CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Understanding Work-Efficiency of label-correcting Single Source Shortest Path Algorithms

How does relaxed data structures affect the work-efficiency of parallel algorithms?

Master's thesis in Computer science – algorithms, languages and logic

Alfred Berglöf and Johan Berg

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Understanding Work-Efficiency of label-correcting Single Source Shortest Path Algorithms

How does relaxed data structures affect the work-efficiency of parallel algorithms?

Alfred Berglöf and Johan Berg

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Understanding Work-Efficiency of label-correcting Single Source Shortest Path Algorithms
How does relaxed data structures affect the work-efficiency of parallel algorithms?
Alfred Berglöf and Johan Berg

Supervisor: Kåre von Geijer, Department of Computer Science and Engineering
Examiner: Philippas Tsigas, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Understanding Work-Efficiency of label-correcting Single Source Shortest Path Algorithms
How does relaxed data structures affect the work-efficiency of parallel algorithms?
ALFRED BERGLÖF, JOHAN BERG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Relaxed priority schedulers have been successfully used to improve the scalability of parallel Single-Source Shortest Path (SSSP) algorithms by reducing synchronization overhead. However, relaxation introduces deviations from strict priority ordering, which may generate redundant work and reduce work-efficiency. This thesis investigates the relationship between scheduling relaxation and work-efficiency in label-correcting parallel SSSP algorithms.

To study this relationship, we extend an existing benchmarking framework with extra measurements. In addition, we introduce a configurable relaxed scheduler, DrPQ, which enables controlled experimentation with different degrees of relaxation. Experiments are conducted on road networks, Random Hyperbolic Graphs (RHGs), and grid graphs using several state-of-the-art relaxed priority scheduler implementations.

Our results show that relaxation error is correlated with redundant work, but is not sufficient on its own to explain work-efficiency. The relationship between relaxation and work-efficiency is influenced by graph structure, queue size, and algorithm design. High-degree RHGs generally tolerate larger relaxation errors while maintaining high work-efficiency, whereas road and grid graphs are more sensitive to the same level of relaxation. Furthermore, the skew between median rank error and delay provides additional information about scheduler behavior: some schedulers achieve high work-efficiency despite large skew, although this skew is not a universal predictor of redundant work.

Overall, the thesis demonstrates that work-efficiency in label-correcting parallel SSSP algorithms emerges from the interaction between relaxation errors, graph characteristics, and scheduler design. No single relaxation metric consistently predicts redundant work across all graph classes, reinforcing that work-efficiency depends on the interplay of multiple factors rather than any isolated measure of relaxation.

Keywords: Single-Source Shortest Path, Graph Algorithms, Dijkstra, MultiQueue, k-LSM, Rank Error, Delay

Acknowledgements

We would like to express our sincere gratitude to our supervisor, Kåre, for his continuous feedback, encouragement, and willingness to explore different ideas with us throughout this project. We are especially grateful for the opportunity to pursue this topic as a Master's thesis. His expertise, insightful guidance, and eagerness to help have been invaluable throughout the research process.

We would also like to thank Marvin Williams and Peter Sanders for making their benchmarking framework open source, upon request. This framework formed a central part of the experimental setup and was essential for producing the results presented in this thesis.

Finally, we would like to thank our fellow students for the many memorable lunch breaks shared throughout this project. Those moments of friendly distraction and good company were greatly appreciated and helped make the journey more enjoyable.

Alfred Berglöf and Johan Berg, Gothenburg, June 2026

Contents

List of Acronyms

k-LSM k Log-Structured Merge-tree. 12–14, 33, 45, 50, 51

CA-PQ Contention Avoiding Concurrent PriorityQueue. 14, 15, 28, 32, 44, 45, 50, 51, 60

CAS compare-and-swap. 3, 6

DrPQ Dijkstra relaxed Priority Queue. 22, 23, 27, 28, 32, 33, 44, 62

MBQ Multi Bucket Queue. 13, 14

MQ MultiQueue. 12–17, 20, 23, 32–34, 44

MQB MultiQueue-Balanced. 13–15, 33, 44, 51

MQF MultiQueue-Fast. 13–15, 33, 34, 51

MQQ MultiQueue-Quality. 13, 14, 33, 34, 51

PQ PriorityQueue. 1–3, 12, 13, 19, 22, 44, 50, 51

RHG Random Hyperbolic Graph. 12–14, 20, 21, 27, 32–34, 44, 45, 50, 51, 59, 61

SMQ Stealing MultiQueue. 13, 14, 28, 32, 50, 51, 60

SSSP Single-Source Shortest Path. 1–5, 8, 9, 11, 14–17, 19, 20, 22, 61, 62

List of Figures

List of Tables

1

Introduction

Amdahl's law is a cornerstone in the understanding of multiprocessing [1]. It states that there are diminishing returns to increasing the number of processors, as long as there is sequential execution somewhere in the workflow. A concrete example is the Single-Source Shortest Path (SSSP) problem, which is finding the shortest path between a source node and all other nodes. Smaller scale SSSP problems can be solved sequentially in reasonable time, but large-scale problems benefit from the computational power of multiprocessing. However, data structures designed for sequential execution become bottlenecks under concurrency, as their operations rely on sequential ordering and a single contention point [2].

In order to increase scalability, the multiprocessing inefficiencies needs to be minimized [1]. This can be achieved in different ways, but a major bottleneck is often the data structures used in schedulers that distribute the work [2]. There are different types of schedulers but the type this thesis is interested in analyzing is priority-based schedulers. In a priority-based scheduler, each task carries a priority, and higher-priority tasks should be processed before lower-priority ones. Strictly enforcing this order, however, can come at a performance cost [3], and the standard choice for such strict scheduling is the PriorityQueue (PQ) [4]. In a multiprocessing environment, contention over the PQ increases with the number of threads. As multiple processes request the highest-priority element, the data structure must serialize access, for example through locking. This undermines multiprocessing, as access to the data structure becomes effectively sequential, and thus a bottleneck in accordance with Amdahl's law [1].

Relaxation is a technique used to loosen the strict ordering guarantees of concurrent data structures [3]. Instead of enforcing perfectly ordered scheduling, a relaxed PQ allows some deviation from the optimal order. This reduces contention and can significantly improve throughput in parallel algorithms such as SSSP, at the cost of additional redundant work [5–7]. The extra work is commonly quantified as the ratio $\frac{\text{amount of work}}{\text{minimal amount of work}}$; following [8], we refer to this ratio as overhead. Work-efficiency is instead defined as $\frac{\text{minimal amount of work}}{\text{amount of work}}$, i.e., the fraction of work that directly contributes to the solution.

From an observational study, we could not find consistent metrics of relaxation. Al-

though relaxation is generally expected to increase redundant work, it remains unclear how strongly commonly used relaxation metrics correlate with work-efficiency in practice, and whether this relationship is consistent across different graph types and scheduler implementations. Consequently, there is currently no clear understanding of how well scheduler relaxation metrics predict algorithm work-efficiency. This is further explained in chapter 3.

This thesis explores how relaxation of priority scheduling affects work-efficiency in SSSP, with the goal of understanding how common relaxation metrics relate to work-efficiency and reduce the current research gap.

1.1 Aim

These are the central questions used to guide the project:

1. **Effects of relaxation:**
   How does the degree of relaxation affect work-efficiency?

2. **Graph characteristics:**
   How do different graph characteristics affect the performance of relaxed priority schedulers?

We aim to evaluate how relaxation errors correlate with improved work-efficiency of relaxed priority schedulers.

1.2 Background

The PQ is a widely used abstract data structure for schedulers. The order of execution in a PQ is what will be referred to as strict. The PQ supports two actions; `push` and `pop`. `Push` is the action of adding an element to the queue, it requires two arguments, a priority and a value. *Pop* is the action of removing the highest priority element from the queue, and returning its value, if the queue is empty `pop` returns an empty special value. One type of algorithm that can make use of PQs are path finding algorithms such as Dijkstra's [9, p.5-6]. The PQ ensures that no redundant work is performed, guaranteeing perfect work-efficiency. For sequential settings, PQs are a great fit, however when in a multiprocessing context the contention of the queue becomes a scalability issue [4].

When multiple threads attempt to access a data structure simultaneously it may lead to data races if the data structure is not specifically designed for concurrent execution. A data race occurs when multiple threads access the same memory location concurrently without synchronization, and at least one of the accesses is a write (modification). This is undesirable as it may lead to unpredictable, nondeterministic behaviors and impact the correctness of operations. A PQ may deal with this by implementing mechanisms that prevent unregulated concurrent access,

but this too comes at a cost [4].

One common approach to regulate access is using locks [1]. For a data structure to be locked simply means that no other thread except the lock owner can access or manipulate the data structure. The other threads must wait until the current operation is complete and the lock is released, at which point it can be claimed by another thread. Deciding which waiting thread becomes the new lock owner is typically done using an atomic *compareAndSwap()* operation, commonly referred to as CAS. An atomic operation appears to execute instantaneously, provided no other thread accesses the same location simultaneously; in practice this guarantees that at most one thread acquires the lock at a time. However, CAS-based locking does not guarantee fairness, a thread may starve and never succeed in obtaining the lock. Because all locked operations are serialized, a lock introduces a sequential bottleneck since only one thread can modify the queue at any moment.

Another solution is the lock-free approach to PQs [1, 10]. Lock-free data structures allow threads to modify shared memory locations using atomic operations such as CAS, while guaranteeing that at every moment at least one thread can make progress. If a CAS fails due to interference from another thread, the operation is retried until it succeeds. The lock-free approach allows `push` operations to be highly parallel. However the `pop` operations may still introduce a sequential bottleneck, as these threads have to contend with one another for the head of the queue. When the number of threads increase the contention increases, limiting scalability. Hence there is a need for a data structure with concurrent access, that have similar ordering properties to a PQ but allows for greater scalability.

When implementing concurrent data structures, there is often a trade-off between performance and strict ordering. Both lock-based and lock-free solutions sacrifice some performance to preserve a strict ordering guarantee. One approach to reducing this cost is to relax the ordering requirements, thereby reducing contention on individual memory locations. Data structures that employ this strategy are commonly used in relaxed priority schedulers. Relaxation has been shown to improve the performance of concurrent data structures by distributing contention across multiple locations, thereby reducing the sequential bottleneck and the associated scheduling overhead [3]. In the context of a PQ, as an example, the `pop` operation may return any of the top n elements rather than always removing the element with the highest priority. The usefulness of relaxation depends on the application. If maximizing throughput is the primary objective and strict execution order is not required, relaxation can be an effective design choice [11].

The SSSP problem concerns finding the shortest path from a single source node to all other nodes in a graph. This problem is widely used in applications such as network routing, traffic flow analysis, and modern map navigation. Parallel SSSP solvers aim to accelerate computation by exploiting multiprocessing. However, increased parallelism can introduce synchronization overhead and redundant computation, meaning that higher degrees of multiprocessing do not necessarily lead to faster

execution [12]. An optimal solution to SSSP is one that achieves perfect work-efficiency, where no computational effort is wasted. The classical implementation of Dijkstra's algorithm attains this property by always selecting the node with the smallest tentative distance from the constructed set, ensuring each node is processed exactly once. Maintaining this strict priority under concurrency, however, creates contention that limits scalability, motivating the use of relaxed scheduling.

A common approach to parallelizing Dijkstra's algorithm is to reformulate it as a label-correcting algorithm. In this context, a label refers to the current best-known distance from the source node to a given vertex. Unlike the classical label-setting variant, which processes each node exactly once in strict priority order, label-correcting algorithms allow nodes to be revisited and their labels to be updated multiple times. When combined with relaxed scheduling, this enables nodes to be processed outside strict priority order whilst producing the same result. This can introduce redundant work, since nodes may be processed more than once. While this reduces work-efficiency, it can increase throughput by lowering synchronization costs associated with strict priority scheduling [12].

In multiprocessing environments, prioritizing throughput over strict ordering can improve utilization of computational resources without compromising correctness of the final solution [4]. However, the extent to which approximate priority is maintained determines how much redundant work is introduced, and therefore how work-efficiency is affected [5]. Relaxation may also inflate queue sizes and thus cause potential memory constraints. Some data structures used for concurrent scheduling may store the same node in multiple queues simultaneously, bloating the queue size. Similarly, some others use thread-local queues, which can result in similarly large queue sizes. This may cause additional wasted work, or wasted memory, and some approaches may incur both.

1.3 Scope

This thesis focuses exclusively on the SSSP problem on fully connected directed graphs with non-negative edge weights. The central metric is work-efficiency, and how it relates to other metrics.

All experiments are conducted on a single multicore CPU, not investigating NUMA[1] machines, so that the analysis isolates the algorithmic trade-off between scheduling relaxation and redundant work, rather than hardware-dependent performance.

[1] **N**on-**U**niform **M**emory **A**ccess

2

Preliminaries

This chapter centers on the preliminaries that forms the foundation for how relaxed SSSP algorithms will be evaluated. Loosened ordering requirements, whilst allowing greater parallelism, may lead to additional work. By defining clear metrics the algorithms can be compared to one another and their level of relaxation can be quantified.

2.1 Dijkstra's implementation

The implementation of the sequential, label-setting, Dijkstra's algorithm is presented in Figure 2.1.

At a high level, the algorithm proceeds as follows:

- Row 3: The graph is read from an input file.
- Rows 5–7: Distance estimates are initialized to ∞ for all nodes.
- Rows 12–15: A priority queue is initialized, and the source node is inserted with distance 0.
- Row 17 and onward: The main execution loop begins. While the priority queue is non-empty, an element is extracted. For the extracted node, all outgoing edges are examined. If the newly computed distance to a neighbor exceeds the currently stored value, the entry is discarded. Otherwise, the distance is updated and the neighbor is inserted into the priority queue.

The `processedNodes` and `ignoredNodes` are used for generating log files and do not play a role in the execution. They count the total number of processed nodes as well as the number of ignored nodes. Each node is either counted as processed or ignored, it cannot be both.

The implementation of the parallel algorithm is presented in Figure 2.2. The implementation is label-correcting, which means that it may explore paths of sub-optimal distance from the origin to any given node, but the shortest path will eventually

```
1  void seqDijkstra(File graphFile, Int source):
2
3      Graph graph = read(graphFile)
4
5      Map <Node, Int> distances
6      foreach node in graph:
7          distances[node] = Inf
8
9      Int processedNodes = 0
10     Int ignoredNodes = 0
11
12     PriorityQueue<Int, Node> pq   // (distance, node)
13
14     distances[source] = 0
15     pq.push(0, source)   // (distance, source node)
16
17     while (!pq.empty()):
18         (dist, node) = pq.pop()
19
20         // Ignore stale entries
21         if dist > distances[node]:
22             ignoredNodes++
23             continue
24
25         // Explore neighbors
26         foreach edge in graph.neighbours(node):
27             neighbour = edge.target
28             weight = edge.weight
29
30             newDist = dist + weight
31
32             if newDist < distances[neighbour]:
33                 distances[neighbour] = newDist
34                 pq.push(newDist, neighbour)
35
36          processedNodes++
```

Figure 2.1: The label-setting, sequential Dijkstra implementation

be explored and overwrite the previously saved distance. Other than that small difference, the algorithms are similar.

In the parallel variant, contention on the shared distance map is typically low for real-world graphs, as parallel updates to the same node are relatively rare. To ensure correctness under multiprocessing, updates to the distance map are performed using a CAS operation (row 38). The priority queue is assumed to be thread-safe; its internal synchronization depends on the specific concurrent implementation used.

The counters for `processedNodes` and `ignoredNodes` are maintained using thread-local variables and can be aggregated after termination. Termination detection is handled externally to the main loop; the condition in the `while` statement ensures that all worker threads continue execution until global termination has been established. The specific mechanism for termination detection is beyond the scope of this work; we refer the reader to Williams and Sanders [5] for a detailed discussion.

```
1  void parDijkstra(File graphFile, Int source):
2
3      Graph graph = read(graphFile)
4
5      SharedMap<Node, AtomicInt> distances
6      foreach node in graph:
7          distances[node] = Inf
8
9      AtomicInt processedNodes = 0
10     AtomicInt ignoredNodes = 0
11
12     ConcurrentPQ <Int, Node> pq   // (distance, node)
13
14     distances[source] = 0
15     pq.push(0, source)  // (distance, source node)
16
17     parallel workers:
18         while (not terminated):
19
20             (dist, node) = pq.try_pop()
21
22             // Ignore stale entries
23             if dist > distances[node]:
24                 ignoredNodes++
25                 continue
26
27             // Explore neighbors
28             foreach edge in graph.neighbours(node):
29                 neighbour = edge.target
30                 weight = edge.weight
31
32                 newDist = dist + weight
33
34                 oldDist = distances[neighbour]
35
36                 // Atomic relaxation
37                 while newDist < oldDist:
38                     if CAS(distances[neighbour], oldDist, newDist):
39                         pq.push(newDist, neighbour)
40                         break
41
42             processedNodes++
```

Figure 2.2: The label-correcting, parallel Dijkstra implementation.

2.2 Graphs

A graph is formally defined as a tuple $G = (V, E)$, where V is a finite set of vertices (or nodes) and $E \subseteq V \times V$ is a set of edges. In this thesis, we consider simple graphs, i.e., graphs without self-loops, meaning that $(v, v) \notin E$ for all $v \in V$. We restrict our attention to weighted graphs, defined as $G = (V, E, w)$, where the weight function

$$w : E \to \mathbb{R}_{\geq 0}$$

assigns a non-negative cost to each edge. The value $w(u, v)$ represents the cost of traversing the edge from node u to node v, which is required for solving the SSSP problem. For a vertex $v \in V$, the degree of v is defined as

$$\deg(v) = |\{u \in V \mid (u, v) \in E\}|.$$

The distribution of node degrees within a graph can significantly influence the performance of shortest path algorithms. Let $d(u, v)$ denote the shortest-path distance between two vertices $u, v \in V$. The diameter of the graph is then defined as

$$\max_{u,v \in V} d(u, v),$$

i.e., the maximum shortest-path distance between any pair of vertices. These structural properties have a direct impact on the performance of SSSP algorithms. In particular, the distribution of node degrees and the magnitude of the diameter can influence both computational work and parallel efficiency [13].

2.3 Rank error and Delay

Rank error and *delay* are two key metrics for quantifying the degree of relaxation in a relaxed priority scheduler. They can both be analyzed theoretically, and measured at runtime. Papers tend to use rank error and delay as metrics when benchmarking their relaxed data structures [5]. Average rank error and delay alone can give a good measurement as to how relaxed a given data structure or algorithm is, although it does not always directly correlate with additional work.

Rank error measures how many elements with a higher priority exist in the queue at the time of popping a given element. Consider an element e with priority p. On pop of e, the rank error is defined as the number of elements with a priority higher than e. Consider a relaxed queue with n elements sorted by priority. If there are k elements in queue with a higher priority than e, where $0 \leq k < n$, popping element e yields a rank error of k.

Delay is the measurement of how many elements n of a lower priority than e, are popped before e, whilst e remains enqueued. Delay is only measured for enqueued elements, and is counted per element. Each time an element of priority p is popped from the queue, the delay d is incremented for all enqueued elements with a higher priority than p. Consider a relaxed queue with n elements that are sorted based

on priority, there is an element at index i, where $0 < i < n$. If the element at i is popped, the delay for all elements at indices $0 \leq x < i$ is incremented by 1.

The two metrics are closely related. If all elements inserted into the queue are eventually popped, the average rank error and the average delay are equal. This follows from the fact that a pop operation with rank error k causes the delay of k higher-priority elements to be incremented by 1. Summed over all operations, this establishes a one-to-one correspondence between total rank error and total accumulated delay.

There are many use cases for these metrics and a lot of information may be gathered by analysis. By looking at the average delay and rank error for algorithms a general idea of how relaxed an algorithm is may be formed.

Additional quantifications of these errors, which are not common in the literature, include:

- Whether or not the rank error and delay varies over run-time.
- Examining the rank error and delay at different percentiles, i.e. is there a large variance in generated rank error and delay.
- Examining the minimum and maximum rank error and delay.
- Examining the standard deviation of rank error and delay.
- The difference between the average and the median.

These metrics may provide insight into the behavior of different relaxed data structures, in relation to different graphs.

2.4 Work-efficiency

Efficiency in parallel SSSP algorithms can be measured in various ways. In [5], rank error and delay serve as quality metrics under the assumption that higher rank error implies more unnecessary operations. In this thesis we define the *work-efficiency*, E, as the ratio of necessary node extractions to total node extractions:

$$E = \frac{N_{opt}}{N_p}$$

Here, N_{opt} is the number of nodes processed by sequential Dijkstra, and N_p is the total number of node extractions performed by the algorithm. Values fall in the interval $(0, 1]$, where $E = 1$ corresponds to fully work-efficient behavior. In our implementation, a node is considered processed when it is extracted from the priority structure and passed to the relaxation routine. The counter `processedNodes` is incremented only after the node has been fully processed (see line 36 in Figure 2.2).

Sequential Dijkstra processes each node exactly once, so N_{opt} represents the lower bound on node extractions for any correct algorithm on a given graph instance.

In pre-existing literature work-efficiency is commonly referred to as $\frac{N_p}{N_{opt}}$, which constitutes the ratio of additional work done. In [8] they refer to this metric as *overhead*, which is the definition that will be used from now on.

Another way to measure work-efficiency would be to count and compare the *edge relaxations*. Edge relaxations do not have anything to do with the algorithmic relaxation mentioned in this thesis, confusingly. It refers to the number of nodes processed, as well as the count of discarded nodes due to a shorter path already being stored in `distances`. Conceptually this is the number of nodes the algorithm has interacted with.

The loss of work-efficiency is also not the only possible undesirable effect of relaxation. Depending on the approach, relaxation may result in a large queue size and thus potential memory constraints. Queue size in this context refers to the total number of nodes present across all queues at any time. Some data structures may store the same node multiple times simultaneously, bloating the queue size. Similarly, the use of local queues in other approaches can result in similarly large queue sizes. These cases represent qualitatively different failure modes: one wastes work, the other wastes memory, and some approaches may incur both.

2.5 Throughput and effective throughput

Throughput, T, is measured as the count of processed nodes per unit of time. Throughput is a common metric for the speed of algorithms. If an algorithm processes n nodes in t seconds, the throughput measured in seconds would be:

$$T = \frac{n}{t}$$

In the context of relaxed algorithms it may be a deceiving metric, as nodes may be processed more than once. Only good work should be taken into account. By introducing *effective throughput*, T_e, only the good work is taken into account, making for a more fair comparison between algorithms of differing work efficiencies. Our method of choice for calculating effective throughput is by $T_e = T \cdot E$. This is the same as calculating the effective throughput by taking the required amount work for a sequential Dijkstra N_{opt}, and dividing it with the time taken for the relaxed algorithm to solve the same graph t:

$$T_e = \frac{N_{opt}}{t}$$

The effective throughput is a good metric for speed, but speed is not the main concern for this thesis, work-efficiency is. The effective throughput still remains as a derivative of the work-efficiency and thus can be used to argue as an additional metric to argue the advantages of different relaxation approaches.

3

Related work

Extensive research has been conducted on SSSP-problems and relaxed data structures. However, the specific relationship between relaxation errors and work-efficiency has not been the focus of many prior studies. This chapter introduces relevant related work in the realm of relaxed data structures and algorithms used in SSSP contexts.

3.1 Julienne

The Julienne framework as introduced by Dhulipala et al. relaxes work allotment by utilizing bucketing [14]. Bucketing entails the grouping of work into sets, depending on their priority. Each bucket contains a certain range of priority values (i.e. one bucket may contain work within the range $1 \leq p < 5$, whilst the next corresponds to $5 \leq p < 7$, where p is the priority value), the ranges of the buckets change over time, depending on the range of priority values of all work currently bucketed. The benefits of bucketing is that the data structure is more lightweight compared to lists, with `push` and `pop` operations being performed in constant time, but this comes at the cost of loose adherence to strict ordering.

The framework can be used to implement Δ-stepping, an efficient SSSP algorithm [15]. Δ-stepping works in *steps*, each expanding the range of priorities allowed to be explored by a fixed parameter Δ. This incremental stepping ensures that all paths within the current step are rigorously explored before proceeding to the next, guaranteeing that upon completion of the final step all candidate solutions have been examined. Proper tuning of Δ is crucial: a value that is too small nullifies the benefits of bucketing, while one that is too large (coarse) leads to excessive relaxation and efficiency loss.

The performance of Δ-stepping is inherently graph dependent [14]. Fixed step sizes struggle on graphs with large variance in edge weights, such as road networks, because some buckets become overfull while others remain nearly empty [16]. There are alternatives to the Julienne implementation of Δ-stepping that do not suffer from this same vulnerability to this extent, such as Galois [17]. The Galois implementation, while performing better on certain graph types, is harder to compare to different approaches, as the speedup can be attributed to system optimizations. It is

contended whether or not Galois or Julienne is more efficient on network structures and RHGs, as different prior work have presented different conclusions [14, 16].

3.2 *k*-LSM

The lock-free k Log-Structured Merge-tree (k-LSM) PQ as defined by Wimmer et al. [6] is a relaxed concurrent data structure that utilizes bounded thread-local queues, as well as unbounded global queues. These queues are based on log-structured merge trees [18], which allows for operations in $O(\log n)$ time for a queue with n items.

k-LSM works by allowing threads to acquire any of the top $p + 1$ priority elements from the global queue whenever their local queue is empty [6]. As a thread processes elements, all newly discovered work is pushed onto its own queue. Once the local queue reaches above a certain length, every element of the queue will be merged into the global queue. If at any point a local queue ends up empty, the thread of which it belongs will acquire work from the global queue. The degrees of relaxation are determined by p, as well as the maximum length of the local queue before merging it onto the global queue, k. The global queue is a resource with minimal contention, but the k-LSM scheduler achieves lock-freeness despite the shared resource (the global queue) [6]. There remains some resources which could be contended, but due to the relaxation on accessing the shared resource said contention is limited.

The relaxation of k-LSM differs from something like Δ-stepping by having two separate queues accessible to each thread, with two different levels of relaxation [6, 15]. k-LSM relaxes the strict ordering constraints which may reduce the work-efficiency. The parameter p reduces the level of contention at the cost of a lower work-efficiency. The lower the p the closer the behavior of a strict PQ the global queue approaches. The parameter k, limits the allowed priority error before synchronizing with the global queue. Large values of k lead to an increase in throughput, but are more tolerant of bad work, however the work efficiency is allowed to drift further [5].

3.3 MultiQueue

The MultiQueue (MQ) as presented in [2, 5, 19] is a relaxed data structure that aims to reduce the problems present in a sequential PQ by easing the strict order requirements. The basic MQ is made out of several sub-queues, which virtually function as their own PQs. The number of sub-queues is configurable, and is determined by the formula $T \cdot c$, where T is the number of threads set to work the queue, and c being a constant larger than or equal to 2. When a process attempts to pop an element from the MQ, a number of sub-queues will be selected. These sub-queues are sampled for their highest priority elements, and the one with the highest priority element will be locked, the highest priority element popped, and then the queue unlocked. When a new element is to be pushed onto the MQ, a random sub-queue will be locked, the element inserted into this queue, and then the lock is released. When used as

a work-scheduler this inherently introduces relaxation to the order of which work is evaluated. The number of sub-queues sampled, d, as well as c are commonly set to 2. A higher sample count may result in higher work-efficiency, at the cost of higher contention. Since the MQ is made out of PQ-based sub-queues it still maintains approximate priority, but with minimized contention over the locks.

A major drawback of the basic MQ is cache-efficiency, for instance when a thread accesses a sub-queue it fetches the data from memory into its own cache, then when another thread wants to access that sub-queue it must fetch that same data into its own cache [5]. Thus optimizations to the MQ aim to reduce inefficiency and improve cache locality. One such optimization is called *stickiness* which introduces the parameter s. This parameter, alongside d, control for how many consecutive `push` and `pop` operations a thread will reuse the same set of $d \geq 2$ queues for $s \geq 1$ amount of operations before choosing a new set. By reusing the same set of sub-queues we reduce the amount of times data is fetched from memory, thus improving performance. In [5] multiple stickiness approaches are suggested, in this thesis we utilize the *simple* (a.k.a. *random*) and *swap* implementations, due to these two variants having distinct behavior from one another, whilst being viable in different situations.

Several improvements of the MQ have been proposed to address its limitations. These include, but are not limited to: the Stealing MultiQueue (SMQ) [20] (see section 3.4), which utilizes buffers, thread-local queues and work-stealing protocol to improve cache locality [20]; the Multi Bucket Queue (MBQ), which employs thread-local buckets instead of PQs to reduce thread idling and scheduler overheads while increasing correctness [21]; lastly [5] proposes three distinct MQ optimizations: MultiQueue-Balanced (MQB), MultiQueue-Quality (MQQ) and MultiQueue-Fast (MQF). They combine buffers and stickiness to mitigate the weaknesses of the basic MQ. See Table 4.2 for the exact configurations used in both [5] and this thesis.

3.4 SMQ

The SMQ [20] expands upon the basic MQ, keeping the core principles while introducing thread-local queues to remove the need for locks and implementing a stealing protocol to distribute tasks. The stealing activates either when a local queue is empty or with a configurable probability p_{steal} in each step. When a steal occurs the local thread compares its top element with that of a randomly chosen queue and returns the one with highest priority. To further optimize the stealing a batch of β tasks are stolen each time. Stolen tasks are stored in a separate thread-local buffer, such that the next time the thread deletes an element it will delete from the buffer instead of the queue. The thread will not try to steal anything new before the steal buffer is empty.

The benchmarks done in [5] suggests that the SMQ and MQB offer similar performance on RHGs, while the SMQ is slightly slower in road networks. The performance of k-LSM differs greatly depending on the parameter k, however both the

SMQ and MQB achieves a faster solve for every graph, independent of the chosen k. Comparing each MQ variant, MQQ offers the lowest rank error, comparable to that of the basic MQ. MQB, while being similar to SMQ in performance, achieves lower rank errors but not as low as MQQ. MQF seems to perform the best on RHGs and generally achieves the highest rank error of any MQ variant, although in some cases it is tied with MQB. While the MBQ is not benchmarked in [5], Zhang et al. [21] directly compare MBQ with Julienne and MQ-o (where "o" denotes an optimized variant, though its precise meaning is unclear). On the USA road network with 48 threads, MBQ outperforms Julienne by a factor of $\sim$9.4$\times$ and MQ-o by $\sim$2.2$\times$. On the LiveJournal graph [22], whose degree and size resemble RHG_{22} from [5], the speedup drops to $\sim$1.72$\times$ and $\sim$1.54$\times$, respectively.

3.5 CA-PQ

The Contention Avoiding Concurrent PriorityQueue (CA-PQ) [7] is a relaxed priority queue that avoids sequential bottlenecks and memory contention by pairing a global *fat skip list* with per-thread `push` and `pop` buffers. When contention is low, CA-PQ behaves like a strict queue: a `push` inserts directly into the global skip list (the thread-local `push` buffer starts with a capacity of zero), and a `pop` removes the globally smallest item by calling a single-item delete on the global structure. The queue estimates recent contention frequency via two per-thread counters, `push_contention` and `pop_contention`, which are incremented or decremented on each operation depending on whether contention was observed. If `pop` contention exceeds a threshold and a thread's `pop` and `push` buffers are empty, the global queue is switched into relaxed mode: a subsequent `pop` triggers a bulk deletion that extracts up to k smallest items from the head of the fat skip list and deposits them in the thread's local `pop` buffer, reducing global head accesses by roughly $k-1$ per k removals. When `push` contention is frequent for a thread, that thread's `push` buffer capacity is raised to a fixed maximum; subsequent `push` calls are then buffered locally instead of accessing the global queue, with the capacity later decreased when contention subsides. Any `pop` operation that finds the local `push` or `pop` buffer non-empty simply returns the smallest available item from those buffers, further limiting global interactions. Because the queue automatically tightens or relaxes its ordering guarantees in response to measured contention, it maintains priority accuracy under light load while preserving throughput under heavy load, without manual parameter tuning.

In [7], CA-PQ is compared on an SSSP benchmark against MQ ($c = 16$), k-LSM_1024 ($k = 2^{10}$), and k-LSM_65536 ($k = 2^{16}$). In these experiments, CA-PQ consistently outperforms the other queues in execution time and in the *$miss* metric (defined as $\frac{\#\text{ L2 cache misses}}{\#\text{ nodes in graph}}$), despite generating substantially more *waste* (redundant work). For example, on RoadNet [22] with unit weights, CA-PQ incurs roughly 200$\times$ more waste than MQ yet achieves a speedup of $\sim$2.6$\times$ due to far fewer cache misses. The sole exception is the LiveJournal [22] graph with uniform edge weights, where MQ is slightly faster by keeping waste extremely low while still maintaining competitive cache performance (see Table 1 in [7] for the full comparison).

Compared to the relaxed MQ variants in [5], CA-PQ shows weaker scaling at high thread counts. Throughput experiments indicate that CA-PQ benefits from additional threads, but the gains typically plateau beyond 32-64 threads, whereas relaxed MQ variants continue scaling more steadily across architectures. A similar trend appears in the SSSP benchmarks: CA-PQ remains competitive at moderate thread counts but benefits less from additional threads than MQF and MQB. This indicates that the adaptive buffering strategy of CA-PQ effectively reduces contention, although the centralized fat skip list still limits scalability under extreme parallelism.

3.6 Wasp

Wasp by D'Antonio et al. [23] is an algorithm designed specificaly to solve SSSP-problems as fast as possible, designed to reduce thread idling and synchronization overhead while limiting relaxation errors. It utilizes a distributed bucketing structure meaning Wasp utilizes thread-local bucketing as opposed to the global bucketing approach used in the definition of Δ-stepping in Julienne [14, 23]. There are multiple levels of buckets belonging to each thread, this is referred to as a bucket-queue, with the bounds of each bucket being defined by the elements in the queue. The threads primarily operate on their local bucket-queue, however Wasp also utilizes a priority-aware work-stealing protocol which distributes work across the per-thread bucketing structure. Threads are grouped based on their `NUMA_distance` from the stealing thread. A stealing thread t_{steal} will first look at the group of threads closest to it, which we will call tier 1, then if no higher priority work is found t_{steal} will move on to tier 2, then tier 3 and so on with each tier increasing in `NUMA_distance`. Once a target has been found, work is stolen from that thread and then processed by t_{steal}. If none of the targets have higher priority work the stealing is aborted and the thread continues with its own local bucket-queue. Additionally, to improve performance on high-degree graphs, neighborhood decomposition is used. In other common algorithms, all edges incident to a node are processed by a single thread. On graphs with highly skewed degree distributions, such as Friendster [24] or Mawi [25], this becomes problematic because shorter paths may not be discovered until the thread finishes processing all edges, potentially causing significant redundant work. To mitigate this, Wasp introduces a tunable parameter θ. When a node's degree exceeds θ, the scheduler splits its edges into chunks that can be processed by different threads, increasing concurrency.

The experiments in [23] compare Wasp against Galois and MQ (using input/output buffers, per-graph stickiness tuning and $c = 2$) on two different CPUs. On the USA road graph with 128 cores, Wasp runs $\sim$1.35$\times$ faster than MQ and $\sim$2.32$\times$ faster than Galois; on 64 cores the corresponding speedups are $\sim$1.60$\times$ and $\sim$2.29$\times$. In terms of scaling from a single thread, MQ attains the highest speedup on the USA road graph (44.3$\times$), followed by Wasp (25.2$\times$) and Galois (12.3$\times$). On the Friendster social graph, however, Wasp achieves the best scaling (73.9$\times$), with Galois (43.7$\times$) and MQ (36.6$\times$) trailing. The same ranking appears on the Mawi graph, where Wasp reaches 24.5$\times$ speedup, while MQ and Galois drop to 4.5$\times$ and 1.1$\times$,

respectively. One reason for this could be the neighborhood decomposition that is not present in either MQ or Galois.

3.7 AdaMW

The AdaMW[1] scheduler by D'Antonio et al. [11] is a scheduler designed for SSSP. It is more nuanced as opposed to the other algorithms/schedulers as its approach is graph-dependent, it can dynamically switch strategy between Wasp and MQ In the pre-processing step it extracts key-values to decide on an algorithm to use, for any given graph. If the degrees or if the average weight of the edges is disproportionally large in comparison to the median, it will use Wasp, otherwise it will use MQ.

AdaMW combines the strengths of both Wasp and MQ. The two schedulers are more efficient than one another based on the graph. By properly tuning the parameters of AdaMW one can utilize the strengths of both while mitigating their weaknesses. Wasp being better at graphs with uniform weight distribution while MQ is best for large-diameter graphs with skewed edges [11].

3.8 Analytical Bounds on Relaxation Overhead

Alistarh et al. [8] provide the first analytical connection between rank error and overhead. They consider a relaxed scheduler that guarantees a rank error of at most k and must schedule the globally highest-priority element within k pops; this property is also referred to as *lateness* [3].

For incremental algorithms with random task order and local dependencies (e.g., Delaunay triangulation, insertion sort), the expected extra pop operations beyond N_{opt} is

$$O(\mathrm{poly}(k) \log N_{opt})$$

so the overhead

$$\frac{N_p}{N_{opt}} = 1 + O(1)$$

remains bounded whenever k is moderate.

For SSSP with a label-correcting parallel Dijkstra (as in Figure 2.2), they prove that the total number of pop operations satisfies

$$N_p \leq N_{opt} + O\left(k^2 \, \frac{d_{\max}}{w_{\min}}\right),$$

giving an overhead bound of

$$\frac{N_p}{N_{opt}} \leq 1 + O\left(\frac{k^2 \, d_{\max}}{w_{\min} \, N_{opt}}\right).$$

[1] **Ada**ptive **M**ultiQueue-**W**asp (AdaMW)

Consequently, on graphs with low diameter and bounded edge-weight variance, the overhead becomes

$$1 + O\left(\frac{k^2 \log n}{n}\right).$$

A lower bound of $\Omega(\log N_{opt})$ extra work is also shown, even for the practical MQ scheduler.

Experiments with a parallel SSSP implementation using MQ confirm that, despite increasing rank error with thread count, the overhead is only approximately 1.01 (about 1% extra work) on low-diameter graphs and reaches approximately 1.05 on the high-diameter USA road network graph, consistent with the theoretical bounds.

These results formalize the intuition that the cost of relaxation in terms of lost work-efficiency is negligible on many graph classes, but can become noticeable on high-diameter networks.

4

Methodology

This chapter describes the experimental methodology developed to evaluate how relaxation errors affect the work-efficiency of parallel SSSP algorithms.

4.1 Benchmarking Framework

A robust benchmarking framework is required to systematically evaluate the performance of SSSP algorithms. For the purposes of this thesis, we extend and adapt an existing framework rather than developing one from scratch.

Several benchmarking frameworks have been proposed in prior work [26–31]. Among these, the open-source framework developed by Williams and Sanders [5] was found to best match our requirements. The implementation is publicly available[1], and serves as the foundation for the modifications introduced in this work. The framework provides a flexible environment for evaluating different PQ implementations in the context of SSSP using Dijkstra's label-correcting algorithm. In particular, it supports interchangeable data structures, enabling controlled comparisons between strict and relaxed priority queues. For each execution of SSSP, the framework reports standard performance metrics such as execution time, the number of processed nodes, and the number of ignored nodes.

To support the analysis conducted in this thesis, the framework was extended to collect additional fine-grained measurements. Per-thread timestamps are collected for `push` and `pop` operations, which are then sorted and saved as a log. When processing the log we extract the global PQ-size at the time of execution of each `pop` operation, as well as the rank error and delay. Additionally we can calculate work-efficiency and whether or not the operation is considered extra work or is necessary for the solution.

It is not immediately obvious how to determine whether a given operation constitutes necessary work. To address this, we define and measure two complementary metrics:

- Redundancy rate (extra work rate): the fraction of popped elements that

[1] https://github.com/marvinwilliams/multiqueue_experiments

do not correspond to the shortest path to their associated node and are not ignored. Each pop operation is represented as a binary value, where 1 denotes extra work and 0 denotes necessary work. The redundancy rate, R, relates to the work-efficiency as $R = 1 - E$.

- Extra work generation: whether a pop operation causes extra work to be performed in the future. This is measured by mapping pop operations to the elements subsequently inserted into the priority scheduler. If any of these inserted elements are later identified as extra work, that extra work is attributed to the originating pop operation. This distinction makes it possible to separate the execution of extra work from the operations that generated it. Only extra work generated at depth one is considered, i.e., extra work directly resulting from a pop operation and its associated pushes. The amount of extra work generated by a pop operation therefore ranges from 0 to the out-degree of the associated node.

Together, these metrics make it possible to isolate and quantify the effects of relaxation in a controlled setting. Using the recorded logs, we can investigate how these metrics influence execution behavior and whether consistent relationships between them emerge across different graph classes. The extended framework is publicly available to ensure reproducibility of the results[2].

4.2 Graph Selection

The characteristics of the input graph significantly influence the performance of SSSP-solvers. As discussed in [11], the Wasp algorithm is preferable on graphs with a uniform weight distribution, whereas the MQ performs better on graphs characterized by a skewed weight distribution.

In this thesis, three types of graphs are considered: road networks, Random Hyperbolic Graphs (RHGs), and grid graphs. These represent common application domains for SSSP algorithms. Road networks model transportation systems and are typically characterized by relatively homogeneous degree distributions and large diameters [13]. In contrast, RHGs exhibit heterogeneous degree distributions, smaller diameters, and strong clustering effects. Such graphs more closely resemble social networks, where nodes form densely connected clusters with comparatively sparse interconnections. Grid graphs with randomly assigned edge weights are included as a controlled synthetic baseline. Their regular structure enables repeated sampling and averaging over multiple instances, thus reducing the influence of randomness and providing a clearer view of the underlying algorithmic behavior.

Road network graphs with skewed weight distributions are sourced from the 9th DIMACS Implementation Challenge[3]. These real-world graphs represent travel times

[2]https://github.com/affe4ever/Understanding_Work-Efficiency_of_label-correcting_SSSP_Algorithms.git

[3]https://www.diag.uniroma1.it/challenge9/download.shtml

between locations, naturally producing skewed edge weights. For graphs with uniform weight distributions, RHG are generated using KaGEN [32] and subsequently converted to the required input format using a custom conversion script. Random grid graphs are also generated with KaGEN and converted; edge weights are then assigned uniformly at random from $[1, 10\,000]$, a range comparable to that of the NY road network graph. This combination of real-world and synthetic graphs ensures a comprehensive evaluation across diverse graph characteristics. As noted in Table 4.1, the target average degree in the RHG generation is set to either 8 or 16, but the exact value varies slightly across different random seeds; the table therefore reports the observed range.

Type	Size	Nodes	Edges	Avg. Degree	Seeds
Road Graph	USA	23 947 347	58 333 344	2.44	1
	CTR	14 081 816	34 292 496	2.44	1
	NY	264 346	733 846	2.78	1
RHG	N24_d16	16 777 216	∼239 000 000	14.0–15.0	5
	N24_d8	16 777 216	123 547 730	7.36	1
	N22_d16	4 194 304	∼60 000 000	14.0–15.0	5
	N20_d16	1 048 576	∼15 000 000	14.0–15.0	5
	N18_d16	262 144	∼3 800 000	14.0–15.0	5
Grid Graph	X13_Y13	67 108 864	268 402 688	4.0	3
	X10_Y10	1 048 576	4 190 208	4.0	5

Table 4.1: Table of all the graphs included in the experiments, where the size corresponds to either an area (Road Graphs) or the parameters used in the KaGen generation (RHG and Grid).

4.3 DrPQ

To isolate the effects of relaxation on SSSP performance, a configurable relaxed PQ is developed. We refer to this scheduler as a Dijkstra relaxed Priority Queue (DrPQ), as it serves as a controllable scheduling abstraction for Dijkstra-style SSSP solvers. Unlike a traditional priority queue, which always returns the minimum element, the DrPQ allows configurable rank errors by relaxing the ordering constraint of `pop` operations. This provides a controlled experimental framework for quantifying how varying degrees of relaxation affect work-efficiency in SSSP computations.

Unlike a traditional PQ, which always returns the minimum element, the relaxed PQ permits bounded deviations from strict priority order through configurable rank errors in `pop` operations. This enables systematic evaluation of the trade-off between relaxation and work-efficiency in SSSP computations.

The relaxed PQ is built on an ordered tree data structure that maintains the correct priority ordering of elements after `push` and `pop` operations. When popping, rather than always selecting the first element, the queue selects an element at an index determined by a configurable probability distribution. As of now it is based on a normal distribution and functions by taking the index i, which is drawn from a normal distribution $N(\mu, \sigma^2)$ and constrained to the range $0 \leq i \leq |Q| - 1$, where $|Q|$ denotes the current queue length.

The DrPQ scheduler exposes three configurable parameters that control its relaxation behavior:

- Mean (μ) – The center (expected value) of the normal distribution used for rank selection.
- Standard deviation (σ^2) – The spread of the normal distribution; larger values increase the likelihood of selecting elements further from the mean.
- Percentile – The probability of selecting an element at the first position (index 0)

There are additional ways of experimenting with the effects of rank errors. As of now the DrPQ features only normal distributions for selecting rank errors, the framework could be extended to support other distributions for more comprehensive analysis of relaxation effects. An example of such a distribution would be one which has a probability p chance of selecting a random element in the queue, and a $1 - p$ chance to select the element at index 0. Depending on how small p is the less additional work must be done, reasonably. But exactly how differing values of p affect the work-efficiency remains unexplored.

To ensure the algorithm terminates, index 0 must be selectable with sufficient frequency; otherwise, the algorithm may only extract the minimum element once the

queue is otherwise empty, and may result in an algorithm needing to explore every single possible path before finally being allowed to terminate. The parameters are interdependent: given any two, the third can be derived. For instance, setting the mean and percentile determines the required variance, while setting the variance and percentile determines the mean.

4.4 Algorithm Parameters

Table 4.2 lists all configurations and tunings of the algorithms used in our experiments. Where possible, our parameter choices follow [5] to ensure comparability with prior work. The only exceptions are the experimental configurations in which we systematically vary the degree of relaxation: larger-c variants of MQ explore the trade-off between contention and work efficiency, while the DrPQ variants isolate the effect of relaxation under controlled conditions.

Algorithm	Name	Parameters	t (threads)
MQ	mq_basic	c=2, no stickiness	1, 8, 16, 32
	mq_quality	c=2, s=4, stick_random	8, 16, 32
	mq_balanced	c=2, s=256, stick_swap	8, 16, 32
	mq_fast	c=2, s=4096, stick_random	8, 16, 32
	mq_sr256_c4	c=4, s=256, stick_random	32
	mq_sr256_c6	c=8, s=256, stick_random	32
	mq_sr256_c16	c=16, s=256, stick_random	32
SMQ	smq	steal prob.=2^{-3}, batch size=64	8, 16, 32
k-LSM	klsm256	k=256	8, 16, 32
	klsm1024	k=1024	8, 16, 32
	klsm4096	k=4096	8, 16, 32
CA-PQ	capq	(none)	8, 16, 32
DrPQ	drpq_mean0	mean=0, stddev=0	1
	drpq_mean1	mean=1, sdtdev=0	1
	drpq_mean1_p001	mean=1, percentile=0.01	1
	drpq_mean10_p001	mean=10, percentile=0.01	1
	drpq_mean100_p001	mean=100, percentile=0.01	1
	drpq_mean1000_p001	mean=1000, percentile=0.01	1
	drpq_mean10000_p001	mean=10000, percentile=0.01	1

Table 4.2: Table with all algorithm configurations included in the experiments. The names closely resemble the names in the produced log files and thus contain parameters in the naming scheme.

4.5 Environment

All benchmarks are executed on a dedicated server at Chalmers University of Technology, referred to as Ithaca. Using a dedicated server minimizes interference from background processes, user sessions, and other system activities that could affect performance measurements. An additional benefit is experimental consistency: all algorithms and configurations are evaluated under identical hardware conditions, eliminating variability due to different clock speeds or system architectures.

The hardware specifications for Ithaca are as follows:

- **Processors:** 2× Intel Xeon E5-2695
- **Logical threads:** 72
- **Memory:** 64 GB RAM

Since all experiments are conducted on a single hardware platform, the effects of different processor architectures or memory configurations are not explored in this thesis. To further limit the scope we will only run the experiments on one CPU, using at most 32 threads, reducing potential communication costs.

4.6 Visualization of Work-Efficiency and Relaxation

We evaluate work-efficiency and relaxation behavior across algorithms and graph instances using two complementary types of visualizations.

Summary plots compare complete runs of different algorithm configurations across graphs. Each point represents a single run summarized using aggregate statistics such as medians or means. The main comparison relates median rank error to median delay, with points shaded according to the redundancy rate. This highlights differences between rank error and delay at a high level. Additional plots examine relationships between other aggregated metrics.

Bucketed run-time plots show how metrics change during execution of a single run. Each run is divided into 200 bins spanning the full execution, resulting in a variable number of operations per bin. For each bin, summary statistics are computed to reduce noise while preserving the overall structure of the run. This representation captures how rank error, delay, queue size, and redundancy change over the course of execution, and can show relationships between metrics that are not visible in the summary plots.

Together, these two visualization types provide complementary views of the same data. Summary comparison plots summarize behavior across configurations, while

bucketed run-time plots show how behavior develops during execution.

5

Evaluation

This chapter presents the experimental results based on the visualization framework introduced in section 4.6. Given the large number of possible figures that can be generated, the results are restricted to a selected set of representative plots that highlight the most relevant patterns in work-efficiency and relaxation behavior.

The selected plots focus on both aggregate summaries and bucketed run-time behavior, as defined in the previous section, allowing comparison of algorithm configurations across graph instances as well as within individual runs.

Previous research [5, 15, 23] commonly used rank error as a primary metric for estimating work-efficiency. This chapter therefore investigates how accurately rank error reflects the actual amount of extra work performed. In addition to evaluating the accuracy of rank error as the primary metric of work-efficiency, we are also interested in examining the skew between median rank error and median delay, and whether differences between these metrics meaningfully influence work-efficiency.

5.1 Comparison of Summary Plots

In Figure 5.1, the relationship between median rank error and redundancy rate is shown across all available algorithm tunings. The plots indicate a clear correlation between rank error and redundancy rate, but it is not a perfect fit, indicating there are other factors impacting the work-efficiency. Some algorithms with relatively high rank error still achieve lower redundancy rates than algorithms with lower rank errors. This is particularly visible when examining DrPQ, where configurations with similar rank error behavior still differ noticeably in work-efficiency depending on graph structure.

Figure 5.2 shows the distribution of data points, shaded by graph type and size/area, but with single threaded algorithms excluded for a more representative visualization of relaxation. Graph types are separated into different layers, with larger, higher degree graphs in general having the highest median rank-error in comparison to their relatively low redundancy rate. The RHGs, with the exception of $rhg24\_deg8$ which behaves differently due to its different tuning, have consistent better performance than the road- and grid graphs. This suggests that graph structure has a significant

effect on how relaxation translates into redundant work.

Figure 5.3 compares the mean and median rank error across all algorithms and graphs. A large discrepancy between the two metrics indicates a rank error distribution heavily influenced by outliers. Most algorithms follow an approximately linear distribution, with the primary outliers belonging to CA-PQ, which exhibits significantly larger mean rank error than median rank error. This behavior cannot fully be understood from just this chart, but it likely has to do with how CA-PQ does its batching, which optimizes throughput over low rank errors.

In Figure 5.4 we examine the skew between the median rank error and delay. There is an approximate linear relation apparent throughout. What can be observed is that along the linear relation, the relative redundancy rate remains low, for the most part, whilst algorithms with a noticeable skew tend to have a higher redundancy rate. CA-PQ and SMQ are outliers in the sense that they exhibit large skews whilst still maintaining relatively low redundancy rates. Similar anomalies exist close to the linear relation as well, low-skew algorithms with high redundancy rates. These are primarily found in road- and grid graphs, particularly at extreme rank error values.

For the remainder of the evaluation, median values are generally preferred over means when comparing rank error and delay. The median more closely reflects the typical behavior of the algorithms, while means are strongly affected by outliers. Additionally, plotting mean rank error against mean delay would produce a perfectly linear relationship, since the two metrics share the same mean by definition. Using medians instead makes skew and divergence between the metrics more visible. However, for some algorithms such as DrPQ, the mean rank error may highlight deviations from the configured relaxation.

The following recurring observations apply throughout the overview results: rank error correlates with redundancy rate but does not fully determine it, graph structure strongly affects redundancy behavior, and median-based comparisons expose skew that is hidden when only means are considered.

Due to the differing behavior between graph types, the remaining evaluation is separated by graph category.

Figure 5.1: Summary plots for median rank error and mean redundancy rate. Shape and shading determined by algorithm and tuning. All algorithms and tunings included.

Figure 5.2: Summary plots for median rank error and mean redundancy rate for all graphs. Shape is determined by algorithm. Shaded by graph type and graph size/area. Single threaded algorithms excluded.

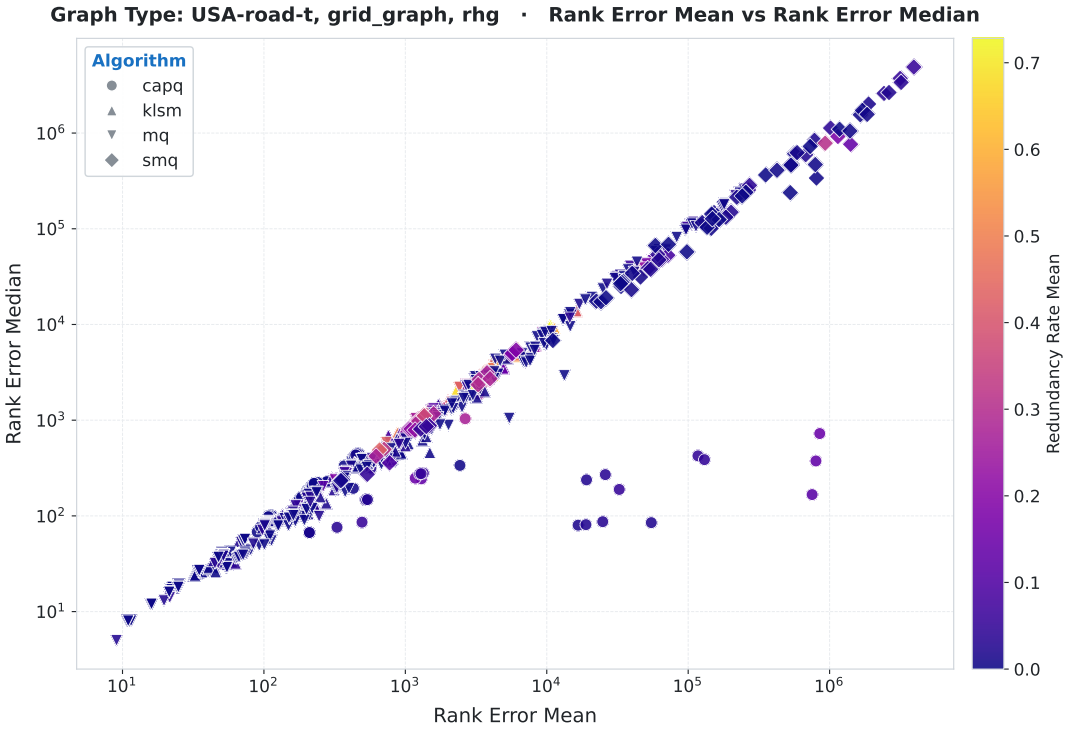

Figure 5.3: Summary plots for mean rank error and to median rank error for all graphs. Shape is determined by algorithm. Shading determined by redundancy rate. Single threaded algorithms excluded.

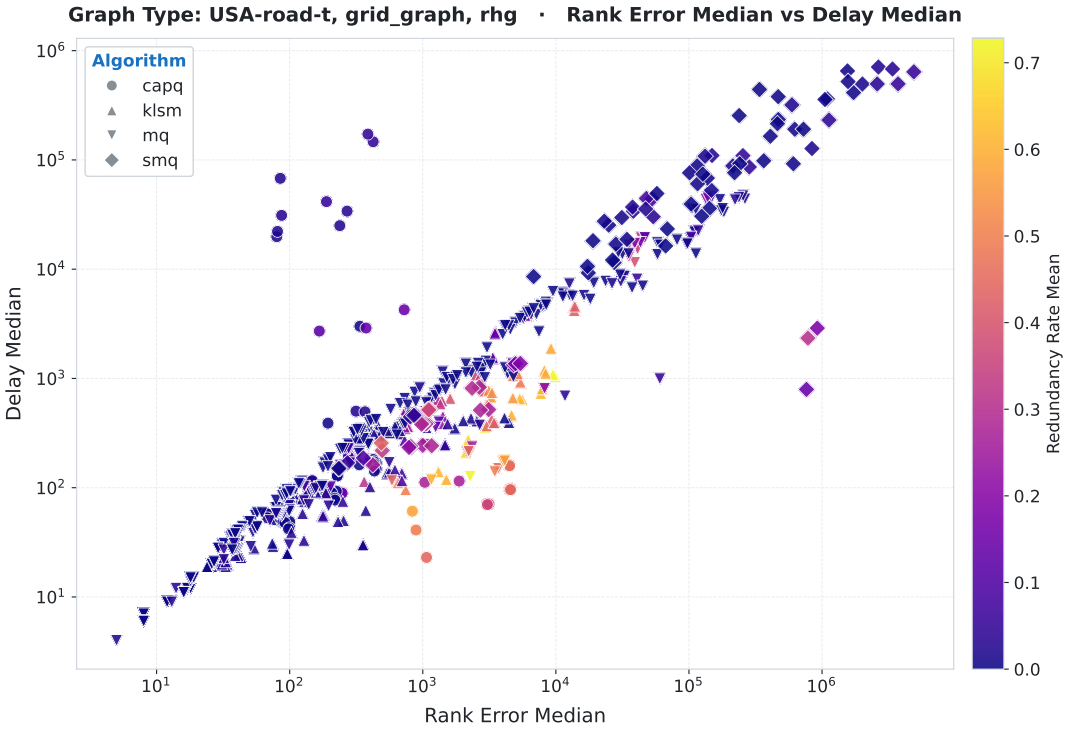

Figure 5.4: Summary plots for median rank error and median delay for all graphs. Shape is determined by algorithm. Shading determined by redundancy rate. Single threaded algorithms excluded.

5.1.1 Observations on Random Hyperbolic Graphs

Figure 5.5 shows the relationship between mean rank error and redundancy rate across all RHGs. To clarify the general trends, Figure 5.6 repeats the comparison after removing several configurations: the experimental as well as single threaded MQ tunings, DrPQ mean 0 and 1, and the $rhg24\_deg8$ graph are omitted. mq_basic_t1, along with $drpq\_mean0$ have a constant 0 rank error and delay, with perfect work efficiency, thus does not provide any valuable insight as to how relaxation affects work-efficiency. $drpq\_mean1$ is excluded due to its inconsistent behavior, as well as it not conforming to the principle about relaxed algorithms having at least a small chance to select the element of highest 0, as established in section 4.3. $rhg24\_deg8$ has a different degree than the other RHGs, thus making it less comparable.

Figure 5.7 depicts the relationship between redundancy rate and mean rank error, and the skew between median delay and rank error, for the $rhg24$ and $rhg24\_d8$ graphs. Rank error remains similar between the two graph types, while redundancy rate differs significantly. The higher-degree graph consistently exhibits lower redundancy rate and therefore higher work-efficiency across all tunings. Since the only difference between these graph types is average degree, this supports the hypothesis that higher degree reduces redundancy while maintaining similar rank error levels. Graph structure therefore strongly affects how relaxation translates into redundant work. Observing the comparison of median rank error and delay, there is a greater skew for the lower degree graph. This may have something to do with said graph being less interconnected than its higher degree peer. This is further explored in subsection 5.1.2.

Returning to Figure 5.6, it is evident that work-efficiency on larger RHGs is less affected by increasing rank error than on smaller graphs. There are data points that do not adhere to this trend, but these will be discussed later in this section.

Looking only at the DrPQ variants in Figure 5.6, a clear relationship between rank error and redundancy rate can be observed. The configurations form distinct vertical clusters corresponding to their configured rank errors (e.g., all mean-100 configurations of DrPQ align, as do all mean-1000 configurations etc.), while the ordering within each cluster is primarily determined by graph size. Since several seeds are used for each graph, some variation in the distribution of data points is expected. Higher configured mean rank errors also cause the data points for a given graph size to cluster more closely together. This effect is particularly noticeable when comparing the DrPQ configurations around rank errors of 10^1 and 10^4. Overall, these observations support the notion that higher rank error tends to reduce work-efficiency, although the preceding examples demonstrate that the relationship is not determined by rank error alone.

The correlation between median rank error and delay for all algorithms on the RHGs is shown in Figure 5.8. Most tunings follow an approximately linear pattern. CA-PQ and some instances of SMQ are the main outliers. All CA-PQ configurations instead occupy a relatively narrow rank error range while exhibiting widely differing

mean delays.

Looking at Figure 5.6 again, the k-LSM tunings are separated into vertical clusters (similarly to how DrPQ behaves) with approximately equal rank error but differing redundancy rates. The larger the k-value the larger the spread, and less consistent the clustering is. This trend is also observable across all graph types, as seen in Figure 5.1. Figure 5.9 shows all k-LSM tunings across different thread counts on all RHGs. Within each column, the upper half is dominated by higher-thread runs, while the lower half mainly consists of lower thread counts. However, there remains a substantial gap between the best- and worst-performing configurations even within the same tuning and thread count. This highlights the limitations of rank error as a direct measure of work-efficiency. Figure 5.10 also shows all k-LSM tunings but highlights the difference between median delay and rank error instead. Here the k-LSM configurations exhibit a noticeable skew between rank error and delay. The largest skews are associated with higher thread counts and lower redundancy rates. Larger k-values lead to higher rank errors, but the increase in extra work is not proportional to either the increase in k-values or the associated rank error. These observations suggest that limiting large rank error outliers is associated with higher work-efficiency.

Figure 5.11 shows the relationship between median rank error and redundancy rate for all MQ tunings. Some tunings, particularly MQ-basic and MQQ, form distinct vertical clusters. In contrast to k-LSM, where ordering within clusters is primarily driven by thread count, the MQ clusters are largely separated by thread configuration, with each cluster corresponding to a single thread count. For MQ-basic and MQQ, rank error can largely be inferred from the tuning and thread count. However, work-efficiency is less directly determined by rank error, as configurations with similar rank error may still exhibit different redundancy rates. The data points for other MQ tunings are scattered, as opposed to being ordered into columns.

Figure 5.12 shows the same relationship with data points shaded by graph size. Taken together, the figures suggest that for MQ-basic and MQQ, graph size does not explain the variation in work-efficiency among configurations with similar rank errors. For the remaining tunings, graph size appears more strongly associated with both rank error and redundancy rate.

The experimental configurations have higher redundancy rates than the other tunings. MQF, which has a stickiness factor of 4096, still achieves overall lower rank error and redundancy rate than even the best performing of the experimental tunings despite all of them having stickiness factors of 256. The impact of an increased c-factor on the median rank error as well as the delay is substantial.

The skew between median rank error and delay for the MQ tunings is shown in Figure 5.13. Most tunings exhibit a slight skew, with the notable exception of MQB which exhibits significantly less. Even so, MQB still exhibits a slight skew on the smallest RHGs. Small increases in redundancy are difficult to observe in this chart,

but an interesting observation is that the algorithms with the highest redundancy rate do not exhibit the largest skew. For algorithms with large c-factors, larger skew instead appears associated with better performance. Another notable observation is that MQF exhibits greater skew at lower thread counts. In contrast, MQ-basic and MQQ show highly consistent performance, with tightly clustered data points.

Overall, work-efficiency across RHGs cannot be explained by rank error or the skew between rank error and delay alone. Instead, it depends on multiple interacting factors that are not fully captured by relaxation metrics. Algorithm tuning plays an important role, but it is not the sole determinant of performance. Even graphs generated with similar parameters can exhibit substantially different work-efficiency under the same algorithm configuration, showing that small structural differences can strongly affect how relaxation translates into redundant work.

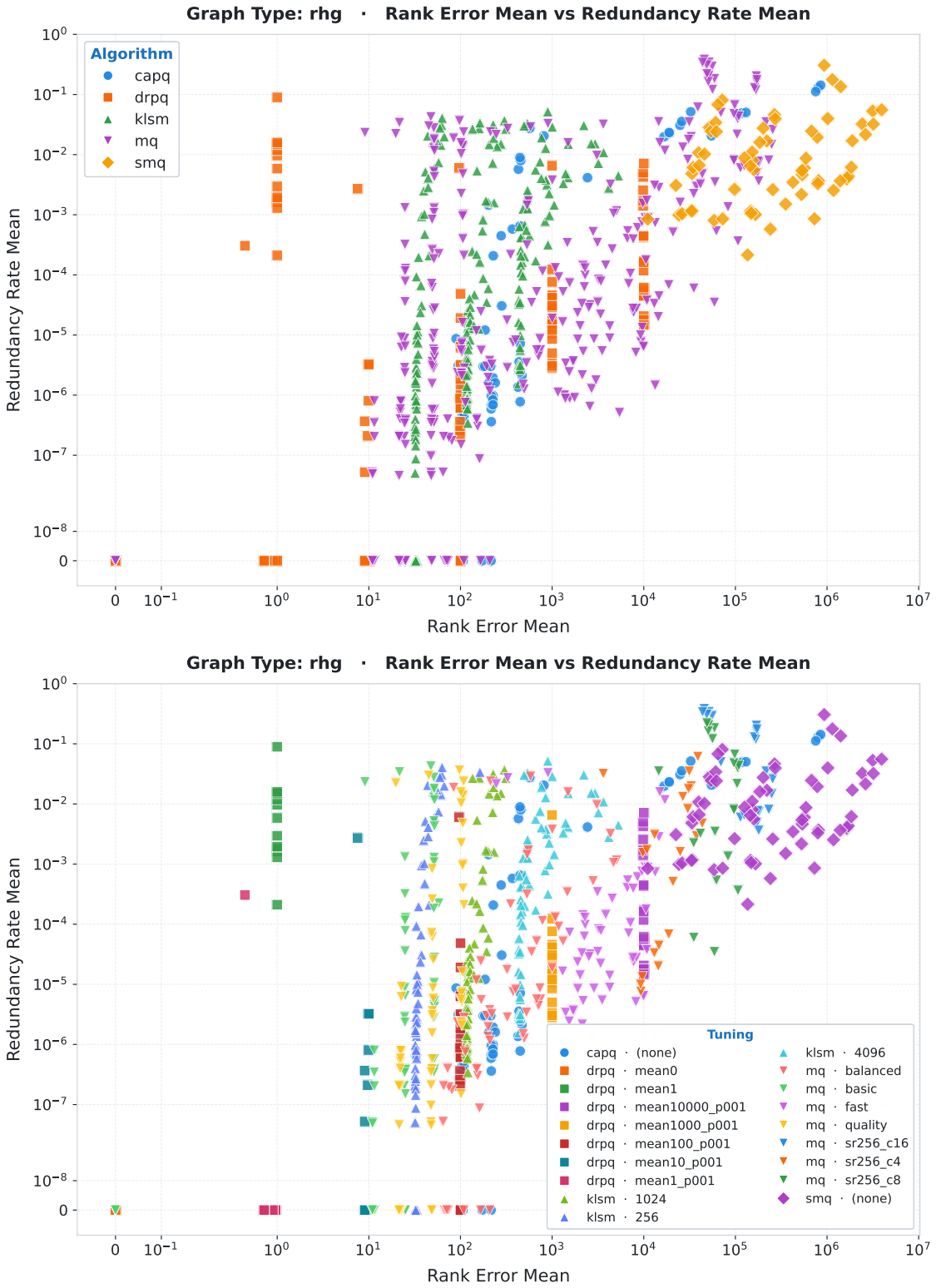

Figure 5.5: Summary plots for mean rank error and redundancy rate on RHGs. Shape is determined by algorithm. Shading determined by algorithm and tuning.

Figure 5.6: Summary plots for mean rank error and redundancy rate on RHGs, $d8$ excluded. Shape is determined by algorithm. Shading determined by algorithm and tuning. *drpq_mean*0 and *drpq_mean*1, experimental *mq* tunings and *mq_basic_t*1 excluded.

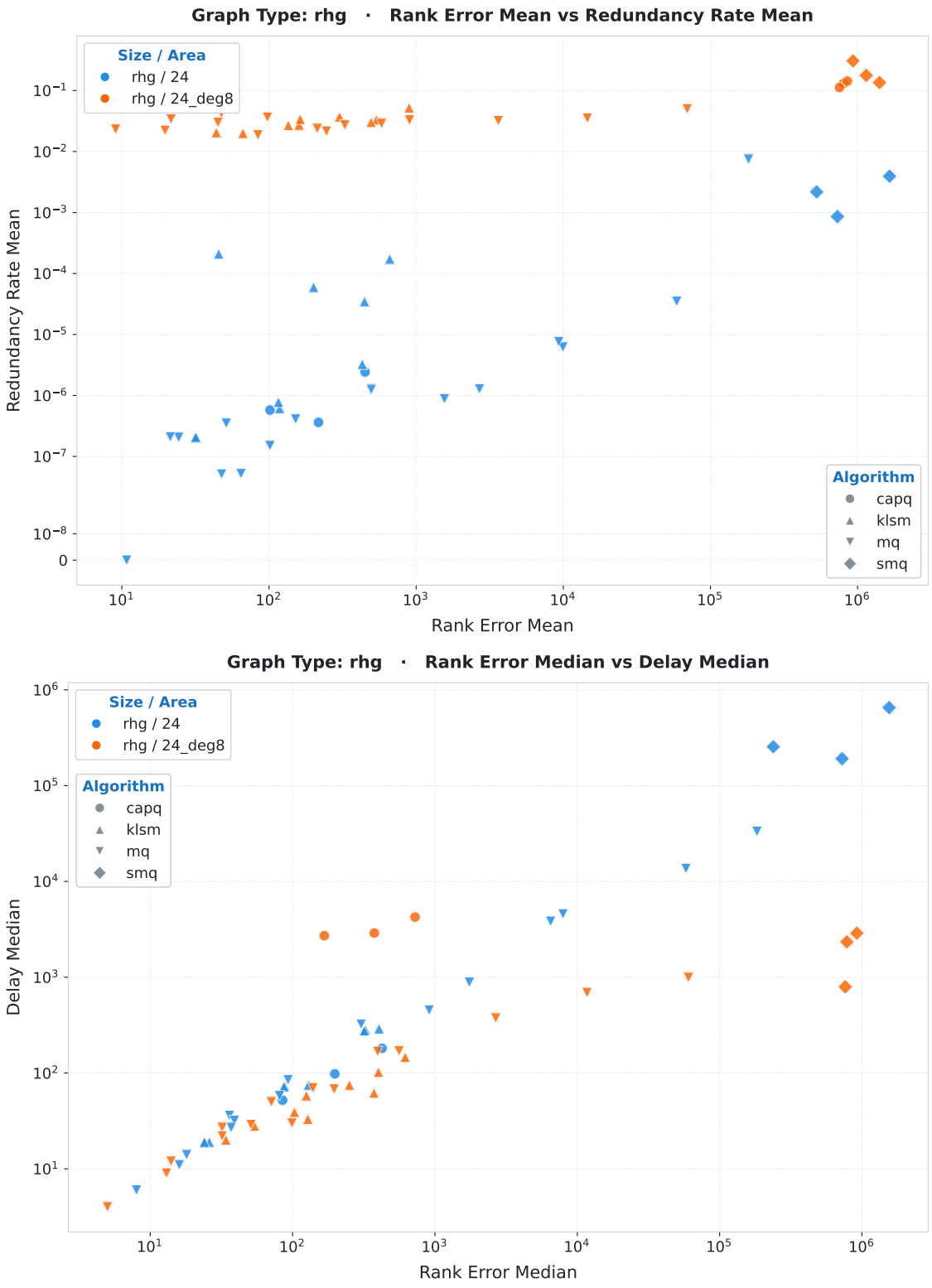

Figure 5.7: Summary plots for mean rank error and extra work, as well as median rank error to delay. Comparing *rhg*24_*d*8 and *rhg*24. Only 1 seed. Shape is determined by algorithm. Shading determined by graph size. Single threaded tunings excluded.

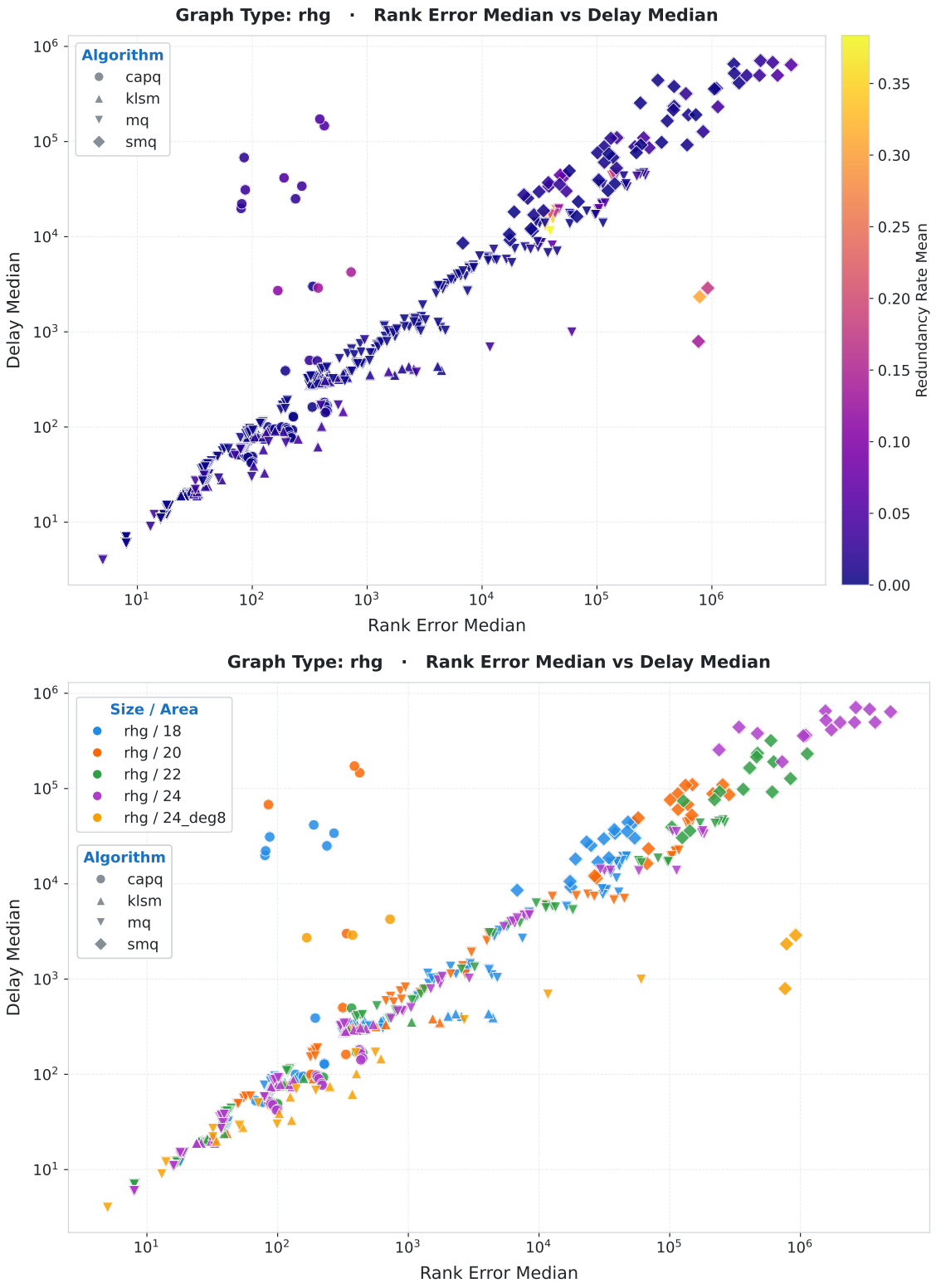

Figure 5.8: Summary plot for median rank error and median delay on all RHGs. Shape is determined by algorithm. Shading determined by redundancy ratio and size/area. Single threaded tunings excluded.

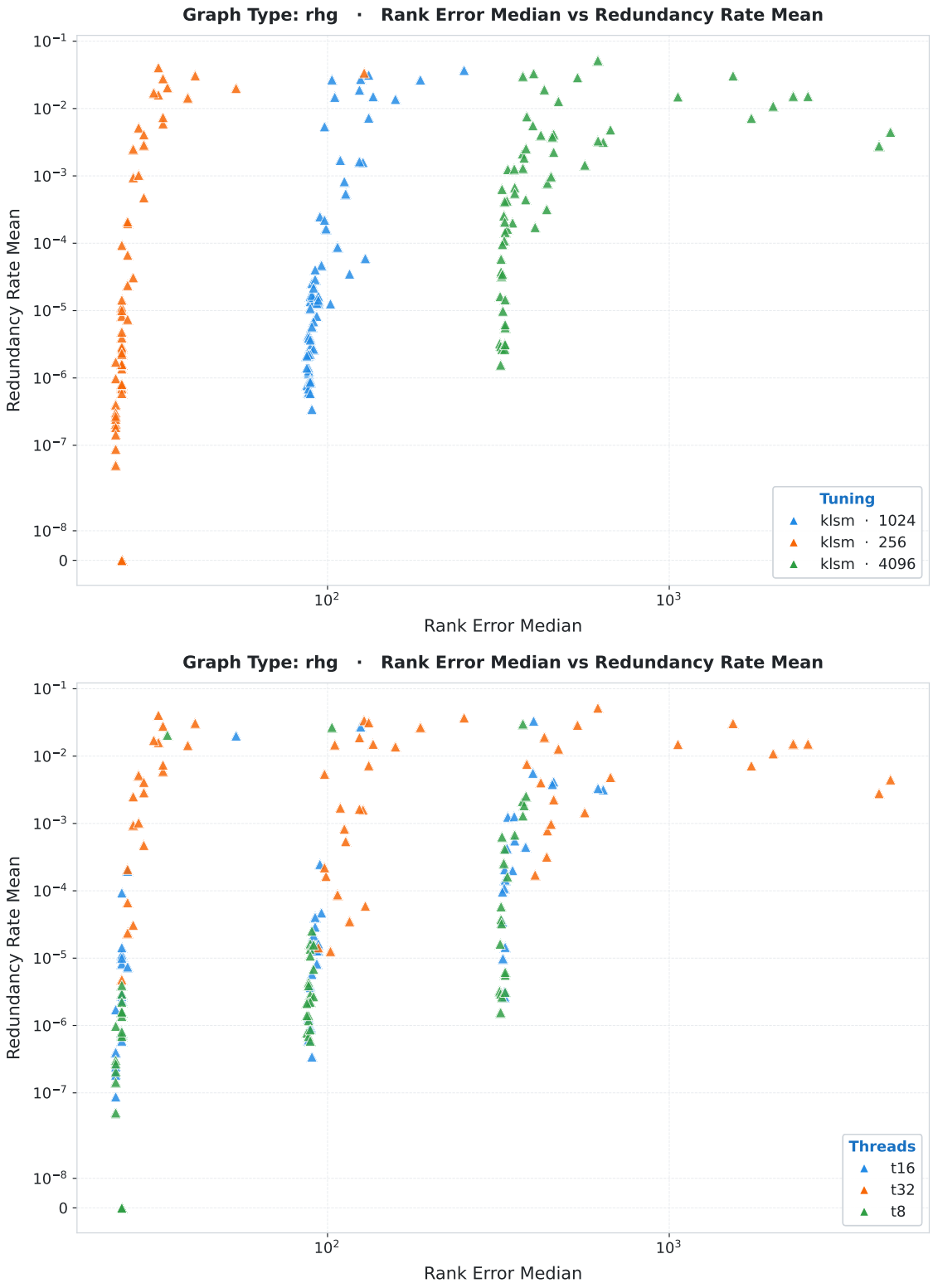

Figure 5.9: Summary plots for median rank error and redundancy rate on all RHGs. Only k-LSM. Shape is determined by algorithm. Shaded by tuning and thread count.

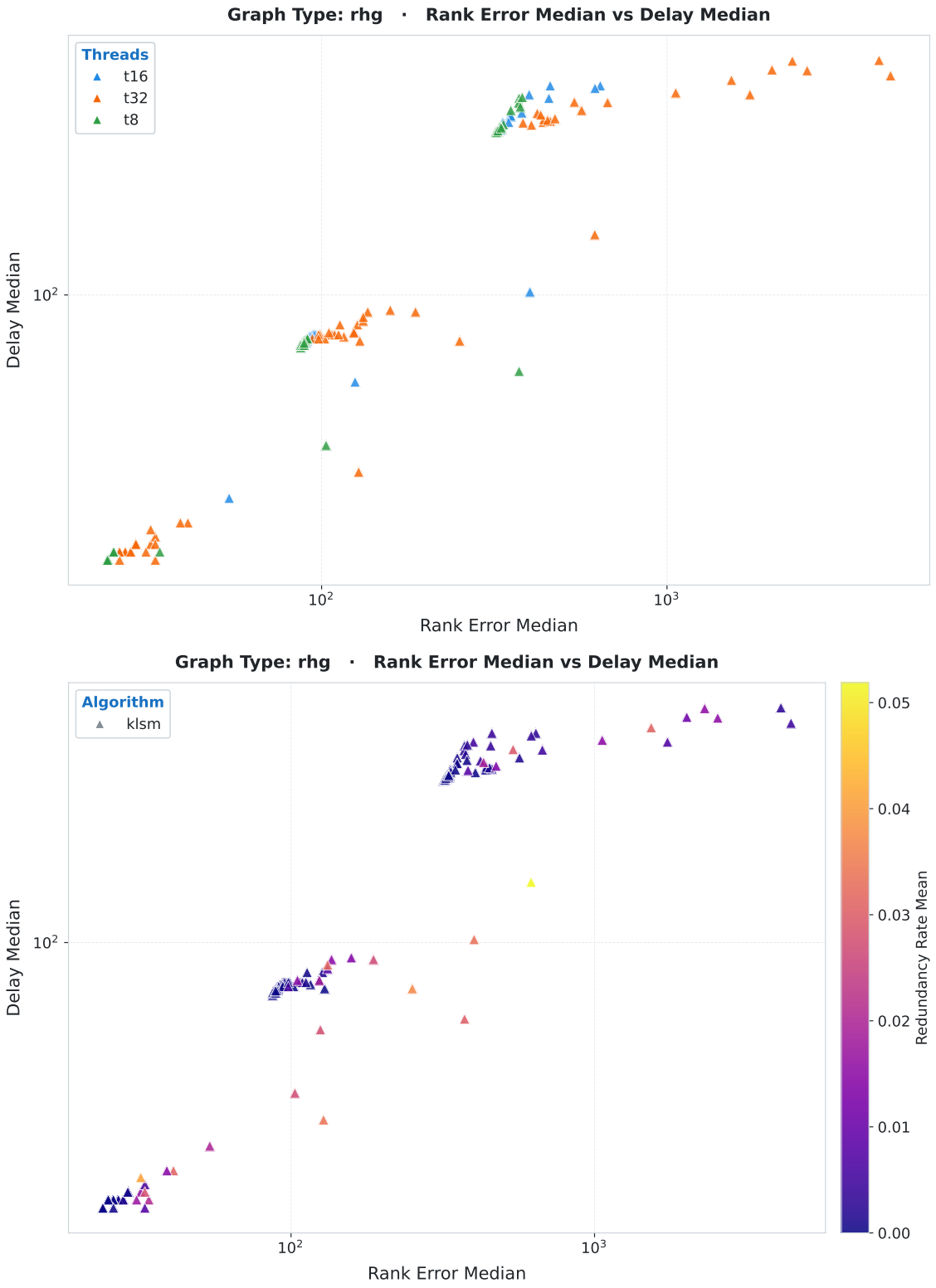

Figure 5.10: Summary plot for median rank error and median delay on all RHGs. Only k-LSM. Shape is determined by algorithm. Shaded by thread count and redundancy rate.

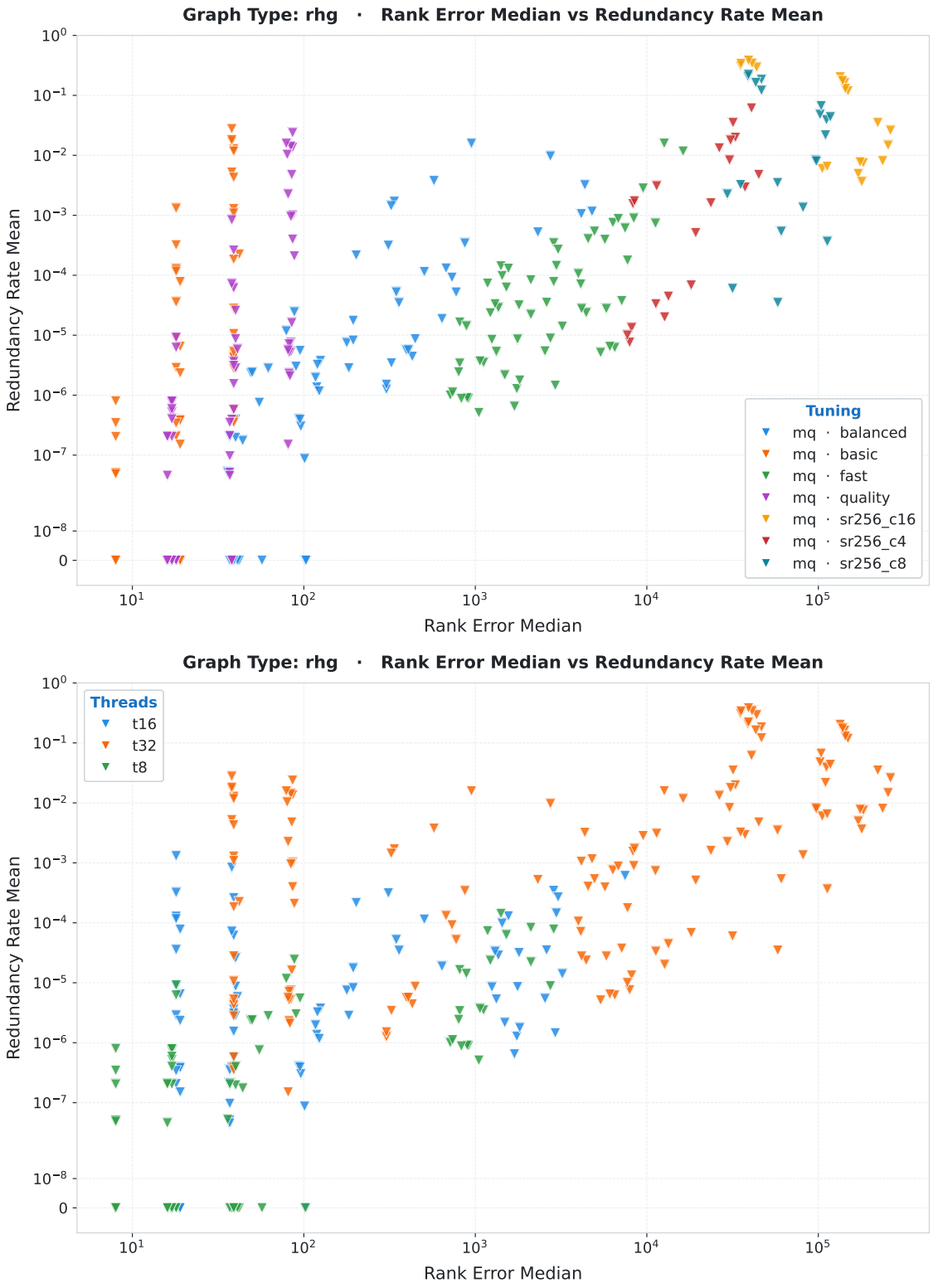

Figure 5.11: Summary plots for median rank error compared to mean redundancy rate on RHGs. Only MQ tunings. Shape is determined by algorithm. Shaded by tuning and thread count. *rhg*24_*d*8 and single threaded tunings excluded.

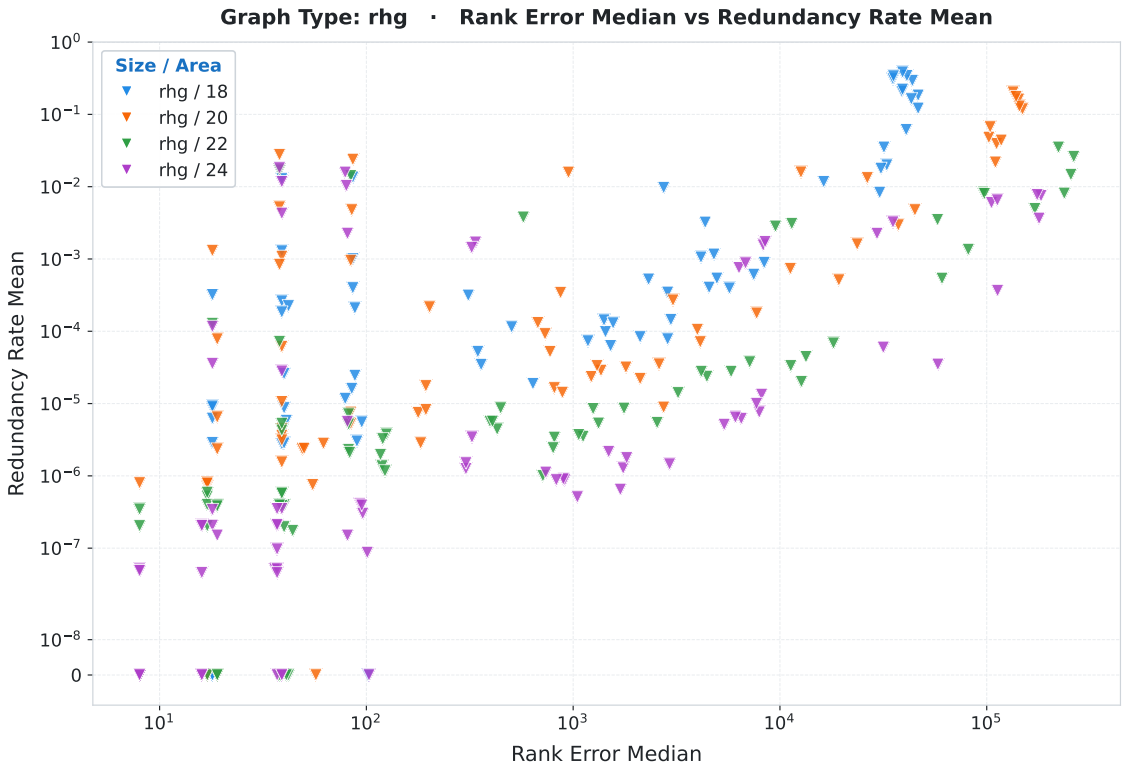

Figure 5.12: Summary plots for rank error and redundancy rate on RHGs. Only MQ tunings. Shaded by the graph size. $rhg24\_d8$ and single threaded tunings excluded.

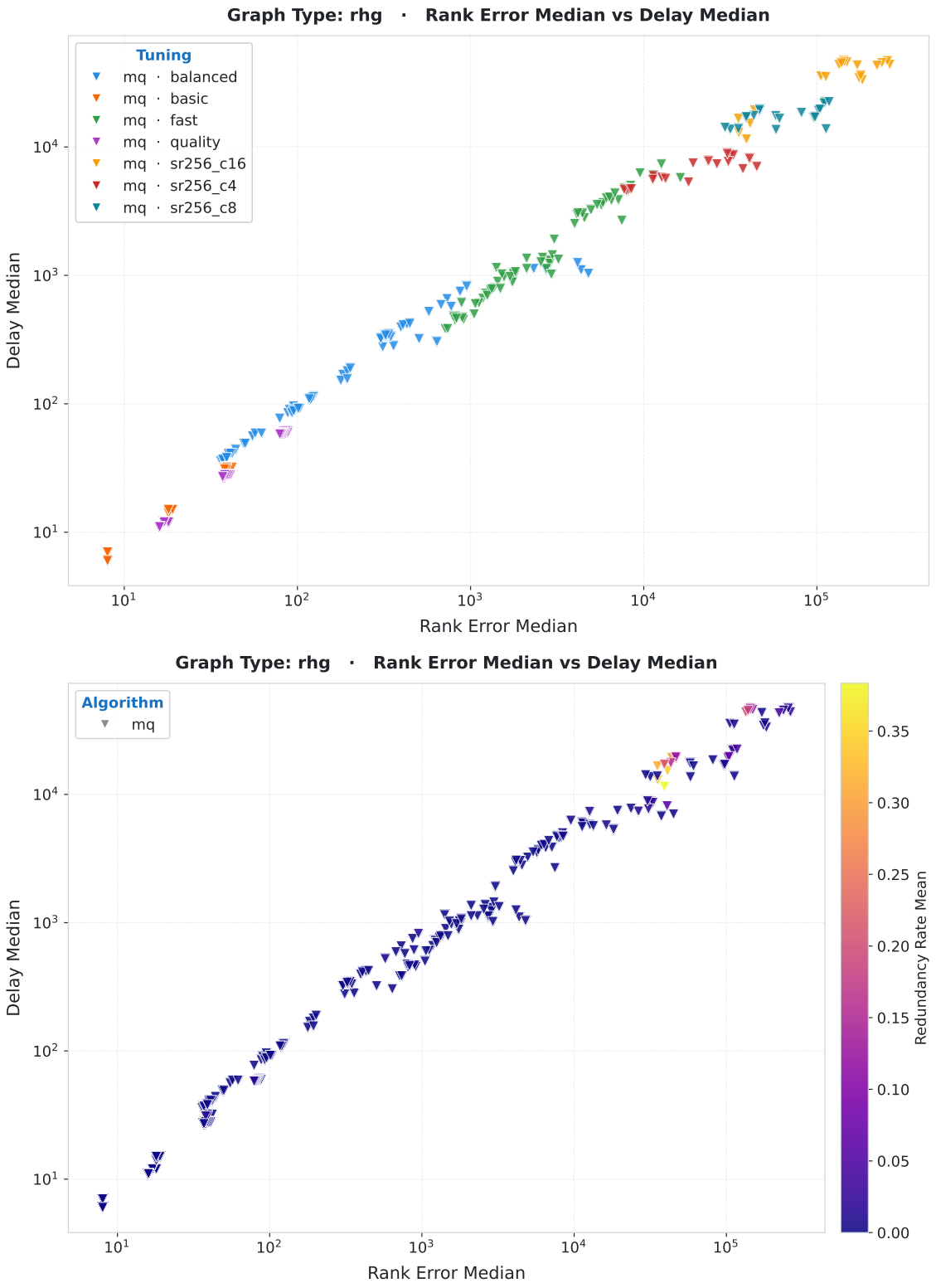

Figure 5.13: Summary plots for median rank error and redundancy rate on RHGs. Only MQ tunings. Shaded by tuning and redundancy rate. *rhg*24_*d*8 and single threaded tunings excluded.

5.1.2 Observations on Road- and Grid Graphs

Road- and grid graphs are grouped together due to their broadly similar behavior, shown in Figure 5.2. As with the RHGs, larger graphs tend to tolerate higher rank errors without a proportional increase in redundancy rate. This trend is visible in Figure 5.14, which shows most tunings across all road- and grid graphs. However, this relationship becomes less consistent as rank error grows. Beyond roughly 10^3, the differences in redundancy rate among configurations noticeably shrink. These data points approach the maximum redundancy rate of 1 at a lower rank error than the more irregularly interconnected RHGs.

A possible explanation for the pronounced effect of rank error on redundancy rate in these graphs is that their queues remain smaller than those of the RHGs, a point discussed further in section 5.2. Figure 5.15 shows two comparisons for most algorithms on all graphs: median rank error against median delay, and median rank error against median PQ-size, with the latter highlighting the redundancy rate. The plot reveals that a larger median queue size is associated with greater tolerance of rank errors in regards to the redundancy rate.

Looking specifically at DrPQ on the road- and grid networks, there is deviation from the configured rank errors, visible in Figure 5.14. Some deviation from the configured mean is expected, since elements removed from non-leading positions have a higher probability of being ignored. Additionally, smaller graphs may contain fewer queue elements than the configured rank error itself. This behavior can also be observed on the RHGs, but it is considerably more pronounced for road- and grid graphs. The attempt to generate data points for *drpq_mean*10000_*p*001 for *grid_graph_X*13_*Y*13 failed, but by observing the log-files for this run we were able to conclude that it would have ended up with a redundancy rate of over 0.9. Comparing this result to that of a similarly sized RHG with a redundancy rate of less than 10^{-4}, the impact a graph type can have on work-efficiency is highlighted.

Examining the skew of the road- and grid graphs, in Figure 5.16 one can observe that the relationship between median rank error and delay is not fully linear, but rather slightly curved. All data points are skewed to have a larger median rank error than delay. The least skewed data points belong to MQB. The best performing in matters of redundancy rate, and the least skewed tunings are all variants of MQ, With the experimental tunings of MQ (*sr*256_*c*4, *sr*256_*c*8, and *sr*256_*c*16) diverging the most.

Figure 5.16 compares median rank error and median delay for road- and grid graphs, with data points colored by redundancy rate and tuning. The relationship is not perfectly linear but slightly curved, and all points fall below the diagonal, showing that median rank error exceeds median delay. The least skewed data points belong to MQB. Overall, MQ variants achieve both the best redundancy rates and the smallest skew, with the exception of the experimental configurations *sr*256_*c*4, *sr*256_*c*8, and *sr*256_*c*16, which diverge considerably more. CA-PQ likely detects high contention, leading to the heavily skewed behavior observed at higher thread

counts. k-LSM exhibits substantially larger skew on road- and grid graphs than on the RHGs. Local buffers seem detrimental when exploring less interconnected graph structures, as elements drawn from them are more likely to be suboptimal, potentially increasing the redundancy rate.

Figure 5.17 shows the same relationships as Figure 5.16, but with data points colored by graph type. The skewed CA-PQ points appear predominantly on the smaller graphs (NY and $X10\_Y10$), suggesting that CA-PQ requires larger graphs to stabilize and achieve higher work-efficiency. This observation is reinforced by the same effect on RHGs seen in Figure 5.8. More broadly, this might suggest that maintaining stable relaxation behavior is harder on smaller graphs. Several configurations on NY and $X10\_Y10$ tend to reach higher redundancy rates than their counterparts on larger instances, while other algorithms remain unaffected.

Overall, work-efficiency on road- and grid graphs cannot be explained by rank error or the skew between rank error and delay alone. It depends on multiple interacting factors that are not fully captured by relaxation metrics. Queue size and graph connectivity play a decisive role as smaller queues and sparser structures make the same relaxation errors more costly. Small graphs amplify this effect, and algorithm tuning, while important, does not determine performance independently of graph characteristics. As with the RHGs, these results show that the translation from relaxation to redundant work is shaped by several interdependent mechanisms beyond the relaxation errors themselves.

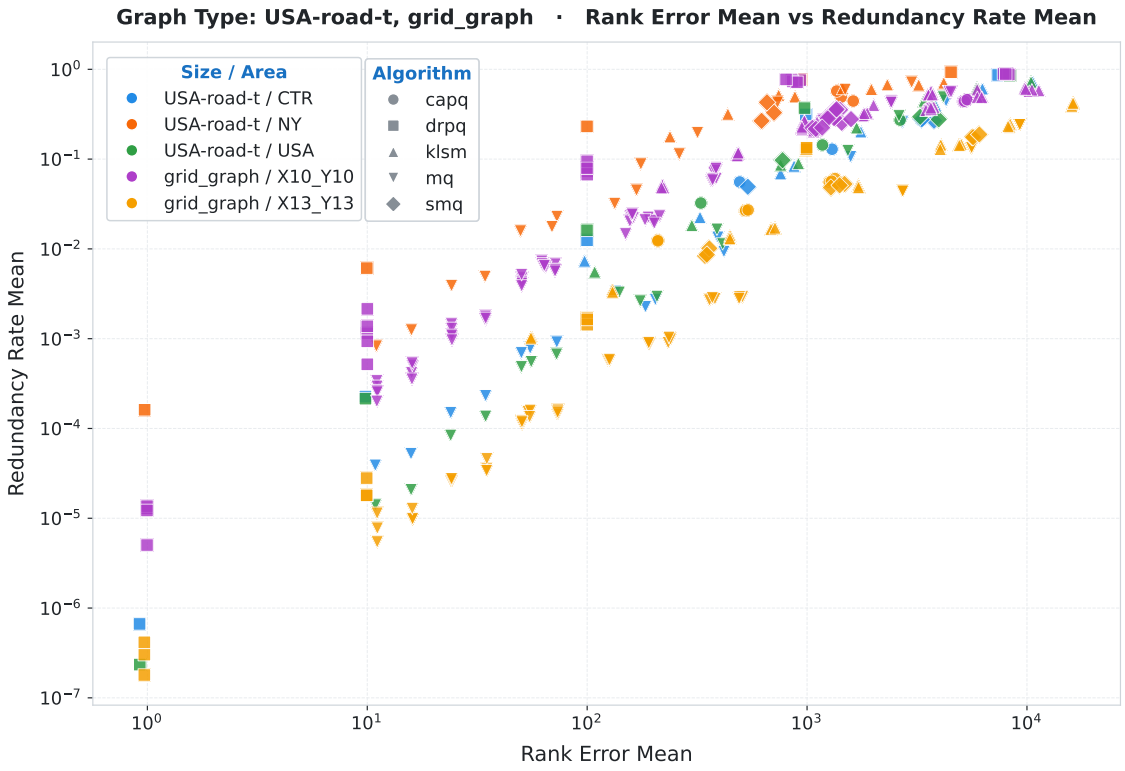

Figure 5.14: Summary plot for mean rank error and redundancy rate for road- and grid graphs. Shape is determined by algorithm. Shaded by graph size/area. *drpq_mean*0, *drpq_mean*1 and *mq_basic_t*1 excluded.

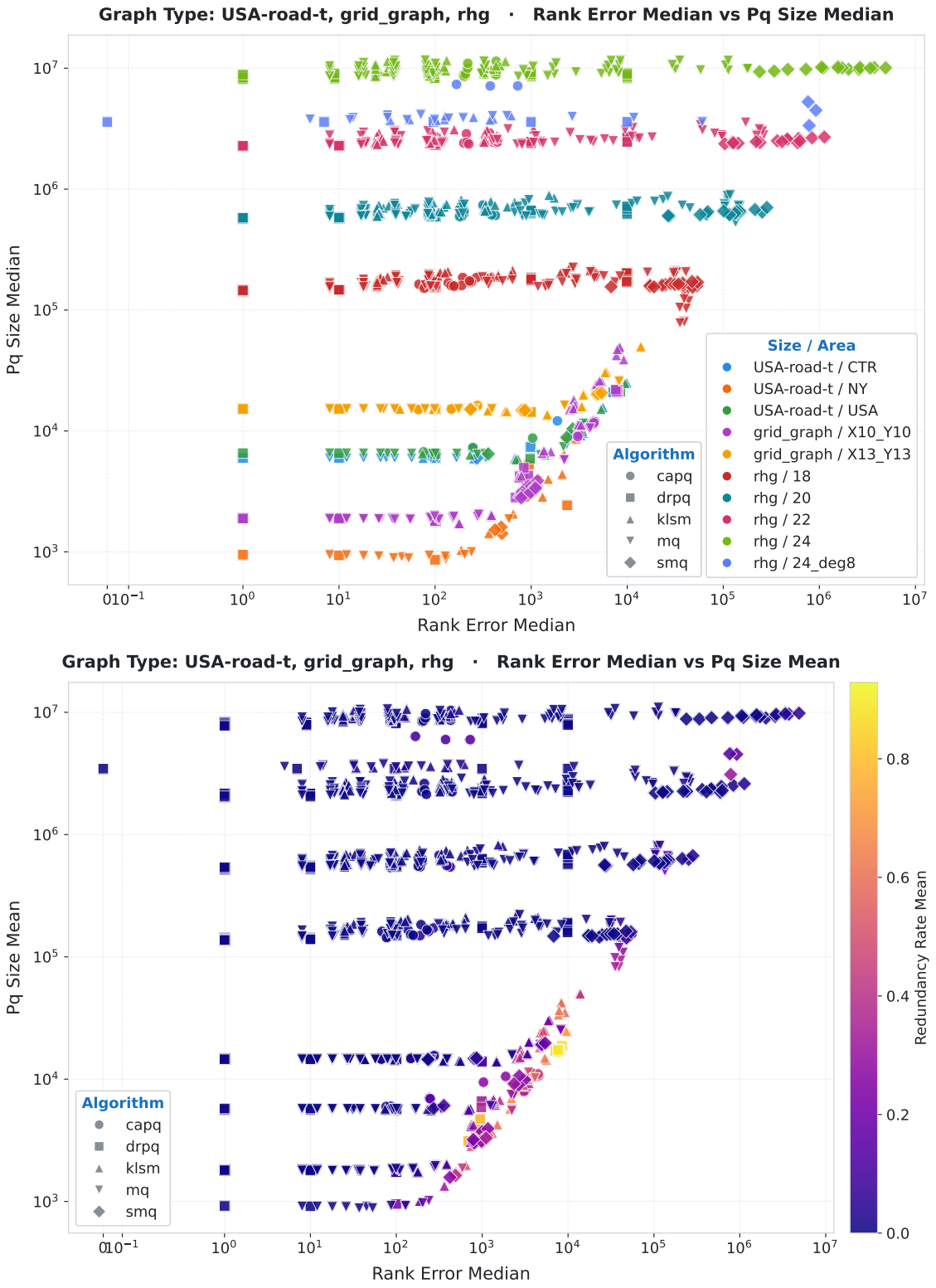

Figure 5.15: Summary plots for median rank error and median PQ-size for all graphs. Shape is determined by algorithm. Shaded by size/area and redundancy rate. *drpq_mean*0, *drpq_mean*1 and *mq_basic_t*1 excluded.

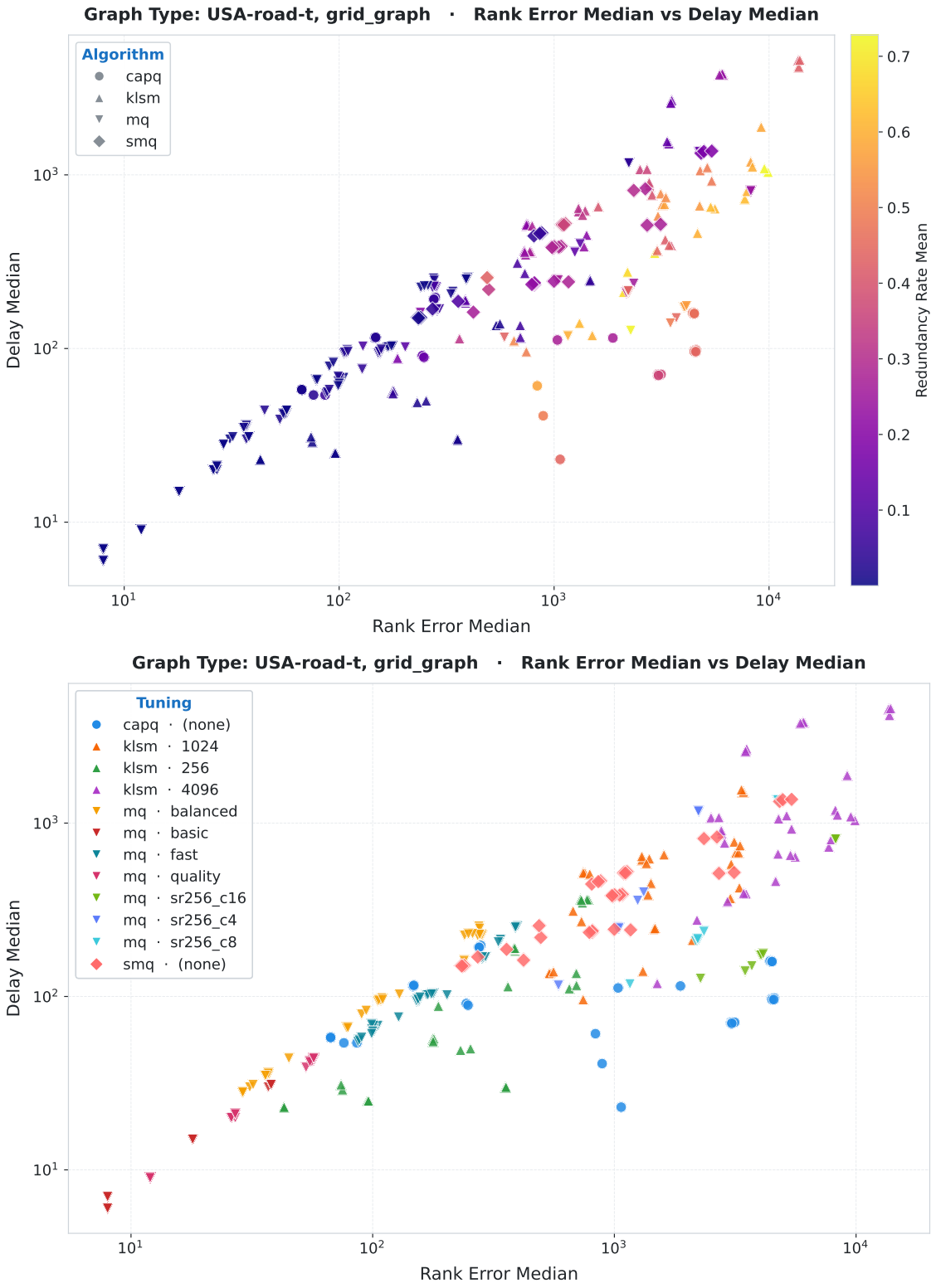

Figure 5.16: Summary plot for median rank error and median delay for road- and grid graphs. Shape is determined by algorithm. Shaded by redundancy rate and tuning. Single threaded algorithms excluded.

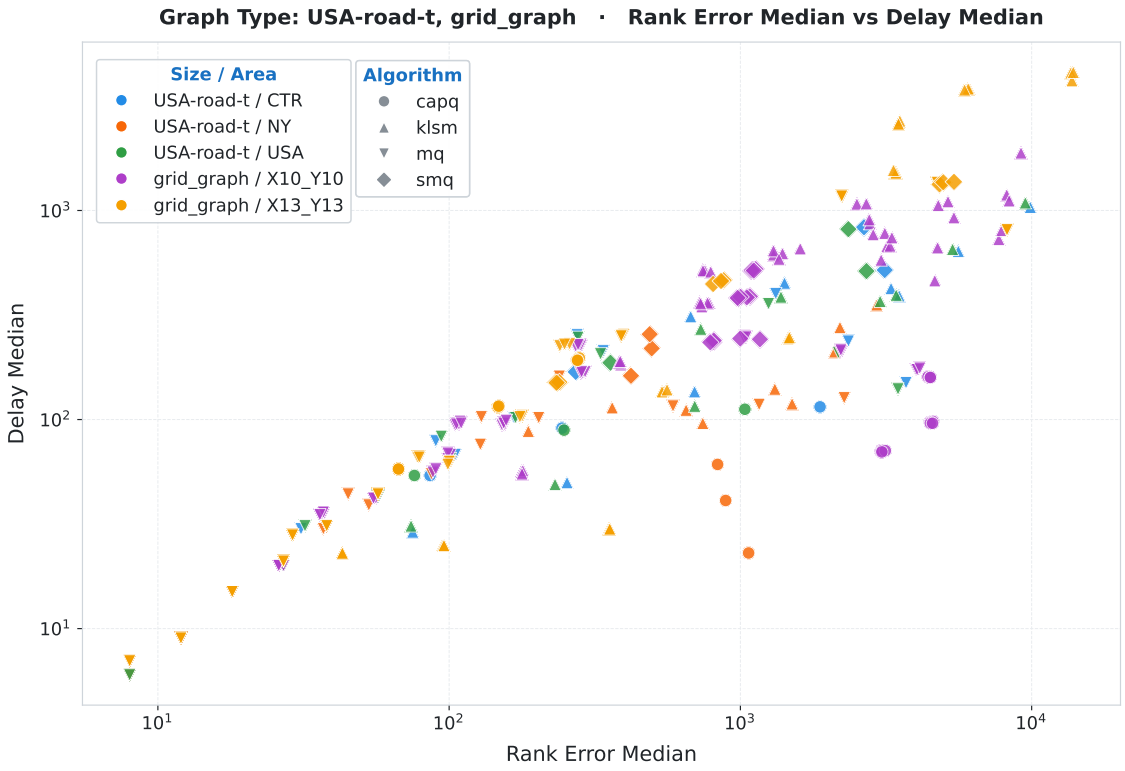

Figure 5.17: Summary plot for median rank error and median delay for road- and grid graphs. Shape is determined by algorithm. Shaded by size/area. Single threaded algorithms excluded.

5.2 Comparison on Bucketed run-time plots

The bucketed run-time plots show when redundancies occur during a run relative to rank error and delay. They also show whether these metrics remain stable throughout execution or vary over time. Across graph types, certain patterns remain consistent.

In Figure 5.18, the chart shows the PQ-size across all RHGs. RHGs rapidly reach their maximum PQ-size and then gradually decrease the queue size over the remainder of the run, which is consistent with expected behavior for this graph type. This pattern can be contrasted with the behavior on a grid graph, shown in Figure 5.19, which compares the median PQ-sizes for different tunings. In the grid graph most algorithms behave similarly, approximately following the same shaped trend line with some jaggedness. However, the k-LSM tuning with the largest k-value has a significantly inflated PQ-size compared to the other algorithms, and for this graph type, that inflation correlates with poor work efficiency.

RHGs tend to generate more extra work early in the run than in later stages, as can be observed in Figure 5.20, which displays the rate of which extra work is generated for algorithms on an RHG over a run. A notable exception of trend is the SMQ which has a consistently worse performance than the other tunings across the run. This is likely due to structural properties of the graph. Community nodes typically have a higher number of connections than non-community nodes, which makes them more likely to be encountered and processed early in the search. While cross-community edges can occasionally lead to alternative paths, they are relatively sparse and therefore contribute less to overall work propagation. As exploration continues, most high-impact nodes have already been processed, leading to a reduction in additional work over time.

Due to the label-correcting nature of the underlying shortest-path computation, once most community nodes have been encountered, further relaxations are unlikely to generate additional queue insertions, for similar reasons as the observed decline in extra work generated.

Algorithms with local buffers, such as k-LSM and CA-PQs, perform well in these settings. This suggests that isolating work within localized thread contexts may align well with the community structure of RHGs, where each region has a relatively strong internal connectivity. Figure 5.21 and Figure 5.22 illustrate the mean extra work generation, in comparison to median bucketed rank error and delay, as well as median pq-size, bucketed, for two tunings of these algorithms. The other tunings within these algorithm types perform similarly, producing very low extra work generation across the entire run. This importance of locality is not something we can definitively prove in this thesis, but merely a hypothesis based on the observations in the data at hand.

By contrast, SMQ does not perform as consistently on these graphs. With similar

hypothetical reasoning, this may be due to sub-queues mixing elements from different communities through work stealing, although the exact mechanism is not fully established, as seen in Figure 5.23 which shows the median rank error and delay, PQ-size and extra work generation of an SMQ on an RHG. More generally, the behavior of SMQ on RHGs differs from that of the other algorithms, as cross-thread node movement appears to introduce additional overhead and reduce work efficiency.

Once again, the results suggest that work efficiency on RHGs is not determined by a single factor. Instead, it emerges from the interaction between algorithm design, graph structure, and how work is distributed over the course of execution. Different algorithms exhibit different sensitivities to these factors, and no single explanation accounts for all observed behavior across the tested configurations.

The following charts reveal how relaxation metrics evolve during execution on *grid_graph_X*13_*Y*13. All plots display the median PQ-size, median rank error, median delay, and either the mean extra work generated, each for a different algorithm: Figure 5.24 (CA-PQ), Figure 5.25 (MQB), Figure 5.26 (MQQ), Figure 5.27 (MQF), and Figure 5.28 (k-LSM, $k = 4096$). Across these configurations, one can observe that the extra work generated peaks whenever the rank error surges as well, but these peaks are not directly correlated or proportional. In these bucketed runs, skews between median rank error and delay lead to higher extra work generation on road- and grid graphs than on the RHGs. The least skewed chart shows the MQB, with very similar rank errors to delays.

The contention avoiding behavior of CA-PQ can be observed in Figure 5.24, in the beginning and in the end, where the PQ-size and extra work generation spike. There is an obvious skew of the rank error in comparison to the delay, with significant spikes in extra work generation wherever they diverge further. Additionally one can observe how CA-PQ stabilizes its rank error and delay over time, which might suggest the unstable behavior on smaller graphs observed in subsection 5.1.2, this can also be observed in Figure 5.22.

Overall, the bucketed run-time plots show that work-efficiency is influenced not only by the magnitude of relaxation errors, but also by when they occur during execution. Algorithms with similar aggregate rank error or delay can be differentiated from by examining patterns of extra work generation over time. The results therefore complement the summary plots by demonstrating that work-efficiency depends on the interaction between graph structure, algorithmic behavior, and the distribution and magnitude of relaxations. No single metric consistently explains the observed behavior across all graph types and algorithms.

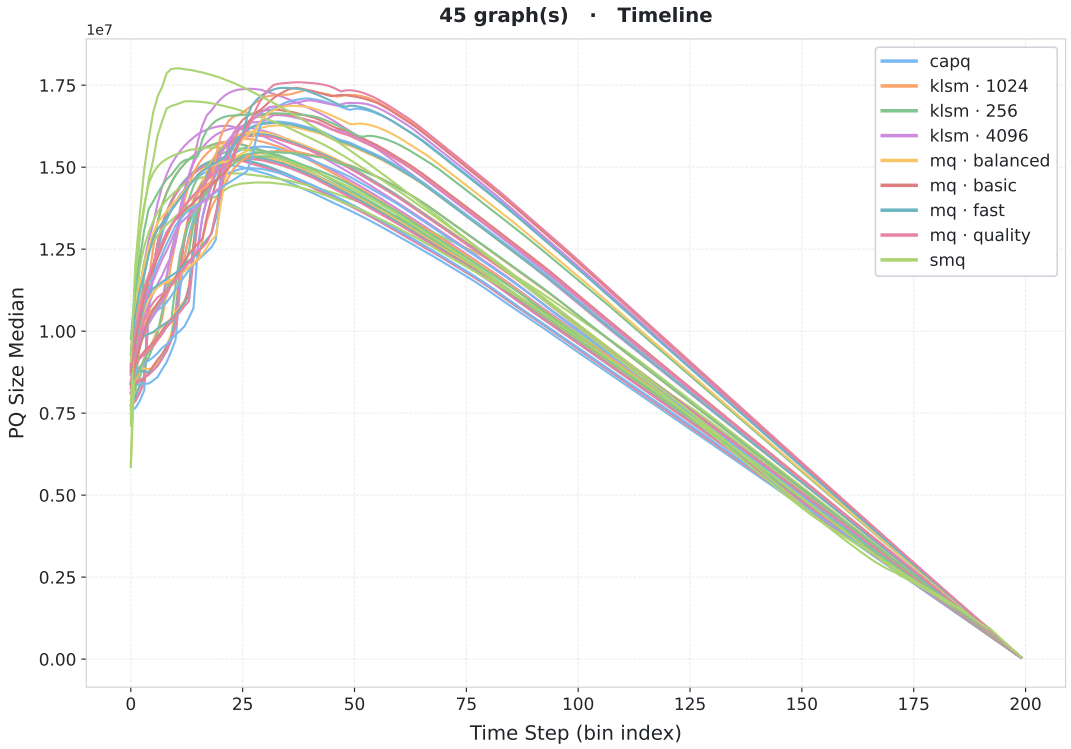

Figure 5.18: Bucketed run-time plot showcasing median PQ-size for most algorithms on *rhg*24. Only 32 threads over 5 seeds. Experimental MQ tunings excluded.

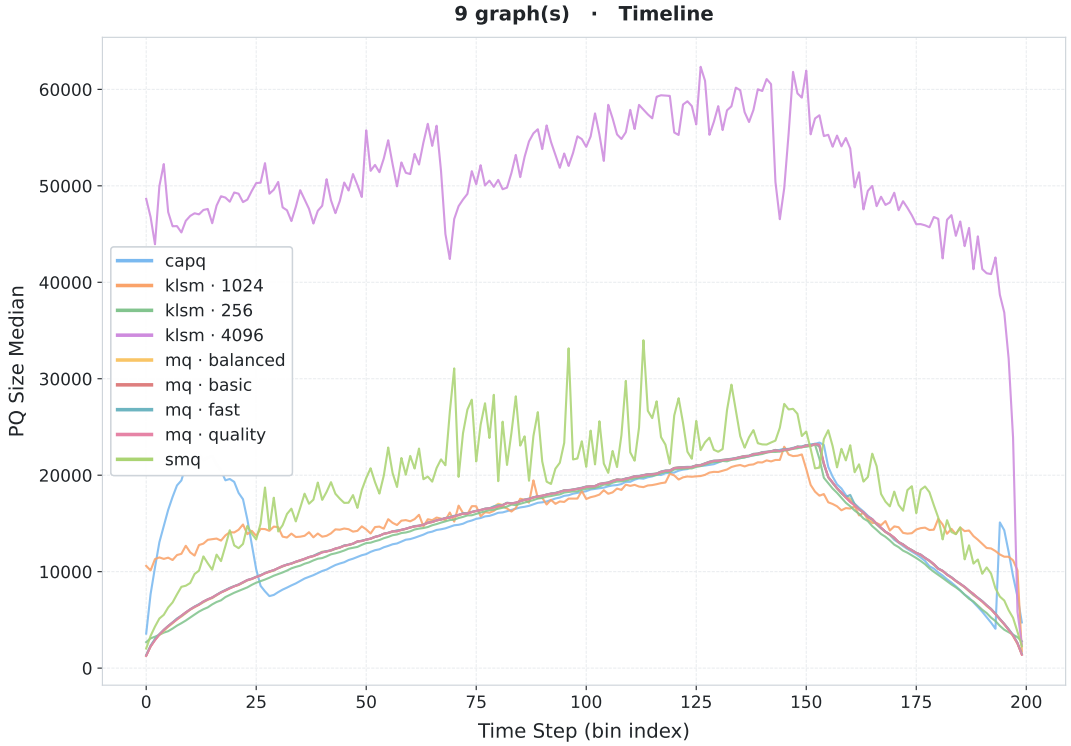

Figure 5.19: Bucketed run-time plot showcasing median PQ-size for most algorithms on *grid_graph_X*13*_Y*13. Only 32 threads and 1 seed. Experimental MQ tunings excluded.

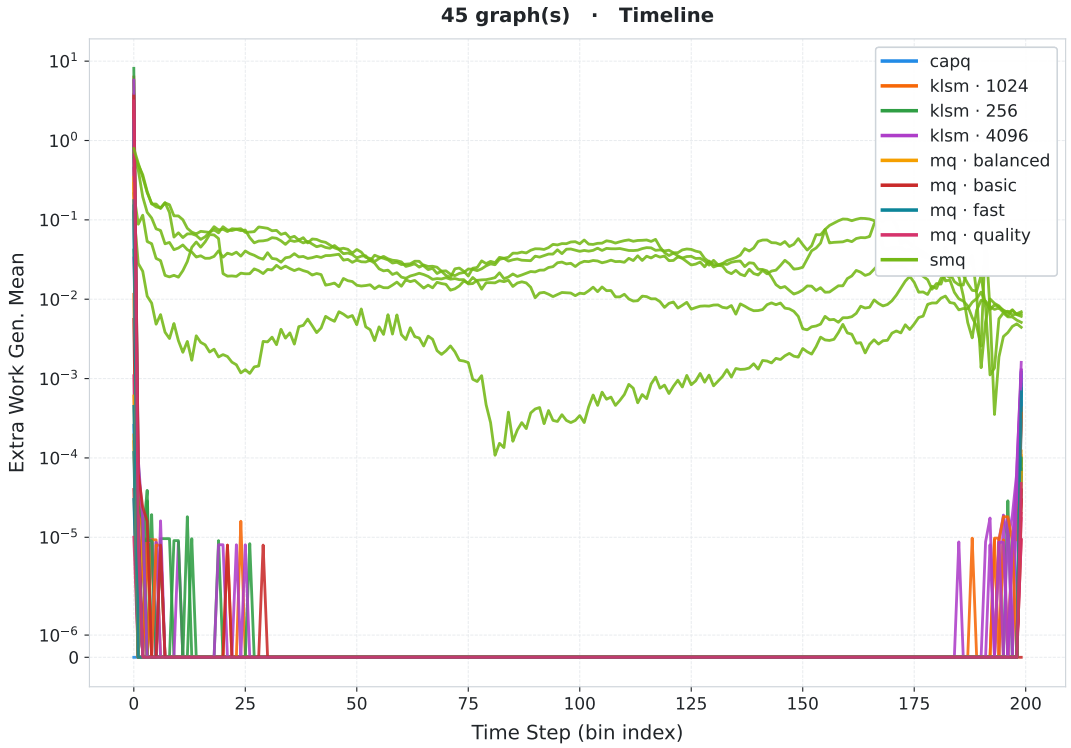

Figure 5.20: Bucketed run-time plot showcasing the extra work generation for multiple algorithms on *rhg*24. Only 32 threads over 5 seeds. Experimental configurations excluded.

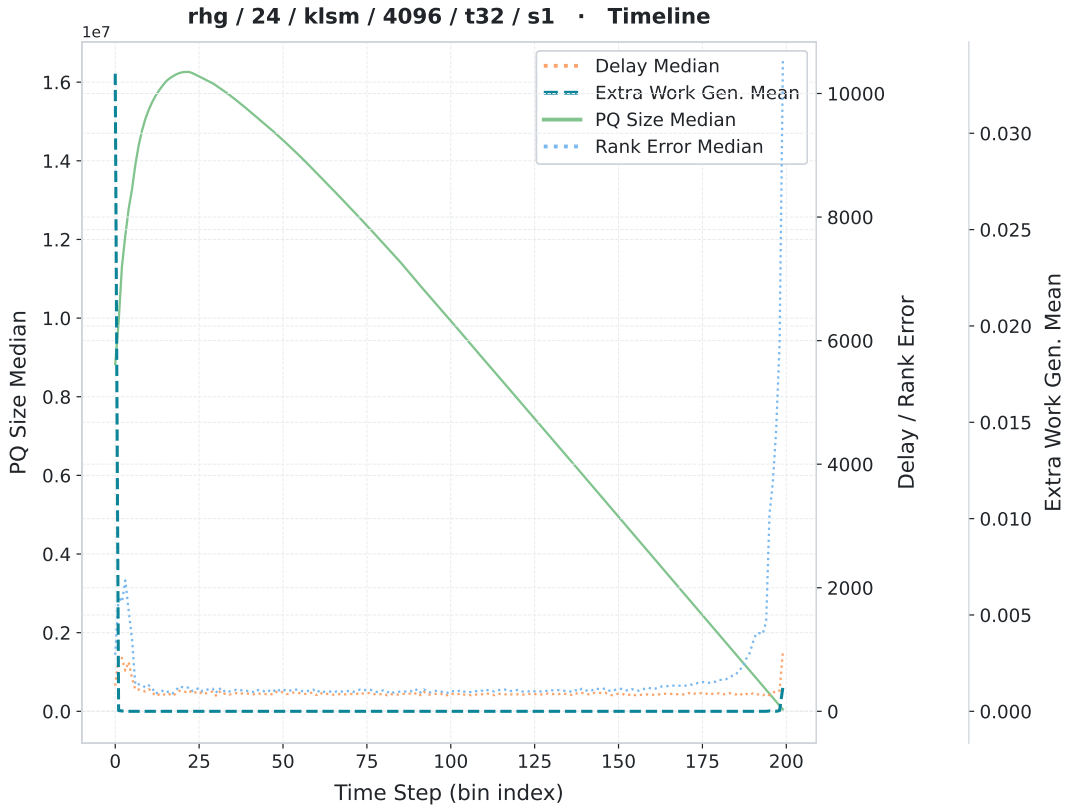

Figure 5.21: Bucketed run-time plot showcasing PQ-size compared to delay/rank error and the extra work generation for k-LSM_4096 on *rhg*24. Only 32 threads and 1 seed.

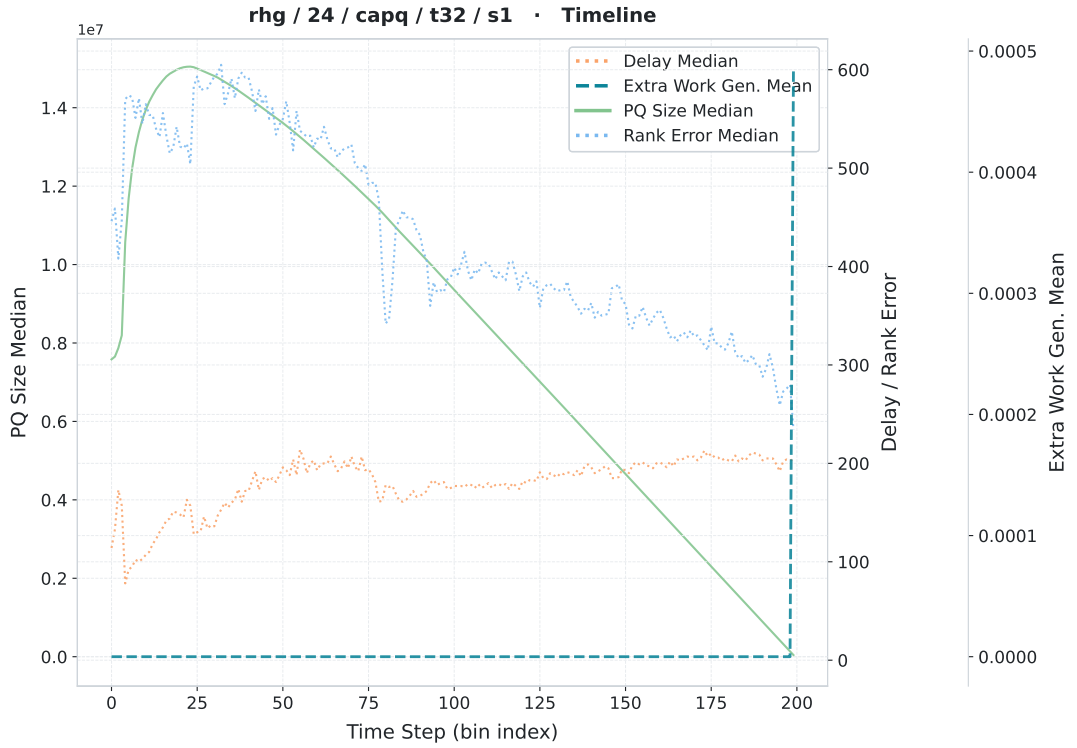

Figure 5.22: Bucketed run-time plot showcasing PQ-size compared to delay/rank error and the extra work generationfor CA-PQ on *rhg*24. Only 32 threads and 1 seed.

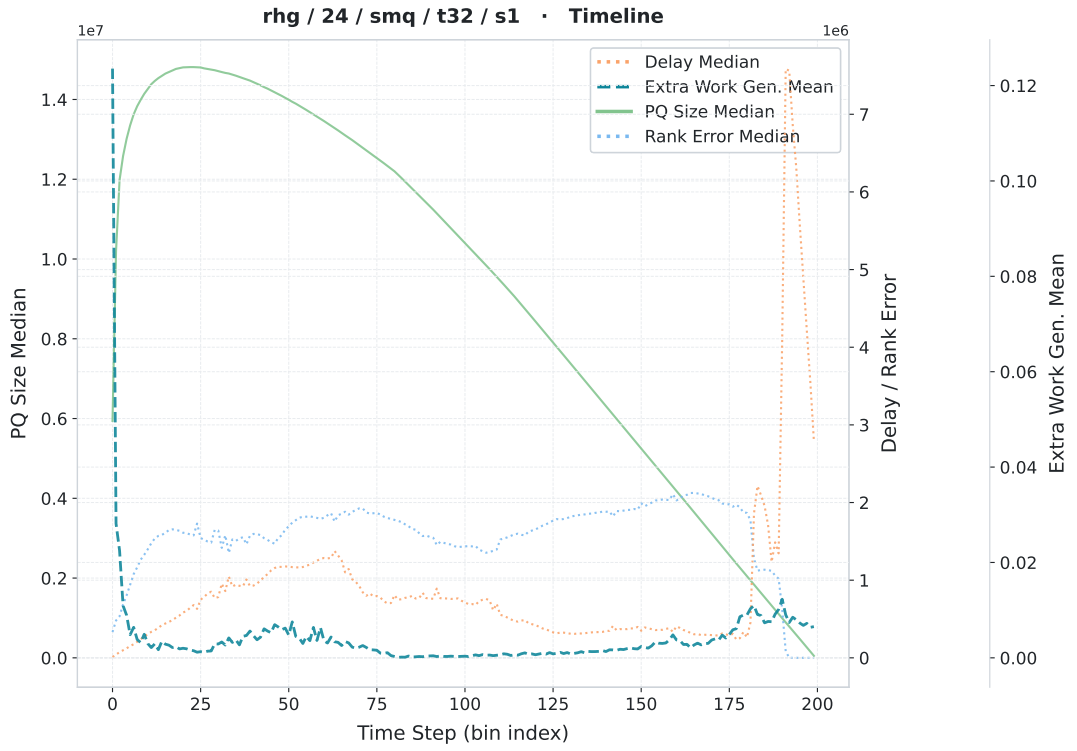

Figure 5.23: Bucketed run-time plot showcasing PQ-size compared to delay/rank error and the extra work generation for Stealing-MQ on *rhg*24. Only 32 threads over 5 seeds.

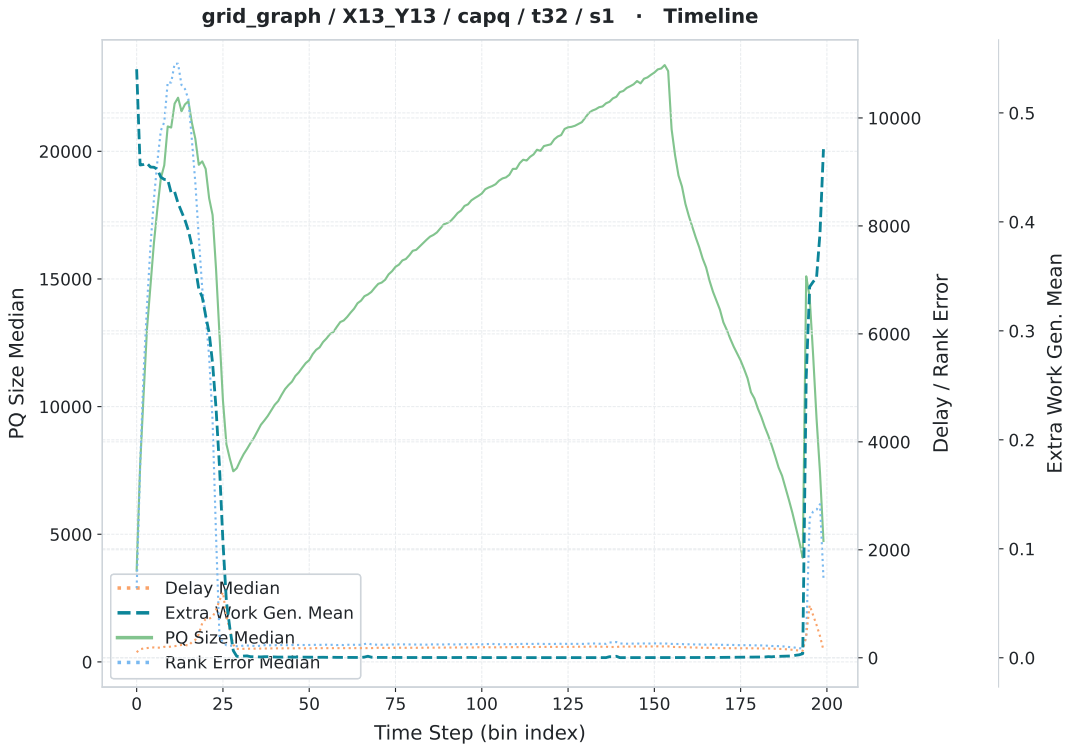

Figure 5.24: Bucketed run-time plotshowcasing median PQ-size compared to extra work generated for CA-PQ on *grid_graph_X13_Y13*. Only 32 threads and 1 seed.

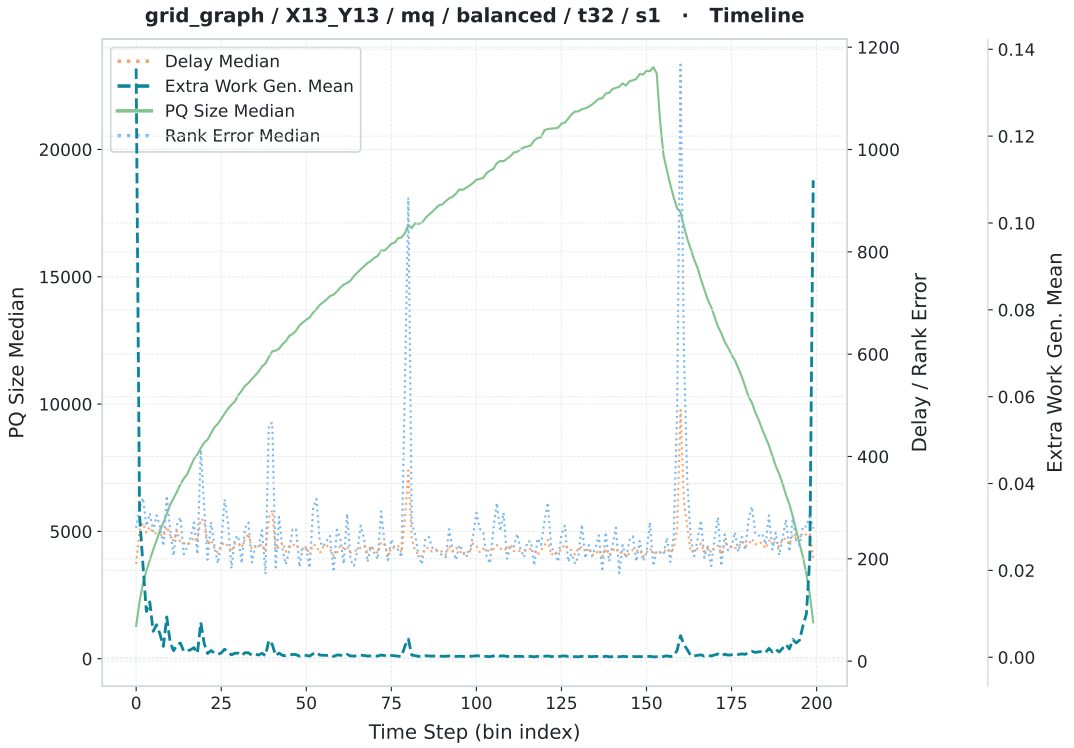

Figure 5.25: Bucketed run-time plot showcasing PQ-size compared delay/rank error and the extra work generation for MQ-balanced on *grid_graph_X13_Y13*. Only 32 threads and 1 seed.

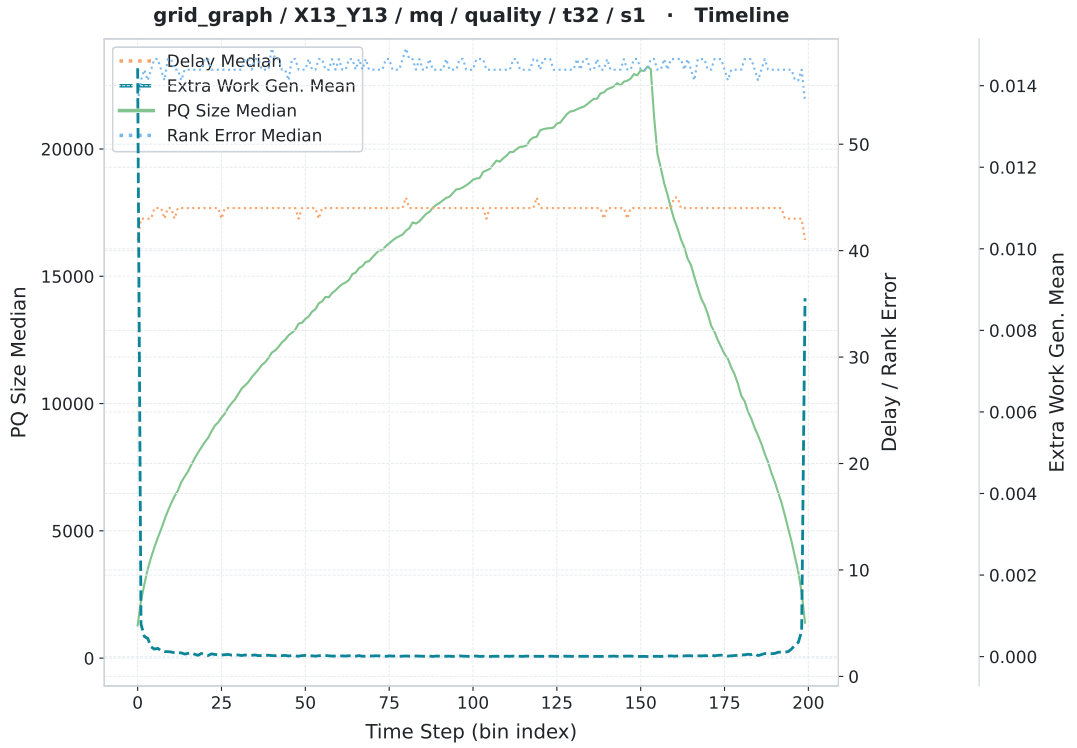

Figure 5.26: Bucketed run-time plot showcasing PQ-size compared delay/rank error and the extra work generation for MQ-quality on *grid_graph_X*13*_Y*13. Only 32 threads and 1 seed.

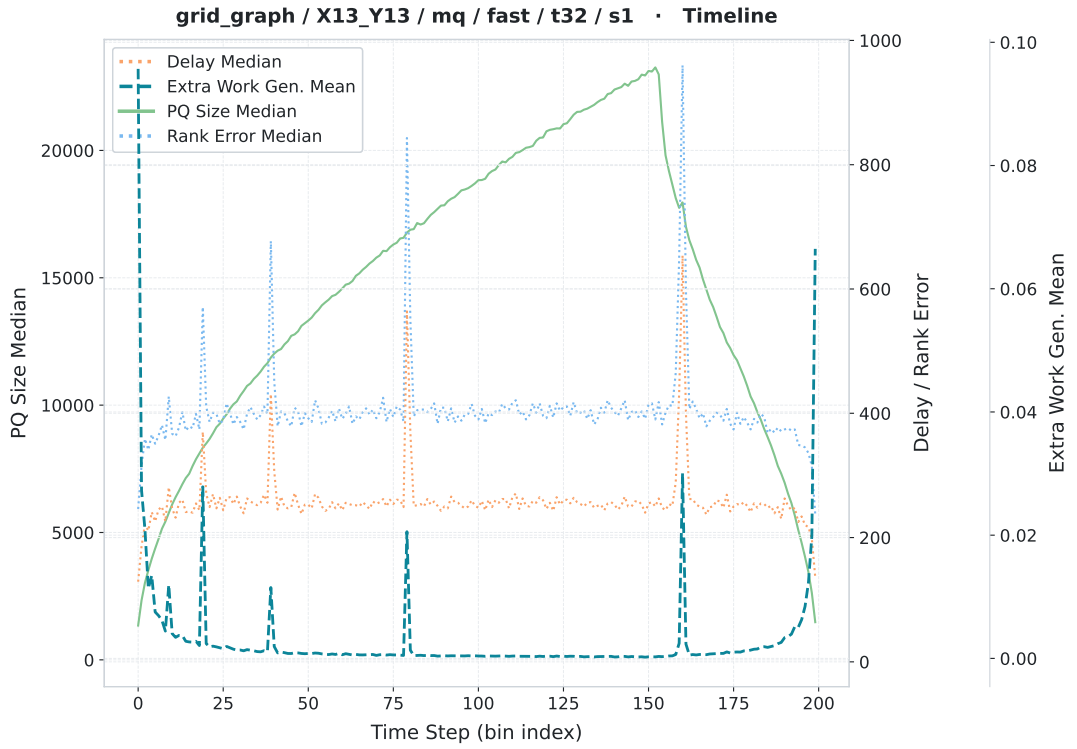

Figure 5.27: Bucketed run-time plot showcasing PQ-size compared delay/rank error and the extra work generation for MQ-fast on *grid_graph_X*13*_Y*13. Only 32 threads and 1 seed.

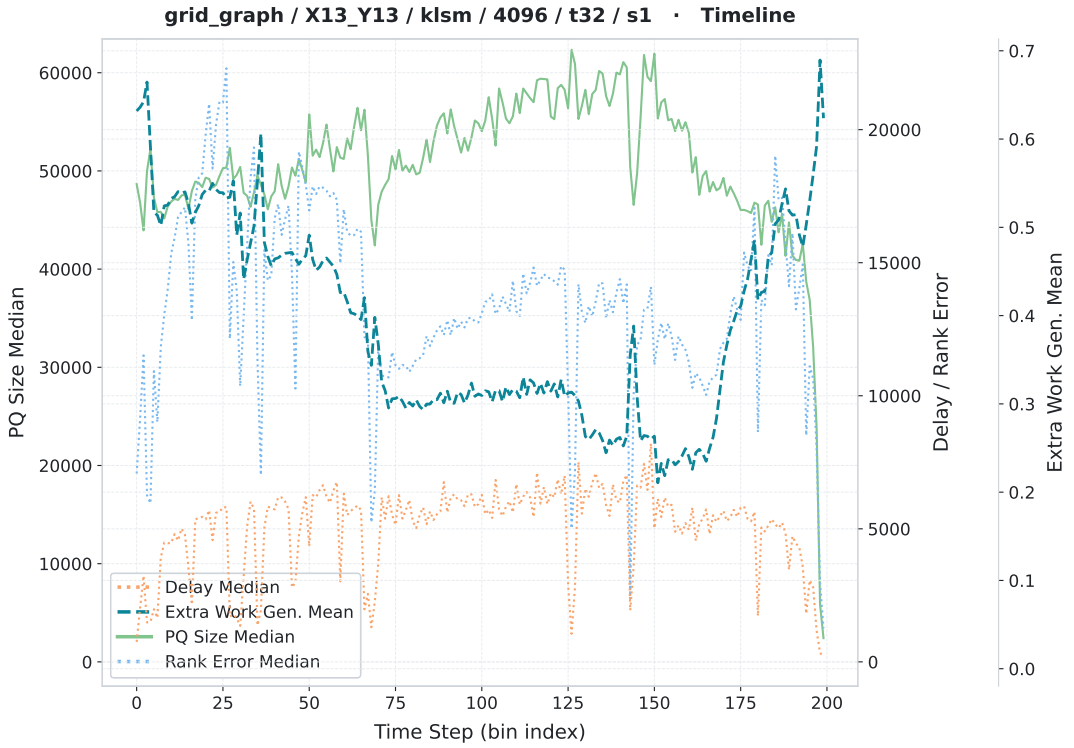

Figure 5.28: Bucketed run-time plot showcasing PQ-size compared to delay/rank error and the extra work generation for k-LSM_4096 on *grid_graph_X*13_*Y*13. Only 32 threads and 1 seed.

6

Future work

In retrospect, several aspects of the study could be refined, particularly regarding graph metrics and selection. The graph set was limited; several types we intended to include were omitted due to time constraints. Notably, additional RHGs with different average degrees would help clarify how degree distribution influences algorithm performance.

Most RHG variants were tested with five random seeds, but $rhg24\_deg8$ relied on a single seed, so its results may be unrepresentative. More seeds across all variants would strengthen our conclusions.

The collected data is heavily skewed toward RHGs, more measurements on road and grid graphs would yield a more balanced comparison. The current grid graphs have an average degree of 3–4 with little per-node variation; exploring grids with a wider degree spread could be insightful. Although grid graphs were intended as synthetic road networks, noticeable differences remain, motivating the search for more faithful synthetic road-like models.

Additional graph types, such as Erdős–Rényi random graphs and MAWI-based traffic graphs, were originally planned but omitted to limit the project's scope. Erdős–Rényi graphs connect each pair of vertices independently with a fixed probability. They would have helped isolate algorithm-specific effects from structural properties like community structure or locality. MAWI graphs, derived from real network traffic traces, would have introduced greater connectivity irregularity and tested work-efficiency under more realistic, less idealized conditions.

Another limitation is that only a subset of the collected metrics was ultimately used in the analysis. While the study primarily focused on median rank error, median delay, and redundancy-related metrics, the data collection process also recorded additional statistics such as means, maxima, and multiple percentile measurements. Metrics related to ignored nodes were likewise collected but remained unexplored. These measurements were omitted in order to keep the scope of the analysis manageable and to focus on the metrics thought to be the most directly related to work-efficiency and relaxation. Future work could investigate whether these additional metrics provide a more complete characterization of relaxation behavior. In partic-

ular, higher percentiles and maximum values may capture extreme events that are not reflected by median-based summaries. Such measures could help explain some of the observed discrepancies between relaxation metrics and work-efficiency, especially in cases where a small number of large outliers appear to have a disproportionate impact on redundant work.

SMQ and CA-PQ are designed to benefit from NUMA architectures, so their full potential is likely unrealized on our single-socket system. Evaluating them on multi-socket hardware would be a natural extension.

Beyond hardware, several algorithms and scheduler designs were excluded from the evaluation to keep the scope manageable. Notably, Wasp, Julienne with Δ-stepping, Galois, and the adaptive AdaMW scheduler all offer alternative approaches to relaxed scheduling, each with distinct trade-offs between parallelism and work-efficiency.

Finally, our work deliberately set aside throughput and speed metrics, as earlier studies had already focused extensively on speedup. Nevertheless, per-operation timestamps were recorded and later discarded during data cleaning. This data could be used to examine how work-efficiency, rank error, and delay relate to throughput and effective throughput over time. Exploring this relationship was beyond our current scope, but it remains a promising direction for future work.

7

Conclusion

This thesis investigated how relaxation affects the work-efficiency of label-correcting parallel SSSP algorithms. The primary objective was to understand whether commonly used relaxation metrics, particularly rank error and delay, can explain the additional work generated by relaxed schedulers, and how graph characteristics influence this relationship.

Our experiments confirm that relaxation and work-efficiency are related, but the relationship is more complex than previously assumed. Rank error generally correlates with redundancy rate, indicating that larger deviations from strict priority ordering tend to increase redundant work. However, rank error alone is not sufficient to explain work-efficiency. Configurations with similar rank errors frequently exhibit substantially different redundancy rates, demonstrating that additional factors influence how relaxation translates into extra work.

The results also show that graph structure plays a decisive role. RHGs consistently exhibit higher tolerance to relaxation, often maintaining good work-efficiency despite relatively large rank errors. In contrast, road networks and grid graphs are more sensitive to relaxation, where comparable rank errors can result in substantially higher redundancy. Higher graph connectivity and larger graph sizes appear to mitigate the negative effects of relaxation, while smaller and less interconnected graphs amplify them.

We further examined the relationship between rank error and delay. Although the two metrics are theoretically equivalent on mean, median-based analyses reveal systematic differences between them. The skew between rank error and delay provides useful insight into scheduler behavior, some schedulers achieve high work-efficiency despite large skew. Nevertheless, this skew is not a universal predictor of redundant work, and several notable exceptions were observed.

The bucketed runtime analysis shows that work-efficiency depends not only on aggregate metrics but also on when relaxation occurs during execution. Peaks in extra work generation often coincide with divergences between the rank error or delay, but the relationship is neither proportional nor consistent across algorithms and graph types. Additionally we can see trends when during runtime extra work is generated

for different stages of the algorithms as well as for different graph types.

Taken together, the results suggest that work-efficiency in relaxed parallel SSSP algorithms cannot be explained by a single metric. Instead, it emerges from the interaction between scheduler design, graph structure and relaxation behavior over time. While average and median rank error remains a useful indicator of relaxation, it is not sufficient to predict the work-efficiency. Instead, we also have to look at both their distributions, and how they are distributed over time, as well as the graph structure.

The main contribution of this thesis is therefore a more nuanced understanding of how relaxation affects work-efficiency in parallel SSSP algorithms. By extending existing benchmarking infrastructure, introducing the DrPQ scheduler, and evaluating multiple graph types and scheduler configurations, we show that the cost of relaxation is highly context dependent. Future work should extend the evaluation to more diverse graph classes, richer metric usage, and additional hardware and algorithmic settings, as these constraints limit how far the current conclusions can be generalized.

Bibliography

[1] N. Shavit, "Data structures in the multicore age," *Commun. ACM*, vol. 54, no. 3, pp. 76–84, Mar. 2011, ISSN: 0001-0782. DOI: 10.1145/1897852.1897873. [Online]. Available: https://doi.org/10.1145/1897852.1897873.

[2] M. Williams, P. Sanders, and R. Dementiev, "Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues," in *29th Annual European Symposium on Algorithms (ESA 2021)*, P. Mutzel, R. Pagh, and G. Herman, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 204, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 81:1–81:17, ISBN: 978-3-95977-204-4. DOI: 10.4230/LIPIcs.ESA.2021.81. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2021.81.

[3] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova, "Quantitative relaxation of concurrent data structures," in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '13, Rome, Italy: Association for Computing Machinery, 2013, pp. 317–328, ISBN: 9781450318327. DOI: 10.1145/2429069.2429109. [Online]. Available: https://doi.org/10.1145/2429069.2429109.

[4] A. Lenharth, D. Nguyen, and K. Pingali, "Priority queues are not good concurrent priority schedulers," in *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 209–221, ISBN: 978-3-662-48096-0.

[5] M. Williams and P. Sanders, "The multiqueue: A simple and fast relaxed concurrent priority queue," *ACM Transactions on Parallel Computing*, vol. 12, no. 4, Dec. 2025, ISSN: 2329-4949. DOI: 10.1145/3771738. [Online]. Available: https://doi.org/10.1145/3771738.

[6] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, "The lock-free k-lsm relaxed priority queue," *SIGPLAN Not.*, vol. 50, no. 8, pp. 277–278, Jan. 2015, ISSN: 0362-1340. DOI: 10.1145/2858788.2688547. [Online]. Available: https://doi.org/10.1145/2858788.2688547.

[7] K. Sagonas and K. Winblad, "The contention avoiding concurrent priority queue," in *Languages and Compilers for Parallel Computing*, C. Ding, J. Criswell, and P. Wu, Eds., Cham: Springer International Publishing, 2017, pp. 314–330, ISBN: 978-3-319-52709-3.

[8] D. Alistarh, G. Nadiradze, and N. Koval, "Efficiency guarantees for parallel incremental algorithms under relaxed schedulers," in *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '19, Phoenix,

AZ, USA: Association for Computing Machinery, 2019, pp. 145–154, ISBN: 9781450361842. DOI: 10.1145/3323165.3323201. [Online]. Available: https://doi.org/10.1145/3323165.3323201.

[9] H. Ortega-Arranz, D. R. Llanos, and A. Gonzalez-Escribano, *The Shortest-Path Problem: Analysis and Comparison of Methods* (Synthesis Lectures on Theoretical Computer Science), 1st ed. Springer Cham, 2015, pp. XV, 71, ISBN: 978-3-031-01446-8. DOI: 10.1007/978-3-031-02574-7. [Online]. Available: https://doi.org/10.1007/978-3-031-02574-7.

[10] D. Cederman, A. Gidenstam, P. H. Ha, H. Sundell, M. Papatriantafilou, and P. Tsigas, "Lock-free concurrent data structures," *CoRR*, vol. abs/1302.2757, 2013. arXiv: 1302.2757. [Online]. Available: http://arxiv.org/abs/1302.2757.

[11] M. D'Antonio, K. von Geijer, T. S. Mai, P. Tsigas, and H. Vandierendonck, "Relax and don't stop: Graph-aware asynchronous sssp," in *Proceedings of the 1st FastCode Programming Challenge*, ser. FCPC '25, The Westin Las Vegas Hotel & Spa, Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 43–47, ISBN: 9798400714467. DOI: 10.1145/3711708.3723446. [Online]. Available: https://doi.org/10.1145/3711708.3723446.

[12] S. Walzer and M. Williams, *A simple yet exact analysis of the multiqueue*, 2025. arXiv: 2410.08714 [cs.DS]. [Online]. Available: https://arxiv.org/abs/2410.08714.

[13] T. Friedrich, "From Graph Theory to Network Science: The Natural Emergence of Hyperbolicity," in *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*, R. Niedermeier and C. Paul, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 126, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 5:1–5:9, ISBN: 978-3-95977-100-9. DOI: 10.4230/LIPIcs.STACS.2019.5. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.STACS.2019.5.

[14] L. Dhulipala, G. Blelloch, and J. 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*, ser. SPAA '17, Washington, DC, USA: Association for Computing Machinery, 2017, pp. 293–304, ISBN: 9781450345934. DOI: 10.1145/3087556.3087580. [Online]. Available: https://doi.org/10.1145/3087556.3087580.

[15] U. Meyer and P. Sanders, "Δ-stepping: A parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003, 1998 European Symposium on Algorithms, ISSN: 0196-6774. DOI: https://doi.org/10.1016/S0196-6774(03)00076-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196677403000762.

[16] X. Dong, Y. Gu, Y. Sun, and Y. Zhang, "Efficient stepping algorithms and implementations for parallel shortest paths," in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 184–197, ISBN: 9781450380706. DOI: 10.1145/3409964.3461782. [Online]. Available: https://doi.org/10.1145/3409964.3461782.

[17] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 456–471, ISBN: 9781450323888. DOI: 10.1145/2517349.2522739. [Online]. Available: https://doi.org/10.1145/2517349.2522739.

[18] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996, ISSN: 0001-5903. DOI: 10.1007/s002360050048. [Online]. Available: https://doi.org/10.1007/s002360050048.

[19] H. Rihani, P. Sanders, and R. Dementiev, "Multiqueues: Simple relaxed concurrent priority queues," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '15, Portland, Oregon, USA: Association for Computing Machinery, 2015, pp. 80–82, ISBN: 9781450335881. DOI: 10.1145/2755573.2755616. [Online]. Available: https://doi.org/10.1145/2755573.2755616.

[20] A. Postnikova, N. Koval, G. Nadiradze, and D. Alistarh, "Multi-queues can be state-of-the-art priority schedulers," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22, Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 353–367, ISBN: 9781450392044. DOI: 10.1145/3503221.3508432. [Online]. Available: https://doi.org/10.1145/3503221.3508432.

[21] G. Zhang, G. Posluns, and M. C. Jeffrey, "Multi bucket queues: Efficient concurrent priority scheduling," in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '24, Nantes, France: Association for Computing Machinery, 2024, pp. 113–124, ISBN: 9798400704161. DOI: 10.1145/3626183.3659962. [Online]. Available: https://doi.org/10.1145/3626183.3659962.

[22] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford large network dataset collection*, http://snap.stanford.edu/data, Jun. 2014.

[23] M. D'Antonio, T. S. Mai, P. Tsigas, and H. Vandierendonck, "Wasp: Efficient asynchronous single-source shortest path on multicore systems via work stealing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '25, Washington, DC, USA: Association for Computing Machinery, 2025, pp. 2109–2125, ISBN: 9798400714665. DOI: 10.1145/3712285.3759872. [Online]. Available: https://doi.org/10.1145/3712285.3759872.

[24] J. Kunegis, "Konect: The koblenz network collection," in *Proceedings of the 22nd International Conference on World Wide Web*, ser. WWW '13 Companion, Rio de Janeiro, Brazil: Association for Computing Machinery, 2013, pp. 1343–1350, ISBN: 9781450320382. DOI: 10.1145/2487788.2488173. [Online]. Available: https://doi.org/10.1145/2487788.2488173.

[25] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011, ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. [Online]. Available: https://doi.org/10.1145/2049662.2049663.

[26] A. Haas, T. Hütter, C. M. Kirsch, M. Lippautz, M. Preishuber, and A. Sokolova, "Scal: A benchmarking suite for concurrent data structures," in *Networked Systems*, A. Bouajjani and H. Fauconnier, Eds., Cham: Springer International Publishing, 2015, pp. 1–14, ISBN: 978-3-319-26850-7.

[27] K. von Geijer, P. Tsigas, E. Johansson, and S. Hermansson, "Balanced allocations over efficient queues: A fast relaxed fifo queue," in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '25, Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 382–395, ISBN: 9798400714436. DOI: 10.1145/3710848.3710892. [Online]. Available: https://doi.org/10.1145/3710848.3710892.

[28] S. Beamer, K. Asanović, and D. Patterson, *The gap benchmark suite*, 2017. arXiv: 1508.03619 [cs.DC]. [Online]. Available: https://arxiv.org/abs/1508.03619.

[29] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," *SIGPLAN Not.*, vol. 42, no. 6, pp. 211–222, Jun. 2007, ISSN: 0362-1340. DOI: 10.1145/1273442.1250759. [Online]. Available: https://doi.org/10.1145/1273442.1250759.

[30] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," in *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[31] X. Dong, Y. Gu, Y. Sun, and L. Wang, "Brief announcement: Pasgal: Parallel and scalable graph algorithm library," in *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2024.

[32] D. Funke et al., "Communication-free massively distributed graph generation," *Journal of Parallel and Distributed Computing*, vol. 131, pp. 200–217, 2019, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2019.03.011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731518304684.