Progress-Based Distributed Queues

Exploring the effects of a novel heuristic with partial queues using fetch-and-add

Master's thesis in Computer science and engineering

Sebastian Hermansson

Elias Johansson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Progress-Based Distributed Queues

Exploring the effects of a novel heuristic with partial queues using fetch-and-add

Sebastian Hermansson

Elias Johansson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Progress-Based Distributed Queue
Exploring the effects of a novel heuristic with partial queues using fetch-and-add
Sebastian Hermansson Elias Johansson

© Sebastian Hermansson, Elias Johansson, 2024.

Supervisor: Philippas Tsigas, Department of Computer Science and Engineering

Co-Supervisor: Kåre von Geijer, Department of Computer Science and Engineering

Examiner: Carl-Johan Seger, Department of Computer Science and Engineering

Master's Thesis 2024
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 2024

Progress-Based Distributed Queue
Exploring the effects of a novel heuristic with partial queues using fetch-and-add
Sebastian Hermansson, Elias Johansson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

With the increasing use of multi-threaded processors comes the challenge of scaling with a higher number of threads. To that end, the semantics of a data structure can be relaxed to lessen contention and achieve better performance. The data structure of interest in this thesis is a relaxed first-in-first-out queue called d-RA, which is made up of partial queues. It utilizes a length-based heuristic, that chooses a queue to perform an operation on. However, it employs relatively slow lock-free partial queues and uses a heuristic that is not definitively proven to be optimal. Furthermore, relaxed first-in-first-out queues lack real-world applications as they can not be used if the order of operations is strict.

This thesis improves on the d-RA algorithm by using faster partial queues and a new progress-based heuristic, which generally increases the data structure's throughput, causes it to scale with more threads, and lowers the level of relaxation. Additionally, we use relaxed queues in an unordered breadth-first-search to calculate the shortest paths in graphs where they are shown to outperform concurrent queues.

Keywords: relaxed first-in-first-out queue, relaxed data structure, lock-free, linearizable.

Acknowledgements

Firstly, we want to thank our supervisor, Philippas Tsigas, for allowing us to take on this unique and challenging project. We appreciate the opportunity to advance our understanding and discover previously foreign concepts. Moreover, we thank his co-supervisor, Kåre von Geijer for being with us every step of the way in this project. His enthusiasm, insights, and advice have been paramount to this thesis. Finally, we give our thanks to our friends and families for their tremendous support in our endeavors.

Sebastian Hermansson and Elias Johansson, Gothenburg, 2024-06-19

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

2 Theory **3**

2.1 Data Structures 3

2.1.1 First-In-First-Out Queues 3

2.1.2 Pool 3

2.2 Concurrent FIFO Queues 3

2.2.1 Atomic Processor Instructions 4

2.2.2 Contention in the Queue 4

2.2.3 Previous Work 4

2.2.4 Michael-Scott Queue 5

2.2.5 Fetch-and-Add Array Queue 6

2.2.6 LCRQ 7

2.3 Relaxation and its Implications 9

2.3.1 Relaxation 9

2.3.2 Linearizability 10

2.3.3 Measuring Relaxation 11

2.4 Distributed Relaxed Queues 11

2.4.1 d-RA 11

2.4.2 2Dd-Queue 13

2.5 Other related work 13

2.5.1 Bag 13

2.5.2 RCQS and RCQD 14

2.6 Concurrency Management in Data Structures 14

2.6.1 Memory Hierarchy 14

2.6.2 Locality 15

2.6.3 Thread Pinning 15

2.6.4 False Sharing 15

2.6.5 Memory Reclamation 15

2.6.6 ABA-problem 16

2.7 Applications for Relaxed Queues 16

2.7.1 Breadth-First Search . . . 16
2.7.2 Unordered Breadth-First Search . . . 16

3 Methods **19**
3.1 Implementation of data structures . . . 19
3.1.1 Scal . . . 19
3.1.2 Preexisting Implementations of Data Structures . . . 19
3.1.3 RCQS and RCQD . . . 20
3.1.4 Length Approximation for d-RA . . . 20
3.1.5 d-RA Emptiness Checks . . . 21
3.1.6 Progress-Based Heuristic . . . 22
3.2 Correctness . . . 22
3.2.1 Lock-Freedom . . . 22
3.2.2 Linearizability . . . 23
3.3 Applications . . . 24
3.3.1 A Breadth-First Search Algorithm With Errors . . . 24
3.3.2 An Unordered Breadth-First Search algorithm . . . 25
3.3.3 Graphs . . . 26
3.4 Benchmarks . . . 26
3.4.1 Calculating Relaxation Distance . . . 27
3.4.2 Configuration of Data Structures . . . 27
3.5 Benchmark Environment . . . 28

4 Results and Discussion **29**
4.1 Results of New Partial Queues . . . 29
4.1.1 Performance of the Producer-Consumer Benchmarks . . . 29
4.1.2 Relaxation in Producer-Consumer Benchmarks . . . 31
4.1.3 Throughput in Sequential Alternating Benchmarks . . . 32
4.1.4 Relaxation in Sequential Alternating Benchmarks . . . 36
4.1.5 Results of Progress-Based Heuristic . . . 39
4.2 Shortest Path Calculations . . . 43
4.2.1 Errors Allowed . . . 43
4.2.2 Unordered BFS . . . 47
4.3 General Discussion . . . 51
4.3.1 Feasibility of Different Partial Queues . . . 51
4.3.2 Feasibility of Relaxed Queues . . . 51
4.3.3 Sources of Error . . . 52
4.3.4 Research Questions . . . 53

5 Conclusion **55**
5.1 Future Work . . . 55

Bibliography **59**

A Appendix 1 **I**
A.1 Complete graphs . . . I
A.2 Supplementary graphs . . . IV

List of Figures

2.1 The design of the Michael Scott queue. 5
2.2 The design of the Fetch-and-Add Array queue 7
2.3 The design of LCRQ 8
2.4 A state description of relaxation 10
2.5 d-RA Heuristic (choice of two) 13

4.1 Throughput results of the producer-consumer benchmark. Higher is better. 30
4.2 Relaxation results of the producer-consumer benchmark. Lower is better. 32
4.3 Throughput results of the sequential alternating benchmark with 0% prefill. Higher is better. 33
4.4 Throughput results of the sequential alternating benchmark with 1% prefill. Higher is better. 34
4.5 Throughput results of the sequential alternating benchmark with 100% prefill. Higher is better. 35
4.6 Relaxation results of the sequential alternating benchmark with 0% prefill. Lower is better. 37
4.7 Relaxation results of the sequential alternating benchmark with 1% prefill. Lower is better. 38
4.8 Relaxation results of the sequential alternating benchmark with 100% prefill. Lower is better. 39
4.9 Relaxation results in sequential alternating with respect to partial queues. 50% prefill, 20000 total elements. One working thread. Lower is better. 41
4.10 Throughput and relaxation of a producer-consumer benchmark comparing the progress-based d-RA FAAAQ with 20 partial threads against d-RA Michael-Scott versions 42
4.11 The elapsed time and average path length of the different data structures on cage15 when errors are allowed. 45
4.12 The elapsed time and the average path length of the different data structures on Sparse 200M when errors are allowed. 46
4.13 The elapsed time and total work, in relation to the baseline, of the different data structures on cage15 during unordered BFS. 49
4.14 The elapsed time and total work of the different data structures on sparse200M during unordered BFS. 50

A.1 Relaxation results of the producer-consumer benchmark. Lower is better. I
A.2 Throughput results of the sequential alternating benchmark with 0% prefill. Higher is better. II
A.3 Throughput results of the sequential alternating benchmark with 1% prefill. Higher is better. II
A.4 Relaxation results of the sequential alternating benchmark with 100% prefill. Lower is better. III
A.5 The elapsed time and average path length of the different data structures on audikw_1 when errors are allowed. V
A.6 The elapsed time and average path length of the different data structures on sparse10M when errors are allowed. VI
A.7 The elapsed time and average path length of the different data structures on sparse50M when errors are allowed. VII
A.8 The elapsed time and total work of the different data structures on audikw_1 during unordered BFS. VIII
A.9 The elapsed time and total work of the different data structures on sparse10M during unordered BFS. IX
A.10 The elapsed time and total work of the different data structures on sparse50M during unordered BFS. X
A.11 Throughput of producer-consumer benchmark with one producer and a varying number of consumers. Higher is better. XI

List of Tables

3.1 The graphs used for benchmarking 26

4.1 The total number of lines of code, excluding comments and blank spaces, of the different versions of d-RA. 51

1 Introduction

With the increasing need to process substantial amounts of data, software has to be more reliant on parallelism. As such, data structures need to facilitate concurrency and parallelism. The conventional way to enable concurrent operations on data structures is to limit concurrent accesses to eliminate race conditions. The problem with this is that a sequential bottleneck can occur when several processes are trying to access the data structure at the same time. This, in turn, can lead to performance issues in regards to scalability, whenever the degree of parallelism is increased [1].

A solution is to relax the semantics of the data structure to increase performance [2]. In other words, the standard behavior of the data structure can sometimes be deviated from. In our case, we will implement and design a relaxed first-in-first-out (FIFO) queue. This means that it is not guaranteed that the oldest element will be returned after a dequeue operation. This relaxation leads to lower levels of contention on the queue's head and tail, allowing it to scale better with more threads.

There has been previous research dedicated to relaxed FIFO queues. The paper: *Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation* [1], introduced a new family of queues called *Distributed queues*. One of these queues is called *d-RA*. Distributed in this case means that the queue consists of several *partial queues* that each individual thread operates on. Which one of these partial queues that a thread gets access to is partially determined randomly and partially by heuristics. The balancer used, which the queue is named after, was introduced in an earlier paper [3]. However, hereafter when used, d-RA will refer to the distributed queue.

The fundamental idea behind this project is that it is believed that the d-RA algorithm could be improved. This is because the partial queues used with d-RA are Michael-Scott queues [4], for which there exist faster alternatives. The paper: *Fast Concurrent Queues for X86 Processors* [5], introduced a new type of concurrent queue for x86 processor architectures called LCRQ, that uses the atomic processor instructions *fetch-and-add* and *double-width-compare-and-swap*. The issue is that this queue will not work on most commercial multicore architectures [5]. However, there exist other fast queues that do not have this limitation, for example, the Fetch-and-Add Array Queue (FAAAQ) [6].

Also, it is unclear whether d-RA's method of selecting which partial queue to perform

an operation on, based on their lengths at the time, is optimal. Therefore, we investigate a *progress-based* heuristic, that instead of calculating the length, takes into account the number of operations done to it.

Moreover, there are not many known applications of relaxed FIFO queues and in many cases, they are used as a pool. They have been used in benchmarks related to breadth-first search [1] (BFS) but it is unclear how feasible they are to use in this manner. This is because the relaxation might cause the algorithm to pick nodes in a less-than-optimal way. In [7] *Unordered breadth-first search* was used, where the algorithm's execution is dependent on scheduling and any possible errors are eliminated after the fact by potentially visiting nodes multiple times. To our knowledge, no work has been done to analyze potential errors in the area of relaxed FIFO-queues. Therefore it is of interest to study if the potential speedup of relaxation can outweigh the costs associated with it and to determine whether or not a relaxed FIFO queue is suitable to use for unordered BFS.

In summary, the following research questions will be answered in this thesis:

- *Research-question 1* How does the d-RA algorithm scale with faster partial queues?
- *Research-question 2* How does a progress-based heuristic affect the d-RA algorithm?
- *Research-question 3* Is unordered breadth-first search a feasible application for a relaxed FIFO queue?

2

Theory

2.1 Data Structures

In order to understand our work it is important to properly specify the behavior of the two data structures, FIFO queues, and pools, that we implement and utilize.

2.1.1 First-In-First-Out Queues

A FIFO queue is a data structure that in its most trivial form supports two operations: *dequeue*, which removes the oldest *element* in the queue, and *enqueue*, which adds an element to the queue. The queue is usually implemented as some form of dynamic list, where the oldest element is stored at the front of it and the most recently added element is stored at the end. These are referred to as the head and the tail of the queue respectively.

2.1.2 Pool

A pool is a data structure that supports the operations *get*, which retrieves an element from the data structure, and *put*, which inserts an element into the data structure. Unlike a queue, a pool does not guarantee any specific ordering of elements. Its function is to store elements that are put in and provide elements when requested, without any requirement for any element to be evicted at any specified point relative to another.

2.2 Concurrent FIFO Queues

A concurrent data structure such as a FIFO queue, is designed to allow multiple threads to perform operations on it while maintaining its specified semantics. In order for it to function as intended, it needs to handle several threads accessing it simultaneously without the risk of concurrency-related issues. There are two main types of algorithms that handle this: *blocking* and *non-blocking*. The difference between these two is that a blocking algorithm makes no guarantee against deadlocks [4]. There are different types of non-blocking algorithms but those that are *lock-free* are specifically relevant for this work. A lock-free algorithm guarantees that some given thread operating on the data structure will make progress during any point in time [8]. The queues designed in this thesis are lock-free and utilize atomic processor

instructions to perform operations on elements, which means they do not require the use of locks, and hence, avoid lock-related deadlocks. However, avoiding the use of locks does not guarantee lock freedom, as will be discussed later on.

2.2.1 Atomic Processor Instructions

Atomic processor instructions perform operations on data atomically, without a thread needing to acquire a lock. This means that if a thread fails or stalls, there is no way for it to hold a lock that may later be required by another thread, thereby stopping it from progressing [9]. Atomic processor instructions influence data atomically on a hardware level. When the instruction begins, it acquires a global memory lock which stops other threads from accessing the specified address in memory. It only holds this lock for the duration of the instruction [10]. The following are atomic instructions used in concurrent data structures to ensure correct behavior.

- ***Fetch-and-Add*** **(FAA)**: Returns the value at an address in memory and atomically adds a specified number to the value at that address [5].
- ***Compare-and-Swap (CAS)***: Updates the value at a specified memory address if that value is the same as a provided one [5]. If this condition is satisfied, the value is exchanged.
- ***Double-Width-Compare-and-Swap*** **(CAS2)**: Compares two adjacent values in memory to two provided values and updates them to two other provided ones, if the comparison is satisfied [5]. This thesis uses a specified macro written by [11] to invoke the instruction *lock cmpxchg16b* [12]. It requires the two inputs to be a pointer and a value respectively.

2.2.2 Contention in the Queue

As mentioned, a concurrent queue allows multiple threads to access it simultaneously while maintaining the correct behavior. When several threads attempt to gain access to the same element in the queue, they effectively contend for that privilege. In this thesis, this will henceforth be referred to as contention. A high level of contention means that many threads are accessing the same shared variable. Ensuring that contention is put on an operation that will not fail is preferable as it would waste less time. This will be exemplified by the preference for putting contention on the FAA instruction, which always succeeds in adding to a value, as opposed to the CAS instruction, which can fail [5].

2.2.3 Previous Work

The first lock-free concurrent queue is the previously mentioned Michael-Scott queue. Its design is simple and will be explained in detail in Section 2.2.4. After this several other more performant lock-free FIFO queues have been introduced. One intermediate step between the Michael-Scott queue and the LCRQ is constituted by [13], introduced by Tsigas and Zhang, that utilizes a cyclic array with the limitation that the queue is bounded in size. The Baskets Queue [14] is reminiscent of the

Michael-Scott queue but threads enqueue into baskets decided by a time interval which allows parallel enqueues. A cache-aware queue was introduced in [15] where the threads try to work thread locally by storing pointers and indexes to avoid reading the shared state when it is not necessary. The LCRQ [5], which uses cyclic arrays, was to our knowledge the first FIFO queue to utilize FAA to distribute work and thereby lower contention. It also as opposed to [13], is not bounded and will be discussed in Section 2.2.6. FAAAQ [6] is a dynamically sized queue that uses FAA but is not cyclic and does not need the CAS2 instruction, unlike LCRQ. The details of this queue are discussed in Section 2.2.5. A very similar queue to FAAAQ is the Wait-free queue [16] with the difference that it has a stronger progress guarantee in the form of being wait-free, where the progress of individual threads can be guaranteed. Finally, an improvement was found for LCRQ in the queue LPRQ [17] which is cyclic and unbounded without using CAS2.

2.2.4 Michael-Scott Queue

The Michael-Scott FIFO queue [4] consists of a linked list of nodes containing the values of the enqueued elements as well as the pointer to the next node in the queue. For an illustration see Figure 2.1. As can be seen, in State 1 there are three elements in the queue. In State 2, an element is inserted into the queue, a new node is created and added to the linked list and it becomes the new tail. In State 3, there is a dequeue operation, the node that corresponds to the head is removed, and the next node becomes the head. To facilitate this, the atomic processor instruction CAS is utilized. The queue has been criticized because of the problems of its scalability related to the use of CAS [5]. In short, multiple threads can attempt to enqueue and dequeue an element at the same place in the queue using CAS, which results in contention being at this instruction. All but one will fail at a time; these failures waste many cycles resulting in poor performance as contention increases.

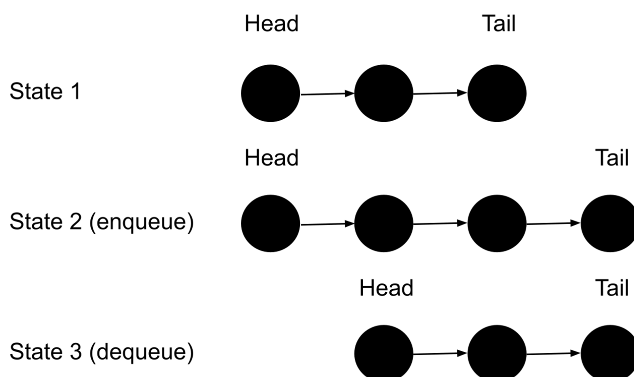


Figure 2.1: The design of the Michael Scott queue.

2.2.5 Fetch-and-Add Array Queue

One of the partial queues used in this thesis is FAAAQ [6]. A similar queue has been proposed elsewhere [16]. An illustration of this queue can be seen in Figure 2.2. This queue is made up of smaller queues that are created once the previous one has been filled. This paper will refer to these as *segments*, and each one holds an array of elements, indexes for enqueueing and dequeuing, and a pointer to the next segment. The illustration shows four different states. In State 1 there is one segment, containing three elements. In State 2, an element is enqueued, causing there to be four filled spots in the array of the segment. In State 3, there is another enqueue, however, since the tail has reached the end of the previous segment's array, a new one is created. In State 4, there has been another enqueue and four dequeues. This has moved the tail forward in the second segment, and removed all four elements from the first one, consequently making the first element in the second segment the head, and removing the first one entirely as it has been emptied.

As was mentioned, each segment contains two indexes, one for enqueueing and one for dequeueing. The queue consists of multiple linked segments, meaning that the actual head of the queue is located at the dequeue index of the first segment, or head segment. Similarly, the actual tail is located at the enqueue index of the last added segment, or the tail segment. As an example, Figure 2.2 shows that, in between States 2 and 3, the tail moves from the first segment to the second segment. This means that the first segment goes from being both the head and tail segment to only being the head segment, while the second one becomes the tail segment. Furthermore, State 4 shows when the first segment has been emptied and is removed. The head segment then becomes the next one and once again the head and tail segments are one and the same.

In the FAAAQ, access to the positions in the array of a segment is distributed through the use of the atomic instruction FAA. Consequently, each enqueue operation acquires a unique tail index, and each dequeue operation acquires a unique head index. This ensures that only one enqueuer and one dequeuer ever have access to the same position in the array. In contrast to the Michael Scott queue, CAS failures on a specific index in the array can only occur if an enqueuer is preempted by a dequeuer with the same index. This significantly reduces CAS failures, which generally results in faster performance [5].

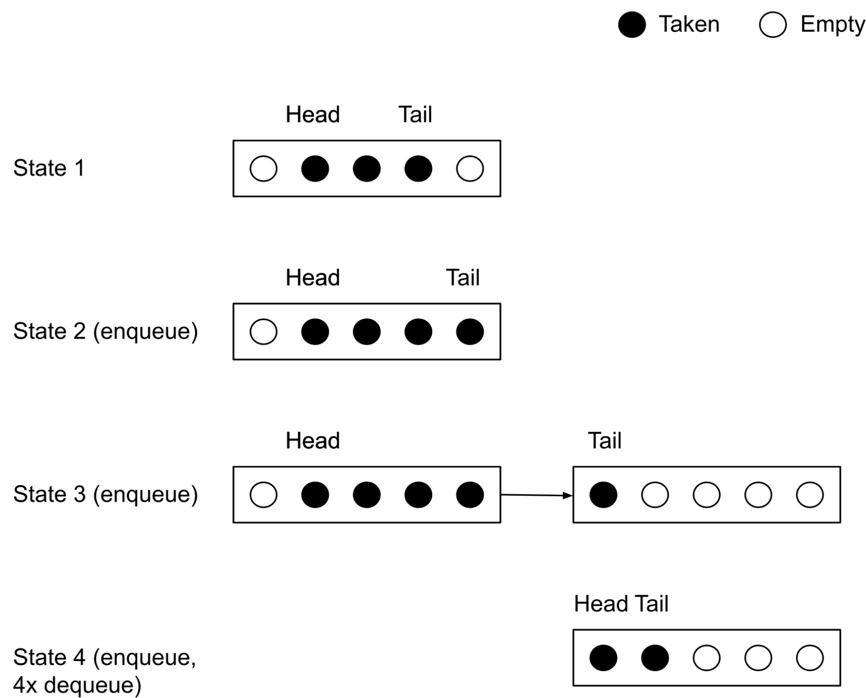


Figure 2.2: The design of the Fetch-and-Add Array queue

2.2.6 LCRQ

The other type of partial queue is the LCRQ [5]. It is made up of concurrent ring queues (CRQs) that behave similarly to the segments in FAAAQ, and will also be referred to as segments. They are cyclical, meaning that enqueuers and dequeuers can operate in the array and wrap around to the beginning once they pass the end. This ensures that memory is not wasted as segments are repeatedly reused. New segments are only created in two circumstances, either when the current one has been filled up or if enqueues fail repeatedly. As can be seen in Figure 2.3, State 1 shows a segment with 4 elements. In State 2, there is an enqueue, which causes the tail to wrap around the array to place the element at the start. In State 3 there is another enqueue, which results in a second segment being created, as the tail catches up to the head, meaning that the first CRQ is full. In State 4, there is a single dequeue operation that removes an element from the first segment and moves the head forward.

Similarly to the FAAAQ, each segment holds two index values that point to the cells where the next dequeue and enqueue operations will be performed. They point to the local head and tail elements of each segment. The actual head of the queue is at the dequeue index of the head segment and the actual tail is at the enqueue index of the tail segment.

Furthermore, LCRQ uses the FAA instruction to acquire and update enqueue and dequeue indexes. Once one has been acquired, the operation attempts to alter the element inside the cell the index points at. If this is successful, the dequeuer uses a double-width-compare-and-swap operation to place the value and the enqueue index in the cell, however, if it fails a new index will be acquired. Because access to cells is acquired through the FAA instruction, contention is mainly on the enqueue and dequeue indexes, which generally results in fewer CAS failures than the Michael Scott queue [5]. It is important to mention that if dequeuers pass enqueuers they will mark the empty cells they encounter, causing subsequent enqueuers to not place elements there and be forced to acquire new enqueue indexes. A cell will also be marked if a dequeuer encounters an occupied cell that holds the index from a previous cycle of the CRQ. If the head surpasses the tail, the tail will be moved forward to decrease the number of failed enqueue operations.

Moreover, the LCRQ uses the atomic CAS2 instruction. However, CAS2 can not be used on many architectures or in languages, such as Java [17]. It is worth mentioning that there exists a paper about a version of LCRQ that does not use double-width-compare-and-swap [17].

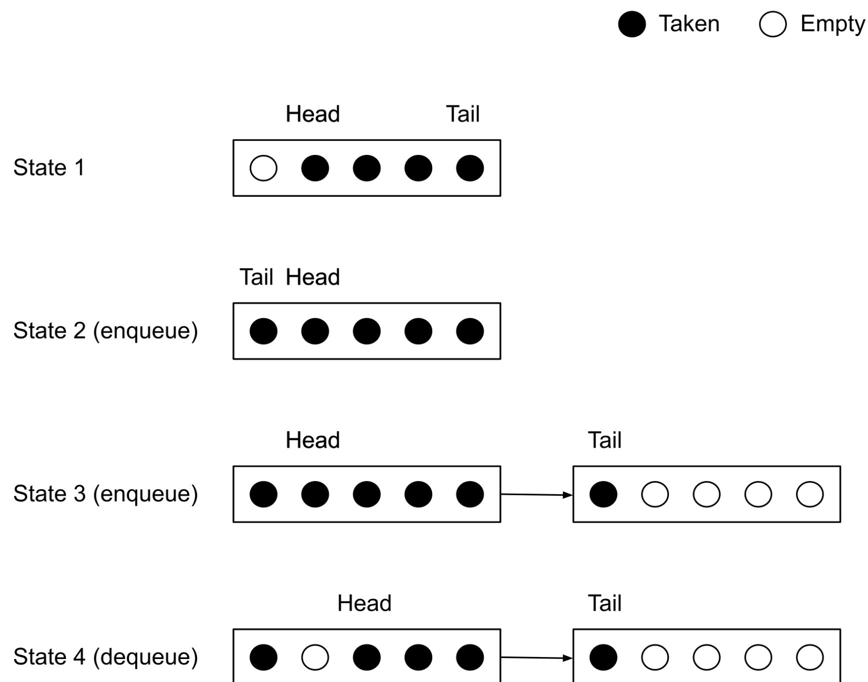


Figure 2.3: The design of LCRQ

2.3 Relaxation and its Implications

As relaxation is a core concept in this thesis, it and other related terms will be explained here.

2.3.1 Relaxation

Relaxation allows the specified constraints of a data structure to be loosened. In the context of this thesis, we will use the concept of relaxation to mean out-of-order relaxation. This means that the relaxation is with respect to the order of the operations done on a data structure, at some point in time during the execution of the operation. This means that the relaxation does not necessarily exclusively rely on the real order of the elements that reside in the data structure but rather at some conceptual points where these operations can be regarded as completed, see Section 2.3.2. Also important to mention, is that the relaxation depends on the state of the data structure at the point of consideration and not simply the ordering of operations.

In the context of a FIFO queue, this means that the oldest element in the queue will not always be the first to be returned. In Figure 2.4 we can see three different states of a FIFO queue. In state 1 it is full, in state 2, the third oldest element has been dequeued, and finally in state 3, once again the third oldest element is removed.

It is also important to mention that there are also other types of relaxation. For example, [18] uses relaxation in terms of multiplicity which means that an item can be returned more than once from a dequeue operation.

The reasoning behind relaxation is that it can improve the performance of data structures as it can decrease contention. However, this comes at a cost associated with the weakened semantics. Generally, relaxation is seen as something that should be minimized as it limits the usability of the data structure. For example, an application of a FIFO queue could be message passing but that would be near impossible with a queue with even the smallest degree of relaxation. This is because in many cases the order of the messages is extremely important.

Because of this, it is of interest to prove the upper bound of relaxation. However, the method of doing this depends on how relaxation is being employed. It can either be bound meaning that the data structure can not be more than *k out of order* [2] or there can be a stochastic element to the relaxation. An example of this random selection of elements would be *d-RA* [1]. Whenever randomness is being employed only probabilistic guarantees can be made in regards to the bound as it is technically possible to always pick the latest added element for a dequeue operation.

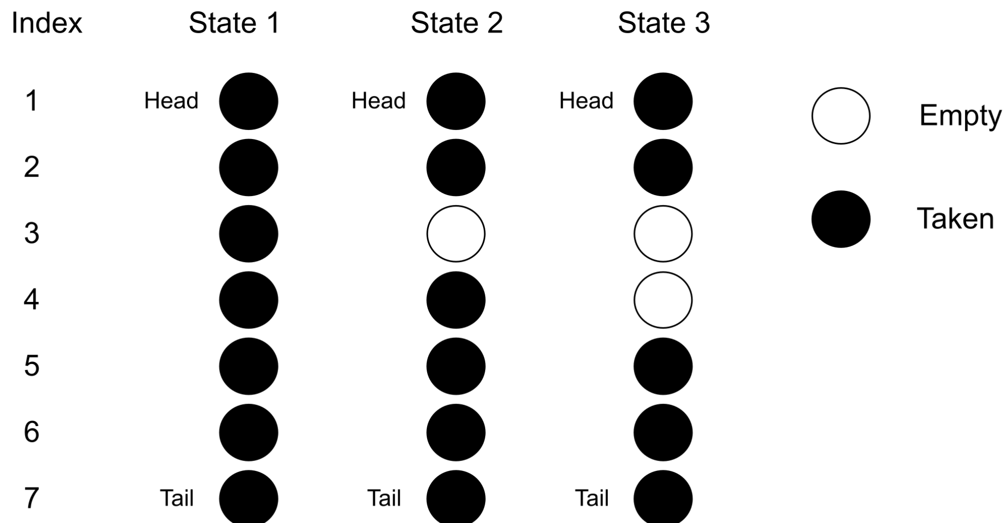


Figure 2.4: A state description of relaxation

2.3.2 Linearizability

Linearizability is a correctness property that can be used to verify the behavior of concurrent data structures [19]. To understand this concept, it is important to define what a *history* is. A history is the operations done on a data structure in the order that they were executed. In simple terms, a concurrent data structure is linearizable if the history of the data structure is identical to a sequential variant of the data structure [19]. Here is a valid history for a FIFO queue, where the oldest element is returned after the dequeue:

```
Enqueue(A)
Enqueue(B)
A <- Dequeue()
```

Here is an invalid history as A has not been added to the queue:

```
Enqueue(B)
A <- Dequeue()
```

Important to mention is that linearizability is always in relation to some specified behavior. A relaxed FIFO queue is not linearizable with respect to an ordinary FIFO queue. See the following example:

```
Enqueue(A)
Enqueue(B)

B <- Dequeue()
A <- Dequeue()
```

This behavior is acceptable for a relaxed FIFO queue but not for an ordinary one.

Therefore, a relaxed FIFO queue is only linearizable with respect to a pool.

2.3.3 Measuring Relaxation

In order to measure relaxation a history of the executed program needs to be stored. There are different ways to accomplish this. One way is to create timestamps of the points of linearization. These points will only be approximations as it is not possible to accurately log the exact time of completion of an operation. After this is done, the history can be evaluated to calculate the average *relaxation distance* like it was done in [1]. The relaxation distance is defined as the difference between the order of operations in a purely sequential data structure compared to its relaxed counterpart. For example in the previously seen example:

```
Enqueue(A)
Enqueue(B)

B <- Dequeue()
A <- Dequeue()
```

The average relaxation distance would be 0.5. The first dequeue (yielding B) would result in a relaxation distance of 1 as it is out of order and the second dequeue (yielding A) would not be out of order as it is the only element left in the queue.

2.4 Distributed Relaxed Queues

The main idea of using distributed queues is that the contention on one partial queue decreases whenever there are more queues to pick from. The downside of this behavior is the introduction of relaxation. Here we will present two different distributed queue algorithms that work in different manners.

2.4.1 d-RA

d-RA is a distributed queue algorithm consisting of multiple partial queues. The algorithm decides which partial queue to interact with by randomly selecting d partial queues, and then uses the heuristic to choose which of these to perform the operation on. The heuristic that is described in [1] involves approximating the length of the underlying data structures in order to pick the most optimal queue. The pseudocode for the algorithm can be found in Listing 2.1.

For the enqueue operation the returned partial queue with the fewest number of elements in it is selected. In the case of dequeue, the partial queue with the most number of elements is chosen as a starting index. The algorithm will then start from that index and check if that partial queue is empty. If it is not, then the element will be dequeued from that partial queue. If the partial queue is empty, then the index will be incremented and another queue will be checked. Figure 2.5 illustrates this heuristic, where two partial queues have been randomly chosen. If the operation is

an enqueue, it will choose to enqueue to queue 1 as it is the shortest. If the operation is a dequeue, it will choose queue 2 as it is the longest.

One specific scenario needs to be accounted for by the algorithm to guarantee linearizability. A partial queue may be empty when a dequeuer tries to remove an element which causes the thread to move to the next partial queue. If then an enqueuer adds an element to the previous partial queue, the dequeuer will return signifying that the distributed data structure is empty even if it is not. Therefore the current tails of all the partial queues will be stored in order to see if an enqueue has occurred after the partial queue has been read as empty. If all partial queues are read as empty, their tails will be compared against their old tails. If any changes are detected, the dequeuing process will restart with the starting index of the changed partial queue.

Listing 2.1: Pseudo code for d-RA

```
bool enqueue(item){
    index = get_index();
    return queues[index]->enqueue(item)
}
bool dequeue(*item) {
    index = get_index();
    while (true) {
        //Initial pass. p = number of partial queues
        for (i = index; i < p + i; i++ ){
            if(queues[i % p]->dequeue(item, tails))
                return true;
        }

        //Check if tail has changed
        for (i = index; i < p + i; i++ ){
            if (tails[i] != queues[index % p]-> get_tail()){
                index = i;
                break;
            }
            else if (((i + 1) % p) == index) return false;
        }
    }
}
```

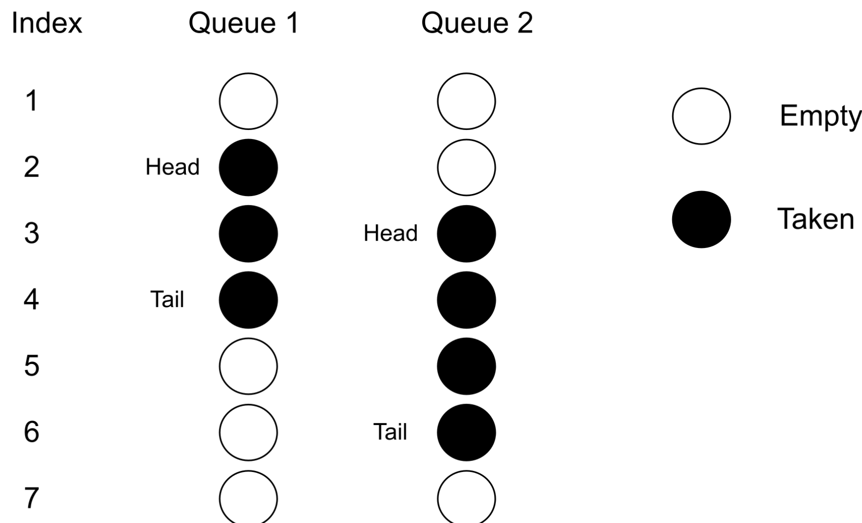


Figure 2.5: d-RA Heuristic (choice of two)

2.4.2 2Dd-Queue

The relaxed queue in [20] works a bit differently from the previously described queues. It is reminiscent of d-RA in the sense that it is a distributed data structure but it also has the notion of windows. What this means is that every partial queue, or *sub-structure*, can only be accessed a set amount of times before another sub-structure is tried. The benefit of this approach is that it improves locality as a thread will only exchange the sub-structure whenever the maximum number of operations done to it is reached. This version of the queue has separate windows for enqueue and dequeue operations. Furthermore, a hard bound on the relaxation can also be had from this algorithm. When the number of operations of a sub-structure is reached, the thread will look for other sub-structures. If no available queues can be found then the window is moved to allow for more operations.

2.5 Other related work

Here other related data structures that also will be used in the benchmarks will be presented.

2.5.1 Bag

A fast pool, called Bag, was developed by [8] and it utilizes the fact that in contrast to queues, the ordering of the elements is not important for pools. This is achieved by having each thread work on a thread-local stack, designed as a linked list of what are called *blocks*. Whenever a new item is added to the Bag, a new block is created. When an item is removed the thread will first try to remove an item from its own linked list. If it is unable to do so, it will try to steal an item from a linked list

that is worked on by another thread. This is one of the pools that was used in the benchmarking of [1].

2.5.2 RCQS and RCQD

Relaxed Concurrent Queue Single- and Double (RCQS/RCQD) are cyclic queues similar to LCRQ, with the difference that they are relaxed, and do not link multiple CRQs together [21]. In RCQS and RCQD, enqueues and dequeues acquire the index for tail and head respectively, and attempt to operate on the corresponding element. If an enqueue operation reaches an occupied slot, it will wait there until the element has been dequeued. If a dequeue operation reaches a free slot it will wait until an element is placed there. What makes this design relaxed is that an enqueue operation that acquires the index of a slot is not guaranteed to be the one to place an element there before another enqueuer. If enqueues wrap around the RCQ, another one can acquire the same index and manage to place an element in the slot first [21]. This would mean that the semantics of FIFO are not adhered to. The same holds true for dequeuers. The difference between RCQS and RCQD is in the handling of a slot's state. The former uses the most significant bit of the data field to tell if it is free or occupied, while the latter uses a separate state variable. This means that RCQS can be implemented using CAS, while RCQD requires CAS2 [21].

2.6 Concurrency Management in Data Structures

As previously mentioned, there are two kinds of algorithms used to design concurrent data structures. blocking and non-blocking. A blocking algorithm commonly locks variables, objects, and critical sections of a program. Consequently, multiple threads can not influence the same element at the same time, and thus, it will not be subject to race conditions. However, the traditional method of using locks to ensure correctness can cause deadlocks, where a thread that acquires a lock may hinder the progress of other threads if the process fails or stalls [9]. Alternatively, there are non-blocking algorithms, which use atomic processor instructions.

2.6.1 Memory Hierarchy

Modern processors use a memory hierarchy consisting of main memory and L3, L2, and L1 caches, where each subsequent level is faster and smaller than the previous. This memory hierarchy is used to increase the speed with which information can be fetched for use by the processor. When data is requested it is fetched as a line consisting of many bytes. It is placed in the level with the shortest access time (L1) and evicted from a cache level if another line takes its place [22]. Fetching in lines can improve performance according to the principles of spatial and temporal locality. These principles state that addresses close together in proximity are likely to be accessed shortly after one another, and an address is likely to be accessed again in the near future if it has been accessed recently [22]. By fetching a line, multiple addresses that are likely to be relevant later are all placed in the cache for

quick access. By retaining recently accessed data in a faster cache level, it remains poised for future utilization, aligning with the likelihood of its imminent retrieval.

2.6.2 Locality

Cache locality can be used to increase the speed with which operations can be performed due to faster access to imperative information for the processor. However, different cores have access to different caches. Commonly, each core has its own L1 cache, hence, to adhere to the principle of spatial locality when working on a data structure it is useful for a thread to do so on the same part continuously. In the case of distributed data structures, this would mean a thread repeatedly and deliberately elects to access the same parts of it instead of always choosing among all sub-structures [20]. This would increase the likelihood of data being accessible from the cache due to spatial locality, whereas it would be much less likely if the choice for distribution allows for other partials and, consequently, other parts of memory having to be fetched from.

2.6.3 Thread Pinning

To be able to leverage the performance improvements that are provided by locality, it is important that threads continuously execute on the same cores in the processor [23]. Additionally, threads should be assigned to separate logical cores to avoid a scenario where the scheduler has to perform context switching, which will temporarily hinder the progression of one thread in place of another. To ensure this, threads may be pinned to specific cores.

2.6.4 False Sharing

To optimize a data structure, certain knowledge regarding how processors execute instructions at a lower level is needed. One such optimization is called *cache line padding*. This is achieved through aligning objects and pointers within a data structure to start at the beginning of a cache line and padding the bytes left over on the line. Cache line padding increases memory consumption but reduces *false sharing*. False sharing occurs when a thread invalidates a cache line by modifying bytes unrelated to the element of interest [24]. False sharing can lead to increased overhead associated with cache coherency [25].

2.6.5 Memory Reclamation

For a data structure to be usable it is important that the memory that it accesses can be reclaimed after it is no longer in use [26]. Otherwise, the use of the data structure could in the worst case lead to a crash caused by a shortage of memory. It is therefore important that such functionality is in place for a data structure to be practically viable. This is not a trivial problem in terms of parallel programs as a thread may be referencing parts of memory that another thread is in the process of reallocating. A suitable solution is to use *Hazard Pointers* [27]. The accessed pointers are placed inside an array to track them, thereby, ensuring they are not

reclaimed whenever some other threads reference them. Whenever no thread uses a specific pointer, it can be safely reclaimed.

2.6.6 ABA-problem

The ABA-problem can occur in lock-free data structures [27]. The problem can occur when a thread reads a certain value that is then changed by other threads to first a new value, and then back to the original value again. The thread which first reads the value assumes that the value has not changed. This can be problematic and cause erroneous behavior. In the context of memory reclamation, this can be solved by using hazard pointers [27] or made less likely to occur by using versioning to tag pointers [28].

2.7 Applications for Relaxed Queues

Previous work has been done on relaxed priority queues and there are several applications for them. For example, a relaxed priority queue can be used to find the shortest path or a minimum spanning tree in a weighted graph [29]. In the case of relaxed FIFO queues their areas of applications are not entirely clear. It is possible to use them in applications where the order of the returned elements is of no importance but in those cases, the data structure would merely act as a pool with some form of ordering.

2.7.1 Breadth-First Search

Breadth-first search (BFS) is an algorithm that finds the shortest path in an unweighted graph. The essence of BFS is that it searches from a starting point outward in layers, where all nodes in any one layer (n) are one step further away from the start than the previous layer (n-1). Hence, a node's distance from the start is one step further than its parent. The search continues through each subsequent layer until all nodes have been visited. A BFS algorithm can be implemented using a FIFO queue. By adding neighbors of nodes to the queue, and always acquiring the next node to access from the queue, it will provide the properties of the layered search [30]. In an unweighted graph, BFS can be used to find the shortest path from one node to all others.

2.7.2 Unordered Breadth-First Search

A BFS algorithm implementing a concurrent FIFO queue can give an incorrect shortest path because it is not possible to guarantee the order of execution of the different threads without synchronization mechanisms. This will result in the search algorithm extending outward in unforeseen ways, potentially causing the path found to be larger than the actual shortest path. In order to eliminate this behavior, more work is incurred as shown in [7]. The extra work leading to incorrect paths is designated wasted work by [29] since it has to be exchanged. Furthermore, a relaxed FIFO queue can also be used in such an algorithm. Due to relaxation, nodes may

be dequeued despite having been added after others. For example, nodes may be chosen in the order of a stack rather than a queue, resulting in behavior similar to a depth-first search algorithm, which has no guarantee of finding the shortest path. However, this is also possible to overcome with extra work.

3

Methods

3.1 Implementation of data structures

Here the tools and methods that were used to implement the different data structures are described.

3.1.1 Scal

Scal [28] is a framework designed to implement and benchmark concurrent data structures. It provides multiple benchmarks to test and compare various concurrent implementations. It has also been used to produce research as in [21]. Furthermore, it contains implementations of data structures. For example, it has an implementation of *1-RA*. For these reasons, and because it was designed by the creators of the d-RA algorithm, it was deemed appropriate to build our implementations in this framework.

An issue with using the framework is that it relies on cyclic memory for memory allocation. Before every benchmark, a maximum size of thread local memory has to be explicitly allocated. This can be problematic in a real-life setting for two reasons. Firstly, the memory demands might not be known beforehand and secondly, this limits the number of put operations done on the data structure. To avoid these problems, the partial queues we use instead utilize hazard pointers. Hazard pointers allow for allocating memory dynamically, although they introduce more overhead than the memory allocator, which is also stated by the authors of the framework [28].

3.1.2 Preexisting Implementations of Data Structures

Initially, an implementation of a Michael-Scott queue found in *Scal* was used but due to poor performance, another implementation of the Michael-Scott queue was found [31]. However, the version of 1-RA found in Scal was still used and was extended to support d-RA. Implementations were also found for LCRQ [11], FAAAQ [6]. All the partial queues use the same hazard pointer implementation [32] that was included with them. An optimized version of the 2Dd-queue from [20] was also used and a version of Bag from [8] that was given to us.

3.1.3 RCQS and RCQD

There were no implementations of *RCQS* or *RCQD* provided by [21], therefore, they were implemented in Scal by closely following the pseudocode they provide. The sleep and wake functionality uses the Linux Futex system call [33]. The pseudocode calls for wait functionality in the instance that a thread reaches a slot but is unable to complete its operation. This was implemented by checking the time stamp counter and waiting until a set number of cycles had passed to achieve a delay. How many times this wait functionality is called is determined by specifying a number of spins. Additionally, the slots in the queue are cache line padded to 256 bytes, to mitigate false sharing.

3.1.4 Length Approximation for d-RA

In order for d-RA to function it is important that the length of the partial queues can be approximated. In the case of the Michael-Scott queue size was calculated by storing the total number of nodes enqueued. During the creation of a node, the previous tail's index is incremented. Using these indexes the length of the queue was approximated. Note that it is an approximation since the head and tail indexes are read separately, meaning that one may change after being read, resulting in the size not being accurate. In the case of [1], ABA-pointers were used for this but as mentioned, their version yielded poor performance during testing. The resulting heuristic can be seen below:

```
queueSize = tail->idx - head->idx;
```

Here, tail->idx denotes the index of the tail node and head->idx is the index of the head node.

In the case of FAAAQ and LCRQ, it is not trivial to approximate their lengths because they are designed as linked lists of lists. In the same fashion as the Michael-Scott queue, the size of these queues is approximated by subtracting the head index from the tail index. For FAAAQ, this was done by storing a segment index belonging to each segment. As a result, the head index could be calculated by taking the index of the next element to be dequeued inside the head segment, and then adding the index of that same segment multiplied by the segment size. This was then subtracted from the tail index, which was calculated by taking the index of the next element to be enqueued in the tail segment and adding the index of the segment multiplied by the size of a segment. The resulting heuristic looks as follows:

```
tailidx = (tail->enqidx + tail->segidx * BUFFER_SIZE);
headidx = (head->deqidx + head->segidx * BUFFER_SIZE);
queuesize = tailidx - headidx;
```

Here, tail->enqidx is the current index of the last enqueued element in the tail segment, head->deqidx is the index of the next element to be dequeued in the head segment, BUFFER_SIZE is the size of a segment, and tail->segidx and head->segidx corresponds to the index of the tail and head segments.

LCRQ has a slightly different design due to the way indexes work in a CRQ. Because

of its cyclic nature, indexes can become substantially bigger than the size of the segment, and therefore the length could not be approximated in the same way. Instead, every time a new segment is created, the latest tail index is brought along as the start index of it. This means that the actual tail index is the next element to be enqueued in the tail segment together with its start index. Similarly, the actual head index is the next element to be dequeued in the head segment added to its start index. The resulting heuristic looks as follows:

```
tailidx = (tail->enqidx + tail->startidx);
headidx = (head->deqidx + head->startidx);
queuesize = tailidx - headidx;
```

Here, tail->enqidx is the index of the next element to be enqueued in the tail segment, head->deqidx is the index of the next element to be dequeued in the head segment, head->startidx and tail->startidx is the start index of the head and tail segments. A small technical detail that is important to mention is that LCRQ uses a bit in the indexes to signify whether or not the CRQ is closed for enqueuers. This bit is ignored by the heuristic.

Note that the length calculation for FAAAQ and LCRQ are approximations. Partially, because enqueue and dequeue indexes are read separately, leading to possible interleavings where one is subject to change after being read. Additionally, it is possible for these indexes to be updated without elements being added or removed, as is the case when the dequeue index becomes larger than the enqueue index.

3.1.5 d-RA Emptiness Checks

In a distributed data structure it can be challenging to discern whether or not it is empty. As mentioned, this was solved in the Michael-Scott-based d-RA by checking if the tail has changed for every partial queue. Only checking for the index of the last enqueued item (the tail) can cause issues related to linearizability whenever FAAAQ or LCRQ is used in a distributed manner. If enough elements are enqueued between the tail checks, a new segment can be created and the enqueue index can reach the same value in the new segment. This would make it look like the enqueue index has not changed. Hence, there is a theoretical interleaving that results in the distributed data structure returning empty even though it is not. This was resolved by creating a unique identifier with the index of the tail segment in the 32 most significant bits and the enqueue index in the 32 least significant bits. If neither has changed, no enqueue operation has been performed.

However, guaranteeing emptiness is problematic for the distributed LCRQ. This is because the emptiness checks do not work without modifying the LCRQ algorithm, since LCRQ will update the tail index of the head segment whenever the segment's head surpasses the segment's tail. This can cause the partial queue to detect changes made to itself whenever it finishes a dequeue operation. This in turn can cause an infinite loop. Therefore, LCRQ was modified to instead let the head pass the tail in order to avoid this behavior as this functionality is an optimization and not needed [5].

Unfortunately, this can result in the head surpassing the tail substantially. Hence, the decision was made to add an early return from the queue before any indexes are incremented. A thread will return empty if the segment's head index is larger than its tail index and if there are no other segments. This does not cause any issues related to linearizability as the problematic behavior, in that case, would be if it was possible that the thread returns, signifying that the queue is empty while it is not. This can in turn only occur whenever the head has not surpassed the tail. This is not possible as the tail index is read after the head index. Moreover, even if this was not the case, meaning that tail could be larger than head and the comparison succeeds, the changes made to the tail would be detected by the algorithm.

Finally, the implementation from [11] did not have the starvation functionality as described in [5] but it was implemented. Whenever a thread fails to enqueue an item a set number of times, a new segment is created. This is needed to guarantee progress for enqueuers which in turn is needed to make the algorithm lock-free.

3.1.6 Progress-Based Heuristic

An alternative heuristic to d-RA's length-based one was implemented. It chooses partial queues based on how much progress has been made by enqueue and dequeue operations. Specifically, enqueue operations are performed on the partial queue that has had the fewest number of elements enqueued, i.e., the one with the lowest tail/enqueue index. Similarly, dequeue operations are done on the one that has had the fewest number of elements dequeued, i.e., has the lowest head/dequeue index. In contrast to the length-based heuristic, this progress-based one has the potential to lower contention since the head and tail would not be acquired simultaneously. Furthermore, intuitively, this heuristic can balance the operations done on the partial queues by always performing them on the one that has made the least progress thus far.

Important to mention however, is that this heuristic has the same type of shortcoming as the length-based one as it will rely on approximations. As stated, in the case of LCRQ and FAAAQ, enqueue and dequeue operations can increment tail and head indexes more than once due to failures. This can cause the indexes to not accurately reflect how much progress has been made.

3.2 Correctness

Here the correctness of the derived queues will be discussed.

3.2.1 Lock-Freedom

No attempt will be made to prove lock-freedom for the used partial queues. This is because there is no published proof of lock-freedom of FAAAQ and since LCRQ has been modified, the existing proof can not be regarded as complete, and writing such proofs is a difficult and time-consuming task. Therefore the assumption will be made that both of these queues are lock-free. However, as the changes made to

FAAAQ only involve one new shared variable that is used for the d-RA algorithm, and the changes made to LCRQ are similar to those of FAAAQ, have been previously argued for, or are supported by [5], this property should not be affected. The only relevant difference in terms of lock-freedom between the original d-RA, which is lock-free [1], and the new ones using the FAAAQ and LCRQ is that an enqueue can change the tail of the queue without succeeding in its operation. As seen with the unmodified LCRQ, this can result in problems since a dequeuer can only return empty whenever no changes to the tail of a partial queue are detected. This can in turn lead to an infinite loop where a dequeuer detects failed enqueues and thus never returns signifying emptiness. The difference is that enqueuers will eventually succeed. In the case of FAAAQ this is true as the segments only allow a set number of operations done to them. In the case of LCRQ, this is true as if an enqueuer starves, it will force the creation of a new segment. As such, there is always an enqueue operation, and consequently, a dequeuer, that will be able to make progress. If a dequeuer can always make progress, then the algorithm is lock-free.

3.2.2 Linearizability

The same assumptions made in regards to lock-freedom will be made here in regard to linearizability for the same reasons. The changes made to the partial queues should not affect linearizability using the same argument as with lock-freedom. Furthermore, the assumption will be made that all elements are unique as was done in [1].

The argument for the linearizability of the d-RA algorithm presented in [1] largely holds true for the modified d-RA. Three conditions are mentioned in [1] for the algorithm to be linearizable with respect to a pool.

1. The first states that an enqueue operation can only occur once for a specific element. As every element is unique, the partial queues are linearizable FIFO queues, and the distributed queue algorithm returns directly after the enqueue operation is finished on a partial queue, this holds.
2. The second condition is that a dequeue operation for a specific element can only occur once. This is true using the same logic as in the case of the enqueue operations.
3. The third is the guarantee of emptiness. Namely, whenever the distributed data structure returns indicating emptiness, there has to be some point in time when all the partial queues are empty [1]. This point is defined to be after the initial pass of the partial queues as in [1].

What is different is the third condition. Their argument has to be extended as for FAAAQ and LCRQ the tail is not returned but instead, a combination of the number of segments added and the current index of the last enqueue operation of the tail. To prove that the queue is linearizable in regards to emptiness the following has to be proven:

The combination of the segments added and their respective enqueue indexes form a

unique identifier.

In order for this to be true, this identifier directly corresponds to a specific unique state of a partial queue with regard to enqueue operations. Dequeues can be disregarded.

This entails that an enqueue operation has to increment either the number of segments added or their respective enqueue indices. Whenever a new segment is added, the variable signifying its addition is incremented before any CAS operation is tried. The enqueue index is incremented atomically before any CAS is attempted. Since these statements are true, a unique identifier can be created.

Since the combination of added segments and their respective dequeue indexes forms a unique identifier, it can be used to show if any changes have been made to the tail of the partial queue. Therefore, if a dequeueing thread returns indicating emptiness, the data structure must have been empty at some point in time. Hence, the queue is linearizable when empty.

3.3 Applications

Throughout this project, efforts were made to identify a suitable application for a relaxed FIFO queue. This was not a trivial task as the intent was to find algorithms where there exists an inherent tolerance to the queue's relaxation. At the same time, the performance speed-up of the queue has to be substantial enough to justify the possible costs associated with the relaxation. In the end, it was decided that the simplest application to investigate would be breadth-first search. Scal included functionality for sequential BFS such as graph file parsing, and that was used to implement the following parallel algorithms.

3.3.1 A Breadth-First Search Algorithm With Errors

Breadth-first search has previously been used in the context of relaxed FIFO queues [1] and in regards to relaxed priority queues [29]. However, to our knowledge, no investigation has been done to find the relationship between relaxation and the errors associated with it in the context of FIFO queues. A simple way to do this is to calculate the shortest path from a root node to the other nodes in the graph. This can be done by continually taking a node from the queue, updating its neighbors' distance and parent, and adding them to the queue. However, as long as the graph is not a tree, behavior deviating from traditional BFS behavior can result in errors. The resulting algorithm, which is a simple parallelization of the BFS-algorithm found in Scal, can be seen below.

1. Search commences at a starting node, which at the start is the current node.
2. The neighboring nodes' distances from the start are calculated as the current node's distance plus one. Distance is assigned to a node with a compare-and-swap operation that only succeeds if the node has no distance previously assigned to it. For performance reasons, the algorithm will only attempt the

compare-and-swap operation if, during a previous check, there was no distance associated with the node.

3. All unvisited neighbors of the current node are added to a FIFO queue if the node's distance was successfully updated.

4. The next node is chosen from the head of the queue and becomes the new current node.

5. Repeat steps 2 through 4 until all nodes have been visited, i.e., the queue is empty.

6. The algorithm terminates when there are no active threads doing work on elements of the queue.

3.3.2 An Unordered Breadth-First Search algorithm

As mentioned, due to scheduling and relaxation the previous algorithm will not necessarily yield the shortest path from the root node to the other nodes of the graph. However, a small modification was done to the previously described algorithm, in order to overcome the drawbacks of this. This version acquires a node from the queue and then attempts a compare-and-swap operation to update its distance if none has been previously assigned, the same way the previous algorithm works. Additionally, even if a distance has been set, it will update it if it yields a shorter path than the one previously assigned. It also adds these neighbors to the queue even if they have been visited. This continues until the queue is empty. Relaxation in the traversal of a graph leads to wasted work [29], however, if the increase in throughput is high enough the costs associated with the relaxation may be overcome. We define work as the total number of successful compare-and-swap operations that update the distance of a node. The algorithm can be seen below and is very similar to the Fixpoint algorithm described in [7].

1. Search commences at a starting node, which at the start is the current node.

2. The neighboring nodes' distances from the start are calculated as the current node's distance plus one. The distance is updated in the case that none has been previously assigned and also if the current node's distance plus one is smaller than what was previously assigned. This is done through a CAS operation. If it fails, the entire step will be repeated for that neighbor.

3. If the distance is updated, all neighbors of the neighboring node are added to the FIFO queue.

4. The next node is chosen from the head of the queue and becomes the new current node.

5. Repeat steps 2 through 4 until the queue is empty.

6. The algorithm terminates when there are no active threads doing work on elements of the queue.

Graph	|V|	|E|	Type
sparse10M	1000000	10502285	Directed (random)
sparse50M	1000000	50497177	Directed (random)
sparse200M	1000000	200456282	Directed (random)
audikw_1	943695	38354076	Undirected
cage15	5154859	47022346	Undirected

Table 3.1: The graphs used for benchmarking

3.3.3 Graphs

To test the speed of the implemented BFS algorithms, multiple graphs were tested and traversed, see Table 3.1. The graphs use a graph-file format called *METIS* [34] and are unweighted. Technically, the randomized graphs do not exactly follow this standard as the format to our knowledge is designed for undirected graphs. Still, it works with the pre-implemented graph parsing code in Scal. This graph format does not count duplicate edges which means that the edges, A-B and B-A are regarded as one edge.

The randomly generated graphs are directed and they were generated by randomizing the outgoing edges for each node. The minimum number of outgoing edges is one and at maximum n. As the effects of relaxation are going to be dependent on the number of edges in the graph, it was important to test different graph densities. Still, due to the benchmarking time, a maximum of $n = 400$ was used. Therefore all the graphs can still be considered relatively sparse as they all have 1 million vertices.

The undirected graphs audikw_1 and cage15 were included in Scal but can also be found in [35].

3.4 Benchmarks

In *Scal*, there are two benchmarks that we will use. The first one is called producer-consumer and it simulates an environment where objects are both enqueued and dequeued in parallel, where some threads act as consumers and some as producers. In between the operations, the threads can wait for c CPU cycles or alternatively compute Pi to the c_i decimal. However, no waits were used in our benchmarks. This benchmark simulates an environment that tests the queue's functionality to act in a manner that is similar to a letterbox, where enqueues and dequeues are in parallel.

The other benchmark is called sequential alternating. Here, all the threads do an enqueue operation followed by a dequeue operation. This benchmark is of special relevance for the Bag [8] as it should be able to realize its full potential in terms of thread local storage.

In the case of sequential alternating there already was an option to start the benchmark where every thread starts to enqueue a specified number of elements in the data structure before any dequeue operations are started. This also means that a thread will have several consecutive dequeue-operations. However, this functionality

was modified slightly as there was an off-by-one error in it. The reason why using prefill is of interest is that both FAAAQ and LCRQ each store a certain number of elements before a new segment is created. This means that it is possible in the case of producer-consumer benchmark, and guaranteed in the case of the sequential alternating benchmark, that the creation and removal of segments is not done. Furthermore, some data structures have dramatically different performance whenever there are few elements in them. Therefore, the inclusion of prefill is regarded as a necessity to accurately measure the performance of the queues.

3.4.1 Calculating Relaxation Distance

In Scal there is no available functionality to measure relaxation so we had to implement our own. Fortunately, there was functionality that logs the operation that was done by the benchmark and it was possible to associate every operation with a timestamp. For every data structure that we have measured therefore we add a timestamp at every point of linearization that is stored thread locally. According to [1], logging only has a small impact on performance. After the benchmark is finished, all the thread-local time stamps are collected and sorted. Then, for every operation, the relaxation distance is calculated, meaning how out of order the operation is. This is done by simulating the execution of the data structures from the logs by calculating how many dequeues precede the item that should be dequeued according to the current enqueue. When this is done for an item, both the enqueue and dequeue operations are marked as finished and ignored in subsequent calculations. If a dequeue operation fails, which can happen in the producer-consumer benchmark, the operation is ignored in the relaxation calculation.

Currently, the downside of this approach is that it is performance intensive and only works for a much smaller number of operations than is used for the ordinary benchmarks. This is deemed sufficient for our purposes but it is still a possible source of error. For example, the memory reclamation functionality will work differently as the smaller number of operations might not trigger garbage collection.

3.4.2 Configuration of Data Structures

Some data structures have configurations associated with them. The configurations are shown and explained below.

- ***FAAAQ*** Segment size is set to 1024, the default size of the implementation taken from [6].
- ***LCRQ*** Segment size is set to 1024, the default value of the implementation taken from [11] and the one found in Scal. Also, LCRQ will regard an enqueuer as starving if it fails six consecutive enqueue operations. This number was found experimentally..
- ***RCQS and RCQD*** Array size is set to 2048 slots, which was the case in [21]. Wait time was set to 5000 cycles, and spin, which signifies the number of waits that are called, was set to 1000. These numbers were found experimentally.

- ***2Dd-Queue*** Relaxation bound was set to 10^3 to make it comparable in relaxation to the other data structures, width (number of sub-structures) was set to the number of threads, and the depth (the number of allowed operations on a sub-structure until another one is tried) was calculated automatically from the previous values.

3.5 Benchmark Environment

The environment used to test and run benchmarks consists of two 128-core AMD EPYC 9754 processors at 2.25GHz with simultaneous multithreading, 256 MB L3 cache, and 755 GB of RAM, running openSUSE Tumbleweed with 7.4.1 Linux kernel, GCC/g++ version 13.2.1 20240206. This platform utilizes the x86 instruction set, which is required to access all the instructions needed. The two processors allow for 512 threads to be run simultaneously, although we only use one of the processors, limiting us to 256 threads. This lets us examine how well the data structures scale at very high core counts. Threads are pinned to separate logical cores. This is done in a round-robin fashion over the Core Complexes (CCX), which means that all CCXs will have one thread pinned to them before any has two, each will have two before any has three, and so on. Since the benchmarks in this thesis use enough threads to require multiple CCXs, this method of pinning will eliminate the change in synchronization time that will come when the thread count exceeds one CCX.

4

Results and Discussion

4.1 Results of New Partial Queues

In this chapter of the report, we show the performance of the queues in the producer-consumer and sequential alternating benchmarks. We also discuss the results and evaluate their implications.

The evaluation of the distributed data structure was done with 80 partial queues. This was based on the results of [1] where 80 partial queues were determined to work well. Only 1-RA and 2-RA were tested as in [1]. All the benchmarks were run 5 times and for every data structure, both the mean and standard deviation of the results are plotted. Throughput is in the unit of operations per millisecond. Furthermore, to achieve these results no wait was employed and the number of operations per thread was set to 1000000 in the throughput benchmarks. The number of operations used in relaxation benchmarks was set to 1000 except in one specific benchmark where 20000 was used, see Figure 4.9. The thread count increases exponentially with a base of 2. The maximum number of threads used is 256, however, the Bag is limited to 64 due to its implementation. The number of operations in the throughput benchmarks corresponds to that of [1]. RCQD is not shown here as it performed almost identically to RCQS in testing.

4.1.1 Performance of the Producer-Consumer Benchmarks

In this benchmark 50% of the threads act as producers and 50% act as consumers. In Figure 4.1 the producer-consumer results of all queues and the Bag can be seen. In this benchmark, the absolutely highest throughput comes from 1-RA FAAAQ, followed by 1-RA LCRQ. 1-RA LCRQ does not perform as well at lower thread counts, which we suspect is because it often encounters empty partial queues, preempting enqueuers, and as a result creates new segments due to the starvation functionality. They are shown to outperform the 1-RA Michael Scott queue by a large amount and in contrast, they continuously scale with the number of threads. However, note that the x-axis is logarithmic, and since no queue shows a doubling of throughput with a doubling of threads, none scale perfectly.

The other queues that are benchmarked here fall behind as contention increases. Seemingly, FAAAQ and LCRQ are able to take advantage of the low amount of contention at low thread counts. Here they outperform the 2-RA algorithm, likely

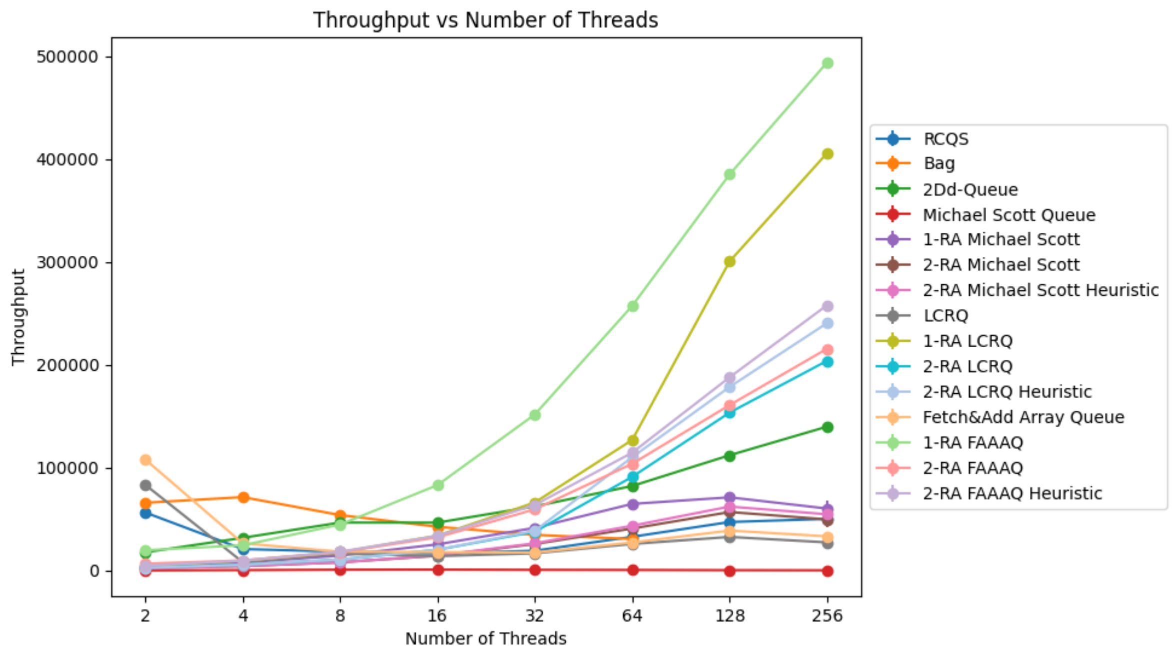


Figure 4.1: Throughput results of the producer-consumer benchmark. Higher is better.

both due to less complexity and better locality since they do not switch between partial queues. As contention increases their performance decreases, although they are able to scale somewhat even at very high thread counts. The Michael Scott queue is the slowest data structure. As described in Section 2.2.4, it is expected that this queue suffers as contention increases due to CAS failures. Moreover, it is made worse by memory reclamation, which is done whenever a node is removed from the queue. Without the memory reclamation, it performs better but is still not close to the other data structures.

The Bag performs well at low thread counts but its performance continually deteriorates. This is an expected result for producer-consumer and aligns well with the results of [1]. The 2Dd-queue is slow when there are few threads, however, as contention increases it scales better than 1-RA Michael Scott by a margin. The fact that it is not empty linearizability likely helps its performance. Also, RCQS initially performs quite well, at a level close to LCRQ and similarly, it diminishes as more threads are added. However, as contention increases it is able to scale better, to the point it almost performs at the level of 1-RA Michael Scott. It is worth mentioning that RCQS is likely aided in this benchmark by not being dynamic and not requiring memory reclamation.

The 2-RA versions continue to scale with the number of threads similarly to, although not as well as, the 1-RA versions. Looking closely, the Michael Scott queue performs better with the 2-RA algorithm than without, which is expected due to lower contention. The benchmark reveals it to scale better than the concurrent

queues LCRQ and FAAAQ, however, they are similar in performance at high thread counts. It is seen to decrease at high thread counts, unlike the LCRQ and FAAAQ versions of 2-RA. This is likely the result of increased contention, which Michael Scott is more susceptible to.

Furthermore, the progress-based heuristic performs better than the length-based one in these benchmarks. The difference is increasingly obvious as the thread count increases. This indicates that as contention increases there is an advantage in choosing based only on either the head or tail, possibly because it decreases contention. This difference is especially visible for the 2-RA queues based on LCRQ and FAAAQ. One explanation for this is that the length-based heuristic has a shortcoming in their case. Since indexes can be incremented, even during failures, it is possible that the heuristic mistakes an empty partial queue for a non-empty one which should impact performance.

4.1.2 Relaxation in Producer-Consumer Benchmarks

Relaxation measurements of the producer-consumer benchmark are shown in Figure 4.2. Note that the concurrent versions of the partial queues were not benchmarked as they are not relaxed. The plot reveals that although the 1-RA queues have strong throughput results, they are accompanied by a large amount of relaxation. We can also see that the 1-RA algorithm with FAAAQ and LCRQ has higher relaxation than Michael Scott. This could be because of a disparity between the execution time between enqueued and dequeues for these implementations. Because on average dequeueing requires more time than enqueueing with these queues, there can be a large number of enqueued elements in the queue at any time. Consequently, elements may be dequeued in a more random and, hence, relaxed manner. This, and the fact that 1-RA chooses partial queues entirely at random, results in a high average relaxation distance. This means these queues are less suited for applications where the order of operations matters.

Regarding the 2-RA implementations Figure 4.2 shows a vastly lower level of relaxation compared to the Bag and much lower than the 1-RA queues. Additionally, there are distinctions between different partial queues and the two heuristics. The FAAAQ and LCRQ versions of 2-RA have substantially higher levels of relaxation than the Michael Scott version. We theorize that this is dependent on the previously mentioned inaccuracies that can be caused by the length-based heuristic. Furthermore, due to these inaccuracies, we expect dequeuers to fail more frequently. As the amount of average relaxation is partially tied to the number of elements that reside in the queue at a time, frequent dequeue failures will increase the number of elements in the data structure. For example, if there are only 20 elements in the queue, the highest relaxation distance possible is 19. However, the progress-based heuristic is shown to lower the level of relaxation to an almost bounded level, and it does so for all versions of 2-RA.

The 2Dd-queue is seen to vary between higher and lower levels of relaxation, but it constantly maintains a higher degree than the 2-RA implementations until 2-RA LCRQ and 2-RA FAAAQ with the length-based heuristic surpass it. RCQS is barely

relaxed until higher levels of contention, which is expected as multiple enqueuers or dequeuers need to reach the same slot for relaxation to happen. At higher thread counts, threads attempting to perform the same operation can cycle around the array to the same slot again before the thread previously assigned to it completes its task [21].

Unsurprisingly, Figure 4.2 shows that the Bag is relaxed enough to extend outside what is shown in the graph. The whole graph can be seen in appendix Figure A.1. As was shown in the previous section it did not perform very well for the amount of relaxation it has. This is not an entirely unexpected result as the producer-consumer benchmark renders it unable to take advantage of locality, which is something it otherwise is proficient in.

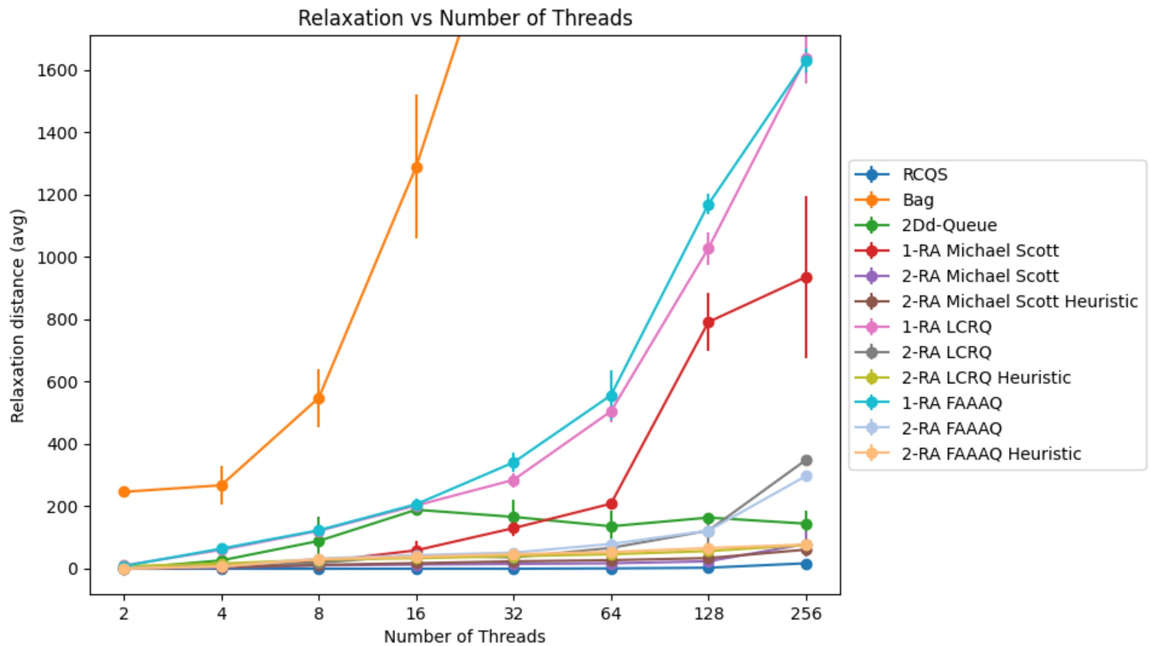


Figure 4.2: Relaxation results of the producer-consumer benchmark. Lower is better.

4.1.3 Throughput in Sequential Alternating Benchmarks

In this benchmark, each thread attempts to enqueue and dequeue 1000000 elements. We show benchmarks with 0%, 1%, and 100% prefill to showcase how the performance differs in different scenarios. A prefill of 10% was also tested but yielded very similar results to that of 1% prefill. As was mentioned, prefill causes each thread to enqueue a specified number of elements before any dequeues are performed. There is, however, no guarantee that threads finish enqueueing at the same time, meaning there may be overlap where some are in the enqueuing phase while others have begun dequeuing. RCQS is not included in sequential alternating because it is static in size and hence, unable to handle prefill.

The 2Dd-queue is not empty-linearizable and as such we allowed it to fail a dequeue

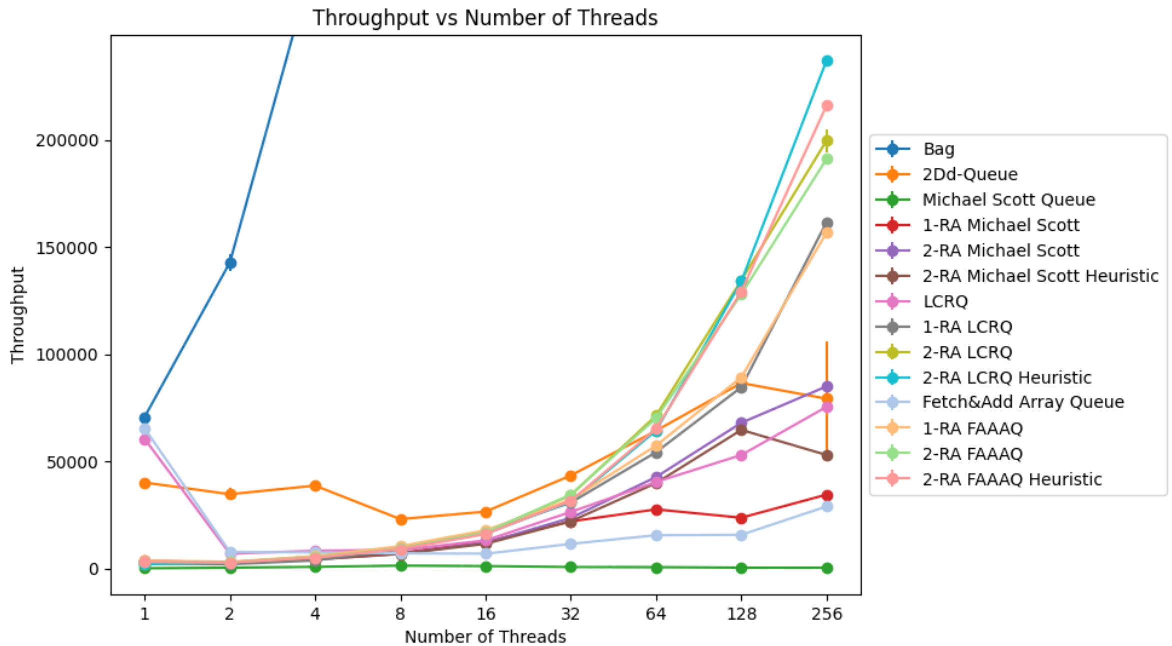


Figure 4.3: Throughput results of the sequential alternating benchmark with 0% prefill. Higher is better.

operation which is not the case for the other data structures. This should not affect its performance dramatically but the fact that it is not empty linearizable will yield performance benefits over the d-RA based queues.

Figure 4.3 shows throughput results in this benchmark. The strongest performance comes from the Bag. It quickly scales too much to be shown with the queues but can be found in the appendix in Figure A.2. Sequential alternating always enqueues an element before dequeuing, hence, there will always be an element in the Bag's thread local storage before a dequeue is performed, which means it can always work thread locally. It manages performance more than an order of magnitude larger than the second-best data structure despite being limited to 64 simultaneous threads.

Furthermore, Figure 4.3 shows that the throughput of the 1-RA queues does not excede the 2-RA versions unlike in the producer-consumer benchmark, which is likely due to the high number of empty partial queues that will be encountered by dequeuers. Unlike 2-RA, 1-RA has no heuristic that can help it avoid empty queues. 1-RA Michael Scott does very poorly in this benchmark.

The results for the other queues differ somewhat from the producer-consumer benchmarks. LCRQ performs significantly better than FAAAQ in this scenario, which is likely because the cyclic design of LCRQ makes it able to take advantage of locality. This is not possible in the same way for FAAAQ since it creates new segments even if there is open space at the start of the segment. The 2-RA algorithm with Michael Scott barely outperforms LCRQ at high thread counts, which may be the result of high contention. Surprisingly, despite the progress-based heuristic only requiring ei-

ther the head or tail at once, it lowers performance when the thread count increases from 128 to 256.

2-RA FAAAQ and 2-RA LCRQ scale better with increased threads, the same way they did in the producer-consumer benchmark. However, LCRQ is faster than FAAAQ with both length- and progress-based heuristics. This might be because of how LCRQ creates new segments to avoid starvation, as versions tested without this feature did not perform as well. The new heuristic is able to further increase throughput in this benchmark for these queues.

The 2Dd-queue performs well here, maintaining a high level of throughput at all levels of contention. It is only surpassed by the 1-RA and 2-RA queues at 256 threads where it has stopped scaling. The strong performance might be because of its ability to work thread locally. However, as the number of threads affects the depth, this ability will be affected by a higher thread count.

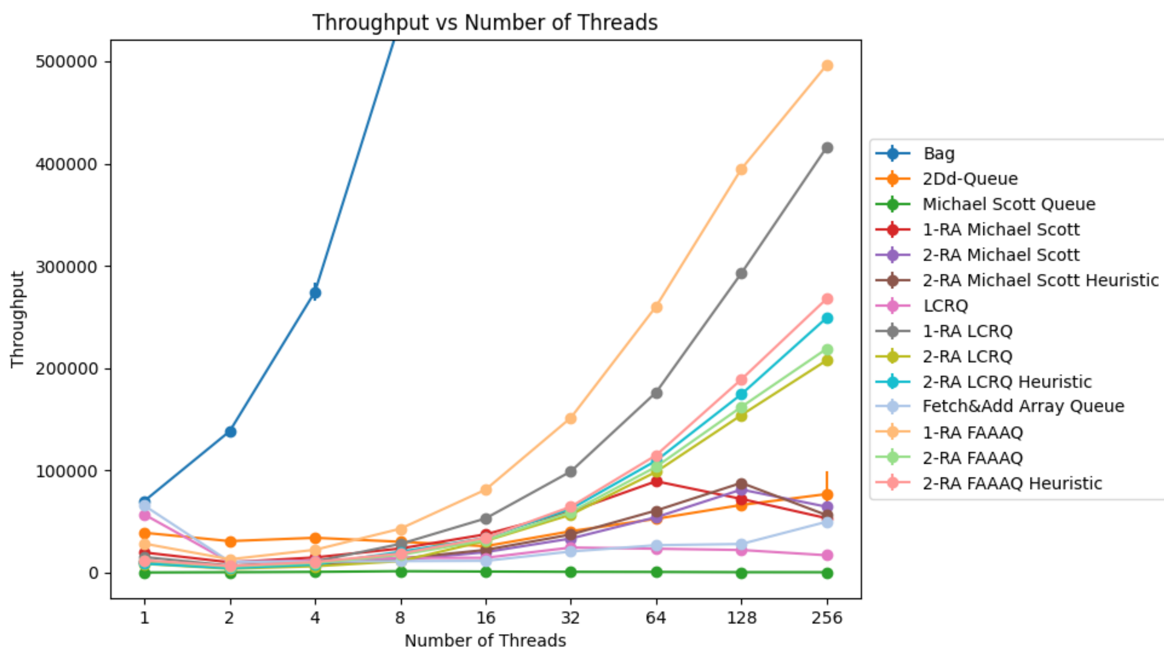


Figure 4.4: Throughput results of the sequential alternating benchmark with 1% prefill. Higher is better.

With 1% prefill, in Figure 4.4 the Bag has a lower throughput than with 0% prefill, yet still high enough to not fit in the plot. The whole plot is found in the appendix in Figure A.3. 1-RA FAAAQ and 1-RA LCRQ are seen to outperform the other queues, which is expected since there will now not be as many empty partial queues, which means that the lack of heuristic is not as problematic as before.

Overall the resulting order is very similar to those in the producer-consumer benchmark, which is not a surprise since in both scenarios the threads operate on partial queues that are less likely to be empty. However, similarly to the benchmark with 0% prefill, 1-RA Michael Scott is still performing worse than its 2-RA counterparts

at high thread counts. This could possibly be dependent on its lack of ability to balance the work of different threads across the different partial queues. During high levels of contention, we expect this to impact performance.

The progress-based heuristic, yet again performs better than the length-based one, except for 2-RA Michael Scott. Since this was also the case with 0% prefill it seems that whether the queue contains many elements or is close to empty, the progress-based heuristic is still able to outperform the other with regard to throughput. There is however an outlier in Michael Scott.

The 2Dd-queue scales more consistently in this benchmark than with 0% prefill. For example, the performance is still increasing at 256 threads. For some reason, the decrease in depth does not seem to affect throughput whenever the queue has more elements. However, the relative performance compared to the 2-RA and 1-RA queues is lower.

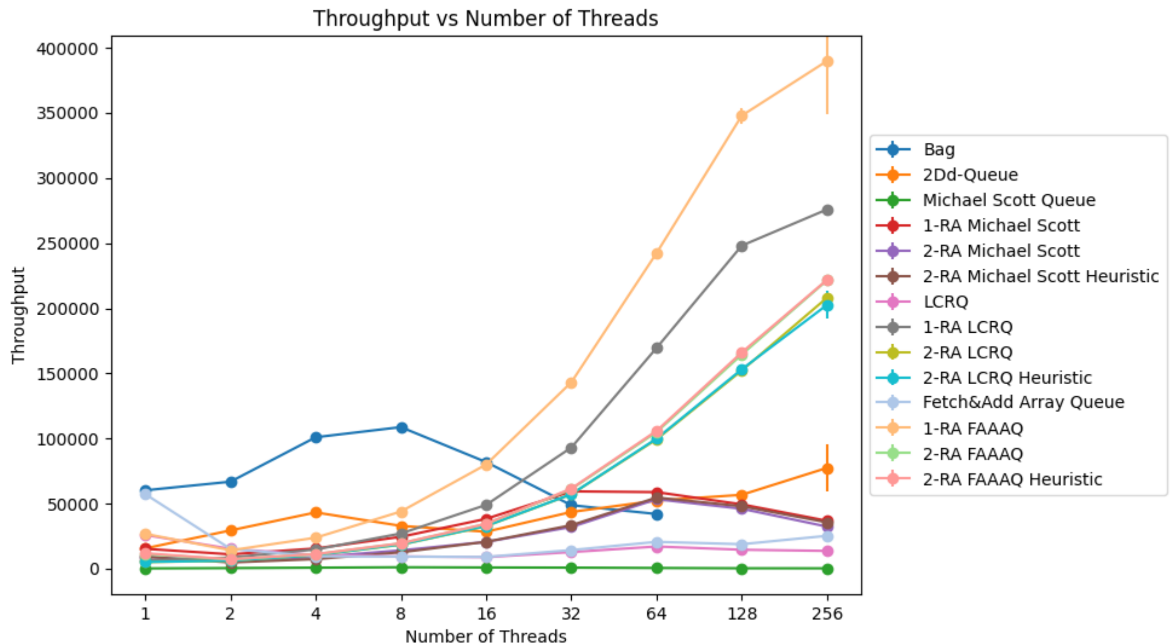


Figure 4.5: Throughput results of the sequential alternating benchmark with 100% prefill. Higher is better.

With 100% prefill, Figure 4.5 shows that the Bag suffers greatly in this benchmark. This causes the 1-RA and 2-RA versions of FAAAQ and LCRQ to surpass it significantly when contention increases. This result likely has to do with how the memory allocation functions for the Bag, where blocks are designed to fit inside a cache line and hence require a lot of memory allocation when many elements have to be added without removing any. This allocation is not as problematic when elements are enqueued and dequeued without filling a block. In testing, a larger block size yielded much higher throughput. Still, the values used to achieve those results dramatically deviated from the default implementation so the decision was made to

leave it unchanged.

Regarding the 2-RA queues, they have a lower throughput here than in producer-consumer and sequential alternating with 1% prefill. This could be explained by increased contention as for the majority of the time this benchmark works in two phases, one enqueue and one dequeue phase. During the first phase, threads will attempt to enqueue all their elements at once, and during the second then they will all attempt to dequeue everything. This means contention should be higher on the tail in the first phase and then higher at the head in the second. This may also be why the Michael Scott versions of 1-RA and 2-RA decrease in performance at lower thread counts than previously.

Furthermore, the order between the 2-RA queues is different here compared to other benchmarks. Here FAAAQ is better than LCRQ with both heuristics and the performance is close to identical between heuristics. Again, this is likely related to the two phases. During the first phase, the choice of partial queue mostly depends on the tail, and in the second, it mostly depends on the head. The progress-based heuristic always operates entirely based on the tail during enqueues and the head during dequeues, which means that in this benchmark the heuristics generally behave identically. Hence, the heuristic seems to have close to no impact on the throughput of the queues, resulting in the only difference being dependent on what partial queue is utilized.

The 2Dd-queue outperforms 1-RA and 2-RA Michael Scott. Again, this seems to be because the Michael Scott queue suffers from the high level of contention put on it here. The 2Dd-queue is, as mentioned, not empty linearizable and therefore does not need to save the tail after every attempted dequeue-operation. The actual throughput is very similar to the benchmark with 1% prefill.

4.1.4 Relaxation in Sequential Alternating Benchmarks

For the relaxation measurements in the sequential alternating benchmarks with 0% prefill, we can see in Figure 4.6, which provides some surprising results. The Bag is shown to have the lowest amount of relaxation here, which is a major deviation from results in producer-consumer and the expected behavior of a pool. We hypothesize that because a thread always enqueues an element before it attempts to dequeue one, it will dequeue the element it just enqueued, meaning that thread locally there will be no relaxation. Furthermore, because a thread is not slowed down by interacting with other threads, it can likely perform operations at a similar rate constantly. This seems to result in elements being enqueued and dequeued in very close to the correct order resulting in a very low level of relaxation.

As expected, the 1-RA versions have higher relaxation than the other queues. However, the outlier here is 1-RA LCRQ, which has one of the lowest levels of relaxation. This has to do with the starvation code in LCRQ as increasing the threshold for when new segments are created eliminated this behavior. Furthermore, the throughput of the relaxation measurement was investigated and it was substantially slower than the throughput measurement of the relaxation benchmark for 1-RA FAAAQ.

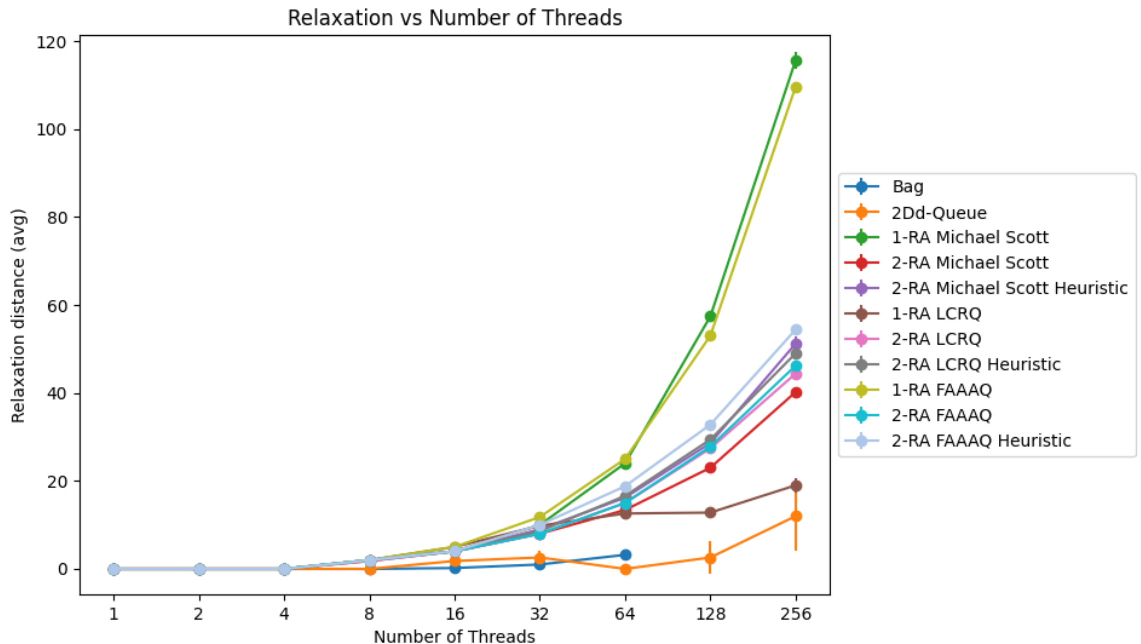


Figure 4.6: Relaxation results of the sequential alternating benchmark with 0% prefill. Lower is better.

We suspect that this measurement highlights a shortcoming of our method to measure relaxation as they should be comparable in performance. We believe that since the 1-RA algorithm does not differentiate between partial queues, it will be quite likely to pick empty queues and therefore cause enqueuers to starve. Creating a new segment should be a relatively costly operation as several enqueuers might attempt it. We deem that this for some reason, must be more likely to occur early on in the benchmark. Since the number of operations is low, this would impact throughput and relaxation to a much more substantial degree than in the throughput benchmark.

Relaxation scales relatively equally between all 2-RA queues, including those with the progress-based heuristic. This result differs from producer-consumer where relaxation for the heuristic looked almost bounded. We suspect that none of the queues ever contain enough elements to begin balancing relaxation. This is also supported by the fact that relaxation never reaches the same level as producer-consumer for any data structure in this benchmark.

The 2Dd-queue shows low levels of relaxation in this benchmark. Similarly to the pool, we expect that this is because of how it is able to work thread locally. An indication for this is that relaxation rises significantly at 256 threads, at the same point when Figure 4.3 shows performance was lowered. This decrease in throughput is believed to be because of its lessened ability to exploit locality. This is because its depth will decrease whenever threads increase due to the relaxation bound.

Figure 4.7 shows results with 1% prefill. The Bag behaves similarly to the 1-RA

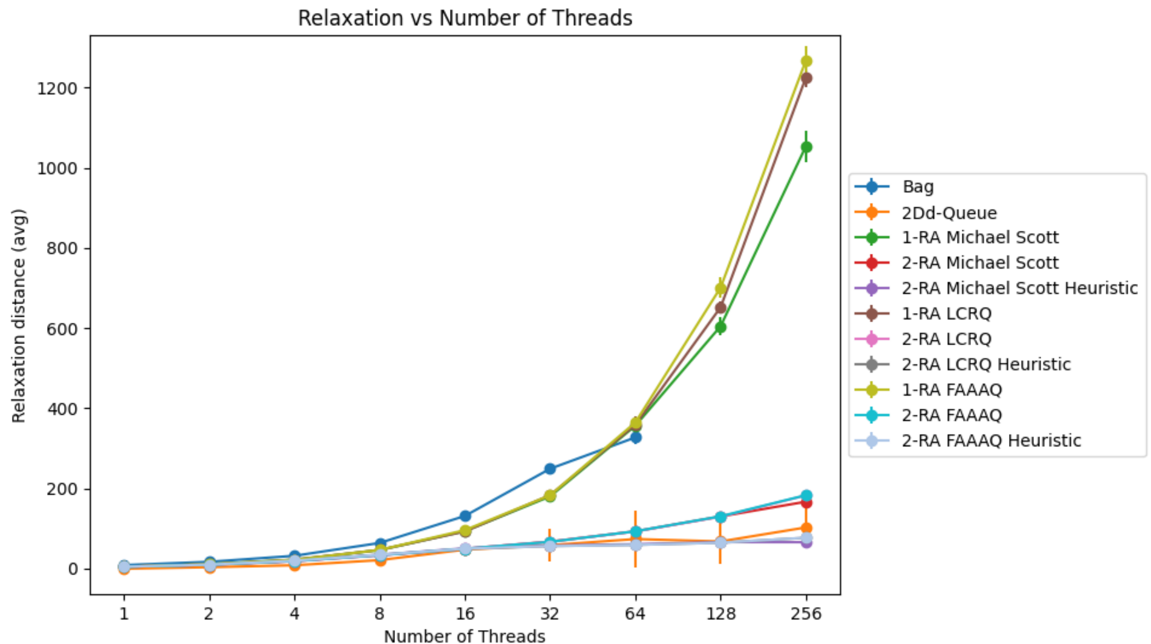


Figure 4.7: Relaxation results of the sequential alternating benchmark with 1% prefill. Lower is better.

algorithm in relaxation. Since it already contains elements when dequeuing begins, and because the thread local storage is implemented like a stack, average relaxation will be made higher as the first elements to be enqueued will remain until the last element is dequeued.

The rest of the results are similar to those in producer-consumer, which is also the case with throughput in this benchmark. 2-RA FAAAQ and 2-RA LCRQ are practically identical, making it difficult to see the LCRQ. The same is true for the progress-based heuristic-based queues. Generally, the new heuristic maintains lower levels of relaxation while it continually increases for the length-based one. This indicates that when the queue contains a sufficient number of elements, the progress-based heuristic chooses partial queues in a way that results in less relaxation than the length-based one.

The 2Dd-queue shows similar levels of relaxation here as the 2-RA progress-based queues. In this test, there are more elements in the queue once dequeuing begins. This shows that the 2Dd-queue handles relaxation relatively well when there are more elements the queue.

Finally, in Figure 4.8 we see the worst-case scenario for the Bag. Here all threads enqueue all their elements, but no barrier is employed before they begin dequeuing. Since the Bag does this in a thread-local stack, it results in the highest relaxation possible. It exceeds the relaxation in 1-RA by a large margin despite those queues also being highly relaxed. To see the graph in full, see appendix Figure A.4.

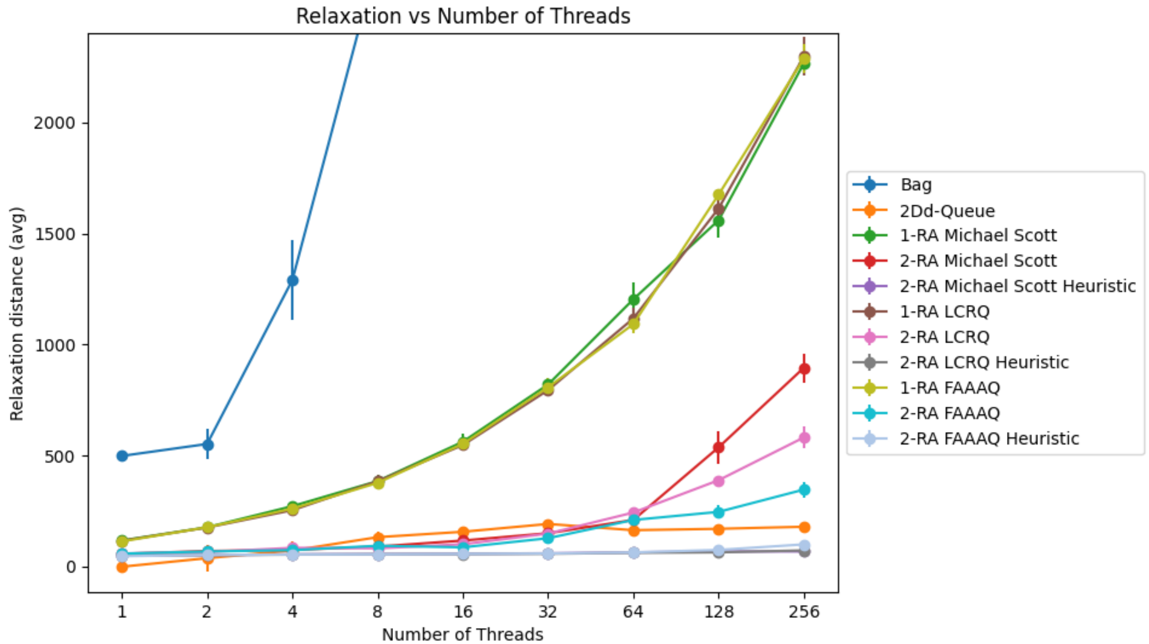


Figure 4.8: Relaxation results of the sequential alternating benchmark with 100% prefill. Lower is better.

Furthermore, as is the case in the other benchmarks, the progress-based heuristic limits relaxation. In contrast, the length-based heuristic scales with the number of threads. Interestingly, the difference in relaxation between the partial queues is notable when the length-based heuristic is applied. The most interesting observation can be found with 2-RA Michael-Scott as it has considerably higher relaxation than the other two 2-RA-based queues. We believe this is caused by contention between enqueuers initially. This is because, for a given Michael-Scott queue, only one enqueuer can succeed at one time as opposed to LCRQ or FAAAQ. This could lead to some threads being preempted while enqueueing, while other threads have started dequeuing. This is problematic for the length-based heuristic (see Section 4.1.5). The reason for the differences between LCRQ and FAAAQ has not been identified but it is probably related to contention between enqueuers.

Finally, we can see that the 2D queue has a stable relaxation that is lower than that of the length-based 2-RA data structures but higher than the progress-based queues.

4.1.5 Results of Progress-Based Heuristic

The progress-based heuristic introduced in this thesis provides interesting results in testing. In benchmarks where the queue is close to empty it does not perform quite as well as the length-based heuristic used in [1]. This can be seen with the Michael Scott version of 2-RA shown in Figure 4.2, which is likely to be close to empty since during measurements enqueues and dequeues require about the same amount of

time. This was not the case for LCRQ and FAAAQ. Here the length-based heuristic outperforms the other at all thread counts except 256. In contrast, the FAAAQ and LCRQ versions, both of which take longer to perform dequeues than enqueues, do have more elements in the queue and do show a substantially lower amount of relaxation with the progress-based heuristic in this benchmark.

In sequential alternating with 0% prefill, as is shown in Figure 4.6, the progress-based heuristic generally results in a higher level of relaxation. Again, this is caused by the fact that the queue will generally not be filled, rendering many partial queues empty. The interpretation is that the progress-based heuristic suffers from this since it does not have the same tendency to filter out shorter, and by extension empty, queues when performing dequeue operations. It is also difficult to distinguish significantly between the 2-RA queues in general in this benchmark, indicating that neither heuristic is balancing well in this scenario.

In other sequential alternating benchmarks where there is prefill of 1% and 100%, as is seen in Figures 4.7 and 4.8, the progress-based heuristic is shown to limit relaxation much better than the length-based one, which as stated is also the case for the FAAAQ and LCRQ versions in producer-consumer. This indicates, with sufficient elements in the queue, that the progress-based heuristic better limits relaxation. Also, the throughput is generally shown to improve as it mostly delivers performance improvement at high thread counts.

Furthermore, in testing the substantial differences between heuristics were not reproducible whenever a barrier was employed between the enqueue and dequeue operations, i.e., when all enqueue operations were done before the dequeue operations began. The heuristics become identical in this case but due to different levels of contention, some differences in relaxation were noted.

Assuming that elements are evenly distributed among the different partial queues and that there are active enqueuers and dequeuers, the length-based heuristic may pick queues in a non-optimal manner. This is because whenever an element is dequeued, the partial queue where that element used to reside is more likely to be chosen for a subsequent enqueue operation. This in turn will make it more likely to be dequeued from again. This will affect relaxation as the heuristic will not differentiate between partial queues that contain recently enqueued elements compared to older ones. This explains why the progress-based heuristic results in less relaxation in benchmarks where there are sufficient elements in the queue.

Figure 4.9 shows that the relaxation improvement from the progress-based heuristic is likely related to the hypothesis described above. In this test, there is only a single thread executing at once, while the number of partial queues varies. This means that there is no contention between threads, yet relaxation is much lower for the progress-based heuristic and there is little variance. As can be seen, the progress-based heuristic is able to perform substantially better even when there is no contention.

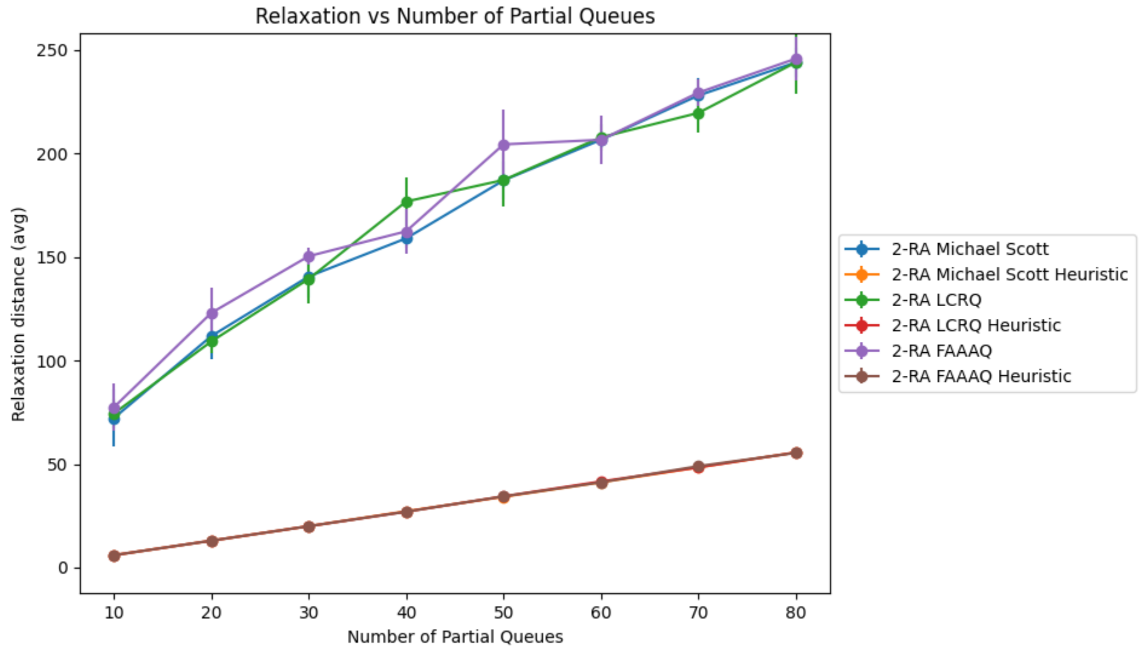
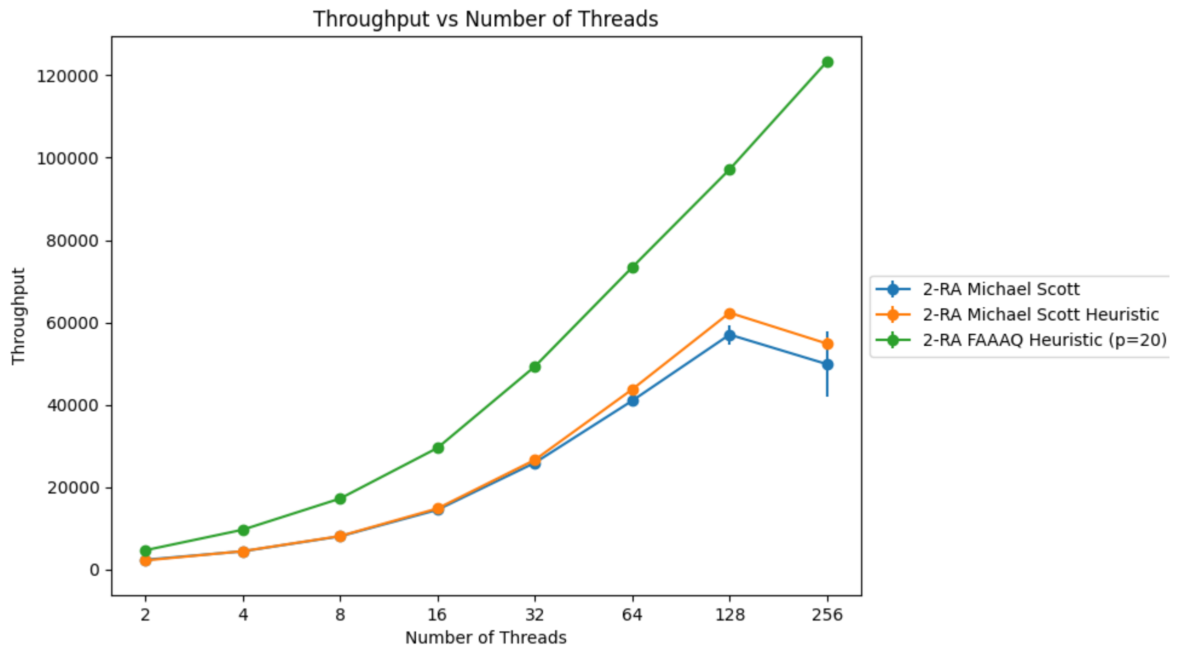
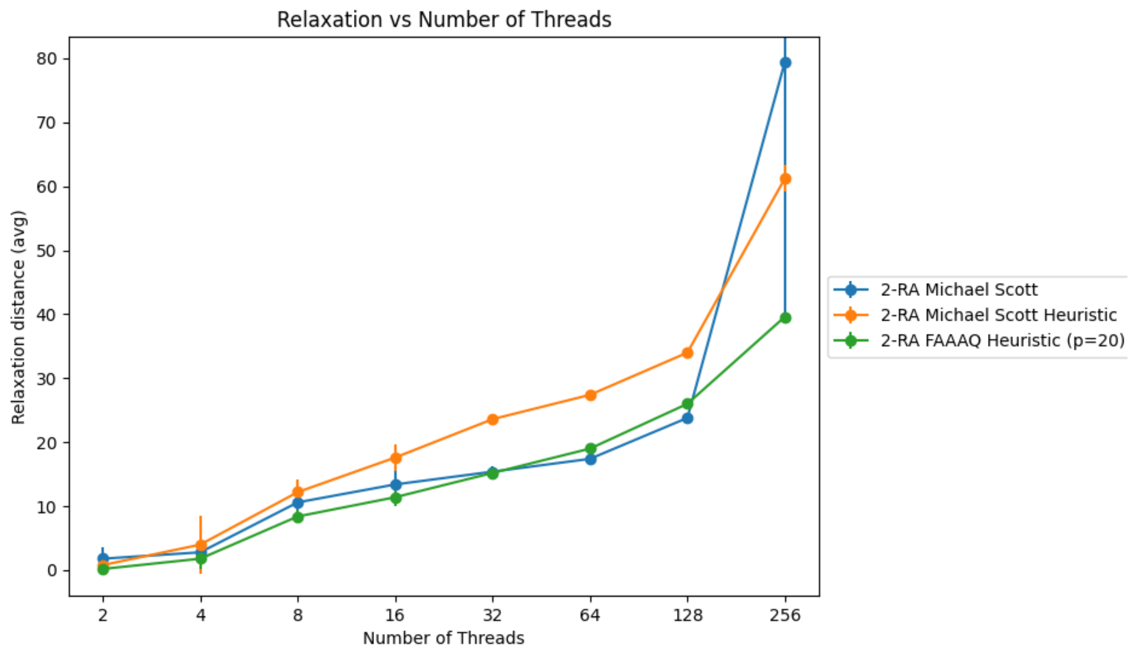


Figure 4.9: Relaxation results in sequential alternating with respect to partial queues. 50% prefill, 20000 total elements. One working thread. Lower is better.

These results can be used to show that it is possible to limit the relaxation substantially by decreasing the number of partial queues. In Figure, 4.10 a) we compare the throughput of 2-RA FAAAQ with 20 partial queues using the new heuristic and the previous results of the different Michael-Scott-based d-RAs in a producer-consumer benchmark. Here we can see that even if the number of partial queues is decreased, a substantial performance increase can be had. At the same time, we can see in Figure 4.10 b), that the relaxation of 2-RA FAAAQ is at worst comparable and at 256 superior.



(a) Throughput measurement



(b) Relaxation measurement

Figure 4.10: Throughput and relaxation of a producer-consumer benchmark comparing the progress-based d-RA FAAAQ with 20 partial threads against d-RA Michael-Scott versions

4.2 Shortest Path Calculations

In the following graphs, results are presented for the shortest path calculation when both errors are allowed and disallowed. For all the benchmarks, in order to ensure that no threads are killed prematurely, the following is done: a thread can only be killed if all threads have failed 1000 consecutive dequeue operations. The end time for the algorithm corresponds to the last time a thread completed a dequeue and subsequent operations. This was done to facilitate the comparison and if threads died prematurely, performance suffered. Initially, the Michael-Scott queue was considered for the benchmarking but it was not used in this comparison due to poor performance during testing. The progress-based heuristic was used for the d-RA versions as it outperforms the length-based heuristic in previous benchmarks. Furthermore, a version with 20 partial queues was tested for the progress-based 2-RA FAAAQ

Five roots were randomly selected for each graph. They were each benchmarked five times per data structure and thread level. The mean results of the roots were then used to calculate the standard deviation.

Due to the similarity of the results, the majority of the graphs, shown earlier in Table 3.1, will be presented in the appendix but still might be referred to. The two presented graphs are cage15 and sparse200M as they differ both in density and the latter is randomly generated.

4.2.1 Errors Allowed

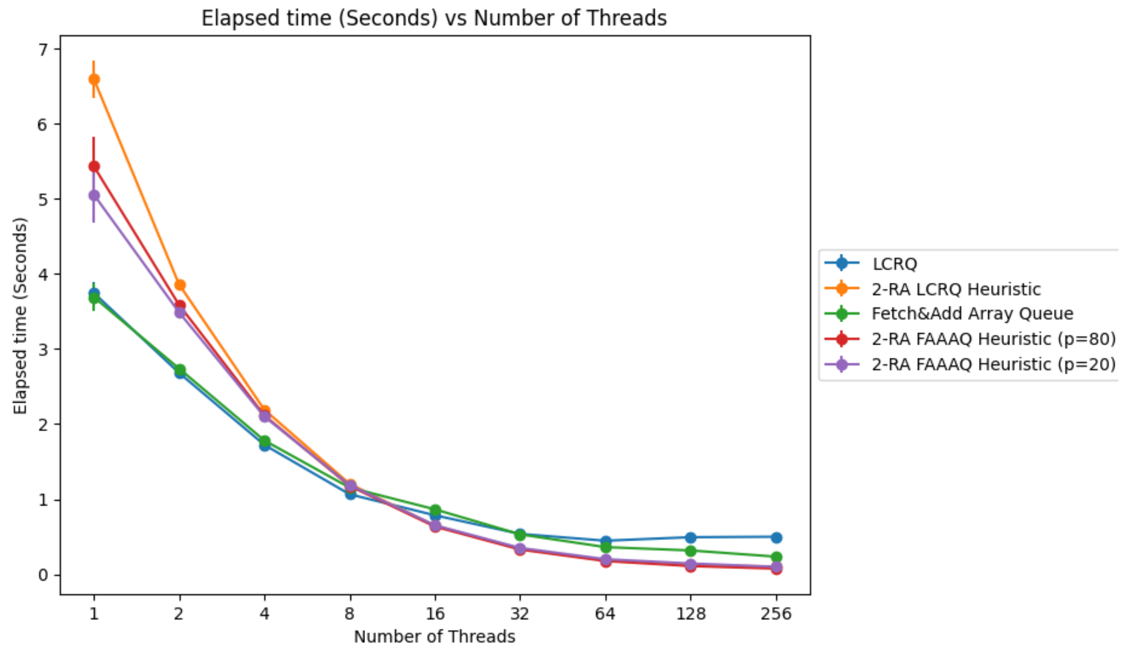
In Figures 4.11 a) and 4.12 a), we can see the elapsed time of the algorithm for two of the graphs. This illustrates a baseline that serves as the believed maximal possible speedup of the relaxed queues. The relaxed queues clearly outperform the non-relaxed ones after 8-16 threads. In fact, the different relaxed data structures are difficult to tell apart due to their similarity in runtime. However, we can see that FAAAQ in general outperforms LCRQ on high thread counts.

In many ways, the more interesting observations can be made in Figures 4.11 b) and 4.12 b). Here the differences in generated path lengths are illustrated. First of all, we can see that the relaxed queues on average generate longer paths than the non-relaxed queues. This is expected due to relaxation. Furthermore, we can note the differences between LCRQ and FAAAQ, as FAAAQ in general seems to generate longer paths. This is related to scheduling and most probably contention. An example of how contention could cause this behavior is that if many enqueue operations are slow, then the out-of-order behavior should be increased further as we would expect more randomness in the traversal. The initial thought was that this could be caused by enqueuers being preempted to a larger extent in FAAAQ than LCRQ. This is because, LCRQ will, as previously mentioned, move the tail forward if the head surpasses it which is not the case with FAAAQ. Measuring the number of failed dequeue operations, however, gave inconclusive results. FAAAQ had a higher number of failed dequeue operations per run than the other data structures but this could be explained by the fact that it should be faster at returning whenever the

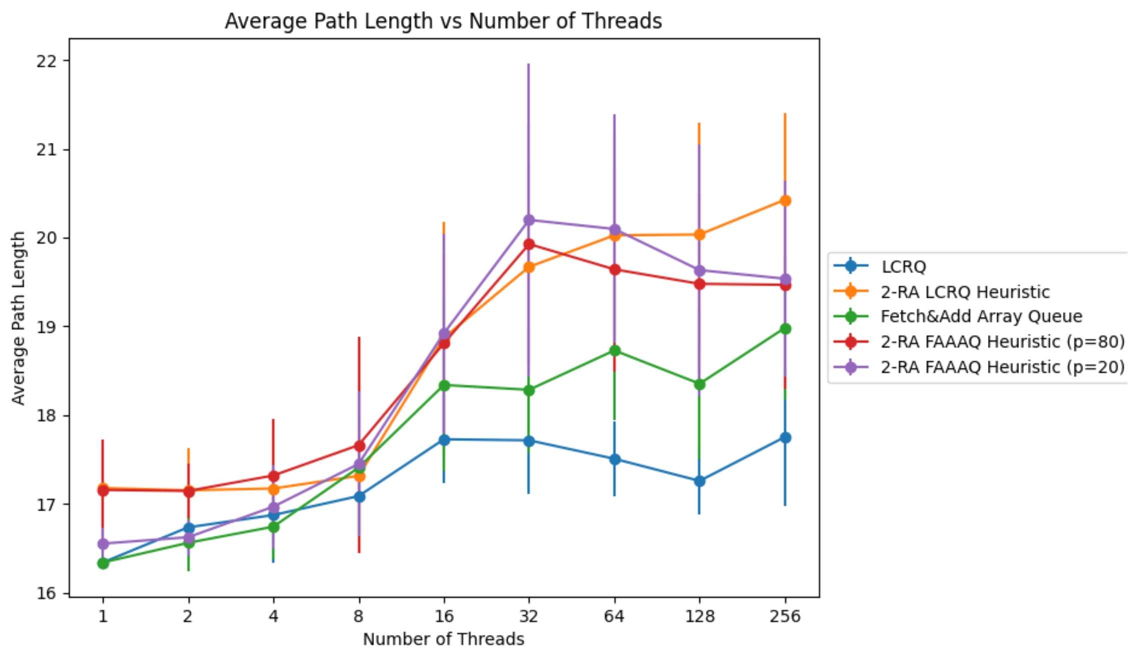
queue is empty due to the implementation. Furthermore, it was not possible to discern a relationship between the generated path lengths and the number of failed dequeues for FAAAQ. The difference in path lengths between the two concurrent queues is observed in all tested graphs.

In regards to the relaxed queues, much as in the case of the runtime, it is difficult to see any systematic differences in behavior. However, it is of interest that 2-RA FAAAQ with 20 partial queues, which should yield lower levels of relaxation, yields similar and sometimes longer path lengths than the 2-RA FAAAQ with 80 partial queues. One explanation could be that fewer partial queues increase the effects of scheduling due to contention. This could also be due to a smaller number of elements in the queue as we would not expect major differences in relaxation in that case.

Furthermore, it is important to mention the differences in path lengths that can be observed between the different graphs. In general, it seems like the effects of scheduling and relaxation are substantially higher in the random graphs compared to the non-random ones. This can be seen when comparing the path lengths at one thread versus higher threads.

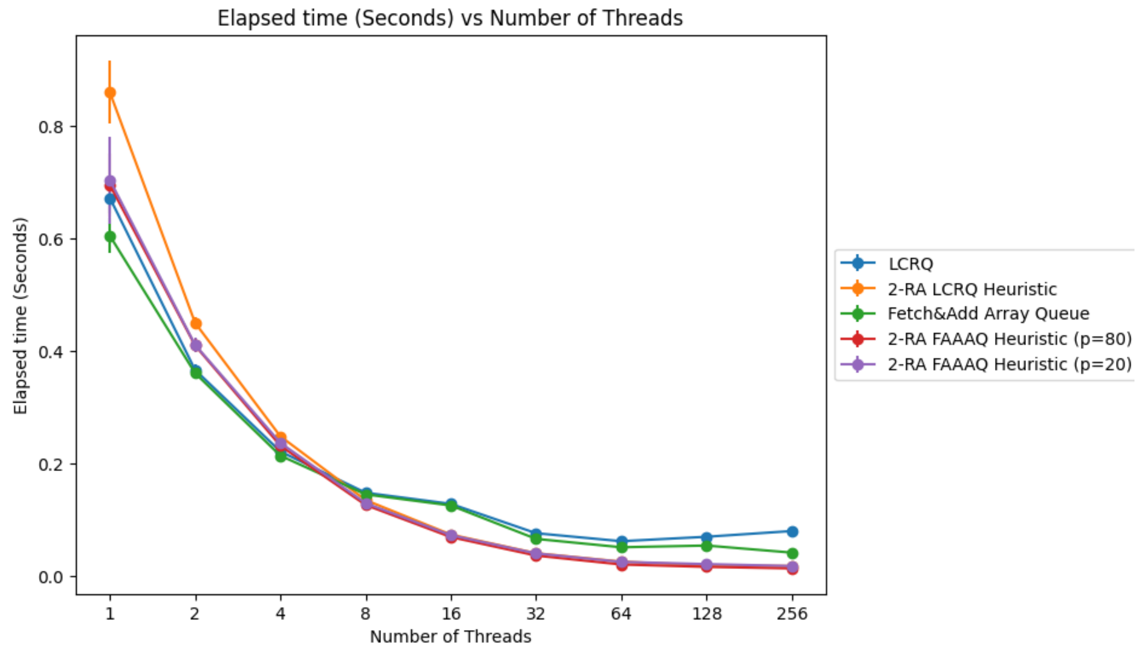


(a) Elapsed time

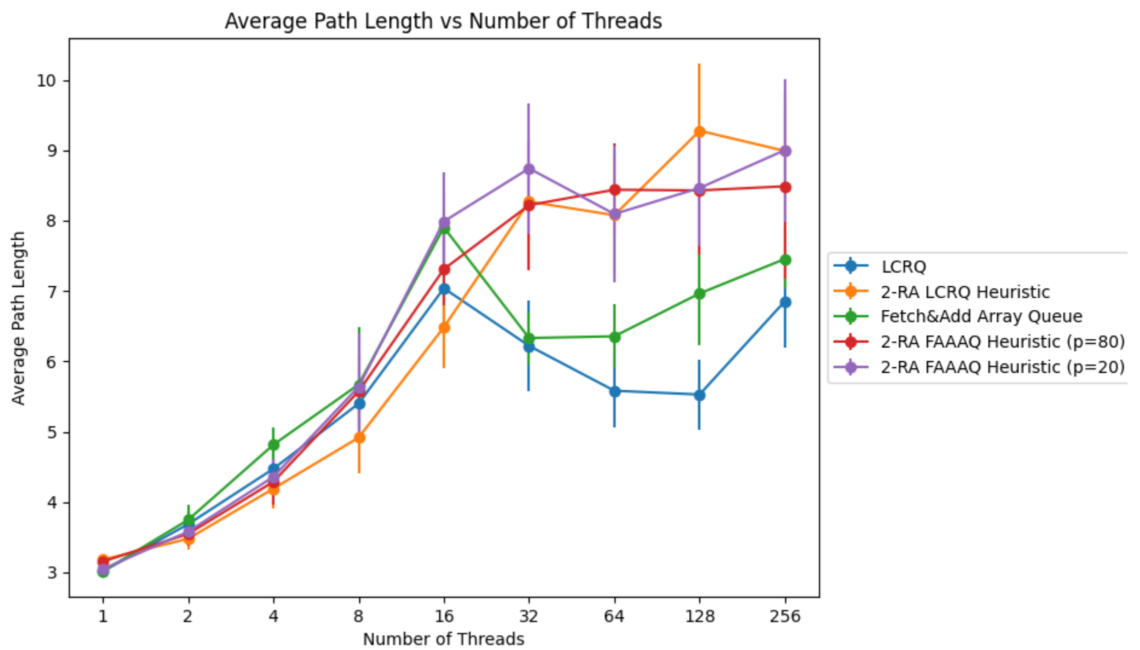


(b) Average path length

Figure 4.11: The elapsed time and average path length of the different data structures on cage15 when errors are allowed.



(a) Elapsed time



(b) Average path length

Figure 4.12: The elapsed time and the average path length of the different data structures on Sparse 200M when errors are allowed.

4.2.2 Unordered BFS

When no errors are allowed, the runtime results are very similar to the case whenever errors are allowed, see Figures 4.13 a), 4.14 a). The relaxed queues outperform the non-relaxed queues starting at 8-16 threads, being slower before that. We can also observe that a lower number of partial queues seems to decrease the elapsed time at low thread counts. After a certain thread count, the differences between the relaxed queues become difficult to discern. This was found to be the case for all the benchmarks. At high thread counts, there is no increase in performance for sparse200M but rather a decrease. As in the previous graphs, LCRQ performs worse than FAAAQ at high thread counts. We can also see that LCRQ outperforms FAAAQ at some thread levels in the randomly generated graphs to an extent that was not visible before. However, this is not visible in cage15 and in fact not the other non-generated graph audikw_1, see Figure A.8 a). In order to explain this, the work has to be investigated.

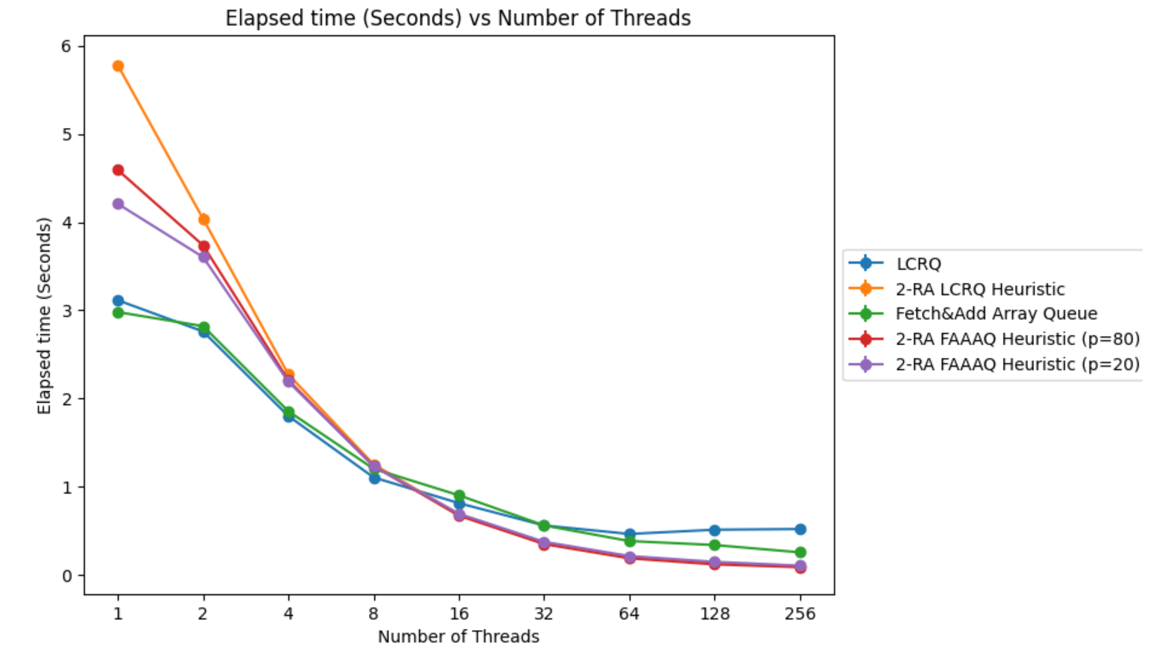
In Figures 4.13 b) and 4.14 b) the work done by the different data structures are shown. As a reminder, work is defined as the number of successful CAS operations. Here we can see that in general the work done is not directly comparable to the path lengths yielded in the previous benchmarks. In some of the graphs, for example sparse200M, we can see that FAAAQ has lower work until a certain point where it is comparable to the relaxed queues. In the case of cage15, FAAAQ surpasses the relaxed queues in work starting at low thread counts. This would also be seen in audikw_1, see Figure A.8 b), if not for the extreme increase in work for 2-RA LCRQ at 256 threads.

Moreover, we can observe that LCRQ in general has the lowest amount of work but it seems to increase substantially at 256 threads for all of the tested graphs. The differences in work between LCRQ and FAAAQ are believed to be the cause of the superior performance of LCRQ at some thread counts. As in the case when we allowed errors, we would expect 2-RA FAAAQ with 20 partial queues to waste less work due to relaxation than its counterpart with 80 partial queues. This seems to be the case at low thread levels and in sparse200M but the opposite is true in cage15. In general, considering all tested graphs, it seems to in general waste less work than its counterpart with 80 partial queues.

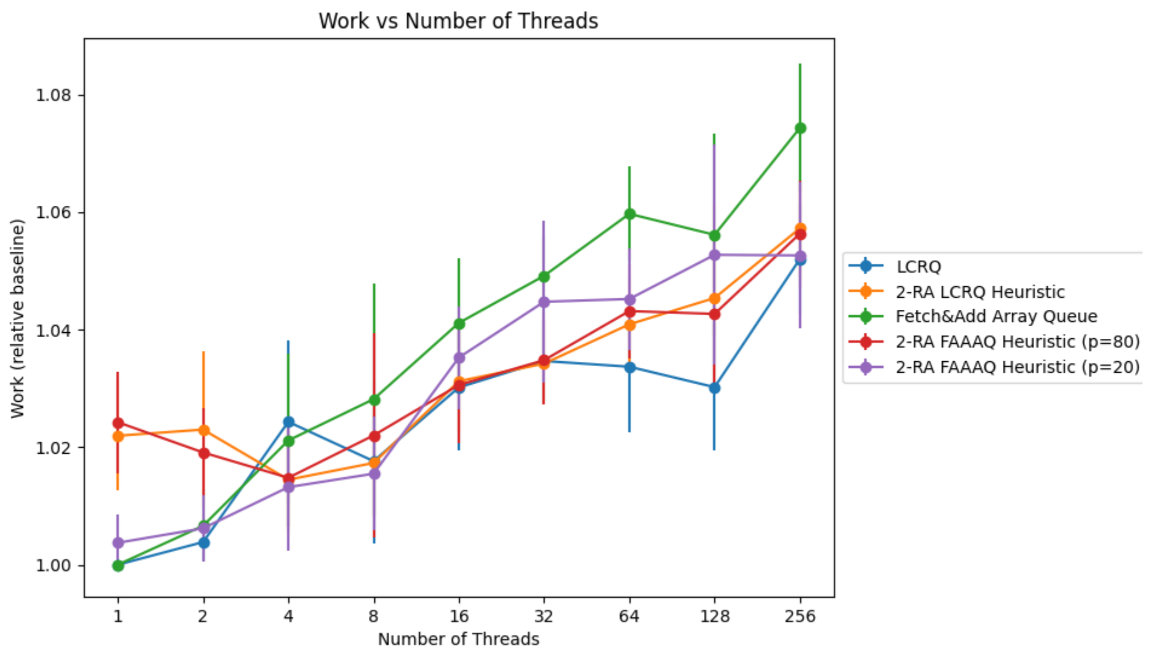
Furthermore, we can observe the differences in the work relative to the baseline. For the randomly generated graphs, there is a considerable amount of wasted work. In the case of sparse200M, the work done is over 300% larger than the baseline. For the other randomly generated graphs, see Figures A.9 b) and A.10 b), we can see the same phenomenon, even though the difference is somewhat smaller. However, in the case of the preexisting graphs, the difference in work compared to the baseline is insignificant compared to that of the randomly generated graphs. This is reminiscent of the previously examined path lengths but to a more noticeable extent. We believe this explains the differences in the performance of LCRQ, where there was only a noticeable performance increase in the randomly generated graphs as opposed to the case of the preexisting graphs, where the difference in work is relatively insignificant. This means that in some graphs, wasted work does not seem to substantially affect

the runtime of the bfs-algorithm. This is further supported by the differences in runtime between when errors are allowed and not. For cage15 at 256 threads the algorithm where errors are allowed is roughly 10 milliseconds faster on average than that when work can be wasted. Considering that the elapsed time is around 0.2 seconds this is a small difference. In contrast, for sparse200M, this difference is over 200 milliseconds, when the runtime is on average around 30 milliseconds for when errors are allowed. The stark difference can probably be explained by our method of graph generation.

During further investigation, it was found that in the randomly generated graphs, the differences in shortest paths for the different nodes were extremely small in comparison to the non-generated graphs. We suspect that this might lead to more feasible paths that could be taken to a node erroneously due to scheduling and relaxation.

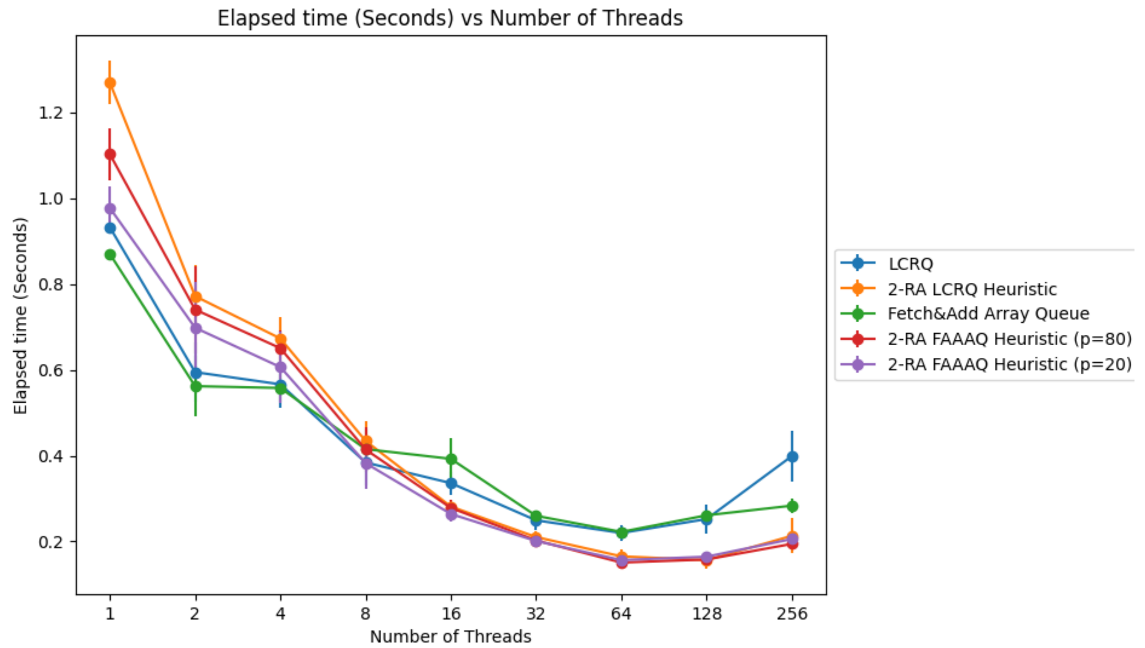


(a) Elapsed time

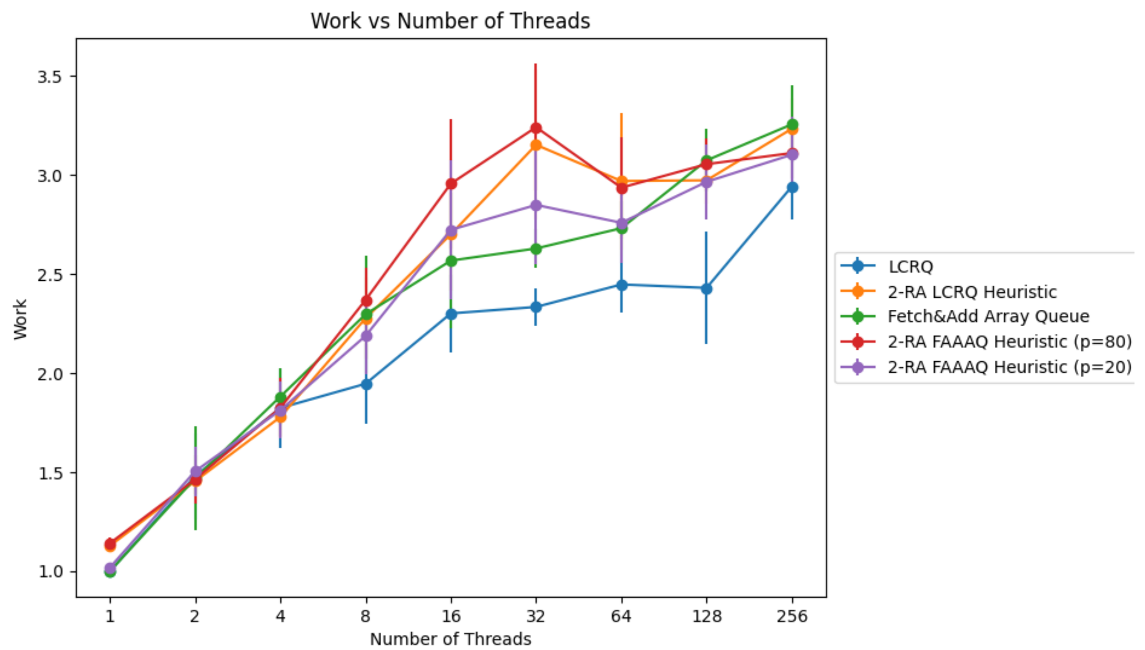


(b) Total work

Figure 4.13: The elapsed time and total work, in relation to the baseline, of the different data structures on cage15 during unordered BFS.



(a) Elapsed time



(b) Total work

Figure 4.14: The elapsed time and total work of the different data structures on sparse200M during unordered BFS.

4.3 General Discussion

With the previous results, a general discussion about them can be had and the research questions can be answered.

4.3.1 Feasibility of Different Partial Queues

As previously seen, d-RA LCRQ generally has worse performance than d-RA FAAAQ. It is entirely possible that increased performance is to be had but in many regards LCRQ is a queue that is not well adapted to being used in a distributed setting. First of all, extensive modifications had to be done in order for it to function as intended. Secondly, the arguably biggest advantage of the queue, its proficiency for cache locality, is almost entirely eliminated when using the queue in this manner. On the other hand, FAAAQ is in our opinion more adapted for this purpose as we view the cyclic behavior of LCRQ as an unnecessary complication when used in a distributed manner. Furthermore, the implementation of the d-RA LCRQ is more complex than that of the other versions of d-RA, see Table 4.1. Important to mention however is that the number of lines of code can not be regarded as optimal as preprocessor directives have been extensively used for different functionalities which will increase the total lines of code. Moreover, the used hazard pointer implementation is not counted as it is not intrinsic to the data structures and other implementations can be used.

4.3.2 Feasibility of Relaxed Queues

From our results, the assumption can be made that relaxed FIFO queues can be regarded as generally superior to concurrent queues in throughput at high thread counts. However, this is not always the case as seen in Figure A.11, where a producer-consumer benchmark is run with one producer and a varying number of consumers. In this benchmark, the relaxed FIFO queues perform poorly and even the Michael-Scott queue achieves stronger performance. Furthermore as shown in [1], setting a wait between operations can substantially affect contention and thereby the performance of a relaxed data structure.

Algorithm	Lines of Code
d-RA (without partial queues)	188
d-RA Michael-Scott	331
d-RA FAAAQ	364
d-RA LCRQ	490

Table 4.1: The total number of lines of code, excluding comments and blank spaces, of the different versions of d-RA.

4.3.3 Sources of Error

One might highlight the different reclamation rates of hazard pointers as a source for error but even if memory reclamation was eliminated, the Michael-Scott-based d-RA had substantially weaker performance. Furthermore, the fact that this could be regarded as a possible source of error further highlights the superiority of the faster partial queues as through their design, memory reclamation is a less frequent occurrence.

Furthermore, as different systems for handling memory were used for the Bag and the 2Dd-queue compared to the distributed queues, one could question the validity of the comparison. Still, considering the substantial performance differences we deem that this is very unlikely to affect the results in a meaningful manner. Moreover, it would be possible to substantially increase the throughput measurements of the 2Dd-queue substantially by increasing the relaxation bound. However, that would make the comparison to other data structures difficult due to the higher relaxation. We expect that in the sequential alternating benchmark without any prefill it would be able to maintain low relaxation with a substantially higher throughput due to its ability to work thread locally. Still, the decision was made to use the same parameters for all the benchmarks for simplicity. In general, optimizations of different parameters such as segment size, and number of partial queues have not been done for the different data structures.

It is also important to discuss the relaxation measurements. One could argue that the lower number of elements used for the relaxation benchmarks might undermine the results. As previously discussed, we have found one such possible example, in the case of 1-RA LCRQ in the sequential alternating benchmark with 0% prefill. However, we very much regard that measurement as an outlier, and the 1-RA algorithms are only used for comparisons but in theory, other data structures could be impacted in a similar manner. For example, memory reclamation might not occur for the Michael-Scott queue. However, we deem that it would be unlikely that the lack of such a process would impact the queue in a negative manner. Secondly, when comparing the relaxation results of the Michael-Scott d-RA with [1], they are relatively similar, although they did not use any sequential alternating benchmarks. All in all, we deem that it is very unlikely that our method of measuring relaxation would have any substantial impact on our conclusions.

As the duration of the benchmarking suite is substantial, some results have been gathered at different points in time. It is possible that some results therefore were benchmarked during different configurations of the computer that was used for benchmarking. Some variance between different runs has also been noted in both the evaluation of the queues and in the BFS. In the case of the evaluation of the queues, this might impact the relative ordering of data structures that are relatively close in performance but it does not affect our conclusions in regard to our research questions due to the strong performance of the FAAAQ and LCRQ-based d-RA. Furthermore, due to the number of benchmarks, we deem that this effect on a systematic level is relatively negligible. In the case of BFS, this effect was more noticeable, where sometimes the execution of the test program was considerably

faster or slower for all data structures. This was handled by benchmarking all data structures together so that if any such effect appears, it affects all tested data structures. This does not eliminate this phenomenon but it makes it small enough to not affect our general conclusions. We believe all of this might be caused by issues related to either Scal itself or the configuration of it on the benchmarking computer.

4.3.4 Research Questions

RQ 1: How does the d-RA algorithm scale with faster partial queues?

From the results gathered the performance of the faster partial queues seem to translate to the d-RA algorithm. The fetch-and-add-based partial queues outperform the Michael-Scott d-RA in every benchmark in throughput. In fact, the queues outperform all state-of-the-art implementations tested except the bag.

RQ 2: How does a progress-based heuristic affect the d-RA algorithm?

The progress-based heuristic generally outperforms the length-based heuristic in both throughput and relaxation. It seems to have a larger relaxation distance whenever the partial queues are relatively empty but as soon as the queues are generally non-empty, the progress-based heuristic is far superior. Furthermore, it was shown that it was possible to limit this relaxation by decreasing the number of partial queues while still maintaining strong performance. Furthermore, we deem that the progress-based heuristic is necessary for the use of the novel distributed queues considering their measured relaxation in some benchmarks.

RQ 3: Is unordered breadth-first search a feasible application for a relaxed FIFO queue?

It was found that for relatively sparse graphs that the costs of relaxation do not outweigh the performance increases of a relaxed data structure in unordered BFS. Therefore, we believe that relaxed FIFO queues can in specific circumstances be used for unordered BFS. However, we expect that if substantial work is done to a node, in between the search operations, the possible performance increases will diminish as contention is lowered. Furthermore, it is not apparent that the FAAAQ and the LCRQ are the most suited concurrent queues to use in this comparison as there may be faster queues for this type of benchmark, especially considering the differences between LCRQ and FAAAQ in work. Therefore, we can not answer this question definitely but we regard the results with cautious optimism.

5

Conclusion

During this thesis, an improved version of the d-RA algorithm [1] was created. It was found that the data structure scales well with fetch-and-add-based queues. This is especially true with many threads where Michael-Scott d-RA is shown to scale poorly due to contention.

Furthermore, a novel heuristic was introduced that generally increased performance and decreased relaxation. These results were particularly apparent for the FAA-based queues, where relaxation scales with contention with the length-based heuristic while it is much more limited with the progress-based one. Furthermore, the novel queues with the progress-based heuristic outperformed state-of-the-art implementations with an increase in throughput to the original d-RA, which at times exceeds 400%. At the same time, relaxation was considerably lower whenever the queues were relatively filled and could be decreased to comparable levels to the Michael-Scott d-RA by limiting the number of partial queues while still outperforming it in throughput.

Additionally, a possible application was found for relaxed FIFO queues in unordered BFS, where the relaxed queues substantially outperformed concurrent queues in sparse graphs at higher thread counts.

5.1 Future Work

As mentioned, fetch-and-add-based queues can perform poorly whenever there are few enqueuers and many dequeuers. Therefore, it would be interesting to use a queue like [16], which does not have this same shortcoming, as a partial queue. In fact, it is reminiscent of the FAAAQ but with the difference that it is wait-free, meaning there are guarantees concerning the progress of individual threads.

Another interesting area where potential improvements could be made is to optimize d-RA to utilize cache locality. One could do this by assigning certain threads to specific partial queues in some scenarios, reminiscent of how the 2Dd-queue works. One would expect that this would increase relaxation but the question is if this could improve performance to the extent that any extra relaxation could be mitigated through a decrease in the number of partial queues.

Moreover, one could investigate if it would be possible to change the used heuristic at runtime. As previously seen, the progress-based heuristic can not be regarded

as superior to the length-based one, at least not relaxation-wise, when the data structure is relatively empty. Therefore an idea would be to use the length-based heuristic until the data structure has a set number of elements. However, it is possible that the different heuristics will not synergize well and lead to an increase in relaxation and a decrease in performance. In same manner it could be interesting to experiment with the number of partial queues used during runtime. One could attempt to modify the d-RA algorithm to only use a subset of the total partial queues if contention is low. One idea is to use the number of failed dequeue-attempts for the partial queues as a heuristic for this. If a threshold is reached, the number of partial queues used increase or decrease. This could potentially lead to lower relaxation without a substantial impact on performance. The idea of being able to change the nature of the relaxation at runtime has been studied previously in the context of the 2D-queue [36].

Additionally, one could experiment with the fetch-and-add-based partial queues to see if it would be possible to track only successful operations done to them. At present an operation can change an index multiple times even through failures. Rectifying this would enable the partial queues to be used to bound the relaxation and make it possible to use them in [20]. However, by doing this we anticipate a substantial performance decrease as this would most likely mean an increase in contention through some new needed shared state.

Furthermore, it would be interesting to see a mathematical analysis of the novel progress-based heuristic in a similar manner as in [29], where the expected relaxation was analyzed. We anticipate however that this will be a complicated proof to write.

Further research can be made in regards to unordered BFS and relaxed FIFO queues. It would be interesting if our results could be verified and to further study the relationship between wasted work and relaxation. As mentioned, it is not clear that the queues compared against are the best suited for unordered BFS, it would be interesting to compare the performance of the relaxed queues to that of a framework such as Galois [37], that was used in [7] to implement a parallel BFS. Also, an investigation could be done to see how a relaxed FIFO queue would perform in other BFS-related algorithms where the unordered behavior would be acceptable. Further work could also be done to evaluate the performance of relaxed FIFO queues on substantially denser graphs than those that were tested in this thesis.

Finally, more potential applications for relaxed FIFO queues could be investigated. For example, it might be possible to use d-RA in a web server to handle requests. There are two identified potential problems with this approach. First of all, there would need to be substantial contention for this approach to be feasible and it is questionable if the latency accompanied with this would be reasonable. Secondly, as d-RA has no bounds when it comes to relaxation, it would be possible for some requests to not be served. Perhaps, one could implement a timer to guarantee that all requests are served within a certain time span. Once the timer runs out, a new list of partial queues could be initiated. Enqueuers would begin work on the new partial queues, while dequeuers continue on the old ones until they have been emptied. Once the old partial queues are empty, it is guaranteed that all

requests made before the timer ran out have been completed. Otherwise, it might be possible to use an established relaxed queue where there are guarantees in regards to relaxation such as the 2Dd-queue.

Bibliography

[1] A. Haas, M. Lippautz, T. A. Henzinger, *et al.*, "Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '13, Ischia, Italy: Association for Computing Machinery, 2013, ISBN: 9781450320535. DOI: 10.1145/2482767.2482789. [Online]. Available: https://doi.org/10.1145/2482767.2482789.

[2] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova, "Quantitative relaxation of concurrent data structures," *SIGPLAN Not.*, vol. 48, no. 1, pp. 317–328, Jan. 2013, ISSN: 0362-1340. DOI: 10.1145/2480359.2429109. [Online]. Available: https://doi.org/10.1145/2480359.2429109.

[3] C. M. Kirsch, H. Payer, H. Röck, and A. Sokolova, "Performance, scalability, and semantics of concurrent fifo queues," in *Algorithms and Architectures for Parallel Processing*, Y. Xiang, I. Stojmenovic, B. O. Apduhan, G. Wang, K. Nakano, and A. Zomaya, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 273–287, ISBN: 978-3-642-33078-0.

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

[5] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," *SIGPLAN Not.*, vol. 48, no. 8, pp. 103–112, Feb. 2013, ISSN: 0362-1340. DOI: 10.1145/2517327.2442527. [Online]. Available: https://doi.org/10.1145/2517327.2442527.

[6] P. Ramalhete, *Faaarrayqueue - mpmc lock-free queue (part 4 of 4)*, https://concurrencyfreaks.blogspot.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html, Accessed: 2023-12-03.

[7] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered and unordered algorithms for parallel breadth first search," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10, Vienna, Austria: Association for Computing Machinery, 2010, pp. 539–540, ISBN: 9781450301787. DOI: 10.1145/1854273.1854341. [Online]. Available: https://doi.org/10.1145/1854273.1854341.

[8] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas, "A lock-free algorithm for concurrent bags," in *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11, San Jose, California, USA: Association for Computing Machinery, 2011,

pp. 335–344, ISBN: 9781450307437. DOI: 10.1145/1989493.1989550. [Online]. Available: https://doi.org/10.1145/1989493.1989550.

[9] J. Turek, D. Shasha, and S. Prakash, "Locking without blocking: Making lock based concurrent data structure algorithms nonblocking," in *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1992, pp. 212–222.

[10] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "X86-tso: A rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010, ISSN: 0001-0782. DOI: 10.1145/1785414.1785443. [Online]. Available: https://doi.org/10.1145/1785414.1785443.

[11] P. Ramalhete, *Lcrq in c++ with hazard pointers*, https://concurrencyfreaks.blogspot.com/2017/01/lcrq-in-c-with-hazard-pointers.html, Accessed: 2023-12-01.

[12] Intel, *Intel 64 and ia-32 architectures software developers manual*, 2nd ed., Santa Clara, California, USA, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (visited on 03/14/2024).

[13] P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems," in *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '01, Crete Island, Greece: Association for Computing Machinery, 2001, pp. 134–143, ISBN: 1581134096. DOI: 10.1145/378580.378611. [Online]. Available: https://doi.org/10.1145/378580.378611.

[14] M. Hoffman, O. Shalev, and N. Shavit, "The baskets queue," in *Principles of Distributed Systems*, E. Tovar, P. Tsigas, and H. Fouchal, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 401–414, ISBN: 978-3-540-77096-1.

[15] A. Gidenstam, H. Sundell, and P. Tsigas, "Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency," in *Principles of Distributed Systems*, C. Lu, T. Masuzawa, and M. Mosbah, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 302–317, ISBN: 978-3-642-17653-1.

[16] C. Yang and J. Mellor-Crummey, "A wait-free queue as fast as fetch-and-add," *SIGPLAN Not.*, vol. 51, no. 8, Feb. 2016, ISSN: 0362-1340. DOI: 10.1145/3016078.2851168. [Online]. Available: https://doi.org/10.1145/3016078.2851168.

[17] R. Romanov and N. Koval, "The state-of-the-art lcrq concurrent queue algorithm does not require cas2," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '23, Montreal, QC, Canada: Association for Computing Machinery, 2023, pp. 14–26, ISBN: 9798400700156. DOI: 10.1145/3572848.3577485. [Online]. Available: https://doi.org/10.1145/3572848.3577485.

[18] A. Castañeda, S. Rajsbaum, and M. Raynal, *Relaxed queues and stacks from read/write operations*, 2020. arXiv: 2005.05427 [cs.DC].

[19] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–

492, Jul. 1990, ISSN: 0164-0925. DOI: 10.1145/78969.78972. [Online]. Available: https://doi.org/10.1145/78969.78972.

[20] A. Rukundo, A. Atalar, and P. Tsigas, "Monotonically relaxing concurrent data-structure semantics for increasing performance: An efficient 2d design framework," en, 2019. DOI: 10.4230/LIPICS.DISC.2019.31. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/11338/.

[21] G. Kappes and S. V. Anastasiadis, "A family of relaxed concurrent queues for low-latency operations and item transfers," *ACM Trans. Parallel Comput.*, vol. 9, no. 4, Dec. 2022, ISSN: 2329-4949. DOI: 10.1145/3565514. [Online]. Available: https://doi.org/10.1145/3565514.

[22] G. Blanchet and B. Dupouy, *Computer Architecture.* John Wiley & Sons, Incorporated, 2012, ISBN: 9781118577783. [Online]. Available: https://ebookcentral.proquest.com.

[23] A. Mazouz, S.-A.-A. Touati, and D. Barthou, "Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on intel architectures," in *2011 International Conference on High Performance Computing Simulation*, 2011, pp. 273–279. DOI: 10.1109/HPCSim.2011.5999834.

[24] T. Liu and X. Liu, "Cheetah: Detecting false sharing efficiently and effectively," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16, Barcelona, Spain: Association for Computing Machinery, 2016, pp. 1–11, ISBN: 9781450337786. DOI: 10.1145/2854038.2854039. [Online]. Available: https://doi.org/10.1145/2854038.2854039.

[25] W. J. Bolosky and M. L. Scott, "False sharing and its effect on shared memory performance," in *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, ser. Sedms'93, San Diego, California: USENIX Association, 1993, p. 3.

[26] N. Cohen, "Every data structure deserves lock-free memory reclamation," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. DOI: 10.1145/3276513. [Online]. Available: https://doi.org/10.1145/3276513.

[27] M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, 2004. DOI: 10.1109/TPDS.2004.8.

[28] 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.

[29] 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.

[30] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in

Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures, 2010, pp. 303–314.

[31] P. Ramalhete, *Michaelscottqueue*, `https://github.com/pramalhe/ConcurrencyFreaks/blob/master/CPP/queues/MichaelScottQueue.hpp`, Accessed: 2023-05-07.

[32] P. Ramalhete, *Hazardpointers*, `https://github.com/pramalhe/ConcurrencyFreaks/blob/master/CPP/queues/HazardPointers.hpp`, Accessed: 2023-06-13.

[33] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in linux," Jan. 2002. [Online]. Available: `https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf`.

[34] G. Karypis and V. Kumar, "Metisa software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing ordering of sparse matrices," Jan. 1997.

[35] 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`.

[36] K. von Geijer and P. Tsigas, *How to relax instantly: Elastic relaxation of concurrent data structures*, 2024. arXiv: `2403.13644 [cs.DS]`.

[37] 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`.

A

Appendix 1

A.1 Complete graphs

The following graphs are the complete versions of those presented in the Results section. They are included in the appendix as they do not show sufficient detail for the main text.

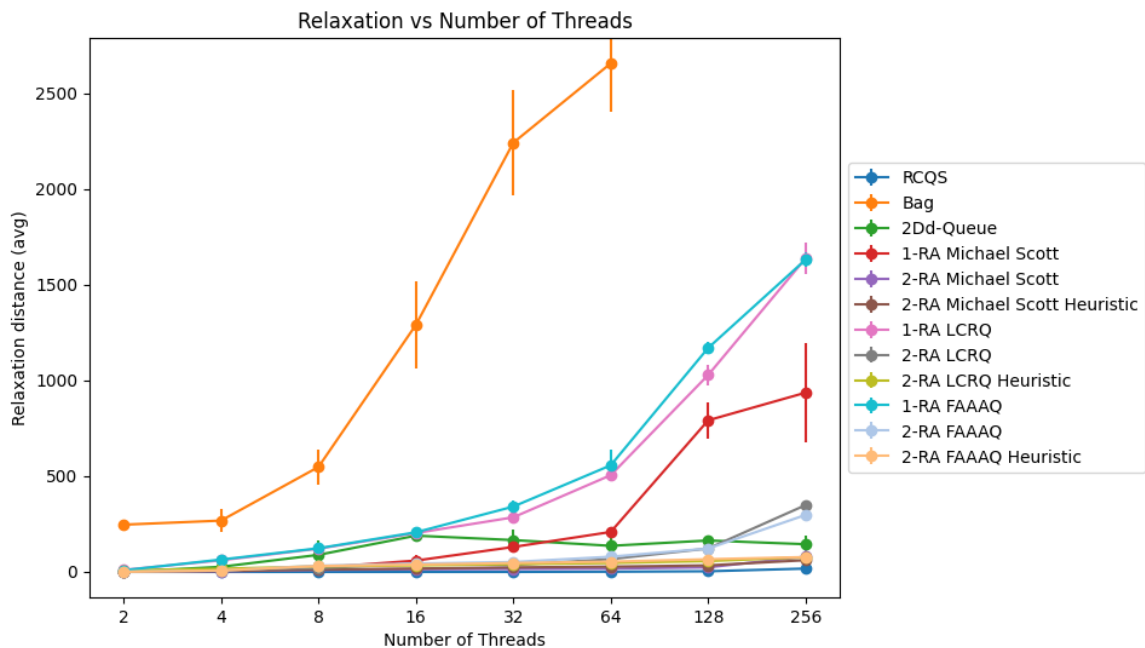


Figure A.1: Relaxation results of the producer-consumer benchmark. Lower is better.

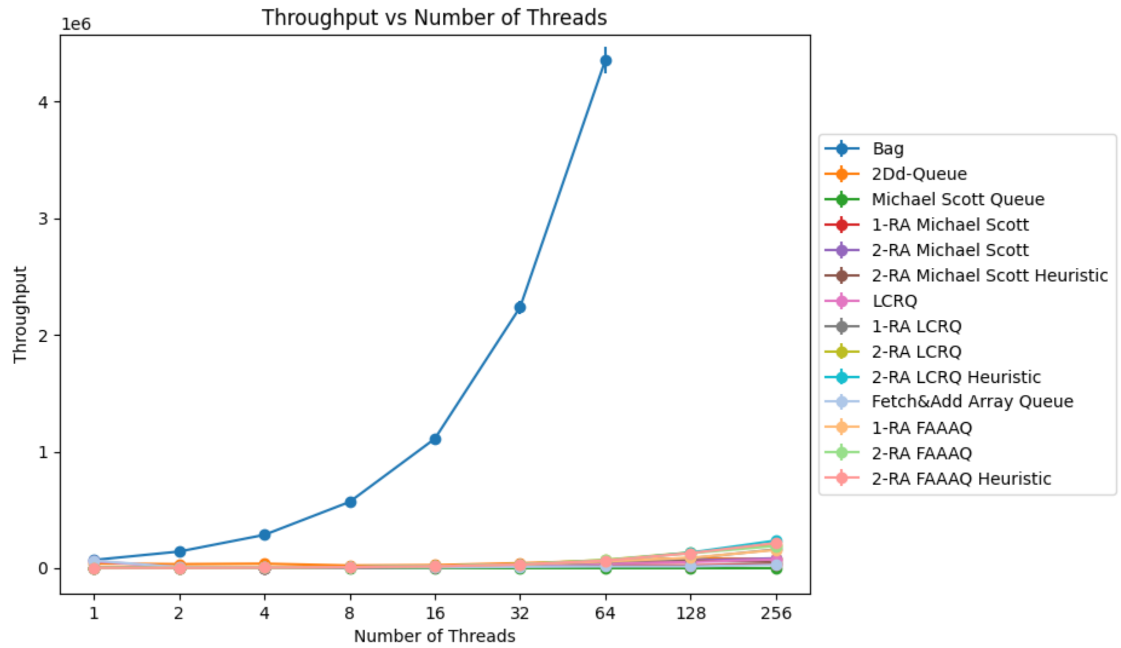


Figure A.2: Throughput results of the sequential alternating benchmark with 0% prefill. Higher is better.

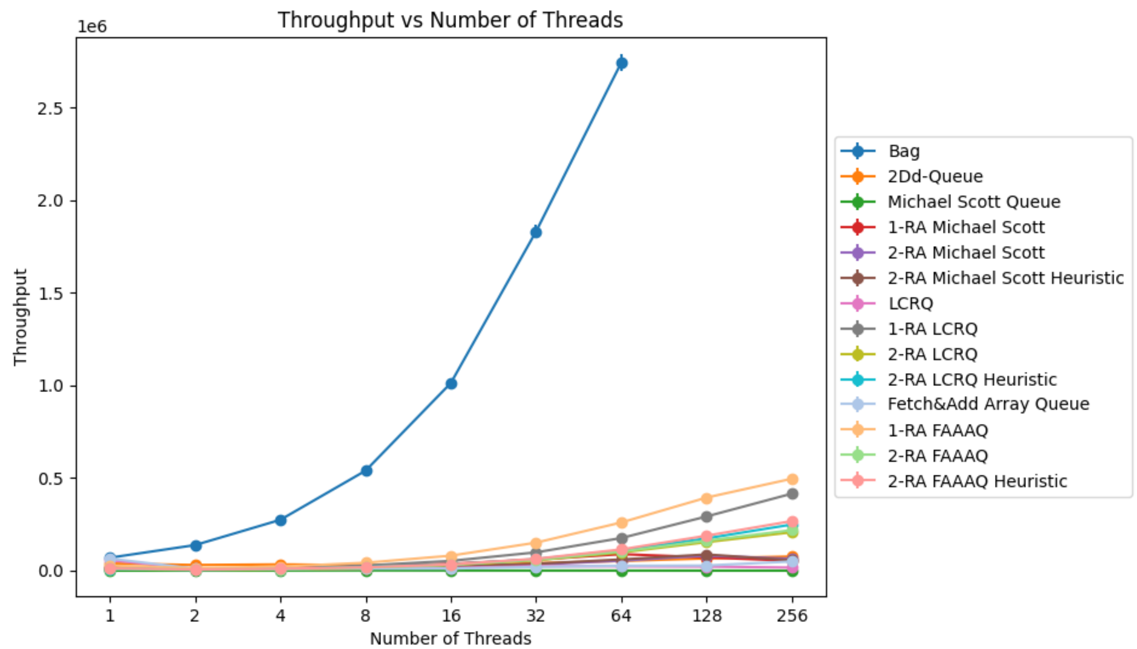


Figure A.3: Throughput results of the sequential alternating benchmark with 1% prefill. Higher is better.

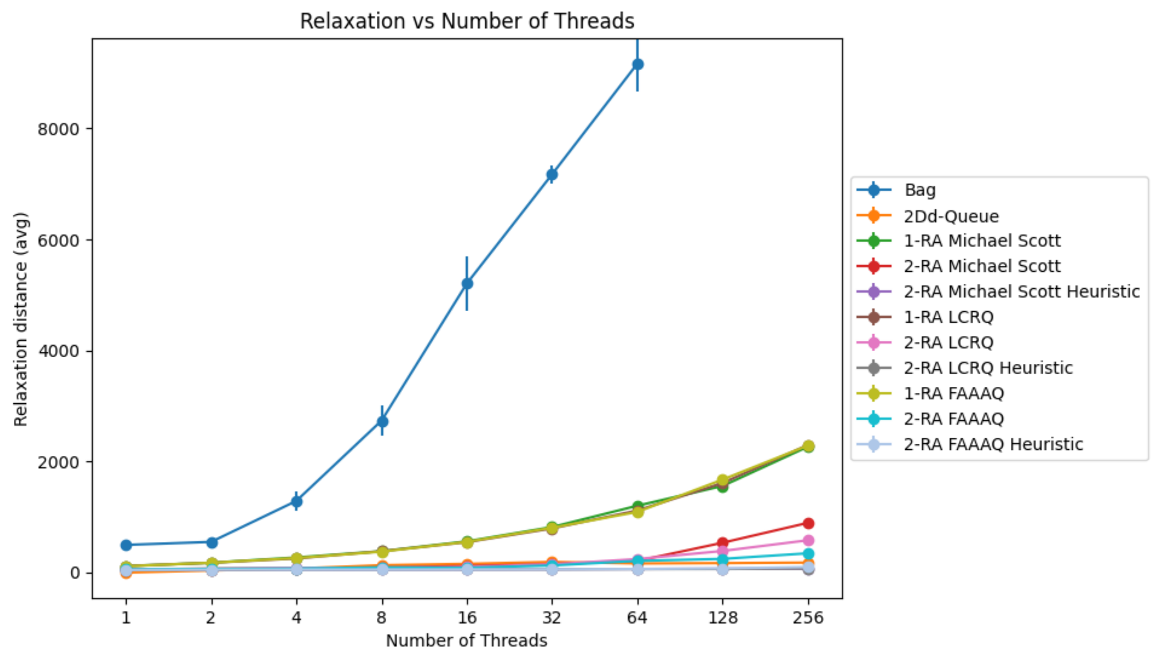
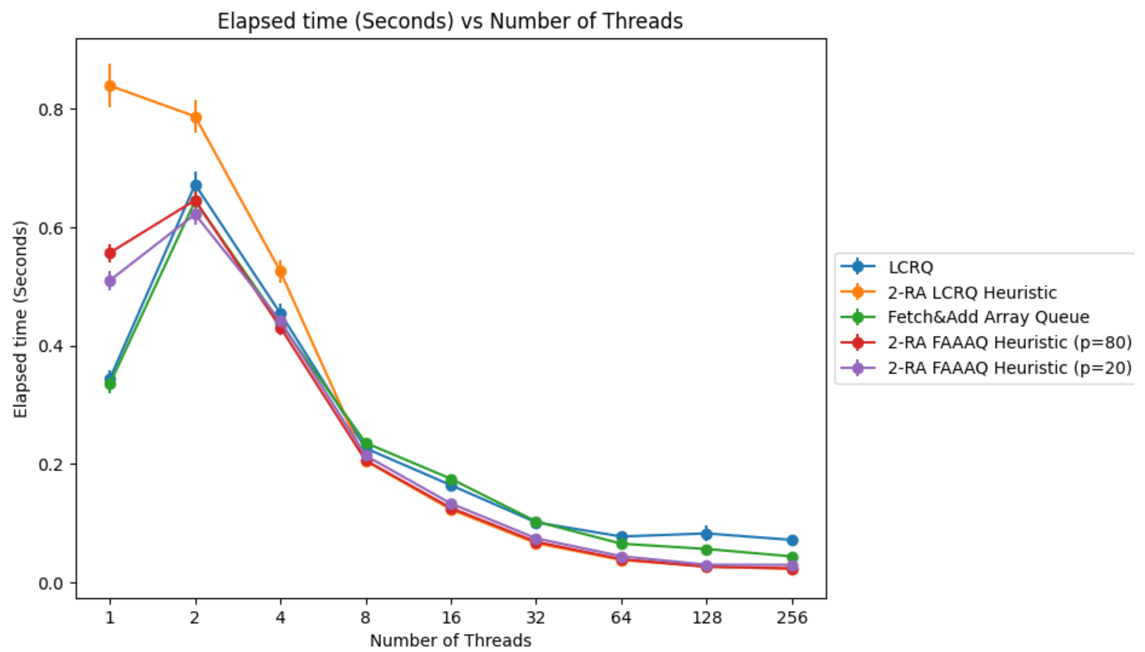


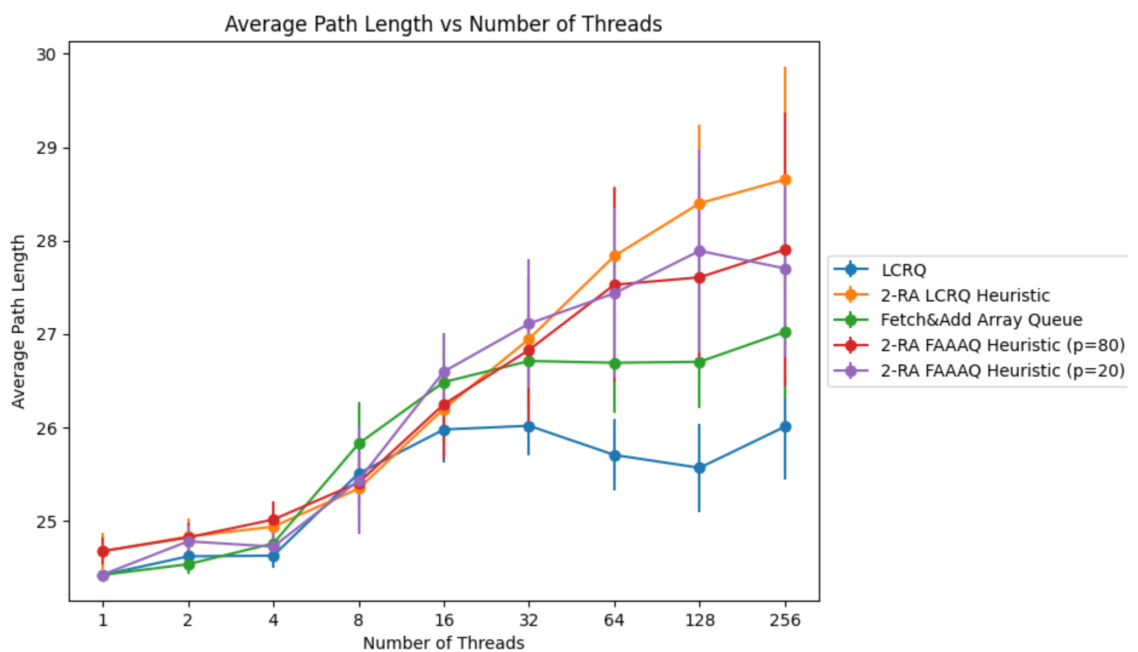
Figure A.4: Relaxation results of the sequential alternating benchmark with 100% prefill. Lower is better.

A.2 Supplementary graphs

Here graphs are presented that were not deemed essential enough to include in the text.

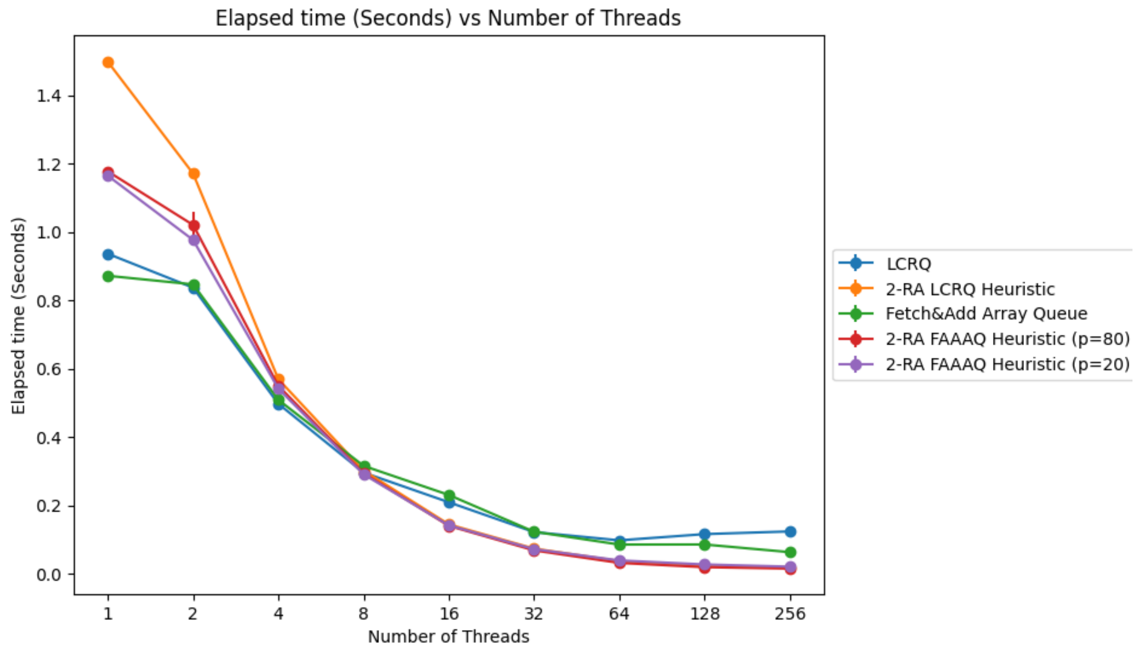


(a) Elapsed time

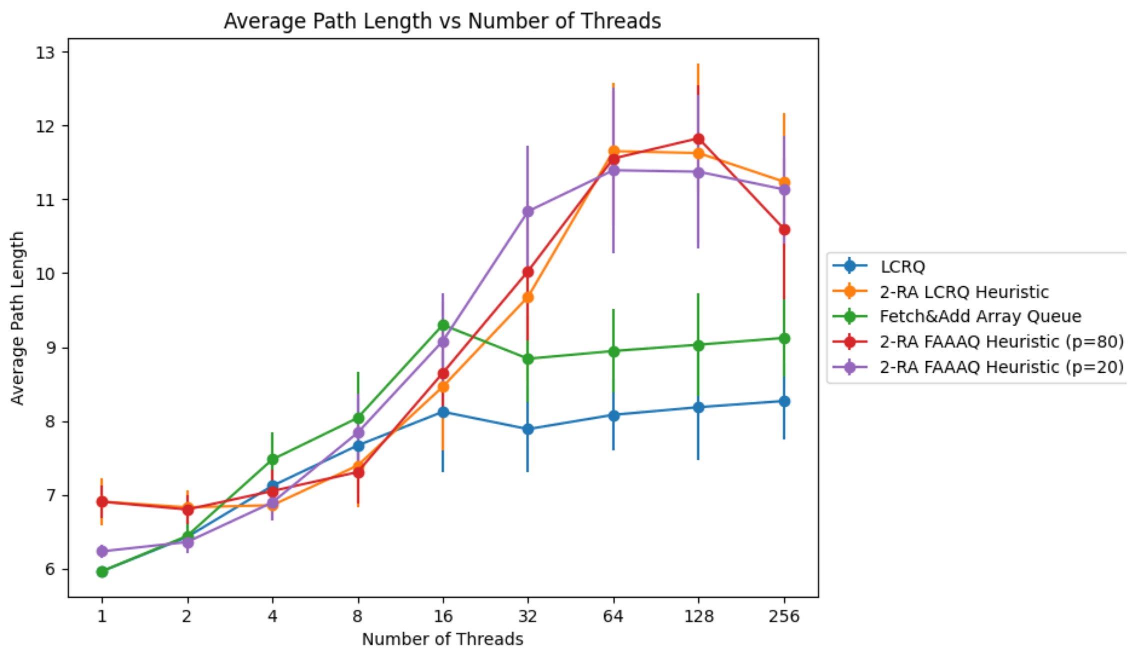


(b) Average path length

Figure A.5: The elapsed time and average path length of the different data structures on audikw_1 when errors are allowed.

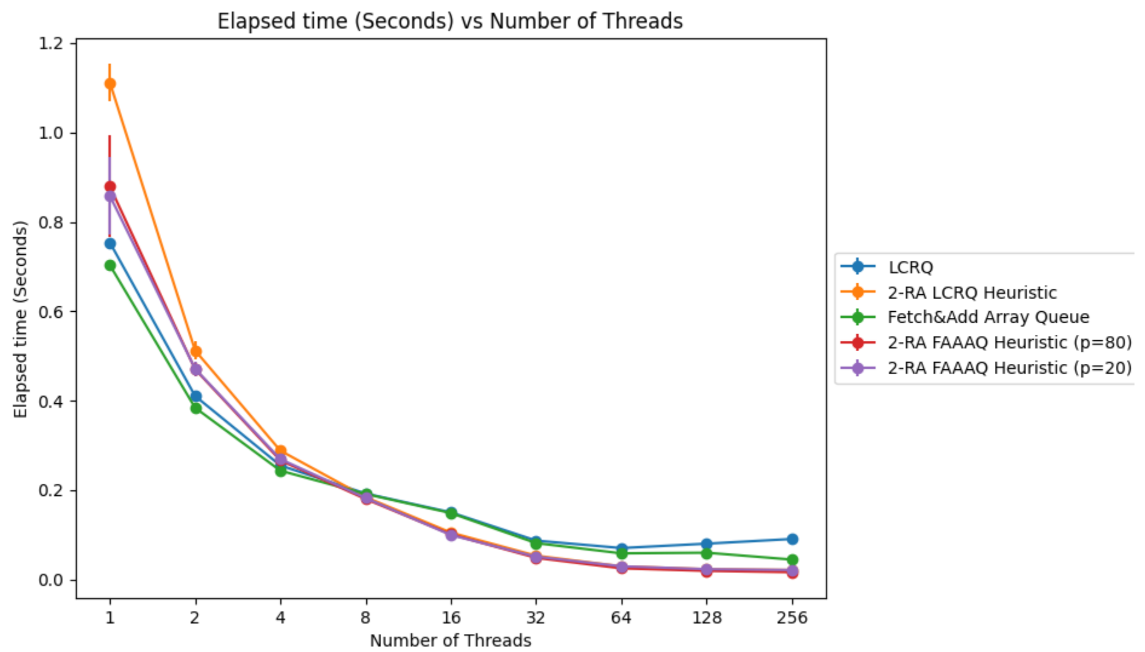


(a) Elapsed time

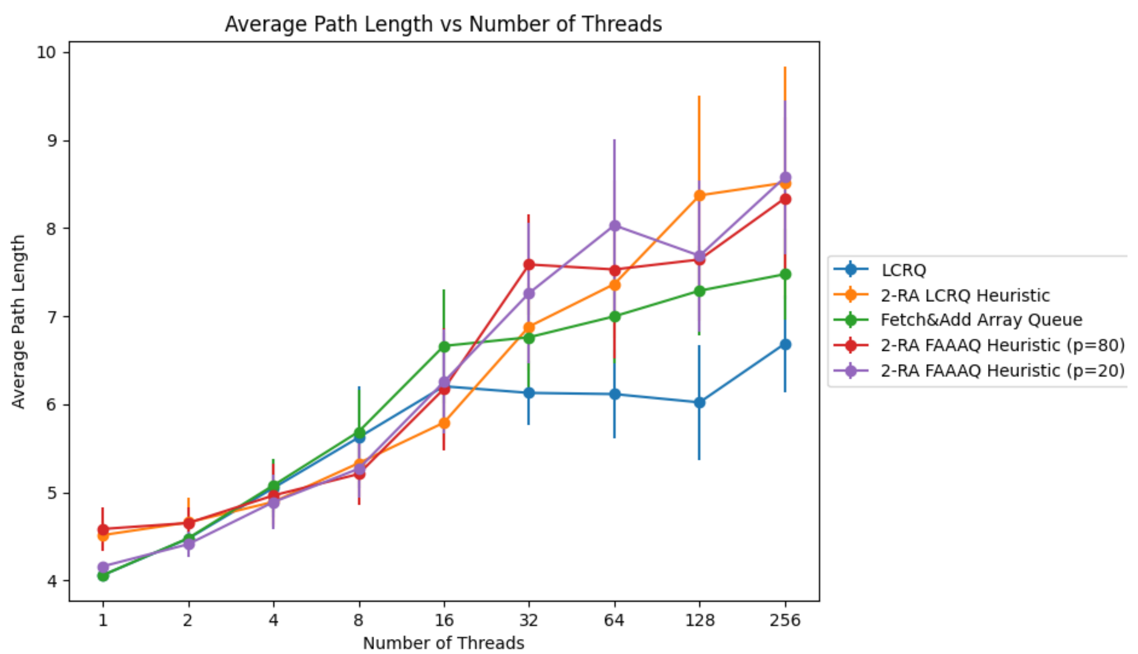


(b) Average path length

Figure A.6: The elapsed time and average path length of the different data structures on sparse10M when errors are allowed.

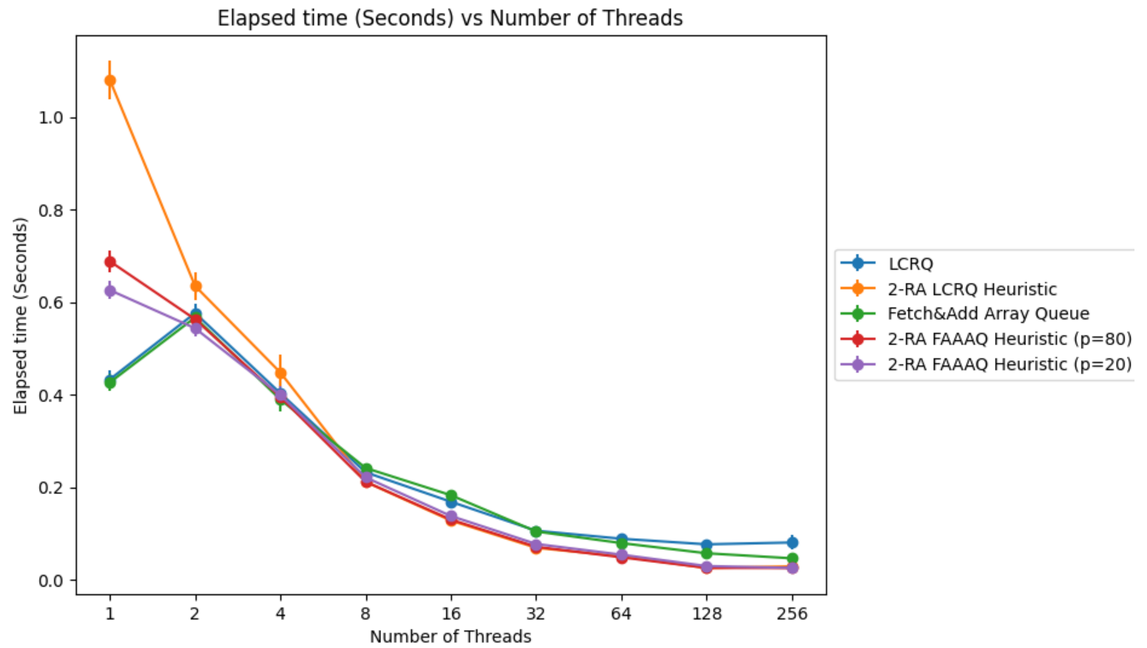


(a) Elapsed time

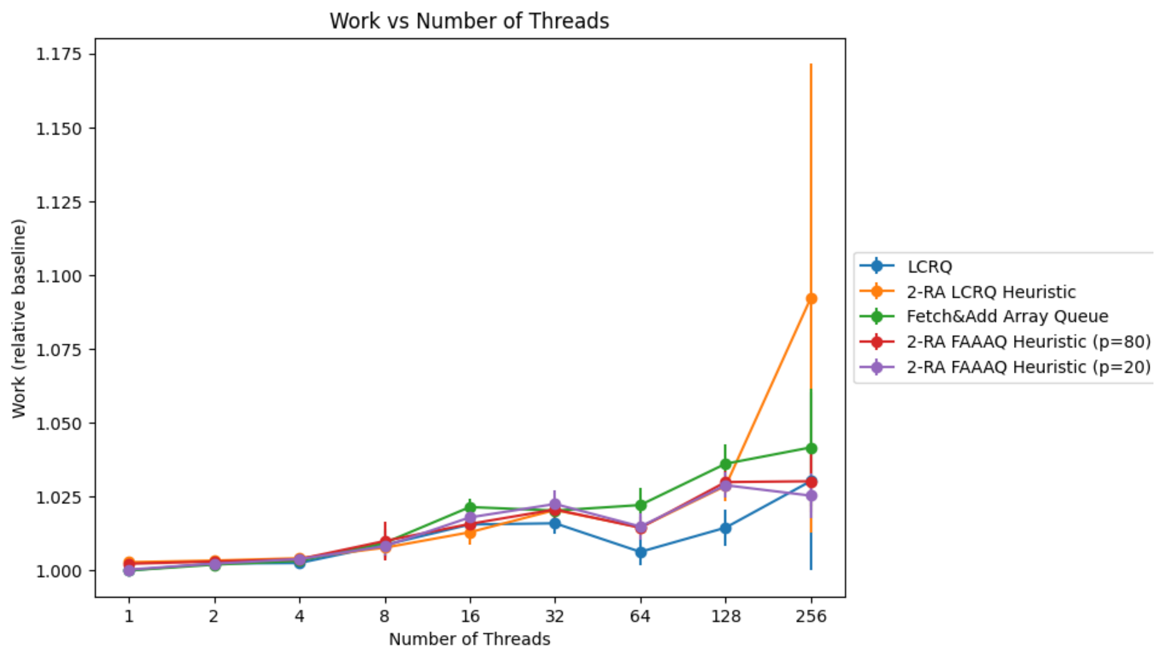


(b) Average path length

Figure A.7: The elapsed time and average path length of the different data structures on sparse50M when errors are allowed.

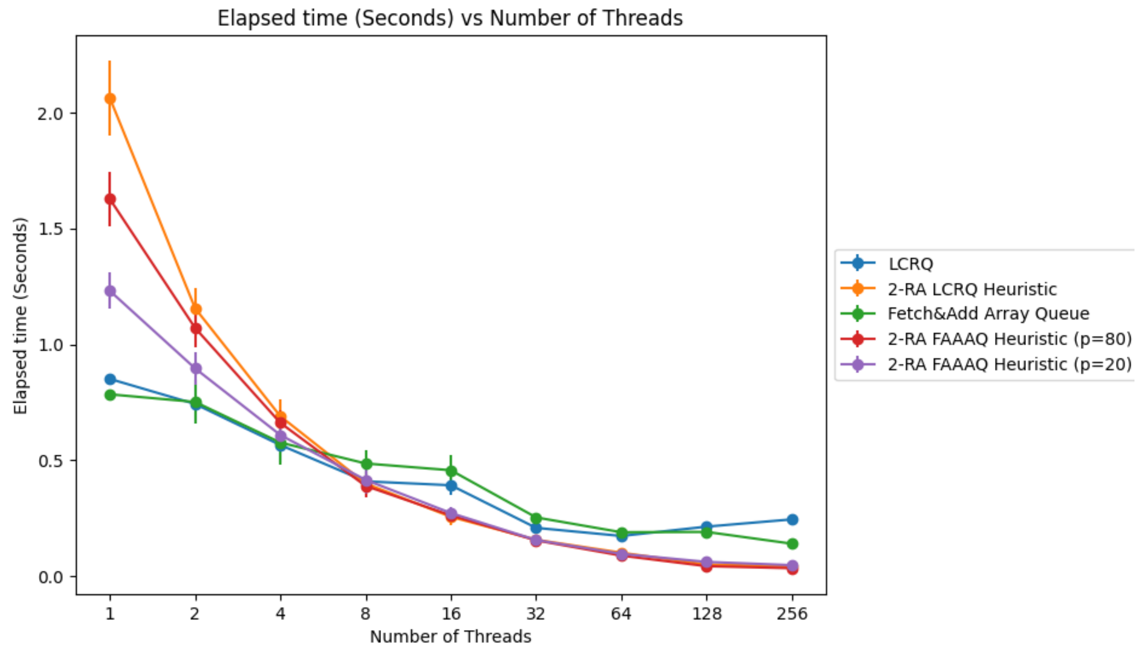


(a) Elapsed time

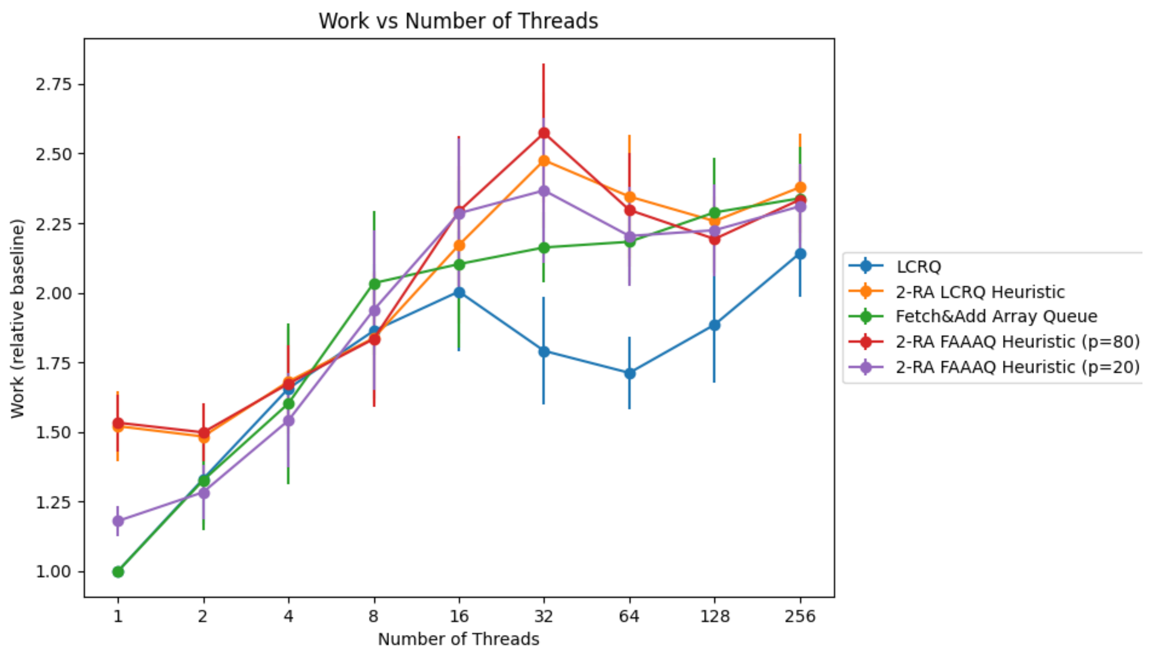


(b) Total work

Figure A.8: The elapsed time and total work of the different data structures on audikw_1 during unordered BFS.

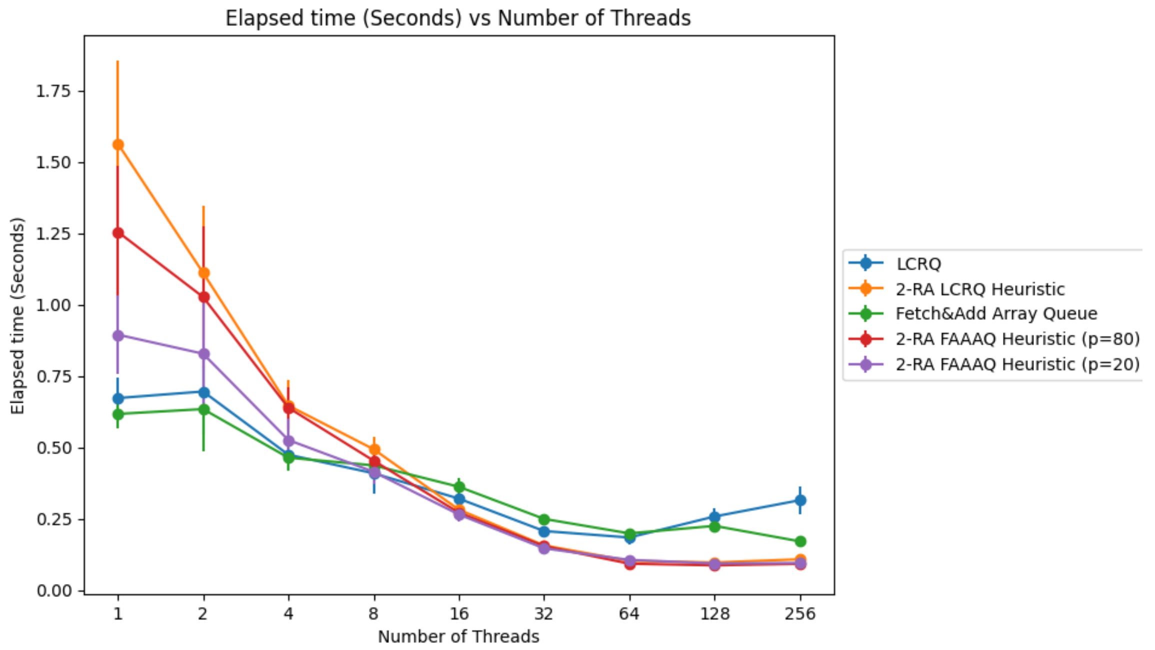


(a) Elapsed time

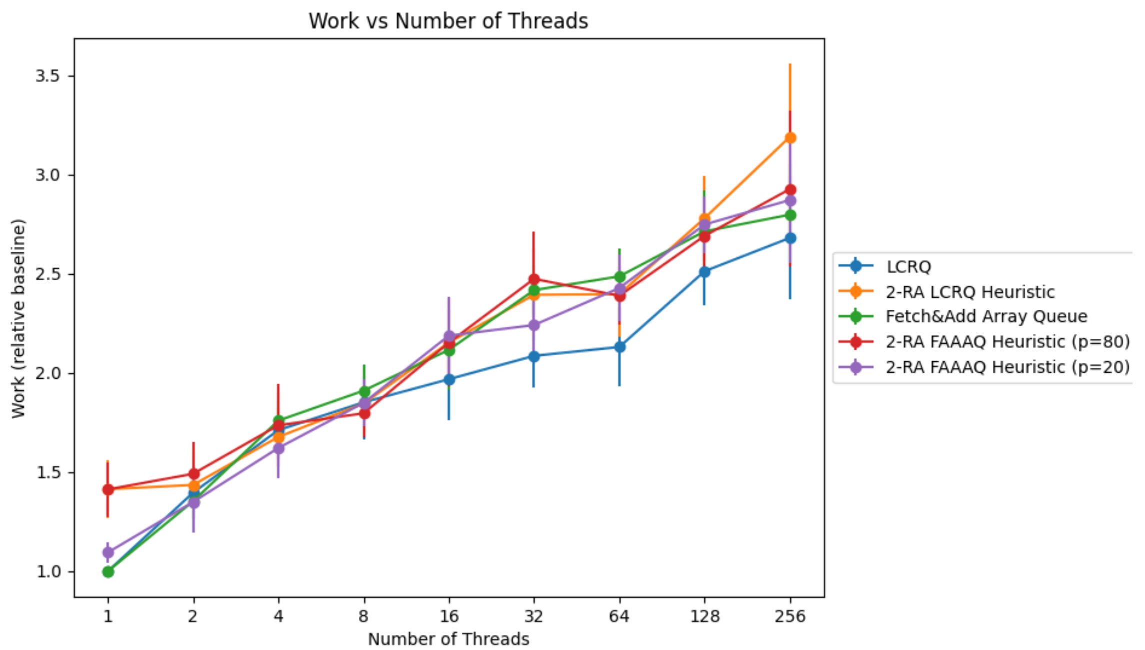


(b) Total work

Figure A.9: The elapsed time and total work of the different data structures on sparse10M during unordered BFS.



(a) Elapsed time



(b) Total work

Figure A.10: The elapsed time and total work of the different data structures on sparse50M during unordered BFS.

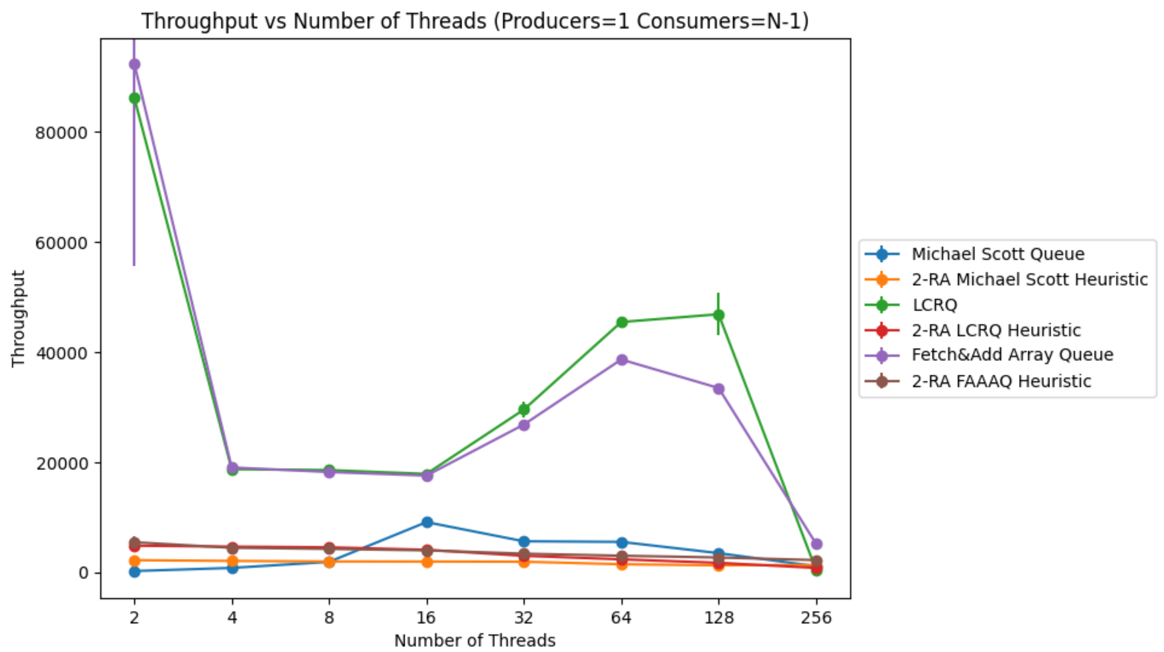


Figure A.11: Throughput of producer-consumer benchmark with one producer and a varying number of consumers. Higher is better.