DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2018

Improving Performance of a Trading System through Lock-Free Programming

HARALD NG

JOSEF KARLSSON MALIK

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Abstract

Concurrent programming is a form of computing, where several computations are executed in overlapping time periods. This can improve a system's capability of handling growing amounts of work and execute faster on multicore processors. Lock is a usual tool used to ensure shared data is handled correctly. However, using locks could also have some performance disadvantages caused by its overhead and waiting time during high contention.

The company FIS believes a lock-free implementation using atomic operations could improve ability to handle growing amount of work and speed of a component in their trading system. Hence, the aim of this study is to provide insight of how impactful lock-free programming could be. This was achieved by developing a new version of the component and comparing its performance with the original lock-based implementation. The new implementation was developed by eliminating locks in the component and replacing them with lock-free data structures. However, a lock was still needed in one of the data structures, making the new implementation only partially lock-free. Results from tests performed directly on the component showed that the partially lock-free version performed better in some areas and worse in other. Furthermore, the partially lock-free implementation performed better in isolated tests which were used to measure parts of the component where direct tests could not be performed. This gives a sign of that a general performance improvement was achieved by using lock-free programming in the provided component.

Keywords – lock-free, concurrency, trading system, performance, atomic operations

Abstrakt

Concurrent programming är en form av programmering, där flera beräkningar exekveras i överlappande tidsperioder. Detta kan förbättra ett systems förmåga att hantera växande mängder av arbete, och dessutom kunna exekveras snabbare på flerkärniga processorer. Lås är ett vanligt verktyg som används för att säkerställa att data i delat minne hanteras korrekt. Användningen av lås kan dock påverka prestandan negativt. Detta är på grund av minimalkostnad som tillkommer vid användning av lås samt väntetid på låset som kan uppstå vid hög konkurrens.

FIS anser att en låsfri implementation baserat på atomiska operationer skulle kunna förbättra förmågan att hantera växande mängd arbete och hastigheten på en komponent i sitt tradingsystem. Syftet med denna studie är därför att ge insikt om hur effektiva låsfria program kan vara. Detta uppnåddes genom att utveckla en ny version av komponenten och jämföra dess prestanda med den ursprungliga, låsbaserade, implementation. Den nya implementationen utvecklades genom att eliminera lås med hjälp av låsfria datastrukturer. Dock behövdes ett lås i en av datastrukturerna, vilket innebar att komponenten endast var delvis låsfri. Resultat från tester utförda direkt på komponenten visade att den delvis låsfria versionen presterade bättre på vissa områden och sämre i andra. I de isolerade testerna dock, som användes för att mäta delar av komponenten där direkta tester inte kunde utföras, presterade den delvis låsfria versionen bättre. Detta ger en indikation på att en generell prestandaförbättring för den tillhandahållna komponenten uppnåddes med hjälp av låsfri programmering.

Nyckelord – låsfrihet, prestanda, tradingsystem, concurrency, atomiska operationer

Table of Contents

1 Introduction

Concurrent programming is a way of computing that has been utilized for decades when developing software [1]. Concurrency is when a processor handles multiple tasks simultaneously by switching between tasks after a certain amount of time [2]. This behaviour has many benefits on a program, such as improving a system's capability of handling growing amounts of work, so called scalability [3]. In recent years, processor manufacturers have been focusing more on multicore-architecture [4], [5]. Due to this reason, concurrent applications could, apart from increasing scalability, also execute faster on multicore processors.

There are two main paradigms within concurrency; *shared memory* and *message passing*. The shared memory paradigm is based on the tasks working on common shared data that is in the memory. Message passing is instead based on sending and receiving messages to handle shared data. This thesis focuses on the shared memory paradigm. [2], [6, p. 2], [7]

1.1 Background

An important aspect when implementing a concurrent program that uses the shared memory model is its correctness when multiple tasks are executed simultaneously. This is achieved by synchronization, which ensures only one task executed by the processor can access and modify shared data in memory at the time. This is usually done using a mechanism called *lock*, a special type of hardware-supported object that ensures only one process at a time can execute a specific block of code [8]. A process that manages to acquire the lock is the only process that gets to execute the specific block of code.

Although using locks is simple, it could introduce some problems which could cause performance issues [9]–[15]. This is where lock-free programming comes to use. Lock-free programming uses atomic operations for synchronization [16], [17] and therefore avoids locking [9], [15], [18]. Atomic implies that the operation is seen as a single operation by other processes [19], hence the operations are either totally completed or not executed at all. This is fundamental for lock-free programming, as it prevents partly finished states and errors caused by multiple processes accessing shared data in memory simultaneously [20, p. 5]. The possibilities of using lock-free programming has increased since the release of C++11, which supports atomic variables and operations [17], [21].

1.1.1 Fidelity National Information Services inc.

This study was conducted on behalf of the company *Fidelity National Information Services inc*, also known as *FIS*. FIS is an international provider of financial services, software and consulting. The company is the world's largest provider of transaction processing, core processing and card issuer

services to financial organizations. They are focused on payment processing, banking software, services and outsourcing solutions mainly on retail and banking. Their headquarter is located in Jacksonville, Florida, and currently has more than 53 000 employees worldwide. Currently they serve more than 20 000 clients in over 130 countries. [22]

As locks could give a certain expense in performance and other issues, FIS sees an opportunity for performance improvements on one of their lock-based components of their trading system using lock-free programming.

1.2 Problem

Although using locks for synchronization is correct, it could also have some disadvantages. *Deadlock, race condition, blocking* and *waiting* are some possible down sides of using locks. Deadlock is a state where the program is stuck permanently. This occurs when a set of processes waits for an event that can only be caused by one of the processes in that set [2], [23]–[25]. Race conditions could cause unwanted behaviours due to multiple processes accesses and modifies data at the same time [25], [26]. This could be caused by forgetting to lock or by locking the wrong lock. Race conditions could lead to devastating outcomes such as system crashes, corrupted data, or security problems [26]. Blocking and Waiting would not affect the correctness of the program, however they could cause performance issues. Blocking is when a process gives up the processor as it did not manage to acquire the lock. This allows the processor to handle another process, but switching between processes is time consuming [27], [28]. Waiting is when a process wastes processing power without actual progress by repeatedly trying to acquire a lock until it succeeds.

FIS believes that these problems could possibly be solved, or at least reduced, by implementing a lock-free solution for a lock-based component in their trading system. A lock-free implementation could provide improvement in speed and scalability [9], [13], [14]. However, lock-free programming also has its own drawback. The program must still remain correct, which could be challenging [9], [10], [14]. Hence, FIS have provided a lock-based component, which will be investigated and compared to a lock-free implementation.

1.3 Research Question

This thesis aims at investigating and documenting the impact of applying lock-free programming in a component of FIS' trading system. Hence, the question that is aimed to be answered is:

"How much improvement in performance could a lock-free implementation of a component in a trading system achieve, compared to one using locks?"

1.4 Purpose

The purpose of this thesis is to investigate how much performance improvement a lock-free implementation of the provided component could achieve compared to a lock-based implementation. This is accomplished by delivering a lock-free implementation of a component in FIS' trading system and comparing it with the lock-based implementation. The experiences and results of this study could give a hint of how much performance improvements could be achieved in relation to time and effort invested. This could be beneficial for FIS as well as other people that are considering doing an optimization through lock-free programming in similar systems.

1.4.1 Benefits, Ethics and Sustainability

The aim of this study is to provide insight of how impactful lock-free programming could be. This does not create any ethical dilemmas. However, it could have an impact regarding sustainability. Problems regarding sustainability are usually divided into three different areas: ecological, social and economic sustainability. [29]

An ecological sustainable system is defined as a system whose consumption of the earth's resources does not compromise the ability to recreate used resources [30]. From an environmental perspective, this thesis does not have a direct impact. However, the currently running software used in this study requires a lot of data to be stored on database servers. Database servers have some environmental impacts as they are energy consuming and produce a lot of heat and sound [31]. As this study is about improving the existing software and not developing a new one, it does not have any additional indirect impacts.

Social sustainability is the ability to satisfy all people and the planets physical and psychological needs. In a global sense, it is often associated to power, justice and rights. [32] This study does not affect this area.

Lastly, the economic sustainability can be defined in two different ways. The first definition is that an economic growth cannot occur at the expense of a decrease in natural and social capital. The second definition is that an economic growth may occur at the cost of natural and social capital if the total growth is higher than the cost. [33] The purpose of this study is to hint the potential performance improvement with the use of lock-free programming. The results are specific to FIS' platform and the design choices taken during the project. This implies that a lock-free implementation may not achieve similar increase in performance when used in other systems, which would result in wasted time and money. Due to this reason, this thesis could have an impact on the economic sustainability.

1.5 Goal

The goal of this project is to deliver a lock-free implementation of the component provided by FIS and to document its performance compared to the lock-based implementation.

1.6 Delimitations

Due to the complexity and size of developing a trading system, the primary focus in this thesis will be on improving a specific component in a trading system provided by FIS. Another delimitation is the choice of programming language. Since concurrency is supported by many different programming languages, it is impossible to determine which language performs best regarding concurrency. Moreover, real-life performance is often problem specific [34]. In this thesis, C++11 is the programming language that will be used. This is due to that the program provided by the company FIS is written in C++11.

1.7 Outline

The rest of this paper is structured as follows: Chapter 2 presents necessary theoretical background for the reader to understand the thesis. This consists of information about concurrency, lock-free programming and description of the provided component from FIS. In chapter 3, the methodologies and methods used in the project will be described. The intention is to give an overview of different research strategies used in order to be able to answer the research question. Chapter 4 presents the practical work that was done using lock-free programming. The design and implementation of lock-free data structures will be explained and discussed. The results gathered to compare the performance of the lock-free and the lock-based component is presented in Chapter 5. Chapter 6 discusses and motivates the gathered results. It also discusses whether the goal was reached and if the research question was answered. Furthermore, the research methods from Chapter 3 will be revisited and reflected upon. Lastly, Chapter 7 concludes the thesis, discussing future uses of the results and possible future research about lock-free programming.

2 Concurrency, Lock-free Programming and the Provided Component

This chapter covers the theoretical background that is required to understand the content of the thesis. Section 2.1 introduces concurrency and relevant areas of it for this thesis. Section 2.2 presents lock-free programming. Furthermore, Section 2.3 presents relevant lock-free data structures which are created by others. This is followed by a description of a trading system in general and the component that was worked with in Section 2.4. At the end of the chapter, there is a section presenting previous work conducted by others that is related to the thesis.

2.1 Concurrency

To be able to understand what concurrency is, threads and processes must first be described. A process is essentially a unit of computation [35] consisting of its own code, memory space and instruction pointer [2], [16], [35]. Threads are spawned by a process and is a sequence of instructions which the processor can execute. The difference between processes and threads is illustrated in Figure 2.1.

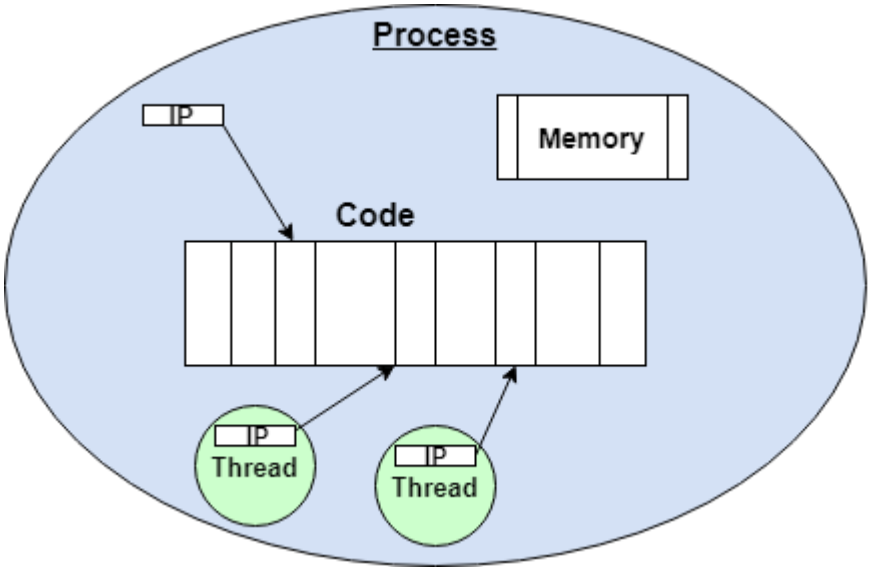


Figure 2.1. Illustrations of the difference between processes and threads. Each thread shares the memory of the process but has its own instruction pointer (IP). Figure created by the authors.

As the threads are spawned by a process, the code and memory are shared between threads in the same process. This implies that threads could execute any part of the process code, including the ones being currently executed by other threads [2], [16] .

Concurrency is having several processes or threads share a single processor or a multiprocessor [2]. The sharing is done through the processor switching between processes or threads after executing it for a certain amount of time. Concurrent programming can be used to improve performance or/and

scalability [2], [36]. Scalability is the ability of a program to handle increased amount of work. This could be achieved through concurrency because concurrent programs consist of processes and threads that run on processors. By increasing the number of processors in the hardware, the amount of workload that can be handled will increase proportionally. [2], [36]

The ability to handle multiple processes or threads concurrently allows the processor to switch between tasks while waiting in another task. This would increase performance through better utilization of resources [2] and increased responsiveness. The speed of a concurrent program increases if it runs on multiprocessors, as the processes or threads could be executed in parallel, so called parallelism. Figure 2.2 illustrates concurrency and parallelism.

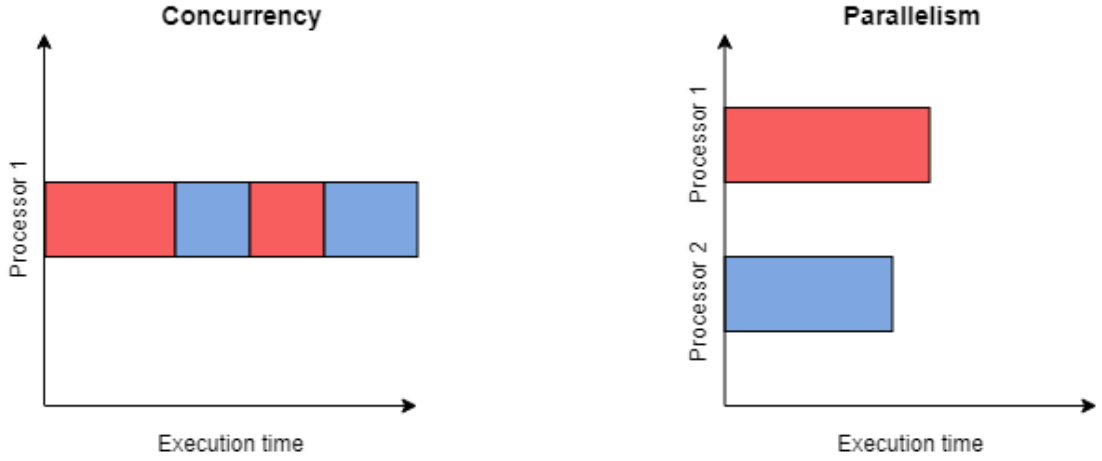


Figure 2.2. Concurrency and Parallelism. Each task is represented by a colour. The tasks are split up and switched between when executing concurrently on a single processor. A parallel execution allows the tasks to be executed simultaneously on different processors. Figure created by the authors.

A concurrent program could theoretically run N-times faster on a multiprocessor with N cores, as the work could be divided and executed by N separate cores [2], [36].

2.1.1 Synchronization

As threads/processes do not run synchronously with respect to each other, it is difficult to keep a concurrent program correct as it runs in an unpredictable manner. As a result of this behavior, problems can occur. An example is if two or more processes or threads simultaneously accesses the critical section, which is part of the code that should only be executed by one process as it accesses shared memory. If that happens, it may cause a situation where the values of variables are unpredictable. This is due to threads are reading half-completed states of operations by other threads and writing to the variable. An example of this is seen at Figure 2.3.

Wanted behaviour

Thread 1	Thread 2	Variable
		0
Reads variable		0
Increments		0
Writes variable		1
	Reads variable	1
	Increments	1
	Writes variable	2

Unwanted behaviour

Thread 1	Thread 2	Variable
		0
Reads variable		0
	Reads variable	0
Increments		0
	Increments	0
Write variable		1
	Writes variable	1

Figure 2.3. Race condition occurs when multiple threads try to increment a shared variable without synchronization. In this case, the variable could end up with the value 1 or 2. Figure created by the authors.

This is also known as race condition. To solve this problem, synchronization tools are needed. These provide mutual exclusion, i.e. only one thread can execute a desired part of the code at a time.

2.1.2 Locks

Lock is a synchronization tool that provides mutual exclusion. The purpose of locks is to protect the critical section from being executed by multiple threads simultaneously. A lock has two states; *locked* and *unlocked*. A thread that manages to lock the lock will be the only thread executing the critical section. To allow other threads to execute the critical section afterwards, the lock has to be unlocked. Although locks functions well for synchronization, it could also introduce some problems:

- **Deadlock**: Deadlock is a state where the program is stuck permanently. This can occur when multiple locks are used, and the processes or threads tries to acquire each other's locks while holding its own lock. An example of this is shown in Figure 2.4.

Figure 2.4. Deadlock caused by two different processes trying to acquire two locks. Process A will never progress as it tries to get lock_2 which is

held by process B, and vice versa for process B. Figure created by the authors.

The program will be stuck forever as the processes will never manage to acquire its desired lock, due to it being held by another process or thread. [2], [23]–[25]

- **Waiting or Blocking**: There are two types of locks, which differ in what the process or thread does if it does not acquire the lock. *Spin locks* are waiting locks, i.e. a process or thread will keep trying to lock a lock until it succeeds. This results in wasted CPU-time and high memory contention. Blocking is when a process gives up the processor to allow other to use the CPU. However, switching between processes is expensive as the CPU must save the current state of the process and then switch to another. This is called context switching. [37, p. 4]

- **Lock contention**: There are two types of lock contention, one is caused by having large critical sections, the other one is contention by frequent lock requests [38]. With large critical sections, other threads must wait long before acquiring the lock [11]. The issue with high frequency lock requests occurs when threads located on different cores are requesting the same lock frequently. Most of the CPU's resources will then be spent on context switching rather than the actual work [38].

- **Lock overhead**: Lock overhead is the extra CPU resources required when using locks. Example of this is the CPU time for initializing and destroying locks and the time spent for acquiring and releasing the lock [11], [39].

2.1.3 Concurrent programming in C++11

C++11 was published in 2011 and was previously known as C++0x. C++11 introduced new techniques and improved concurrent programming compared to earlier versions. The class *thread* from the standard library can be used when developing a concurrent program. In C++11, a thread is a single thread of execution. Threads makes it possible to execute multiple functions concurrently. [40], [41]

2.2 Lock-free Programming

The formal definition of a *lock-free* method is that it guarantees progress of at least one process or thread executing it [42]–[44]. This implies that a single process or thread will never prevent other processes or threads to progress through its own lock-free operations [18].

The key concept of lock-freedom is to think in transactions. As there are no locks used for synchronization, A transaction has four main properties; *atomicity, consistency, isolation* and *durability* (ACID). Atomicity implies that

a transaction is all-or-nothing, i.e. other processes or threads cannot see a partially updated state. Consistency is that data will be taken from one consistent state to another through a transaction. Isolation means that two threads will never simultaneously work on the same data. Durability implies that a committed transaction will never overwritten by a second transaction due to it missing the results of the first transaction. [9]

Lock-freedom could in practice be achieved by using atomic operations instead of locks for synchronization purpose. The atomic operations are fundamental for lock-free programming as they are seen as a single operation by other processes or threads, hence acting as the transactions. This prevents partly completed states and therefore also problems caused by race conditions. An example of atomic operations is *Compare-And-Swap* (CAS). CAS can set the value of a variable in one atomic step, if it at that time holds a value specified as an argument to the operation, as depicted in Figure 2.5. [44], [45]

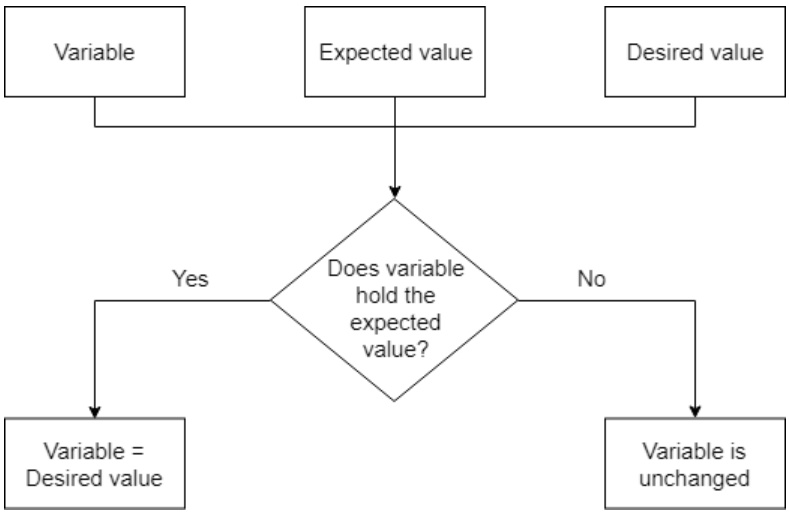


Figure 2.5. Compare-And-Swap (CAS). *Variable*, *Expected value* and *Desired value* represent the parameters of the operation. Figure created by the authors.

Using normal operations for the same purpose would require three operations: read, compare and writing the variable. This is where problems could occur if not protected by locks, another thread can access or modify the variable between the operations and thus causing race conditions. But as CAS is an operation that will be completed in a single step, there will be no partly completed states. This implies that no race conditions will occur even without using locks.

However, there are also disadvantages with atomic operations. There exists only a limited set of atomic operations [45] and they are expensive compared to normal read and write operations [44].

Lock-freedom should not be interchanged with *wait-freedom*, which is a stronger level of lock-freedom. The definition of wait-freedom is that every call

will finish in a finite number of steps without interruption [9], [42], [44]. Hence, a method that is wait-free is also lock-free, but not vice versa.

2.2.1 Atomics in C++11

Lock-free programming in C++11 is based on the *Atomic operations library*. It contains a set of atomic types and operations that are free from race conditions [46]. This assures that the behaviour of multiple processes accessing shared data in memory simultaneously is well-defined [21]. The atomic operations enables synchronization between processes without using locks. This is crucial as other processes' operations cannot interfere or interrupt an operation from when it is started until it has finished execution of its steps [47]. An example of how atomics can be used for synchronization is depicted in Figure 2.6.

Pushing a value into list

Using locks

```
mutex sharedVar;

void push_front(T t){
    auto p = new Node;  // create the new node
    p->t = t;           // set its element value

    //Lock since the shared value read and updated is accessed
    sharedVar.lock();
    p->next = head;     // set its place in the list
    head = p;           // publish it at the head
    sharedVar.unlock();

}
```

Using atomics

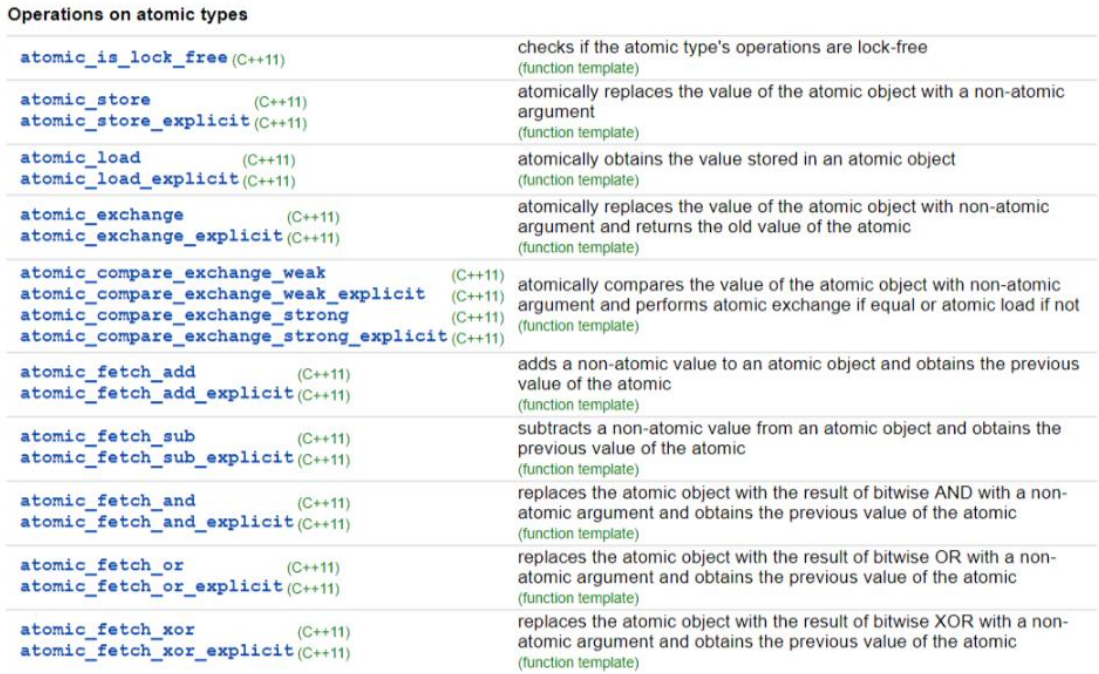
```
void push_front(T t) {
    auto p = new Node;           // create the new node
    p->t = t;                    // set its element value
    p->next = atomic_load(&head);// set its place in the list and
    while( !head.compare_exchange_weak(p->next, p) )
    { }                          // try to swap it in until successful
}
```

Figure 2.6 An example of how to use atomics in C++11 instead of locks. Assume that the following functions are called multiple times by different processes. Both will have the same outcome, only difference is that the one using atomics is lock-free. Figure created by the authors.

Instead of using `lock()` and `unlock()`, an atomic variable and its operations can be used in the critical section to avoid race condition. The reason for that is that an atomic operation is seen as a single operation by other processes or threads. The `compare_exchange_weak()` in the atomic version, works as a

CAS. The CAS loop will loop until it gets to be the one that updates the head from the expected value to the desired. For each iteration the expected value will be updated.

In the Atomic operations library, there are a few atomic read-modify-write operations (RMW-operations), as shown in Figure 2.7. These are operations that could read a variable in the shared memory and simultaneously write a different value to it [48].



Operations on atomic types

`atomic_is_lock_free` (C++11)	checks if the atomic type's operations are lock-free (function template)
`atomic_store` (C++11) `atomic_store_explicit` (C++11)	atomically replaces the value of the atomic object with a non-atomic argument (function template)
`atomic_load` (C++11) `atomic_load_explicit` (C++11)	atomically obtains the value stored in an atomic object (function template)
`atomic_exchange` (C++11) `atomic_exchange_explicit` (C++11)	atomically replaces the value of the atomic object with non-atomic argument and returns the old value of the atomic (function template)
`atomic_compare_exchange_weak` (C++11) `atomic_compare_exchange_weak_explicit` (C++11) `atomic_compare_exchange_strong` (C++11) `atomic_compare_exchange_strong_explicit` (C++11)	atomically compares the value of the atomic object with non-atomic argument and performs atomic exchange if equal or atomic load if not (function template)
`atomic_fetch_add` (C++11) `atomic_fetch_add_explicit` (C++11)	adds a non-atomic value to an atomic object and obtains the previous value of the atomic (function template)
`atomic_fetch_sub` (C++11) `atomic_fetch_sub_explicit` (C++11)	subtracts a non-atomic value from an atomic object and obtains the previous value of the atomic (function template)
`atomic_fetch_and` (C++11) `atomic_fetch_and_explicit` (C++11)	replaces the atomic object with the result of bitwise AND with a non-atomic argument and obtains the previous value of the atomic (function template)
`atomic_fetch_or` (C++11) `atomic_fetch_or_explicit` (C++11)	replaces the atomic object with the result of bitwise OR with a non-atomic argument and obtains the previous value of the atomic (function template)
`atomic_fetch_xor` (C++11) `atomic_fetch_xor_explicit` (C++11)	replaces the atomic object with the result of bitwise XOR with a non-atomic argument and obtains the previous value of the atomic (function template)

Figure 2.7. List of RMW-operations in C++11. Retrieved from [46] using the license CC BY-SA 3.0 [49].

The reason for why there are not too many other RMW-operations is that every RMW-operations can be implemented through `compare_exchange_weak`, hence it is the most essential RMW-operation [48]. `compare_exchange_weak` takes a minimum of two arguments, in this case called *expected* and *desired*. If the variable holds the value *expected*, it will be written to *desired* and the operation returns true. Otherwise, the value of the variable will be written to *expected* and the operations returns false [9], [48], [50].

2.2.2 Facebook Open Source Library

Facebook Open Source Library, also known as Folly, is an open source library developed and used by Facebook. Folly is a collection of different C++11 libraries with the purpose of complementing existing libraries. It contains a set of lock-free data structures based on atomics in C++11. These facilitate the development of lock-free programs. [51]–[53]

2.2.3 Boost

Boost is a collection of peer-reviewed portable C++ source libraries which is highly regarded and used worldwide by companies and research groups [54]. Many of Boosts founders are on the C++ standards committee and several Boost libraries has been included in the C++11 standards and even more libraries has been proposed for C++17 [55], [56]. Boost has a set of lock-free data structures that could be useful for lock-free programs [57].

2.2.4 Thinking in Lock-Free Manner and the ABA-problem

An important way of thinking when programming in a lock-free manner is how the atomic operations work. Atomic operations guarantee no half-completed states; however, operations could occur between two atomic operations. This implies that although an atomic operation is followed by a second atomic operation on the next row in the code, there could be other operations that has happened between the first and second atomic operation. An example of this is shown in Figure 2.8.

```
int i = 1;
if (atomic_read(i) == 1){
  // Other atomic operations can happen BETWEEN the atomic reads and writes.
  //In this case, if an atomic_write() was performed here by another thread, the logic will be incorrect
  atomic_write(i, 2);
}

// Hence, CAS should be used instead
int i = 1;
int expected = atomic_read(i);
int desired = 2;
// try until CAS succeeds i.e. returning true. If CAS fails, 'expected' will be updated to the latest read value.
while(!compare_and_swap(i, expected, desired));
```

Figure 2.8. Atomic operations guarantee no half-states. However, atomic operations can happen between two atomic operation. Figure created by the authors.

Due to the possibility of atomic operations happening between two dependent atomic operations, the two dependent atomic operations must be executed in one atomic operation. For example, an atomic read that will be used in an if-statement to then be written, must be replaced by one single Compare-And-Swap operation. However, CAS are not completely problem free either. The ABA-problem is a common problem that could occur when using the atomic Compare-And-Swap operation in lock-free programs. The ABA-problem is when a CAS is wrongly succeeded, as depicted in Figure 2.9.

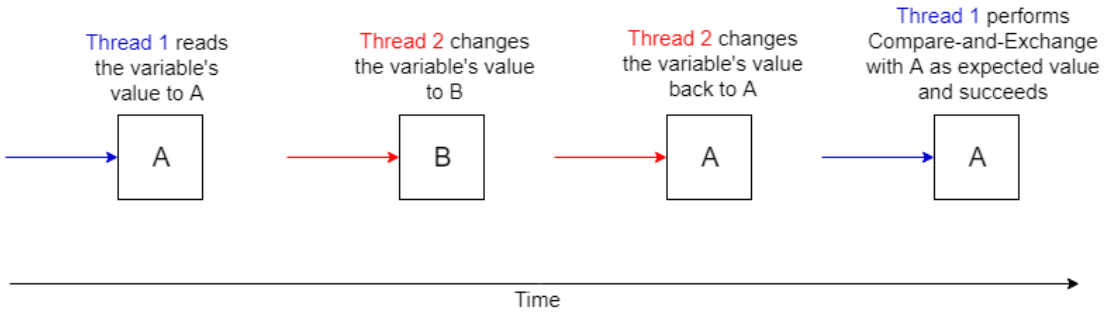


Figure 2.9. The ABA-problem. Thread 1 will not recognize the changes done by thread 2. Figure created by the authors.

To perform the CAS, the variable must first be read to use it as the *expected* argument in the operation. Then on the next line a CAS operation can be called, using the *expected* value and a desired value as arguments. The read and CAS are both performed atomically, however, there could be operations executed on the same variable by other threads between these operations. If the other threads then write back the same value as the first thread, the CAS will succeed, even though the variable has been manipulated. The thread will wrongly assume that the variable is in an unchanged state. This would be problematic for missing side effects or for example removing and adding elements in linked lists. [9], [58], [59]

2.3 Lock-free Data Structures

When using data structures in a concurrent program for storing and removing shared values, problems such as race condition and deadlock can occur if there is no synchronization. Therefore, locks are needed when using operations that will modify the data structure. However, if the data structure is implemented in a way that it can handle concurrent behaviour, locks can be avoided when performing the certain operations, therefore making it lock-free [10]. This section presents lock-free data structures that can be used to prevent the use of locks.

2.3.1 Atomic Single Producer Single Consumer Queue

An atomic single producer single consumer (SPSC) list is presented in Herb Sutter's article *Writing Lock-Free Code: A Corrected Queue* [60]. It is a singly-linked list, where each element has a pointer `next` that points to the next element in the list. The list allows one thread to pop elements at the front and another thread to push elements to the back simultaneously, by arranging them to always work on different parts of the list. This is based on having three pointers: `first`, `divider` and `last`. The first element in the list is the one pointed by `divider`. The last element in the list is the one pointed by `last`. `first` is used to clear up elements that have been popped, but not removed yet.

The thread that only pushes elements is called the producer. The producer owns all elements before `divider`, the `next` pointer in `last` and the ability to update `first` and `last`. The thread that only pops elements, the consumer, owns all the other elements and the ability to update `divider`. This is depicted in
Figure 2.10.

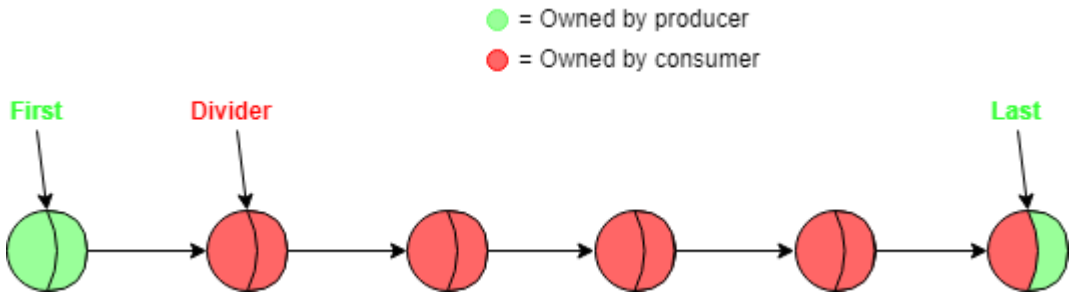


Figure 2.10. The design of the Single Producer Single Consumer list. The producer and consumer work on different parts of the list. Figure created by the authors.

`pop_front()` will only be called by the consumer thread. To check if the list is empty, the consumer performs a normal read to divider and an atomic read to `last`. If `divider` equals `last`, then the list is empty. Note that reading `divider` does not have to be atomic, as `divider` can only be changed by the consumer thread itself. However, `last` could be changed by the producer thread, and therefore must be read atomically. If the list is not empty, `divider` will be set to its next element atomically. It is important that write is performed atomically, as the producer thread can also read `divider`. Another important part of the method is that the consumer only moves `divider` rather than actually removing the element. The element will instead be removed at the next call of `push_back()`, as producer is the only thread allowed to update `first`. This is to assure that consumer and producer will work on different parts of the list.

When the producer calls `push_back()`, it will set the `next` pointer of `last` to the new element. Then, it will atomically write the new element to `last`. This publishes `last`, which implies that the consumer will now see the new updated `last`. Again, it is crucial that `last` is written atomically, as the consumer thread reads it in `pop_front()`. After each `push_back()`, the producer thread will remove the already popped elements by looping and deleting each element until `first` is `divider`.

2.3.2 Boost Multiple Producer Multiple Consumer Queue

In the library `boost::lockfree` there is an atomic queue that can handle multiple producers and multiple consumers. The advantage of this queue is that it provides a `push()` and `pop()` that can handle multiple threads performing it concurrently in a lock-free manner. However, a disadvantage that comes with

this data structure is that to keep the operations lock-free, the queue must be fixed-sized. If the queue is not fixed-sized and the memory allocated to the queue is exhausted, the push function must allocate new memory. This may not be lock-free. [61] [57]

2.3.3 Folly Atomic Unordered Map

A map is a data structure which contains pairs of keys and values. A key is mapped to a value. In C++11, there is a `std::unordered_map` that works as described. A lock-free version of the `std::unordered_map` called *Atomic Unordered Map* exists in the open source library Folly. The main advantages of the Atomic Unordered Map were the lock-free insertions and wait-free reads [62]. Furthermore, it allowed keys and values of arbitrary type. However, the Atomic Unordered Map has some limitations as well; entries are not erasable, inserted values cannot be modified and the map cannot be resized [62].

2.4 Trading System

The idea of trading is to make profit through business transactions. A business transaction is an interaction between a person or another business to an enterprise, where products, information, service requests or money is exchanged. The communication between the participants is often done over a computer network, with the recordings of required data stored on computers for better reliability and less cost [5]. A trading system is a group of specific parameters and conditions combined to create buy and sell signals [6], [7]. These parameters and conditions could be seen as trading strategy [6], as they specify important conditions such as avoiding great losses or selling when satisfactory profit has been made. Trading systems have also other important tasks such as managing orders, linking many buyers and sellers, providing up-to-date market data and analysing risks [8].

2.4.1 Component in FIS' System

The component that was provided was a part of a platform consisting of an internal market, several simulated external markets and a client program called OMNI. External markets are exchange markets such as *Eurex* and *SWXess*. The internal market works as a hub of external markets, from which clients of FIS can make orders through OMNI. The internal market places the orders on the external market, which will then send two types of replies to each order. One is a direct reply to the client. The other is a *multicast* (MC)-message. This MC-message that confirms this order has been placed will also be received by other clients in the internal market which has requested for updates. The MC-messages are sent through the network protocol *Transmission Control Protocol* (TCP). Although TCP is intended for reliable transport between two endpoints [63] and hence does not support multicast, it is still named "multicast" by FIS as it has the functionality of sending the same message to multiple clients. The process of placing an order from OMNI and getting replies is illustrated in Figure 2.11.

Figure 2.11. Clients makes orders on OMNI, which will be sent to the internal market (IM). The internal market makes the orders on the external markets(EM's), which replies to each order message. These replies will get forwarded to the clients. Figure created by the authors.

The component where this study was focused on handled the incoming MC-messages from OMNI and external markets to the internal market, as depicted in Figure 2.12.

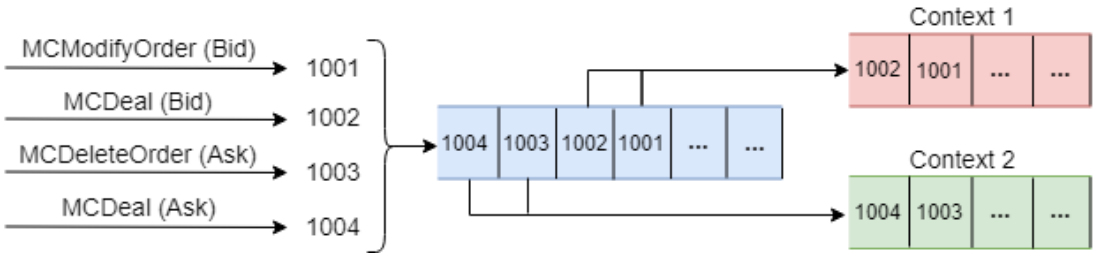


Figure 2.12. The worked component assigned incoming messages to the right context. Figure created by the authors.

The messages form a queue, which will then be handled by different contexts. A context could be seen as a working thread which processes the MC-message. An MC-message is assigned to a context via an unordered map. Each MC-message has some fields in its data structure which will be hashed to a value that maps to a context. Some MC-messages are related to each other, and therefore must be executed by the same context. However, when a message is assigned to a context, it is not removed from the queue until it has actually been executed. This is to ensure the order between messages are kept as they come in. The context will instead iterate through the queue and remove completed messages once it has processed its own messages. The reason for this is reliability. If a system crash occurs, then next session will recover from the last processed MC-message.

The current implementation of this component is lock-based. It uses a lock to ensure only one external market at the time can do two things. One of them is assigning a context and enqueuing a message to the work queue. The other one is dequeuing. Each context also has a lock for adding MC-messages to its queue.

2.5 Related Work

There exist several studies comparing performance of lock-free and lock-based implementations of known data structures. A study by Hunt *et al.* [64] measured and compared the performance of lock-free and lock-based implementations of *FIFO-queue*, *double-ended queue* and *sorted linked list*. The used benchmarks simulated high contentious workloads for the data structures. In general, the results were varying. The lock-free sorted linked list got a linear speedup as the number of threads grew. The lock-free FIFO-queue

gained about 50% speedup when the number of threads were beyond the hardware limit, otherwise it performed worse than the lock-based implementation. The lock-free double-ended queue performed worse than a lock-based implementation in most cases.

In another study by Sundell [47], a lock-free concurrent *priority queue* based on a self-designed lock-free *skip list* is presented. This was compared to some of the most efficient lock-based implementations of priority queues known at the time. The result showed that the lock-free version scaled very well and outperformed the corresponding lock-based implementation with three threads or more.

Cederman *et al.* [44] have investigated the performance of a *multiple-producer multiple-consumer queue* implemented in a lock-based and a lock-free way. In the result, the lock-free queue performed better than the lock-based one regardless the amount of threads and for both high and low contentions. Additionally, the lock-free queue had better scalability.

In a study made by Herlihy [65], a method for designing non-blocking and wait-free implementations is proposed. In this study, experimental results showed that the non-blocking and wait-free algorithms performed better than the common locking techniques used at that time and it was also competitive with the more refined locking techniques.

3 Methodologies and Methods

The following chapter presents the research methods and process used in this study. Section 3.1 describes and motivates the used research methods. Section 3.2 presents an overview of the research process. This is followed by more in-depth descriptions of each phase in the research process. Section 3.3 presents how the data collection phase was carried out. Section 3.4 describes the design and implementation phase. Lastly, the evaluation method is described in section 3.5.

3.1 Research Methods

Research methods are usually divided into *qualitative* and *quantitative* methods. Qualitative methods are associated with opinions, observations and experiences [66], [67]. Common ways to gather qualitative data are interviews and participation [67]. Quantitative methods are associated with numbers and measurable data that could be gathered by experiments and surveys [66], [67].

The choice of research method is dependent on the topic and the research question. However, the qualitative and quantitative methods are complementary and could therefore be combined [66]. In order to gain the required knowledge to be able to answer the research question, both a quantitative and a qualitative method were used in this study. The quantitative data was gathered through performing tests and benchmarks of the completed component. The qualitative data was collected through a literature study and an interview.

3.1.1 Literature Study

A literature study is a process of gathering information about a specific topic from various academic sources such as books, articles and thesis. The objective of a literature study is to situate the chosen topic within the context of other researches made in the same field [68]. This allows the researcher to identify what has been done before, what is already understood about this topic and if similar researches already have been conducted. More importantly, the literature study will identify the gap between the topic and existing literature [68] [69].

3.1.2 Interview

Interviews are particularly useful for getting detailed information of a person's experiences as it is more personal and the interviewer can ask follow up questions [70]. There are different kinds of interviews, one of them being *semi-structured* interview, which was the one used in this study. Semi-structured interview is when the interviewer prepares a set of questions that will be asked, but also uses follow up questions during the interview [71]. The purpose of the interview was to learn from experienced software engineers about common

problems that could occur when using lock-free programming in trading systems.

3.2 Research Process

The work that formed this thesis was divided into four phases; *Data collection*, *Design and Implementation*, *Tests and Evaluation* and *Summarization*. A phase consisted of several stages and different types of work were conducted in each phase. The research process is illustrated in Figure 3.1.

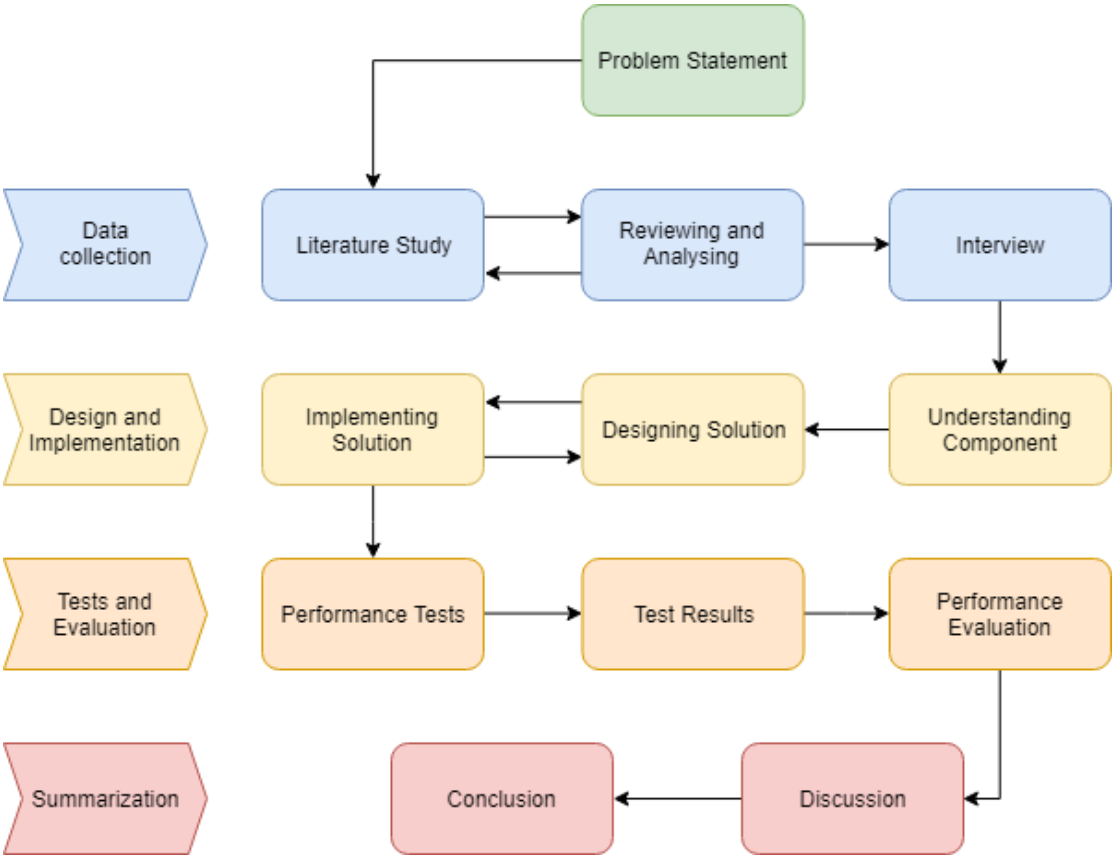


Figure 3.1. Outline of the research process. Each phase is represented by a colour. Arrows going back and forth between two stages represents an iterative process. Figure created by the authors.

The first phase had a theoretical focus, which consisted of literature study and interview. The goal was to create a strong foundation with knowledge in relevant fields, which would act as a useful preparation for the development phase. When enough background knowledge was obtained, the work continued with the development phase. To be able to develop a lock-free implementation on the existing component, an understanding of the component had to be gained first. This was achieved through analyzing the code and explanations by the original developer. When sufficient understanding of the component was gained, the next stages of the development phase could be continued with. These were design and implementation stages, which were conducted in an iterative manner. This was caused by finding possible improvements during the

implementation. The next phase was gathering results through performing benchmarks and tests. The results were evaluated and discussed, which lead to conclusions in the last phase.

3.3 Data collection

This section describes the methods used to gather background data, which resulted in a foundation for the practical work carried out later.

3.3.1 Literature Study

Initially, a literature study was conducted to find relevant academic sources for the thesis. Appropriate sources were found by utilizing the *KTH Library* search tools along with *Google Scholar*. The sources were then reviewed to decide whether they were relevant to the thesis or not. More related work could also be found while reading the initial sources. These often discuss relevant work done by others. This became an iterative process as once more knowledge was obtained, new topics got discovered and needed to be examined.

3.3.2 Interview

The literature study gave insight to this research area, however, more practical questions about lock-free programming remained. To answer these questions, an interview with two senior developers at FIS, Roger Persson and Björn Törnqvist, was performed. They explained that lock-free programming is a new and modern technology that is useful in practice. It is an optimistic approach that if a conflict occurs, then the threads will back off rather than keep spinning and trying to gather a lock. This avoids a reoccurring problem when implementing lock-based solutions: deadlocks. There is also the advantage of avoiding the overhead when using locks. Other relevant technologies used in FIS' system was also briefly explained. [72] These provided basis for the content in Chapter 2.

3.4 Design and Implementation

When the data collection phase was completed, enough knowledge about this topic was gained to be able to design and implement a lock-free solution. This section presents the methods used for designing and implementing a lock-free solution.

3.4.1 Design of Lock-free Solution

Although the work aimed at improving an existing component, some parts of the component needed redesign to be able to implement it in a lock-free manner. However, before that, it was important to fully understand the component worked with so that the right approach to the problem could be carried out. Both existing data structures and algorithm had to be modified.

- **Data structures**: Existing data structures needed modification to be able to support a lock-free implementation with atomic operations. Knowledge and understanding of lock-free programming and atomic operations was a key part in redesigning the data structures.

- **Algorithm**: As lock-free data structures have some restrictions compared to data structures that are intended to be used sequentially or with locks, some redesigning of the algorithm was needed. This required good understanding of the component and tradeoffs made.

3.4.2 Implementation of Lock-free Solution

The implementation of a lock-free solution was done through an iterative process with testing and debugging. Testing of the component was important to confirm its correctness. As more testing was conducted, new aspects and problems of the component were discovered, which would require redesigning to solve.

3.5 Evaluation Method

To evaluate the improvements of the lock-free implementations, certain method of evaluations was needed. The method needed and used in this project was summative evaluation. Summative evaluation is conducted at the final stage of a study when everything is complete to measure the efficiency and usability of the finished product. [73], [74]. The points focused when conducting the method on the lock-free implementation was:

- Speed: The time it takes for an MC-message to be processed.
- Scalability: The time it takes for an MC-message to be processed with an increasing number of external markets.

The summative evaluation was carried out by conducting performance benchmarks. The benchmarks consisted of having 100 sell orders of a stock placed evenly on N-number of external markets. Then, a buy order with 100 of the same stock with the same price was placed on the internal markets. The buy and sell orders will then be matched to create deals that will generate MC-messages to the internal market. The time to process each MC-message in the worked component was recorded. The tests were performed five times each and on 1,2,4 and 8 external markets. The same benchmark was also carried out on the lock-based version to be able to do a comparison.

4 Development of Partially Lock-free Solution

This chapter describes in detail the process of developing a corresponding solution of the provided component using lock-free programming. Due to some restrictions which will be described in Section 4.2.4, a completely lock-free solution could not be achieved. Therefore, making the component only partially lock-free. Section 4.1 describes how an understanding of the component was gained to create a basis for designing and implementing the partially lock-free solution. Section 4.2 presents the design of the partially lock-free solution. Lastly, Section 4.3 presents how the solution was implemented.

4.1 Analysis of the Lock-Based Component

Having got a brief introduction of the provided component, more analysis was needed. Identifying where the locks are placed and why, was necessary to fully grasp how a lock-free design was to be conducted. By performing test runs and using *break points*, the component could be analyzed step by step, facilitating the understanding of the component. Analyzing certain functions and data structures was also simplified using the functionality *step into*. *Step into* allows one to go in to a certain function call/data structure to see what the program performs/reads. The test runs were performed by placing simple orders on the client application OMNI, which would then be traced in Visual Studio.

4.1.1 Overview of the Component

Through the analysis of the component, an overview of how the provided component worked could be visualized, which is seen in Figure 4.1.

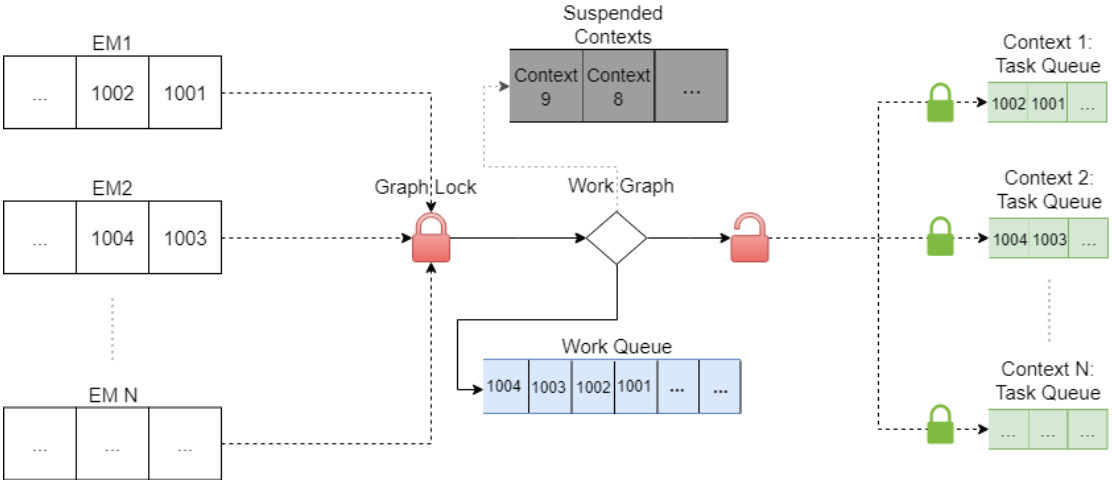


Figure 4.1. Overview of the component that was worked with. Figure created by the authors. EM represents external markets. Figure created by the authors.

The provided component has the task of handling the incoming MC-messages from external markets. The messages are placed in a queue called *Work Queue* to keep track of which received message were not executed yet. The threads that would execute the MC-messages are called *contexts*. Multiple MC-messages

could be assigned to the same context. Hence, each context has a *Task Queue* that is protected by a lock. An MC-message is assigned to a context through the unordered map *Work Graph*. If there are too many incoming MC-messages at a time, contexts can be suspended. These will be placed in the *Suspended Contexts* queue. *Graph Lock* is used to protect *Work Graph, Work Queue* and *Suspended Contexts* from being accessed by multiple threads at the same time.

4.1.2 Contexts, Task Queue and the Suspended Contexts Queue

A *Context* is a working thread that will execute the MC-message. This implies writing the content of the MC-message to the database of FIS. However, if all the database connections are busy, the context will be suspended. The context will then be added to the *Suspended Contexts* queue. Suspending and resuming contexts from Suspended Contexts also require having Graph Lock.

Each context has a *Task Queue* containing the functions to execute the MC-messages. Multiple threads can add MC-messages to the Task Queue of each context due to the MC-messages can be moved from a Task Queue of one context to another. This occurs when there are MC-messages that are dependent on each other. These messages must be executed by the same context and will hence be moved from multiple Task Queues to one common. However, only one thread at a time can pop the elements from each Task Queue. The messages are either executed or popped and added to another Task Queue. Hence, the Task Queue of each context was also protected by a lock.

4.1.3 Work Graph

An incoming MC-message gets assigned a context through the *Work Graph*. The Work Graph is an unordered map which uses the identifiers of MC-messages as keys and contexts as values. The identifiers of MC-messages are not unique, as some messages are dependent of each other and should execute in a certain order by the same context. Hence, the Work Graph functions as a look up table of MC-messages and contexts. If the incoming message is unrelated to other contexts executing MC-messages, it will be assigned to a new context. If another context is executing a dependent MC-message, then the MC-message will be assigned to that same context. The last possible case is if two independent MC-messages are being executed by two different contexts, and the incoming MC-message is dependent on these two previous messages. Then the incoming MC-message will have to wait until the two previous messages are finished before being executed by a new or existing context. To prevent race conditions caused by multiple threads adding and reading elements in the map, a thread must acquire the Graph Lock before accessing Work Graph.

4.1.4 Work Queue

When an MC-message has been assigned a context, it is placed in the *Work Queue*. The purpose of the Work Queue is to keep the order of the received MC-messages and act as a recovery point if a system crash occurs. This implies that

the MC-messages are not removed until they have actually been executed by a context. As the contexts execute in different rate, a later received message could be finished before an earlier received message. To keep the actual received order, an MC-message that was received later can never be removed from the Work Queue before the earlier received messages. To achieve this, an iterable queue is used and each element in the queue has a completed flag. When a context is finished with its message, it will iterate from the start of the queue and remove all MC-messages that are marked completed. However, if an uncompleted MC-message is found before the context's own completed message, the removing will be stopped. The context will then continue with the iteration to find its MC-message and mark it as completed. To avoid race conditions when adding and removing from the Work Queue, accessing the Work Queue requires acquiring the Graph Lock.

4.1.5 Threads using the Component

Threads using the component are the contexts and the threads that handles the incoming MC-messages from the external markets. All the threads compete for the Graph Lock. The contexts also compete with each other for the lock of the Task Queues of each context, when moving MC-messages to another context's Task Queue due to dependency.

4.2 Designing Component from a Lock-free Perspective

Designing the partially lock-free solution was done in several steps. First, there was a need to identify how to replace the locks and what properties the data structures were needed to have. Then, an assessment of how useful already existing lock-free data structures from other libraries was made. This was followed by designing the needed lock-free data structures that could not be found in any library.

There exist lock-free data structures in open source libraries that could be used. Using existing libraries is convenient and work-saving. However, in some cases the existing libraries could lack some needed functionalities for this specific solution. The aim of this section is to motivate usage of existing libraries contra writing own data structures that were needed.

4.2.1 Lock-free Work Graph

The *work graph* in the component was designed using an unordered map that required locks to remain correct when threads access it. A replacement was necessary. A lock-free version of the `std::unordered_map` was found in Folly called *Atomic Unordered Map*. However, as described in section 2.3.3, Folly's map has some disadvantages. Although its limitations, it was regarded suitable for the component with certain workarounds. The limitations and workarounds will be further described in Section 4.3.4

4.2.2 Lock-free Work Queue

The work queue was implemented using the data structure `std::list`, which is implemented as a doubly-linked list. As a lock-free version of the `std::list` was not found in any existing libraries, an implementation of it had to be accomplished. To implement a similar list that additionally is lock-free is a complex problem. However, it was noticed that not all properties of `std::list` was needed in the existing component. The functions that were necessary and used in the component were:

- `push_back()`: Adds an element to the end of the list.
- `pop_front()`: Discards the first element in the list.
- `begin()`: Returns an iterator at the beginning of the list.
- `end()`: Returns an iterator to the end.
- `erase(pos)`: Erases an element, given an iterator positioned to the element.

Essentially what was needed is an atomic queue that was iterable. Additionally, it needed to be able to handle one thread calling `push_back()` and another thread calling `pop_front()` simultaneously (so called *single producer single consumer*), without race conditions. An atomic singly-linked list could be utilized to achieve the similar behaviour and functionality in a lock-free manner. However, existing atomic linked-lists that were suitable for this component was not encountered. Due to this reason, a self-implemented version had to be carried out. The self-implemented version was achieved by using the directions Herb Sutter described about a single producer single consumer list in the article presented in section 2.3.1.

4.2.3 Lock-free Suspended Contexts Queue

The *Suspended Context* queue in the component is needed to be able to handle multiple threads calling `push_back()` and `pop_front()` simultaneously, making it a so called *Multiple Producer Multiple Consumer queue*. The Multiple Consumer Multiple Consumer queue that was found in the *Boost Lockfree library* and described in section 2.3.2 could be used to replace the current lock-based structure.

4.2.4 Partially Lock-free Task Queue

Lastly, *Task Queue* was a queue that handles *Multiple producers single consumer* (i.e. having only one thread perform `pop_front()` and many calling `push_back()`). The queue needs to be iterable, as well as also being able to control when the first element is pushed into the queue. A lock-free data structure that matches these requirements was not found in any existing libraries. Due to this reason, similarly to the problem with work queue, a self-implemented version had to be carried out.

The lock-free Task Queue is based on the same design as the Work Queue, using shared pointers `divider`, `first` and `last`. However, since the queue has to

able to be handle multiple producers, the function `push_back()` needed to be modified. Unfortunately, a completely lock-free `push_back()` was not achieved. This was due to both `last` and `last`'s `next` must be updated atomically to remain correct. For this, a double compare-and-swap would be needed. Such an operation could not be found in any library; hence it was decided to use a simple spinlock instead. Therefore, making it only partially lock-free. It was regarded acceptable as this would be a lock with a small critical section. In the critical section, the new element will be stored into `next` of `last`. And then `last` will atomically be updated to its `next`. As in the lock-free Work Queue, the already popped elements will be removed by updating `first` to the atomic read value of `last`. The atomic operations are needed as the consumer can read `divider` and `last` as well simultaneously.

Additionally, the `pop_first()` function was divided into two separate functions; `peek()` and `pop()`. This was a result of how the list was going to be used in the worked component. The elements of the list would be functions, and a consumer thread would enter a loop if the list is not empty to pop these functions and execute them. This is depicted in Figure 4.2.

```
1   function<void()> lFunction;
2   while (m_TaskQueue_lockfree.peek(lFunction))
3   {
4       lFunction();
5       m_TaskQueue_lockfree.pop();
6   }
```

Figure 4.2. How `peek()` and `pop()` were used in the worked component. Figure created by the authors.

Although there is only one consumer at the time, there could be a sort of race condition if using an `empty()` as the condition for the while and then `pop_first()`. This was due to during the function being executed in the loop, another thread would see the queue as non-empty, and also entering the loop and execute the same function. Hence, `peek()` was designed to check if the list is empty. If not, it returns true and writes the first element to the given parameter. Else, it will return false. The function `pop()` only removes the first element of the list. This would allow a thread to get the function, and not call `pop()` until the function is completed. In this way, no other thread can enter the loop as the function is still in the list.

Another non-trivial function needed was `push_back_is_first()` for the producers, which would return true if the element was pushed into an empty list. This was done by using a boolean variable called `isFirst` inside the critical section. To check if the pushed element was first, `last` and `next` of `divider` was compared in the critical section.

4.3 Implementation

This section describes how the self-written data structures were implemented in C++11. Furthermore, it will present how the lock-free data structures were integrated into the working component, so it would still have the same functionality.

4.3.1 Lock-free Work Queue

The lock-free Work Queue followed the design in Sutter's article that was presented in 2.3.1 [60], but was implemented in a different way. `first`, `divider` and `last` were implemented as *shared pointers*. A shared pointer in C++11 (`std::shared_ptr`) is essentially a smart pointer, where an object could be owned by several shared pointers. The object is deleted from the memory when no shared pointers are owning that object anymore [75]. This simplified the process of removing popped elements from the list. When `first` is set to `divider`, the popped elements will be deleted as no shared pointers will point at them anymore. This is depicted in Figure 4.3.

```
#pragma once

#include "stdafx.h"
#include <atomic>
using namespace std;

template<typename T, shared_ptr<T>(T::*PtrToNext)>  // *PtrToNext is m_Next in CMCWorkItem
class SPSC_atomic_list
{
    shared_ptr<T> first;
    shared_ptr<T> divider, last;
public:

    SPSC_atomic_list() {
        first = divider = last = make_shared<T>();
    }

    void push_back(shared_ptr<T> t) {
        atomic_load(&last).get()->*PtrToNext = t;       // Set last's next to new
        atomic_store(&last, t);                         // Publish it
        Size++;
        first = atomic_load(&divider);                  // Remove already popped elements
    }

    bool pop_front() {
        if (divider != atomic_load(&last)) {                    // If queue is not empty
            atomic_store(&divider, divider.get()->*PtrToNext);  // Publish that it has been popped
            return true;                                        // Success
        }
        return false;           // List is empty
    }

    bool empty() {  // Only called by same thread as pop_front()
        return divider == atomic_load(&last);
    }

    shared_ptr<T> begin() {
        return divider.get()->*PtrToNext;
    }
};
```

Figure 4.3. The implementation of Atomic SPSC list. Figure created by the authors.

A crucial part of the design was to ensure atomicity when modifying and reading `divider` and `last`. For this, `atomic_load()` and `atomic_store()` were

used from the standard library. Furthermore, the functions `empty()` and `begin()` were implemented as the list needed to be iterable.

As this list was created specifically for a data structure created by the original developer, some adaptions could be made. A list is usually implemented by having a private node class that would have a value field and a next pointer. When an element is pushed, the list will allocate a new node and put the element as value. However, in this case, there was no such class. Instead, a shared pointer called `m_Next` was added to the data structure itself. This allowed the list to use less memory, as the objects would form the list through the `m_Next` field, rather than having to allocate new node objects internally by the list. The `m_Next` is accessed by declaring that the type used in the list will have a member that is a shared pointer, which was named `*PtrToNext`.

4.3.2 Partially Lock-free Task Queue

Unlike the lock-free Work Queue, a node class was needed when implementing the lock-free Task Queue. This was due to the elements in the list was going to be functions, therefore was it not possible to have a next pointer implemented in the data structure like in the lock-free Work Queue. The node class would be implemented as a data structure with a generic datatype called `value` and a shared pointer called `next`. Otherwise, the implementation was similar to the lock-free Work Queue, as depicted in Figure 4.4.

```cpp
template <typename T>
class MPSC_atomic_list {    // Not COMPLETELY lock-free. One spinlock for producers.

private:
    struct Node {
        T value;
        shared_ptr<Node> next;


    };

    shared_ptr<Node> first;
    shared_ptr<Node> divider, last;
    atomic<int> Size = 0;
    atomic<bool> producerLock;

public:
    MPSC_atomic_list() {
        first = divider = last = make_shared<Node>();
    }
    ~MPSC_atomic_list() = default;


    bool push_back_is_first(T t) {
        auto p = make_shared<Node>();
        p->value = t;
        p->next = NULL;
        bool isFirst;
        while (producerLock.exchange(true)) {}  // Acquire exclusvity
        last->next = p;     // Set last's next to new
        atomic_store(&last, last->next);    // Publish it
        first = atomic_load(&divider);  // Remove already popped elements
        isFirst = atomic_load(&divider).get()->next == last;
        producerLock = false;       // Release exclusivity

        Size++; // size can be updated atomically
        return isFirst;

    }

    void push_back(T t) {
        auto p = make_shared<Node>();
        p->value = t;
        p->next = NULL;
        while (producerLock.exchange(true)) {}  // Acquire exclusvity
        last->next = p;     // Set last's next to new
        atomic_store(&last, last->next);    // Publish it
        first = atomic_load(&divider);  // Remove already popped elements
        producerLock = false;       // Release exclusivity

        Size++; // size can be updated atomically
    }

    bool peek(T& result) {  // Gets the value of the first element, but does not remove it
        if (divider != atomic_load(&last)) {                    // if queue is nonempty
            auto current = atomic_load(&divider.get()->next);
            result = current->value;
            return true;                            // success
        }
        return false;           //List is empty
    }

    void pop() {        // Removes the first element.
        if (divider != atomic_load(&last))
        {
            atomic_store(&divider, current);
            Size--;
        }
    }
```

Figure 4.4. The implementation of the Concurrent MPSC list. Figure created by the authors.

Shared pointers were used for `first`, `divider` and `last`. The `peek()` function has an address as a parameter, which would be used to write the first element of the list to. However, this was done without writing `divider` to

divider's next. This was instead done in pop(), which was identical to pop_front() in the lock-free Work Queue.

Due to the lack of a double compare-and-exchange operation, a simple spinlock was implemented using an atomic boolean variable named producerLock. A producer would get access to the critical section by repeatedly calling the function std::exchange() on producerLock. The function std::exchange() returns the old value of the variable, so if std::exchange() returns false, then the lock was free and would be written to true by the current thread. This thread can then enter the critical section. The new element would then be written to last's next and last would be updated to it. Already popped elements would be deleted by writing first to divider, as in the lock-free Work Queue.

4.3.3 Integrating the Lock-free Work Queue

Integrating the self-implemented lock-free Work Queue into the component was a trivial task. As the behaviour of the std::list and the functions used had been replicated in a lock-free manner, replacing the data structure used was all there was needed. So instead of calling std::list the lock-free Work Queue was called. The only difference is that the iteration of the list was carried out in a different manner. To iterate the lock-free Work Queue, begin() is called, which returns the first element in the list, and from there next is called until there are no more elements. In the std::list, an iterator is used to go through the list. However, since an iterator was not necessary to iterate the Work Queue, it was not implemented.

4.3.4 Integrating the Lock-free Work Graph

As the lock-free version of the Work Graph had the limitation of not being resizable, and also not being able to erase entries, work-arounds had to be performed. To solve the problem of the map not being resizable, the size was set to a number it was known that the program will not exceed, 50 000.

As erasing entries is necessary in the current program, it was needed for the lock-free Work Graph to be able to do so as well. To come around this problem, a counter was added to the data structure that is stored as value in the map. The counter is used to keep track of the number of MC-messages each context will perform. When using emplace(), the counter is incremented and where erase() is called, the counter is decremented. When the counter is 0, it is known that the value of the entry will not be used anymore and should be erased. Since the value stored in each entry is a data structure that has a shared_ptr and a counter, the function std::shared_ptr::reset() is called on the shared_ptr. This will erase the context pointed by the shared_ptr and free up some memory. The disadvantage of only doing so is that the function will only empty the pointer of the value in the map and not erase the entry. This implies that using this map will have the consequence of

using large amount of memory over time. However, it was still regarded as an acceptable solution as the keys could be reused and there were no memory limitations.

4.3.5 Integrating the Lock-free Suspended Contexts Queue

The Boost MPMC queue was intended to be used as the Suspended Contexts queue. For this, an atomic boolean variable named `Postponed` would be needed as a member in each context. This was to keep track if the context had been suspended and put into the queue yet. By using the `compare_exchange_strong()` on `Postponed`, it prevented the same context being placed into the suspended contexts queue by multiple external markets. However, a suspended context could have multiple dependent MC-messages that should be executed when it is popped and resumed from the suspended contexts queue, hence it would also have a new atomic integer member named `SuspendedCount`. This would act as a counter for how many MC-messages that were waiting to be executed by that context.

A limitation of the Boost MPMC queue was that it had to be fixed sized to be completely lock-free. To ensure lock-freedom, an appropriate size of the queue had to be chosen. The queue was going to be used for placing contexts that had been suspended in this case. There was a constant named `MAX_NUMBER_OF_MARKETS` that defined how many external markets could be maximum in use. The number of contexts cannot exceed the number of external markets and hence it was set to the size of the queue.

Apart from adding two new members to the context data structure and setting the queue to a fixed size, the queue only used primitive functions such as `pop()`, `push()` and `empty()`.

4.3.6 Integrating the Partially Lock-free Task Queue

The integration of the self-implemented lock-free Task Queue was trivial as it was only to replace the current one. This is because the corresponding functions had already been implemented. Apart from replacing `pop_first()` with the two separate functions `peek()` and `pop()`, as described in section 4.2.4, no more changes were added into the component to make the lock-free Task Queue work as intended.

5 Performance of the Partially Lock-free and Lock-based Component

This chapter presents the tests and results that were used to compare the performance of the partially lock-free and lock-based component.

5.1 Experimental Setup

All the testing was performed on a *Hewlett Packard Z420 Workstation* [76] with the following specifications:

- Intel XEON CPU E5-1650 v2, 6 cores and 12 virtual threads [77]
- 32GB RAM
- Hewlett Packard Motherboard 1589
- MICRON 2,5" SATA SSD, 256GB, ATA MTFDDAK256MAY-1A SCSI disk [78]
- Windows 10 Enterprise Edition, 64-bit, version 1607, OS Build 14393.2248 [79, p. 10]

Both the partially lock-free and lock-based programs were compiled and built on *Microsoft Visual Studio Enterprise 2017, version 15.7.3* [80] with the following installed:

- Windows SDK version 8.1 [81]
- Microsoft .NET Framework, version 4.7.02053 [82]
- Microsoft Architecture Diagrams and Analysis Tools
- Microsoft Visual C++ 2017 [83]
- Microsoft Visual C++ Wizards 1.0
- Microsoft Visual Studio VC Package 1.0
- NuGet Package Manager 4.6.0 [84]
- Application Insights Tools for Visual Studio Package 8.12.10405.1 [85]
- ASP.NET and Web Tools 2017 15.0.40522.0 [86]
- Visual Basic Tools 2.8.3-beta6-62923-07 [87]
- Visual Studio Code Debug Adapter Host Package 1.0
- Visual Studio Tools for CMake 1.0
- Visual Studio Tools for Universal Windows Apps 15.0.27703.2026

The tests were conducted using FIS' provided logging tool named `flowtrackpoint`. By placing these out in the code, the execution time between two `flowtrackpoints` could be gathered for each handled MC-message. Each test was performed using 1,2,4 or 8 external markets. With a greater number of external markets, more MC-messages and more contention is generated. However, the `flowtrackpoints` could only track the normal case of execution for an MC-message. If an MC-message is placed in the Suspended Contexts queue or moved around in different Task Queues in contexts, the `flowtrackpoints` cannot track it. Hence, for these data structures, isolated tests were performed by writing test programs that would use the same operations as in the actual component. The execution time was

measured using `std::chrono::high_resolution_clock`. Each test was performed five times without rebooting the system in between. No background programs were running apart from the ones needed for the tests. These were *OrderSpray* 2.1.0.0 [88], a program to place stock orders on multiple external markets simultaneously, *SMART* 6.15.999 [89], a program for each external market that matches buy/sell orders with the internal market and *OMNI* 6.74.1.0 [90], the client program which showed if all placed orders were matched.

5.1.1 Tests on Work Graph and Work Queue

The `flowtrackpoints` measured the performance of Work Graph and Work Queue. The Work Graph's performance was measured when an MC-message would get assigned a context. This would involve an `emplace` operation. An emplace operation implies that the key-value pair is inserted if the key does not already exist in the Work Graph. Otherwise, it will return false. If the emplace operation returns false, then a `find()` operation will be made to find the dependent context that should execute the MC-message.

The other `flowtrackpoints` measured the performance of the next part of the component, which was when the MC-message was pushed into the Work Queue after getting assigned a context.

5.1.2 Test Programs for Isolated Tests

To test the data structures that could not be tracked using `flowtrackpoints`, test programs were made. The data structures that needed isolated tests were the Suspended Contexts queue and the Task Queue. The Suspended Contexts queue was compared to an `std::queue` used with an `std::mutex` as lock. Three different tests were made:

- push 50 elements using 1, 2, 4 or 8 threads.
- pop 50 elements using 1, 2, 4 or 8 threads
- multiple producers multiple consumers: 1,2 or 4 threads pushes a total of 50 elements while 1, 2 or 4 other threads pops a total of 50 elements simultaneously. At the end of the test, no elements will be left in the queue.

50 elements were chosen for the test programs as that was the maximum size of the Suspended Contexts in the original component as well.

The Task Queue was compared to an `std::list` with an `std::mutex` as a lock. A list was used because an important property of the Task Queue was it being iterable. Two test programs were created. One was for testing the `push_back_is_first()` function. This was done by calling the function one million times using 1, 2, 4 or 8 threads. To check if the pushed element was first in the lock-based test program, the function `empty()` would be called. The second test program was to simulate when MC-messages could be added or

moved between Task Queues by multiple contexts, while simultaneously popping. This was achieved by having 1, 2, 4 or 8 threads that would push elements while one thread would pop elements, hence called *multiple producer single consumer*. 1 million elements in total were used. Both the lock-free and lock-based test programs used `int` instead of MC-messages for simplicity.

5.2 Execution Time in the Component

The performance of two data structures were measured using `flowtrackpoints`. The first one was the execution time for an MC-message to get assigned a context and then to be put in the Work Graph. The second measured data structure was Work Queue. The execution time of pushing an MC-message to the Work Queue was measured. The results showed from the first one was that the lock-free Work Graph, i.e. the Folly Unordered Map, performed in average better than the lock-based one in all cases except when having eight external markets. This is illustrated in Figure 5.1.

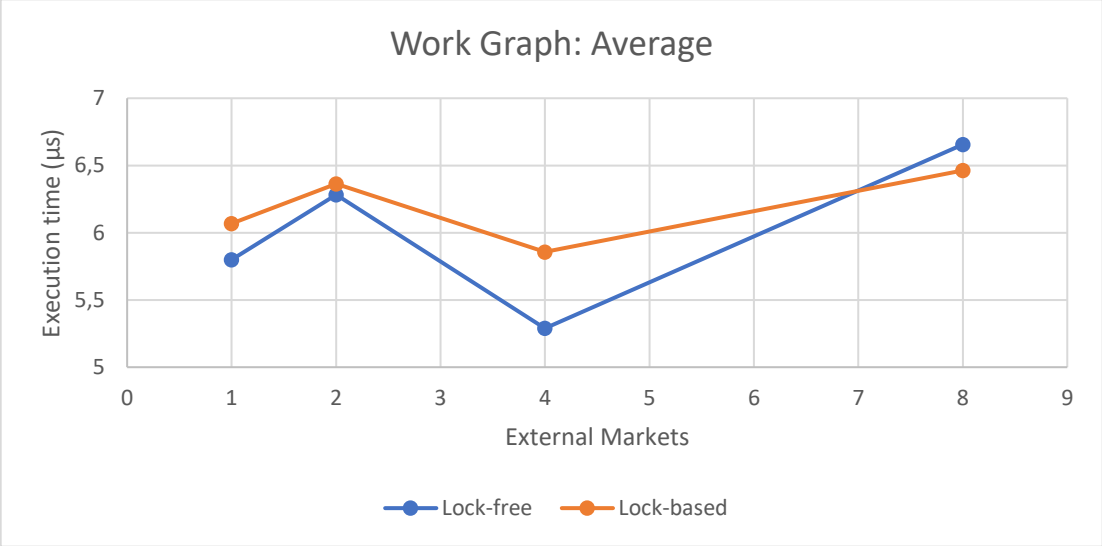


Figure 5.1. Average execution time of the Work Graph. The X-axis represents number of external markets used and the Y-axis represents the average execution time in µs. Figure created by the authors.

When having one external market, the execution time for the lock-free was in average 5,8 µs compared to 6,068 µs in the lock-based implementation. Using two external markets, the lock-free took 6,284 µs while the lock-based took 6,364 µs. The biggest difference was when having four external markets. The lock-free took 5,289 µs compared to 5,587 µs of the lock-based. However, using eight external markets, the lock-based performed better with 6,464 µs compared to the lock-free which took 6,656 µs.

The results of the second measurement, as seen in Figure 5.2, is that the lock-free version of Work Queue performed in average much worse than the lock-based one.

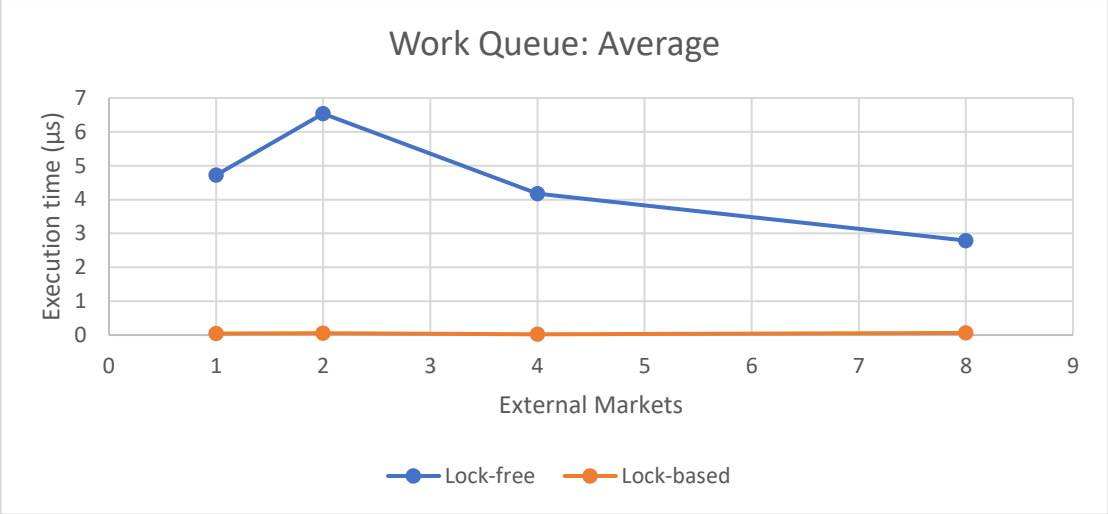


Figure 5.2. Average execution time of pushing an MC-message to the Work Queue. Figure created by the authors.

In the lock-based one, it took in average less than 1 µs in all cases. In the lock-free implementation, it took in average at least more than 2 µs in all cases. However, this was due to the inconsistency in the lock-free version. The median for both implementations were less than 1 µs. 5,9% of the MC-messages took more than 1 µs to push to the Work Queue in the lock-free implementation compared to only 0,8% in the lock-based. The inconsistency in execution time can also be seen in Figure 5.3.

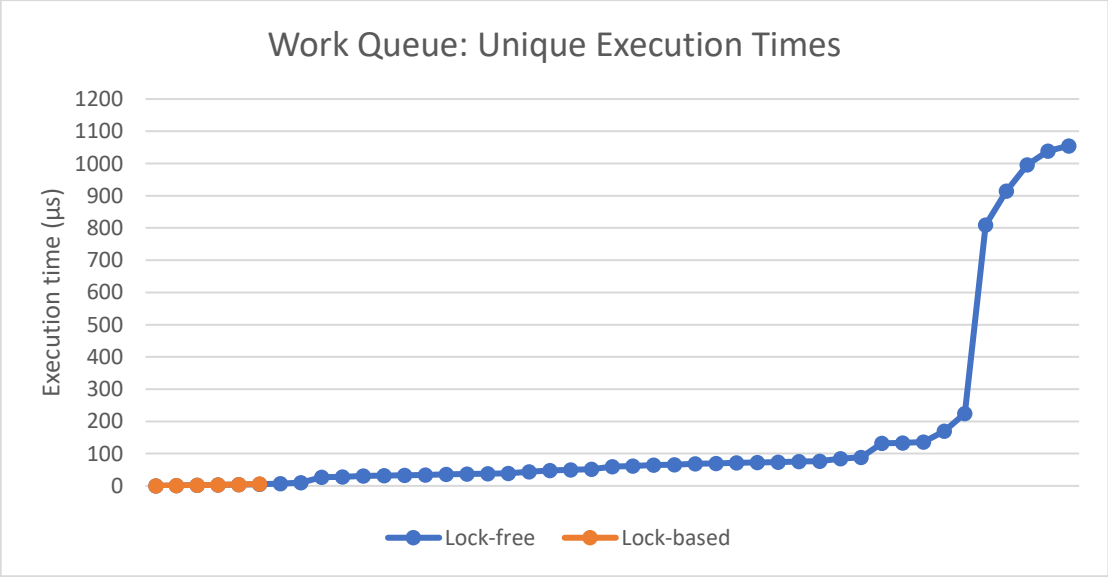


Figure 5.3. The unique values in execution time when pushing an MC-message to the Work Queue. Each dot represents a unique value. Figure created by the authors.

The execution time in the lock-based implementation would vary between less than 1 µs to 6 µs. This interval was much bigger in the lock-free implementation, varying between less than 1 µs to 1055 µs.

This implied that the total execution time in this part of the component (where `flowtrackpoints` could track the same MC-message) was longer in average for the lock-free than the lock-based implementation. This is shown in Figure 5.4.

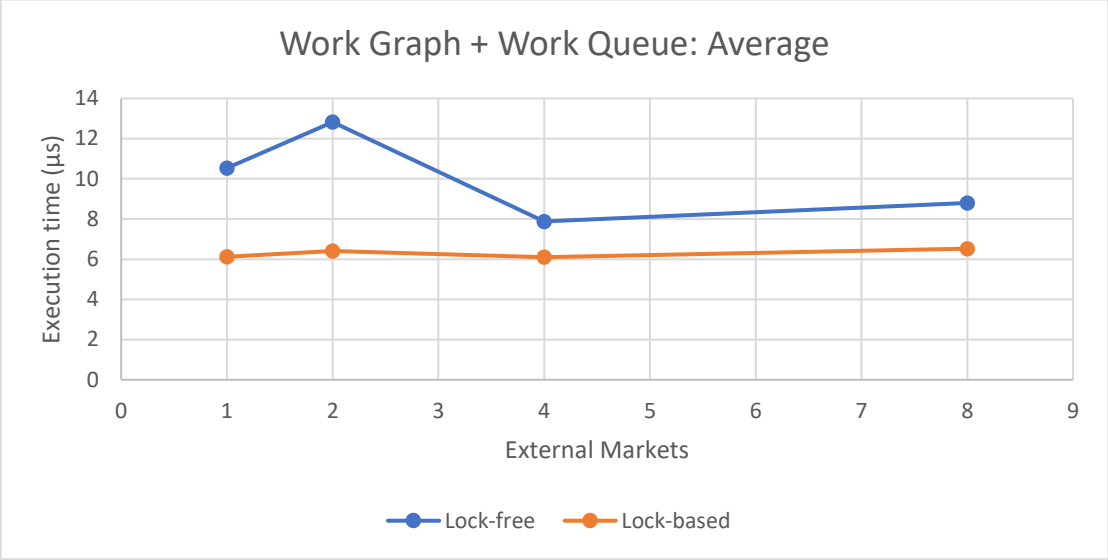


Figure 5.4. Average execution time for an MC-message in the part of component where `flowtrackpoints` could track the message. Figure created by the authors.

Using one external market, the execution time was 10,525 µs in the lock-free compared to 6,117 µs in the lock-based. The difference was largest using two external markets, the lock-free took 12,821 µs and the lock-based took 6,414 µs. Using four external markets took 7,878 µs and 6,099 µs respectively. Lastly, with 8 external markets, the lock-free implementation took 8,8 µs and the lock-based took 6,525 µs.

However, the lock-free implementation had in median shorter execution time than the lock-based one, as illustrated in Figure 5.5.

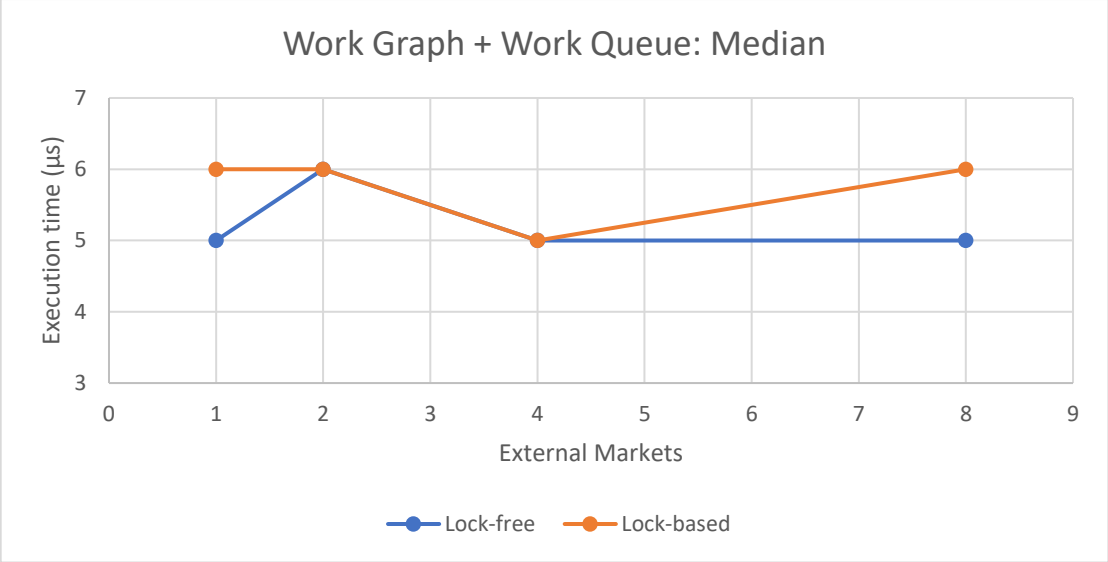


Figure 5.5. Median execution time for an MC-message in the part of the component where `flowtrackpoints` could track the message. Figure created by the authors.

When having two and four external markets, both the lock-free and lock-based implementation had a median execution time of 6 and 5 µs. The lock-free implementation had a shorter execution time in median when using one and eight external markets. In both cases, the lock-free implementation had a median of 5 µs compared to 6 µs in the lock-based.

5.3 Execution Time in Isolated Tests

The lock-free Suspended Contexts queue performed better than the lock-based version when pushing 50 elements, as seen in Figure 5.6.

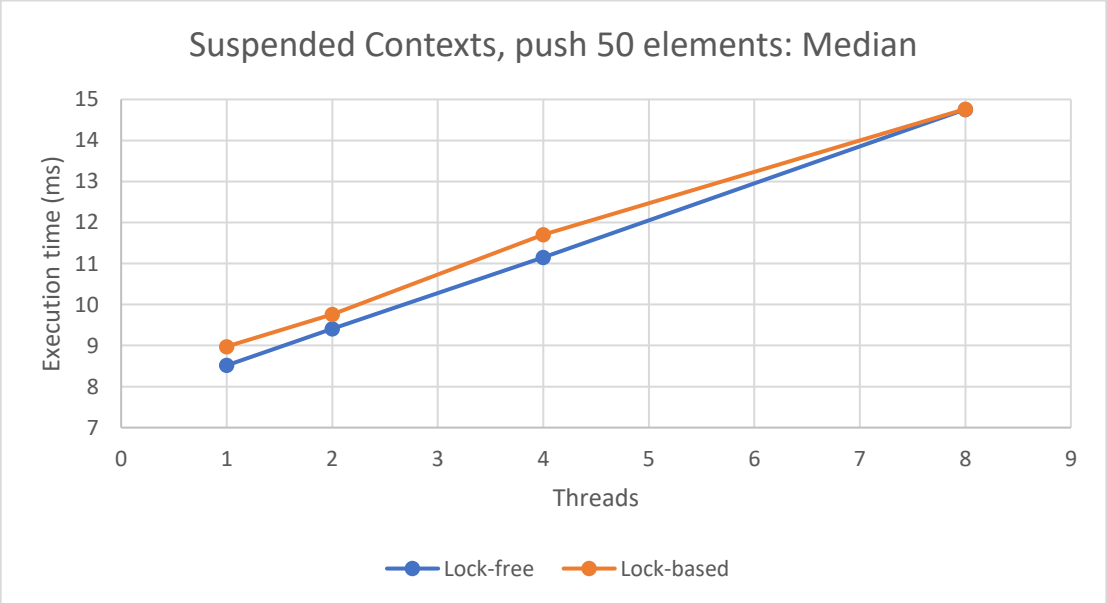


Figure 5.6. Median execution time of pushing 50 elements into the Suspended Contexts queue. The X-axis represents number of threads pushing elements and

the Y-axis represents execution time in milliseconds. Figure created by the authors.

When having only one pushing thread, the lock-free Suspended Contexts queue took in median 8,517 ms to push 50 elements. The lock-based one took 8,976 ms. Using 2 threads, the lock-free implementation took 9,406 ms compared to 9,764 ms in the lock-based. It took 11,148 ms and 11,701 ms respectively when having 4 threads. The lock-free implementation was also faster when having 8 threads, 14,757 ms compared to 14,765 ms.

However, the lock-based Suspended Contexts performed better in the test program for `pop()`. As seen in Figure 5.7, the lock-based was faster in every case except when having 4 threads.

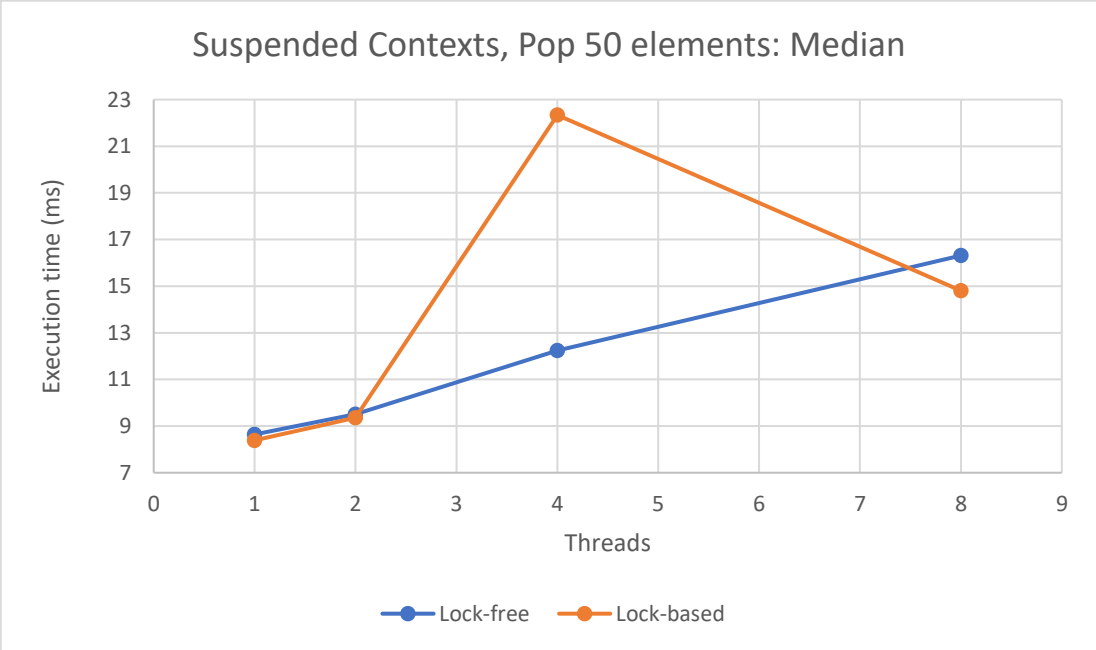


Figure 5.7. Median execution time of popping 50 elements from the Suspended Contexts queue. Figure created by the authors.

The execution time of the lock-free implementation was 8,642 ms compared to 8,389 ms of the lock-based when having one popping thread. Furthermore, when using 2 threads, the lock-free Suspended Contexts queue took 9,502 ms compared to 9,354 ms of the lock-based. When using 4 threads, the lock-free was faster than the lock-based one, 12,241 ms compared to 22,336 ms. However, when using 8 threads, the lock-free was slower again. It took 16,312 ms and 14,807 ms respectively.

In the multiple producer and multiple consumer test, the lock-free performed better. This can be seen in Figure 5.8.

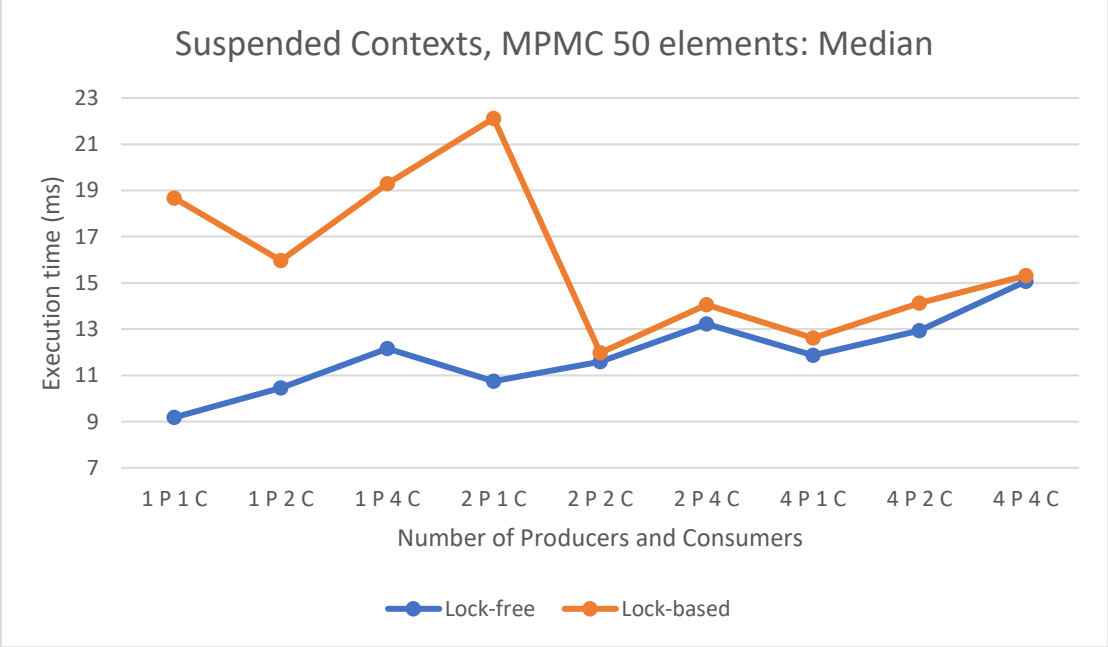


Figure 5.8. Multiple Producer Multiple Consumer test. The X-axis represents how many producer (P) and consumer (C) threads were used. The Y-axis represents the execution time in milliseconds. Figure created by the authors.

When having 1 producer and 1 consumer, the lock-free implementation took 9,185 ms. The lock-based took 15,964 ms. Using 1 producer and 2 consumers, it took 10,452 ms and 15,964 ms respectively. The lock-free implementation executed in 12,156 ms when having 1 producer and 4 consumers. The lock-based took 19,287 ms. The lock-free was also faster when having 2 producers. For 2 producers and 1 consumer, it took 10,754 ms and 22,121 ms respectively. Having 2 producers and 2 consumers took the lock-free implementation 11,59 ms compared to 11,975 ms of the lock-based. Using 2 producers and 4 consumers took 13,224 ms for the lock-free while the lock-based took 14,064 ms. Furthermore, the lock-free was faster having 4 producers as well. When having 4 producers and 1 consumer, the lock-free took 11,866 ms compared to 12,615 ms of the lock-based. For 4 producers and 2 consumers, the lock-free took 12,93 ms and the lock-based took 14,133 ms. Lastly, having 4 producers and 4 consumers, the lock-free recorded 15,07 ms compared to the 15,321 ms recorded by the lock-based implementation.

The results of the test programs for the Task Queue showed that the partially lock-free implementation was slower when doing `push_back_is_first()`. This is shown in Figure 5.9.

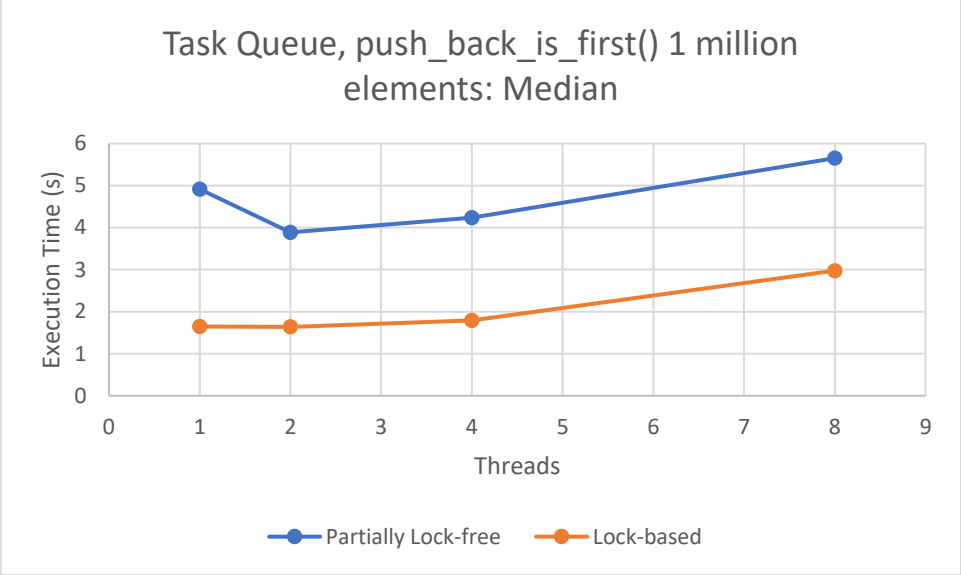


Figure 5.9. Execution time in median when doing `push_back_is_first()` a million times. The X-axis is the execution time in seconds and the Y-axis is the number of threads used. Figure created by the authors.

When having 1 pushing thread, it took the partially lock-free implementation 4,916 seconds to push 1 million elements. The lock-based took 1,644 seconds. Using 2 pushing threads, the partially lock-free recorded at 3,887 seconds compared to 1,637 seconds in the lock-based implementation. Having 4 threads took 4,238 seconds and 1,793 seconds respectively. Lastly, the partially lock-free was slower when using 8 threads. 5,652 seconds compared to 2,976 seconds of the lock-based.

However, the partially lock-free Task Queue performed better in the multiple producer single consumer test, as seen in Figure 5.10.

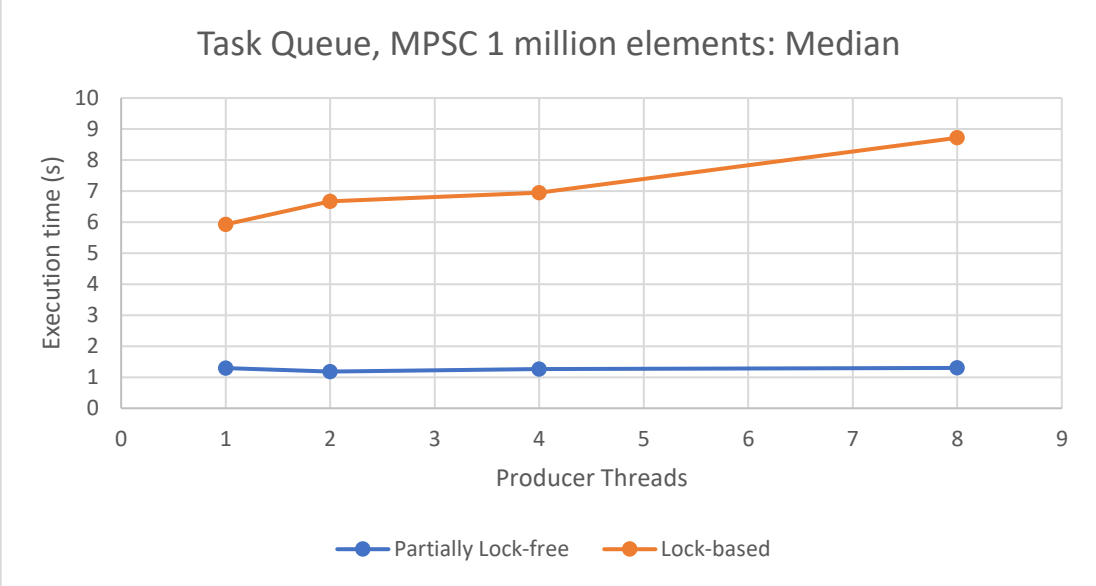


Figure 5.10. Execution time of the multiple producer single consumer test. The X-axis represents the number of producer threads used. The Y-axis is the execution time in seconds. Figure created by the authors.

When using 1 producer, the partially lock-free implementation took 1,294 seconds compared to 5,927 seconds of the lock-based. Furthermore, using 2 producers also resulted in faster execution time for the partially lock-free implementation. 1,183 seconds compared to 6,667 seconds. Using 4 producers took 1,259 seconds and 6,947 seconds respectively. Lastly, when using 8 producers, the partially lock-free implementation recorded an execution time of 1,3 seconds while the lock-based recorded 8,717 seconds.

6 Discussion

This chapter discusses the experiences and results of the project. Section 6.1 assesses the method and methodologies that were used. Section 6.2 analyses the results to answer whether the goal and research question was achieved and answered in Section 6.3. This is followed by Section 6.4 which discusses the validity and reliability of the results.

6.1 Method and Methodologies

This section covers the method applied to answer and evaluate the problem research questions. The approach used in this thesis was divided into four phases: *Data collection, Design and Implementation, Tests and Evaluation* and *Summarization*.

6.1.1 Data Collection

As there was lack of knowledge and experience in lock-free programming and trading systems in general, performing a data collection was necessary. The approach taken in this phase was to first conduct a literature study to obtain enough knowledge in lock-free programming and concurrent programming in general. Secondly, to achieve an understanding of the component worked with and some guidance on lock-free programming, an interview with the developers at the company was performed. These steps were crucial for this thesis in order to make right design choices for the lock-free version of the component.

6.1.2 Design and Implementation

When enough knowledge in both the component and in lock-free programming was achieved, the designing and implementation of potential solutions was performed. In order to obtain a lock-free version and answer the research question, the main focus went to the data structures instead. Through analyzing the component and its algorithms, it was noticed that the locks used could be removed through obtaining atomic and lock-free data structures. Therefore, only changes of data structures used was carried out. The approach of first searching for suitable existing libraries, before implementing an own data structure was shown as a good decision as the both existing libraries used had good performance.

6.1.3 Tests and Evaluation

In order to evaluate and investigate the research question whether a lock-free implementation is an improvement compared to one using locks, summative evaluation was used. Since the focus of the thesis was on how the speed of the lock-free implementation would compare to the lock-based version, solely using summative evaluation was considered enough. Using the data retrieved from the benchmarks, it was possible to draw a conclusion.

6.2 Analysis of the Results

The results from the tests showed that the lock-free Work Graph performed better than the lock-based Work Graph in every case except when having 8 external markets. This could be explained by the design of the lock-free Work Graph, where elements that are added will never be deleted. When having 8 external markets, more MC-messages will be generated. This implies that when an MC-message is dependent on another message, finding the same context in the Work Graph will take longer. These results indicate that the lock-free Work Graph is perhaps more suited for internal markets that are not connected to many external markets.

From the results of the lock-free and lock-based Work Queue, it could be seen that the lock-free version had a better execution time in median, but worse in average. This was caused by the inconsistency in execution time, where some MC-messages could take up to 1055 µs, compared to the median value that was less than 1 µs. Due to this only happened on a small number of messages and without any pattern, more testing would be needed to get an understanding of the inconsistent execution time.

The lock-free and lock-based Suspended Contexts queue performed quite similar when pushing. The results of the `pop()` test were also close, except when having 4 threads, where the lock-based version performed much worse. However, due to the results being close in all the other cases, the performance of popping was also concluded as similar between the lock-free and lock-based implementation. An explanation of the result from 4 threads could be bad timing, where the threads would try to acquire the lock at the same time. Furthermore, the lock-free version performed better in all cases in the multiple producers multiple consumers test. The reason for this is that in the lock-based version, the producers and the consumers have to compete for the same lock and therefore creates high lock contention. In the lock-free version (from the Boost library), it uses atomic operations on separate pointers that represent the head and the tail of the queue instead. This implies that both the producers and consumers can work simultaneously.

The partially lock-free implementation of the Task Queue performed worse than the lock-based in the `push_back_is_first()` test. The reason for this is that in both implementations, a lock must be acquired first. However, in the partially lock-free implementation, it has to additionally perform two atomic operations; `atomic_store()` for pushing and `atomic_load()` for checking if it is first. The atomic operations are more expensive compared to the corresponding normal operations that are used in the lock-based implementation. It may appear as a bad design to use both a lock and atomic operations in the partially lock-free Task Queue. However, this was a tradeoff that was intentionally made. The atomic operations allow a consuming thread to work on the queue simultaneously, as there will be no visible half-states when pushing. This led to the positive results shown in the multiple producers single consumer test. In the lock-based version, the producers and consumer conquer for the same lock and thus create lock contention.

The results from tests that was performed directly on the component showed that the partially lock-free implementation had a similar performance compared to the lock-based implementation. This was due to the lock-free Work Graph performed better than the lock-based version, but the lock-free Work Queue did not perform as well as the lock-based one. However, there are signs of that the partially lock-free component has better real-life performance as the isolated tests showed that the lock-free versions of the data structures performed better in most cases.

It is important to remember that this component was designed as a lock-based system, and this project only replaced all lock-based data structures with lock-free ones. A more interesting comparison would be having a component that was designed from scratch using lock-free algorithms and data structures. However, if improving performance of the provided component was the only objective, then a combination of both lock-free and lock-based data structures would be more suitable. An example would be keeping the lock-based Work Queue and use the lock-free Work Graph.

6.2.1 Performance in Median and Average

An interesting aspect of the results was the difference when using median and average. The partially lock-free component performed better in median but worse in average compared to the lock-based component. This was due to its inconsistency, with some specific executions being up to 1000 times slower. This led to a worse execution time in average. More investigation would be needed to find an explanation of this. Unfortunately, there were no possibilities to further investigate this in this project.

6.3 Revisiting the Goal and Research Question

This section discusses whether the goal has been reached and if the research question has been answered.

6.3.1 Goal

The goal of this thesis was to deliver a lock-free implementation of the component provided by FIS and document its performance compared to the lock-based implementation.

The outcome of this project was a lock-free version of the component, except for a spinlock in one of the data structures. A comparison in performance of both versions was also delivered. Although the component was not completely lock-free, the goal of this project was still regarded as achieved. The design of the new version was still based on lock-freedom and using a spinlock in one of the data structures was considered acceptable when locks on a higher level with larger critical section could be eliminated. More importantly, the results showed positive signs from the usage of lock-free programming. The experience

of this project also showed that it is not always possible to make a program lock-free or it may be too complex and time consuming to do so. These are useful experiences for the company and hence is the goal still regarded as achieved.

6.3.2 Research Question

The research question was following:
"How much improvement in performance could a lock-free implementation of a component in a trading system achieve, compared to one using locks?"

As seen in Chapter 5, performance improvements could be seen in some parts of the component, as well as decreases in other parts. However, performance of the whole component could not be measured due to the limitations of the provided measurement tool. The isolated tests showed signs of improvement in total performance, but because a comparison of the entire component could not be done, the research question cannot be answered properly.

6.4 Validity and Reliability of Results

Validity refers to the credibility of the research [91], i.e. if the results are genuine to be used to determine whether the goal was reached and answer the research question. To be regarded as valid in this case, the partially lock-free component must be correct. The partially lock-free component was correct as it had the same functionality as the lock-based component, which was handling incoming MC-messages by scheduling, executing and logging. This was seen in the client program OMNI, which was a part of the experimental setup for the tests. If the component work, all the placed orders in the internal and external markets will be matched.

Reliability is about how well the problem is solved. A research is reliable if the tests could be done repeatedly and the result remains the same [91]. The results from the tests performed directly on the component should be regarded reliable as it is tested through FIS' own logging tool. The isolated tests could also be regarded as reliable due to its simplicity. A variable in the tests are the timing of when threads are started and when they try to acquire the lock. However, by doing each test five times and using the median value, the results should not be affected.

7 Conclusions

This chapter conclude the thesis with a summary of the whole project and discusses future work that could be done that is related to the thesis.

7.1 Summary

The goal of this thesis was to deliver a lock-free implementation of a component in a trading system provided by FIS and compare its performance with the existing lock-based implementation. This was achieved by replacing the data structures in the component that was protected by locks with lock-free ones. Some of these lock-free data structures were from existing libraries, such as the lock-free unordered map named Work Graph and the multiple producer multiple consumer queue called Suspended Contexts. Other data structures needed specific design and support for specific operations. These were therefore self-implemented. A single producer single consumer queue called Work Queue and a multiple producer single consumer queue named Task Queue were self-implemented. However, the Task Queue could not be implemented without a spinlock. This made the component not completely lock-free and therefore called partially lock-free.

Two types of tests were conducted to measure performance of the lock-free and lock-based implementations. Direct tests were tests that was performed directly on the component. However, the direct tests could not be applied when the program is not executed in the most usual branch and thus cannot measure the performance of some of the data structures. Hence, for these data structures, isolated tests were performed. These consisted of simple test programs that would test the operations that was used in the actual component. The results of the direct tests showed that the lock-free Work Graph performed in general better than the lock-based version. However, the lock-free Work Queue performed worse than the lock-based implementation due to inconsistency in execution time. In the isolated tests, it could be seen that the lock-free data structures performed in general better than the lock-based ones. This was mostly caused by the queues could be worked by multiple threads simultaneously from both the head and the tail.

The results could not properly answer the research question of how much performance increase the partially lock-free component had, as the tests did not cover the whole component's real-life performance. However, the tests give signs of that lock-free programming gave the component a total performance increase.

7.2 Future Work

This thesis successfully provided a partially lock-free implementation of a lock-based component in a trading system, while also not decreasing the performance of the whole component. However, there are multiple other solutions and directions that could have been taken. Firstly, due to the

complexity and size of the whole trading system, the focus of this thesis was only on making part of the system lock-free. To get a proper way to compare a lock-free and a lock-based trading system, the whole system could be made lock-free using some of the directions taken in this thesis.

Secondly, as lock-freedom is achievable in multiple ways, other directions could be explored. The direction in this thesis was to use lock-free data structures. Other directions such as changing the algorithms or rewriting the component from scratch that could be investigated as well. Doing this, the design choices can be written in a way that suits lock-free algorithms and data structures.

Thirdly, the self-implemented Multiple Producer Single Consumer list could be rewritten to be fully lock-free instead of using a spinlock for updating the two variables. This can be achieved by implementing an effective double compare-and-swap.

References

[1] E. W. Dijkstra, “Solution of a Problem in Concurrent Programming Control,” *Commun. ACM*, vol. 8, no. 9, pp. 569–, Sep. 1965.

[2] V. Vlassov, “Concurrent programming(ID1217): Lecture 1 - Introduction, Parallel Programming Concepts, Models and Paradigms,” 10-Jan-2018. [Online]. Available: https://kth.instructure.com/files/689202/download?download_frd=1. [Accessed: 12-Apr-2018].

[3] B. Johnson, “Infrastructure requirements,” in *Algorithmic Trading & DMA: An introduction to direct access trading strategies*, London: 4Myeloma Press, 2010, p. 529.

[4] P. S. Pacheco, “Chapter 1 - Why Parallel Computing?,” in *An Introduction to Parallel Programming*, Boston: Morgan Kaufmann, 2011, pp. 1–14.

[5] D. Patterson and J. Hennessy, “Chapter 1: Fundamentals of Quantitative Design and Analysis,” in *Computer Architecture: A Quantative Approach*, 5th ed., San Fransisco: Morgan Kauffman, 2011, pp. 3–4.

[6] V. Vlassov, “Concurrent programming(ID1217): Lecture 2 - Shared Memory Programming: Processes and Synchronization,” 10-Feb-2018. [Online]. Available: https://kth.instructure.com/files/689253/download?download_frd=1. [Accessed: 25-Jul-2018].

[7] V. Vlassov, “Concurrent programming(ID1217): Lecture 15 - Distributed Programming with Message Passing,” 10-Feb-2018. [Online]. Available: https://kth.instructure.com/files/780063/download?download_frd=1. [Accessed: 25-Jul-2018].

[8] P. S. Pacheco, “Chapter 2 - Parallel Hardware and Parallel Software,” in *An Introduction to Parallel Programming*, Boston: Morgan Kaufmann, 2011, pp. 15–81.

[9] H. Sutter, “Lock-Free Programming (or, Juggling Razor Blades),” presented at the CppCon 2014, Washington, 2014.

[10] A. Alexandrescu, “Lock-Free Data Structures,” *Dr. Dobb’s*, 01-Oct-2004. [Online]. Available: http://www.drdobbs.com/lock-free-data-structures/184401865. [Accessed: 22-Apr-2018].

[11] J. Preshing, “Locks Aren’t Slow; Lock Contention Is,” 18-Nov-2011. [Online]. Available: http://preshing.com/20111118/locks-arent-slow-lock-contention-is/. [Accessed: 23-Apr-2018].

[12] Microsoft Windows Dev Center, “Lockless Programming Considerations for Xbox 360 and Microsoft Windows (Windows).” [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ee418650(v=vs.85).aspx. [Accessed: 23-Apr-2018].

[13] S. Al Bahra, “Nonblocking algorithms and scalable multicore programming,” *Communications of the ACM*, vol. 56, no. 7, p. 50, Jul. 2013.

[14] O. Goffart, “Introduction to lock free programming - or how to use QAtomic classes,” presented at the Qt Developers Days 2014 Europe, Berlin, 2014.

[15] D. Kalinsky, “Is lock-free programming practical for multicore?,” *Embedded Systems Design*, vol. 24, no. 3, 04-Apr-2011.
[16] J. Montelius, “Operating System(ID1206): Lecture 9 - Concurrency,” 10-Aug-2017. [Online]. Available: https://people.kth.se/~johanmon/courses/id2206/lectures/concurrency-handout.pdf. [Accessed: 22-Apr-2018].
[17] B. Goetz, “Going atomic,” 23-Nov-2004. [Online]. Available: http://www.ibm.com/developerworks/library/j-jtp11234/index.html. [Accessed: 22-Apr-2018].
[18] J. Preshing, “An Introduction to Lock-Free Programming,” 12-Jun-2012. [Online]. Available: http://preshing.com/20120612/an-introduction-to-lock-free-programming/. [Accessed: 23-Apr-2018].
[19] J. Preshing, “Atomic vs. Non-Atomic Operations.” [Online]. Available: http://preshing.com/20130618/atomic-vs-non-atomic-operations/. [Accessed: 03-May-2018].
[20] P. S. Pacheco, “Chapter 5 - Shared-Memory Programming with OpenMP,” in *An Introduction to Parallel Programming*, Boston: Morgan Kaufmann, 2011, pp. 209–270.
[21] “std::atomic - cppreference.com,” 15-Apr-2018. [Online]. Available: http://en.cppreference.com/w/cpp/atomic/atomic. [Accessed: 22-Apr-2018].
[22] “About Our Company - FIS.” [Online]. Available: http://www.fisglobal.com/about-us/about-our-company. [Accessed: 15-Apr-2018].
[23] Oracle, “Deadlock (The Java™ Tutorials > Essential Classes > Concurrency).” [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html. [Accessed: 20-Apr-2018].
[24] W. Stallings, “Chapter 6 Concurrency: Deadlock and Starvation,” *Operating Systems: Internals and Design Principles*. [Online]. Available: https://cs.nyu.edu/courses/fall12/CSCI-GA.2250-001/slides/Chapter06.pdf.
[25] Microsoft Support, “Description of race conditions and deadlocks,” 18-Jun-2012. [Online]. Available: https://support.microsoft.com/en-us/help/317723/description-of-race-conditions-and-deadlocks. [Accessed: 20-Apr-2018].
[26] G. Kroah-Hartman, A. Rubini, and J. Corbet, “Chapter 5: Concurrency and Race conditions,” in *Linux Device Drivers*, 3rd ed., 2005.
[27] C. Li, C. Ding, and K. Shen, “Quantifying the Cost of Context Switch,” in *Proceedings of the 2007 Workshop on Experimental Computer Science*, New York, NY, USA, 2007.
[28] L. Sawalha, M. P. Tull, and R. D. Barnes, “Hardware thread-context switching,” *Electronics Letters*, vol. 49, no. 6, pp. 389–391, Mar. 2013.
[29] sustainability@kth.se, “Hållbar utveckling | KTH,” 13-Dec-2017. [Online]. Available: https://www.kth.se/om/miljo-hallbar-utveckling/utbildning-miljo-hallbar-utveckling/verktygslada/sustainable-development/hallbar-utveckling-1.350579. [Accessed: 15-Jun-2018].

[30] sustainability@kth.se, “Ekologisk hållbarhet | KTH,” 24-Jun-2015. [Online]. Available: https://www.kth.se/om/miljo-hallbar-utveckling/utbildning-miljo-hallbar-utveckling/verktygslada/sustainable-development/ekologisk-hallbarhet-1.432074. [Accessed: 15-Jun-2018].
[31] L. Minas and B. Ellison, “The Problem of Power Consumption in Servers,” *Dr. Dobb’s*, 05-Mar-2009. [Online]. Available: http://www.drdobbs.com/the-problem-of-power-consumption-in-serv/215800830. [Accessed: 15-Jun-2018].
[32] sustainability@kth.se, “Social hållbarhet | KTH,” 05-Jul-2016. [Online]. Available: https://www.kth.se/om/miljo-hallbar-utveckling/utbildning-miljo-hallbar-utveckling/verktygslada/sustainable-development/social-hallbarhet-1.373774. [Accessed: 15-Jun-2018].
[33] sustainability@kth.se, “Ekonomisk hållbarhet | KTH,” 25-Jun-2015. [Online]. Available: https://www.kth.se/om/miljo-hallbar-utveckling/utbildning-miljo-hallbar-utveckling/verktygslada/sustainable-development/ekonomisk-hallbarhet-1.431976. [Accessed: 15-Jun-2018].
[34] M. D. Pierro and D. Skinner, “Concurrency in Modern Programming Languages,” *Computing in Science Engineering*, vol. 14, no. 6, pp. 8–10, Nov. 2012.
[35] J. Montelius, “Operating System(ID1206): Lecture 3- Processes,” 10-Aug-2017. [Online]. Available: https://people.kth.se/~johanmon/courses/id2206/lectures/processes-handout.pdf. [Accessed: 22-Apr-2018].
[36] J. Armstrong and S. D. Pfalzer, *Programming Erlang: software for a concurrent world*, Second edition. Dallas, Texas; Raleigh, North Carolina: The Pragmatic Bookshelf, 2013.
[37] V. Vlassov, “Concurrent programming(ID1217): Lecture 4 - Critical Sections, Locks, Condition Variables,” 10-Feb-2018. [Online]. Available: https://kth.instructure.com/files/733221/download?download_frd=1. [Accessed: 22-Apr-2018].
[38] H. Ding, X. Liao, H. Jin, X. Lv, and R. Guo, “Reducing lock contention on multi-core platforms,” 2014, pp. 158–165.
[39] D. Sehr, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, *LCPC’97*, vol. 9. Springer Science & Business Media, 1997.
[40] “std::thread - cppreference.com,” 17-Apr-2016. [Online]. Available: http://en.cppreference.com/w/cpp/thread/thread. [Accessed: 25-May-2018].
[41] The Department of Computer Science at the University of Chicago, “Concurrency in C++11,” 2013. [Online]. Available: https://www.classes.cs.uchicago.edu/archive/2013/spring/12300-1/labs/lab6/. [Accessed: 25-May-2018].
[42] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. ACM Press, 2006.
[43] R. Farber, *Parallel Programming with OpenACC*. Newnes, 2016.
[44] D. Cederman, B. Chatterjee, and P. Tsigas, “Understanding the Performance of Concurrent Data Structures on Graphics Processors,” in

Euro-Par 2012 Parallel Processing, vol. 7484, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 883–894.
[45] Intel Developer Zone, "Atomic Operations." [Online]. Available: https://software.intel.com/en-us/node/506090. [Accessed: 20-May-2018].
[46] "Atomic operations library - cppreference.com," 18-Nov-2017. [Online]. Available: http://en.cppreference.com/w/cpp/atomic. [Accessed: 03-May-2018].
[47] H. Sundell, *Efficient and practical non-blocking data structures*. Göteborg: Chalmers Univ. of Technology, 2004.
[48] J. Preshing, "You Can Do Any Kind of Atomic Read-Modify-Write Operation." [Online]. Available: http://preshing.com/20150402/you-can-do-any-kind-of-atomic-read-modify-write-operation/. [Accessed: 07-May-2018].
[49] "Creative Commons — Attribution-ShareAlike 3.0 Unported — CC BY-SA 3.0." [Online]. Available: https://creativecommons.org/licenses/by-sa/3.0/. [Accessed: 21-Aug-2018].
[50] "std::atomic::compare_exchange_weak, std::atomic::compare_exchange_strong - cppreference.com," 14-Feb-2018. [Online]. Available: http://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange. [Accessed: 07-May-2018].
[51] J. Delong, "Folly: The Facebook Open Source Library," 02-Jun-2012. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/folly-the-facebook-open-source-library/10150864656793920/. [Accessed: 23-May-2018].
[52] A. Simpkins, "folly: An open-source C++ library developed and used at Facebook," *Github*, 23-May-2018. [Online]. Available: https://github.com/facebook/folly. [Accessed: 23-May-2018].
[53] H. Fugal, "Futures for C++11 at Facebook," *Facebook Code*, 19-Jun-2015. [Online]. Available: https://code.facebook.com/posts/1661982097368498. [Accessed: 23-May-2018].
[54] Boost, "In House Boost." [Online]. Available: https://www.boost.org/users/uses_inhouse.html. [Accessed: 10-Jun-2018].
[55] "Boost C++ Libraries." [Online]. Available: https://www.boost.org/. [Accessed: 18-May-2018].
[56] The Boost C++ Libraries, "Introduction," *The Boost C++ Libraries*. [Online]. Available: https://theboostcpplibraries.com/introduction. [Accessed: 10-Jun-2018].
[57] "Chapter 21. Boost.Lockfree - 1.67.0." [Online]. Available: https://www.boost.org/doc/libs/1_67_0/doc/html/lockfree.html. [Accessed: 18-May-2018].
[58] D. Dechev, "The ABA problem in multicore data structures with collaborating operations," in *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, 2011, pp. 158–167.

[59] D. Dechev, P. Pirkelbauer, and B. Stroustrup, “Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs,” in *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2010, pp. 185–192.
[60] H. Sutter, “Writing Lock-Free Code: A Corrected Queue,” *Dr. Dobb’s*, 29-Sep-2008. [Online]. Available: http://www.drdobbs.com/parallel/writing-lock-free-code-a-corrected-queue/210604448. [Accessed: 13-Jun-2018].
[61] T. Blechmann, “Boost: Class template queue - 1.67.0.” [Online]. Available: https://www.boost.org/doc/libs/1_67_0/doc/html/boost/lockfree/queue.html#id-1_3_22_6_3_1_1_1_11_4-bb. [Accessed: 14-Jun-2018].
[62] *Folly: AtomicUnorderedMap.h*. Facebook, 2018.
[63] Information Sciences Institute, University of Southern California, “Transmission Control Protocol.” [Online]. Available: https://tools.ietf.org/html/rfc793#section-1.1. [Accessed: 03-Jul-2018].
[64] N. Hunt, P. S. Sandhu, and L. Ceze, “Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures,” presented at the 2011 15th Workshop on Interaction between Compilers and Computer Architectures, San Antonio, TX, USA, 2011, pp. 63–70.
[65] M. Herlihy, “A Methodology for Implementing Highly Concurrent Data Objects,” *SIGOPS Oper. Syst. Rev.*, vol. 26, no. 2, pp. 12–, Apr. 1992.
[66] “Kvalitativa och kvantitativa metoder - Medicinska Fakulteten, Lunds Universitet.” [Online]. Available: https://www.med.lu.se/content/download/63335/476973/file/Kvalitativa%20metoder_del_3.pdf. [Accessed: 11-May-2018].
[67] S. E. De Franzo, “Difference between qualitative and quantitative research.,” *Snap Surveys Blog*, 16-Sep-2011. [Online]. Available: https://www.snapsurveys.com/blog/qualitative-vs-quantitative-research/. [Accessed: 11-May-2018].
[68] “Doing a literature review — University of Leicester,” 2009. [Online]. Available: https://www2.le.ac.uk/offices/ld/resources/writing/writing-resources/literature-review. [Accessed: 09-May-2018].
[69] “What is a literature review?,” *The Royal Literary Fund*. [Online]. Available: https://www.rlf.org.uk/resources/what-is-a-literature-review/. [Accessed: 09-May-2018].
[70] D. Valenzuela and P. Shrivastava, “Interview as a Method for Qualitative Research,” *Arizona State University*. [Online]. Available: http://www.public.asu.edu/~kroel/www500/Interview%20Fri.pdf.
[71] D. Cohen and B. Crabtree, “Semi-structured Interviews,” Jul-2006. [Online]. Available: http://www.qualres.org/HomeSemi-3629.html. [Accessed: 11-May-2018].
[72] B. Törnqvist and R. Persson, “Exjobbsintervju,” 10-May-2018.
[73] University of Minnesota, “Different Types of Evaluation | CYFAR.” [Online]. Available: https://cyfar.org/different-types-evaluation. [Accessed: 23-May-2018].
[74] H. R. Hartson, T. S. Andre, and R. C. Williges, “Criteria For Evaluating Usability Evaluation Methods,” p. 37, 2003.

[75] “std::shared_ptr - cppreference.com.” [Online]. Available: http://en.cppreference.com/w/cpp/memory/shared_ptr. [Accessed: 13-Jun-2018].
[76] “HP Desktop Workstation Z420 | HP Workstations,” *HP*. [Online]. Available: http://www8.hp.com/us/en/campaigns/workstations/z420.html. [Accessed: 17-Aug-2018].
[77] “Intel® Xeon® Processor E5-1650 v2 (12M Cache, 3.50 GHz) Product Specifications,” *Intel® ARK (Product Specs)*. [Online]. Available: https://ark.intel.com/products/75780/Intel-Xeon-Processor-E5-1650-v2-12M-Cache-3_50-GHz. [Accessed: 17-Aug-2018].
[78] “M550 2.5-Inch NAND Flash SSD,” p. 37, 2013.
[79] “Windows 10 Enterprise.” [Online]. Available: https://www.microsoft.com/sv-se/windowsforbusiness/windows-10-enterprise. [Accessed: 17-Aug-2018].
[80] Microsoft, “Visual Studio IDE,” *Visual Studio*. [Online]. Available: https://visualstudio.microsoft.com/vs/. [Accessed: 17-Aug-2018].
[81] Microsoft Windows Dev Center, “Windows SDK-arkiv – utveckling av Windows-appar.” [Online]. Available: https://developer.microsoft.com/sv-se/windows/downloads/sdk-archive. [Accessed: 17-Aug-2018].
[82] “Microsoft .NET Framework 4.7,” *Microsoft Download Center*. [Online]. Available: https://www.microsoft.com/sv-se/download/details.aspx?id=55167. [Accessed: 17-Aug-2018].
[83] Microsoft, “Visual C++ Documentation.” [Online]. Available: https://docs.microsoft.com/en-us/cpp/. [Accessed: 17-Aug-2018].
[84] NuGet, “NuGet.LibraryModel 4.6.0,” Mar-2018. [Online]. Available: https://www.nuget.org/packages/NuGet.LibraryModel/. [Accessed: 17-Aug-2018].
[85] “Application Insights | Microsoft Azure.” [Online]. Available: https://azure.microsoft.com/en-us/services/application-insights/. [Accessed: 18-Aug-2018].
[86] “Microsoft ASP.NET and Web Tools - Visual Studio Marketplace.” [Online]. Available: https://marketplace.visualstudio.com/items?itemName=JacquesEloff.MicrosoftASPNETandWebTools-9689. [Accessed: 18-Aug-2018].
[87] Microsoft, “Visual Basic.” [Online]. Available: https://docs.microsoft.com/en-us/dotnet/visual-basic/. [Accessed: 18-Aug-2018].
[88] R. Persson, “Beskrivning av dialogfönstren för OrderSpray”, Personal e-mail. 23-Aug-2018.
[89] FIS Front Arena, “User Guide: SMART (FCA4108-29).” Available from FIS, Jan-2018.
[90] FIS Front Arena, “OMNI Help (FCA2404-49).” Available from FIS, 28-Jun-2018.
[91] University of California, “Introduction: Reliability and validity.” [Online]. Available:

http://psc.dss.ucdavis.edu/sommerb/sommerdemo/intro/validity.htm. [Accessed: 23-Jul-2018].

TRITA EECS-EX-2018-514