Module number: CS4204
Matriculation number: 200017941

P1: Concurrent Data Structure Implementation Report

Overview

The aim of this coursework is to program a set of concurrent algorithms for a multi-set data structure. As part of the requirement, I have implemented four concurrency strategies, two of which are high-performance. These include: coarse-grained synchronisation, fine-grained synchronisation, optimistic synchronisation and lazy synchronisation. For each of these algorithms, a correctness argument has been provided along with measurements of its performance.

Design and Implementation

The multi-set data structure has been implemented as a linked-list using *structs* in C. For each of the algorithms, this same linked-list structure has been used, with slight variations in the fields within the *struct* for each. Each linked-list is initiated with a global head node to represent the start of the linked-list. This is important when it comes to ensuring that all the threads accessing the multi-set starts from the same point. The end of the linked-list is marked using NULL which is important when traversing the list. When threads that reach this point, this means that the element being searched for is not present. Since this is a multi-set, each linked-list node has a count field for the number of occurrence of an element, which is used as part of the *count* operation.

The core functionality of *contains*, *count*, *add* and *remove* are all the same for each algorithm. They differ in how they utilise concurrency tools to ensure mutual exclusion. These are described below.

Coarse-grained synchronisation

A global *mutex* is used to ensure safe access, where the start and end of each operation has a *mutex lock* and *unlock* function calls. This means that each time, there can only be one thread at a time in the multi-set performing an operation.

This approach is very restrictive since it only allows one thread to access the multi-set at a time due to there only being one lock.

Fine-grained synchronisation

Each linked-list node has its own local *mutex*, so that each node in can be locked independently. Each time a thread is performing an operation, the current and previous nodes are locked to ensure mutual exclusion. This means that there can be multiple threads accessing the multi-set. It is important to note that threads all traverse the linked-list in the same direction and cannot overtake each other.

This approach improves on the restrictive nature of coarse-grained synchronisation since it allows multiple threads to access the multi-set at time. However, a drawback of this approach is the overhead due to the extensive amount of locking and unlocking done. There is also memory overhead since every node has a *mutex*.

Optimistic synchronisation

This implementation still uses a local *mutex* for each node, except the locking mechanism is not done as often and is only done when needed. This is happens when an element has been found in

the multi-set, which causes the current and previous nodes to be locked. Additionally, before each operation is performed, the current node is validated to make sure that it is still present in the multi-set. This is achieved by simply traversing the list from the head and checking to see if the current node is reachable. If this fails, the locks are released and the operation is performed again.

This approach improves on fine-grained since it doesn't require every single current and previous node to be locked and unlocked. However, in scenarios where there is a lot of contention occurring, the performance will be drastically impacted. The validation functionality also adds additional overhead since every time it fails the linked-list has to be traversed.

Lazy synchronisation

The implementation is for this algorithm is similar to optimistic synchronisation, except it requires even less locking. For read operations which include *count* and *contains*, locking is not done at all. For write operations which include *add* and *remove*, locking is done in the same way as for optimistic synchronisation. Additionally, each node is only physically removed when needed, therefore a *marked* field is included in each node to indicate whether a node is logically deleted. Finally, the validation involves simply checking that the current and previous nodes have not been marked.

This approach is a slight improvement over optimistic synchronisation since it gets rid of the overhead associated with validating the nodes. This is because instead of traversing the entire linked-list, only the marked fields for the current and previous nodes need to checked. And each time instead of always having to physically delete a node, it can instead be logically deleted at first which is much quicker.

Correctness

To prove the correctness of my implementations, I will use both a theoretical and a practical approach. The theoretical approach involves identifying linearisable points, which are points in the algorithm where every operation appears to happen instantaneously. This is relevant when it comes to operations that modify the concurrent multi-set and so the proof is mainly focused on *add* and *remove* functions. For the addition of elements, the linearisable point occurs when the element is added to the multi-set successfully. For the removal of elements, the linearisable point occurs when the element is added to the multi-set successfully. For read operations, since the state of the multi-set is not modified, there is no effect that appears to happens from an outside perspective. Below the linearisable points have been highlighted and explained for each algorithm.

Coarse-grained Synchronisation

- *add*: the linearisable point is when either the *count* field being increments if the element is already present, or the pointer of the end node being updated to point to the new node. All this happens after acquiring the lock at the start of the function and before releasing it at the end of the function, and therefore this appears to be instantaneous.
- *remove*: the linearisable point is when either *count* value is decremented or the *next* pointer of the previous node is assigned the *next* pointer of the current node once the correct element has been found. This is done after acquiring the lock at the start of the function and before releasing it at the end of the function, and therefore this appears to be instantaneous.

Fine-grained Synchronisation

- *add*: the linearisable point is the same as coarse-grained, except the lock is acquired and released for the current and previous nodes instead of the entire multi-set. Between the acquiring and the releasing of the locks for both nodes is when the add operation happens.
- *remove*: the linearisable point is the same as coarse-grained, except the lock is again acquired and released for the current and previous nodes instead of the entire multi-set. Between the acquiring and the releasing of the locks for both nodes is when the remove operation happens.

Optimistic Synchronisation

- *add* and *remove*: the linearisable point is the same as fine-grained. The only difference is that there is a function to validate the current node and so if this fails the operation doesn't take into effect.

Lazy Synchronisation

- *add* and *remove*: the linearisable point is the same as optimistic synchronisation.

The practical approach involves examining traces from running particular test programs. In order to do this, I have placed print statements at every point in the algorithm and then after running the test programs, I went through the output trace to verify the correctness of my implementations. As part of the output, I also printed the final state of the multi-set. The result of performing this examination is included in the *output_log* sub-directory, where I ran *custom_test.c* as the test program with each of the concurrent algorithms. I chose to run it with 10 threads and 20 operations per thread. This is however done for demonstration purposes to provide an insight into how I approached proving the correctness practically. The result of this approach was that I was able to conclude that the algorithms implemented are deadlock free and starvation free since the program terminated with an output result and very thread was able to complete all of its operations. Additionally, mutual exclusion is ensured since every time one thread acquired a lock, other threads were unable to acquire it.

Benchmark Performance

The performance of each algorithm has been evaluated using a series of multi-threaded programs. There are three main programs in particular: *more_reads_test.c*, *more_writes_test.c* and *equal_reads_and_writes_test.c*, where each is characterised by whether it priorities read operations or write operations. The purpose of this is to highlight different use case scenarios since this can have an impact on the performance of the algorithms.

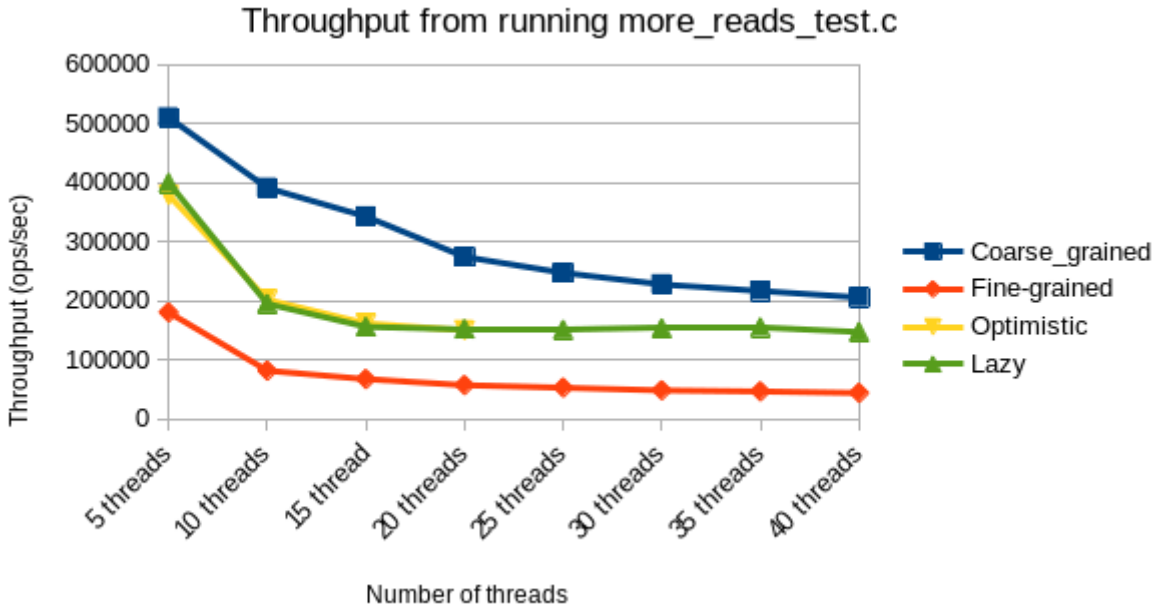
Each thread randomly selects an integer between 0 and 1000, as well as randomly selecting the operation to perform. The choice of operation needs to be done in such a way where it has a bias towards either read operations (*count* and *contains*) which is done in *more_reads_test.c* or write operations (*add* and *remove*) which is done in *more_writes_test.c*. For *equal_reads_and_writes_test.c*, a bias is not used so there is an equal chance of picking either. Once the choice has been made to do either a read or a write operation, the specific operation to perform is irrelevant and so it is chosen randomly. This experimental design has been inspired by the paper "A Lazy Concurrent List-Based Set Algorithm" by S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit.

Module number: CS4204
Matriculation number: 200017941

The independent variable of this experiment is the number of threads spawned, which ranges from 5 to 40. The dependent variable is the throughput which is calculated by recording the number of operations done and the time taken to complete them. Each thread does 1000 operations, and so this is the constant variable. The result of benchmarking each concurrency algorithm using the three test program is provided below where for each concurrency algorithm and the number of threads spawned, the throughput is given in the number of operations per second. Note that some of the measurements for optimistic synchronisation are not provided due to the algorithm being unable to terminate with a final result. This behaviour is observed mainly with larger number of threads.

Results after running *more_reads_test.c*

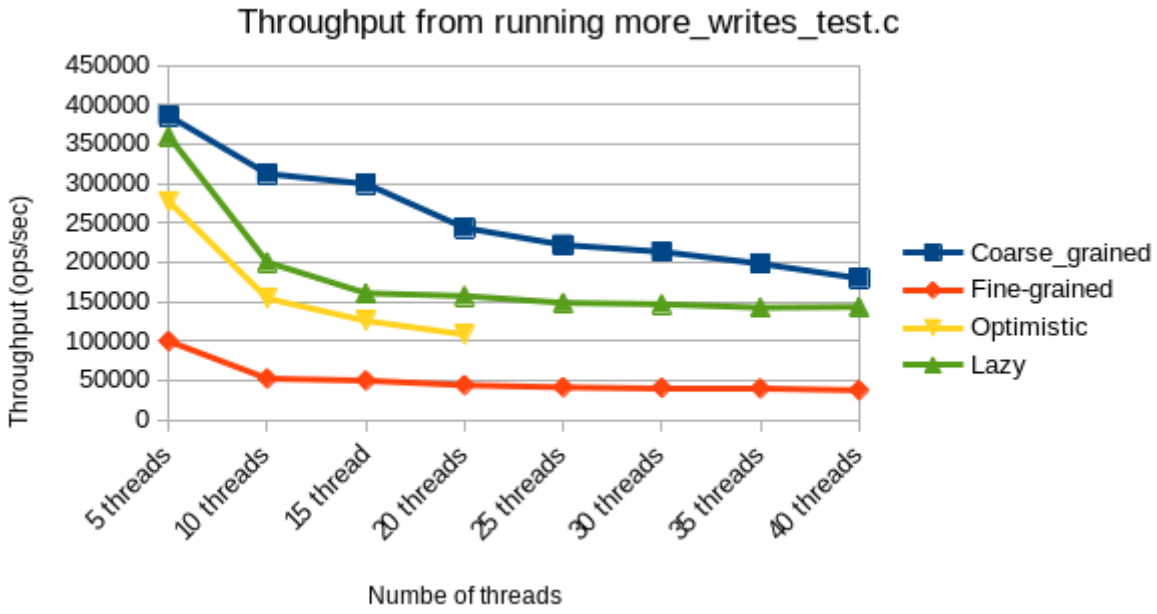
	Coarse_grained	Fine-grained	Optimistic	Lazy
5 threads	509580	179817	378215	399042
10 threads	390549	81227	201401	194492
15 thread	342130	67084	161242	155477
20 threads	273916	56878	149651	151816
25 threads	246897	52444		150559
30 threads	227330	48064		153091
35 threads	216315	46251		154617
40 threads	205329	43675		146797



Results after running *more_writes_test.*

	Coarse_grained	Fine-grained	Optimistic	Lazy
5 threads	385594	99575	277070	359842

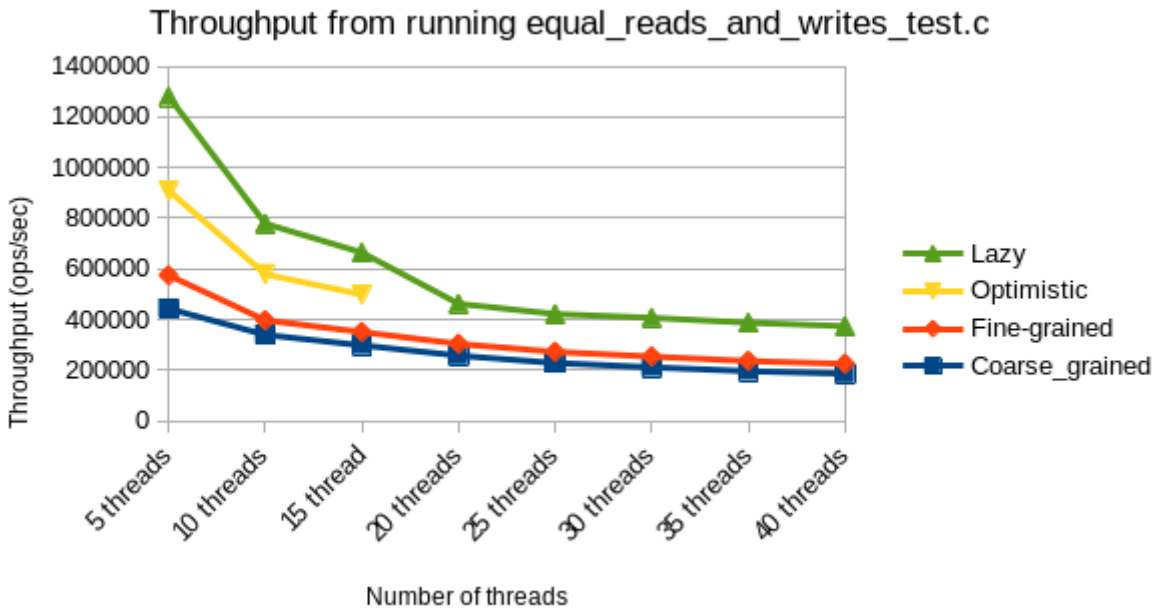
10 threads	312062	52227	153879	199860
15 thread	299401	49696	125488	160692
20 threads	243486	43884	108086	157080
25 threads	221950	41203		148518
30 threads	213235	40519		146686
35 threads	198161	39648		142476
40 threads	179432	37435		143261



Results after running *equal_reads_and_writes_test.c*

Coarse_grained	Fine-grained	Optimistic	Lazy
442321	131759	334582	370562
339109	57273	182235	199556
297649	52398	148034	165922
256987	46361		158728
227741	44026		149504
211201	42210		153032
195520	40302		151135
185666	38830		148808

Module number: CS4204
Matriculation number: 200017941



For the charts with a bias on either read operations or write operations, the results seem to be consistent. Coarse-grained synchronisation seems to have the highest throughput followed by lazy synchronisation and optimistic synchronisation which seem to have similar throughput. Fine-grained synchronisation is proven to have the lowest throughput in both types of tests. When it comes to the chart for equal reads and writes, lazy synchronisation appears to have the best performance, followed by optimistic synchronisation, fine-grained synchronisation and lastly coarse-grained synchronisation. An additional observation is that the graphs for all four algorithms seem to converge slightly as the number of threads increases.