vinutha, Srikanth Sastry, and Hernán A. Makse. The jamming transition is a k-core percolation transition. *Physica A: Statistical Mechanics and its Applications*, 516:172–177, 2019.

[26] J.T. Oplinger, D.L. Heine, and M.S. Lam. In search of speculative thread-level parallelism. In *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)*, pages 303–313, 1999.

[27] Alexander Outman. Web data commons - hyperlink graphs, 2017.

[28] Gilead Posluns, Yan Zhu, Guowei Zhang, and Mark C. Jeffrey. A scalable architecture for reprioritizing ordered parallelism. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 437–453, New York, NY, USA, 2022. Association for Computing Machinery.

[29] Anastasiia Postnikova, Nikita Koval, Giorgi Nadiradze, and Dan Alistarh. Multi-queues can be state-of-the-art priority schedulers, 2021.

[30] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. Programming with relaxed synchronization. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, RACES '12, page 41–50, New York, NY, USA, 2012. Association for Computing Machinery.

[31] Hamza Rihani, Peter Sanders, and Roman Dementiev. Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, page 80–82, New York, NY, USA, 2015. Association for Computing Machinery.

[32] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second edition, 2003.

[33] Stephen B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, 1983.

[34] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. Corescope: Graph mining using k-core analysis — patterns, anomalies and algorithms. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 469–478, 2016.

[35] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 135–146, New York, NY, USA, 2013. Association for Computing Machinery.

[36] Julian Shun, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 68–70, 2012.

[37] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph, 2011.

[38] Marvin Williams, Peter Sanders, and Roman Dementiev. Engineering multi-queues: Fast relaxed concurrent priority queues. *CoRR*, abs/2107.01350, 2021.

[39] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. Optimizing ordered graph algorithms with graphit. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 158–170, New York, NY, USA, 2020. Association for Computing Machinery.