

*CS4204 Concurrency and Multi-core Architectures P1 -
Concurrent Data Structures*

Concurrent Multiset Strategies

16/02/2024 - 230033234

1 - Introduction	3
1.1 Checklist	3
1.2 Compile & Execute	3
2 - Design and Implementation	4
2.1 Structure & Overview	4
2.1.1 Libraries Used/C++ Environment	4
2.2 Single Lock Synchronisation (Coarsed-Grained)	5
2.2.1 Correctness	5
2.3 Optimistic Synchronisation	6
2.3.1 Correctness	6
2.4 Lock-free/Non-blocking Synchronisation	7
2.4.1 Correctness	7
3 - Performance Analysis / Benchmarking	8
3.1 Methodology	8
3.1.1 Preparation	8
3.1.2 Execution	8
3.2 Results	8
3.2.1 Balanced (50% reads / 50% writes)	9
3.2.2 Read-Heavy (80% reads / 20% writes)	9
3.2.3 Write-Heavy (80% reads / 20% writes)	10
3.3 Critical Analysis & Implications	10
4 - Evaluation	11
4.1 General Evaluation	11
4.2 Future iterations & Possible Improvements	12
References	13

1 - Introduction

Concurrent data structures are designed to allow multiple threads/processes to access the shared resource element of the structure without leading to inconsistent states. These data structures are used in parallel computing to improve performance and efficiency via parallel execution. Proving correctness for these algorithms are important in ensuring safety. One way of demonstrating this is showing that an algorithm is **sequentially consistent**. (*Do these operations appear to take place in some total order, where the order is consistent with the order of operations on each individual process?*)[2]

It is critical to highlight the importance of **linearisability**, which is an important property in helping prove correctness for concurrent operations. The idea behind this is the ensurance that all operations on a data appear to be executed instantaneously, in some order that is consistent with the real-time ordering of these operations. If *Operation1* finishes before another operation *Operation2*, then *Operation1* must **also** appear to happen before *Operation2*, in the sequence of operations applied to the given data structure.

In this practical, in attempt to avoid concurrent anomalies and inconsistencies in such data structures, different synchronisation strategies have been used to develop three implementations for a concurrent multi-set data structure.

Instructions for running the system can be found in section 1.2. The design and implementation, with arguments for correctness is reported in section 2.

1.1 Checklist

Part	Section	Status
Single Lock Synchronisation	<i>Code Implementation</i>	Completed
	<i>Stress Test and Benchmarking</i>	Completed
	<i>Correctness Argument</i>	Completed
Optimistic Synchronisation	<i>Code Implementation</i>	Completed
	<i>Stress Test and Benchmarking</i>	Completed
	<i>Correctness Argument</i>	Completed
Non-blocking (w/ Lazy Synchronisation)	<i>Code Implementation</i>	Completed
	<i>Stress Test and Benchmarking</i>	Completed
	<i>Correctness Argument</i>	Completed

1.2 Compile & Execute

Access the '/src' folder, and compile (this example uses the GCC compiler):

```
g++ run_tests.cpp -o main
```

2 - Design and Implementation

The following section provides an overview of the structure of the different algorithms, and then goes in depth arguing for correctness for each strategy, also providing designed stress tests to empirically validate correctness.

2.1 Structure & Overview

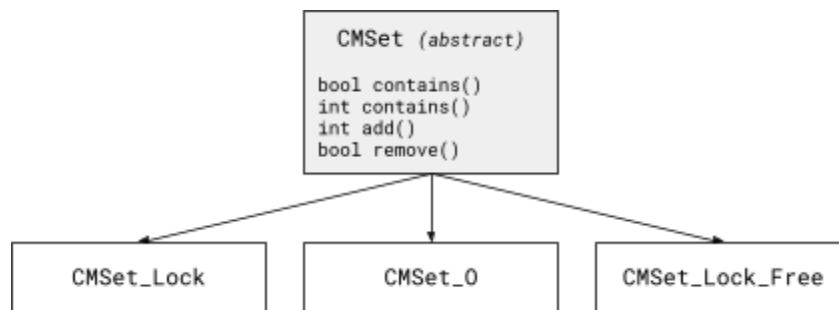


Figure 1 : Class structure of the different Concurrent Multiset algorithms

The concurrent multi-set data structure is designed to manage collections where duplicates are allowed. Intending to guarantee thread-safe add(), remove(), count() and contains() methods. In reference to figure 1 above, an abstract superclass 'CMSet' is used to define the interface for these methods. The different concurrent algorithms will derive this interface to provide concrete implementations

The three distinct strategies for handling concurrency are: '**CMSet_Lock**' which uses a single-lock per multiset, '**CMSet_O**' which refers to optimistic synchronisation, and the '**CMSet_Lock_Free**' which follows a lock-free non-blocking/lazy synchronisation approach.

A **singly-linked list** is used to represent the multi-set, due to its high potential for fine-grained locking, where an individual node in the list can be locked independently, allowing the potential for less contention and therefore higher concurrency. It is also a good fit for lock-free techniques which utilise atomic operations such as compare-and-swap (CAS). All operations require the traversal of the list, we continuously update a '**current**' node pointer to do so.

2.1.1 Libraries Used/C++ Environment

The implementations take advantage of standard C++ libraries, more precisely ‘<mutex>’ is used for mutual exclusion locks in the Single-Lock and Optimistic Synchronisation strategies. Whilst make use of ‘<atomic>’ for atomic operations in the 3rd lock-free non-blocking strategy.

2.2 Single-Lock Synchronisation (Coarsed-Grained)

This strategy follows a more coarse-grained approach by using a single mutex (wrapped with a lockguard) ‘**mtx**’ to protect the entire data structure, ensuring that only one thread can access the multi-set and modify it at any time. A lockguard was experimented to ensure a thread only owned the mutex for the given operation’s scope (RAII Style)[3]. *A lock guard will only be used for this strategy. Other strategies require more fine-grained control.*

(Refer to CMSet_Lock Class in the CMSet.hpp file, for more in detail comments of its implementation)

Coarse-grained synchronisation strategies such as this can be suitable when the chance of concurrent access/contention is minimal. However in scenarios where many threads attempt to access it (high-contention), it may result in reduced performance due to sequential bottlenecks.

2.2.1 Correctness

The single lock implementation guarantees exclusive access to the multiset by locking the entire data structure of each operation (add, remove, contains, count). Since we lock the entire data structure each time we run one of these operations, mutual exclusion is met and therefore ensures that all operations are atomic, therefore preventing data races and maintaining consistency. The acquisition of the mutex can be seen to serialise access to the linked list, as only one thread can hold the mutex at a time. So **sequential consistency** is met due to its coarse-grained exclusive access guarantee. Order is guaranteed in the order they acquire the mutex.

The implementation also achieves linearisability, as the acquisition and release of the mutex around each operation ensure an instantaneous execution from the perspective of other threads (between starting the operation and completion). This respects real-time ordering, as operations can be seen to be linearised in the order they acquire the mutex, therefore global order is also consistent with the actual time sequence of the events.

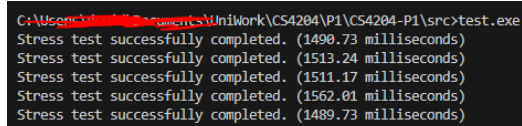
Empirical validation via Stress Test:

A stress test has also been used to empirically validate the correctness and robustness of the implementation under extreme conditions to see if it can successfully execute. To do this, the test function ‘**run_stress_test()**’ is used which stimulate a scenario of high thread contention to test for potential deadlocks and race conditions. We’ve set up **100** threads for only **10,000** total operations. The stress test function runs a mix of operations randomly (add, remove, count, contains), with additional random sleeping to stimulate real work.

```
int num_threads = 100;
int num_ops = 10000;

run_stress_test(cmset_lock, num_threads, num_ops);
```

The experiment was ran **5** times, where there execution successfully terminated, indicating no adverse results such as deadlocks or livelocks. *Although Race conditions could be reasoned with the theoretical justifications, a thread sanitizer could be used in later iterations to confirm such arguments.*



```

C:\Users\chris\Documents\UniWork\CS4204\PI\CS4204-PI\src>test.exe
Stress test successfully completed. (1490.73 milliseconds)
Stress test successfully completed. (1513.24 milliseconds)
Stress test successfully completed. (1511.17 milliseconds)
Stress test successfully completed. (1562.01 milliseconds)
Stress test successfully completed. (1489.73 milliseconds)

```

Figure 2 : Stress test screenshots for the Single Lock Algorithm

2.3 Optimistic Synchronisation

This strategy involves delaying the locking any nodes whilst traversing the list, and then locking the node if a target node is found, then introducing a validation/confirmation step to check if there are unexpected changes caused by conflicts. If the 'is_valid()' step returns true, then the correct node has been locked and we continue the operation execution, otherwise we release the locks and re-try (modifications have been made by other threads).

(Refer to CMSet_O Class in the CMSet.hpp file, for more in detail comments of its implementation)

This algorithm reduces chances of lock contention compared to coarse-grained locking, allowing multiple reader thread accesses and also more fine-grained locking for writer threads.

2.3.1 Correctness

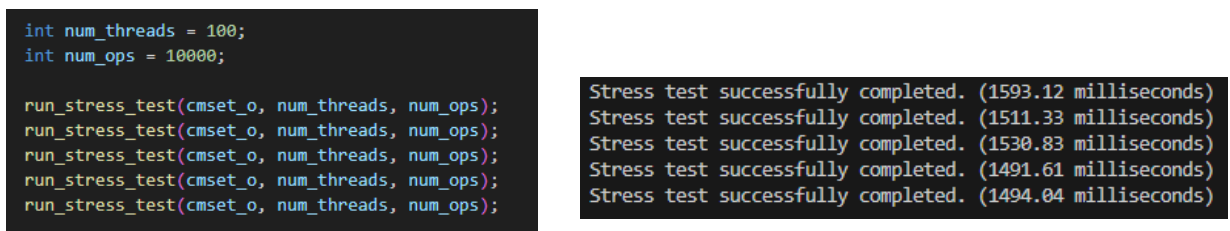
The importance here is that there is a validation step executed **before** modification of the data but **after** the individual node has been locked. The validation step is used to confirm that the current node is still reachable from the head and has therefore not been removed meanwhile by another thread. Furthermore, not only is the '**current**' node tracked for traversal means, we also track its predecessor ('**pred**'). We can claim validity by also checking if '**pred->next**' is **still** referring to the 'current' (no modifications/removals have been made by other threads). If valid, the current thread can now modify the shared data in the current node as it guaranteed no other thread can access the shared data.

There is also an ensurance that there is a global order on how locks should be acquired, where the pred node is first locked, then the current node is locked. We maintain this order throughout all locking and unlocking of the nodes. If this global order constraint is not present, then there's a chance a deadlock can result if two threads attempt to acquire mutexes in opposite orders.

In general, the correctness is based on the validation step that ensures that any modification is still based on a consistent view of the data structure, even though there is reduce chance of contention due to the more fine-grained approach. The strategy guarantees that if the validation step is unsuccessful, the operation restarts, ensuring that only consistent and valid modifications can be made, maintaining data integrity. To summarise, sequential consistency is maintained because of this verification/validation step before comitting changes, this ensures that the final state reflects a sequence of operations that could have been executed serially.

Empirical validation via Stress Test:

Using the same stress test set up as before, but with the optimistic algorithm:

The image contains two screenshots. The left screenshot shows C++ code for a stress test:

```
int num_threads = 100;
int num_ops = 10000;

run_stress_test(cmset_o, num_threads, num_ops);
run_stress_test(cmset_o, num_threads, num_ops);
run_stress_test(cmset_o, num_threads, num_ops);
run_stress_test(cmset_o, num_threads, num_ops);
run_stress_test(cmset_o, num_threads, num_ops);
```

The right screenshot shows the output of the stress test:

```
Stress test successfully completed. (1593.12 milliseconds)
Stress test successfully completed. (1511.33 milliseconds)
Stress test successfully completed. (1530.83 milliseconds)
Stress test successfully completed. (1491.61 milliseconds)
Stress test successfully completed. (1494.04 milliseconds)
```

Once again, no random crashes or adverse conditions. Shows we have not deadlocked, livelocked, seg faulted or any other adverse conditions.

Figure 3 : Stress test screenshots for the Optimistic Synchronisation Algorithm

2.4 Lock-free/Non-blocking Synchronisation

This strategy implements a lock-free approach. Taking advantage of the ‘<atomic>’ library, atomic operations are used to manage the multi-set in a non-blocking fashion. Scalability is greatly improved as many threads are allowed to proceed and traverse the list without waiting for locks. This implementation uses both logical deletion and compare-and-swap (CAS) (<atomic>’s member function ‘compare_exchange_weak’) in its operation. Helper functions were implemented to handle the marking and unmarking process for the logical deletion step. We approach this by converting the ‘node->next’ pointer to an unsigned int, then applying a bitwise operation (setting the LSB to be 1) in order to mark a node to be logically deleted.

(Refer to CMSet_Lock_Free Class in the CMSet.hpp file, for more in detail comments of its implementation)

Furthermore, operations do not need to block other threads or wait for operation to complete, reducing the potential for deadlock and increasing concurrency.

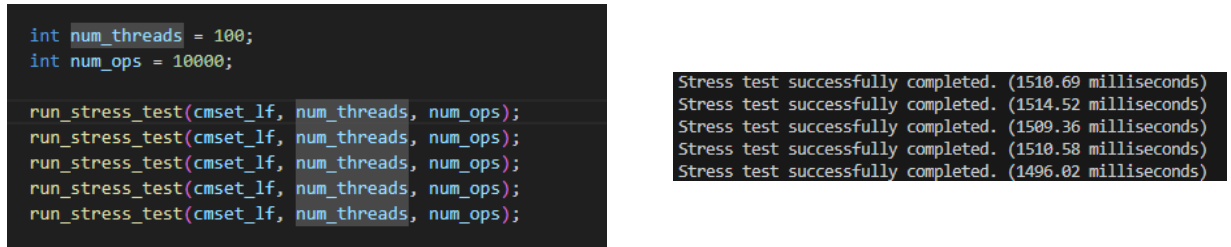
2.4.1 Correctness

In terms of correctness, atomic operations are used to ensure that all modifications to shared data are atomic. These atomic operations are the compare-and-swap (CAS) operations, which were used to add, remove and update nodes. A successful CAS operation could be seen as a clear linearisation point, ensuring that the operation occurs instantaneously to other threads. This is because the CAS operation is atomic, meaning that there are no intermediate states visible to other threads. It could be described in a way that because of CAS, each operation’s effect is reflected in the global state (perspective of all threads) in a way that could also be replicated by a sequential execution of operations.

The current implementation also ensures that each method (add, remove, contains etc) is completed in a finite number of steps. There is no situation where the execution is indefinite. The eventual completion guarantee also ensures that the data structure is **sequentially consistent**. Additionally, careful use of the ‘memory_order_acquire’ and ‘memory_order_release’ memory orderings were used when using CAS to ensure that potential re-ordering of writes and reads were prevented in critical cases where this may result in unpredictable behaviour, further ensuring sequential consistency.

Empirical validation via Stress Test:

Using the same stress test set up as before:



```
int num_threads = 100;
int num_ops = 10000;

run_stress_test(cmset_lf, num_threads, num_ops);
run_stress_test(cmset_lf, num_threads, num_ops);
run_stress_test(cmset_lf, num_threads, num_ops);
run_stress_test(cmset_lf, num_threads, num_ops);
run_stress_test(cmset_lf, num_threads, num_ops);
```

```
Stress test successfully completed. (1510.69 milliseconds)
Stress test successfully completed. (1514.52 milliseconds)
Stress test successfully completed. (1509.36 milliseconds)
Stress test successfully completed. (1510.58 milliseconds)
Stress test successfully completed. (1496.02 milliseconds)
```

A successful completion of the stress test indicates no deadlocks, livelocks, segfaults or additional adverse effects have resulted from this implementation. Aiding to empirically validate the implementation's correctness.

Figure 4 : Stress test screenshots for the Lock-Free Synchronisation Algorithm

3 - Performance Analysis / Benchmarking

3.1 Methodology

The algorithms will be benchmarked in a diverse set of test scenarios. These scenarios will measure the algorithms' performance in situations of high vs low contention, where there is a random mixture of operations and for different ratios of reads to writes (Read-Heavy vs Write-Heavy). Most importantly we will be measuring **throughput** and **average latency** for each algorithm. The **contention** and **scalability** can be analysed implicitly from these metrics.

3.1.1 Preparation

For each test scenario, we first initialise the specific multiset implementation and then test it using the 'run_benchmarking_scenario()' function. This function allows for controlled testing, as we can input a given number of threads, total operations and a read/write ratio. We will be running our scenarios using an AMD Ryzen 7 5800X CPU, which has 8 cores, using GCC with MinGW as the compiler on a Windows OS.

3.1.2 Execution

The benchmarking function spawns the given number of threads, assigning each thread a mix of operations based on the function input. The '<chrono>' library is used for high-resolution timers to calculate the **throughput** and **average latency** metrics. We will run a total of **10,000,000** operations (add, remove, contains, count).

3.2 Results

We will define our **throughput** as *how many operations the system can handle per a second (ops/sec)*. The following figures show the throughput as the number of threads increase (balanced distribution of read and writes). Whilst we define our **average latency** as *the average time taken to complete a single operation in the system (ms/ops)*.

Running the benchmarking test for different ratios of read-to-writes and then compiling the results into a graph using python's '**matplotlib**' library, has resulted in the figures below:

3.2.1 Balanced (50% reads / 50% writes)

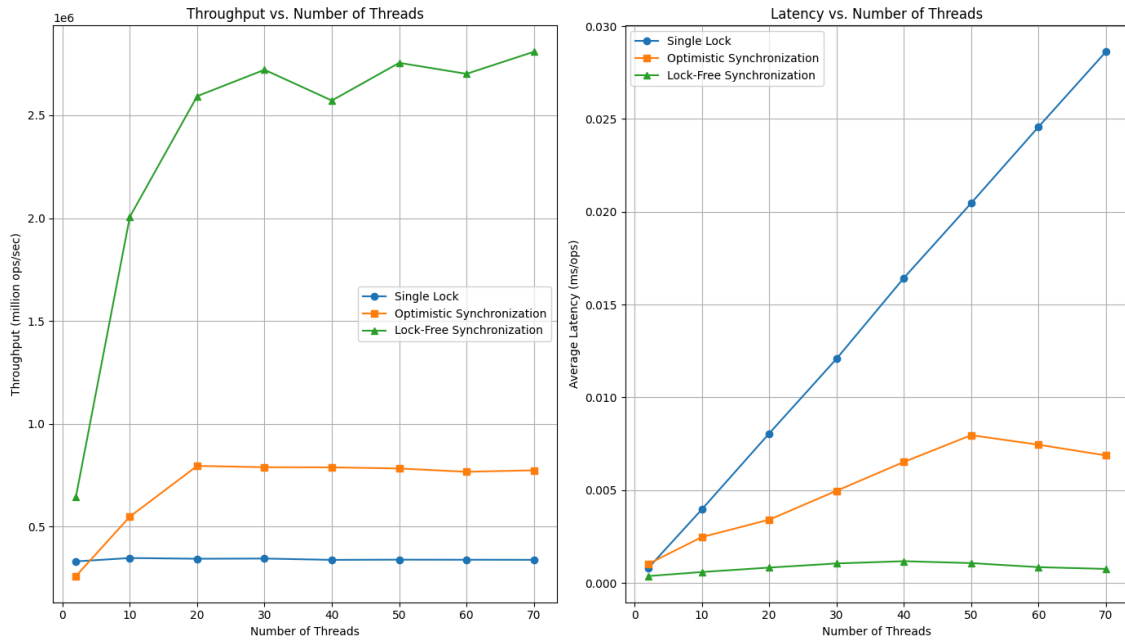


Figure 5 : Graph showing both the 'Throughput vs Num of Threads' and 'Latency vs Number of Threads' for 50/50 read-write ratio.

3.2.2 Read-Heavy (80% reads / 20% writes)

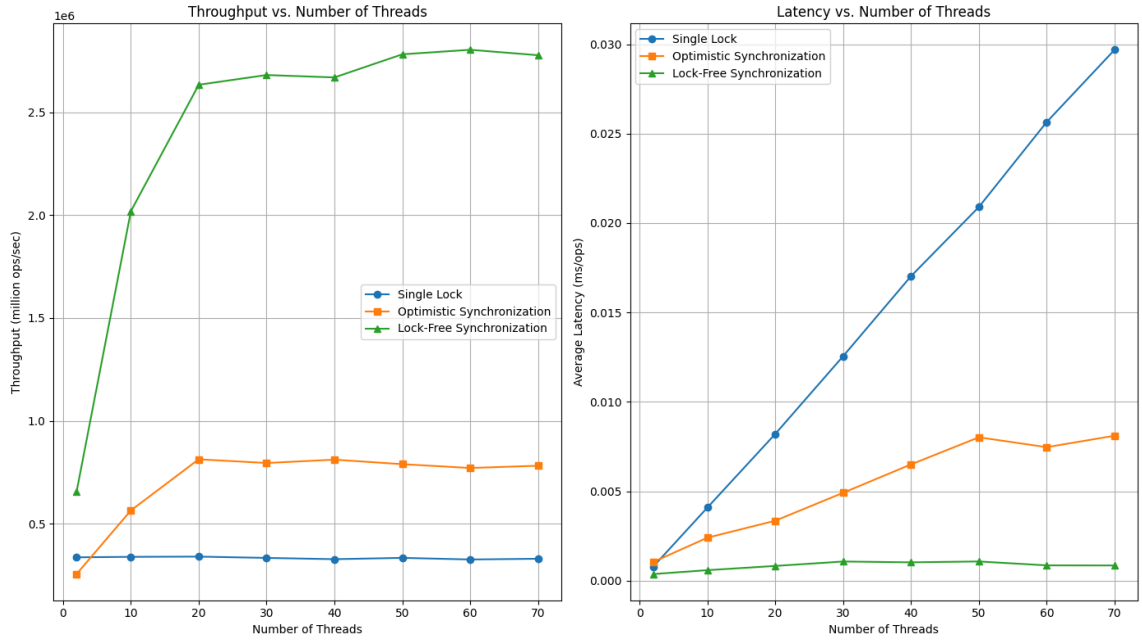


Figure 6 : Graph showing both the 'Throughput vs Num of Threads' and 'Latency vs Number of Threads' for 80/20 read-write ratio.

3.2.3 Write-Heavy (80% reads / 20% writes)

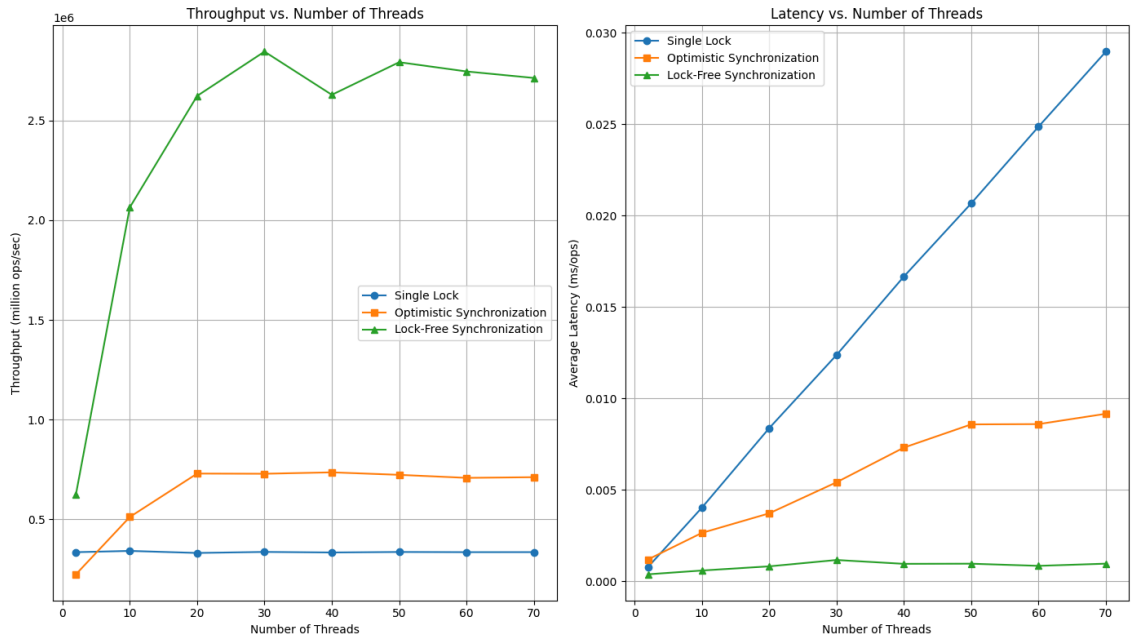


Figure 7 : Graph showing both the 'Throughput vs Num of Threads' and 'Latency vs Number of Threads' for 80/20 read-write ratio.

3.3 Critical Analysis & Implications

3.3.1 Single-Lock Strategy

In all tested scenarios, the throughput remains relatively flat although the number of threads are increasing. This verifies the predicted case of inefficiency mentioned briefly in section 2, where the use of a global mutex results in a sequential bottleneck, limiting scalability. This is evident as increasing the threads does not result in any considerable improvement in throughput, suggesting that threads are likely waiting/blocked in an attempt to acquire the mutex.

The average latency can be seen to increase almost linearly with the number of threads. This is consistent with our predictions based on the locking contention. As the threads increase, there is increased competition to acquire the mutex, leading to longer waits and therefore increase latency.

3.3.2 Optimistic Synchronisation Strategy

Initially, the throughput increases as the number of threads increase, showing better scalability in comparison to the single lock strategy. However in all tested scenarios, the throughput eventually begins to plateau and there is even a slight decline as the number of threads continue to increase. The plateau may be a result of high-contention due to a high thread count. The overhead of restarting the traversal step once the validation step is failed may have resulted in this slight decline in all tested scenarios. The optimistic synchronisation allows for around a **2x** increase in throughput compared to the single lock implementation.

In terms of latency, the initial latency may greater with fewer threads compared to the other methods, but the scale is much better than the single lock (less steep compared to the single lock). Whilst the single lock is linear, the optimistic synchronisation plateaus off eventually. Indicating a better handling of contention at higher thread counts.

3.3.2 Lock-Free Strategy

In terms of throughput, there is an immediate steep increase with the number of threads. Although an eventual plateau, the lock-free strategy reaches the ranges of **'2.5+ million operations'** in comparison, which is an average of **7.3x** increase compared to the single lock and **3.7x** compared to the optimistic strategy. This shows the excellent scalability of the approach, suggesting that the strategy is able to increase the thread count without resulting in significant contention.

The latency remains the lowest with no trend of increase across all thread counts, indicating the wait-free quality of the strategy. Threads can complete their operations quickly without being blocked. The method is extremely efficient in terms of scalability and handling high-contention scenarios due to the wait-free property. Empirically, we can deduce that the lock-free synchronisation outperforms the other two strategies in both throughput and latency metrics.

4 - Evaluation

4.1 General Evaluation

Implementing and testing the three distinct strategies for managing a concurrent multi-set reveal significant insights into how to effectively guarantee correctness theoretically through algorithm design and empirically through stress testing. Furthermore, the benchmarking process reveal the performance characteristics in reference to the key metrics, **throughput** and **average latency** under test scenarios that vary in terms of thread contention. The chosen strategies themselves are all distinct in terms of their approach to synchronisation, so completing a thorough analysis allowed us to see how each approach directly impacted efficiency and scalability.

The **Single-lock** followed a simple-to-implement, coarse-grained approach - locking the entire data structure per an operation. This resulted in flat throughputs even as the number of threads increased. This trend demonstrates the strategy's inability to scale effectively, due to bottlenecks as a result of a global lock/mutex. This limits any potential gains in performance that could be from increasing cores/parallelism. As a consequence, this strategy is less suitable for environments of high-thread contention. This can be further validated by the linear increase in latency, demonstrating its inefficiency in handling concurrency.

The **optimistic synchronisation** strategy improved in terms of scalability compared to the single-lock per multiset approach, shown with the initial rise of throughput with the number of threads. However, the eventual plateau and slight decline can indicate an upper-limit to its scalability, due to the overhead of the validation step. A failure to validate results in a retraversal of the linked list, this can be seen to have limited the throughput in situations of high-thread contention. However, we can still see an approximate **2x** increase in throughput compared to the Single-Lock method. Although having the highest latency (out of all strategies) at lower thread counts, the rate of increase in latency is less steep, even plateauing, compared to the linear rate of increase shown with for the single-lock. This behaviour indicates that this strategy is more efficient at handling contention where there is a **higher** thread count. Showing its suitability to environments with moderate to high thread contention.

However, the **lock-free** strategy (non-blocking with lazy synchronisation) can be considered the clear leader in performance, in terms of both the throughput and latency metrics (and implicitly scalability). The wait-free approach reaches an average throughput that is around **7.3x** and **3.7x** higher than the single-lock and optimistic strategies, respectively. The strategies demonstrate a strong ability to handle high contention without bottle-necks or performance reductions due to overheads that are present in the other strategies, which is further evident by latency remaining constantly low across all thread counts. Although the implementation is the most complex, its efficiency in high-contention scenarios justify this strategy being used in potential applications that require high throughput and low latency.

4.2 Future iterations & Possible Improvements

Although the lock-free implementation was empirically evaluated to be successful and aligned with predictions to perform better than the other strategies. It could be reasoned that there was an expectation to have a noticeable difference in performance between read-heavy test scenarios vs write-heavy (due to the non-blocking nature of the algorithm). This could spark future investigations in more detail to the significance of read-write ratios whilst implementing these algorithms. However, the report showed a clear demonstration of the effective performance of wait-free, non-blocking strategies relative to other approaches.

Furthermore, extensive analysis of additional metrics such as memory usage for the different strategies could be deemed critical. In the current implementations, only the single-lock approach does a thorough memory of deletion with the use of a destructor. The more fine-grained structures such as the lock-free strategy may need additional protection such as Hazard Pointers, which manages a list of pointers to nodes it is currently accessing. This list must be checked before physical deletion[4]. This could perhaps be implemented in future iterations.

References

- [1] Herlihy, Maurice, and Nir Shavit. *The Art of Multiprocessor Programming*. 2008.
- [2] “Sequential Consistency.” *Jepsen*, <https://jepsen.io/consistency/models/sequential>. Accessed 18 February 2024.
- [3] “std::lock_guard - cppreference.com.” *C++ Reference*, 6 July 2023, https://en.cppreference.com/w/cpp/thread/lock_guard. Accessed 18 February 2024.
- [4] M. Michael, Maged. “Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects.” 2004.