CHALMERS

Concurrency and Parallel Methods for multi-core Platforms

Master of Science Thesis in Software Engineering and Technology

Johannes Fredén Jansson and Jonas Hellberg

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2010

Concurrency and Parallel Methods for multi-core Platforms
© Johannes Fredén Jansson and Jonas Hellberg, 2010

Examiner: Per Stenström

Department of Computer Science and Engineering
Chalmers University of Technology
SE-41296 Gothenburg
Sweden

Tel. +46-(0)31 772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden 2010

Abstract

This master thesis has been written at *Saab Electronic Defence Systems*. Its main purpose is to evaluate the amount of resources needed to rewrite sequential Java components running in Saabs systems, in such a way that they can take advantage of multi-core processors. The research of the thesis addresses issues with concurrent programming in Java, available frameworks and methods and how they can be implemented for different kinds of applications. The research has been used to parallelize a Saab component called *Threat Evaluation and Weapons Allocation (TEWA)*. As a result of the parallelization, a general purpose concurrency framework, called *Concurrent Event-handling and Loop Parallelization (CELP)* framework, has been developed. *CELP* can be used, together with standard Java concurrency tools, to parallelize more of Saabs event based components, making them scalable and safe. This has been shown with the results of the parallelized *TEWA*, which includes good scalability, 15 times performance increase compared to the sequential version and improved deterministic behavior.

Contents

1 Introduction

Electronic Defence Systems (EDS) is a part of the Saab group and is working with different kind of sensors, for example radar IR and laser. The department *OEGPPK*, where the thesis is written, handles system development for communication and tactical functions that connects sensors and weapon systems.

The thesis uses some of the software running on the *Giraffe AMB*, which is one of the leading products in the field of radar in its price range. This software has a major part in the logic used to handle collected data about the position, speed, altitude etc. of air-units (ex. airplanes) within range of the radar.

The hardware used in the products by Saab, more and more depend on the software implementation to give the customer a tactical advantage. This is complicated by the fact that sensors deliver huge amounts of high precision data and it is the responsibility of the software to analyze and evaluate this data to create a set of services for an operator.

The requirements of the software also increase when more functionality is added to the software, both in the sense of complexity and performance. Today there are a lot of computers in many of Saabs products and they work together to provide the services that the customers require. The reason for this approach is that redundancy in computing performance is required, but also that the tasks in the systems are natural to divide among several computers to increase performance. There is however a limit as of how many computers that are practically feasible to have in one product.

The research in multi-core technology, where one single computer can contain several independent processor cores, provides the possibility to use one computer instead of several. In traditional software development with only one processor core present, the parallelization of a system does not necessarily mean increased performance. It is more common that the overhead, because of task switching, decreases the performance of the software. For this reason Saab has chosen to implement their algorithms in a sequential manner. A lot of performance can be gained by changing existing sequential algorithms to parallel algorithms and run the software on a multi-core system.

Saab has no general guidelines or methods for developing concurrent software, or any applications that scale with multi-core systems today. This thesis provides a major study of guidelines for concurrent programming in Java and the study can later be used as a reference document at Saab when parallelizing software components. The study is also used as the theoretical foundation of the master thesis, which is then used to implement a parallelized version of the *Threat Evaluation and Weapons Allocation (TEWA)* component. As a part of this component parallelization, a general framework suitable for Saabs components is developed. The goal with the thesis is that by using the study, the framework and the parallelized *TEWA* component, the software developers at Saab will be able to parallelize other existing components, or write completely new ones that are scalable and safe.

The theoretical part only goes through some of the basic concurrent programming terms and assumes that the reader has some knowledge of concurrent programming and programming in general. The programming language is limited to Java, since that is what is used at *OEGPPK*. The Java software running in the *Giraffe AMB* consists of several components, this thesis focuses on one of the more CPU demanding components.

The implementation has only been tested in a simulated environment, not on the live systems due to the extensive testing and verification needed for software to be allowed to execute there. Since the simulation hardware has a maximum of eight cores we have not been able to test the performance scaling of the implementations on more cores than eight.

1.1 Method

To structure the work of the thesis, it is divided into two main stages. The first stage is the *study*, which also is divided into two parts. The first part is presented as a theoretical chapter (Chapter 2), the second part of it is the experimentation chapter (Chapter 4) in which the theories are tested and conclusions are drawn. The second stage is the *implementation*, Chapter 5 and Chapter 6, where the results and conclusions from the previous stage are used to parallelize the *TEWA* component running in the software for the *Giraffe AMB*. As a result of this component parallelization an general parallelization framework called *CELP*, was developed as well.

1.1.1 The Study stage

The study stage focuses on concurrent programming in Java and investigates tools in current and upcoming Java versions, along with general methods and tools for concurrent programming. It has been created by using the latest published articles on Java concurrency, as well as the most common methods from the best selling books on the subject. An experimentation part is conducted, where some of the latest technologies are tested and evaluated by developing and performing experiments that parallelize common sequential algorithms, with the help from modern concurrency tools.

1.1.2 The Implementation stage

The theories presented and the knowledge gained from the study stage have been used to parallelize the *TEWA* component. The component was first analyzed in several stages and then carefully modified to allow concurrency while still maintaining its initial functionality. Initial testing and verification was done on dual-core Windows based machines, with an environment that roughly simulates the real, *Giraff AMB*, system where the *TEWA* component is used. Further simulation was done in a lab at Saab where the real environment have been realistically simulated. The parallelized software then executed on an eight-core multi-threaded server. To avoid unnecessary and hard to avoid programming and concurrency errors all advanced algorithms have been implemented using *Pair Programming* as the development technique.

2 Java Concurrency and Theory

The basic building block for all Java concurrency frameworks is the Java class `Thread`, which supports an easy way to create and execute threads. Threads are instantiated as normal Java classes, with a *runnable* task representing the task being performed concurrently with the main execution. Alternatively the `Thread` class can be overridden to define what that *runnable* task should be. When the thread is started it is up to the *JVM* and the operating system to decide on which physical processor core the thread will execute. Because the *JVM* handles the thread mapping, it is not possible to control how the threads are mapped between the available CPUs on the system. This is generally not a problem though, since the *JVM* is smart enough to optimally map *CPU* intense threads to available *CPU* cores. [10]

Up until Java 5 the only way to synchronize these threads was through low-level language features. Java 5 changed this by adding the `java.util.concurrent` package, allowing more high-level synchronization. The package was further improved in Java 6 by for example adding new features such as improved concurrent queues and will be further improved in the upcoming Java 7 [1].

2.1 Concurrency Concepts

In this Section some basic concurrency concepts will be covered.

An application is *concurrent* if parts of it are divided in such a way that they are to be run at the same time. If an application is concurrent it does not necessarily mean that the concurrent parts physically run at the same time, they could for example be run by a scheduler to get a concurrent behavior.

Parallel is a subset of *concurrent*. When an application is parallel, the concurrent parts of the software are physically running at the same time[2].

A *task* is a *computation* that is performed in a *thread* and it can be a sub-problem of a bigger problem. Tasks that represent sub problems of a bigger problem can be executed in parallel and combined to give a complete solution.

A *thread* run one *task* at a time. Many threads can process several tasks in parallel to speed up the computations of a problem. For example one problem can be divided into hundreds or thousands of tasks which is processed by only a few threads to speed up the computation.

The *granularity* of a computation is the number of tasks the problem is divided into. A computation that is divided into many tasks is called a fine-grained computation and a computation that does not involve a lot of tasks is called coarse-grained. So, the more tasks the more fine-grained is the computation. [2]

An *atomic operation* is when an action like incrementing a number is executed in one single instruction, with one possible outcome; success or failure.

When queuing up threads and tasks different techniques can be used to decide who is next in line, this is often refereed to as *fairness*. If a thread eventually is guaranteed to get to execute, the queue is *fair* otherwise it is said to be *unfair*.

Context switching is performed when the operative system switches out a running thread, in favor for another thread that is allowed to execute instead. This is done by a scheduler whenever a single CPU runs more then one thread, to get a concurrent behavior. Context switches causes big performance penalties and the operative system therefore generally tries to avoid them.

2.2 Java 5 and Java 6 Concurrency

The `java.util.concurrent` package abstracts away many of the low level difficulties. Programming constructs such as `new Thread()`, `Object.notify()` and `synchronized` should, as much as possible, be avoided. In favor for such low level constructs, the classes from the `java.util.concurrent` package should be used[11]. The `ThreadPoolExecutor` which is widely used in Java based server applications[6] was added via this package as a part of the Executors framework, together with many other classes that help with synchronization. Java 6 added even more features resulting in abstracting away even more of the need to use the low level features in most cases. Advanced queue algorithms is available since Java 6, for example `SynchronousQueue` and `LinkedBlockingDeque`[1][18] which eliminates the need to use error prone `Object.wait()` and `Object.notify()` methods to wait for conditions like empty lists.

2.2.1 The Task Execution Framework

The main idea of the executor framework is to have designated *executors* which execute tasks in separate threads. This framework handles the creation and termination of threads and tasks so the developer does not have to. In the simplest versions the user only needs to create a `ThreadPoolExector` instance with the number of CPUs on the system as argument and then use `ThreadPoolExector.submit(mRunnable)` method to submit a task which a thread will process.

Much performance can be gained by submitting tasks to running threads instead of repeatedly creating and tearing down threads. For example to create and execute a task in a new thread using `new Thread(mRunnable).start()` causes much more overhead than submitting the task to a running thread in a thread pool.

To determine the amount of threads to use the method `RunTime.availableProcessors()`, obtaining the number of *CPU*s (N_{cpu}), can be called. Usually $N_{cpu} + 1$ is an optimal amount of threads for compute intensive non-blocking implementations. How to configure the `ThreadPoolExector` along with rejection polices, thread factories and other configuration options is explained in great detail in *Concurrency in Practice* by Doug Lea[1] as well as in the API documentation for `ThreadPoolExector`[18].

The `ThreadPoolExector` implementation uses a single queue (which is customizable for different needs) to queue task waiting to be processed. This causes some inefficiency and synchronization problems when task counts go up and many threads compete for the queue. The fork/join thread pool implementation of Java 7 has another approach where each worker thread has its own queue and also solves some of the other problems that exists with the `ThreadPoolExector`, see Section 2.3.1.

2.2.2 Collections

Java has both synchronized collections that wrap all vital methods in `synchronized` blocks and concurrent collections which use other means of synchronization that allows for a greater throughput. The concurrent collections were released in Java 5 and Java 6 added some more blocking queues which are excellent when implementing a producer-consumer design.

Synchronized implementations for a lot of the available collections exist and can be created with the methods:

```
Collections.synchronizedCollection(Collection c)
Collections.synchronizedList(List list)
Collections.synchronizedMap(Map m)
Collections.synchronizedSet(Set s)
Collections.synchronizedSortedMap(SortedMap m)
Collections.synchronizedSortedSet(SortedSet s)
```

Synchronized collections are thread-safe but there still exists some problems the developer must be familiar with. To iterate over a synchronized collection until it is exhausted is still thread-safe, but might not give the expected result. If for example a counter says that the last element in a collection is at a certain index and the next operation is getting that element from the collection, another thread can interleave between the operations and remove the next element. The operation might then return `null`, something unexpected or throw an exception. This problem addresses the main issue with synchronized collections, just because their methods are thread-safe does not mean one can stop caring about concurrency. To solve the problem the whole iteration can be synchronized or copied, iterated and returned. If a lot of threads are working with the collection, a performance gain can be seen when copying, but there is of course a trade off between performance gain and copying overhead. [1]

Concurrent collections were first introduced in Java 5. The difference between the concurrent collections and the synchronized collections is that the concurrent collections are designed to improve performance when there is a lot of concurrent access to the collection, while the synchronized collections are designed to be thread-safe. All the concurrent collections provide iterators that do not throw concurrent modification exceptions and that can handle concurrent access to the collection while iterating. This makes it unnecessary to lock the collection when iterating over it. However, the iterators are *weakly consistent* instead of fail-fast as the synchronized versions are. Weakly consistent means that the iterator can tolerate concurrent modifications, it iterates over the elements as they were present when the iterator was constructed and may reflect changes in the collection while iterating. Following is a list of the available concurrent collections:

ConcurrentHashMap
`ConcurrentHashMap` is a lock-free implementation of the `Collections.synchronizedMap()` which is a thread-safe version of the Java `Map`. The `ConcurrentHashMap` should almost always be preferred over the synchronized version when writing concurrent applications, but there are some rare situations where it should not be used. For instance when it is important that the `size()` operation returns an exact value should it not be used, since the number of elements can be updated, and then it can not return an exact value. In some cases the whole collection needs to be locked for concurrent access, this feature is not included in the `ConcurrentHashMap`. These two situations are the *only* cases where the synchronized version is preferred, otherwise the `ConcurrentHashMap` should be used.[1]

CopyOnWriteArrayList
`CopyOnWriteArrayList` is a concurrent replacement for the synchronized `List`. It is, as the name suggests, a `List` that when written to is copied. It preserves its *thread safety*[1]by being immutable. It can not be modified after creation. Copying the entire list on each write is a costly operation, but if there are a lot of reads and not many writes, huge performance gain can be achieved by using the `CopyOnWriteArrayList`. The case where a lot of reads are performed and not that many writes, are common in for example event-notification systems. [1]

[1]The following exhaustive definition of thread safety is made in *Concurrency in Practice*: *"A class is thread-safe if it behaves correctly when accessed from multiple threads, regardless of the*

CopyOnWriteArraySet
The `CopyOnWriteArraySet` works in the same way as the `CopyOnWriteArrayList`, but instead of a `List` it implements concurrent functionality for a `Set`.

SynchronousQueue
A `SynchronousQueue` is a queue that has no internal capacity. If a thread wants to add an element to the queue there must be some other thread waiting for an element. That means that if an element is put in the queue with the `put(E element)` the operation will block until some other thread is performing a `take()`. A `SynchronousQueue` is very well suited for hand-off implementations where one thread must wait and synchronize information with another thread.

PriorityBlockingQueue
The `PriorityBlockingQueue` is an unbounded queue and therefore does not block on the put operation, it still blocks on the retrieval operations. The elements in the queue are ordered in their natural order, which means that all elements must implement the `Comparable` interface.

LinkedBlockingQueue
The `LinkedBlockingQueue` is like the `PriorityBlockingQueue` but it is ordered in a FIFO (first-in-first-out) manner. So, the head of the queue is the element that has been the longest on the queue and tail is the one that has been in the queue the shortest time.

ArrayBlockingQueue
The `ArrayBlockingQueue` has the same properties as the `LinkedBlockingQueue` but it is bounded, that means a fixed sized array is holding the elements in the queue.

DelayQueue
A `DelayQueue` is an unbounded queue where each element has a delay time, which means they can not be taken from the collection before that time has passed.

LinkedBlockingDeque

scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code."[1]

The `LinkedBlockingDeque` is a two sided version of the `LinkedBlock-ingQueue`. This means that it has operations to take elements from the back of the queue as well as from the front. The two sided nature of this queue means that it is effective to use when implementing the work stealing principle[1].

2.2.3 Other Tools

In this section some useful tools from Java 5 and Java 6 for developing efficient concurrent Java application will be covered.

Synchronized Barrier
A barrier is used to make a number of threads wait at a certain point before continuing. In Java this is possible through the class `CyclicBarrier`. The term *cyclic* refers to the possibility for threads to continue and reach the barrier several times until the computation is performed. This is useful for computing result where parallel computations are dependent on previous computations before continuing.

Atomic Variables
Atomic variables are wrapper classes that provide lock-free and thread safe programming on single variables. Instead of locking they provide operations based on low level assembly instructions such as `compare-and-swap`. Though they are very limited and only provide a few methods per class, they can be very useful in various applications. For example in shared loops they can be used to increment or decrement counters. [1][18]

Reentrant locks
A reentrant lock is a form of extension of the synchronized block in Java that inherits the `Lock` interface [18]. Generally, using a concurrent collection from the `concurrent` package is a better solution to achieve synchronization. But in cases where locking is really needed usual synchronized blocks are more than enough and when an application requires more features a reentrant lock can be used. It provides the possibility to interrupt a thread while it is waiting for a lock and the possibility to acquire a lock without being willing to wait for it forever. It should be used carefully though; in contrast to the synchronized block which is released when the block ends, it must be released by the programmer. This can be a problem if some error occurs, if for example an exception is thrown that result in an unexpected execution path. To get around this the release of a

lock should always be put in a `finally` block to guarantee that the lock is released[1].

Latch
A latch is a synchronizer that waits for a number of threads until they all reach a terminal state. It can be thought of as a gate. Until the application reaches a terminal state no threads can pass the gate. It can for example be used when a task has been divided into many subtasks and the application must wait for all of the subtasks to finish before continuing. In Java there exists a latch implementation called `CountDownLatch` which provides a class that can be instantiated with an initial number to count down from. Then threads can use the thread safe method `CountDownLatch.countDown()` to decrease the counter. The thread/threads that wait for the latch calls `CountDownLatch.await()` and when the counter reaches zero they are released. [1]

Future
A `Future<E>` instance is used as an asynchronous response of a computation. The asynchronous methods for executing threads of `ThreadPoolExecutor` and `ForkJoinPool` returns `Future<E>` instances representing the result of a task handled by their worker thread(s). The blocking method `Future<E>.get()` returns the generic result of the computation. This means that a task can be started asynchronously and then some work can be performed and when the result of the task is needed the application can block until it is available.

2.3 Java 7 Concurrency

In the upcoming release of Java JDK 7 which is due to late 2010 there will be an update of the concurrency library `java.util.concurrent`. A package containing the updates which is targeted for the JDK 7 release can be downloaded from the JSR-166 Interest Site [10]. It contains the classes targeted for the next release, another package called extra166y can also be downloaded from there but those classes are not targeted for the JDK 7 release.

The jsr166y package contains the official concurrency news for Java 7, the main improvement in this package is the fork/join framework [10]. This new framework allows for speed-ups where parallel computations is applicable, it is not suited for all types of problems though. It is mostly suited for divide-and-conquer type problems, this means it can not completely replace all kinds of concurrency issued dealt with using Executors and Threads from Java 5 and 6 [17]. Other new classes are the Phaser class and the ThreadRandom class. The Phaser implements functionality similar to the CyclicBarrier and CountDownLatch from Java 5 and the ThreadRandom generates thread specific random data[18].

2.3.1 Fork/Join

The fork/join framework is the most promising news that most likely will be included in the util.concurrent package of Java 7. This framework shows good performance on test cases compiled by Doug Lea, who leads it specification and implementation[20]. In general it does show very good scalability and tests we have performed shows that it scales better in some cases compared to similar frameworks like the ThreadPoolExecutor from Java 5. It has its own kind of thread pool which implementation is similar to the ThreadPool from Java 5, so it does not suffer from creating new threads for each new tasks, it just submits the work to the thread pool. However, there exist some minor limitations in scalability when the number of threads increases due to the fact that the number of tasks that are processed per second decreases[10].

The framework is based on the idea of *Divide and Conquer* style algorithms which takes a set of input data and repeatedly splits the data set in two until it reaches some limit where the problem is small enough, sub problems are then solved and combined to form the full solution. This type of algorithms is usually expressed with recursion where each recursion step is independent of the others, this makes it more suitable for some problems than others[10].

```
Result compute(Task task){
    if(problem is small){
        Solve sequentially
    }
    else{
        split into: Task left , Task right
        fork(solve(left), solve(right))
        Wait for results of subtasks
        compose results
    }
}
```

Listing 1: Divide and Conquer Structure.

This basic idea for the fork/join framework is to create new tasks every time a *Divide and Conquer* algorithm splits the data into two new sub problems, as seen in listing 1. These tasks can then be solved in parallel by letting the framework divide them among the active worker threads. It is a very light-weight framework because it has been optimized to solve such problems where usually the only synchronization is for tasks to wait for subtasks. This approach helps the framework scale very well.

Another reason for its good scalability and performance is that the fork/join framework implements work stealing which is based on the producer-consumer design pattern and the use of *deques*. The idea is to have a number of worker threads, for example equal to the number of processors, which processes a large number of tasks.

The challenge is to divide a problem into suitable number of subtasks which depends as little as possible on each other. To use worker threads and tasks is not new for fork/join but the main difference from the closest version from Java 5, the `ThreaPoolExecutor`, is this use off work stealing [17]. This is implemented by letting each worker thread have its own private work queue that contains tasks which it processes.

The work is pushed and popped to the queue by the thread that owns it. The take operation is protected by a lock but since pop and take (steal) operations work on different ends of the queue they rarely interfere. It is vital to size these tasks so that they are not too big and not too small. If they are to small the overhead of creating tasks will be greater then the gain and if they are to big the threads will not be able to steal work from each other. In other words, the tasks should have a fine-grained partitioning while still out-weighing the overhead of tasks management[19].

Table 1 shows a sequence which illustrates the potential of work stealing and why it is important to have fine granularity tasks. At *t=0* both threads have their work queues full of work, illustrated by *o* for thread one and *x* for thread two. At *t=1* thread two has run out of work; because of inhomogeneous size of the tasks of the problem or uneven distribution of *CPU* time on the *CPU*s which each thread is running. When thread two detects that its work queue is empty it randomly select another queue by scanning through all other worker queues until it finds one that is not empty. In this example there is only one other queue and it is not empty, so it performs a take operation on the end of thread ones queue and steal the work. This example shows why a problem needs to be divided into many sub problems for the work-stealing to be effective. If for example both threads had one task each with four times the size of their current tasks thread two would not be able to steal any work at *t=1* and would go idle instead and waste precious clock cycles [20][10].

Threads & Workqueues, t=0

Threads	Queue content	Queue Size
Thread#1	o o o o	4
Thread#2	x x x x	4

Threads & Workqueues, t=1

Threads	Queue content	Queue Size
Thread#1	o o - -	2
Thread#2	- - - -	0

Threads & Workqueues, t=2

Threads	Queue content	Queue Size
Thread#1	o - - -	1
Thread#2	o - - -	1

Table 1: Work stealing sequence

2.3.2 Phaser

A new feature in the new Java 7 release is a class called `Phaser`. The Phaser was originally a part of the `ForkJoinTask` class, but after a lot of requests it was extracted and put in a separate class. The `Phaser` is an updated version of the `CyclicBarrier` and `CountDownLatch` from Java 5 which are covered in 2.2.3. The new functionality is that after creation new parties can be added and removed from the `Phaser`. In comparison to the `CyclicBarrier` where the number of parties using the barrier is set when the barrier is created, the `Phaser` can add and remove from the number of parties dynamically. When using the new fork/join framework and a barrier is required, the `Phaser` is recommended for optimal performance. [17]

2.3.3 ThreadLocalRandom

When random numbers are generated through the `java.util.Random` concurrently by many threads it can cause a lot of overhead and contention. Rather then sharing a `Random` class instance between threads the static method `ThreadLocalRandom.currrent()` should be used to obtain a `ThreadLocalRandom` instance that can be used to generate pseudo random numbers in threads.

2.4 Java Virtual Machine Tuning

The *Java Virtual Machine* (JVM) has several hundreds of options that can be set to tweak performance or to customize the behavior of a running virtual machine. There are three categories of options that can be given to the virtual machine at startup:

- Regular options that are supported for any version of JVM for a specific operating system.
- Options that begin with -X are non-standard (not guaranteed to be supported on all JVM implementations) that are subject to change without notice in subsequent releases of the JDK.
- Options that are specified with -XX. Not stable and are not recommended for casual use. These options are subject to change without notice. [12]

Almost all of the options regarding concurrency are -XX options, that is why when tuning the JVM one must always keep in mind the risks and the possibility of the option used on the current JVM may not exist on another. Some of the options that are available as -X options are useful for concurrent applications even though it is not their main purpose. Setting the minimum heap size to something bigger than the default values, which is 1/64th of the machine physical memory or at least 3670kB, and the maximum to something bigger than the default which is the minimum of 1/4th of the physical memory and 1GB, is important for memory intense applications. This can be achieved by setting the options:

- -Xms <initial java heap size>
- -Xmx <maximum java heap size>

This is useful for applications with a lot of threads since each new thread get its own stack which by default is set to 256 kB [13]. So, with a lot of threads the memory requirements soon increase. When *thread pool* solutions are used instead, certain situations may require a big amount of tasks to achieve fine granularity through creating many task class instances, which in turn may require much memory. If for example a solution with the fork/join library is to be implemented, which is discussed in detail in Section 2.3.1, several thousands of tasks can be created and these can each contain individual data.

2.4.1 Java Ergonomics

Since the release of Java 5 the JVM default settings regarding default minimum and maximum size of the heap, what garbage collector to use, maximum garbage collection pause time and throughput goal are set dynamically by looking at the platform the system is running on. This is called *Java Ergonomics*. It is a very good functionality if only one JVM is running on a system, but with several JVMs' present there might be problems since it does not take that into account.

If an application does not perform as expected or if several JVMs' are running on one machine it can be a good idea to tweak the parameters to the JVM. These are:

- -XX:+UseSerialGC, use the single threaded garbage collector.
- -XX:+UseParallelGC, use the parallel garbage collector.
- -XX:MaxGCPauseMillis=<n>, Maximum GC pause time goal.
- -XX:GCTimeRatio=<n>, Throughput goal (GC Time : Application time = 1/(1 + n) e.g. -XX:GCTimeRatio=19 (5% of time in GC)).

And as mentioned before:

- -Xms <initial java heap size>
- -Xmx <maximum java heap size>

2.4.2 Garbage Collecting

One very important issue to address when developing concurrent applications is to tune the Java garbage collector. On single core systems with only one processor core a small application that has one percent of its total throughput assigned to garbage collecting seem reasonable. When the same application is run on a multi-core system the throughput spent on garbage collecting increases. In Figure 1 it can be viewed how throughput is affected when moving the software to multi-core platforms. [14]

To avoid these problems there exist three different garbage collectors for the JVM: the serial garbage collector, the parallel garbage collector and the concurrent garbage collector.

The serial garbage collector is most well suited for single core machine since it does all the garbage collecting in one thread. It can also be used on multi-core platforms where applications handle small data sets (up to 100 MB). To enable it, give the -XX:+UseSerialGC flag to the JVM.

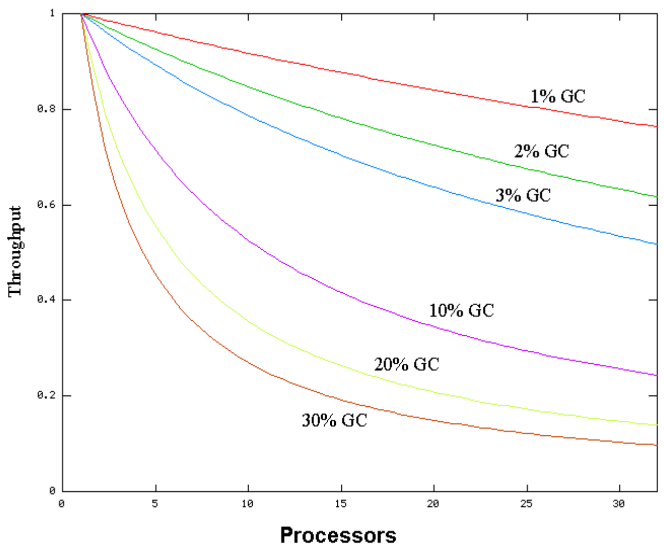


Figure 1: Scalability of a small software running on a unicore system with one percent garbage collecting, up to a 32 core system. [14]

The parallel collector or throughput collector performs minor collections in parallel. It can reduce overhead and is designed for multi-core platforms with medium- to large data-sizes. It can be enabled by giving the -XX:+UseParallelGC flag to the JVM.

The concurrent collector performs garbage collection concurrently with the application. It is enabled by giving the -XX:+UseConcMarkSweepGC flag to the JVM.

The Java ergonomics selects one of the serial and parallel collectors but does not include the concurrent one. On multi-core systems that *"prefer shorter garbage collection pauses and that can afford to share processor resources with the garbage collector while the application is running"* [14], running the concurrent garbage collector can result in big performance gain, so it should always be considered.

2.5 Code Analysis

A lot of different tools exist to help a software developer to write more secure and dependable code. To abstract as much as possible without losing too much performance is important. For concurrency, libraries like the fork/join library, the `ThreadPoolExecutor` from Java 5, different collections, atomic variables and a lot more tools are used to be able to construct dependable software on a higher level. These tools are good ways to work towards safer code. A complement to these tools, using static and dynamic code analysis to find bugs and design flaws in a system is important. However, finding bugs in a concurrent environment can be hard. A lot of the bugs may come from low-probability events that are sensitive to load, timing and might be hard to reproduce. [1]

Static code analysis finds bugs in the code by checking the code, while dynamic code analysis finds bugs by running the program and checking for errors. We will only cover the static analysis.

2.5.1 Findbugs

Findbugs is a tool for static code analysis for Java programs. It is a general bug finding software but can be used to find bugs in concurrent applications. Findbugs is available as a plugin for the Eclipse IDE and is very easy to use. It has support for finding the following concurrent bugs:

Inconsistent synchronization. Reported when an object using a lock on access to its variables is inconsistent. That is, the variable is sometimes locked on access and sometimes not.

Invoking `Thread.run()`. Reported when the method `run()` in `Thread` is invoked directly. Usually it is desired to use `Thread.start()`.

Empty synchronized block. Empty synchronized blocks are often used incorrectly and almost all of the time there exists a better solution.

Starting a thread from the constructor. Introduces a lot of risks like subclassing problems and escaping `this` references.

Notification Errors. These errors occur when a `notify()` or `notifyAll()` is called without changing the state of the monitor they are called from.

Condition wait errors. When a `wait()` or an `await()` is called outside a loop that is looping on the condition the thread is waiting for.

Misuse of `Lock` and `Condition`. Using a Lock as the lock argument of a synchronized block is probably a typo as well as calling `wait()` on a condition variable instead of `await()`.

Sleeping or Waiting while holding a lock. Since there probably are a lot of other threads waiting for a lock it is a bad idea to sleep while having it.

Spin loops. Also known as busy waiting loops takes a lot of CPU and should be avoided as far as possible [1][15].

2.5.2 Concurrencer

Concurrencer is the result of a study done at MIT by Danny Dig, John Marrero, and Michael D. Ernst [5]. It is not a tool for finding bugs, but rather a static code analyzer that updates code for the user. Concurrencer is used to automatically exchange integer variables in the code to `AtomicInteger`, `HashMap` to `ConcurrentHashMap` (for more information see Section 2.2.2) and Recursion to `ForkJoinTask` (for more information see Section 2.3.1). The Concurrencer is a plugin for the Eclipse IDE and is very simple to use. It is simply a matter of selecting the object you want to change to a concurrent version and the Concurrencer will do the rest for you. In case of the recursion exchanged to a `ForkJoinTask` the user just provides a sequential threshold where the algorithm will stop executing concurrently and do the rest of the work sequentially.

2.6 Design

When building a safe concurrent program in Java it is important to follow some general guidelines, more details about these guidelines can be found in *Java Concurrency in Practice* by Doug Lea [1]. Several important design patterns that are very helpful when designing concurrent applications exists along with some patterns which do more harm then good.

2.6.1 General Design

The Java memory model, defined in *Java Language Specification, third edition*[3], has certain rules which decide when objects shared between threads are visible. These rules are very important when writing safe concurrent programs. Shared objects can be divided in two parts, mutable and immutable.

Mutable objects can be protected with a weak form of synchronization using the `volatile` keyword. The specification guarantees that an object declared as `volatile` which is written to is visible for all subsequent reads. Note the specification allows for read and writes of 64 bit values like `long` and `double` to be divided into separate 32 bit operations. This means that normal reads and writes of such values are not guaranteed to be atomic and should always be declared as volatile or be somehow synchronized[3]. Also note that the `++` operation is not an atomic write operation, the `java.util.concurrent.atomic` package provides such methods, see Section 2.2.3. Other ways to make sure that shared mutable objects are visible to all accessing threads at the same is time is to publish them safely as following:

- Static initializing (see Singleton Pattern, Section 2.6.2)
- Storing the reference in a `final` `AtomicReference`
- Only accessing the reference through a lock

Immutable objects are created with the `final` keyword, it can be used both for primitive types and references. It can also be used to assure visibility among all threads *but* it must be initialized before it is published and if it is a reference it is only its final- or volatile fields which are guaranteed to be up to date. This only holds until an object is fully constructed before it is published to other threads.

The `final` and `volatile` keywords rely on which sequences actions are executed, or what *happens before* anything else. The *Happens-before-* and *synchronization* order relationships are defined in the specification and guarantees, among other things, that all actions in a given thread happens in the expected order. More interestingly it guarantees that a thread invoking `Thread.join()` on another thread synchronizes on all actions that thread has performed. It also guarantees the read/write order mentioned when using the `volatile` keyword, that is: writing to volatile field is always visible to a subsequent read of this field.

These relationships also means that a thread must never be started in a constructor since it often is an inner class which keeps a hidden reference to its parent. This could result in a thread getting access to the `this` reference of the constructing class before it is fully constructed. The specification only guarantees that all actions before a thread is started are visible to the started thread, so any actions after starting the thread in the constructor might or might not be visible to the thread [3][11].

It is important to encapsulate and not leak private references, the restrictions of `final` or `volatile` —keywords, or other synchronization, does not hold if the protected object is dereferenced by another unprotected reference. As a good practice all non-mutable objects should be made final and any mutable shared objects should be accessed with synchronization. It will not, for example, help if a `ConcurrentHasMap` instance is published to two threads if it is not guaranteed to be an up to date reference of the object. Any references that can be made static should be so and be initialized when they are defined[1].

2.6.2 Design Patterns

Here some general useful *design patterns* as well as *anti-patterns*, which should be avoided since they describe design techniques that are not recommended, are described and discussed.

The Java Monitor Pattern
The *Java Monitor Pattern* is Javas version of the Monitor Pattern proposed in the 70's. It can be used by surrounding code in the `synchronized` block. Less sophisticated synchronized collections like `Vector` uses it to handle concurrent writes and reads. It is generally better to use more sophisticated collections like the ones mentioned in Section 2.2.2 to achieve synchronization [1]. This pattern also supports the `Object.wait` and `Object.notify` operations which can be used to block threads and notify other threads when they should wake up [3]. These actions are very error prone and again it is best to use well tested collections in the `java.util.concurrent` package which use these operations to implement safe and well tested means of synchronization.

The Producer-Consumer Pattern
The *Producer Consumer Pattern* is based on the idea of a set o producers which add data to queue that is shared with a set of consumers. The producer and consumers need not be aware of each other, they should only communicate by taking and adding data to the shared queue. This is a very useful pattern and it is best implemented with the blocking queues introduced in Java 5 and 6[1]. It is used in the fork/join and Executor frameworks to implement the sharing of tasks among worker threads.

Normally in a *Producer-Consumer* implementation the producers asynchronously add work to the work-queue and the consumers synchronously obtain work by blocking if the queue is empty. Another way to do is to let both producers and consumers block, this means that the queue instead works as a direct hand-off between threads. Such a queue, called `SynchronousQueue`, was first introduced in Java 5 but was later replaced with a new version in Java 6. The new queue greatly outperforms the Java 5 version regardless off fairness, but mostly in fair mode [6]. It can be used as a producer-consumer direct exchange queue, by passing it as the queue implementation when creating a `ThreadPoolExecutor`. If the conditions are right it can greatly improve the `ThreadPoolExecutor` performance but it is most suitable in a many consumers - few producers scenario so that there always is a consumer thread waiting to accept work [1][6].

The queues can be bounded or unbounded. Generally it is better to have a bounded queue and handle the case of it getting full to avoid resource exhaustion problems[1]. This is possible with the blocking queue implementations of the `java.util.concurrent` package. For example, by using the `LinkedBlockingDeque.offer()` method which returns false if the queue is full and the element could not be inserted [18].

Singleton Pattern

The singleton pattern is used to have a private single instance of a class which can be easily obtained by a using a public method as seen in listing 2. The problem with this example in a concurrent environment is the race condition[2]when checking if the instance is null. It is possible that a thread obtains an instance which has not been fully constructed.

```
class Singleton{
    private static Singleton s;

    public static Singleton getInstance(){
            if(s == null){
                s = new Singleton();
            }
            return s;
    }
}
```

Listing 2: *Not thread safe* Singleton Pattern

An easy way to make the example in listing 2 thread safe is to use static initialization as seen in listing 3. This causes the JVM to initialize the object when the class is loaded which guarantees that it is fully constructed before it is published.

```
class Singleton{
    private static Singleton s = new Singleton();

    public static Singleton getInstance(){
            return s;
```

[2]A race condition is defined as a timing error, where the result of some action depends on the execution timing of two ore more threads[2].

```
    }
}
```

Listing 3: *Thread safe* Singleton Pattern

Barrier Pattern
The idea of the Barrier Pattern is to synchronize a set of threads at a given point which they all must reach before they continue. The class `CyclicBarrier` in the `concurrent` package supports this, see Section 2.2.3. The class `Phaser` from Java 7 also acts as a barrier and is designed to be used together with the fork/join framework, see Section 2.3.2.

The Double-checked locking AntiPattern
The Double Checked AntiPattern is explained in listing 4. This might seem efficient since it prevents some contention of the lock but it is very dangerous. The reason for this is that `obj` might get a non `null` value before it is constructed and a thread might return with a reference to an object in an invalid state[1].

```
class DOCClass{
    Object obj

    obj getObject(){
            if(obj == null){
                synchronized(this){
                    obj = new Object()
                }
            }
            return obj;
    }
}
```

Listing 4: *Not thread safe* D.C.L Anti-Pattern

2.7 Theory Conclusion

There are a lot of different tools that can be used to tweak the performance of a Java application. There is almost always a better solution than using synchronization blocks over several lines which can lead to poor concurrency . For single integer/boolean/long operations one can use the non-blocking atomic variables and gain a lot of performance since they require only one assembly instruction that need no synchronization.

To gain most throughput on multi-core platforms the concurrent garbage collector should be used and when tweaking an application the garbage collector should always be taken into account. As mentioned in

Section 2.4 JVM tuning is a very important thing to consider and a lot of performance gain, gained by programming in a good concurrent way can be lost if the garbage collector, heap size and thread stack size is not set properly. Another issue/opportunity with the garbage collector is the Java ergonomics (Section 2.4.1), which sets the size of the heap, which garbage collector to use and some more properties depending on which platform the system is running on. This is very good when running a single JVM on a machine, but when several JVMs are running on a system it does not take that into account. So, one must keep that in mind if efficient use of several concurrent JVMs is desired.

When trying to implement a safe multi-threaded application with the least effort it is import to use the right tools. In our research we have found that the `java.util.concurrent` package has many useful tools that should be used as much as possible to write safe programs with a minimum amount of effort. This package together with tools like `FindBugs` and `Concurrencer` allows for writing, refactoring and performing static analysis of concurrent programs that are safe in the shortest possible time.

Since Java is designed with concurrent support in mind with keywords like `synchronized` it is easy to use these low level tools instead of safer, more effective and often easier to use tools from the concurrency package. It is however important to use some basic keywords like `final`, `volatile` and `static` as explained in Section 2.6.1.

When designing concurrent solutions that are handled by the `ThreadPoolExecutor` and/or the `ForkJoinPool` the tasks should be constructed with isolation in mind. The `ThreadPoolExecutor` should not have any task dependencies which could cause a deadlock and thread-safety can be guaranteed if the task data is isolated from each other. In the `ForkJoinPool` case the basic structure of a *divide-and-conquer* algorithm assures isolation between the tasks, which only wait for each other through *join* operations. If no other dependencies are introduced thread-safety can be guaranteed through isolation.

When sharing more data which is accessed in more complex ways than incrementing a number it is often not enough with atomic numbers, instead concurrent collections can be used to achieve consistency. We found that many concurrent collections such as `LinkedBlockingQueue` can be used safely when sharing data between threads. The different implementations are good for different problems. The `LinkedBlockingQueue` is good for implementing the producer-consumer pattern, a `LinkedBlockingDeQue` are good when implementing the work stealing principle. Collections like `ConcurrentHashMap` has many uses as well but some operations like the `size` operation is slower than in the synchronized counterpart and returns a value that may not be up-to-date. There are a lot of issues with the synchronized collections, for example: the individual operations are thread safe but that does not mean that compound operations are safe (eg. iterating over a synchronized collection) and the programmer can not stop caring about concurrency. A full list of concurrent and synchronized collections and further explanations can be seen under Section 2.2.2.

3 Tools

To develop and test our implementations, and perform analysis on existing systems used within Saab we have used several different tools and simulated environments. For a full list of tools used, see appendix A.

The eclipse *Integrated Development Environments (IDE)* has been used for development and for local —and remote debugging. The *Threat Evaluation and Weapons Allocation (TEWA)* component that has been parallelized, is very complex; therefore several analysis tools have been used to perform both static and dynamic analysis. Study related experiments have been analyzed as well but primarily with dynamic tools that monitor CPU and memory usage etc. Since most of the experiments and simulation has been performed on a multi-core server, several tools have been used to monitor them via a remote machine.

The development of the experiments, concurrency framework and *TEWA* parallelization has been done on Windows machines. Therefore initial testing has been done with local windows based environment which roughly simulates the real system. More advanced testing and verification has been performed in a lab environment which very realistically simulates the real *Giraffe AMB* system.

3.1 Development and Debugging

The *eclipse IDE* running on a Windows machine has been used as the main development environment. The study related experiments has been developed in this IDE and debugged with the built in debugger for troubleshooting and to verify their correctness. Critical classes are also tested using *JUint* test-cases to further verify them. To draw any conclusions in regards to performance, runtime benchmarks have been performed for all implementations, using the *JFreeChart* library to generate vector graphic based charts. As of the writing of this thesis, the current Java JDK version is number 6, so *JRE 1.6* has been the main execution environment of all tests and implementations. To add support for the latest concurrency tools supported by Java 7, an external library, `jsr166y.jar`, which contains all the coming Java 7 concurrency tools has been used.

The Linux based lab simulated environment is isolated from the normal network, therefore *eclipse*, which includes the *Java JDK*, has been installed on this Linux system as well. The intent of this is to attach the *remote debugger* on the remote JVM which runs the parallelized software on the multi-core server. The *Giraffe AMB* radar system is simulated on several different machines which runs Linux and communicates via an isolated network. This network is not accessible from the external network due to security reasons. In contrast to the windows based simulation, a *GUI* is available when running this, lab environment, simulation.

3.2 Analysis

The *TEWA* component has been statically and dynamically analyzed with different tools to find internal dependencies as well as to measure its performance. The documentation tool *doxygen* together with the graph drawing software *Graphviz* are used to perform a static dependency analysis of the component. This software creates *call/caller* graphs which show how methods invoke each other, it also creates class diagrams that represent the dependencies between classes. Further static analyze is supported by *FindBugs* which finds common programming errors, including inconsistent locking and other dangerous synchronization problems.

Dynamic analysis are done with *JConsole* and *VisualVM*, they both attach to an application and performs runtime analysis of CPU, memory, thread status and more. *VisualVM* also support runtime profiling analysis to find hotspots in the code that take up a large portion of CPU time, and therefore are suitable to parallelize. Both *JConsole* and *VisualVM* support remote monitoring of applications. With *VisualVM* this is enabled by running the *jstatd* application on the server that is running the remote JVM. To gain more information about a specific application running on a remote machine, a JMX connection can be used. To allow a remote JVM to listen for such JMX connections, it must be started with the correct arguments[22].

Both *JConsole* and *VisualVM* are Java based, they are therefore multi-platform, and they have been used in both Linux and Windows environments. When running a Windows environment, the tool *perfmon* has been used to monitor more operative specific information, like context switching and current CPU load on the different cores. A similar, but more limited, tool called *top*, has been used in the Linux environment to monitor how the load is distributed between the different CPU cores.

3.3 Simulation Environments

The Windows based simulator uses a set of programs which communicates via *CORBA* interfaces, just as the real components do. When running the TEWA component in this environment, the real system can roughly be simulated using the test equipment. This is valuable when performing initial testing, as well as executing and verifying use cases. The software can then be easily debugged in *eclipse* without the need to set up a remote debugging session. This environment is limited though, it does not support the real GUI used, it is difficult to simulate any realistic scenario and the architecture is only a dual-core machine which is not enough when performing scalability analysis.

The lab simulated environment simulates real scenarios very realistically. It has a view that shows a map, among with position and other data, including threat values and designation to air units from the *TEWA* component. The *GUI* and the logical components, like the *TEWA*, runs on different machines. The logical components including the *TEWA* run on the eight-core multi-thread server. In this environment the parallelized components can be tested for correctness, scalability and be compared to its sequential counter-parts in a realistic way.

4 Experimentation

To test the theories and compare algorithms with various characteristics, using different frameworks from Java versions 5, 6 and 7, a few test cases have been implemented. Timings are measured on these different implementations, to see how well they scale when tested on a multi-core architecture. To be able to draw any conclusions about the timings, the tests have been performed without *garbage collection* (GC) during the actual test runs. GC is instead performed between test runs. To assure that the Java JIT compiler does not affect our tests, each unique test run is preceded by a warm up run. This minimizes the risk of any compilation from byte code taking place when the timings are measured. Each test run is also further strengthened by taking the mean of several runs. To monitor memory usage and OS related issues, like context switching, we have used *jConsole* and the windows tool *perfmon*.

Algorithms with some different characteristics have been implemented using different methods:

- *Matrix multiplication*, where all basic computations can be performed independent of each other, which makes it suitable for concurrent computations.

- *Merge sort*, which cannot, in contrast to the previous, compute all tasks individually since tasks must wait for subtasks to compute their results.

These problems are suitable to be solved directly without any synchronization between tasks, except for the waiting for subtasks in merge sort. To test various synchronized collections, from the various Java versions, along with design patterns and atomic operations, two other implementations have been developed as well:

- *Minimum spanning tree* problem, which requires synchronization with shared lists

- *Producer-consumer* implementation, where producers communicate their computation results through a shared queue, with consumers. The consumers then use the results obtained from the queue, to dynamically updated graph.

The goal of the tests have been to find optimal settings for each algorithm on different platforms and to compare the different methods in sense of scalability, consistency, isolation and programming effort.

4.1 Merge sort Implementation

Merge sort is one of the most well-known sorting algorithms and it is a divide and conquer algorithm. Divide and conquer algorithms are especially well suited for the fork/join library, since it is what the fork/join library was designed for. The merge sort algorithm sorts a sequence of comparable items by dividing them in half and then sorting the divided halves recursively. A sequence is divided until it reaches a single item, then that item is merged and sorted with the other half of that subset, which can be one or two items. The result of the merge and sort is then merged and sorted with the corresponding result of the other half of its subset and so on. A sequential version of merge sort can look like in listing 5.

```
void mergeSort(a,  tmpArray, left , right ) {
    if( left < right ) {
        center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }
}

void merge(a, tmpArray, leftPos, rightPos, rightEnd ) {
   leftEnd = rightPos - 1;
   tmpPos = leftPos;
   numElements = rightEnd - leftPos + 1;

    while( leftPos <= leftEnd && rightPos <= rightEnd )
        if( a[ leftPos ].compareTo( a[rightPos] ) <= 0 )
            tmpArray[tmpPos++] = a[ leftPos++ ];
        else
            tmpArray[tmpPos++] = a[rightPos++];

    while( leftPos <= leftEnd )
        tmpArray[tmpPos++] = a[leftPos++];

    while( rightPos <= rightEnd)
        tmpArray[tmpPos++] = a[rightPos++];

    for( int i = 0; i < numElements; i++, rightEnd--)
        a[rightEnd] = tmpArray[rightEnd];
}
```

Listing 5: Sequential merge sort.

At the recursive call to `mergeSort`, the divide part of the algorithm is performed. The two subparts are totally independent from each other until the result is to be merged. That is why that part can be parallelized by invoking the `mergeSort` on each subpart as a separate task, wait for the two subparts to complete, and then merge and sort the result back into the main array.

```
void mergeSort(a, tmpArray, left,  right )
{
   if( left < right ) {
     center = ( left + right ) / 2;

     concurrentInvoke {
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
     }

     merge( a, tmpArray, left, center + 1, right );
```

```
    }
}
```

Listing 6: Concurrent merge sort.

As seen in listing 6 this approach enables the possibility to execute the subparts concurrently (fork them), wait until the two subtasks complete (join them) and then merge the result. Even though the worst case time complexity still is O(n log(n)) as in sequential merge sort (since the decrease in computational time is constant due to a limited amount of processor cores), huge performance increase can be observed when sorting big data sets.

Three implementations with merge sort (using a sequential approach, the new Java 7 fork/join approach and an approach using the Java 5 `ThreadPoolExecutor`) have been developed and compared focusing on scalability, consistency and isolation. The sequential algorithm has no parameters, except the size of the input data, affecting the execution time of computing the result. The parallelized versions, the `ThreadPoolExecutor` and fork/join implementations, on the other hand, have some settings that affects the execution time.

4.1.1 Fork/Join Implementation

Since the fork/join implementation always has as many threads as processor cores, no context switching is needed. Instead, to balance the computations it has implemented work stealing, work stealing is explained in detail in Section 2.3.1. Since it uses work stealing with a constant number of threads in the fork/join pool, the number of tasks is the main concern when tweaking the performance of the algorithm. The number of tasks is set by adjusting the sequential threshold of the algorithm, which specifies at what limit, it is not feasible to divide the work into more concurrent subtasks, because of granularity problems with too much overhead. In more general terms, the sequential threshold is the limit where the divide and conquer algorithm stops assigning its work to new tasks, and the task are computed sequentially. It also specifies the size of the smallest task. The larger the number of tasks, or the more fine-grained the granularity is, the more work-stealing can be performed, but the overhead for spawning new tasks also increases and the amount of memory required increases because more tasks need to be queued.

When dealing with problems like merge sort it is hard to follow the recommendation specified in the fork/join api: *"As a very rough rule of thumb, a task should perform more than 100 and less than 10000 basic computational* *steps."* [17]. The reason for this is that the largest task always is as large as the number of elements in the array to sort (O(n)), the second biggest is as big as half the number of elements and so on. To find the best sequential threshold tests have been run on one 8 core machine with hyper-threading and on a dual core machine. The result can be viewed in Figure 2.

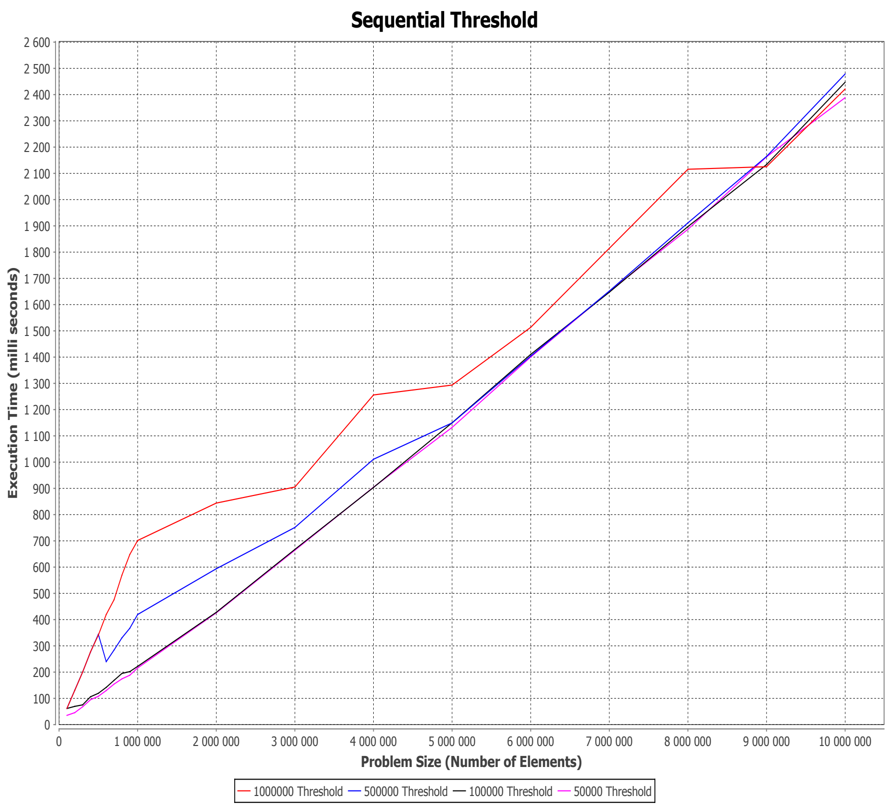


Figure 2: A comparison of different fork/join thresholds.

From Figure 2 it is clear that the sequential threshold may be pretty high without actually affecting the performance. There is no reason to set the threshold lower than necessary because that would result in more subtasks and more subtasks require mote memory.

4.1.2 ThreadPoolExecutor Implementation

The `ThreadPoolExecutor` take one parameter, which is the number of threads to use. Since the merge sort blocks while waiting for subtasks to complete, the number of tasks to compute, must *not* be more than number of threads in the `ThreadPoolExecutor`, otherwise the computation will deadlock. A deadlock will occur if all current threads blocks, waiting for subtasks to complete, but there are no more available threads in the thread pool. This is called *thread starvation deadlock* and there is always a risk that it will occur if tasks submitted to a `ThreadPoolExecutor` are not independent[1]. Considering this, regular threads could be used instead, since one thread is needed per task anyway, making the work queue of the `ThreadPoolExecutor` redundant. But the `ThreadPoolExecutor` is easier to use, making it preferable with this type of problem. This is much thanks to the `ThreadPoolExecutor.invokeAll` method that allows for easy submitting of subtasks. This makes the blocking part of the algorithm a lot easier then for example manually starting new threads in each task. However, regular threads could be used anyway, possibly resulting in a small performance improvement since the `ThreadPoolExecutor` has some overhead for task and queue management.

It is difficult to determine which thread count that result in the fastest execution times. The amount of threads may vary between different platforms, different algorithms and different implementations of an algorithm. To see how a various number of threads affect the execution time, tests have been performed on a dual core processor and on an 8 core processor with hyper-threading. The result can be viewed in Figure 3 and in Figure 4. These two graphs shows how various amount of threads affect the execution times, on different problem sizes, and different systems with a varying amount of processors. From the graphs it can be seen that an efficient amount of threads on the 8 core machine with hyper-threading is around 80 threads and on the dual core machine it is around 20. The difference however, is not that big. It can be seen that for small problem sizes, a smaller amount of threads is better but when the problem size increases a bit more threads are better.

4.1.3 Comparison

The different methods all have their pros and cons but, since the fork/join library is made for these kinds of problems it should overall perform best.

Scalability

The merge sort algorithm is a scalable algorithm. Its work can be divided among a lot of worker tasks. With a big problem size, a system with a fixed amount of processors can at some point during the execution keep all of them busy to some extent. But, since the division of work requires

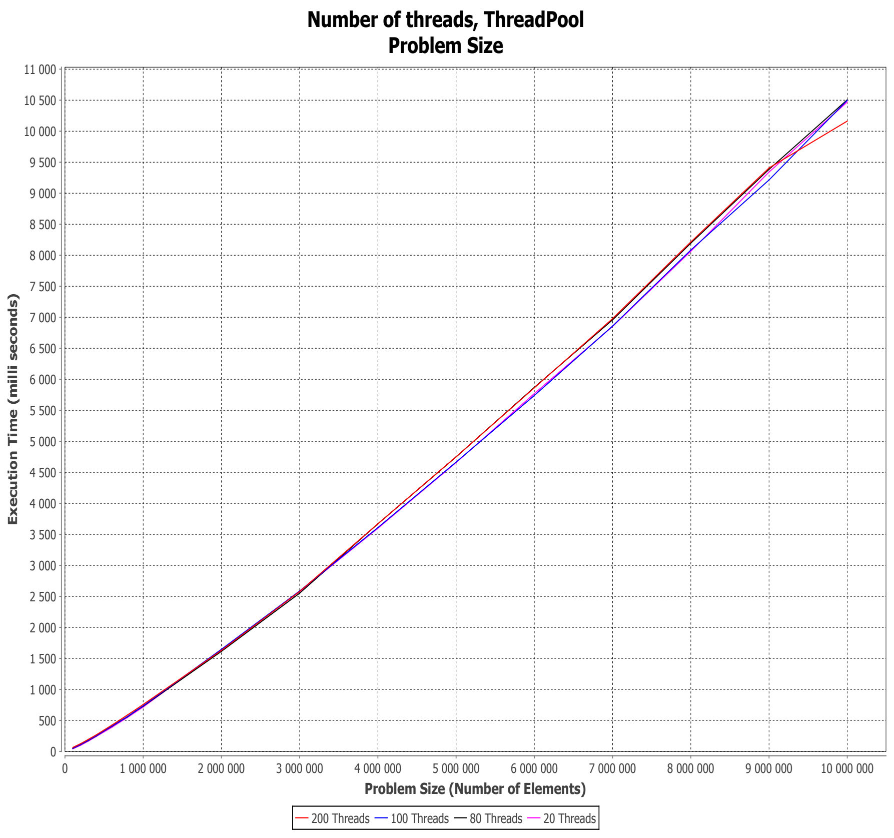


Figure 3: Execution Time vs. Problem Size with different amount of threads on a dual core machine.

each subpart to wait for its underlying tasks, the algorithm can not be fully parallelized. Therefore the scalability is limited and the speed up achieved can not come close to of the merge sort is far from linear.

Different parts of the problem require different amounts of ”merging”, that is why the problem gets unbalanced. The time some of the top merging operations must wait before being able to merge its subparts, varies a lot on big calculations. While a task is waiting, the thread currently occupied by the task is also waiting. The `ThreadPoolExecutor` implementation and the fork/join implementation handle this in two different ways:

The `ThreadPoolExecutor` freezes the current thread and does nothing. When it has used up all its assigned threads it stops assigning its work to subtasks and calculates the rest of the problem sequentially. While a task is waiting for its subtasks to complete, the task in the thread can be exchanged

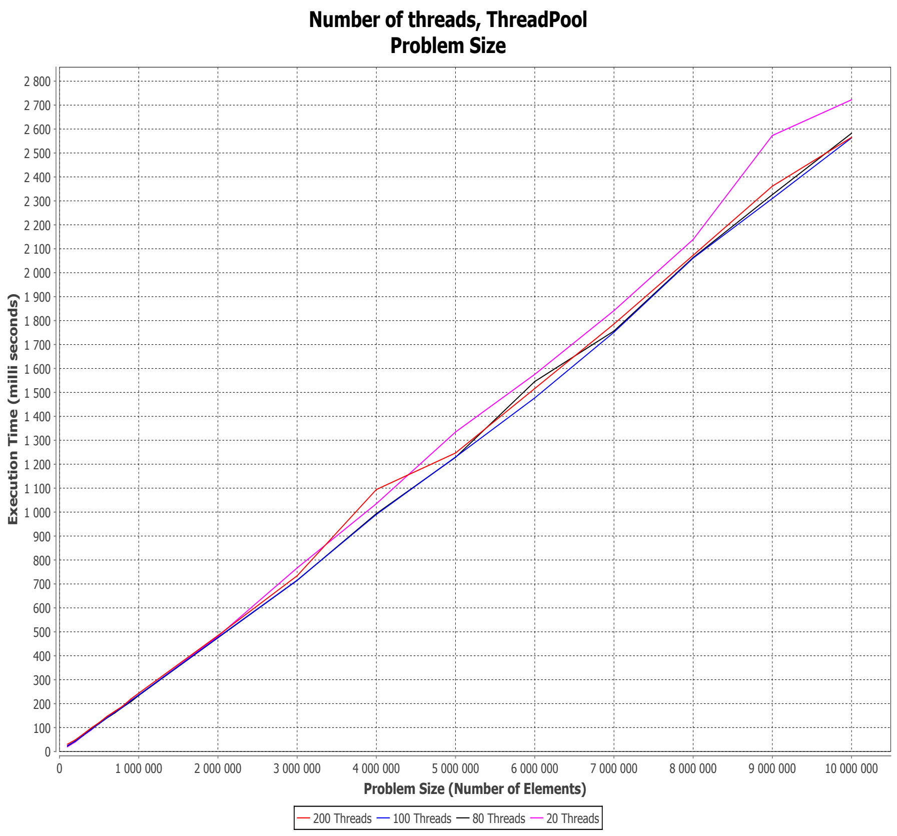


Figure 4: Execution Time vs. Problem Size with different amount of threads on an 8 core machine with hyper-threading.

for another task while that task is waiting for execution. This is performed by, so called, work stealing. That is exactly what it does; a so called context switch is performed. This can be taken advantage of by creating a bit more threads than there are processor cores on the system and letting the operating system context switch between these threads. One must keep in mind that since a context switch has a lot of overhead, creating too many threads may decrease performance. In the sense of scalability it is not feasible to do too much context switching and after a certain amount the overhead is just too big. This makes the `ThreadPoolExecutor` implementation scalable to a certain level but not further than that.

The fork/join is more scalable than the `ThreadPoolExecutor` algorithm. It always creates as many threads as there are processor cores on the machine and then divides the work among these threads. While a task is waiting for its subtasks to complete the task in the thread can be exchanged for

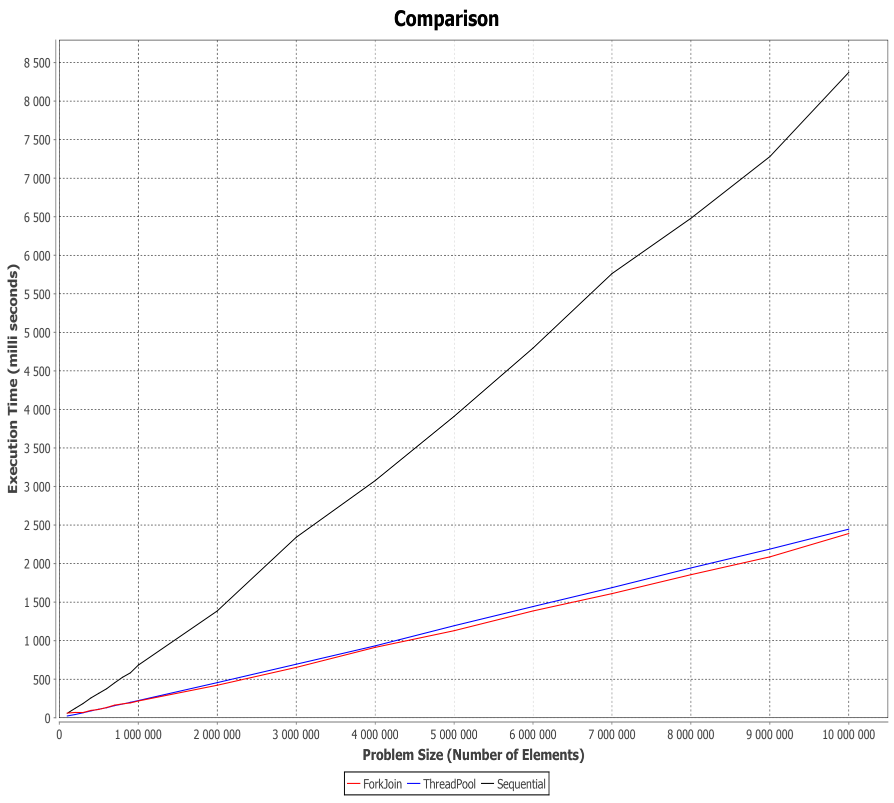


Figure 5: The difference in execution time for fork/join, `ThreadPoolExecutor` and a sequential implementation of merge sort, on an 8 core machine with hyper-threading.

another task that is waiting for execution, by performing so called work stealing. Work stealing is discussed in 2.3.1.

The difference between fork/join, `ThreadPoolExecutor` and a sequential implementation can be viewed in Figure 5 and 6. As one can see in the figures, both the `ThreadPoolExecutor` and the fork/join implementation scales very well up to 8 cores with hyper-threading (simulating 16 cores). In theory the fork/join should scale a lot better than the `ThreadPoolExecutor` on more cores.

Memory

The memory required for the sequential implementation compared to the two concurrent ones is a bit less memory than for the `ThreadPoolExecutor` implementation and a lot less memory than for the fork/join implementa-

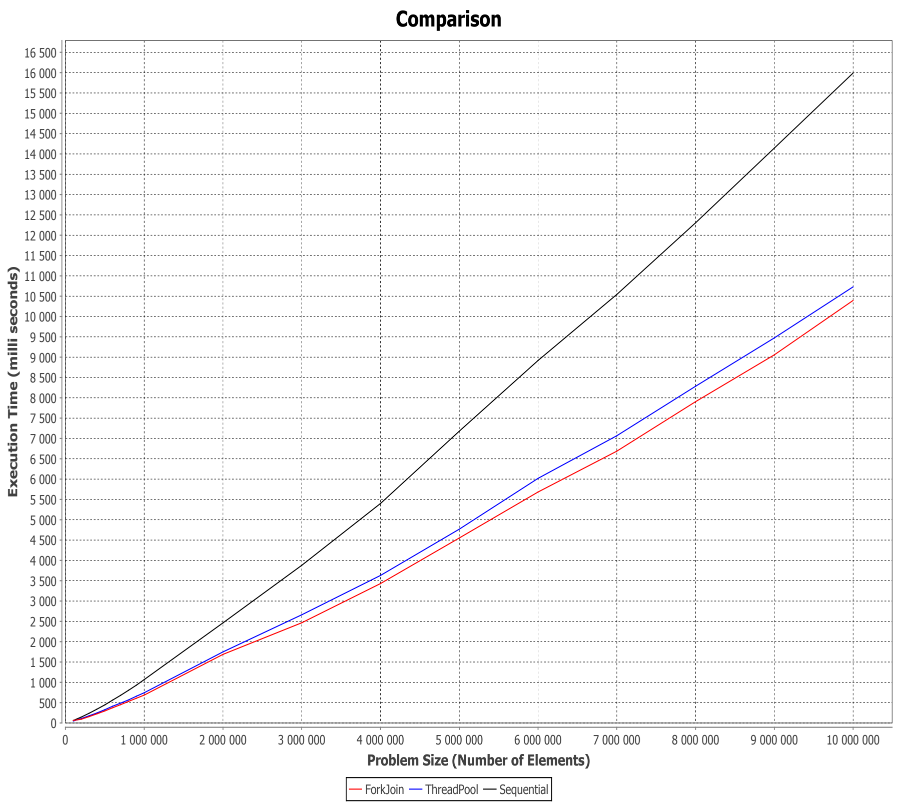


Figure 6: The difference in execution time for fork/join, `ThreadPoolExecutor` and a sequential implementation of merge sort, on a dual core machine.

tion. The `ThreadPoolExecutor` implementation creates a new thread and a new class for each task. Since the fastest configuration does not require very many tasks, the total memory required is not a lot. The fork/join implementation on the other hand, where a lot of tasks are created in the fastest configuration, may require huge amount of memory when dealing with large computations. The fork/join creates a queue of tasks to each worker-thread to be able to perform work stealing, and for this work stealing to be efficient, there must be many small tasks. These tasks are new objects and when dealing with a lot of tasks, the total memory requirements can reach above 2 gigabytes.

Programming Effort and Thread Safety

The fork/join approach is by far the safest and easiest way to program a divide and conquer algorithm, which is not surprising since it is what it is designed for. On the other hand, the `ThreadPoolExecutor` approach might cause a *thread starvation deadlock*, caused by the thread pool running out of threads to handle more subtasks. This complicates the implementation and if done poorly, it might be fatal for the application. The fork/join implementation does not have this problem. It will queue the result and switch tasks when one task is waiting for a subtask, removing the risk for the *thread starvation deadlock* situation. This makes it safer and easier to implement.

Conclusion
The merge sort example is the one used by the authors of the fork/join framework, so it should really show the advantages of it. [10][8] It does show the advantages, but on the whole it does not completely outperform the other implementations and it uses a lot of memory compared to the other solutions. But, a performance increase is seen and the scalability is better than with the others. Looking at the safety and programming effort; programming a solution with the fork/join is a lot simpler and the risk of a deadlock which the other implementations suffer from is gone.

4.2 Matrix Multiplication

With this implementation two thread frameworks (fork/join, Executors) and one barrier synchronized worker thread pool will be compared. A Matrix multiplication calculation is a good example of how a computation can be parallelized. Because when multiplying two matrices the resulting matrix indexes can be computed individually. There is no need for any synchronization or blocking in the form of task dependencies so this comparison show how the implementations work in a fully non blocking case. For example, an 8x8 matrix times an 8x8 matrix forms a new equally sized matrix, where all indexes can be computed separately. This theoretically means that $8 * 8 = 64$ individual tasks can be created which are solved in parallel on a machine with an equal amount of cores. In the above example this would mean that the problem is split six times, $2^6 = 64$, to create the 64 sub-tasks.

Three different concurrent implementations have been developed for this algorithm, similar to the previously discussed merge sort. A sequential implementation is made for verification and the concurrent implementations are made with the `ThreadPoolExecutor`, `ForkJoinPool` and a thread pool synchronized with the `CyclicBarrier`. All implementations have different settings which influence their execution time.

4.2.1 Barrier Implementation

By manually constructing worker threads that compute individual index ranges the problem can be solved very efficiently. When using this approach no work queue exists and threads are started directly with a designated `Runnable` (a task) which computes the assigned indices. The `CyclicBarrier` class is used but the computation is performed on only one cycle, so it would be possible to get the same effect by using for example a `CountDownLatch`, which counts down for every thread that has finished. The `CyclicBarrier` is created with the number of worker threads plus one as argument, the main thread also waits for the workers using the barrier which is why the extra one is needed.

When using an approach with many threads, the matrix can be computed very fast as seen in Figure 9. There is a certain amount of memory needed to handle big amount of threads and there is a risk of a lot of overhead due to the context switching. But when running this implementation the memory requirements were rather low when measured with *jConsole* and there was almost no *context switching* when monitoring Windows with the *perfmon* tool. The reason for the low amount of context switching is that the implementation is basically lock free, except for when joining the all the threads in the end of the computation. This also leads to good scalability with an amount of threads equal to the number of processors, since the threads are allowed to finish their computations before being switched out. If they where switched out, it would result in them having to be switched back in to get CPU time again and finish the computation, resulting in unnecessary overhead.

Since this type of solution does not use a work queue and the developer is fully responsibly of the threads life cycles it has several problems. It is for example not possible to start a group of barrier synchronized workers and assign them more work while they are running since they do not share any work queue or any other individual queues. This means that all threads have to be started every time the computation is performed which results in large overhead for starting and tearing down the threads. The `ThreadPoolExecutor` implementation shown in the next section deals with these problems.

4.2.2 ThreadPoolExecutor Implementation

The `ThreadPoolExecutor` approach is implemented by creating runnable tasks, which are submitted, in an iterative manner, to the executor instance. This is comparable with the `CyclicBarrier` implementation but the tasks do not have to be handed directly to the threads when they are stared, the framework handles all thread and task management. The tasks are put in the queue with `ThreadPoolExecutor.submit`, waiting to be processed by a thread. It is also possible to hand over the tasks indirectly to a thread without waiting in a queue by using a `SynchronousQueue` as the queue implementation (which is passed with the constructor of `ThreadPoolExecutor`). Using a `SynchronousQueue` is only useful when there always is a worker thread waiting to get work from the queue [1]. This is also implemented and the result can be seen in Figure 4.2.4.

The basic implementation of this problem with the `ThreadPoolExecutor` can be implemented as shown in listing 7, where the `ThreadPoolExecutor` is created and then the computing tasks are created. After each task has finished its computation a `CountDownLatch`, which is set to the number, of threads is decreased. The method `latch.await()` blocks until the latch reaches zero which means that the computation is finished.

The `ThreadPoolExecutor` is created using the `Executors.newFixedThreadPool(NTHREADS)`, with the number of worker threads as argument. A customized instance with a `SynchronousQueue` is created for comparison, as well.

```
ThreadPoolExecutor exec(N_THREADS)
Latch latch(N_THREADS)
...
exec.submit(new Runnable(){
    @Override
    public void run() {
        for(all index in this tasks index range){
            Calcualte result into result matrix
        }
        latch.countDown()
    }
})
...
latch.await()
```

Listing 7: Matrix Multiplication using ThreadPoolExecutor.

Tests show that the amount of threads may have a great impact on performance just as with the `CyclicBarrier` implementation. Compared to the merge sort algorithm, where any non leaf tasks have to merge the results of its subtasks and thus have a dependency between tasks, matrix multiplication does not have any dependencies between tasks. This makes the implementation with the `ThreadPoolExecutor` simpler since there is no risk for *thread starvation deadlock*. Deadlocks might occur when using fewer threads than there are tasks and there are dependencies between tasks[1].

The scalability chart in Figure 7 show how the `ThreadPoolExecutor` with a standard *FIFO* queue scales when the amount of threads increase, it has its peek around sixteen threads. This optimal thread count is in line with the general $N_{cpu} + 1$ for a block free computation intense implementation.

The `ThreadPoolExecutor` with a `SynchronousQueue` scale slower than the one with a standard *FIFO* queue and does not reach optimal performance until around 32 threads, as seen in Figure 7. The reason for this is that the `SynchronousQueue` requires that there always is a waiting thread to receive a task, to achieve optimal performance. For this example it is necessarily not the case when the thread count is low, which makes it a bad choice since a normal *FIFO* queue scale faster.

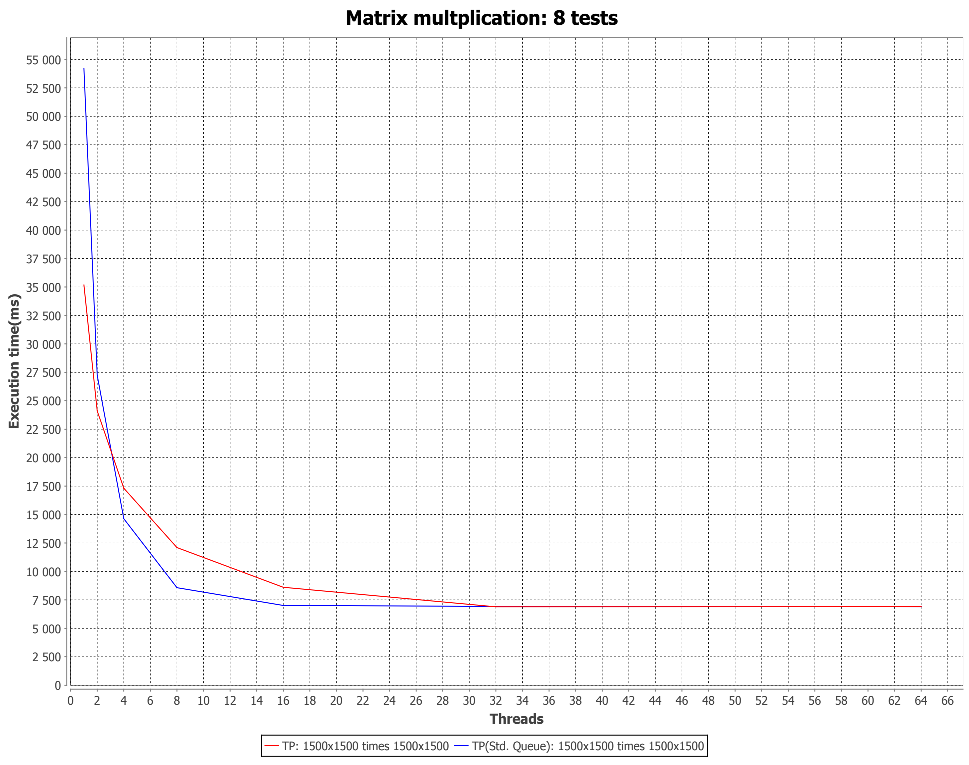


Figure 7: The difference in execution time for `ThreadPoolExecutor` with a standard *FIFO* queue and a `SynchronousQueue`.

The task size is another important aspect when using a `ThreadPoolExecutor`. Unlike the `CyclicBarrier` implementation the amount of tasks are not equal to the amount of threads due to the work queue. The task size here is controlled by the range of indices computed as seen in listing 7, the smaller range the more tasks are needed to compute the full solution and the smaller each task are. As seen in Figure 8, when the amount of computations/task is low and there are many tasks, the overhead for managing the tasks is too great and the execution time is long. There is also the aspect of uneven workload between threads due to the amount of tasks and the amount of processors to divide them among. This explains the peaks in the execution time that the `ThreadPoolExecutor` implementation exhibit for some task sizes. The work stealing design can be used to address this problem as explained in the next implementation, also see the fork/join Section 2.3.1.

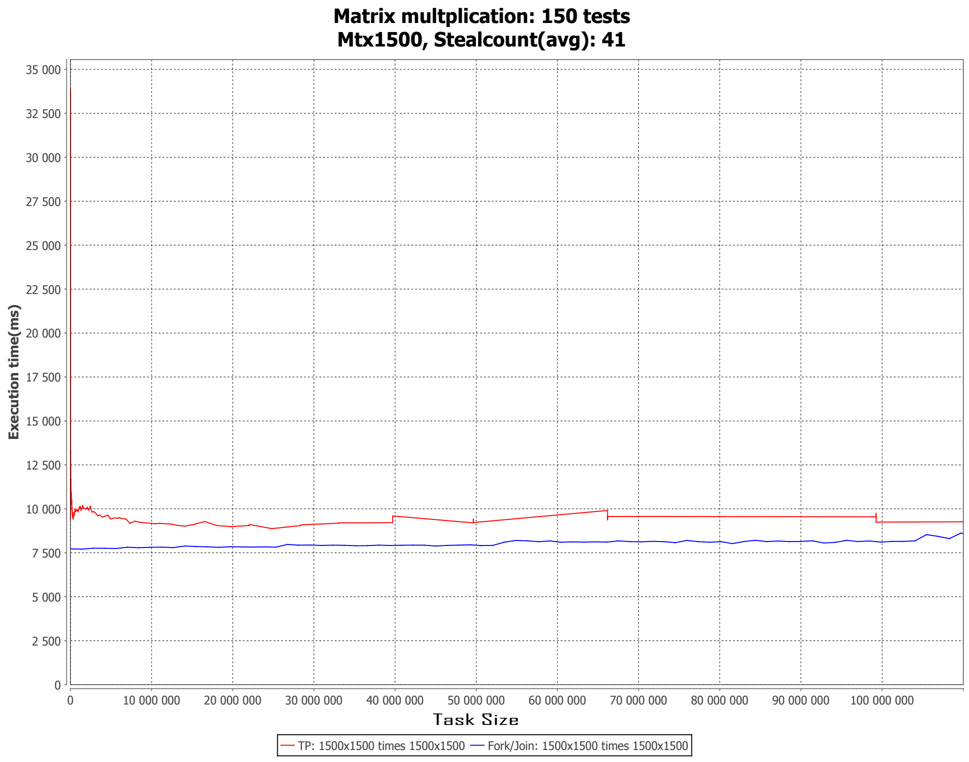


Figure 8: The difference in execution time for fork/join and `ThreadPoolExecutor` in relation to tasks size.

4.2.3 Fork/Join Implementation

When using the fork/join approach the solution is similar as with the `ThreadPoolExecutor`, since the fork/join version of the `ThreadPoolExecutor` class, the `ForkJoinPool` class, implements the same interfaces. Fork/join, however, is easier to use and reason about when designing the solution in a recursively manner, see Section 2.3.1. Assuming that a matrix computation task is represented as a `MtxTask` class and a matrix is represented as a `Matrix` class this can be written in pseudo code as seen in 8.

```
final Matrix mtxLhs
final Matrix mtxRhs
Matrix mtxProduct

/** class that stores which indexes of the result matrix to
   compute. Extends RecursiveAction of the F/J framework.
 */
class MtxTask extends RecursiveAction{
    @Constructor
    MtxTask (indexes to compute)
    ...

    @Override
    void compute(){
        if(Only one index to compute){
            Sequentially compute result of mtxProduct for the
               index stored in this MtxTask
        }
        else{
            split into parts: MtxTask Left(left half of indices)
               and MtxTask right(right half of indexes)
    //Compute parts in parallel
    fork( left.compute(), right.compute() )
        }
    }
}
```

Listing 8: Pseudo code for matrix multiplication implementation with the fork/join framework.

This pseudo code is based on the class `RecursiveAction` of the fork/join framework. Extending it allows for overriding the `compute()` method, which is called from the framework to solve a task. `Recursive-Action` is meant for computations that do not return a value (it in turn extends `ForkJoinTask<Void>`). That is suitable for this example, since all tasks write their results to individual indexes in the shared result matrix data structure, instead of returning an explicit value from `compute`. In the else statement of compute two new `MtxTask` instances are created, each holding half the index count. This is continued until the last recursion step where a task only hold one index which is computed since the `if` statement becomes true. If relating to the previous $8x8$ matrices, this would causes all

64 indexes to be computed by one task each.

If for example eight threads compute these tasks and there are a total of 64 tasks it is a good opportunity for work stealing. All threads have their own work queues which can then store eight tasks each. If one thread computes all its eight tasks while another thread only manages to compute two tasks, the first thread can steal a task from the others queue, instead of going idle and wasting processor time.

This approach has a problem: In reality this would not be efficient due to the overhead of creating and managing each task. The sequential threshold for a task should instead be related to the size of the matrix, since the bigger the matrix, the more computations is needed for that index. In this case (with an 8x8 matrix times an 8x8 matrix) each index in the resulting matrix requires 64 multiplications, which is too little and the overhead for managing the task will outweigh the gain. A better way is to compute the threshold according to the matrix size, for example three indexes per task as seen in listing 9. The sequential threshold is also discussed in Section 4.1.1

```
if(This task has less then 4 indexes to compute){
    Compute result of mtxProduct for the index stored in this task
}
```

Listing 9: Example of computing threshold.

This type of solution is more suited for bigger matrices where the computations can be divided into bigger tasks that outweigh the overhead. But it should still be plenty of tasks so that the work stealing method is efficient. Figure 8 shows how the execution time increases when the task size increase. The execution time also gets a lot harder to predict when tasks go up since there are a lot of different factors which can effect how much CPU time a specific tasks computation will get. At this stage, it gets a lot harder for the work stealing method to balance any uneven computation time between the threads.

The optimal configuration for thread pool size in this implementation is around the `ForkJoinPool` default setting which is set to the amount of available processors, as seen in Figure 9.

There is a compromise between how big and small a task should be depending on the overhead for managing the queues. There is also the fact that the main advantage with fork/join, the work-stealing, basically only gets into effect when threads are running out of work. This happens in the end of a computation, since this implementation has tasks that are equally large, it means that the longer a computation gets and the more coarse-grained it is, the less influence works-stealing has.

4.2.4 Comparison

The three implementations are compared considering their performance, safety and memory usage as well as how much programming effort is needed to implement them.

Performance and memory usage
Considering performance and scalability all implementations show good speed-ups when increasing the amount of threads on the target system. This can be seen in Figure 9, where it also can be seen that they scale about the same. The `ForkJoinPool` report some work stealing but it is less than a percent of the total work consumed and this problem is not a real *divide-and-conquer* problem, which the fork/join framework is optimized for. Merge sort, on the other hand, is and its implementation in Section 4.1.1 with the fork/join approach shows different performance advantages from the ones seen here. The `ForkJoinPool` uses a queue for each worker thread, compared to the `ThreadPoolExecutor`, where all threads are contending for the same queue. When the task sizes are tuned they both show good performance. The `CyclicBarrier` show good performance but it lacks much of the flexibility and safety which is seen in especially the fork/join framework but also in the Executors framework.

The fork/join approach shows some limitations when attempting to divide too big problems into many small tasks since the memory requirements get very high.

When creating worker threads and using the `CyclicBarrier` as synchronization all thread management is performed by the programmer and even though it shows good performance it is not recommend to use this approach. One reason for its good performance is that the tasks are completely independent of each other. Therefore each task is easily solved by each thread without any context switching due to lock contention and/or blocking due to waiting for subtasks (like the merge operations in merge sort, see Section 4.1). For big problems it would require a huge amount of threads to simulate the behavior of the fork/join framework [8]. If instead a higher threshold is used to keep the thread count down, performance will suffer due to big tasks that take too long and are switched out before they finish. Threads are also expensive to create and to be able to keep the thread creation overhead down the threads must continuously

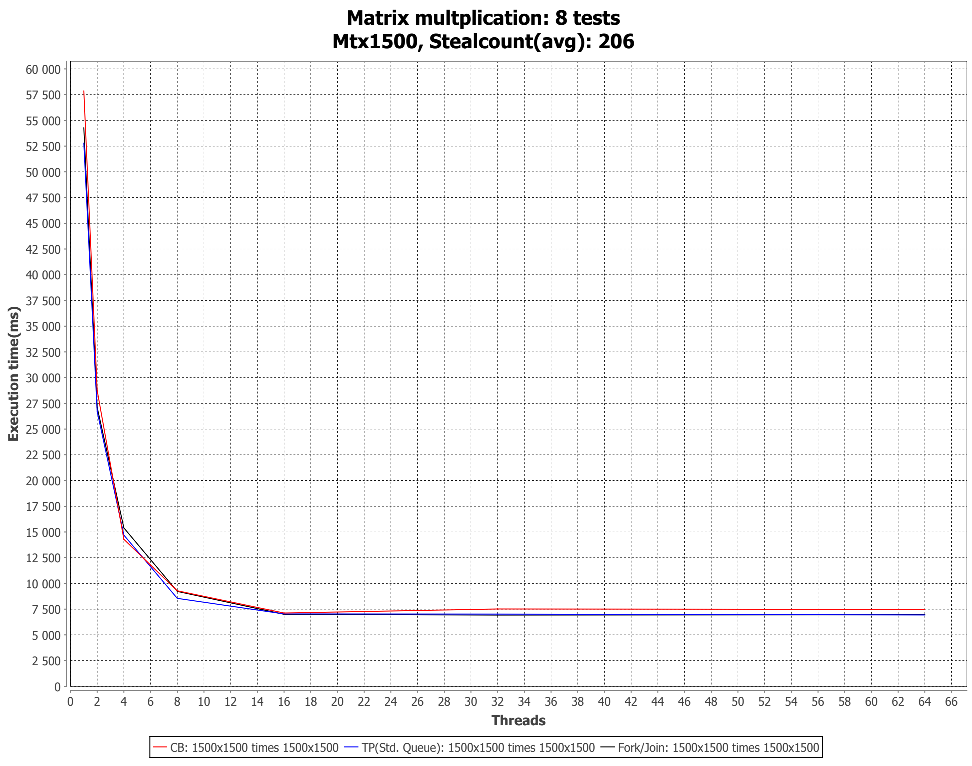


Figure 9: Scalability measured for fork/join, `ThreadPoolExecutor` and `CyclicBarrier` synchronized —implementations of the matrix multiplication algorithm.

be fed with work. To do this, some queuing system must be used and that is already implemented in the mentioned frameworks, which also are very configurable. It is for example possible to supply a thread factory and queue implementation to the `ThreadPoolExector`. If a direct hand off between the main thread and a worker thread is needed, without a queue, a `SynchronousQueue` can be used which might show very good performance increases in some cases[1][6]. In this case it resulted in slower scaling performance though, as explained in the `ThreadPoolExector` implementation (Section 4.2.2). When using this approach with the `ThreadPoolExector` it is possible to continuously feed the threads with work as they run, without the tasks having to wait in a queue.

Safety and programming effort

To implement the matrix multiplication using the `ForkJoinPool` of the fork/join framework is very straight forward, especially if the programmer is used to implement recursive algorithms. All problems are divided in the same way by using a threshold value which defines how many indexes are computed per task. This is easy with the fork/join framework by simply letting a task class extend the `RecursiveAction` class and override the `compute` method. It is for example easy to know when all computations are done by the `ForkJoinPool` worker threads by simply waiting for the first submitted task (by calling the `Future.get()` method of the submitted tasks future, see 2.2.3) since it recursively waits for all subtasks. This could be done with the `ThreadPoolExector` as well because the matrix multiplication does not have the `join` part of the fork/join approach, i.e. tasks do not wait for subtasks to join their results. This means that there is no risk of *thread deadlock starvation* if the tasks are submitted using an asynchronous method like `ThreadPoolExector.submit`, when using this recursive approach. If there are such dependencies the `ThreadPoolExecutor` must be carefully configured so there is no risk of all threads waiting forever when there are no more available threads in the pool. The `ForkJoinPool` does not have this risk at all since it is designed for tasks that wait for subtasks [8] but in this example some of its design advantage is lost since this is not a pure *divide-and-conquer* problem.

The `CyclicBarrier` approach is not very flexible and requires the most effort to implement since all worker threads have to be created and started with runnable tasks. It is normally safer and more flexible to let a framework handle the threads life cycles.

Conclusion

All the implementations show equal performance and scalability. It should be noted though, that if repeated computations where to be made, the executors and fork/join frameworks, in theory, will show much better overall performance then the `CyclicBarrier`. This advantage comes from the fact that, the frameworks have thread pools that can be kept alive between the computations, while the `CyclicBarrier` needs to restart all its threads, causing more overhead.

The safest and easiest implementation is done by using either; the `ForkJoinPool` and the `RecuriveTask` of the fork/join framework or the `ThreadPoolExecutor` with a standard *FIFO* queue, of the executors framework. The fork/join framework has some advantage due to the work stealing. The `ThreadPoolExecutor` approach has the advantage of being more flexible, with its queue implementation being exchangeable. This problem is not a *divide-and-conquer* problem, it lacks the specific *join* part of the fork/join design, and that causes it to lose some of the advantages when using the fork/join framework to solve it.

4.3 Minimum Spanning Tree

To test a not so straight forward parallelization problem, that can be tried out with different parallelization techniques, an implementation of Prim's algorithm has been tested. Prim's algorithm finds the minimum spanning tree in a weighted graph. A *minimum spanning tree* (MST) is a sub graph of a graph with the following properties: It is a tree, contains all nodes in the graph and has the minimal sum of edges. There can be several spanning trees in a graph (a sub graph that is a tree and contains all nodes) and there can be several *minimum* spanning trees in a graph. The two most common and well known algorithms for finding a MST in a graph are called *Kruskal's algorithm* and *Prim's algorithm*. In this example Prim's algorithm has been used because it is very well suited for concurrency [2].

With the MST algorithm, the performance of the `java.util.concurrent.ConcurrentLinkedQueue` and the `Collections.synchronizedList` have been compared and evaluated, also an implementation with a `synchronized` block instead of a synchronizing collection has been tested. The difference between these collections is that the `ConcurrentLinkedQueue` is implemented with a *non-blocking* algorithm for synchronization[21], while the other two (`synchronizedList` and implementation with `synchronized` block) uses *course grained locking* for synchronization, that locks the entire collection on any access. When many threads attempt to access the *course grained locking* collections, at the same time, only one gets through and the others have to wait. This should cause bad throughput when contention of the collection is high. The *non-blocking* technique, on the other hand, should allow for better throughput, since many threads can access the collection at the same time.

Prim's algorithm works in three steps. As an initialization it picks a node to start from then the following three steps are iterated:

1. Find the node that is closest to the current node.

2. When the closest has been found, the edge from the current node to this closest node is saved in a result list.

3. Mark the found node as used. Go back to 1. and continue until all

nodes have been visited.

A sequential Java implementation is listed in listing 10.

```java
public void prims() {
   int i, j, k = 0;
   int[] nearNode = new int[N];
   float[] minDist = new float[N];
   float min = 0;

   //Save the distance to all other nodes from node 0 as init
   minDist = Arrays.copyOfRange(weightMatrix[0], 0, weightMatrix.
       length);

   //Find all edges in the partial tree, N-1 edges in tree
   for (i = 0; i < N-1;i++) {
      //Step 1.
      //Find the shortest edge from node nearNode[k]
      //to any other node and save it in k
      min = Integer.MAX_VALUE;
      for (j = 1; j < N; j++) {
         if (0 < minDist[j] && minDist[j] < min)  {
            min = minDist[j];
 k = j;
         }
      }

    //Step 2.
    //Closest node is saved in result array
    T[i][0] = nearNode[k]; //From
    T[i][1] = k; //To
    minDist[k] = -1; //So it won't be taken again

    //Step 3.
    //Check if the next nearest node is on the new node
    //Else just keep going on the current one
    for (j = 1; j < N; j++) {
        if (weightMatrix[j][k] < minDist[j]) {
           minDist[j] = weightMatrix[j][k];
           nearNode[j] = k;
        }
    }
   }
}
```

Listing 10: Sequential Implementation of Prim's algorithm

Prim's algorithm might at first glance not be very straight forward to parallelize. The main loop can not be run concurrently since when a minimal distance to a node has been found it must be entered into the partial tree in a sequence, because that is what the partial tree contains, a sequence of edges. So, the content of the main loop must be parallelized instead of the whole loop somehow. Looking at Step 1 and the first loop inside the main loop, one can observe that the loop only writes values to the `min` variable and the `k` variable. This makes the loop possible to parallelize by dividing the loop into intervals and either synchronizing on the min and k element, or when done, having some kind of join operation on the subparts to obtain the correct value of `min` and `k`. Step 2 can not be parallelized as mentioned before because here the result is saved in a partial tree and the partial tree contains a sequence of edges in the graph which makes up the minimum spanning tree. After looking at the loop in Step 3, one soon realizes that each iteration of the loop is completely independent, so it can be divided into subtasks very easily. In this case no tests are needed to decide how many sub tasks the iterations will be divided into. The optimal solution is always when there are as many threads as processor cores since there is no waiting in the threads and the application will deadlock if there are more tasks than threads.

Four different methods have been implemented to test different Java concurrent utilities. The different implementations are constructed to test how the various lists and queue implementations, that uses two main different synchronization techniques, affects the execution time of the algorithm. Only the `ThreadPoolExecutor` from the executors framework has been used in the various methods. Previous implementation also used the fork/join framework, but Prim's algorithm is not a divide and conquer style algorithm, which is what fork/join framework is designed for. And since the main goal with this implementation is to test various synchronization techniques, the fork/join framework has been omitted in this case. The different methods are:

- Sequential approach where no concurrency exists, called `PrimsSequential`.
- `ThreadPoolExecutor` with `ConcurrentLinkedQueue` approach, called `PrimsThreadPool`.
- `ThreadPoolExecutor` with `SynchronizedList` approach, called `PrimsThreadPoolCollection`.
- Using an unsynchronized collection and a `synchronized` block in each loop iteration, called `ThreadPoolSync`.

4.3.1 ThreadPoolExecutor Implementation with ConcurrentLinkedQueue

The implementation with the `ConcurrentLinkedQueue` has the main difference from the sequential implementation that the loops in Step 1 and Step 3 from listing 10 are divided among several subtasks. The code for the first loop can be viewed in listing 11. The result of each subtask is saved in the `results` list and then the results of the subtasks are joined in a join operation that iterates over the results of the subtasks and compiles a result.

```
//Step 1
ConcurrentLinkedQueue<MinFinder> results = new
    ConcurrentLinkedQueue<MinFinder>();
//Find all edges in the partial tree
for (i = 0; i < N-1;i++) {
   //Find the shortest edge from node nearNode[k] to any other
       node and save it in k
   results.clear();
   latch = new CountDownLatch(nrTasks);
   int index = 1;
   for (int ii = 0; ii < nrTasks ; ii++) {
      MinFinder mf;
      if (ii == nrTasks - 1)
         mf = new graphalgorithms.PrimsThreadPool.MinFinder(index,
              index + sizeOfItr + rest);
      else
         mf = new graphalgorithms.PrimsThreadPool.MinFinder(index,
              index + sizeOfItr);

      threadPool.submit(mf);
      results.add(mf);

      index += sizeOfItr + 1;
   }
   k = join(results);
         .
         .
         .
```

Listing 11: Step 1 in the loop of the `ThreadPoolExecutor` with ConcurrentLinkedQueue implementation of Prims's Algorithm.

For each iteration of the inner loop an object `MinFinder` is created. `MinFinder` calculates the minimum distance from the current node in the graph to the nodes in a range between the indices provided in the constructor when the `MinFinder` is created. So, for each subtask the minimum distance is found and then the total minimum distance is found by iterating over the results from the subtasks. Step 2 still can not be parallelized since that is where the result is saved in the sequential minimum spanning tree. Step 3 can be parallelized in the same way as Step 1, but without a join operation since the results of each iteration is completely independent of each other. Step 3 will look like in listing 12.

```
latch = new CountDownLatch(nrTasks);
index = 1;
for (int ii = 0; ii < nrTasks ; ii++) {
   NodeUpdater updater;
   if (ii == nrTasks - 1)
      updater = new NodeUpdater(index, index + sizeOfItr + rest, k
          );
   else
      updater = new NodeUpdater(index, index + sizeOfItr, k);

   threadPool.submit(updater);
   index += sizeOfItr + 1;
}

try {
   latch.await();
} catch (InterruptedException ex) {
   ex.printStackTrace();
}
```

Listing 12: Step 3 in Prim's concurrent version

As can be viewed in listing 12 there is no need to join the results, so another class called `NodeUpdater` takes care of the update and then finishes.

4.3.2 `ThreadPoolExecutor` Implementation with SynchronizedList

The `ThreadPoolExecutor` implementation with `SynchronizedList` is exactly the same as in Section 4.3.1 except it uses a `SynchronizedList` instead of a `ConcurrentLinkedQueue`.

4.3.3 `ThreadPoolExecutor` Implementation with Synchronization Block

The `ThreadPoolExecutor` implementation with the synchronized block does not need a join operation in Step 1, but instead all the subtasks synchronize when updating the minimum distance to the next node. This is achieved with a synchronized block that can be viewed in listing 13.

```
if (0 < minDist[j] && minDist[j] < min)  {
   synchronized(minLock) {
      if (0 < minDist[j] && minDist[j] < min)  {
    min = (int)minDist[j];
    minIndex = j;
      }
    }
}
```
Listing 13: Synchronized block used in Prims's Algorithm.

However, the synchronization is not necessary to perform every time, since it is only needed when the value is supposed to be updated. That is why there are two if clauses doing the same thing. Instead of the join operation in the collection implementations a synchronization is performed in the `MinFinder` implementation.

4.3.4 Comparison

The `ThreadPoolExecutor` implementations with the `SynchronizedList` and with the `ConcurrentLinkedQueue` have almost the same performance if looking at the graphs in Figure 10 and Figure 11. Even though it is not visible from the graphs, on average, the `ConcurrentLinkedQueue` is a few milliseconds faster than the `SynchronizedList` on 8 cores with hyper-threading, and a little bit slower on the dual core machine. However, for this particular application it seems that the two perform about the same. The paper that the `ConcurrentLinkedQueue` is based on says that for more than 2 threads and a lot of concurrent access to the queue it is a lot more effective than a synchronized version [7]. On the dual core machine there are exactly two threads accessing the queue, and looking at the test, it can be seen that the `SynchronizedList` is slightly faster. On the 8 core machine with hyper-threading the `ConcurrentLinkedQueue` is a bit faster but not a lot. That is because each thread only accesses the queue once, so there is not a lot of concurrent access to it.

Scalability
The scalability for Prim's algorithm is quite good. Looking at the graphs it can be observed that the performance of the concurrent algorithms compared to the sequential is significantly increased when running on an 8 core machine with hyper-threading compared to running on a dual core machine. On the 8 core machine with multi-threading the increase in performance is roughly 2.4 and on the dual core it is roughly 1.3.

For an increase in threads and computational size, the `ThreadPoolExecutor` with `ConcurrentLinkedQueue` is the best choice, because it scales best for more concurrent access [7].

The Implementation with synchronized blocks is, as can be viewed

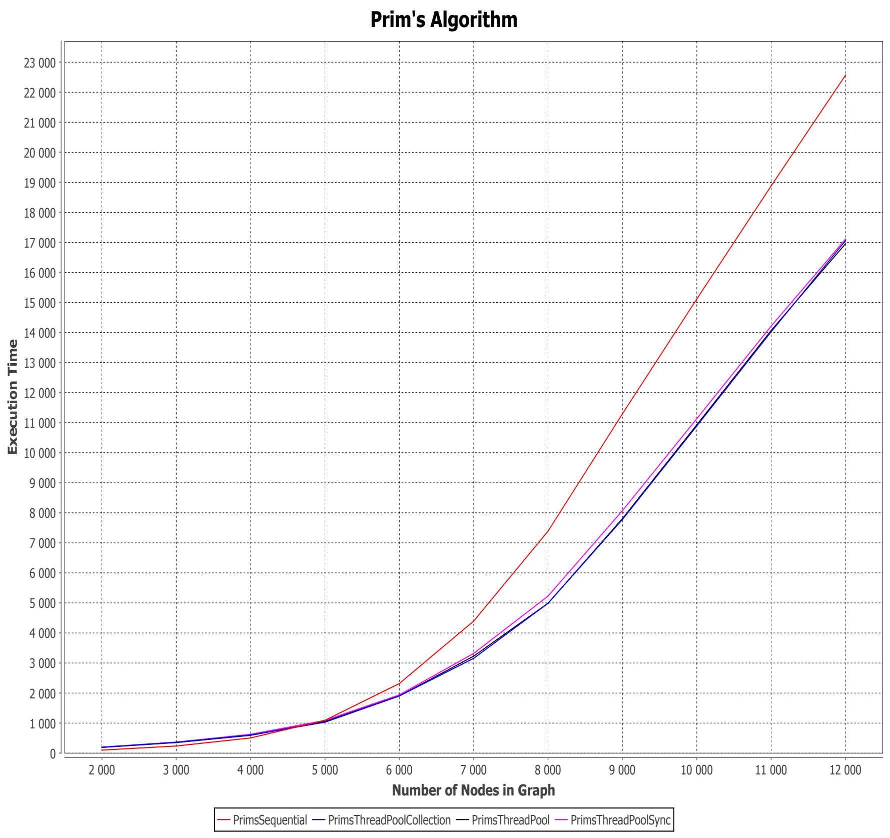


Figure 10: Comparison of Sequential, ConcurrentLinkedQueue, SynchronizedList and Synchronized block on a dual core machine.

in Figure 10 and in Figure 11, a bit slower on the dual core machine and a lot slower on the 8 core machine with multi-threading. This is due to the amount of synchronizations needed. With more threads and bigger computations more synchronizations are needed and therefore the synchronized implementation does not scale very well. So, in the sense of scalability it is the worst choice.

Memory

As for memory usage, the implementation with `ThreadPoolExecutor` requires a queue, but since the number of tasks are as many as the number of threads, and the thread count is low, this is not an issue. The two `ThreadPoolExecutor` implementations also save the result of each subtask in a `ConcurrentLinkedQueue` or a `SynchronizedList`. These two take up roughly the same amount of space and the maximum size of these are

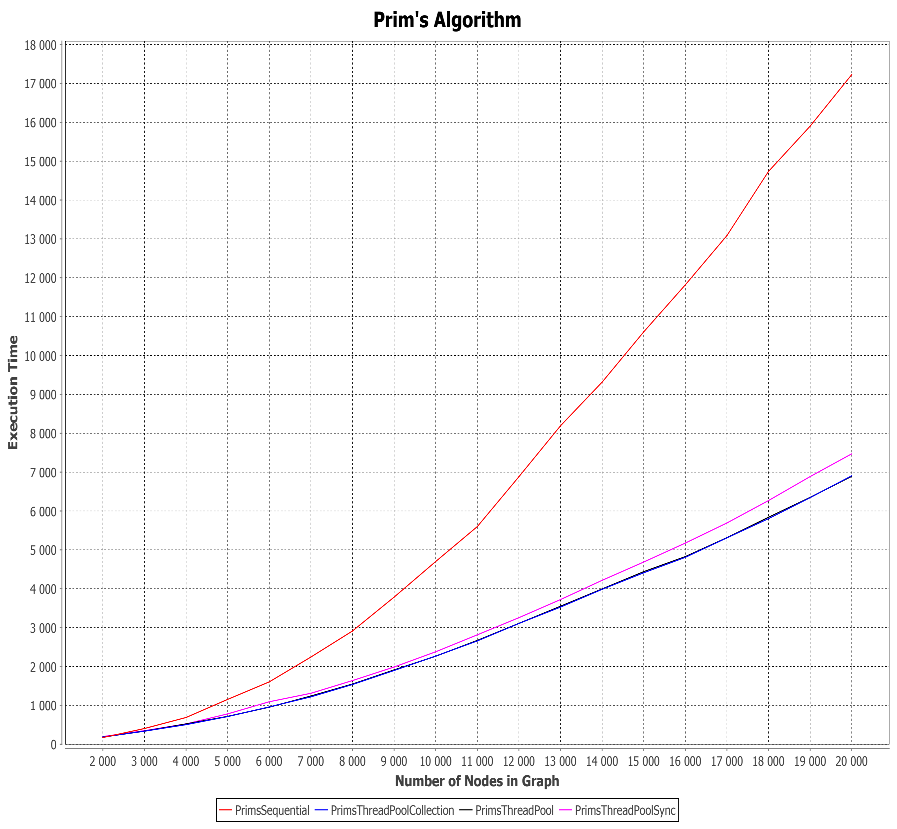


Figure 11: Comparison of Sequential, ConcurrentLinkedQueue, SynchronizedList and Synchronized block on an 8 core machine with hyper-threading.

as big as the number of nodes in the graph. So when calculations on big graphs are performed, a lot of additional memory is needed, compared to the sequential version and the version using synchronized blocks.

Programming Effort and Safety

Even though parallelizing Prim's algorithms is not as straight forward as for example the merge sort in 4.1 it is not very hard to do. The different concurrent implementations both have their pros and cons. The implementations with the `ConcurrentLinkedQueue` and `SynchronizedList` have the downside that they must perform a join operation, but still that is not much of a safety issue and is not very hard to implement. The collections handle all the thread safety which is a very good thing for the developer. The synchronized block on the other hand can be hard to get right. Even though it is not much data that need to be synchronized it is easy to implement it the wrong way and end up with race-conditions. The best thread safety is achieved when the programmer does not need to implement thread-safety manually.

Conclusion
As a conclusion it can be stated that a MST implementation is not the ultimate test for comparing a `ConcurrentLinkedQueue` and a `SynchronizedList`, but it most certainly is a very good algorithm to parallelize. The difference in execution time between the `ConcurrentLinkedQueue` and the `SynchronizedList` is very small. As for safety it is always good to use the concurrent version in a concurrent environment, but if one has a complete system, this test shows that it is not always feasible to change an entire system because of the low performance gain compared to the `synchronized` version.

4.4 Producer/Consumer based Graph Viewer

The primary purpose of this implementation is to compare some different collections, available in Java, by using them as a queue that is concurrently accessed by threads both reading and writing to it. The secondary purpose is to investigate the *Producer/Consumer* and *Java Monitor* —patterns, as well as using atomic operations through the `AtomicLong` class. This is done by implementing a *Producer/Consumer* based application that dynamically updates a graph, which is displayed on a `JFrame` window.

The producers perform calculations and submit data to the shared queue, which is read by the consumers who fetch the data and display it to the `JFrame` window, in the form of a graph. The queue implementation is based on an interface, so the differences in the queue implementations are invisible for the producers and consumers. The interface has methods for adding and getting an object to and from the display list, i.e. the shared queue, which is processed by the consumers for values to display on the graph. The interface can be seen in listing 14, where the `getDisplayObject()` method should block. Both the producers and consumers share an instance of an `IDisplayList` implementation. Various implementations can then handle the actual queuing of work in different ways.

```
public interface IDisplayList<E>{
    E getDisplayObject();
    void addDisplayObject(E obj);
}
```

Listing 14: Display queue interface

The IDisplayList interface has been implemented using two different collections as the underlying work queue implementation:

- Standard non thread safe LinkedList, needs to be manually synchronized and handle any blocking if the queue is empty.

- A LinkedBlockingQueue, from the concurrency package, supporting access from several threads as well as blocking if the queue is empty.

To keep track of the amount of tasks processed by the producer and consumer instances, an AtomicLong, which is declared as final (to guarantee visibility, see Section 2.6.1 about general design) is used. When sharing a final instance of an atomic number between threads it can be used atomically and count common operations without any extra synchronization, this would not be safe when using an int and the ++ operation. To generate random work in the producer threads the ThreadLocalRandom class is used to minimize overhead and contention between the worker threads when generating random numbers.

A simple example, showing the idea of this design, is shown in listing 15. In this listing, the queue implementation is shared between producers and consumers tasks. For this design to be thread safe, the implementation o f the IDisplayList interface must guarantee thread safety. An arbitrary number of producer and consumer tasks can then be run safely in several threads. The AtomicLong.incrementAndGet() operation prevents ans, regarding the work statistics, when several threads are producing or consuming work concurrently.

```
//*Producers and Consumers that share a work queue*

class GraphProducer implements Runnable{
    private final IDisplayList<Work> queue = ...;
    private final AtomicLong workProduced = ...;

    public void run(){
        while(true){
            queue.addDisplayObject(new Work());
            workProduced.incrementAndGet()
        }
    }
}
...
class GraphConsumer implements Runnable{
    private final IDisplayList<Work> queue = ...;
    private final AtomicLong workConsumed = ...;

    final public void run(){
        while(true){
            Work producedWork = queue.getDisplayObject();
            showOnGraph(producedWork);
            workConsumed.incrementAndGet() ;
        }
    }
}
```

Listing 15: Display queue interface

In the following subsections the two different queue implementations of the `IDisplayList` interface, used as seen in listing 15, are explained. Note that the call, `queue.getDisplayObject()`, in the `GraphConsumer` class, must block, otherwise there will be a busy waiting loop polling the status of the queue. This is straight forward when using the `LinkedBlockingQueue` as the underlying queue implementation, since it has a blocking operation that does not return until the queue is non-empty. The `synchonized LinkedList`, on the other hand, does not have such a blocking operation, so it must be implemented manually.

4.4.1 `Synchonized LinkedList` Implementation

As stated earlier, the implementation of the `IDisplayList` interface must guarantee thread safety. When using a non thread safe `LinkedList`, as the work queue implementation, this must be achieved manually in some way. One way is to use the static method `Collections.synchronizedList(new LinkedList<V>())`, which returns a wrapper around the linked list, where all its methods are synchronized using a single lock, see Section 2.2.2 about collections. This would solve the synchronization problem, but not the blocking problem mentioned earlier. Since the queue could be empty for a long period of time, the `IDisplayList.getDisplayObject()` method implementation should block if the queue is empty, until a producers adds work to the queue.

As the `LinkedList` has no blocking operations, all *get* operation return immediately with either failure or success. To handle this, two simple approaches can be made:

- Poll the queue for new content added by the producers, until it is updated.
- Sleep and check the queue status in regular intervals.

Both these approaches have major drawbacks. The first idea of polling might lead to overuse of the CPU and the second approach might led to oversleeping and bad responsiveness. A better way is to use the *Java Monitor Pattern*, to wait on an object and let any thread that modifies the queue notify the waiting thread, which then immediately will be notified of the queue modification. This has been implemented as seen in listing 16. When using this method, the `synchronized` blocks must be added manually, so the wrapper returned by `Collections.synchronizedList(new LinkedList<V>())` (which adds `synchronized` blocks internally) has not been used. The `IDisplayList.addDisplayObject` implementation is synchronized, adds the new object to the list, representing the work queue, and then calls `queue.notify()`. If any consumer threads are waiting, in the `queue.wait(WAIT_TIMEOUT)` operation, since the queue was empty, it will automatically be notified. The method `queue.notifyAlll()` could also be used but it is not necessary in this example, since there is only one condition predicate(if the queue is empty) and only one thread can be notified at a time (max one thread is first in line to use `getDisplayObject()`).

```
public void addDisplayObject(V obj){
    synchronized (_queue) {
            _queue.add(obj);
            _queue.notify();
    }
}
...
public V getDisplayObject() {
    synchronized (_queue) {
         //If empty block for some time and wait for notification
         if(_queue.isEmpty()){
             try {
                 _queue.wait(WAIT_TIMEOUT);
             } catch (InterruptedException ...) {...}
         }
         //Check reason for queue.wait to unblock
         //Could be timeout or new queue content
         if(!_queue.isEmpty())
             dispObj = _queue.remove(queue.size() - 1);
    }
}
```

Listing 16: Implementation with synchronized list

4.4.2 `LinkedBlockingQueue` Implementation

The `LinkedBlockingQueue` use sophisticated internal *fine grained locking* which should give it better concurrent performance than the `synchronized LinkedList`, since it allows several threads to access the list at the same time. The `synchronized LinkedList` was implemented with a single lock for the entire collection, or `course grained locking`, meaning that only one thread can access it at any given time.

With the `LinkedBlockingQueue` it is possible to block when the queue is empty, allowing for good responsiveness when the queue is modified, without directly using the *Java Monitor Pattern*. This is very helpful since, as noted in Section 2.6.2, it is not recommend to implement the *Java Monitor Pattern* directly, as done in the previous section, due to the safety risks. This approach makes the implementation of the `IDisplayList` interface very simple and safe, as seen in listing 17.

```
public void addDisplayObject(V obj){
    public void addDisplayObject(V obj) {
        queue.push(obj);
    }
}
...
public V getDisplayObject() {
    V val = null;
    try {
        val = queue.pollLast(TIMEO, TimeUnit.MILLISECONDS);
    } catch (InterruptedException ...) {...}
    return val;
}
```

Listing 17: Implementation with synchronized list

4.4.3 Comparison

The implementations are compared regarding scalability, programming effort and safety.

Scalability

Initially not much performance could be gained when using the `Linked-BlockingQueue`, even though many producers accessed the queue at the same time as an equal amount of consumers accessed the other side of the queue. The problem seems to be to that the contention of the shared queue was not high enough for the `LinkedBlockingQueue` to show any big concurrent improvements. It also seems that objects had to be allocated at such a high rate that garbage collection and other factors made performance advantage over the synchronized linked list impossible to notice. But when minimizing object allocation and letting producer workers add data to the shared queue at a very high rate the performance advantages with the `LinkedBlockingQueue` became visible, as seen in Figure 12.

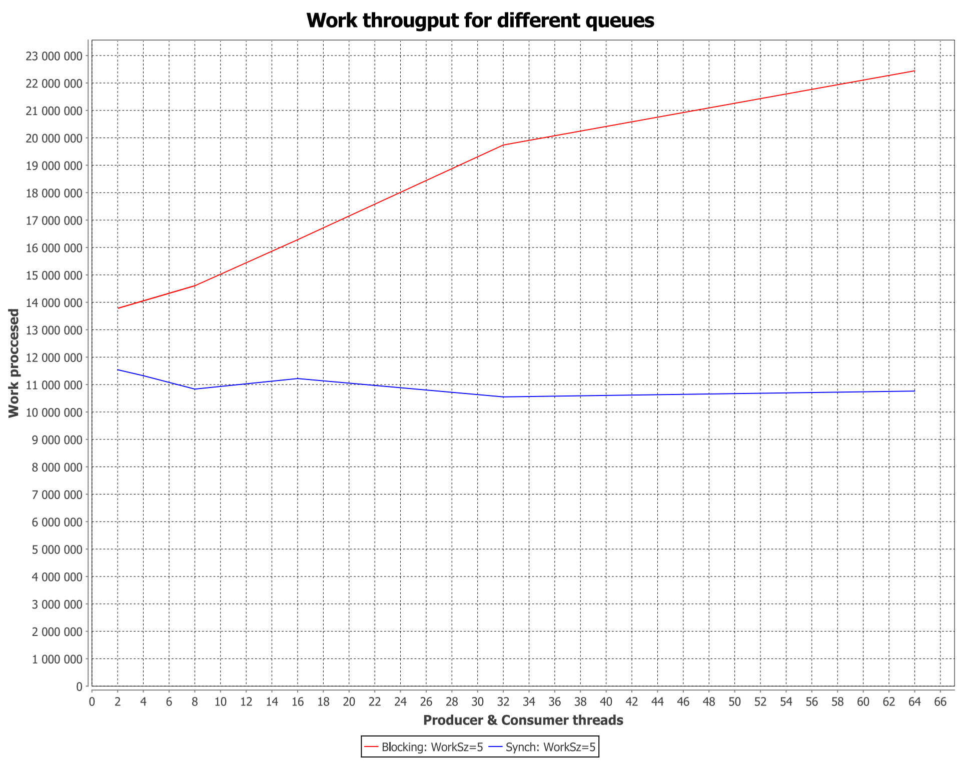


Figure 12: Comparison of LinkedBlockingQueue and LinkedList on a 8 core machine with hyper-threading.

Due to the shared lock of all write and read operations when using the synchronized `LinkedList`, it performs poorly when the list lock contention increase. This can be seen in Figure 12 as the amount of processed work can not keep up with the `LinkedBlockingDeQue` implementation.

Programming Effort and Safety

To use a `LinkedList` with the `synchronized` blocks or the `Collections.synchronizedList(new LinkedList<V>())` together with `notify()` and `wait()` methods is not recommended. It is much more error prone and complicated to implement than simply using a blocking queue from the `java.util.concurrent` package. Even more effort is required to develop an implementation that has similar performance to the *API* queues and is even more error prone. Therefore it is recommended to use a queue implementation from the `concurrent` package.

Conclusion

It was surprising how much tweaking was needed to see any real performance gain from using the `LinkedBlockingQueue`. However, the safety issues alone are argument enough to prefer it over the synchronized counterpart and if using it correctly, it also performs better.

4.5 Experiment Results

The result of the experiment part is an overview of what methods, frameworks, collections and other tools that are suited for which problems and their pros and cons.

4.5.1 Performance and Scalability

Comparing the four implementations in Chapter 4, it can be seen that they scale very differently. Some solutions are better for certain problem sizes, some are better on more cores and some are better on fewer. A lot of the problems scale well to a certain level, but *some parts of a program just is not concurrent*.

- The merge part of the merge sort algorithm is not concurrent (Section 4.1).
- The step where a new edge is saved to the minimum spanning tree in the minimum spanning tree algorithm is not concurrent (Section 4.3).
- The display of the graph in the graph viewer is not concurrent (Section 4.4).
- The only algorithm that is completely concurrent is the matrix multiplication and that is a very specific problem (Section 4.2).

4.5.2 Fork/Join Framework and `ThreadPoolExecutor`

The fork/join library is very good for a specific type of problems: recursive divide and conquer problems and in most other cases the `ThreadPoolExecutor` should be used. The most important usage of the fork/join framework is to be able to program secure divide and conquer algorithms with no risk of deadlock. The merge sort algorithm in Section 4.1 scales very well with the fork/join implementation, because algorithms like merge sort is exactly what the fork/join library is designed for. However, due to the properties of the merge sort algorithm the scalability is not linear. The matrix multiplication is not a typical fork/join problem, the matrix multiplication does not *join* like the merge sort does and therefore it can be solved by different means and still achieve about the same performance with the same programming effort. Though, there is no loss in performance if using the fork/join approach with the matrix multiplication. The graph viewer in Section 4.4 is not a suitable problem for the fork/join implementation and the minimum spanning tree problem in Section 4.3 is not either, since they can not be implemented efficiently as recursive algorithms.

4.5.3 Context Switching vs. Work Stealing

The graphs in Figure 6 and in Figure 5, pages 35 and 34, shows that the `ThreadPoolExecutor` scales almost as good as the fork/join approach when solving the merge sort algorithm. Here the work stealing is responsible for the advantage of the fork/join compared to the `ThreadPoolExecutor`. The `ThreadPoolExecutor` uses context switching to balance calculations, which works on the platform used (8 cores with hyper-threading). If the number of cores would increase the advantage of using the fork/join and work stealing would be more visible, because the overhead of the context switching would increase.

4.5.4 Memory

A downside with the fork/join library and other frameworks/methods that use a lot of tasks has been discovered; they use a lot of memory. Since a new object is allocated for every new task there can be several thousand new objects created while running the calculation. The memory consumption can however be kept down by not creating more tasks than needed.

4.5.5 Collections

In the implementations of Prim's algorithm (Section 4.3) and in the graph viewer (Section 4.4) different collections have been researched and tested. How good the collections scale varies a lot from application to application. It is of course in every case recommended to use the concurrent implementations of all collections since they are *developed to be concurrent while the synchronized collections are developed to be thread-safe*.

Prim's algorithm implementation shows almost no difference at all between concurrent and synchronized collections, only a few milliseconds. The graph viewer showed significant improvements with the concurrent collection case, but only after a lot of tweaking. It was tweaked to keep object allocation down so that the queue could be accessed often enough to show the benefits of the concurrent collection. The result from these tests addresses the issue with the concurrent collections and when they are actually needed. They are in many cases more secure but if there are not a lot of threads accessing a collection, refactoring an entire system too change their collections to concurrent ones is not worthwhile. However when designing a new system the concurrent versions are *always* preferable.

Collections like `ConcurrentHashMap` is very useful, but some operations like the `size` operation is slower than in the synchronized counterpart and the values returned is not guaranteed to be up to date. The concurrent collections might not always outperform the synchronized collections as much as expected as seen in implementations of *producer-consumer* graph viewer and Prim's algorithm. There are other issues with the synchronized collections, the individual operations are thread safe but that does not mean that compound operations are safe (for example iterating over a synchronized collection) and the programmer can not stop caring about concurrency. A full list of concurrent and synchronized collections and further explanations can be seen under Section 2.2.2.

4.5.6 Atomic Variables

When sharing a counter or using primitive operations that need to be synchronized an atomic variable like `AtomicLong` can be used. This has been implemented and tested in the *producer-consumer* implementation in Section 4.4, it is also used in the fork/join framework to keep track of the amount of work stolen between threads. It allows for basic increment and compare and get operations in a single instruction, which for example means that several threads with a shared up to date reference of such a class safely can increment it with minimal performance loss.

5 Threat Evaluation and Weapons Allocation Component

The Giraffe AMB software system consists of several software components that are responsible for different parts of the system. *Threat Evaluation and Weapons Allocation (TEWA)* is a component that handles threat evaluation of air units, weapons allocation and bookkeeping of designations in the system. The *TEWA* algorithm will periodically retrieve data of the local air picture and data about available firing units and defended assets. A threat picture is created and then it is used to calculate an optimized engagement plan. The result of the calculations will be presented to an operator, so engagements may be initiated. The algorithm can also be instructed to select engagements automatically. When an engagement between an air unit and a firing unit has been chosen, either by the operator or automatically by the *TEWA* algorithm, the firing unit and air unit is put in an engagement and sent to the rest of the system. The *TEWA* algorithm can manage firing control over locally connected firing units, including hold and cease fire commands. The *TEWA* algorithm also calculates the availability of firing units based on their reported status. The result of the algorithm can be viewed in Figure 13 where the red lines represent an engagement between a firing unit and an air unit.

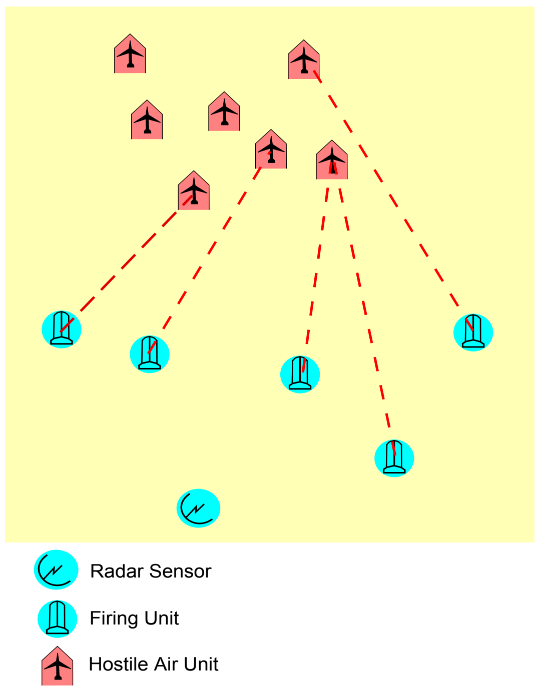


Figure 13: A scenario from the simulation of the TEWA system. The light blue firing units are engaged to the red hostile airplanes, this is visualized with the red lines.

The *TEWA* algorithm is performed in four steps:

- First, all information is updated. All air units are gathered from other components in the system and the status and availability of all firing units and defended assets are updated.

- Second, a protection of friends step is performed. It makes sure that a hostile engagement or command is not put on an air unit that is a friend.
- Third, a threat evaluation is performed based on the defended assets and air units.
- Fourth, a suggestion of how weapons would be assigned in a possible counter attack is proposed, or the *TEWA* algorithm automatically assigns firing units to air units.

Communication with other parts of the system is handled through *Common Object Request Broker Architecture (CORBA)*. The *CORBA* approach enables the possibility for other components in the system to communicate with the *TEWA* component at any time throughout execution. To be able to answer requests and perform actions right away several threads are continuously running, and are ready to process the request as soon as another component tries to communicate with the *TEWA* component. This means that at any time a thread can receive an event that may modify internal data structures. For example a hold fire command on an air unit can be received while allocating weapons and suddenly a certain air unit must be excluded from the weapons allocation.

5.1 Sequential Implementation

The currently used implementation of the *TEWA* in the Giraffe AMB is sequential. To handle direct manipulation of internal data structures from several *CORBA* threads, all access to internal data structures have been synchronized using the `synchronized` keyword. The synchronization is performed in several layers to make sure that the access to data structures will not be done concurrently.

5.1.1 Problem

To gain linear performance increase when the number of processor cores increases every part of a system must be made concurrent. In the sequential version of *TEWA* the only concurrent parts are the *CORBA* threads. In this case they are not performance intense at all which implies that when the number of processor cores increase, the performance of the *TEWA* remains constant. To keep the data structures from being corrupted by several threads reading and writing the same data, synchronization in several layers has been performed. Since the sequential *TEWA* implementation has several authors and has been rewritten a few times, a lot of the synchronization done is unnecessary and the effect is a huge overhead time for unnecessary synchronization. In conclusion the two main problems with this implementation are that it does not scale to a multi-core system and it has a lot of unnecessary synchronization to protect the data from being corrupted by *CORBA* threads.

5.2 Parallelization

To resolve the issues with the sequential *TEWA* implementation a concurrent solution has been developed in this thesis. The concurrent solution resolves the problems by running the most performance critical parts of the system concurrently, removing all old and unnecessary synchronization and by handling the *CORBA* threads in a separate event handler.

5.2.1 Method

The parallelization of the *TEWA* component has been preformed with the intention of making the locking of the code more fine grained than before. That means moving high level locks to an as low level as possible and to use the tools from the Java concurrency library when-ever needed. The more fine grained the locking is in the implementation, the better it scales to a multi-core system. In a sense, contradictory, very high level locking has been introduced as well; but this only intended to prevent unimportant events from slowing down the critical main execution, by exposing low level locks to unnecessary lock-contention. Such events could also be dangerous for any algorithms that require that specific data-structures are not updated at certain times, more about that in Chapter 6. The parallelization has been preformed in a few steps: *static analysis*, *software profiling*, *thread profiling*, *debugging*, *library development* and *implementation*.

Static Analysis
The *static analysis* has been performed by generating call graphs and UML diagrams with the applications *doxygen* and *Graphviz*, these are also discussed in Section 3.2. The call graphs and UML diagrams have been very valuable when investigating where synchronization is needed and where it is unnecessary. They can be used to learn which functions modify which data structures and which functions are called from several threads. The result of a completed static analysis may result in code that has no unnecessary synchronization and no unused code, it is also a good way to

review the code in general and a lot of bugs can be found this way.

Software Profiling
To find the most performance critical sections of the code, a software profiler called *Visual VM* has been used, for more information see 3.2. *Visual VM* was run locally on a dual core machine in a simulation environment. After the profiling had been preformed the most invoked methods and most performance intense parts, with a high percentage *self time*, of the software was found to be, as excepted, in loops. The two most performance intense steps of the *TEWA* algorithm were found to be the threat evaluation and the weapons allocation step, which are the third and forth steps of the main loop. In the threat evaluation it was the loop that iterates over all defended assets for each air unit to determine a threat value for each air unit. The most CPU was used by the weapons allocation, more specifically the part that calculates which weapons to assign to what air unit.

Thread Profiling
Before removing the old means of synchronization the software was profiled. The profiling was run with the CPU intensive loops parallelized, running the loops on 16 concurrent threads. This was done to determine how much better the new modern synchronization techniques are in comparison to the old synchronization. Thread profiling was also done after the implementation of the new synchronization to assure that the threads do not use unnecessary locks.

Debugging
To determine the exact behavior of an application can be hard, especially in this case, where running *CORBA* threads can invoke methods that modify data structures at any time. To find out how these *CORBA* threads behave we have used the debugger included with *Eclipse IDE*. It was run on the actual simulation system for the *Giraffe AMB* software. This made it possible to test the behavior of all the methods that are invoked remotely by other components. When each methods behavior was decided, they were all classified into different type of events, more on how they are classified can be found in Section 6.2.

Framework Development
Based on the result from the profiling, most of the execution time was spent in loops computing performance critical tasks, so functionality to handle loop concurrency was developed. The *CORBA* threads must be

handled somehow, this is achieved by separating them from the rest of the application and handling them in a more controlled manner. We realized that whenever a *CORBA* thread received an event the internal state of the *TEWA* component could be changed, so functionality to handle the different *CORBA* threads and their events were developed. This soon evolved into a framework that we call *CELP*, for more information see Chapter 6.

Implementation Method
The implementation and parallelization of the system was performed in several steps.

- Exchange all concurrently accessed collections with new modern collection from the `concurrent` package.
- Exchange all concurrently accessed primitive types with their equivalent atomic wrappers.
- Parallelize the most performance intense loops with the *CELP* framework.
- Move all the *CORBA* thread handling by creating events (by extending `celp.Event<E>`), which are then added to the internal `EventHandler` in the `CriticalTaskExecutor` from the *CELP* framework.
- Prioritize all the different events received in the *CORBA* threads.
- Identify critical tasks.
- Remove all old synchronization with the `synchronized` keyword.
- Identify code that can use the fork/join framework and parallelize that code.

5.2.2 Exchanging Collections

The first step was to exchange all the collections that can be accessed concurrently with their concurrent counterpart and remove the old synchronization. All the get and set operations have the exact same behavior as before but the iterations over the collections vary a bit. As described in Section 2.2.2 the concurrent collections do not guarantee to mirror all the updates done to the collection immediately while iterating over it. After carefully examine the old code we found that this was not an issue since none of the collections that were iterated could gain or loose elements during an iteration. The collections we have used are: `ConcurrentHashMap`, `ConcurrentLinkedQueue`, `CopyOnWriteArrayList`, `LinkedBlockingQueue` and `PriorityBlockingQueue`. We will here address the ones that are used the most, for more information on the other collections, see Section 2.2.2.

ConcurrentHashMap

The most valuable collection has been the `ConcurrentHashMap`, since functionality that took many lines of code in the old implementation, to achieve thread safety only takes one line with the `ConcurrentHashMap`. An example of how the `ConcurrentHashMap` has replaced the non thread-safe `HashMap` can be viewed in listing 18 and in listing 19. In the old *TEWA* component there was a higher level lock that locked the function in listing 18 but since we want to move the locking dependency from the programmer to the internal collections, that locking needed to be removed in the concurrent version. A higher level lock also decreases the lock granularity and has the effect that only one thread at a time can run the method. Listing 18 is not thread safe on its own so it needed to be changed, this was achieved by exchanging the `HashMap` by `ConcurrentHashMap` and some minor changes in the implementation.

```java
private final Map<Short, Integer> _overkillCount =
            new HashMap<Short, Integer>();

public int getOverkillCount(short atTn) {
    if (atTn == 0)
        return 0;
    Integer i = _overkillCount.get(atTn);
    if (i == null)
        return 0;
    return i;
}

public void increaseOverkillCount(short atTn) {
    if (atTn != 0) {
        int i = getOverkillCount(atTn);
        _overkillCount.put(atTn, i + 1);
    }
}
```

Listing 18: `HashMap` used in the old sequential *TEWA*. This function increases a variable called overKillCount it contains information about how many firing units that are assigned to a certain air unit.

The difference between the two implementations is the putIfAbsent method, which is available in the ConcurrentHashMap, and the use of AtomicInteger instead of Short. putIfAbsent is atomic which means that only one thread can execute it at a time, it returns the old value if a previous mapping for the current key atTn was found, otherwise it returns null. Since the inserted AtomicInteger is one by default it will be incremented to a correct value if putIfAbsent is called by a second thread, where putIfAbsent returns *that* AtoimicInteger resulting in the statement if(oldKillCount != null) being true. Because it is atomic, the reference returned can be guaranteed to be valid. The returned AtomicInteger is an atomic wrapper for an integer which means that only one thread can update it at a time, so the increment function is always valid. The old implementation does not use atomic variables and therefore they can not guarantee that the value is correctly updated if several threads are running the function concurrently.

```
private final ConcurrentHashMap<...> _overkillCounts =
    new ConcurrentHashMap<Short, AtomicInteger>();

/**
 * Increase the overkill count thread safe.
 * @param atTn
 */
public void increaseOverkillCount(short atTn) {
    if (atTn != 0) {
        AtomicInteger overkillCount = new AtomicInteger(1);
        AtomicInteger oldKillCount =
            _overkillCounts.putIfAbsent(atTn, overkillCount);
        if(oldKillCount != null)
            oldKillCount.incrementAndGet();
        }
}
```

Listing 19: ConcurrentHashMap used in the *TEWA* parallelization. This function increases a variable called overKillCount it contains information about how many firing units that are assigned to a certain air unit.

CopyOnWriteArrayList

Another valuable collection is the CopyOnWriteArrayList which is very useful when using a collection that has a lot of reads and only a few writes, for more information see Section 2.2.2. During the weapons allocation step in the *TEWA* a value that prioritizes the different engagements between a firing unit and an air unit is calculated. In the old implementation this value is calculated in two nested loops sequentially, these loops are illustrated in listing 21. To be able to do it concurrently the loops have been substituted for a concurrent loop from the *CELP* framework. In listing 20, the `compute(Integer loopIndex)` is run as many times as the nested loops in listing 21, *numberOfFiringUnits* * *numberOfAirUnits* times. Each iteration gets a value from the `CopyOnWriteArrayList` with firing units and one from the `CopyOnWriteArrayList` containing the air units. Usually no removals or additions to the collections are performed, but they may occur. This makes the handling optimal for the `CopyOnWriteArrayList`, see listing 20. This is not possible with a regular list implementation like `LinkedList` since it is not thread safe, and the `ConcurrentModificationException` can be thrown.

```
@Override
public void compute(Integer loopIndex) {
    // Get the two indices for current AirTrack and current
        FiringUnit from the loop index
    // which goes over all indexes.
    int airTrackIndex = loopIndex / nrOfFiringUnits;
    int firingUnitIndex = loopIndex % nrOfFiringUnits;

    CopyOnWriteArrayList<AirTrack> _ats;
    CopyOnWriteArrayList<FiringUnit> _fus;

    AirTrack at = _ats.get(airTrackIndex);
    FiringUnit fu = _fus.get(firingUnitIndex);
    .
    .
    .
```

Listing 20: Used to calculate a constant that determines the priorities between an air unit and a firing unit. New thread safe implementation using the *CELP* framework.

```
LinkedList<AirTrack> _ats;
LinkedList<FiringUnit> _fus;

for (AirTrack at : _ats)
      for (FiringUnit fu : _fus)
        .
        .
        .
```

Listing 21: Two nested loops used to calculate a constant that determines the priorities between an air unit and a firing unit. Old non thread safe implementation.

5.2.3 Exchanging Primitive Types for Atomic Wrappers

Instead of using synchronized blocks when updating primitive types, atomic variables can be used, see Section 2.2.3 for more information. All operations on atomic variables are executed in a single instruction and therefore require no locking. This can be very useful for example in concurrent loops with a shared counter, or when a class need to be updated concurrently and contains a shared variable, or when running a concurrent loop on a shared condition. An example how this has been used in the parallelization can be viewed in listing 22, the previous non-atomic version can be viewed in listing 23. It is an example from one of the containers in the *TEWA* that hold information on at what time the container was last updated. If several threads are updating the container at the same time, the access to that variable need to be thread safe and that is why using an atomic variable is a good approach.

```
private final AtomicLong _lastUpdate = new AtomicLong(0);
...
public boolean addDesignation(DesignationData designationData,
        T_Designation_Command command,
        T_Engagement_Status status,
        DesignationType type) {
    ...
    _lastUpdate.set(System.currentTimeMillis());
}
```

Listing 22: When a container is updated the time need to be saved by the application this is done in a local class variable. In the concurrent implementation that variable is atomic and therefore thread safe.

```
private final long _lastUpdate = 0;
...
public boolean addDesignation(DesignationData designationData,
        T_Designation_Command command,
        T_Engagement_Status status,
        DesignationType type) {
    ...
    synchronized (_designations) {
        _lastUpdate = System.currentTimeMillis();
    }
}
```

Listing 23: When a container is updated the time need to be saved by the application this is done in a local class variable. In the old implementation the variable was a primitive type and therefore had to be synchronized.

5.2.4 Parallelizing Loops

The three most performance intense hot spots in the code are in three loops. The first is in the main loop in the threat evaluation, the second is in the main loop in the weapons allocation and the third is an optimize function where all possible engagements between fire units and air units are optimized. To parallelize the two first loops the *CELP* framework has been used, for the last one the fork/join library from Java 7 has been used.

Threat Evaluation Loop
The first loop in the threat evaluation in its sequential form works as follows:

1. Collect all air tracks and Defended Assets.
2. Order calculation of threat values.
3. Send threat levels to the rest of the system.

The basic implementation looks like in listing 24. It basically calculates the threat value for each air track based on all defended assets.

```
public void evaluate() {
    //Air units
    AirTrack[] atArr = _airTracks.getAirTracks();
    // Defended assets
    DefendedAsset[] defendedAssets =
        _defendedAssets.getDefendedAssets();
    // Collection to save threat values in
    List<T_Lap_Threat_Value_Data> tvDataList =
        new ArrayList<T_Lap_Threat_Value_Data>();

    //Iterate over all air units
    for (AirTrack at : atArr) {
        //Calculate the threat value for the air unit
        ThreatValue currentThreatValue =
            evaluate(at, defendedAssets);
        at.setThreatValue(currentThreatValue);

        //Filter the treat value
        double currentThreat = currentThreatValue.getValue();
        double filteredThreat = filterThreat(at);

        //Calculate the threat level
        T_Threat_Level oldLevel = at.getThreatLevel().getLevel();
        T_Threat_Level newLevel =
            _threatLevelCalculator.calculateThreatLevel(
                filteredThreat, oldLevel
            );

        //Create threat level object and set it in the air unit
        ThreatLevel threatLevel =
            new ThreatLevel(filteredThreat, newLevel);
        at.setThreatLevel(threatLevel);

        //Add the calculated threat value to a  list of calculated
        //threat values
        tvDataList.add(
            new T_Lap_Threat_Value_Data(
                at.getTrackNumber(), threatLevel.convert())
            );
    }
}
```
Listing 24: Sequential implementation of the threat evaluation.

To parallelize this loop the *CELP* framework has been used. The first thing done was to create a task from the loop-computations, which means that we tried to extract each iteration from the loop and make them independent so they can run concurrently. In the *CELP* framework that means implementing the `LoopComputation` interface, the implementation can be viewed in listing 25, for more information on the *CELP* framework see Chapter 6. To make sure each iteration is independent each call to a function in the loop has been analyzed by going through all call graphs. To make each iteration independent all shared collections have been changed to concurrent ones and the content of the loop has been moved to the compute method.

```
/**
 * Task used to compute all threats for a range of tracks
 */
private class ThreatEvalComputation implements LoopComputation {
    //Queue shared among threads to store calculated threat levels
    private final ConcurrentLinkedQueue<T_Lap_Threat_Value_Data>
        _tvQueue;
    //List of all airtracks. All tasks use individual indexes.
    private final CopyOnWriteArrayList<AirTrack> _atArr;
    //List of all defended assets. All indexes accessed by all
        tasks.
    private final CopyOnWriteArrayList<DefendedAsset>
        _defendedAssets;

    /**
```

```
     * Constructor, sets the shared resources for the loop.
    */
    public ThreatEvalComputation(
        ConcurrentLinkedQueue<T_Lap_Threat_Value_Data> q,
        CopyOnWriteArrayList<AirTrack> ats,
        CopyOnWriteArrayList<DefendedAsset> das){

        _tvQueue = q;
        _atArr = ats;
        _defendedAssets = das;
    }

    @Override
    public void compute(int i) {
        computeThreat(_atArr.get(i));
    }

    /**
     * Does the actual threat computation in each iteration.
     */
    private void computeThreat(final AirTrack at){
        ThreatValue currentThreatValue =
            evaluate(at, _defendedAssets);
        ...
        _tvQueue.add(
            new T_Lap_Threat_Value_Data(
                at.getTrackNumber(), threatLevel.convert())
            );
    }
}
```

Listing 25: Concurrent implementation of the threat evaluation.

When the loop computation has been implemented it is simple to run it in the loop task factory until there are no more tasks. This can be viewed in listing 26. To determine the number of tasks to divide the loop into, the benchmarking utility in the *CELP* framework has been used, for more information see Section 6.2.2. For the *CELP* framework to be able to handle an `InterruptedException`, the exception thrown by `taskFac.awaitLoop()` can not be caught. If it would be caught the framework can not effectively stop critical tasks when a *type 3 event* is received, more on this in Chapter 6.

```
LoopTaskFactory taskFac =
    new LoopTaskFactory(_nrOfTasks, atArr.size(), comp);
try {
    while(taskFac.hasNextTask())
        exec.submit(taskFac.createNewTask());
} catch ...
```

```
taskFac.awaitLoop();
```

Listing 26: The task is run in a loop like this.

Weapons Allocation Loop

The loop in the weapons allocation step performs the most *CPU* demanding computations in the *TEWA* component, and has also been parallelized with the *CELP* library. The allocation loop iterates over all air and firing units and finds all possible engagements between these two types of units. It then adds the engagement to a list containing all possible engagements. In the sequential case the loop looks like in listing 27. The heavy computations are performed in the `calculateEngagementValue(fu,at)` operation, which is run in every iteration. The calculations are independent of each other and are therefore very well suited to run in a task.

```
for (AirTrack at : ats) {

    short atTn = at.getTrackNumber();
    boolean external =
        _externallyDesignatedTracks.contains(at.getTrackNumber());
    for (FiringUnit fu : fus) {
        short fuTn = fu.getTrackNumber();
        DesignationData dData = new DesignationData(
            new T_Designation_Data(fuTn, atTn, source)
        );
        ...
        double ev = calculateEngagementValue(fu, at);
    }
}
```

Listing 27: The sequential allocation loop.

The concurrent version can be viewed in listing 28. This version is very similar to the sequential one but there are some important modifications. The loop has been modified so it is performed as one loop instead of two nested loops, in that loop each iteration can be run concurrently. In each loop, the compute method is run by the framework, and the air unit and the firing unit is obtained from the containers that holds all firing units and air units. The implementation is slightly slower than the nested loop solution in the single threaded case, because of the get operation, but since the work now can be divided over several tasks, the gain on a multi-core system is much greater. In Figure 14 the possible division of work in the old implementation can be viewed and compared with the new in Figure 15. From these two figures it can be seen that a more fine grained division of work allows for more tasks, and therefore a more fine grained solution. A more fine grained division is always preferable and scales a lot better to a multi-core system, it also allows for a better partition of the work when input is small.

```
private class WeaponAllocComputation implements LoopComputation{
    private final short _source;
        //Used to access airTracks between all tasks
        private final CopyOnWriteArrayList<AirTrack> _ats;
        //used to access all firing units concurrently
        private final CopyOnWriteArrayList<FiringUnit> _fus;
        private final int nrOfFiringUnits;

        public WeaponAllocComputation(CopyOnWriteArrayList<
            AirTrack> ats,
        CopyOnWriteArrayList<FiringUnit> fus, short source) {
            _source = source;
        _ats = ats;
        _fus = fus;
        nrOfFiringUnits = _fus.size();
        }

    @Override
    public void compute(int loopIndex) {
        // Get the two indexes for current AirTrack and current
            FiringUnit from the loop index
        // which goes over all indexes.
        int airTrackIndex = loopIndex / nrOfFiringUnits;
        int firingUnitIndex = loopIndex \% nrOfFiringUnits;

        AirTrack at = _ats.get(airTrackIndex);
                short atTn = at.getTrackNumber();
                boolean external =
            _externallyDesignatedTracks.contains(atTn);

        FiringUnit fu = _fus.get(firingUnitIndex);
        short fuTn = fu.getTrackNumber();

        DesignationData dData = new DesignationData(new
            T_Designation_Data(fuTn, atTn, _source));
        ...
        double ev = calculateEngagementValue(fu, at);
        _evList.addDesignation(dData, (float) ev);
            }
}
```

Listing 28: The concurrent allocation loop.

Optimize Function

When a weapons allocation has been performed it is sent to an optimize function that optimizes away unnecessary information from the proposed

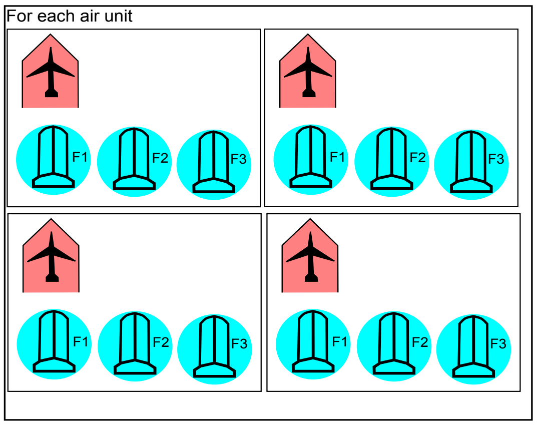


Figure 14: The old implementation where the two loops are nested.

allocation, for example engagements between fire and air units that can not be used. It also optimizes for the best probability of taking out a target and how much threat that target is to the defended assets. Since the algorithm is classified it can not be displayed here, but it can be explained in general terms. After some investigation we realized that the algorithm is a specialized version of bubble sort, which have a lot of things in common with the merge sort that was implemented in Section 4.1. So, the merge sort implementation with the fork/join framework from Java 7 could be applied to the optimize algorithm easily. The speedup gained when running the optimized version concurrently is almost identical to the merge sort even though the implementation is a bit customized. It works very well, and since the optimize function is a pretty performance intense function a lot performance is gained.

5.2.5 *CORBA* Event Handling

To get rid of all the unnecessary synchronization in the system the non-deterministic behavior of the *CORBA* threads had to be removed and exchanged for a more controllable approach. In the old *TEWA* implementation, a *CORBA* thread could at any time request access to a shared data structure and modify its content. To have a completely safe version of that approach all data structures must be locked when a running *CORBA* thread receives an event. The old implementation did that, even though the initial thought probably was to lock only a part of the system. The result of this behavior is that the system does not scale to a multi-core system at all. So, to be able to remove all the synchronization the handling of the *CORBA*

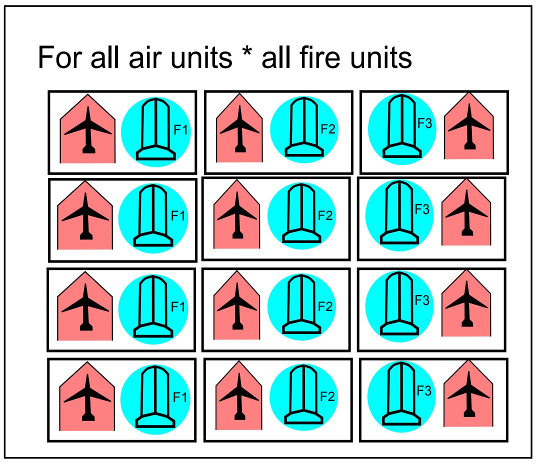


Figure 15: The new implementation where the two loops have been exchanged for one.

threads has to be performed at safe points in time when no other threads are accessing the internal data structures.

To achieve this, the software must be divided into critical tasks, or tasks that run *event critical code sections* as described in Section 6.2. When a certain task is running it has exclusive rights to all the data structures and if a *CORBA* thread receives an event that modifies any data it must wait to be executed after the task is done. An exception from this rule is if it is a *type 3* event like for example cease fire or hold fire (see Chapter 6), then it will interrupt the current running task, execute the event, then the main program is responsible for handling any corrupted data and resubmission of the task. As recalled from earlier the main loop of the software is divided into to four main parts:

1. All information is updated
2. Protection of friends
3. Threat evaluation
4. Weapons allocation

Each of the main parts is implemented as a critical task in the *CELP* framework. The division into *critical tasks* that run *event critical code sections* are described in Chapter 6.

In the new *TEWA* version, the remote component communication is handled via *CORBA* threads as events. They have been categorized accordingly to the three different types of events as explained in Chapter 6.2. Everywhere where an event may be received it is, instead of executing it directly, handled by the event handler from the *CELP* framework. For more information on the *CELP* framework and the event handling see Chapter 6. The result of the event implementation with the *CELP* framework is that, *all* the synchronization previously present in the component can be removed, and the huge overhead for all unnecessary synchronization is no longer present. The only synchronization that is still needed is when executing the *critical events* and when running the concurrent loops. Both of which are handled internally by the *CELP* framework. The concurrent collections used to assure concurrency also handles all synchronization internally. This results in that the *TEWA* component has been implemented without the use of the `synchronized` keyword or without performing any other dangerous locking. The scalability and performance results can be viewed in Chapter 7, where both the sequential and concurrent *TEWA* are run and compared in different environments.

6 Concurrent Event-Handling and Loop Parallelization Framework (CELP)

As part of the process of parallelizing the *Threat Evaluation and Weapons Allocation (TEWA)* component, used for threat analysis/weapons allocation of hostile radar detected air units, a general concurrency framework has been developed. It is motivated by the need to ease the implementation of scalable event-based components, like the ones used at Saab.

6.1 Introduction

This framework handles two major problems concerning concurrency that where detected in the analysis of the TEWA component, and similar components, used at Saab.

- Components communicate synchronously by invoking methods in other components, which can be invoked *at any time in a separate thread* handled by the *Common Object Request Broker Architecture (CORBA)* architecture.

- No scalability in current algorithms. To gain scalability CPU intensive loops must be parallelized in an effective way, so that the right balance between overhead and task granularity is found.

The first point requires an explanation of the *CORBA* architecture and how it is used in SAAB components. *CORBA* is used to enable components written in different languages to communicate with each other through common interfaces. In Java, this is implemented by having a `ThreadPoolExecutor` instance run several threads which may receive call-backs or events from other components in separate threads. Each thread blocks to receive an event from the *CORBA* naming service server, the events may for example update internal data-structures or just return data. This may make parallelization of a component especially difficult since it is impossible to know when a data-structure is updated. This is further explained in Section 5 and illustrated in Figure 16.

The second point regards the very CPU intensive loops that for example exist in the *TEWA* component. Several difficulties must be dealt with even if not considering any possible dependencies between loop iterations. First off, the iterations must be divided into separate computable *tasks* that can be distributed to running threads and mapped to available processor cores on the system. Secondly these tasks must be correctly sized somehow. Too small and the overhead will overcome any performance gain and too big tasks will result in poor concurrency since the computations can not be performed in parallel.

Figure 16 illustrates the first problem by showing how the main thread runs both safe and *event critical code*. The critical code could for example be a loop that accesses a shared data-structure that is not allowed to be modified during the entire loop. This shared data-structure could easily be made thread-safe *and* support concurrency by using a concurrent collection (see Section 2.2.2). *But* the algorithm that uses the shared data-structure might still require that it is not modified for the entire loop. In other words, it runs code that is sensitive to events that modify shared data and therefore runs an *event critical code section*. To handle this case in an easy way without a global lock and still handling the events in an efficient way, complicates the parallelization of any CPU intensive parts in the main thread. Figure 16 also shows that some events always are safe to invoke since they do not modify any critical state, i.e. they do not enter any *event critical code section* like *Corba thread #3* in the Figure. *Corba thread #1* executes an event critical code section but does not interfere with the *main thread*s critical section so it may safely execute. However, *Corba thread #2* executes a critical section that interferes with the *main thread*s critical section, which is dangerous and might have nondeterministic consequences.

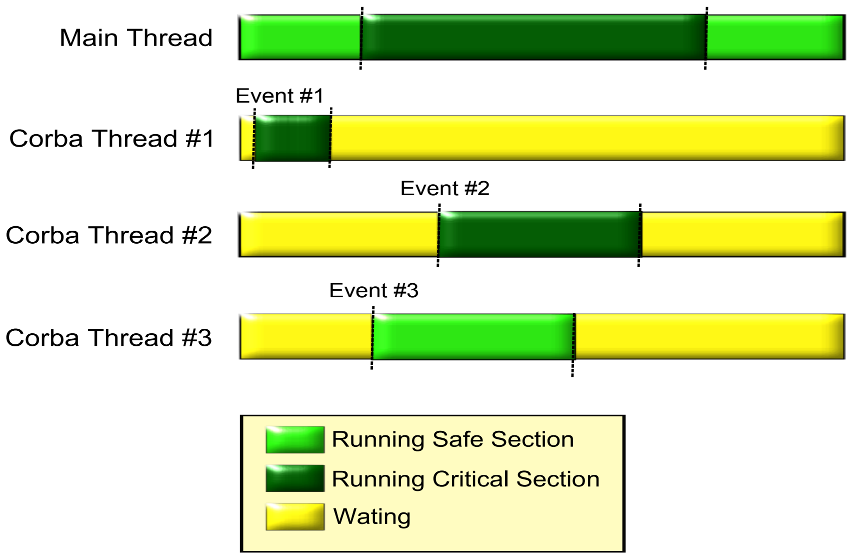


Figure 16: An illustration of how CORBA threads invoke events concurrently with the main execution.

The idea of the CELP framework is to make it easy to handle such events in a safe way with an event-handler and define critical code sections as critical tasks. The other idea is to support easy loop parallelization by dividing loops into the optimal sized tasks, in a way that allows for easy *and* performance effective parallelization. The different parts of the framework work excellent together. Loops that for example share data structures with an event might not be possible to parallelize without making the event thread safe. This can easily be handled with use of the built in event handling and loop parallelization. When using this technique it is also possible to easily interrupt long running computations if important events are received. This is possible since the loop parallelization supports interruption and the `CriticalTaskExecutor` supports handling of those interrupts. To use the framework properly, however, it is still very important to have a good knowledge about how the *Java Memory Model* works and how to use the `java.util.concurrent` package (see Chapter 2 Section 2.6.1).

6.2 Implementation

The implementation has been done in a package called `concurrent.celp` with a sub package called `concurrent.celp.benchm`, all classes reside in these packages which can be reviewed in appendix B.

Event-handling, critical task execution and loop parallelization are handled with the use of the three classes: `CriticalTaskExecutor`, `CorbaEventHandler` and `LoopTaskFactory`. The `CriticalTaskExecutor` executes critical tasks, and uses the internal event-handler to handle events. Events are handled by executing them if possible, or by queuing them for later execution otherwise. A critical task is defined by implementing the `CriticalTask` interface, shown in listing 29. Any `execute` implementation of this interface is not meant to be called directly, instead, this is handled by the `CriticalTaskExecutor.execute(CriticalTask task, ...)` method in the framework. This method executes the task in a safe way and when the main thread calls this method it is equivalent to the *main thread* entering the critical section, as in Figure 16.

```
public interface CriticalTask<E, V> {
    V execute(E arg) throws InterruptedException;
}
```

Listing 29: Interface that defines a critical task. The implementation of `executed` is considered to be critical. Generic parameters are used for variable return and parameter types.

When such a critical task is executed incoming events are handled differently considering how they have been invoked by the event-handler. The `EventHandler` instance should be obtained from the `CriticalTaskExecutor` with the `CriticalTaskExecutor.getEventHandler()` method. Events can then be invoked by the different `invoke*` methods which are listed in the Event-Handling Section 6.2.4. The `invoke*` methods takes implementations of the `Event<E>` interface as arguments, the interface is seen in listing 30.

```
public interface Event<E> {
    E handleEvent() throws Exception;
}
```

Listing 30: Interface that defines an event that could be both critical and non-critical it can also throw an exception and/or return a value defined by the generic parameter.

Depending of how the events are invoked, i.e which `EventHandler.invoke*` method is used, the events are treated differently. They have been classified according to their importance, timing requirements and the side effects of the code they execute, as listed here:

- **Type 1**: *Time Critical Events* that run *Non-Critical Event Code Sections*. These events must be executed right away, concurrently with the running main thread. The most common case is getters, for example when data is obtained from the *TEWA* component by some other component in the system. They can be executed in parallel with the main execution and other events since the code they run are not critical to the main thread or other events.

- **Type 2**: *Non Time Critical Events* that run *Critical Event Code Sections*, are events that are executed as soon as the current critical task is done. For example the getting and updating of references from a `ConcurrentHashMap`. These events must not intervene with any critical tasks in the main thread or other events that are running critical code sections.

- **Type 3**: *Time Critical Events* run *Critical Event Code Sections*. They are events that can interrupt the execution of the currently running critical task in the main thread and can therefore also be called *Interrupting Events*. This applies to events that are time critical but at the same time execute at least one code section that is critical for the main thread or other *Type 2* event, for example cease- and hold- fire commands in the *TEWA* component.

An example of how the `CriticalTaskExecutor` can be used is shown in listing 31. Here an implementation of a `CriticalTask` is executed concurrently while receiving events that modify shared data as seen in listing 32. The `invokeLater()` method is used to invoke the events as *Type 2* events which means that they are enqueued if the current main thread execution is in the `try` { `criticalExec.execute(...)` } block, and invoked *after* that block has finished executing instead.

```
static void main(String[] args) {

    CriticalTaskExecutor criticalExec =
        CriticalTaskExecutor.getInstance();

    //Start event receiving and run safe initializing code
    startEventReceiving();
    intialize();

    Integer taskArg = new Integer(0);
    try {
        //Run critical code
        criticalExec.execute(new CriticalTaskImpl(), taskArg);
    } catch (...) {...}
}
```
Listing 31: An example of how critical tasks can be executed with the use of a `CriticalTaskExecutor` and event handling as shown in listing 32.

```
public class EventReciever implements EventListener {

    private final Collection _sharedData = ...
    private final _eventHandler =
        CriticalTaskExecutor.getInstance().getEventHandler();

    @Override
    public void eventCallBack(){
        ResizeEvent event = new ResizeEvent();
        _eventHandler.invokeLater(event);
    }

    private class ResizeEvent implements Event<Void> {

        @Override
        public Void handleEvent() {
            // Modifies the shared data when the event is invoked
            _sharedData.resize(...);
            return null;
        }
    }
}
```
Listing 32: Example of how to invoke an event later *if* a critical task is executed at the same time otherwise it will be executed directly.

When using these facilities, for example as shown in listing 31 and 32, an event based system can be made thread-safe without any further synchronization. This alone does not give the system any scalability though, for this to happen any CPU intensive loop must be parallelized, which can be achieved by using the `celp.LoopTaskFactory` class. This may execute several loop iterations in parallel with a customizable amount of indexes per task. Its implementation and use is further explained in Section 6.2.1.

6.2.1 Loop Parallelization

The loop parallelization part of the framework is meant as a tool to divide loop computations into reasonably large subtasks. It is up to the user to find the correct size of a task, so that the performance gain of parallel computations out-weights the overhead induced by creating and distributing the tasks among running threads.

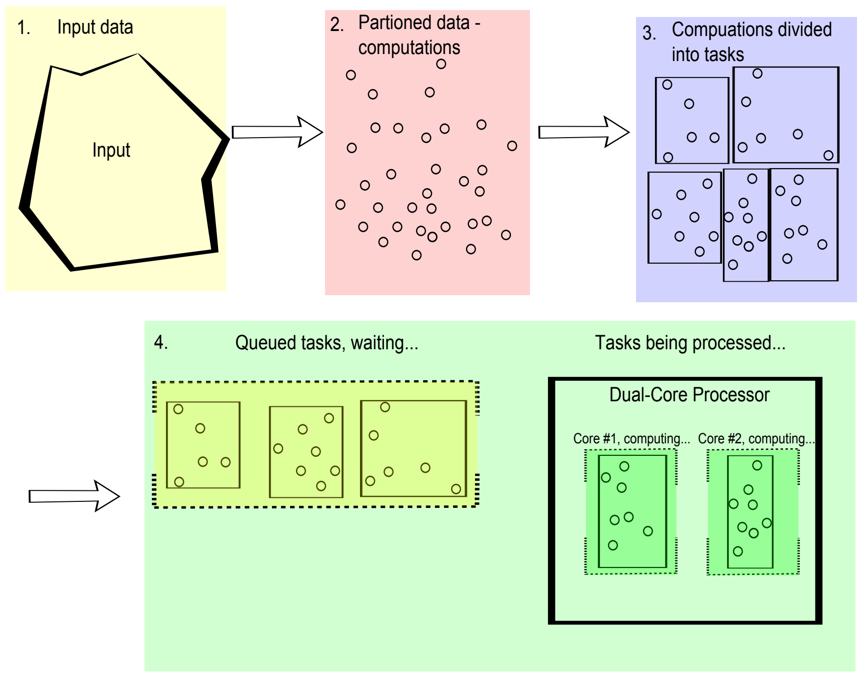


Figure 17: How input data can be partitioned, divided into tasks and mapped to different processor cores.

The general idea of how to parallelize an algorithm by distributing its smallest computations among tasks is illustrated in Figure 17, which is influenced by the *PCAM* model[4]. The flow is as follows:

1. Identify the input data. This could for example be a data-structure used for some computation.

2. Identify each minimal computation performed on the input data. If using the collection data-structure this could for example be a computation performed on each entry in the collection, i.e. iterating over the entire collection.

3. Each of these computations must now be distributed among suitably large tasks. Each task will be computed sequentially on a separate processor core. Two things are especially important at this stage; first dependencies must be detected, for example, is any computations depending on other computations being performed prior to them? Secondly, how demanding is each computation? I.e. how many computations must be done in a task so that the overhead of distributing them is less than the gain of parallel computation?

4. When the computations are distributed among tasks they should be queued up to wait being processed by available processor cores. This can be easily done in Java by for example submitting the tasks to a `ThreadPoolExecutor`

Each computation of stage two in Figure 17 is represented in *CELP* by the interface `LoopComputation`, shown in listing 33. The parameter supplied to `compute` tells which index in the loop is currently being processed or which entry in an array is currently being processed.

```
public interface LoopComputation<E> {
    public void compute(E entry);
}
```

Listing 33: Interface representing a basic computation

When defining a computation by implementing this interface a `Loop-TaskFactory` instance can be used to create tasks of the desired size. By emanating from the example showed in Figure 17 it can be shown how the `LoopTaskFactory` makes loop parallelization easier. As seen in listing 34 where input data stored in the collection `list`, is divided into the five tasks from Figure 17.

```
LoopTaskFactory<E> taskFac =
     LoopTaskFactory(5, list, new LoopComputationImpl<E>());
```

Listing 34: `LoopTaskFactory` usage where the iteration over a List `list` is divided into five tasks.

These tasks can easily be submitted to an `ExecutorService` which in turn queues the tasks to available threads that optimally runs on individual cores, represented by the fourth stage in Figure 17 and shown in listing 35. The `compute` method is called with the correct parameter in the `Runnable` returned when creating new loop computation tasks with the `LoopTaskFactory.createNewTask()`. Depending on the size of the task, a different number of computations will be performed. For example, in the left-most task of stage three in Figure 17, `compute` will be called seven times. Note that the `LoopTaskFactory.awaitLoop()` throws an `InterruptedException`, this should *not* be caught if the computation is used inside a `CriticalTask` implementation (which is the intention). It should be re-thrown since this means that it can be used to interrupt a lengthy computation, by for example receiving a critical interrupting event.

```
private final _exec = new ThreadPoolExecutor(...)
...
try {
    while(taskFac.hasNextTask())
        _exec.submit(taskFac.createNewTask());
} catch ( ... ) { ... }

taskFac.awaitLoop();
```

Listing 35: Submitting tasks and waiting for computation to complete

The `LoopTaskFactory(...)` has two constructors, as seen in listing 36. They differ at one argument, the first one takes a `List<E>` interface and the second takes the number of iterations as argument. The one taking the number of iterations allows for the users own implementation of what value to *get* when compute is called since the index is returned (as an `Integer wrapper`), instead of the actual generic entry from the collection. The index version might be useful when iterating over multiple collections but the `List<E>` might be more convenient when only one collection holds all values used for the computations and each entry is used in exactly one computation. Both of the constructors take the number of tasks that should be created as argument, it also takes the implementation of the computation. The actual amount of tasks could differ from the argument since it must be smaller or equal to the size of the loop being parallelized (second argument to the constructor). The size of a task is defined by how many computations it computes sequentially, which is calculated by dividing the total amount of computations with the amount of tasks. This must not be less than one since one computation basically is one iteration in the parallelized loop and thus the smallest possible computation, the size is defined by equation 1. The remainder from this division is distributed among the created tasks to get uniformly sized tasks. This can be very important when performing computations on big input data to prevent unevenly sized computations and poor concurrency as seen from the Experiments section.

```
public LoopTaskFactory(int, List<E>, LoopComputation<E>)
public LoopTaskFactory(int, int, LoopComputation)
public boolean hasNextTask()
public Runnable createNewTask()
public void awaitLoop()
```

Listing 36: `LoopTaskFactory` methods.

$$taskSize = max(1, \frac{nrComputations}{nrTasks}) \qquad (1)$$

The `LoopTaskFactory.hasNextTask()` method, seen in listing 36, checks if the all computations has been returned in the form of tasks with the `LoopTaskFactory.createNewTask()` method. In other words, `LoopTaskFactory.hasNextTask()` returns false when for example all five tasks in Figure 17, containing all computations, have been created. `LoopTaskFactory.createNewTask()` creates the actual task which is returned in the form of a `Runnable`, the `run` method is implemented as shown in listing 37. The difference between the `start` and `end` variables defines the size of the task which also is how many times the `LoopComputation.compute(...)` method is called. The private field `_entryGetter` is a interface, the implementation used determines if the index *or* the generic entry of the supplied collection should be sent to `compute(...)`. This is decided by which constructor is used, the one taking a `List<E>` interface or the one taking an `int` representing the number of iterations. A `CountDownLatch` is used to keep track of when all tasks have been computed. This latch is called with `CountDownLatch.await();` in `LoopTaskFactory.awaitLoop()` to block until the completion of all tasks.

```
new Runnable(){
    public void run() {
        for (int i = start; i < end; i++)
            _loopTask.compute(_entryGetter.get(i));

        _latch.countDown();
    }
};
```

Listing 37: The runnable returned by `LoopTaskFactory.createNewTask()`.

6.2.2 Benchmarking Tool

To obtain optimal results for a loop parallelization all the variables that affect the outcome must be tweaked. The variables in a loop parallelization are the *task size* and the *number of threads*. For this reason a package containing benchmarking tools for optimizing those variables has been implemented in the framework. The benchmarking utility creates statistics that can be displayed in a graph or in text. It tests different combinations of task sizes and number of threads for a given loop computation, for more information on the loop utility see Section 6.2.1.

To use the benchmarking tool an instance of the `BenchmarkConcurrentTaskSize<E>` class must be created. There are a few different constructors that provide the programmer with different choices. All the constructors take an implementation of the `BenchmarkLoopTaskFactory<E>`, which is a factory similar to the regular `LoopTaskFactopry<E>`. It is created in the same manner but without the task size and its implementation is exactly the same as the regular `LoopTaskFactory` with some benchmarking tools added. The best way to use the benchmarking utility is to use it in the actual implementation, so where a loop task factory is created also create a `BenchmarkLoopTaskFactory<E>`. This is illustrated in listing 38.

```
_comp = new SomeComputation(someParameters...);

LoopTaskFactory<Integer> taskFac =
    new LoopTaskFactory<Integer>(_nrOfTasks, someList.size(),
        _comp);

BenchmarkLoopTaskFactory<Integer> benchmarkTaskFac =
    new BenchmarkLoopTaskFactory<Integer>(someList.size(), _comp);
```

Listing 38: An implementation of a regular loop factory and following is the benchmarking loop factory

When a factory has been created an instance of the `BenchmarkConcurrentTaskSize<E>` can be created with one of the different constructors. As an option the programmer can pass an implementation of the `BenchmarkReset` interface, that is intended to reset modified states so that the benchmark will not harm the rest of the execution of the algorithm. The method `resetBenchmark()` is run after each performed benchmark.

```
BenchmarkConcurrentTaskSize(BenchmarkLoopTaskFactory<E> factory,
    int nrTests, int taskIntervall, int maxNrTasks, TimeUnit t,
    List<Integer> nrThreadsToTest)
```

Listing 39: A constructor taking a factory, the number of tests to obtain a mean from, the interval or step between each new task in the loop, the time unit for the tests (nano or milli) and a list of different number of threads to test.

```
BenchmarkConcurrentTaskSize(BenchmarkLoopTaskFactory<E> factory,
    int nrTests, int taskIntervall, int maxNrTasks,
    List<Integer> nrThreadsToTest)
```

Listing 40: A constructor taking a factory, the number of tests to obtain a mean from, the interval or step between each new task in the loop and a list of different number of threads to test.

```
BenchmarkConcurrentTaskSize(BenchmarkLoopTaskFactory<E> factory,
    int nrTests, int taskIntervall, int maxNrTasks,
    BenchmarkReset resetImpl, TimeUnit t,
    List<Integer> nrThreadsToTest)
```

Listing 41: A constructor taking a factory, the number of tests to obtain a mean from, the interval or step between each new task in the loop, the maximum number of tasks to create, a reset implementation, a time unit (only nano and milli) and a list of different number of threads to test.

```
BenchmarkConcurrentTaskSize(BenchmarkLoopTaskFactory<E> factory,
    int nrTests, int taskIntervall, int maxNrTasks, TimeUnit t)
```

Listing 42: A constructor taking a factory, the number of tests to obtain a mean from, the interval or step between each new task in the loop, the maximum number of tasks to create, a time unit (only nano and milli) and a list of different number of threads to test.

```
BenchmarkConcurrentTaskSize(BenchmarkLoopTaskFactory<E> factory,
    int nrTests, int taskIntervall, int maxNrTasks,
    BenchmarkReset resetImpl,TimeUnit t)
```

Listing 43: A constructor taking a factory, the number of tests to obtain a mean from, the interval or step between each new task in the loop, the maximum number of tasks to create, a reset implementation and a time unit (only nano and milli).

```
BenchmarkConcurrentTaskSize(BenchmarkLoopTaskFactory<E> factory,
    int nrTests, int taskIntervall, int maxNrTasks,
    BenchmarkReset resetImpl)
```

Listing 44: A constructor taking a factory, the number of tests to obtain a mean from, the interval or step between each new task in the loop, the maximum number of tasks to create, and a reset implementation.

```
BenchmarkConcurrentTaskSize(BenchmarkLoopTaskFactory<E> factory,
    int nrTests, int taskIntervall, int maxNrTasks)
```

Listing 45: A constructor taking a factory, the number of tests to obtain a mean from, the interval or step between each new task in the loop and the maximum number of tasks to create.

When the benchmarker has been created it can be used to run several tests. To be able to reuse the same benchmarker each run, a `set` method for the loop task factory has been implemented. It can be called with a new instance of a factory with `setLoopTaskFactory(BenchmarkLoopTaskFactory<E> factory)`, this allows for the test to obtain the mean of several benchmarkings. The benchmarker is started by calling the `benchMark()` method. When the benchmarking is done the result can be written to a graph file. To write the values to a graph-svg file, use the `printAsGraphLatestMean(String title, String folder, String filePrefix)` function. It will write the mean of all run benchmarks so far and clear out the means so that a clear benchmarking can be started. There are also three methods for getting the total mean of all benchmarking run since the object was created, it is the `getBestTime()`, `getBestNrThreads()` and `getBestNrTasks()`. To reset the values returned by `getBestTime()`, `getBestNrThreads()` and `getBestNrTasks()` the object must be recreated.

6.2.3 Critical Task Executor

The *Critical Task Executor*s main responsibility is to execute *Critical Tasks* without the risk of receiving any *Critical Event* that may corrupt important data-structures. It also support fast interruption of these critical tasks *if* they are waiting for a parallelized loop *and* re-throws any `InterruptedException`. Otherwise the interruption response is depended upon how often the threads interrupted status is polled, or the use of any blocking methods. The task execution is handled from a singleton instance of the `CriticalTaskExecutor` which has an internal `EventHandler` that should be used to handle any events, see listing 46.

Since the idea of this executor is to handle the main execution of programs, which parts in turn are parallelized, it *is not thread-safe* and should only be used in the main thread to execute its critical tasks.

```
private final EventHandler _evHandler =
    new EventHandler();
...
public EventHandler getEventHandler(){
    return _evHandler;
}
```

Listing 46: The internal `EventHandler` of the `CriticalTaskExecutor`.

A critical task must implement the `CriticalTask` interface, for example as shown in listing 47. To make sure that the execution is safe when executing this critical task, as seen in listing 31 on page 83, a `ReentrantLock` is used internally by the `EventHandler`. How this is done is explained in detail in Chapter 6.2.4.

```
class ConcurrentAnalyzer implements CriticalTask<Integer, Void>{
    @Override
    public Void execute(Integer arg) throws InterruptedException {
        this.analyze(arg);
        return null;
    }
```
Listing 47: Ex critical task.

The method `<E> boolean execute(final CriticalTask<E, Void>, final E)` of `CriticalTaskExecutor` is the only method that needs to be used by a developer. Internally there are some other methods including an override of the `Listener.update()` used to listen on the `EventHandler` for events that should interrupt the execution in the `execute(...)` method. All of the methods are as listed here in listing 48:

```
private CriticalTaskExecutor()
public static CriticalTaskExecutor getInstance()
public EventHandler getEventHandler()
public <E>
boolean execute(final CriticalTask<E, Void>, final E)
private <E>
void invokeCriticalTask(final CriticalTask<E, Void>, final E)
@Override public void update()
```
Listing 48: `CriticalTaskExecutor` methods.

The `CriticalTasks` are executed in a *single-threaded* executor which is created using `Executors.newSingleThreadExecutor()` in the constructor. The constructor is private and is invoked when creating the instance using the `getInstance()` method. In classes that are listeners for events, like the example in listing 32 on page 84, the events should be handled by the `EventHandler` instance retrieved by the `getEventHandler()` method. The `execute` method executes a critical tasks with a generic input parameter by invoking the private method `invokeCriticalTask(...)`, this method uses the internal event-handler to lock any critical event from being executed as seen in listing 49. This is executed in the internal *single-threaded* executor to allow interruption and handling of the critical task while it is running, this is shown in listing 50. Such interruption is done through the cancellation of the submitted tasks from the callback method `update()`, this is explained further in the next paragraph.

```
_eventHandler.lockCriticalEvents();
try {
    task.execute(execParam);
} catch(InterruptedException){
    // *Fall through to finally block and unlock*
} finally {
    _eventHandler.unlockCriticalEvents();
}
```
Listing 49: The execution of a critical task while locking the event-handler from invoking critical events. This is implemented in `invokeCriticalTask(...)` method.

```
private volatile Future<?> _taskResult;
private final AtomicBoolean _hasBeenCanceled =
    new AtomicBoolean(false);
...

// Wait for any executing interrupt events to finish before
// submitting the task.
_evHandler.waitForInterruptEvent();
_taskResult = _mainExecutor.submit(cTask);

// Cancel the task directly if an interrupting event is waiting
if(_evHandler.hasInterruptEventsWaiting())
    _taskResult.cancel(true);

try {
    _taskResult.get();
} catch (CancellationException e){
    _hasBeenCanceled.set(true);
    ...
}
...
```

Listing 50: The submittal of a critical task to the single threaded executor. It is possible for another thread to interrupt the execution which is handled with an atomic boolean and a `Future<?>` instance representing the asynchronous response of the critical task

To handle the cancellation off a critical task some special measures have to be taken. Critical tasks can be canceled if a *Type 3* event that may *interrupt* is received as defined on page 83, the implementation details are also explained in Chapter 6.2.4 about event-handling. The `Future<?>` instance returned from the `Excecutor.submit(...)` method is stored in a local `volatile` field of the `CriticalTask` and represent the pending result of the critical task, as seen in listing 50. The `update()` method is called when the internal `CorbaEventHandler` receives an event that may interrupt. The `CriticalTask` constructor registers itself as a listener to its internal event-handler to receive such updates, when an update come the current running critical tasks is canceled as seen in listing 51.

When a critical task is canceled it results in the thread that runs this task gets *interrupted*, this basically means that the threads interrupt status flag is set to true. The flag can be checked manually by polling the `Thread.isInterrupted()` method or, automatically by catching an `InterruptedException` that is thrown by any `wait` method . Since the `LoopTaskFactory.awaitLoop()` is such a method a `CriticalTask` that performs such a parallelized loop computation will always support fast interruption.

For this to work it is *important that any caught* `InterruptedException` *in the critical task is re-thrown or not caught at all*, otherwise the executor will never know that the task has been interrupted.

```
@Override
public void update() {
    _taskResult.cancel(true);
}
```

Listing 51: Cancellation of running critical tasks through a callback caused by a critical event that may interrupt.

If one or several events that interrupts the critical tasks execution is received it is important that they get invoked before another critical task is executed. That is why the `_hasBeenCanceled` flag is set to true and checked in listing 50. The call `_eventHandler.waitForInterruptEvent()` is there to wait for any such events in case a critical task has been canceled and a new one has been executed before the critical tasks are done executing.

Figure 18 attempts to illustrate how the different events, received through *CORBA* threads, interacts with the main thread. The main threads execution of this Figure could for example be as shown in listing 52 where the comments represent the main execution in Figure 18. The sequence of executions and events in Figure 18 is described in the following list.

1. Before the first critical task is executed, a *Type 2* event is received, and executed since the main thread is in a non event critical code section. Since this event is executing at the time of calling `criticalExec.execute(...)` the main execution is blocked until the event has completed. this is represented by the orange *Waiting to Execute* part.

2. The main thread executes the critical task until a *Type 3* event is received. This causes the critical task to be canceled and the thread it runs in gets interrupted. When this happens the `criticalExec.execute(...)` returns false which causes `if(!status)` to be true. When a critical task has been canceled it is the implementers' responsibility to make sure it is handled in a proper way.

3. The cancellation has been handled and the second critical task is executed but as seen in the Figure the *Type 3* event is not finished executing and a second *Type 3* event has been received and begun execution as well. The critical task is not executed until both of these *Type 3* events have finished execution.

4. The second critical task has finished execution without any cancellation. A *Type 2* event is received during executing this critical task, this i handled by the event-handler in `criticalExec` which enqueues the event. All such enqueued events gets invoked after the critical task is done.

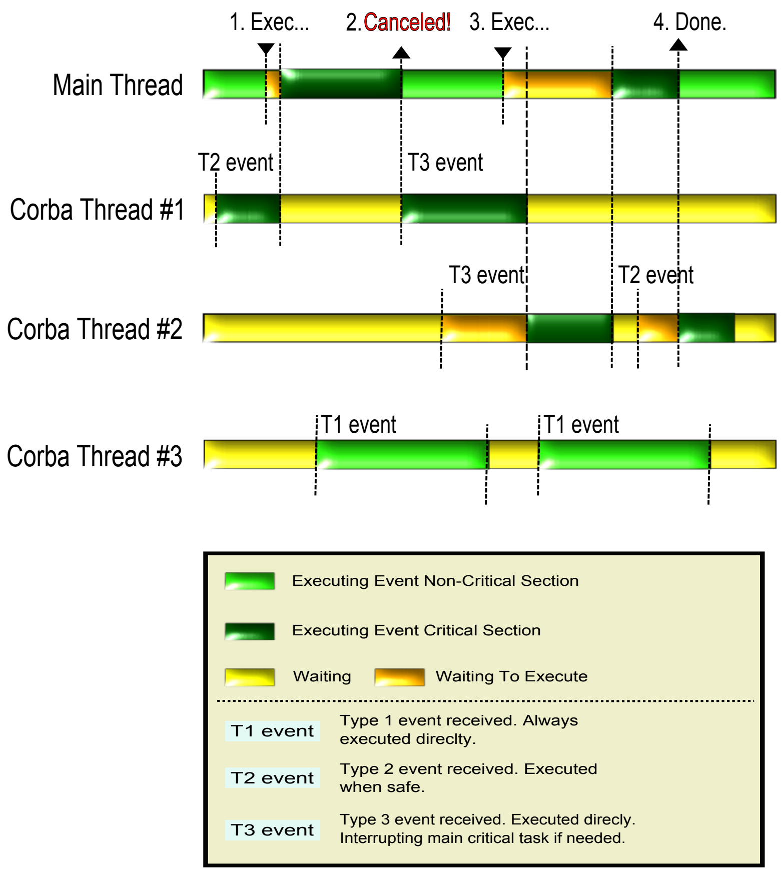

1. Exec...
2. Canceled!
3. Exec...
4. Done.
Main Thread
T2 event
T3 event
Corba Thread #1
T3 event
T2 event
Corba Thread #2
T1 event
T1 event
Corba Thread #3
Executing Event Non-Critical Section
Executing Event Critical Section
Waiting
Waiting To Execute
T1 event
Type 1 event received. Always executed direclty.
T2 event
Type 2 event received. Executed when safe.
T3 event
Type 3 event received. Executed direcly. Interrupting main critical task if needed.

Figure 18: Diagram representing the sequence when handling different types of events concurrently with the main execution of non-critical and critical -event code sections.

```
final CriticalTaskExecutor criticalExec =
        CriticalTaskExecutor.getInstance();
boolean status = false;


//Run non event critical code
initalize();
```

```
try {
    // 1. Exec...
    //Run critical code
    status =
        criticalExec.execute(new FirstCriticalTask(), ...);
} catch (...) {...}

if(!status){
    logError("First critical task was canceled"); // 2. Canceled!
    // *Handle cancellation*
}

//Run non event critical code
prepareSecondTask();

try {
    // 3.
    //Run critical code
    status =
        criticalExec.execute(new SecondCriticalTask(), ...);
} catch (...) {...}


if(!status){
    ...
}
else {
    // 4. Done
}
```
Listing 52: The main thread running the CriticalTaskExecutor illustrated in figure 18.

6.2.4 Event-Handling

The events discussed in the previous chapters is handled by the class `EventHandler`. The `CriticalTaskExecutor` has an internal instance of this class, which should be obtained by `CriticalTaskExecutor.getEventHandler()`, when handling events used while executing critical tasks.

To invoke events as the three different types defined on page 83 the different methods `invokeNow(Event<E>)`, `invokeLater*(Event<E>)` and `invokeNowInterrupt(Event<E>)` can be used. They correspond to *Type 1, 2 and Type 3* —events. The methods, together with all `EventHandler` methods can be viewed in listing 53.

```
//Event invoking methods
public void invokeLater(Event<Void> event)
public void invokeLaterLowPrio(<Void>event)
public void invokeLaterHighPrio(Event<Void> event)

public <E> E invokeNow(Event<E> event) throws Excpetion
public <E> E invokeNowInterrupt(Event<E> event) throws Excpetion

//Event lock/unlock methods
public void lockCriticalEvents()
public void unlockCriticalEvents()

//Blocks when Type 3, interrupting, events are executed
public void waitForInterruptEvent()

//internal invoking/queuing methods
private boolean tryInvokeCritical(Event<Void> event)
private <E> E invokeBlockCritical(Event<E> event)

private void addEvent(EventQueueEntry<Event<Void>> event)
private void processEventQueue()
```

Listing 53: `EventHandler` methods.

The three `invokeLater*(Event<E>)`, seen in listing 53, are all put in an internal `PriorityBlockingQueue`. Depending on which invoke method is used different instances of the abstract class `CorbaEventQueueEntry` are created, for example `CorbaEventQueueHighPrioEntry<E>` for high priority entries and `CorbaEventQueueFifoEntry<E>` for *First-In First-Out (FIFO)* entries. The *FIFO* entries are only *FIFO* ordered relative other instances of the `CorbaEventQueueFifoEntry<E>` class, not relative to the high and low priority events.

The `EventHandler` protects the execution of the events accordingly to how they are invoked. This is primarily done through the *Critical Event Lock* (`criticalEventLock`) seen in Figure 19 and the `Phaser` seen in the same Figure. When for example the `CriticalTaskExecutor` wants to protect a critical task from being interfered by any events that execute event critical code sections it uses its internal `EventHandler` to take the *Critical Event Lock*, as seen in listing 54. The `eventQueue` is processed whenever the `criticalEventLock` is released, since the only time events will be queued is when this lock is taken (must be done in a `finally` block as seen in listing 49 on page 92).

```
public void lockCriticalEvents(){
    _criticalEventLock.lock();
}
```

Listing 54: Take critical lock in `EventHandler`

When this lock is taken a received *Type 2* event will not be executed, but queued instead as seen in listing 55.

```
public void invokeLater(Event<Void> e) {
    // Try to take lock and invoke, otherwise queue the event
    if(!tryInvokeCritical(e))
        // Add event to PriorityBlockingQueue
        addEvent(new EventQueueFifioEntry<Event<Void>>e);
    }
}
...
private boolean tryInvokeCritical(Event<Void> e){
        if(_criticalEventLock.tryLock()){
            try{
                invokeNow(e);
            }
            catch(Exception exc){...    }
            finally{
                _criticalEventLock.unlock();
            }
        return true;
    }
    return false;
}
```

Listing 55: *Type 2* events are queued if they can not be executed

Type 3 events may interrupt the running critical task and must be executed as soon as a critical task has been canceled, this is done as seen in listing 56 and as illustrated in Figure 19. The call `notifyListeners()` notifies any `CriticalTaskExecutor` (that must listen on its event-handler) that a *Type 3* event has been received and that the critical task must be canceled if running. When the critical task is running it holds the `criticalEventLock` lock, to make sure that this event is not executed until it is released the `invokeBlockCritical(event)` call blocks waiting for this lock to be available. This method is potentially dangerous since there is a blocking operation while holding a lock (the event is registered on the `Phaser`). The framework guarantees that the lock is released when the critical task finishes, but if the implementation of the critical task deadlocks, the framework will deadlock. The `CriticalTaskExecutor` only waits using the `waitForInterruptEvent()` call when *not* holding this lock as well, which assures that no deadlock will occur there.

```
public <E> E invokeNowInterrupt(Event<E> event)
    throws Exception {

    E res = null;
    _criticalInterruptPhaser.register();
    try
    {
        notifyListeners();
        res = invokeBlockCritical(event);
    } finally {
        _criticalInterruptPhaser.arriveAndDeregister();
    }
    _nrDireclyInvokedInterruptEvents.incrementAndGet();
    return res;
}
```

Listing 56: *Type 3* events are executed when any critical task has been canceled. This is assured by the method `invokeBlockCritical` that blocks until the cancellation of the critical task. New critical task execution are prevented by using the `Phaser` instance where interrupt events register them-self and new critical tasks are not allowed to execute until all of them have deregistered.

As seen on page 95 in Figure 18 new *Type 3* events may arrive when such an event is already executing. To make sure that these important event gets executed before a new critical task is executed the events *registers* on an internal `Phaser`. Before launching a new critical task all these events must *unregister* on this phaser before the method seen in listing 57 unblocks. The `CriticalTaskExecute.execute()` calls this method on its internal event-handler if it was recently canceled.

```
public void waitForInterruptEvent() {
    _criticalInterruptPhaser.arriveAndAwaitAdvance();
}
```

Listing 57: The `Phaser` unblocks when all registered parties has unregistered. New parties may register during this waiting period.

7 Simulation

Since the *TEWA* implementation is a complex component that is dependent on a lot of other components it requires a proper test environment. Major design changes have been made and the only way to verify the correctness, as well as any scalability and performance improvements, is through proper simulation. The goal is that the simulation should resemble the real complex environment as much as possible. To understand how the *TEWA*

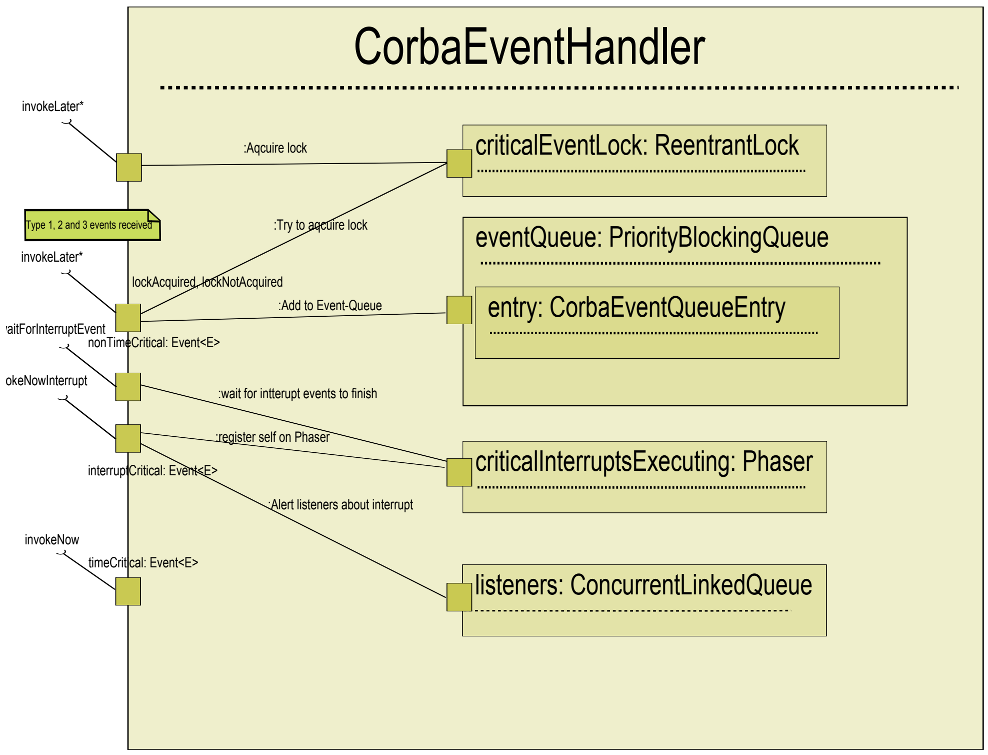

CorbaEventHandler
invokeLater*
:Aqcuire lock
criticalEventLock: ReentrantLock
Type 1, 2 and 3 events received
:Try to aqcuire lock
eventQueue: PriorityBlockingQueue
invokeLater*
lockAcquired, lockNotAcquired
:Add to Event-Queue
entry: CorbaEventQueueEntry
aitForInterruptEvent
nonTimeCritical: Event<E>
okeNowInterrupt
:wait for intterupt events to finish
:register self on Phaser
criticalInterruptsExecuting: Phaser
interruptCritical: Event<E>
:Alert listeners about interrupt
invokeNow
timeCritical: Event<E>
listeners: ConcurrentLinkedQueue

Figure 19: A Composite Structure Diagram representing the internal structure of the `EventHandler` class.

component is simulated its behavior must be roughly understood, this is explained in Section 5.

7.1 Setting up the Environments

To allow for the testing and verification of the parallelized *TEWA* component the different environments need to be configured. The basic details about the environments are explained under Tools, Chapter 3.

The *TEWA* component is part of a set of components responsible for the logical computations in the *Giraffe AMB*. Other set of components handle *GUI* presentation and other details needed for the *TEWA* to operate. This is simulated using different computers connected through a closed network in the Linux based lab environment. The logic part, including the *TEWA*, run on an 8-core server. On the Windows based development platform this is simulated on a single dual-core system using a set of applications instead.

Since timing tests of algorithms are sensitive to different settings and varying system load it is important to perform many varying test cases. The settings that affect the execution of the *TEWA* component is: which parts of the component that is active, how much that is supposed to be processed, number of tasks, number of threads and old and new synchronization methods. The different settings used are described in table 2. The simulations where performed by designing test cases according to these settings, the results of the computations are viewed a bit differently depending on which environment is being used. The timings are collected in the same way on both systems using a benchmark implementation, which is an extension to the *CELP* library.

Algorithm Main Loop Settings
The main loop has two main bottlenecks which have been parallelized
Only Threat Evaluation: *TE*
Threat Evaluation and Weapons Allocation: *TEWA*

Input and System Load Settings
The system load is determined by the number of air units and the defended assets input.
The total System Load is defined as Low, Medium and High (L, M, H) and is determined by the number of air units (AUs) and firing units (FUs):
System Load (L – 2 FUs 4 AUs, M – 8 FUs 22 AUs, H – 12 FUs 91 AUs)

Concurrency Settings
The performance can be optimized by tweaking the number of threads used,
as well as how many tasks the input is distributed among for the different algorithms.
Thread count: *Threads* (1+)
Task size: *Tasks* (1 – input size)

Implementations
The original sequential TEWA is compared with the new parallelized version.
Sequential unoptimized TEWA: *ST*
Concurrent TEWA: *CT*

Table 2: Different settings used to test the *TEWA* component

7.1.1 Windows based, Basic Environment

Initial testing and simulation is done on the development platform which is a Windows based dual-core system. To simulate the *TEWA* component, a set of test applications, provided by Saab, have been used. An example of this environment can be seen in appendix C. Since the components in Saabs system communicate through the *CORBA* architecture, using a name server to look-up class instances (which runs by default in the lab environment), such a server must be reachable when performing the simulation. A name server called `tnameserv.exe` that comes bundled with the *Java Runtime Environment* is run on the Windows system, as a part of the simulation. It listens on a local port through which the different applications can communicate. The important applications used are (their purpose might be better understood by comparing them to how the *TEWA* component function, this is explained in Chapter 5):

- **SimMissionManager** – Simulates the mission settings sent to the *TEWA* component. For example radar position; firing units position, ammo and status etc.; Protection values and engagements.
- **TrackTest** – Simulates air units (or air *tracks*) sent to the *TEWA* component. For example position, velocity and identity. It also receives updated from the *TEWA* about calculated threat values.
- **SimMMI** – Simulates the interface used to communicate with the *TEWA* component, these commands are normally sent by an operator in the *Giraffe Amb*. Could for example be *Cease Fire* orders and manual engagements between firing —and air units. This interface also determines if the only threat evaluation or both threat evaluation and weapons calculation should be used in the *TEWA*.

7.1.2 Linux based, Lab Environment

As described in the Tools chapter, nr. 3, a lab environment that simulates the *Giraffe AMB* system is present at Saabs facilities. This environment have been used in this thesis to evaluate, benchmark and debug the concurrent *TEWA* implementation. The simulations have been performed on an 8-core server with hyper-threading running a Linux operative system. The system is implemented in such a way that the different components run on different *JVMs* and they are communicating via *CORBA* interfaces. The logic part, which contains the *TEWA* component, is run on the 8 core sever which means that the *TEWA* has to share the system resources with other components. The other logic components are not that *CPU* intense and none of them are implemented to exploit a multi-core architecture, meaning the *TEWA* can make full use of the system. Also sharing the resources on the server system are some simulation applications which simulate the actual radar of the *Giraffe AMB*. This requires some *CPU* but this is less then a percentage of the total computing power, so it is not a problem when performing the simulations and timed runs.

The lab environment is a closed environment with no Internet access, so to be able to debug the software and to profile it we had to do use the remote tools present in the applications used.

7.2 Parallelized *TEWA*: Varying Concurrency Settings

The Parallelized *TEWA* component has been run using a varying amount of threads and tasks. To find the optimal settings, the benchmarking utility from the *CELP* framework has been used, for more information see Section 6.2.2. The benchmarking can be customized to run under different settings. In this implementation it has been used to test the performance of the *TEWA* component with varying task size and a few different thread settings. The benchmarking is used to find the best settings for concurrent loops, which the two main loops in the concurrent *TEWA* are. The result when running the benchmarking on the threat evaluation loop and on the weapons allocation loop during heavy load can be viewed in Figure 20 and in Figure 21.

From the graphs it can be determined that both the *threat evaluation* and the *weapons allocation* works best on around 16 threads and tasks. The *threat evaluation* graph shows that the 8 threaded implementation is a bit faster than the one with 16 threads, but the difference is barely noticeable. The reasons for this behavior are the hyper threading, rounding of the measurements and the very fast algorithm.

The benchmarking has also been run on medium and low load. The threat evaluation then is almost negligible and when running the algorithm on low load the task size and thread size does not matter since the execution time is so low anyway. One could tweak the performance even further and maybe gain one or two milliseconds, but at this stage it seems unnecessary. The weapons allocation is still optimal on 16 threads and 16 tasks and the graph is almost identical to Figure 21. Since each task created has about the same execution time as the others, around 16 tasks and threads always seems to be the optimal setting. When the load is too low this is limited by the number of tasks that are possible to create from a given loop.

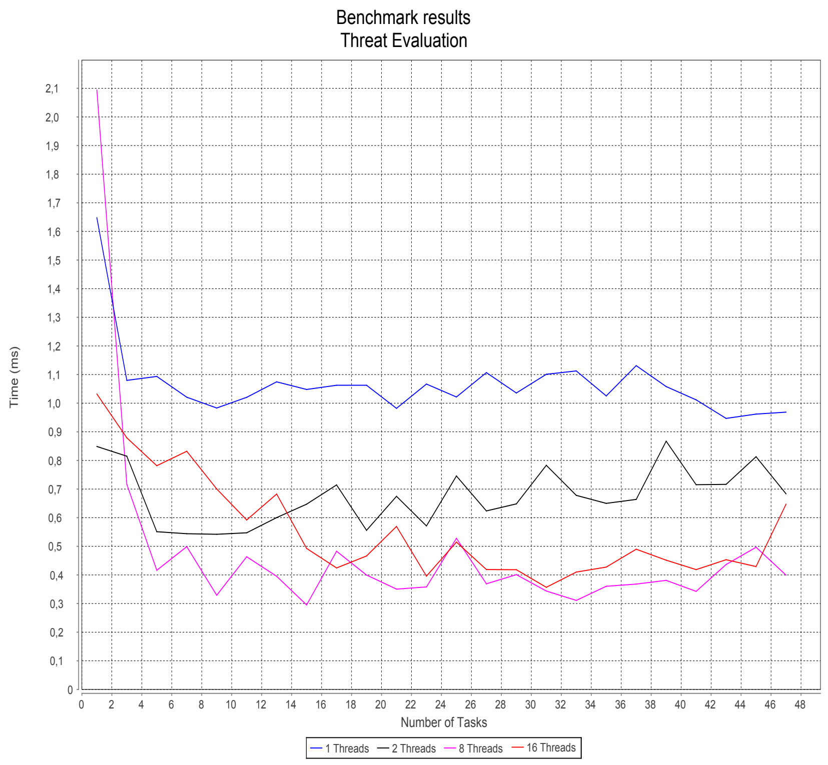


Figure 20: Benchmarking of the threat evaluation during heavy load.

When using this optimal task and thread settings the scalability of the *TEWA* component is as seen in Figure 22, during high load. At medium system load the scalability is good as well, as seen in table 3 on page 109, where the execution times during medium loads and full parallelization level is shown. A decrease of the speedup can be seen per doubling of the threads, this is due to the sequential part of the component that can not be parallelized. This is in line with *Amdahls Law* about possible speedups in parallelized software. The performance increase is less in the 8 to 16 threads step, this is reasonable since the *Hyper Threading* technique only simulates the extra doubling of available cores.

A summarization of the previous graphs can be seen together with the sequential test data in table 3 on page 109, where some timings using specific settings on different loads are shown. The tasks size refers to the number of tasks that both the *threat evaluation* and the *weapons allocation* input computations has been distributed among. This amount may vary some but they are about the same in these test runs so one number is used for both. For most cases in this parallelization it is very efficient to divide the

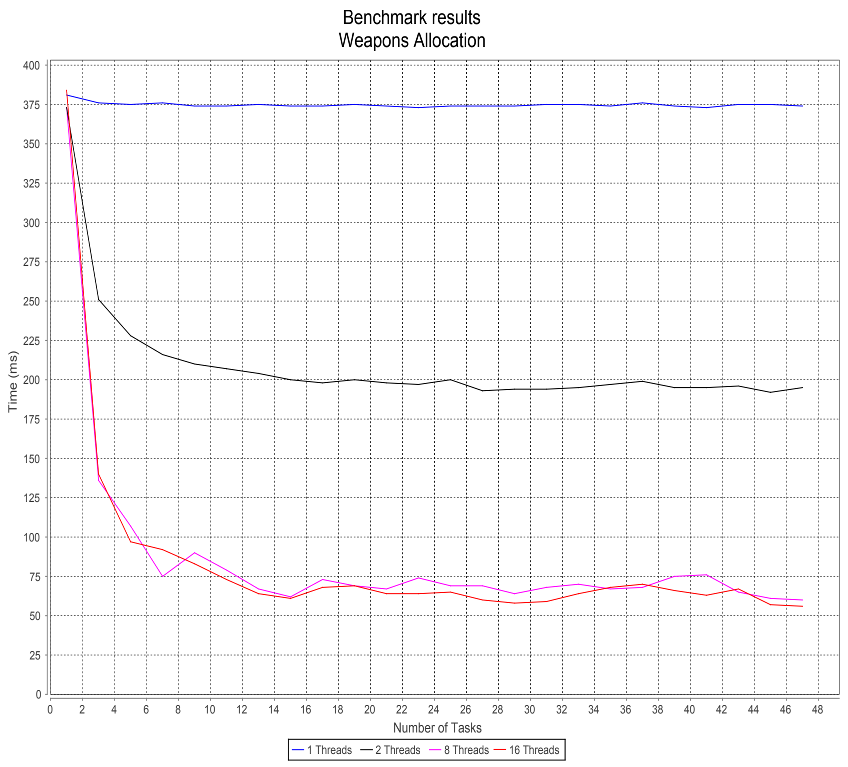


Figure 21: Benchmarking of the weapons allocation during heavy load.

computations to an amount of tasks that are equal to the number of cores and threads used. In the case of *low* loads there are simply not enough input data to distribute the computations to that many tasks. This is not a problem though, since the low load computations are very fast, even with the lower parallelization level. Another note about the *low* load timing runs; the *total execution time* of the *TEWA* during these runs are so short that the benchmark graphs showing optimal task and thread settings has been omitted. That data can be seen in the table in the *low* load column.

7.3 Parallelized *TEWA*: Varying Synchronization Methods

Before the *TEWA* algorithm had been fully parallelized it still contained old means of synchronization, mostly with a lot of unnecessary `synchronized` blocks. To see how the different worker threads in the loop parallelizations worked when the old synchronization was still present, visual vm has been used to monitor the *TEWA* algorithm running live in the lab simulation system.

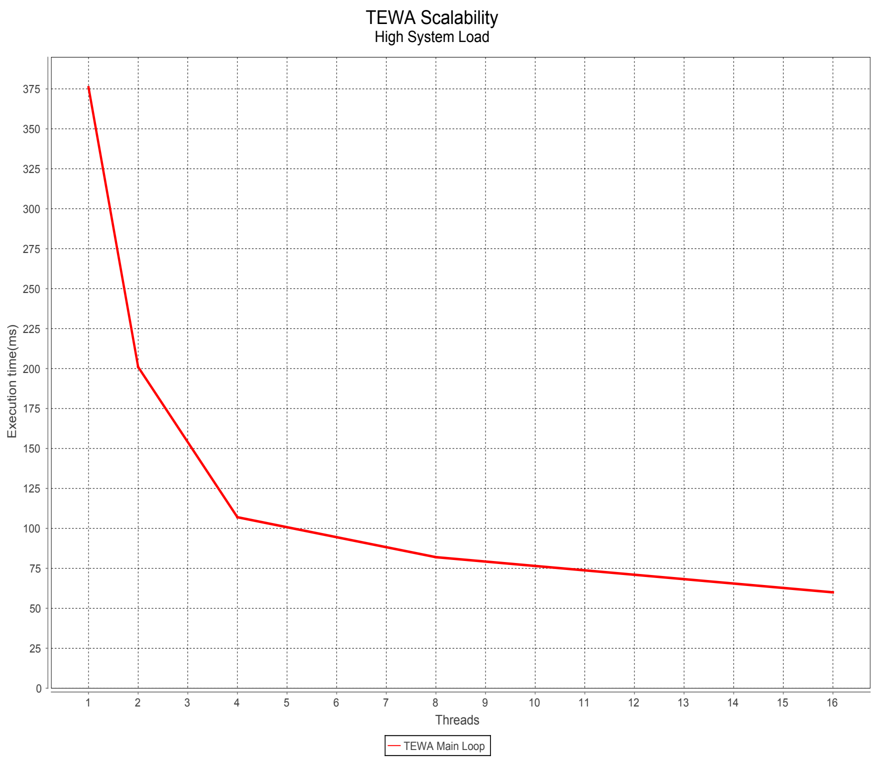


Figure 22: 8-core w. *Hyper Threading*: The scalability of the *TEWA*, using optimal task and thread settings, during heavy load.

The result of the simulation showed a huge difference in how the worker threads used their time, the result can be viewed in Figure 23 and in Figure 24. In the old implementation the threads spend a lot of time waiting for and holding locks (monitor) and in the new implementation no time is spent on waiting for locks. When a thread is waiting for a lock it is actually waiting for another thread because another thread is currently holding that lock. At some points in Figure 23 this is very obvious. The Figure speak for them selves and a performance increase with almost 50 percent can be observed.

7.4 Timings Comparison of Parallelized and Sequential *TEWA*

Simulations, similar to the ones performed on the concurrent *TEWA*, has been performed on the sequential *TEWA* as well to compare the two versions. The difference in execution times under high load can be seen in Figure 25. As expected the sequential version does not scale at all when

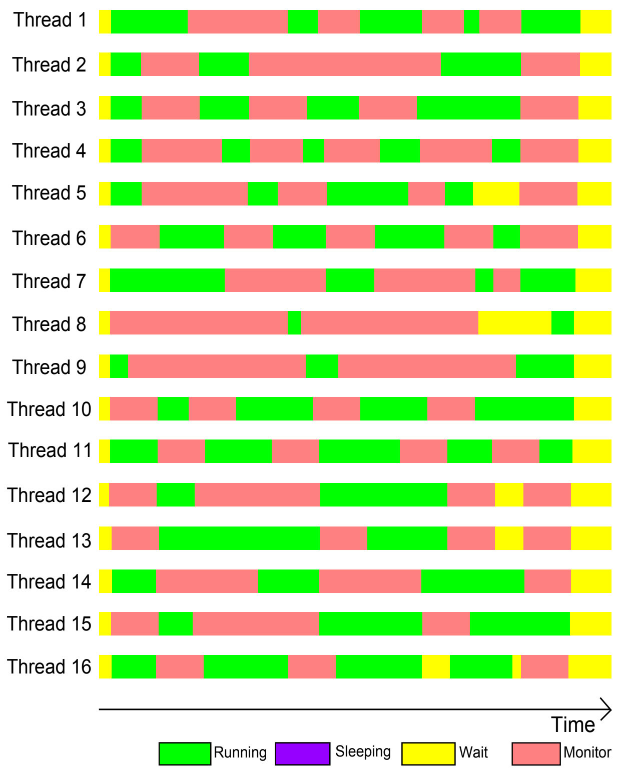


Figure 23: A snapshot displaying the 16 worker threads running with the old synchronization, a lot of waiting/holding locks (monitor) and a long execution time.

more threads are used since all computation in *CPU* intense loops, like the *threat evaluation* and the *weapons allocation*, are performed in a single thread and thus can not distribute the workload over more then one *CPU*.

The difference in execution times is also clear under medium load as can be seen in table 3 on page 109. The input size under medium load is still big enough to allow for good distribution of computations between different tasks, without the overhead exceeding the gain of parallelization.

In Figure 25, as well from table 3 regarding lower loads, it is very clear that the concurrent version of the *TEWA* component is significantly faster then original sequential version, even in the single-threaded case. This can not be explained by the parallelization of the *CPU* intense algorithms, since they rely on threads to distribute the computations among several *CPU*s. The explanation for this is, as described in Chapter 5, that all old synchronization has been replaced in favor for more modern lock free methods using tools form the `java.util.concurrent` package. The original code

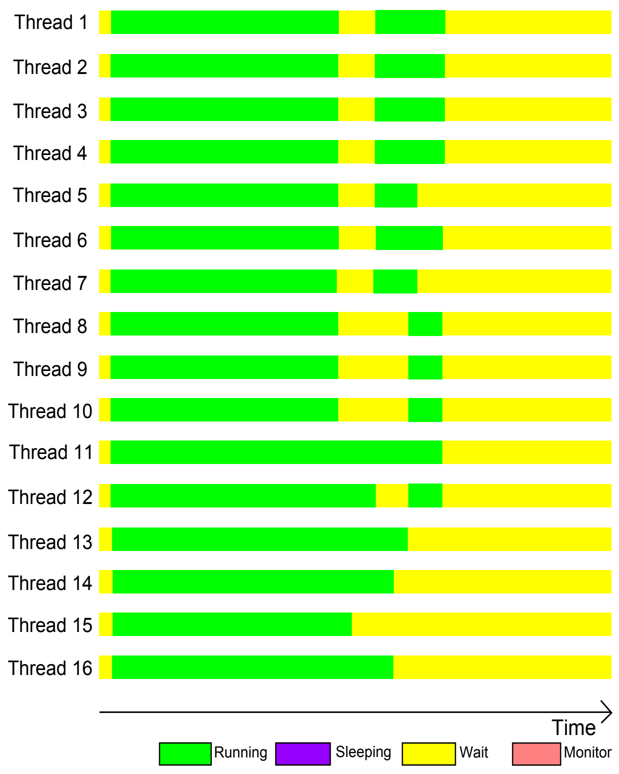


Figure 24: A snapshot displaying the 16 worker threads running with the new synchronization methods, no locking is required and the execution time is greatly reduced.

also contained several layers of synchronization which in many cases was unnecessary. Some of the optimization can also be explained by other code optimizations, but not nearly enough to explain all the performance gain seen in the single threaded case.

Some of the more important data collected from the previously mentioned graphs has been summarized in table 3. In this table the great impact that the parallelization has had on the *TEWA* component on the execution times, for *all* system loads, is clear.

The sequential and concurrent *TEWA* has also been run on the dual-core windows based system, results of this are shown in table 4. An interesting note about this table is that when the thread and task count is increased above the number of cores the performance decreases. This is a result of unnecessary overhead due to unused resources when using more threads and tasks then can be utilized by the system. This table also shows that the concurrent *TEWA* performs significantly better then the sequential one

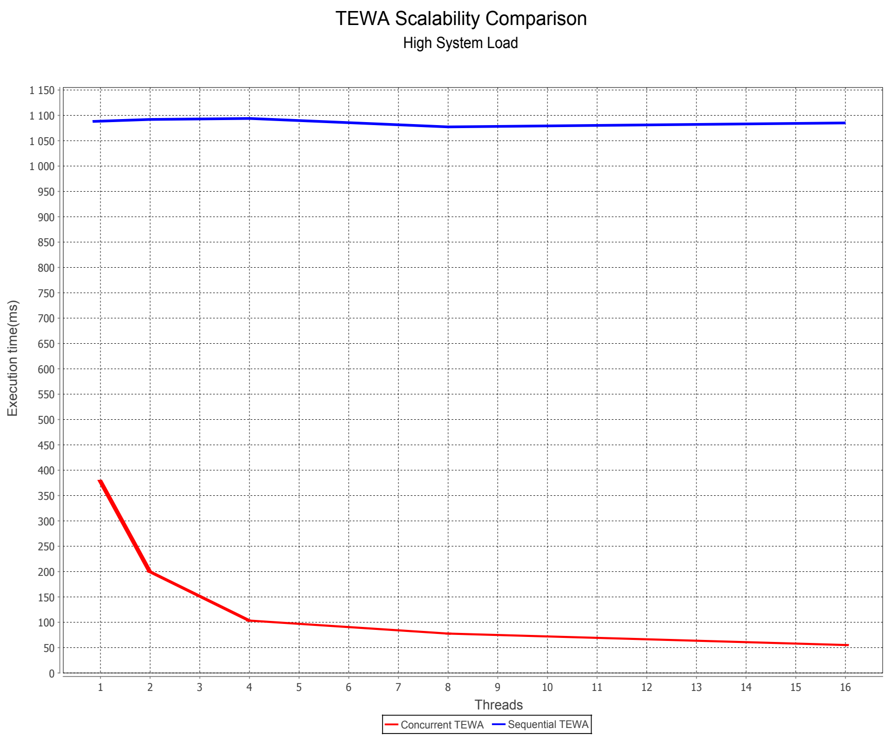


Figure 25: Comparison of execution times when running the *TEWA* component under high system load.

TEWA Execution Time: 8-Core w. Hyper-Threading CPU

	Sequential				Concurrent			
Test Nr.	1	2	3	4	1	2	3	4
Main Loop	TEWA	TEWA	TE	TEWA	TEWA	TEWA	TE	TEWA
System Load	L	M	H	H	L	M	H	H
Threads	-	-	-	-	16	16	16	16
Tasks	-	-	-	-	6	16	16	16
Time (ms)	**15**	**125**	**67**	**1065**	**2**	**12**	**<1**	**65**

Table 3: Comparison between the sequential and concurrent *TEWA* component, showing the total execution times using different settings.

on this system as well. This is the type of system that is currently used in the *Giraffe AMB* system which means that significant performance improvements are possible at Saab today, without the need for any hardware upgrades.

TEWA Execution Time: Dual-Core CPU

	Sequential		Concurrent			
Test Nr.	1-2	3-4	1	2	3	4
Main Loop	TEWA	TEWA	TEWA	TEWA	TEWA	TEWA
System Load	M	H	M	M	H	H
Threads	-	-	4	2	4	2
Tasks	-	-	4	2	4	2
Time (ms)	**200**	**1490**	**43**	**42**	**315**	**310**

Table 4: Comparison between the sequential and concurrent *TEWA* component running on a dual-core machine, showing the total execution times using different settings.

8 Discussion

The results of the *TEWA* parallelization show good scalability on an 8-core system, but the scalability with a multi-core system that uses more cores can be discussed. As Amdahl states in his law, a software system is not faster than its slowest sequential component. For a system to scale linear to the number of processor cores, it must be completely parallelized, meaning that there basically are no such sequential components. The *TEWA* implementation does not scale linear, since it still has some sequential parts, but it scales very well with the 8-core system that has been used for the simulations. Following Amdahls law, the sequential part of the *TEWA* will not increase in performance on a system with more cores, so the performance gain will decrease when more cores are used. In the general case this means that good scalability is always depended on how big a part of a component that can be parallelized. This is important to take into account when investing resources into parallelization of software, such as education and refactoring. However, the only way to use modern hardware is to parallelize software, so the pros and cons must be weighed against each other.

The results show that, achieving a good scalability is done by minimizing a components sequential parts. The usage of low level constructs such as `synchronized`, `wait()` and `notify()` should therefore be avoided as far as possible. However, it can sometimes be hard to avoid for advanced implementations, the *CELP* framework uses some locks and the concurrency library in Java uses a lot of low level constructs. These constructs are hard to use, error-prone and hard to get a scalable behavior with, and the frameworks and libraries are implemented so that the developers does not have to use them.

Other important aspects when investing in parallelization is that there are other programming paradigms, some of which removes one of the main problems that exists with concurrent programming in Java, *shared data*. The most common ones are functional languages such as Ericssons Erlang, Microsofts F#, Haskell and Scala for the JVM. The advantage of these languages are that they are designed for concurrency from the start and have a minimum of, or none, shared data. The main problem with them is that the programming style is very different from traditional object oriented programming that many developers are used to, and are therefore not widely used.

9 Conclusions

The purpose of this thesis was to review the latest technology for concurrent programming in the Java programming language and to parallelize an existing sequential component, running at Saab Electronic Defence Systems, making it concurrent and scalable. Based on the research of Java concurrency and the parallelization of the *Threat Evaluation and Weapons Allocation* component, it has been concluded that *a lot of software is written in a thread safe way but with none or very little scalability*. The problem resides in the fact that, up until recently, it has not been necessary to write software that take advantage of multi-core architectures. This has resulted in that, many programmers are not used to write scalable concurrent software. Our research and implementations shows that, concurrent software can be developed efficiently and safely if: *Software is designed for concurrency from the beginning of the development process* and if the developers are *educated in Java concurrency*.

A problem seen in Saabs software components, including the *TEWA*, was that they communicate with each other synchronously using *CORBA*, which complicates parallelization of the components. To be able to safely parallelize any of Saabs components, the *CORBA communication between components must be handled so it does not disturb important algorithms and parallel computations*. The *CELP* framework was developed to solve this problem in a general way and also contains tools for parallelizing loops, to allow for efficient parallel computations that scales to multi-core architectures. From the research and implementation performed in this thesis the following general conclusions can be drawn on the developed *CELP* framework and the Java concurrency library:

- *Avoid older tools from before Java 5 and avoid using `synchronized` blocks*, since they greatly limits how many threads that can access shared resources at the same time. Older tools may also result in safety issues.

- *Use modern concurrency tools, as those found in the standard `java.util.concurrent` package.* They generally allows for many threads to access shared resources at the same time, thus allowing for scalability.

- The new Java 7 standard contains some new concurrency tools like the fork/join framework. This framework, and other news from Java 7, are good for certain problems (divide and conquer problems, in the fork/join case) *but they are in no way any new general tools that should be used for all problems.* Much of the tools from Java 5 and 6 like atomic variables, the executors framework and the concurrent collections are more useful in the general case.

- *The best scalability and performance increase is achieved with the right tools and different problems require different tools.* The *TEWA* parallelization and the experimentation section both shows that different problems are solved best with different tools. Tools from the entire Java concurrency library have been used, and just because a certain tool is more recent than some other it does not mean that it is better for a certain implementation.

- *Design for concurrency from the beginning, make use of known design patterns and use framework- or library tools to handle parallelization in a general way.* This can be achieved by using *design patterns* like the ones mentioned in the theory chapter, understanding Javas Memory Model and using the *CELP* framework from this master thesis.

- *Parallelize CPU intense parts of the programs*, such as loops, by dividing them into tasks which can be computed concurrently, for increased performance and scalability. This is seen in the results of the parallelized *TEWA* component where modern concurrency tools are combined with loop parallelization tools of the *CELP* framework. This has resulted in the *TEWA component scaling very well and a speedup of over fifteen times was measured* on an 8-core system with hyper-threading.

- *Allow for time critical features by cancellation of long running computations.* This is especially important in *CORBA* based components where important events can be received from other components during a lengthy task. *By using the CELP framework, long running tasks can be canceled fast and safely, allowing for important time critical features*. This is seen in the parallelized *TEWA* component where a cease fire event from other another component may cancel a running task, like weapons allocation, very fast. This adds much better determinism to the component than just waiting for a `synchronized` block until it unblocks, which was the case in the sequential *TEWA*.

- *Educate for concurrency*. Implementation of concurrent software varies a lot from traditional sequential software and developers need to be aware of the tools and design patterns available. To be able to write concurrent software that scales well to multi-core architectures in a safe way, education is needed. Otherwise the result might be poor scalability and safety issues.

It is important to take extra note about the last point regarding education, since *developing concurrent software requires a new way of thinking for the developer*. This regards issues such as how shared data is handled in the Java Memory Model and when loops should be run in parallel instead of sequential, all these issues *requires education* to be handled correctly. Since developing concurrent programs with scalability properties is the only way to make use of modern multi-core processor architectures, *the cost required for such education is likely to be covered by the features of the new concurrent software*. For example, as in the parallelized *TEWA* component, where the number of air- and firing units that the system can handle has been greatly increased, compared to the sequential version. It should also be noted that if using modern concurrency tools instead of older synchronization, like `synchronized` blocks, correctly, the components are likely to be *faster even in the single CPU case*. This can also be seen in the *TEWA*, which is twice as fast in the single *CPU* case. The parallelized *TEWA* component also uses the *CELP* framework which, to be used efficient and safe, requires that it is used with good knowledge of how concurrency in Java works, and thus requires education to be used as well.

10 Future Work

The main future work should be done on the *CELP* framework and verifying the usage of it. Possible extensions could be:

- Allow for a single `CriticalTaskExecutor` to be shared among threads, e.g. two concurrently running main loops, safely. That is, to make it thread safe. Currently, it is only supported to be used in one thread safely.
- Limit the usage of the important classes in the *CELP* framework, to avoid problems with shared data when used in a multi-threaded context.
- Implement critical tasks that can not be interrupted and implement different priorities, similar to the event handling.
- Write more tests for validation of the framework.
- Extend the benchmarking utility for more customizable benchmarking and more output.

The *TEWA* implementation does not execute on its own in the giraffe system and if an overall performance increase is desired the rest of the components in the *Giraffe AMB* should be parallelized as well. This could require some kind of interface between the different components to tweak the concurrency setting to maximum.

The *TEWA* component is not designed for concurrency to start with and a lot performance and security could be gained if it would be rewritten with that i mind from the start. If a lot more cores are added to the platform some of the less performance intense loops could be required to parallelize for the system to scale as much as possible.

References

[1] Brian Goetz, *Java Concurrency in Practice*, 2006.

[2] Clay Breshears, *The Art of Concurrency*, 2009.

[3] J Gosling, B Joy, G Steele, G Bracha *The Java Language Specification, Third Edition*, 2005

[4] I.T. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, Reading, MA (1994).

[5] Danny Dig, John Marrero, and Michael D. Ernst, *Refactoring Sequential Java Code for Concurrency via Concurrent Libraries*, 2008

[6] WN Scherer III, D Lea, ML Scott, *Scalable synchronous queues*, 2006

[7] Maged M Michael, Michael L. Scott, *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*, Department of Computer Science University of Rochester, 1996

[8] Brian Goetz, *Java theory and practice: Stick a fork in it, Part 1*, IBM developerWorks, 2007

[9] `java.util.concurrent` package source(including Java 6 and 7 updates), *http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent*.

[10] Doug Lea, A fork join framework for java, *Proceedings of the ACM 2000 conference on Java*, State University of New York at. Oswego, 2000.

[11] Doug Lea, *Concurrency: where to draw the lines*, research.ibm.com, 2004

[12] The Java HotSpot Options Page, internet: *http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp*.

[13] The Java HotSpot Documentation, internet: *http://java.sun.com/javase/technologies/hotspot*

[14] Tuning Java GC , internet: *http://java.sun.com/javase/technologies/hotspot/gc*

[15] FindBugs Webpage, internet: *http://findbugs.sourceforge.net*

[16] Concurrency JSR-166 Interest Site, internet: *http://gee.cs.oswego.edu/dl/concurrency-interest/*.

[17] API specs for package jsr166y, internet: *http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/*.

[18] Java SE 6 API, internet: *http://java.sun.com/javase/6/docs/api/*.

[19] Java Specification Request(JSR) #166: Concurrency Utilities, internet: *http://www.jcp.org/en/jsr/detail?id=166*.

[20] Java 7 concurrency package(jsr166y) source, internet: *http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/jsr166y/*.

[21] Java 5-6 concurrency package(jsr166x) source, internet: *http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/jsr166x/*.

[22] Setting up a JMX connection, internet: *http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html*.

Appendices

A List of Tools and Libraries

Tools		
Name	**homepage**	**Desc.**
Eclipse IDE for Java	*eclipse.org*	Integrated Development and debugging environment.
Doxygen	*doxygen.org*	API and documentation generation. Can be used together with *graphviz*.
Graphviz	*graphviz.org*	Graph generating software, can be used with Doxygen to generate method
Eclipse Omondo	*ejb3.org*	UML generation and manual design, integrated into Eclipse.
FindBugs	*findbugs.sourceforge.net*	Automatic bugs detection tool.
JConsole	*java.sun.com*	Monitor applications running on Javas virtual machine. Included into Java JDK since version 5.0
VisualVM	*visualvm.dev.java.net*	Monitor and visualize Java applications. Includes support for profiling.
Perfmon		Montor Windows operative systems for data such context switching frequencies.

Table 5: Table of tools

Libraries		
Name	**homepage**	**Desc.**
jsr166y.jar	*gee.cs.oswego.edu/dl/concurrency-interest/*	Java 7 concurrency package.
JFreeChart	*www.jfree.org/*	Open source chart generation library.
Batik	*xmlgraphics.apache.org/batik/*	SVG graphics generation library.

Table 6: Table of libraries

B CELP Framework Class Diagram

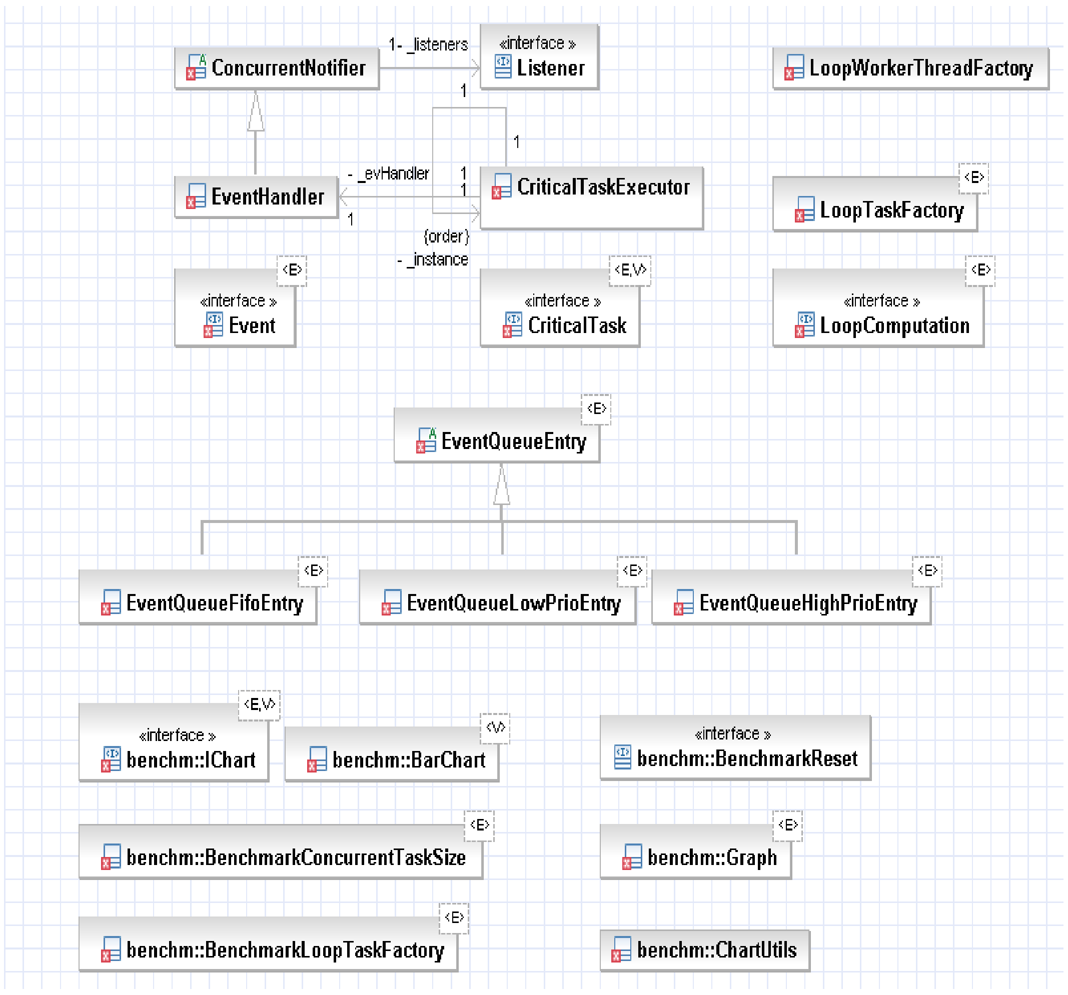
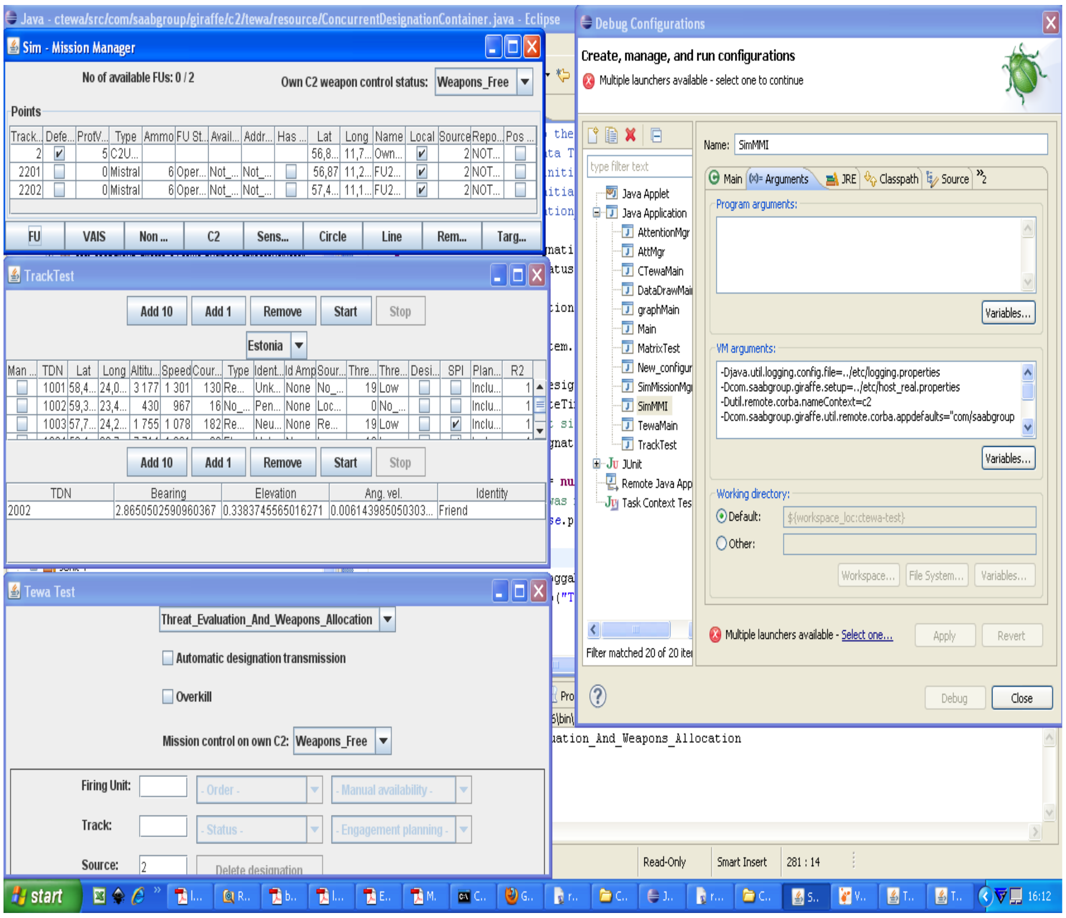


Figure 26: A class diagram overview of the CELP Framework.

C Windows Simulation Environment



Java - ctewa/src/com/saabgroup/giraffe/c2/tewa/resource/ConcurrentDesignationContainer.java - Eclipse

Sim - Mission Manager

No of available FUs: 0 / 2

Own C2 weapon control status: Weapons_Free

Points

Track...	Defe...	ProtV...	Type	Ammo	FU St...	Avail...	Addr...	Has ...	Lat	Long	Name	Local	Source	Repo...	Pos ...
2	✓	5	C2U...						56,8...	11,7...	Own...	✓	2	NOT...	☐
2201	☐	0	Mistral	6	Oper...	Not_...	Not_...	☐	56,87	11,2...	FU2...	✓	2	NOT...	☐
2202	☐	0	Mistral	6	Oper...	Not_...	Not_...	☐	57,4...	11,1...	FU2...	✓	2	NOT...	☐

TrackTest

Estonia

Man ...	TDN	Lat	Long	Altitu...	Speed	Cour...	Type	Ident...	Id Amp	Sour...	Thre...	Thre...	Desi...	SPI	Plan...	R2
☐	1001	58,4...	24,0...	3 177	1 301	130	Re...	Unk...	None	No_...	19	Low	☐	☐	Inclu...	1
☐	1002	59,3...	23,4...	430	967	16	No_...	Pen...	None	Loc...	0	No_...	☐	☐	Inclu...	1
☐	1003	57,7...	24,2...	1 755	1 078	182	Re...	Neu...	None	Re...	19	Low	☐	✓	Inclu...	1

TDN	Bearing	Elevation	Ang. vel.	Identity
2002	2.8650502590960367	0.3383745565016271	0.006143985050303...	Friend

Tewa Test

Threat_Evaluation_And_Weapons_Allocation

☐ Automatic designation transmission

☐ Overkill

Mission control on own C2: Weapons_Free

Firing Unit: - Order - - Manual availability -

Track: - Status - - Engagement planning -

Source: 2 Delete designation

Debug Configurations

Create, manage, and run configurations

Multiple launchers available - select one to continue

Name: SimMMI

Program arguments:

VM arguments:

```
-Djava.util.logging.config.file=../etc/logging.properties
-Dcom.saabgroup.giraffe.setup=../etc/host_real.properties
-Dutil.remote.corba.nameContext=c2
-Dcom.saabgroup.giraffe.util.remote.corba.appdefaults="com/saabgroup
```

Working directory:

Default: ${workspace_loc:ctewa-test}

Other:

Multiple launchers available - Select one...

Figure 27: A print screen showing some of the running applications that simulate the *TEWA* component on a Windows machine.