P1 - Multiset

CS4204

200007413

February 2024

Contents

1 Introduction

1.1 Aims and Objectives

The main aim of this submission was to demonstrate and develop my understanding of concurrent algorithms in C++ such as lock-free, wait-free, and high-performance variants. A secondary aim was to develop understanding of lower-level programming principles such as generics and multiple implementations.

The objective was to design and develop at least three implementations of a generic Multi-set data structure.

1.2 Report Structure

The first subsection discusses the design and implementation process. Each of the different algorithms are discussed in detail, with a justification of their correctness using various correctness heuristics.

Following will include a brief discussion of the testing methodology, and it's utility in validating the correctness of each implementation.

1.3 Code Structure

The `README.md` file contains instructions to build and run the code and corresponding tests.

The `src` folder contains each of the implementations.

The `test` folder contains the source code for each of the Boost.Test tests which include basic unit tests, and some more complex stress-testing. The tests are linked in separate executables to the `build/test/` folder when built.

2 Single Lock Algorithm

The first implementation was the very simple single-lock algorithm. Each instance has a single private mutex variable `lock`.

I decided to implement this very basic implementation as it is an easy way to convert sequential code which works without multi-threading to multi-thread supported code because the locks just work around the existing code with no other modifications except for locking and unlocking the global lock around accessing shared resources. Given my relative experience, this was useful for writing the initial tests as it's correctness is easier to justify.

The multi-set is stored as a `std::map` data structure, where the entries are stored as keys pointing to their integer multiplicity values. As defined in the cpp specification, accessing keys not yet stored in a map are initialized to 0, which means no special check for non-added entries is required because the first addition increments the default value 0 to 1.[1]

The shared mutex is locked before the shared map is accessed, and then unlocked before exiting the function.

This is a blocking algorithm, with waiting, because if one thread locks the mutex, then any other thread making any calls to the set will have to wait for the function to complete, including waiting for code which doesn't directly interact with the shared store to complete. This makes the performance non-optimal, as the c++ specification states that containers (such as std::map) are thread safe when accessing different elements.[2]

3 Multi Lock Algorithm

Following from the performance limitations of section 2, the next implementation designed was a multi-lock version. This introduces individual mutex locks for each element in the set.

These are kept track in a map data structure, with the elements as keys, and mutexes as values. This allows multiple elements to be modified by different threads safely, without waiting on a global lock.

However, when an element is not present in the element mutex map, accesses generate a new lock. This introduces a potential conflict, where two processes access a new value in the map at the same time, generating two locks. Therefore, an additional global mutex was introduced, which only locks when a new entry is added. This reduces wait-times as only new entries globally lock.

Therefore, the `MultiLock` implementation utilises this, using separate mutex locks for each elements in the set. These locks are stored in an additional map, linking elements to mutex locks. When an element is accessed, the relevant mutex is fetched and locked, allowing different threads to access different elements simultaneously.

4 Atomic / Wait-Free Algorithm

The goal of this implementation was to be wait-free. The simplest way to do so is to use atomic instructions on the shared data. This algorithm is also lock-free.

The `store` map structure had to be modified to store `atomic_int`, which avoids the issue where new additions break concurrency, as the creation on addition is also atomic, and so the first thread will create a new atomic, and the next will not be able to overwrite it.

5 Lock Free Algorithm

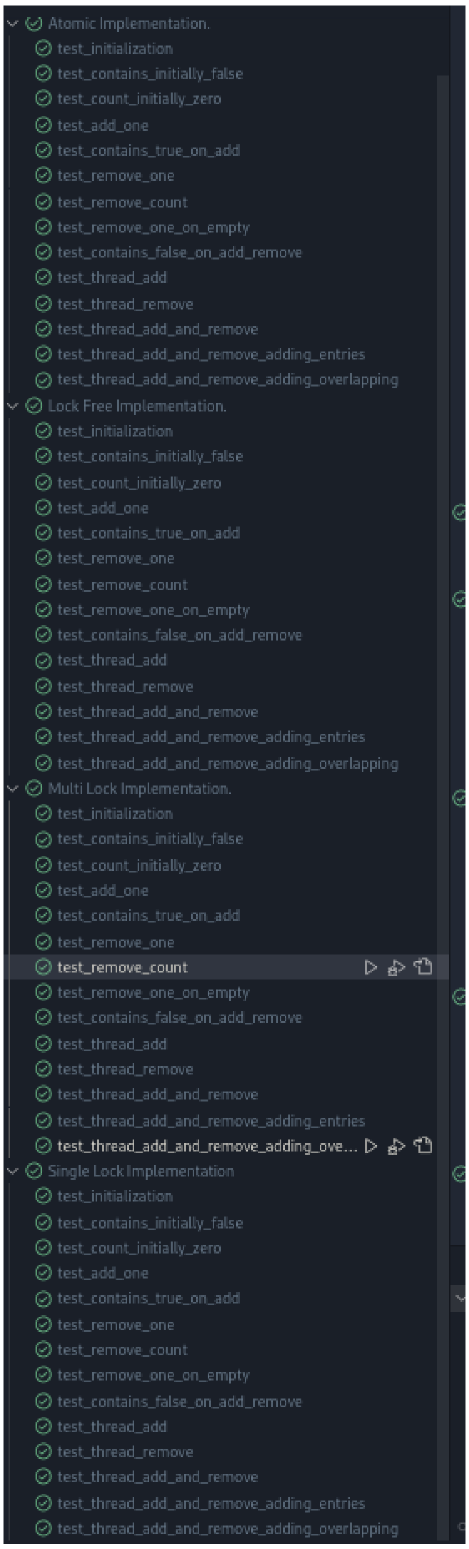
For the custom lock-free algorithm, the `atomic_compare_and_swap_weak` function was used, which checks whether the accessed value has changed since access and write. This is wrapped in a for loop, which retries the addition/removal until successful.

This means that this algorithm is lock-free, but not wait free, as threads must wait for the function to successfully read-write without the variable being modified.

6 Testing

Along with the logical argumentation given for each implementation. I used the testing framework Boost.Test to create unit tests with mutli-threading stress-tests to validate both the correctness and features (e.g. wait-free) of the algorithms.

A single `test/tests.cpp` file contains tests for a sequentially consistent multi-set. This allows the different implementations to be sourced in cmake creating a separate test executable for each algorithm. All implementations should behave the same in terms of end result, and this made adding new tests for all algorithms easy.



```
Atomic Implementation.
    test_initialization
    test_contains_initially_false
    test_count_initially_zero
    test_add_one
    test_contains_true_on_add
    test_remove_one
    test_remove_count
    test_remove_one_on_empty
    test_contains_false_on_add_remove
    test_thread_add
    test_thread_remove
    test_thread_add_and_remove
    test_thread_add_and_remove_adding_entries
    test_thread_add_and_remove_adding_overlapping
Lock Free Implementation.
    test_initialization
    test_contains_initially_false
    test_count_initially_zero
    test_add_one
    test_contains_true_on_add
    test_remove_one
    test_remove_count
    test_remove_one_on_empty
    test_contains_false_on_add_remove
    test_thread_add
    test_thread_remove
    test_thread_add_and_remove
    test_thread_add_and_remove_adding_entries
    test_thread_add_and_remove_adding_overlapping
Multi Lock Implementation.
    test_initialization
    test_contains_initially_false
    test_count_initially_zero
    test_add_one
    test_contains_true_on_add
    test_remove_one
    test_remove_count
    test_remove_one_on_empty
    test_contains_false_on_add_remove
    test_thread_add
    test_thread_remove
    test_thread_add_and_remove
    test_thread_add_and_remove_adding_entries
    test_thread_add_and_remove_adding_ove...
Single Lock Implementation
    test_initialization
    test_contains_initially_false
    test_count_initially_zero
    test_add_one
    test_contains_true_on_add
    test_remove_one
    test_remove_count
    test_remove_one_on_empty
    test_contains_false_on_add_remove
    test_thread_add
    test_thread_remove
    test_thread_add_and_remove
    test_thread_add_and_remove_adding_entries
    test_thread_add_and_remove_adding_overlapping
```

Figure 1: Test Suite Results

7 Evaluation

Overall, I have successfully implemented more than the required 3 algorithms, and justified their correctness in multi-threading applications. I have attempted to integrate my understanding of different strategies, and compare their utility in terms of wait-times and overall performance.

Given more time, it would have been good to implement more performance-focused algorithms.

References

[1] cppreference.com. *Containers > Map*. URL: `https://en.cppreference.com/w/cpp/container/map` (visited on 2024-02-20).

[2] cppreference.com. *Containers > Thread Safety*. URL: `https://en.cppreference.com/w/cpp/container#Thread_safety` (visited on 2024-02-20).