Comparison of Concurrency Frameworks for
the Java Virtual Machine

Thomas Georg Kühner

vorgelegt am
25. Oktober 2013
VS-B13-2013

Gutachter
Prof. Dr. Frank Kargl
Contents

1. Introduction 1
   1.1. Motivation ................................................. 1
   1.2. Scope of this Thesis ..................................... 2
   1.3. Methodology .............................................. 2
   1.4. Road Map for this thesis ................................. 2

2. Basics and Background about Concurrency and Concurrent Programming 3
   2.1. Terminology ............................................... 3
   2.2. Small History of Concurrency Theory ........................ 5
   2.3. General Programming Models for Concurrency .................. 6
   2.4. Traditional Concurrency Issues ............................. 7
       2.4.1. Race Condition ....................................... 7
       2.4.2. Dead-Lock ........................................... 8
       2.4.3. Starvation .......................................... 8
       2.4.4. Priority Inversion ................................... 9
   2.5. Summary .................................................. 9

3. JVM Fundamentals 11
   3.1. Motivation for the JVM .................................... 11
   3.2. Approach to Concurrency .................................. 12
   3.3. Java Memory Model ....................................... 13
   3.4. Summary .................................................. 14

4. Models and Concepts for Concurrent Programming 15
   4.1. Models ..................................................... 15
       4.1.1. Shared State and Threads ............................. 15
       4.1.2. Actor Model ......................................... 16
       4.1.3. Dataflow ............................................ 17
       4.1.4. Sequential Processes ................................. 18
   4.2. Concepts .................................................. 18
   4.3. Summary .................................................. 20

5. Design of the Comparison 23
   5.1. Levels of Measurement .................................... 23
   5.2. Qualitative Properties .................................... 23
   5.3. Quantitative Properties ................................... 24

6. Generic concurrency problems 25
   6.2. Specific Problems for Comparison .......................... 28
## Contents

6.2.1. Counting Server and Computation Intensive Server ........................................ 28
6.2.2. Transactions between Spaceships ................................................................. 29
6.2.3. Frequency of Visits .......................................................................................... 30
6.3. Summary ............................................................................................................... 31

7. Overview of Frameworks .......................................................................................... 33
7.1. Requirements of the Frameworks ........................................................................... 33
7.2. JSCOOP ................................................................................................................ 33
7.3. Guava .................................................................................................................... 34
7.4. Disruptor ............................................................................................................... 34
7.5. GPars .................................................................................................................... 35
7.6. Vert.x .................................................................................................................... 36
7.7. Akka ..................................................................................................................... 36
7.8. Summary ............................................................................................................... 37

8. Qualitative Comparison ............................................................................................ 39
8.1. Available Programming Languages ....................................................................... 39
8.2. Approaches to Concurrency .................................................................................. 40
8.2.1. Approach of Vert.x ......................................................................................... 40
8.2.2. Approaches of Gpars ....................................................................................... 41
8.2.3. Approach of Akka ........................................................................................... 45
8.3. Akka actors vs GPars actors .................................................................................. 48
8.4. Akka Dataflow vs GPars Dataflow ...................................................................... 49
8.5. Vert.x Multireactor Pattern vs Actor Model ......................................................... 50
8.6. Usability ............................................................................................................... 50
8.7. Scalability/Distribution ......................................................................................... 52
8.8. Summary ............................................................................................................... 55

9. Implementation ......................................................................................................... 57
9.1. Requirements and Specifications for Transactions between Spaceships ............... 57
9.1.1. Implementation with Java ................................................................................... 58
9.1.2. Implementation with Vert.x ............................................................................. 58
9.1.3. Implementation with Akka ................................................................................ 59
9.1.4. Implementation with GPars ............................................................................. 60
9.2. Requirements and Specifications for Frequency of Visits .................................... 61
9.2.1. Implementation with Java ................................................................................... 61
9.2.2. Implementation with Akka ................................................................................ 63
9.2.3. Implementation with GPars ............................................................................. 63
9.3. Requirements and Specifications Counting Server and Computation Intensive Server 64
9.3.1. Implementation with Java ................................................................................... 65
9.3.2. Implementation with Akka ................................................................................ 66
9.3.3. Implementation with Vert.x ............................................................................. 66
9.4. Summary ............................................................................................................... 67

10. Quantitative Comparison ....................................................................................... 69
10.1. Benchmarking on the JVM .................................................................................. 69
10.2. Actual Benchmarks .............................................................................................. 70
10.3. Results ............................................................................................................... 72
10.4. Summary ............................................................................................................... 75
1. Introduction

1.1. Motivation

Research on the field of concurrency goes back to the early 1960s. According to "The origin of concurrent programming", "The development of concurrent programming was originally motivated by the desire to develop reliable operating systems. From the beginning, however, it was recognized that principles of concurrent programming have a general utility that goes beyond operating systems"[22]. The consequences of this statement, can be recognized today. Usually all modern programming languages allow to develop concurrent programs. Yet, concurrent programming is still a topic programmers often struggle with. Even worse, bugs caused by concurrent issues can draw fatal consequences. They can cause a total failure of an application. Furthermore, debugging can be like looking for a needle in a haystack.

In terms of hardware, there was a huge paradigm shift. Over many decades new computer chips improved by having more transistors while requiring less space and increasing the clock frequency. At some point, chips vendors reached the physical limits. Increasing the clock frequency of one processor caused too much heat. In order to keep increasing the number of transistors, they decided to put multiple processors on one chip. These share a common memory, enabling to execute parts of concurrent software simultaneously. In the past, only few people from the industry and research had computers with such an architecture. But today, the common run of mankind owns computers with multiple cores. Thus, it is important to benefit from this additional power when developing software for daily use. Programmers should be aware of handling concurrent programming more than ever before. On the other hand, the variety of hardware devices on which software is expected to run on is huge today. Especially, the demand for mobile devices increased enormously. In consequence to the large number of different operating and hardware, platform independence has taken on great significance. A popular solution to provide platform independence is the Java Virtual Machine of Oracle. Regarding the need of portable and concurrent applications, a broad support for concurrency appeared for the JVM. Besides its native support for concurrency, multiple other approaches are offered in the form of frameworks. The reason for that is that its native approach to concurrency with languages such as Java is often seen as too complex and error-prone for building large applications. Frameworks try to tackle that by providing appropriate abstractions and applying other concurrency models.

However, developers who decided to use the JVM to create concurrent software have the agony of choice. Several frameworks exists but there is no reference which compares them. This thesis shell counteract that circumstance. It intends to figure out what frameworks exist, pick out the promising ones and compare them based on qualitative and quantitative properties.
1. Introduction

1.2. Scope of this Thesis

Each comparison requires a basic set of elements to be compared. Thus, picking out promising concurrency frameworks forms the basis of this thesis. Dedicated frameworks are compared based on qualitative and quantitative properties. Measuring quantitative properties is done by evaluating implementations of specific problems. In order to define these problems, essential characteristics for concurrent applications are deduced. In multiple benchmarks, these implementations are evaluated to retrieve quantitative results. On top of the results of the comparison, it is concluded which frameworks suit in which situations.

This thesis does not replace detailed references how to program with one of the frameworks. Moreover, it offers no guideline for developing whole applications. Instead, it focuses on describing advantages, disadvantages and limits of models and concepts provided differently by each framework. Generally, most of the time the pragmatic topic of concurrent programming is targeted rather than concurrency theory.

1.3. Methodology

Various books and public research papers were employed to provide the basic theoretical background which is required. At certain points there was a lack of references. For instance, defining what essentials characteristics of concurrency problems are in practice. These ideas were developed single-handed in discussions with my adviser Benjamin Erb. Any other information stems from public documentation, books or slides available for each framework. Additionally, the contact to the developers and communities was used whenever all available references failed to answer a question.

1.4. Road Map for this thesis

This thesis begins with several basic introduction about concurrency in chapter 2. Then we have a look on concurrency relating to the Java Virtual Machine in chapter 3. In chapter 4, models and concepts of concurrent programming, important in the scope of the thesis, are introduced. Next, the actual design of the comparison which qualitative and quantitative properties are regarded can be read in chapter 5. Chapter 6 intends to determine specific concurrency problems which are to be implemented. The decisions which frameworks are evaluated can be found in chapter 7. All qualitative properties which have been figured out are discussed in chapter 8 which is followed by an description how the specified concurrency problems are implemented in chapter 9. All results as well as the benchmark to gain them is explained in chapter 10 resulting in a final conclusion in chapter 11.
2. Basics and Background about Concurrency and Concurrent Programming

This chapter gives a short overview over concurrency. Therefore, the chapter contains a brief introduction into the history of concurrency theory as well as an overview about concurrent programming. Additionally, confusing terms and discussions are explained and summarized. Finally, the basic concurrency issues are explained.

2.1. Terminology

The main contribution of this section aims to prevent any misunderstanding. Therefore, it is briefly explained what confusing terms and distinctions mean in the scope of this thesis.

Definition of Concurrency

Concurrency describes a property of programs or systems and indicates that the pending flow is divided into multiple portions. The reason for that division is to improve the performance which is realized by reducing latency, hide latency and increase the systems throughput [19]. The actual execution order of concurrent tasks is generally non-deterministic. In consequence, mechanisms such as synchronization and coordination are inevitable to handle non-determinism.

Process vs Threads

Referring to operating systems, a process represents a specific environment with its own identifier, virtual memory, security context and priority etc. Additionally, each process contains at least one thread of execution. In contrast, threads are placed inside processes and share the same memory space of the surrounding process. The operating systems is responsible for scheduling processes as well as threads inside the processes.

When talking about concurrency, especially about models of concurrent programming, the terms processes and threads are often interchangeable. That means if we talk about processes later in a concurrency model, in most of the cases actually threads are meant because one instance of a
Java Virtual Machine (JVM) is always running inside one process. Concurrency on the JVM is provided by threads as described in 3.2.

**Concurrency vs Parallelism**

Concurrency and parallelism are often confused because both terms are often used as synonyms although they do not mean the same. The former describes a property of a program or system in which tasks are executed concurrently. That means the order of execution of tasks underlies a certain nondeterminism at run time. Tasks can be executed sequentially or simultaneously depending on the underlying hardware and the scheduling resulting in possible hazardous interleaves. Therefore, concurrency is even possible on a single processor. In contrast, parallelism indicates that tasks are really executed simultaneously at run time. Without proper hardware such as multi-core, multiprocessor or distributed systems parallelism is impossible.

**Concurrent Programming vs Parallel Programming**

Concurrent programming is mainly about handling the non-deterministic control flow of a program. Therefore, concurrent programming has to tackle the problem of hazardous interleaving which may cause various problems such as dead lock or starvation as described in 2.4. Despite of preventing these classic concurrency problems, the main goal of concurrent programming is to increase the throughput and decrease or hide latency [19]. On the contrary, the main goal of parallel programming is to increase the throughput without the loss of a deterministic control flow. To achieve this goal, the creation of parallel executable work, their distribution and finally their reduction are the main issues parallel programming tackles.

**Events vs Threads**

In an event-driven system the execution flow is controlled by events. That means, certain parts of the system handle certain sets of events and react by processing them and creating new events. The underlying scaffolding for event-driven systems is called event-driven architecture and specifies precisely how events are distributed, how parts of the systems can listen to events and how events are compounded or classified etc. A message-passing system resembles an event-driven system and both terms are sometimes used as interchangeable. The term process-based system is equivalent to thread-based system most of the time.

In 1978, Lauer and Needham tried to address a persistent discussion about the difference of message-passing systems and shared-state systems. Although the basic approach to concurrency on the JVM is thread-based, many frameworks build an event-driven architectures on top of that as described later. In order to resolve the discussion, Lauer and Needham created a mapping between message-passing systems and process-based systems. The main statement of their mapping was that the models are duals of each other, logically equivalent and provide equal performance. Although the mapping lacks regarding some properties when it is applied to event-driven systems, its correctness is accepted. Thus, it is recommend to choose between both systems depending on the appropriateness for the given use case [19].
2.2. Scalability

Scalability describes "the ability of a computer application or product (hardware or software) to function well as it (or its context) is changed in size or volume in order to meet the requirements of an user; and the ability not only to function well in the rescaled situation, but to actually take full advantage of it, for example if it were moved from a smaller to a larger operating system or from a uniprocessor to a multiprocessor environment [9]." Referring to the underlying hardware, it is distinguished between two ways of scaling which are Scale-up and Scale-out. The former is a solution in which an upgrade of a single machine targets to increase the overall performance of an application. In order to achieve that, usually the amount of processors as well as the amount of working memory they share is increased. In contrast, Scale-out means to increase the number of interconnected computer where the application is running on. Now, we continue by having a short look on the history of concurrency theory.

### 2.2. Small History of Concurrency Theory

This section is intended to give a short overview of concurrency theory by classifying the different models which might help later to classify the models and concepts used by the frameworks described in chapter 4.

In the past, concurrency has become a separate area of research in computer science because of two issues. First, multiple flows of control have led to unwanted interleaving such as deadlock or starvation. The second reason is that concurrent systems are reactive. That means they are designed to handle an ongoing sequence of interactions with its environment. These systems such as operating systems or databases no longer acted in the common way as they did not terminate. Thus, the term of correctness had to be defined new because it was no more possible to just compare the output with the expected result.

Although there are many different models for concurrency they all have one assumption in common. This assumption is that in a concurrent system, parts of system behavior are atomic. According to "Strategic directions in concurrency research" [17], the models can be classified by three different properties:

- intentionality versus extensionality
- interleaving versus true concurrency
- branching time versus linear time

The main focus of intentionality, also called operational, is to describe the actions of a system using state and transitions. In contrast, extensional models which is often referred to as denotational describe what aspects of a system can be observed from the outside. For instance, Petri nets or Communicating Sequential Processes are intentional models. Petri nets describe a systems behavior through a graph with discrete states and transitions whereas Communicating Sequential Processes describes exactly the order of created events and processes which handle them and
elicit new events. Thus, both models meet the requirements of an intentional system because the explicitly describe the actions of a system using state and transition. Examples of extensional models are acceptance trees or failure sets. Both describe the reaction of a system after handling a certain sequence of events and actions. Thus the various outputs an observer can see is specified by this model.

The second characteristic of the models is whether they are interleaving or truly concurrent. "Interleaving models “reduce” concurrency to nondeterminism by treating the parallel execution of actions as the choice between their sequentializations" [17]. In order to avoid anomalies such as deadlock, starvation further mechanisms are imposed. Unlike interleaving, true concurrency describes the behavior of the system by the causality between the different events created by parts of the systems. Relating to the examples above, Communicating Sequential Processes is an interleaving model since it specifies a system by the order of events and processes handling them concurrently instead of sequentially. In contrast to that, Petri nets define the relations between parts of a system by the events connecting them.

Finally, the third characteristic dichotomy describes whether the concurrency model is based on a time branching models or linear time models. The former describes a process definitely by the sets of observable data, arisen in possible partial executions, whereas the latter focuses and stores the points where computation start diverging. There has been a extensive discussion about both models and which one is more appropriate in practice. According to Vardi [38], linear branching wins the discussion because it is just the more pragmatic approach in terms of his expressiveness, computational complexity and compositionality.

After we have learned the very basics about the history of concurrency history we will introduce the more pragmatic topic of concurrent programming. The main focus will lie on characterizing the different types of concurrent programming.

### 2.3. General Programming Models for Concurrency

According to “Concepts, Techniques, and Models of Computer Programming” [33] there are four different models for concurrent programming. An overview of the models and how they relate is shown in the figure 2.1 below.

**Sequential programming**

In sequential programming there is no concurrency at all. That means all operations of a program are executed in a fixed order. Moreover, there are two relaxed models which are still considered to be sequential since they are deterministic. The first one is "Order-determining" which has also a total order of execution but the order is hidden from the programmer. The second model is "Coroutining". In this model each program is responsible to explicitly pass around the control.

**Declarative concurrency**

In contrast to sequential programming, declarative concurrency is non-deterministic. The significant property of this model is "logical equivalence". That means, all executions lead to the
2.4. Traditional Concurrency Issues

This section deals briefly with the classic concurrency problems because these problems are often referenced in this thesis and any misunderstanding must be prohibited. Each problem can occur depending on the employed concurrency model as well as the practical implementation of these models.

2.4.1. Race Condition

A race condition describes a situation in which the result of a computation depends on the timing of concurrent actions. Depending on which actions are performed first, or figuratively
which actions win the race, the results might differ. In worst case, even unwanted effects are the consequence. The classic example to illustrate this is a "lost update". Imagine, there are two threads A and B sharing access to a counter they want to increment both. To increment a counter, three machine instructions are required. First step is to extract the value of the counter from the main memory. Next, the value is incremented by one and finally it is written back to the main memory. Since thread A and B run concurrently, the three actions from A can be mixed in any order with the one of thread B. In worst case, A could not write back the value before B read it or the other way round. In that case, B have read an obsolete value from the main memory because thread A already increased the value but had not the chance to write it back so far. In consequence, A and B will both write back the same value, so one update is lost. One way to prevent such inconsistency caused by race condition, is employing locks to guarantee critical sections such as increment a counter cannot be entered by more than one thread. However, this may lead to another concurrency issue we discuss in the following section.

2.4.2. Dead-Lock

The dead-lock is the most popular but also most harmful problem. It describes a situation in which two or more processes wait for each other to get access to resources locked by the opposing process. In consequence, no process makes any further progress and thus the program is "dead". Handling dead-lock can be split up into three parts:

**Detection and Recovery** In this approach programs or systems let dead-lock just happen. If one occurs the dead-lock is recognized and the system then recovers from it. For instance, the operating system cancels one process.

**Prevention** Prevention is a term representing several different techniques which do not admit any dead-lock. To achieve that locks must be demanded always in a certain order or processes must expose their future behavior, for example by flow charts.

**Avoidance** Dead-lock avoidance is a technique which checks each transition of a process if it could possibly lead to deadlock. Then only processes marked as safe are actually executed. In practice, the lack of required information often prohibits this technique.

Another a relative of the dead-lock is a so called live-lock. Similar to the dead-lock processes in a live-lock do not make any progress but the processes do not stuck in one specific state. Instead, the processes continuously change their internal state. For instance, the processes recognize that they cannot make progress, in consequence, they release locks and request new ones but if the other processes act the same way the locks are only passed around but no process can get all the locks which are required simultaneously.

2.4.3. Starvation

The problem of starvation describes a situation in which a process does not get the resources it needs. In general, two different kinds of starvation exist:
2.5. Summary

**CPU Starvation**  It occurs when the scheduling of an operating system makes "bad" decisions and neglects a process or thread so that this process never gets any execution time on the CPU.

**Lock Starvation**  It is similar to a dead-lock as a process suffering from lock-starvation cannot get the required locks to progress. However, the difference to the dead-lock is that the rest of the program or system can still make progress and only certain processes are affected.

2.4.4. Priority Inversion

Priority Inversion can be ascribed to CPU starvation. It occurs if a process with lower priority influences the time a higher prioritized has to wait for a specific resource. For a better illustration of priority inversion, here an example stated similarly in "Operating System Concepts" [36]:

There are three processes with the numbers 1,2,3 which also indicates their priority. At the beginning process 1 is running and owning a resource process 3 wants access to. In consequence, process three has to wait until process 1 releases the resource. But then process 2 gets runnable and preempts process 1. Hence, the release of the resource depends on the running time which process 1 still needs but the running time process 1 gains is influenced by the process 2 which "steals" CPU-time. In consequence, the priority of process 2 and 3 are inverted because process 2 influences when process 3 can run. Avoiding Priority Inversion is possible by priority-inheritance. This technique imposes a process, owning a resource which is demanded, to become temporarily a higher prioritized process. In this example, process 1 would temporarily gain priority 3 and prevent to be preempted by process 2.

2.5. Summary

After this chapter it is obvious that concurrency is a broad field of research. To sum it up, concurrency and parallelism are not equivalents. Moreover, there are many different models for concurrency and their overall similarity is that each of the models assumes that parts of system behavior are atomic. Relating to concurrent programming, message-passing concurrency and shared-state concurrency are the two means of choice. Finally, the classic concurrency issues explain why concurrency is so difficult but also hazardous if developer are not careful enough. In the next chapter we discuss the JVM itself and its approach to concurrency.
2. Basics and Background about Concurrency and Concurrent Programming
3. JVM Fundamentals

This thesis only regards concurrency frameworks which can be used on the JVM. Hence, this chapter intends to provide necessary basics which should be known about the JVM in general and relating to concurrency. We begin describing what the JVM actual is, motivate its use for concurrent software and finally explain its approach to concurrency which requires also some background about the underlying memory model.

The Java Virtual Machine is an abstract machine designed to execute bytecode. This bytecode is generated by compiling languages such as Java, Scala and many more. After compilation the bytecode is interpreted which has made Java the reputation to be slow. To improve execution speed, modern compilers use just-in-time compilation. Just-in-time compilation is a technique which translates platform independent code into native machine code when the code is actually required. Thus, the compiler can exploit additional information gathered during run-time to enhance optimization.

Technically speaking there is not one version of the Java Virtual Machine. In fact there exists only a specification "The Java Virtual Machine Specification Java SE 7 Edition" [30]. But there are multiple different implementations of this specification such as the two popular ones JRockit and HotSpot. The different implementations are a consequence of the circumstances the specific JVMs are designed for. For instance, the JRockit was mainly designed to run server software. Independent of the actual implementation of the JVM, it suits well to build concurrent application as described in the next section.

3.1. Motivation for the JVM

Programming concurrency is possible in various languages, each with native support and libraries as well as additional support by external libraries and frameworks but there are many reasons why the JVM outperforms other platforms in terms of concurrency.

**Platform independence** The JVM provides platform independence which is a crucial property relating to distribution as it is very costly and time-consuming to deploy platform dependent software on different machines.

**Execution speed** Former drawbacks such as a slow execution speed are no longer up to date. Especially in Java, sophisticated optimizing through the compiler improved the performance significantly [16].
3. JVM Fundamentals

**Language agnostic** The specification of the JVM is independent from any programming language. Besides Java, the JVM supports a bunch of different programming languages. The languages can easily be mixed within a project and therefore it is always possible to use the language meeting the requirements of a given problem.

**Wide support** Another reason for the JVM is the wide support to programming in general mirrored by the amount of libraries, frameworks, IDEs for the available languages.

**Active development** The JVM and especially Java are very popular [11]. Thus, further development will improve the JVM and Java even more.

### 3.2. Approach to Concurrency

Before explaining the approach of the JVM to concurrency it is necessary to know how the JVM interacts with the operating system (OS). For each started Java process an additional JVM is started on a separate OS-process. Each java process consists of several threads, mapped to OS-threads, which can be either custom Java threads or internal threads such as code-optimizer or garbage collector. At the time of the first JVM implementations, Java threads could be either green threads or native threads. Green threads are scheduled by the JVM itself, not the OS, whereas native threads are mapped to OS threads as mentioned before. Because of the drawback such as the lower performance because green threads cannot exploit multi-core architectures green threads were no longer supported in later implementations of the JVM [8].

Since the threads of one JVM instance are located in one process the approach to concurrency is quite obvious. All data of the JVM running on this process is stored in a shared memory which can be accessed by all threads inside this process. This model for programming concurrency is called shared-state concurrency as mentioned before in section 2.1. In order to avoid inconsistency and other classic concurrency problems as described in section 2.4 proper synchronization is inevitable. For that issue the JVM offers monitors to provide thread-safe critical sections. A monitor is an object which limits the access to its properties by mutual exclusion. That means only one thread can have access to the properties (e.g. methods) of a monitor. This kind of synchronization can be done explicitly by the JVM-instructions “monitorenter” or “monitorexit” specified in “The Java Virtual Machine Specification Java SE 7 Edition” [30] or implicitly by setting a flag in the run-time constant pool. All methods are stored in this pool and before each method invocation of a thread this flag is checked either granting or denying access. "One peculiarity of Java monitors is that they are recursive: it is legal for a thread to re-acquire a monitor that it has already locked."\(^1\) Otherwise, a thread could deadlock itself, for instance when calling a synchronized recursive function.

Apart from that, it must be stressed that even if the basic approach to concurrency of the JVM and Java is shared-state concurrency other models such as message-passing can still be employed on top of that.

\(^1\)from lecture on http://faculty.ycp.edu/~dhovemey/spring2011/cs365/lecture/lecture17.html
3.3. Java Memory Model

When programming concurrent software for the JVM, it is essential to understand the Java Memory Model (JMM) in order to prevent mistakes. The JMM is similar to an abstract symmetric multi-processing machine. That means the threads have access to the same main memory but each thread has also his own local cache. Furthermore, the JMM is build on top of the underlying hardware memory model but from the perspective of a java programmer only the JMM needs to be understood. More precisely, at least the three main issues atomicity, ordering and visibility, listed in the specification "The Java® Language Specification" [27], should be known.

Atomicity is a property of an instruction which indicates that this instruction cannot be interrupted. According to the specification of the JMM, the access to variables of primitive type and references is atomic except for the types long and doubles. The access to volatile variables inclusive long and double is also atomic. However, in case of a volatile reference, only the access to the reference is atomic, not to the whole object. Besides that, there are some objects in the package java.util.concurrent.atomic and operations in java.util.concurrent which guarantee atomic access.

Additionally, the JMM specifies how the sequence of instruction can be ordered. In general, the JMM allows various re-ordering due to optimization issues. From the view of a particular thread the instructions seem to be executed sequentially since reordering is only allowed if results stay the same. In consequence, another thread may see the instructions made by another thread in a different sequence as they were defined.

Finally, visibility describes when changes made to data are visible for other threads. Generally, each threads works on his cache updated from the main memory by a refresh or written back by a flush. The JMM specifies when these two operations are executed as summarized in table 3.1.

<table>
<thead>
<tr>
<th>refresh</th>
<th>flush</th>
</tr>
</thead>
<tbody>
<tr>
<td>start of thread</td>
<td>end of thread</td>
</tr>
<tr>
<td>gain a lock</td>
<td>release lock</td>
</tr>
<tr>
<td>read volatile</td>
<td>write volatile</td>
</tr>
<tr>
<td>read atomic variables</td>
<td>write atomic variables</td>
</tr>
<tr>
<td>first write of final variable</td>
<td>first read of final variable</td>
</tr>
</tbody>
</table>

Figure 3.1.: Summary when cash refresh and flush is triggered in the JMM

2source: http://www.angelikalanger.com/Articles/EffectiveJava/38.JMM-Overview/38.JMM-Overview.html
3.4. Summary

When talking about the JVM it is important to know that the JVM is only an abstract Machine Specification of a machine executing bytecode. Furthermore, several implementations of this specification exist which not only comply the JVM Specification but also the Java Language Specification including the Java Memory Model. This memory model describes the guarantees a developer can expect relating to atomicity, ordering and visibility which are a essential knowledge when programming concurrent software. The entire design of the JVM, the bytecode as well as the JMM is built around the notion of multiple threads sharing access to a common memory. Thus, much effort in the JVM targets to protect shared state with mechanisms such as Monitors. Nevertheless, other concurrent programming models such as message-passing concurrency can still be exploited on top.
4. Models and Concepts for Concurrent Programming

In this chapter the different models and concepts for concurrent programming will be introduced and briefly summarized. However, it is noteworthy that this chapter will not deal with all existing models. Instead, it will deal with the most important ones provided or used by the compared frameworks. First, several models are explained, then some concepts. The difference between both terms is, when we speak about concepts, we mean a general notion to tackle a specific problem. Programming models exist for the same reason but the scope of problems they intend to solve is more spacious.

4.1. Models

The models for concurrent programming which are explained in this section are shared state and threads, the actor model, dataflow and communicating sequential processes. All introduced models are provided at least by one of the frameworks discussed in this thesis.

4.1.1. Shared State and Threads

As mentioned in 3.2, shared state and threads is the basic approach to concurrency on the JVM. In each thread, the execution follows an imperative sequential style. Expressing tasks as a sequence of instructions is often natural for programmers. The model introduces real independent execution of threads which share access to the same memory. In consequence, a non-deterministic interleaving between the threads is the main characteristic of shared state and threads. With this model concurrent programming can get quite difficult because you need to reason all possible interleaving. For any of them, your program must produce correct results and not suffer from inconsistency. Reasoning all interleavings, however, is inconceivable because of its complexity. For that reason, additional mechanism such as locks, monitors and transactions are used.

Locks ensure that a specific section of code is only accessed by one thread at the same time, also known as mutual exclusion. With locks, any instruction can be implemented as it is atomic. Monitors extend locks by providing coordination between threads. Each monitor controls a set of waiting threads which want to gain access to the instructions protected by the monitor. If a thread gains the monitor lock, it can call wait inside the monitor to make the thread be placed back in the waiting set. For instance when the queue is full, the thread cannot push further elements...
on it until another thread pulled some elements. In addition to wait, threads can call notify to wake up a thread of the waiting set and let it try again to gain the monitor lock. Although these techniques avoid to reason interleaving, they are still very error-prone. If locks are gained in different orders by various threads, a deadlock can occur very fast. Furthermore, if locks are too coarse-grained others threads are prohibited from running simultaneously. On the other hand, too fine-grained locks are hazardous since it is complex to think through if more interleaving still produces correct results.

In terms of debugging, with the callstack of each thread, the programmer has access to lots of information to locate errors. Though, there are also two big drawbacks referring to bugs. On the one hand, locking is difficult to use and limits the capability of parallel execution when several threads need to wait because of locks. On the other hand, once a concurrency issue such as a deadlock occurs, it is very hard to detect it. This is because all interleaving is non-deterministic, thereby deadlocks are also non-deterministic. Thus, it is difficult to reproduce and locate them. Apart from debugging, the model is generally not designed to run distributed. On top of each node of a distributed system you can use this model. But in general, distribution makes locking even more difficult.

### 4.1.2. Actor Model

One instance of message-passing concurrency is the actor model. It picks up the basic assumption about concurrent systems that some parts of system behavior are atomic [39, 20]. An actor is a representative of a small part of system behavior. Hence, the actor model is particularly suitable to describe an inherently concurrent system. Each actor can be seen as a separate object encapsulating an internal state. To interact with other actors, they send messages. It is important to emphasize, in its basic design, message passing is the only way to exchange information and there is no shared data. Every actor has a queue, similar to a mailbox, where messages arrive. The only task of an actor is, to process these messages. When processing a message, an actor can react by updating his internal state, sending new messages or creating new actors. Originally, all communication is performed asynchronously which means there is no guarantee in which order messages arrive at the receiver. A sender immediately continues execution and does not wait for any reply or reception of its message. Sometimes the basic guarantees of the actor model are extended by additional protocols to ensure a deterministic order of message delivery between two actors.

Referring to error-proneness, the actor model is more robust than shared state and threads. In theory, there should not occur dead locks since the basic notion of the actor model relinquishes locks at all. However, when actors are employed wrongly, situations with same effects as a dead lock can appear again. For instance, two actors discard all messages until they get a specific one they are waiting for but both are waiting for each other to send the message. Consequently, they stuck forever which like having a dead lock. Harmful interleaving access to the same data is prevented by actors due to hiding their internal state. Only the actor itself can modify its state and provide other actors required information about it. Even though the model is less error prone, in other areas it imposes additional effort. Already simple tasks, such as determining the end of execution is not trivial any more. In shared state and threads the execution implicitly is
finished when all threads terminate. In contrast, actors never stop until someone explicitly stops them.

Running on a distributed system, is what the actor system suits much better than shared state and threads. This is because it already is based on message passing. To communicate in a network, messages are the basic approach. Hence, it is more natural to use one way for communication in the entire system.

4.1.3. Dataflow

Dataflow belongs to declarative concurrency and is a programming paradigm which represents programs by a directed graph. In the graph, nodes can be seen as instructions whereas edges are the actual data. Similar to the transitions of a petri net, the instructions are executable as soon as the required data is available. Hence, allows that several instructions are executable at the same time which results in a high potential for parallel execution, even for a single machine instruction. Therefore, a lot of research and development tried to support dataflow through special hardware-architectures, but finally the best solution was to use a hybrid architecture between the Von Neumann architecture and special hardware support for dataflow [28].

Programming in dataflow is employed by using a special data structure called dataflow variable. Its special behavior is that the access to this variable is blocked until it is bound to a value for the first time. To be exact, dataflow variables can be partially bound multiple times. But a partial bind must not change the consistency of the variable. The value or partial values of a dataflow variable cannot be changed once they were set. dataflow variables are usually used in combination with streams to communicate between threads. A stream is an unbounded list of messages and its tail is a dataflow variable. On one side of the stream a single producer can bind the dataflow variable at the tail. On the other side, one to many consumer can pull variables from the stream as soon as they are bound to a value. Until that, the thread reading from the stream is blocked. This mechanism provides an implicit synchronization between threads as it is characteristic for declarative concurrency models. This mechanism is similar to the pattern known as "Producer Consumer Pattern" [33].

With regard to concurrency issues, dataflow brings along some interesting advantages. First of all, there are no race-conditions because variables can be set only once. The results are independent from the timing in which the variables are set. Every access will block implicitly until the variable is bound. Even better, all deadlocks are deterministic. If there is a deadlock in a dataflow program, the program will always deadlock. The reason for that is if threads do not bind variables the other threads are waiting for, they deadlock. But this binding a variable is always deterministic. If it is not, for example, when we bind a variable depending on random, this would not count as pure dataflow anymore. Thus, pure dataflow as a declarative concurrency model is limited in its expressiveness. Concepts such as exceptions or randomness do not exist in pure dataflow.
4. Models and Concepts for Concurrent Programming

4.1.4. Sequential Processes

Communication Sequential Processes (CSP) is a process algebra to describe processes operating independently from each other [25]. More specific, CSP offers few primitives to describe the whole behavior of a system including the communication with events between processes as well as required synchronization or the real independence of processes. Real independence means the processes can run in parallel without any synchronization or coordination.

The model allows processes to communicate among each other through message-passing. But this message-passing is completely synchronous which is a main difference to the actor model. As a consequence of synchronous communication, CSP is deterministic, thus all deadlocks, too. Though, applying non-determinism is possible by using alternatives. An alternative describes a non-deterministic choice of a process which of multiple events of its input channels it processes first. Nonetheless, even with alternatives deadlocks stay still deterministic since alternatives can only defer but not prevent their occurrence. Furthermore, processes are not addressed directly, instead each process has his own channel for communication as its interface to environment. The lack of direct addressing is another difference to the actor model.

Relating to debugging, CSP has two main advantages. As mentioned before, deadlocks are deterministic. Therefore it is easier to replicate and locate them. Another advantage is, there are several tools for automatic analyzing and checking correctness of a CSP program. For example, proofing the absence of deadlocks is possible with CSP and special software. Very likely, its rather mathematical formalism which needs some time to grasp is the reason why it is used rather rare in practice.

4.2. Concepts

Up to this point, we discussed several main models for concurrent programming. Now we introduce some concepts for concurrent programming. These are Futures, Active Objects, Agents, Software Transactional Memory and finally a pattern called reactor pattern.

Futures

Futures are a concept which was introduced to facilitate synchronization between dependent tasks [33]. A future is placeholder for the result of a computation. For instance, there is one thread invoking a method which is executed in another thread. As the result, the method immediately returns a future. This future serves as a reference to the result of the computation which has not completed yet. After this computation finished the result is bound to the future. Each future is bound only once in their lifetime. Accessing the result of a future causes the calling thread to block until the future is bounded. Thus, tasks can continue working and block eventually when they require the result of the future.

Active Objects

"Active objects combine the abilities of Object Oriented Programming and the abilities of message-passing concurrency [33]." Each active object consists of three parts. It has one object which is an
instance of a class defining its behavior. Another is an input stream storing incoming messages and finally there is a thread reading the messages. Sending messages to an active object happens asynchronously. That means, the active objects returns immediately, usually with a future. When the thread is processing a message, it invokes the demanded method on the encapsulated object. Any result, returned by the method call, is bound to the future. You can also use active objects synchronously. If sending a message to an active object does not return until the message has been processed it is considered as synchronously. However, a synchronous processing is usually not reasonable because you could invoke the methods of the object directly instead. Active objects would only add additional overhead then.

Agents

An Agent is a concept introduced by the programming language Clojure, a dynamic programming language for the JVM. In Clojure, agents are a programming construct, used to protect an individual storage against concurrent modification [1]. To change the state, protected by the agent, a caller must send a function to the agent. The agent sequentially executes all incoming functions and the return value of the function becomes the new state of the agent. Retrieving the state of an agent is done by a direct call to its value. In Clojure, reading the state is always immediately possible by multiple threads. In general, an agent shares many similarities with an active object. Invocations to change the state return immediately and their processing is postponed. Moreover, an agent keeps track of an internal object which is the state. Though, it also shows differences to active objects. For instance, an agent receives the methods he shell apply on its state which can be theoretically any possible function while active objects have a fixed amount of methods during runtime.

Transactional Memory and Software Transactional Memory

The original idea of a transactional memory stems from the concept of transactions known from databases. In practice, a transactional memory (TM) or software transactional memory (STM) is used for shared-state concurrency to replace synchronization through locking. Generally, a transaction is a sequence of instructions whose execution is always atomic and serialized. That means from the view of threads, changes made to data shared among them, are either performed completely or not at all. Additionally, the order of the changes made by a transaction is defined clearly.

Different between transactional memory and software transactional memory is that the former is implemented in hardware by extending cache coherence protocols whereas the latter is implemented in software in many variants. Most of the STM implementations use an optimistic approach for transactions. That means each transaction directly modifies data. In the case of interleaving through other transactions during this process, all preformed changes are rolled back. How they progress after rolling back, varies for each STM. Some do a limited number of retries before they finally throw an exception. Other implementations immediately employ locking to guarantee consistent data.

The main advantage pleading for the use of a TM or STM is the avoidance of dead locks. Research has proved that transactional memory based on software does not result in a performance boost compared to common fine grained lock techniques but due to the performance being not dramatically slower than common locking the advantages of a STM supersede [24, 19]. On the
contrary, there is a significant performance increase when it is based on hardware. Therefore, next processor generation Haswell of Intel already comes along with hardware-support to transactional memories [32].

**Reactor Pattern**

Up to this point, all discussed concepts tackled a specific type of problem. Now, we look at a programming pattern called Reactor Pattern. Although a pattern is something different than a concept but it is also no model for concurrent programming we still put it here.

First of all, the Reactor Pattern is one approach to implement event-driven architectures. An event driven architecture has two main characteristics. One is, it relinquishes the callstack and the second is, the control flow is not determined in advance. Instead, the control flow of a program is determined by the external events sent to the program or internals events exchanged between entities of a program [26, 19]. To be exact, the callstack is not missing totally as long as the used programming language offers one but it cannot be used to observe actions invoked by the exchange of events.

According to Schmidt, "The Reactor architectural pattern allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients [35]." In particular, this pattern decouples demultiplexing and dispatching of events from the application logic. The latter is represented by event handlers which are registered at the reactor. The reactor itself waits until the event demultiplexer signals the arrive of events. Available events are dispatched by the reactor to its appropriate handlers. Generally, the pattern is single-threaded by definition but it can also be used in multi-threaded environments. Then instead of one reactor, multiple reactor dispatch events received from a global eventbus.

The advantage of this approach is, the programmer does not need to care about the underlying mechanism of event dispatching. Nonetheless, handlers of different reactors are executed concurrently. Furthermore, it is guaranteed that event handlers are executed only by one thread, the corresponding reactor. Thus, there cannot occur concurrency issues inside the code executed by one reactor. With multiple reactors it is possible to implement coarse-grained concurrency. This pattern is especially useful, when a program must handle a lot I/O which can be done asynchronously [26, 19]. While the result event of an I/O operation is computed the reactor can dispatch other events. Consequently, costly context switches between threads are bypassed. If certain calls of event handler block, the whole reactor gets idle. Thus, all operations should be asynchronous or at least block only for short times.

**4.3. Summary**

Shared state and threads, dataflow, the actor model and CSP were the four different models for concurrent programming we discussed in this chapter. Shared state and threads tends to be error-prone if techniques for synchronization are applied wrong. Dataflow belongs to declarative concurrency. In this model any concurrency is implicitly handled through data itself, so it is less error-prone to concurrent issues at the expense of an limited expressiveness. Furthermore, we discussed CSP and the actor model belonging both to message-passing concurrency. The main
advantage of CSP is it can be formally proved to be correct in terms of concurrent issues such as the absence of deadlocks. However, its pure formalism to describe a system is rather too complex for daily production. Similar to CSP, the actor model also employs message-passing. When messaging-passing concurrency is applied properly it is way less prone to stuck an application like an dead lock does. However, the model cannot completely prevent from creating programs threaten by infinite waiting. Moreover, programmers used to imperative programming often need to adapt their way of thinking. Besides the models, we dealt with concepts to protect mutable state (agents and STM) and mechanisms useful for coordination and asynchronous programming (Futures and Active Objects). Finally, the reactor pattern was introduced which can be employed to develop asynchronous event-driven applications. Relating to concurrency, the pattern is ascribed to message-passing concurrency.
4. Models and Concepts for Concurrent Programming
5. Design of the Comparison

The following describes the concrete structure of the comparison and delivers a short introduction on levels of measurement. Yet, the decision which frameworks are evaluated is postponed to chapter 7. In short, the comparison evaluates qualitative as well as quantitative properties and the aim is to finally merge the results to decide when each concurrency frameworks is appropriate.

5.1. Levels of Measurement

Levels of Measurement deals with the different type of scales to measure. Either scales measure qualitative or quantitative properties. The difference between qualitative and quantitative is the former measures the existence or the distinctness of an abstract property such as "support". In contrast, quantitative measures the numeric characteristic of a property such as "execution time". Qualitative scale types are either a nominal scale or an ordinal scale. On a nominal scale you can only determine if a property maps to one of the values on the scale. For instance, which languages are supported by a framework. On the other hand, if additional distinction of one property is possible ordinal scales can express this. However, differences on a ordinal scale do not describe mathematical proportional distances. Marks in school, for example, are measured with ordinal scales but the mark A only expresses that a pupil scored more points than a student scoring a D but you cannot guess what their actual difference of points was.

Measuring quantitative properties, one can use interval scales or ratio scales. Both measure the exact characteristic of a property by ascribing a numeric value, so that proportional and absolute differences between the characteristics are reflected. Ratio scales differ from interval scales as the former indicate an natural zero point. For instance "lines of code" for a certain problem are measured on a ratio scale since there is the a natural zero point. Determining geographic coordinates is a use case for interval scales because there is not natural zero point.

5.2. Qualitative Properties

Before listing all qualitative properties we define the ordinal scale used for the comparison. If there is a qualitative property which can be measured on a ordinal scale the scale will consist of items from 1 to 5. A low score on the ordinal scale indicates the characteristic of the property is either low or bad depending on the property (see scale 5.1). In contrast, a high score indicates the property is very distinct. Though, one should notice, all properties measured on a nominal scale are facts whereas the other qualitative properties stem from my assessment. Thus, I will argue each of my assessments on ordinal properties.
5. Design of the Comparison

Figure 5.1.: Scale use for qualitative properties

The next step is to state all qualitative attributes which are regarded. In the table 5.2 all quantitative properties are listed and which scale is used to measure it.

<table>
<thead>
<tr>
<th>property</th>
<th>scale type</th>
</tr>
</thead>
<tbody>
<tr>
<td>available programming languages</td>
<td>nominal</td>
</tr>
<tr>
<td>approach to concurrency</td>
<td>nominal</td>
</tr>
<tr>
<td>usability</td>
<td>ordinal</td>
</tr>
<tr>
<td>scalability</td>
<td>ordinal</td>
</tr>
</tbody>
</table>

Figure 5.2.: Overview over estimated qualitative properties

5.3. Quantitative Properties

Most of the qualitative properties can be determined without implementing any use cases with the frameworks. Unlike, the quantitative properties need quite more effort to estimate them. In general, all quantitative properties measure some kind of efficiency or effectiveness of the framework relating to the implementation of a generic concurrency problem described in chapter 6. Although the observed characteristics can differ among the problems we define later, the table 5.3 briefly surveys possible quantitative properties and what should be cared about.

<table>
<thead>
<tr>
<th>property</th>
<th>requirements for comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>lines of code</td>
<td>same language</td>
</tr>
<tr>
<td>execution time</td>
<td>consistent influences</td>
</tr>
<tr>
<td>used memory</td>
<td>using same data structures</td>
</tr>
<tr>
<td>responsiveness</td>
<td>equally good implementations</td>
</tr>
<tr>
<td>throughput</td>
<td>equally good implementations</td>
</tr>
<tr>
<td>latency</td>
<td>determine mean</td>
</tr>
</tbody>
</table>

Figure 5.3.: Overview over estimated quantitative properties
6. Generic concurrency problems

The aim of this chapter is to specify different concurrency problems. Each one is implemented with the help of the frameworks to figure out which frameworks suit which problems. At this point, it is important to mention that there are no references dealing with the topic of generic concurrency problems. Therefore, we try to define generic concurrency problems by figuring out independent characteristics which describe each problem in an abstract way. Finally, we take specific problems which are implemented and try to classify them by their characteristics.

6.1. Characteristics of Generic Concurrency Problems

Before we can determine any characteristic of generic concurrency problems we need a model to describe it. In 6.1 you can see the model of a generic concurrency problem used in the following to describe its characteristics. The model describes generic concurrency problems very abstractly by splitting it up into three parts. First, there is the task itself which is independent from any implementation and describes what work has to be done. Second, in the real world we need an underlying execution environment where our problem is executed in. In this context, the term execution environment includes hardware as well as software. Furthermore, the hardware could be one computer, multi-cores, multi-processors or a distributed system. Third, to actually start the execution each problem needs input. Thus, the third part is called external handler and describes the external interface required for this task. Either there is a continuous exchange with externals, e.g. external handlers are clients of a web-server, or the system may listen to other external events such as sensor-events or the system is started explicitly for one call and stops again after replying. The optional output is represented by the dotted arrow. However, most generic concurrency problems have an output although one could imagine tasks where no output to externals is required. For instance, there could be a system which listens to external events and just logs them without any reply. Apart from that, one should notice that this model of generic concurrency problems targets an abstract description of the concurrency including the implementation of concurrency problems in practice. That is a difference to the models of concurrency theory. The models of concurrency theory, focus on describing concurrent system with different formalism but these models do not target to state anything about the characteristics of a given problem as a whole. In contrast to that, the generic concurrency model concentrates on classifying a concurrency problem by its complexity relating to various characteristics of practice use. From the perspective of an developer, you can only influence one part of a generic concurrency problem which is the execution environment. The concurrency programming models and software as well as the composition and architecture of the hardware can be influenced. The task itself and the required external interfaces are invariants.
With the model back in mind, now it is possible to determine independent characteristics of generic concurrency problems. An independent characteristic means the degree of one characteristic does not influence the degree of another characteristic. In the following, four characteristics of generic concurrency problems are described as well as possible metrics to measure them. In fact, they are not completely independent but their dependency does not limit their expressiveness to classify different concurrency problems.

**I/O intensity** describes the amount of I/O which has to be done on the execution environment to handle the problem. A possible metric for the I/O intensity could be:

\[
I/O \text{ intensity} = \frac{\text{time I/O is done}}{\text{total execution time}}
\]

This characteristic is influenced by task itself but also the underlying execution environment including the used programming models and architecture of the system. That is a problem, since it is difficult to determine, that before we have any concrete solutions. Nonetheless, we are able to determine the characteristic approximately because we know how real computers work and can determine how much I/O is needed at least. For instance, the problem requires that data is saved persistently or the problem is so big that you need multiple computers to handle it. Then you certainly know there is certainly a lot of I/O required for the network communication as well as the databases, even you cannot determine a precise value in advance.

**Computation intensity** describes how much has to be computed to solve the problem. More precisely, computing means executing machine instructions. Similar to measuring I/O intensity, it is reasonable to measure the ratio of the time used for computing:

\[
\text{Computing intensity} = \frac{\text{time for computing}}{\text{total execution time}}
\]
6.1. Characteristics of Generic Concurrency Problems

It was also considerable to measure the ratio of I/O intensity and Computation intensity, but two problems with an equal ratio could still have extremely different Computation and I/O intensity.

**Capability for parallelization** expresses the possibility to execute portions of the problem simultaneously. A problem with high capability is called embarrassingly parallel whereas the opposite is inherently sequential. This characteristic could be measured with $1 - B$ where $B$ is the fraction of code which is strictly sequential used in Amdahl’s law\(^1\) to determine the maximum improvement by executing parallel.

**Amount of State** measures how much state is necessary to be kept in the system to produce the expected results. One way to measure the amount of required state could be to measure the number of causal operations (read and writes) on mutable data:

$$\text{Amount of State} = \frac{\text{number of causal operations on mutable data}}{\text{total number of machine instructions}}$$

Illustrating these characteristics of a generic concurrency problem can be done easily with a radar chart. In figure 6.2 you can see an imaginary example of a generic concurrency problem with totally balanced characteristics. The generic concurrency problems are only required to determine if the frameworks are able to implement different kinds of concurrency problems properly. Thus, there is no benchmarking to figure out the accurate degree of each characteristic. Instead, the radar charts are based on theoretical considerations which describe rather the relation between different problems than accurate numbers. In all radar charts of this thesis the degree of each characteristic is between 0% and 100% but as you may have recognized the extreme values cannot occur because the characteristics are not totally independent. For instance, if there is only I/O to do, the processor must at least start the I/O work which has to be done even if the fraction is very small.

In order to compare concurrency problems on the basis of specific implementations in practice, it would be better to measure the absolute degree of each characteristic rather than their percentage degree. The reason for that is in practice you want to know if solving the same task with different execution environments results in different performance which is not reflected by the percentage degree. In the context of the thesis, we only vary the used software of the execution environment, more precisely the frameworks. Moreover, each framework influences the characteristics of I/O and computing intensity in some kind whereas the inherent characteristics of a task, namely, the required amount of state and the capability for parallelization, are independent from the execution environment. Although the exact degree of I/O and computing cannot be measured exactly beforehand because of its dependency on the used programming models, the overall amount of required I/O and computing can be approximately estimated. For instance, if you already know that you need a distributed system to implement the problem there is certainly a specific amount of I/O required for network communication. That means, the radar charts show the approximately theoretical estimation of the I/O and Computing which is required independently from the used concurrency framework. Precise classifications based on benchmarking would exceed the scope of this thesis.

\(^{1}\)http://en.wikipedia.org/wiki/Amdahl%27s_law
6.2. Specific Problems for Comparison

On the basis of the characteristics, we are able to define various specific concurrency problems and classify them by their degree of each characteristic. The aim of this section is to conceive specific use cases. By classifying each problem, it will be ensured they target different characteristics in order to represent bigger problems with similar traits.

6.2.1. Counting Server and Computation Intensive Server

For the first use case, we take two web server. The first is a counting server, a HTTP server, which counts the incoming requests and replies with the number of the corresponding request. Incoming requests cause concurrent modifications of the counter. Referring to its characteristics, it is quit simple to roughly classify the counting webserver.

In figure 6.3 one can see that the counting webserver has less computing intensity because each request does not initiate any intensive computations before the response is created. On the contrary, the I/O intensity is very high caused by the required network communication. The amount of state is low, since modifying one counter is actually not much but on the other hand the modification happens concurrently. Hence, additional mechanism to synchronize the access to the counter is necessary and increases the effort to care about the state. Finally, the potential for parallelization of the server is rather low, as the counter is a bottleneck. Indeed, this applies even more generally as the characteristics are not totally independent. That means, keeping track of state concurrently always limits the capability for parallelization in a certain degree.
6.2. Specific Problems for Comparison

Computing intensity

Capability for Parallelization

I/O Intensity

Amount of State

Figure 6.3.: Characteristics of the Counting Server

The second use case is a computation intensive server, again a HTTP server, but each request starts a costly computation. In particular, for each request PI is approximated with a certain precision and sent back to the client. For the computation the following infinite series is used:

\[ \frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} \] [15]

In terms of its characteristics (see figure 6.4), the computation intensive server can care less about state than the counting server, since each request can be handled independently. In consequence, the capability for parallelization is higher as well as the computation intensity. The required I/O by the server stays equally to the counting server because HTTP requests and responses do change only minimal. This kind of problem is able to scale out very well. The reason for that is, the problem has high potential for parallelization while there is no state which has to be cared about concurrently. Incoming requests can be dispatched on independent workers which do not need to interact among each other.

6.2.2. Transactions between Spaceships

The next use case is fictional and about a fleet of spaceships. Imagine, people can freely travel from one spaceship to another by beaming. However, each ship has an upper and lower bound of people it can carry. That implies, that there is some protecting mechanism to ensure, that the upper or lower bound is not exceeded when several people move simultaneously to the same spaceship and vice versa. This fictional use case, represents any other in which atomic transactions are needed to ascertain given limitations (e.g. a bank system). Under the term atomic
transaction, a change from one state to another is meant which either succeeds completely or not at all. In order to be able to classify the characteristics, it is determined that the implemented model of this use case will run locally on one machine. Moreover, the sequence of transaction is known before the execution to guarantee an equal burden for each implementation.

Relating to the characteristics (see figure 6.5), the problem has to track the number of people carried by each ship. Abstracting this more, the problem deals with having several counters which are modified concurrently and it is mandatory to keep each counter in a valid state. To achieve that, different techniques and models for concurrency can be employed but nevertheless the capability for parallelization of the problem is limited. The amount of state which is cared about is medium to high, because tracking a counter is only few work but caring about their concurrent modification needs more effort. Moreover, modifying multiple counters is not computing intensive and the I/O intensity will mostly depend whether it is executed locally or on a distributed system. The latter would cause a higher I/O intensity due to the network communication.

### 6.2.3. Frequency of Visits

For the third problem, we take one which offers a high potential for parallel execution or in other words, an embarrassingly parallel problem. A typical problem would be to determine the frequency of something in a set of data. In this case, we have a list of users and each user owns a sequence about the places he has visited so far. For example, visitedPlaces=\{a, b, c, d, a, e, f, b\ldots\}, so each place can occur several times. Now we want to know how often each place has been visited. This problem is embarrassingly parallel because you can split up the list of users and
count for each user separately. After processing the split work, the separate results must be merged again.

What are the characteristics of the problem? As illustrated in figure 6.6 the problem offers a high capability for parallelization. There is certainly a small to medium Amount of State required to store the results but since the data is not shared between parts of the system during processing the overall effort to keep about the consistency of the state is rather low. In terms of computing intensity, a considerable amount of work is required consisting of handling many long sequences. The I/O intensity is low because we extract the sequences only once and use the same sequence several times.

6.3. Summary

Based on the generic concurrency model we presented, it was possible to determine four main characteristics which can be applied to classify any concurrency problem. These are I/O intensity, computation intensity, amount of state and capability for parallelization. Although they are not completely independent they are suitable to roughly express the circumstances of a problem relating to concurrency. Furthermore, possibilities were given to measure each characteristic but there is still room to particularize the measuring more precisely. On top of that, we could determine three fictional problems which differentiate through their characteristics. The first, transaction between spaceships, has an high amount of state which is modified concurrently as its main characteristic. To simulate high I/O intensity combined with either concurrently accessed state or computing intensity, the counting server and computing server was chosen.
(we count them together as one problem since they are quite similar). Finally, frequency of visits targets to offer more potential for parallelization but also some computing intensity. In the next chapter we choose which frameworks are evaluated and employed to implement the problems we have defined.
7. Overview of Frameworks

After knowing which problems will be implemented, now we introduce several interesting concurrency frameworks for the JVM. These are JSCOOP, Guava, Disruptor, GPars, Vert.x and Akka. Furthermore, we reason the decision for each framework whether it is evaluated or not based on the requirements next.

7.1. Requirements of the Frameworks

In the scope of this thesis the concurrency frameworks which are evaluated have to meet two main requirements. First, they must offer an additional approach to concurrency compared to the basic concurrency support of Java and its native libraries. In particular, the frameworks must offer additional models or concepts. Because if they would not you could just use the native Java support. Second, the frameworks need to be flexible, so it is possible to implement various generic concurrency problems. Keeping the main requirements back in mind, we now review the frameworks which are discarded for the comparison. After that the evaluated frameworks are explained followed by the summarize of their qualitative properties.

7.2. JSCOOP

JSCOOP is the short term for Java Simple Concurrent Object Oriented Programming. In late 2008 the university of York released the public paper of the project. JSCOOP is a Java prototype implementation of the concurrency model Simple Concurrent Object Oriented Programming (SCOOP). SCOOP has its origin in the object oriented programming language Eiffel. The basic motivation of SCOOP is to simplify shared-state concurrency which is very error prone relating to the classic issues such as deadlocks and lock-starvation. Therefore, SCOOP introduces two special annotations. These are separate and await or for JSCOOP, @separate and @await. @separate indicates that a newly created object or an argument of a method is handled by a separate processor than the current one which created the object. In particular, in SCOOP a processor is an abstract object which is mapped to the actual execution threads of an virtual machine or OS. Additionally, @await is used to describe preconditions for method calls to express when a method actually can start allocating resources. For instance, there is a method dosomething(@separate Object a, @separate Object b...). JSCOOP will translate these annotation into pure Java classes realizing the specified behavior. If the method is called then, the marked resources a and b are ordered. An underlying scheduler cares about a fair distribution of the resources. As long as the method can not get the resources it will block. In Summary, JSCOOP is a an easy way to ease
the difficulty of programming shared-state concurrency. Nevertheless, JSCOOP is not regarded in the thesis. The main reason for that is that JSCOOP is only a prototype implementation. Another reason is that there is only few support for JSCOOP. Beyond the official paper of the project you cannot find much information.

7.3. Guava

When searching for concurrency on the JVM you will probably encounter on Guava. Initially, the library started as the Google Collection Library which was then replaced by Guava. Guava is one of core libraries of Google for Java-based projects. It is intended to collect tools which are needed in daily Java development. The support includes, collections, caching, string processing, I/O etc. but also concurrency. Guava extends the native libraries of Java for shared-state concurrency with some helpers. Examples for these helpers are atomic classes such as AtomicDouble or AtomicDoubleArray. Moreover, it extends the functionality of basic Java Future with classes such as ListenableFutures. On these you can register handlers which are invoked after completion of the Future. Other classes such as CycleDetectingLockFactory offer special locks applicable to detect deadlocks. This could be useful if you cannot relinquish locks but you also do not want to impose a total ordering of all locks preventing deadlocks. Under the surface, a directed graph notices when locks are demanded [4]. Nodes in this graph represent locks whereas edges represent an ordering of locks from the same thread. Whenever an edge is created the graph is checked for possible deadlocks indicated by a cycle in the graph. If a possible deadlock is detected, a configurable policy is invoked which decides how to handle the detection. However, Guava does not provide other concurrency models on top of shared state and threads. That is the reason why it is not evaluated.

7.4. Disruptor

In 2011, the Disruptor framework won the Duke’s Choice Award for Innovative Programming¹. This open source concurrency framework supports inter thread communication in Java. It is intended to provide the tools needed for building fast Java programs which work similar to an assembly line. In other words, the Disruptor is designed to implement problems which fit the producer consumer pattern. In this pattern, there are threads producing work and threads consuming work. Usually, the communication between producing and consuming threads is implemented by message-passing. The threads put work on a queue and consuming threads read from it and put the results on further queues. Similar to an assembly line, the problems, feasible through Disruptor, can be described by different stages. In each stage a thread handles a specific state of an event or object by changing or adding properties. The queues, connecting the threads, are implemented with a special data structure called circular buffer. This is a fixed-size buffer working like a closed ring. That means, each entry in the queue has a predecessor, so there is no end. Elements are put in the buffer with the principle first in first out. Because of its fixed size the circular buffer is usually implemented with an array. Hence, the buffer is fast, because basic read and write operations all need constant time. However, the main advantage is you can relinquish

¹https://blogs.oracle.com/java/entry/why_nominate_someone_for_a
several queues between different stages of processing. This is because, the circular buffer (called RingBuffer in Disrupter) stores a sequence number which references to the last event put on the buffer. All consuming threads, called EventProcessors, also store their own sequence number indicating which events have been processed so far. Therefore, they know what events still have to be processed. In consequence, they can share the events through the buffer and do not need to passed them around with additional queues. Between the different stages of EventProcessors SequenceBarriers ensure that events are read from later EventProcessors only when the events in the RingBuffer reached a certain progress. To avoid inconsistency when multiple threads read and modify events simultaneously, it is mandatory that the modification of an event does not overlap between different EventProcessors. Consequently, costly synchronization unnecessary which explains why the Disruptor can be very fast. Nevertheless, the use case of Disruptor is very specific and its limited flexibility is the reason why it is not regarded in this thesis.

### 7.5. GPars

GPars, standing for Groovy Parallel Systems is a collection of various concepts and models to ease concurrency on the JVM. In late 2008 Vaclav Pech started an open source project called GParallizer. It was intended supply Groovy user with the concepts of Fork/Join and ParallelArray known from Java. Groovy is an object oriented programming language for the JVM. In contrast to Java it is dynamic. That means, many actions which are usually done during compilation are postponed to execution.

After adding additional several other features to GPars such as actors or dataflow it was renamed to GPars and is now a standard package of the groovy language. GPars is implemented in pure Java with a Groovy-based domain specific language (DSL) on top it. A domain specific language is intentionally designed for solving certain tasks of the same type. In this case, it is describing concurrent workflow. Internally, the code written in the Groovy DSL is translated into Java code. As a consequence, GPars can be employed either from Groovy or you can employ the underlying Java directly. According to Pech\(^2\), ”GPars is a multi-paradigm framework, which offers several different abstractions and let’s you pick the one that works best for the problem at hand or combine several of these abstractions, where it makes sense.”

Now let us look what GPars offers towards concurrency. As mentioned before it offers parallel collections and fork/join for Groovy, so for Java users that is nothing new. Despite that, it offers several models for concurrent programming. These are Actors, Communicating Sequential Processing, dataflow. Though, it is noteworthy at this point that CSP is not implemented directly, instead it is explained how the dataflow tools could be used to model CSP. On concept level, GPars provides Agents and STM. Both concepts are designed to protect mutable state from inconsistency. Referring to the requirements 7.1 to evaluate a framework, GPars meets all of them as it offers several additional approaches to concurrency. This why GPars is evaluated.

\(^2\)From a personal conversation
7. Overview of Frameworks

7.6. Vert.x

Even younger than GPars is Vert.x which was started by Tim Fox in 2011 during his work at VMWare. In the meantime Vert.x was moved to the Eclipse Foundation. Three different reasons mainly motivated this project. The first one was "the growing interest in non-blocking, asynchronous approaches to applications [12]." Another was to offer a simpler API than other application frameworks. Lastly, the approach and success of node.js motivated to create something similar for the JVM. Node.js is a framework for building scalable network application in JavaScript. Based on this motivation, the project was developed and the result was "a polyglot, non-blocking, event-driven application platform that runs on the JVM [13]." As we can see, Vert.x is not intentionally planned to be a concurrency framework providing different concurrency concepts. Instead it offers a single approach to concurrency. In Vert.x, an event-driven architecture is employed similar to the reactor pattern, as described in 4.2. In short, Vert.x employs multiple reactors. Each reactor is single-threaded, but multiple reactors run concurrently. Moreover, they can communicate by exchanging events. As a consequence of the single-threaded execution, any concurrency issue is avoided in advance. Programmers do not need to deal directly with error prone concurrent programming anymore. Referring to distribution and scalability, Vert.x uses Hazelcast, a Java platform to architect clustering and highly scalable data distributions. Currently, you can use Vert.x with JavaScript, Ruby, Java, Groovy. The reason to include Vert.x in the comparison is, that single-threaded execution applied partially is an interesting approach. We want to find out how flexible or limited it is compared to other concurrency models.

7.7. Akka

Jonas Boner initially started the open source project Akka in 2009. Two years later, the company Typesafe was founded by the creators of Scala and Akka. Typesafe offers commercial support beyond the open source products such as training, consulting or additional development tools. Akka was strongly influenced by the Erlang programming language, a language designed for distribution, concurrency as well as fault tolerance [3]. Erlang offers very lightweight processes which do not share state. Instead they communicate by message passing. The main properties of Erlang are the same Akka intends to provide.

To implement them, Akka exploits the actor model (see 4.1.2). Actors imitate the notion of a lightweight process which communicate to others by message passing. To guarantee they are lightweight, each actor is not bound to a thread, instead actors share a thread pool. Furthermore, Akka offers a Software Transactional Memory as well as applying its concept of transaction to actors. Dataflow concurrency is also available but currently only for the Scala-API. In terms of safety, Akka relies on the "let it crash" model including supervision and monitoring. When a certain part of a system crashes, other parts of the system which observe the crashed part, are informed. Subsequently, the supervisors can decide how to react properly on exceptions. You can use Akka either with Java or Scala. The latter is a hybrid programming language which offers object oriented features as well as functional features for the JVM. All in all, Akka is regarded in the comparison. It suits the requirements to provide other approaches to concurrency than shared state and threads. Despite that, the motivation to facilitate scalability plus robustness sounds very promising.
7.8. Summary

Up to this point, we have learned several different concurrency frameworks for the JVM. Let us briefly summarize which one are evaluated. The first one is GPars. With the actor model and Dataflow it offers two main approaches for concurrency. Vert.x also participates in the comparison. Its event-driven approach with the single threaded execution are an interesting approach to concurrency as well as its design for scalability. Finally, Akka, focusing the actor model, is included. Its intention provide scalability and robustness encourage. In contrast to these three frameworks, JSCOOP, Guava and Disruptor will not be regarded. JSCOOP discards because its only a prototype implementation. Guava offers some useful helpers but no other approach to concurrency than pure Java. Lastly, Disruptor is limited to special use cases. It suits only when the workflow of the problem resembles an assembly line.
7. Overview of Frameworks
8. Qualitative Comparison

Evaluating the qualitative properties of the frameworks Vert.x, GPars and Akka which have been listed in table 5.2 is the aim of this chapter. Together with chapter 10 it contains the essential information of the whole comparison. First, properties measured on a nominal scale are elaborated, then properties where rating makes sense are discussed (see 5.2). Lastly, there is a short summary of the results.

8.1. Available Programming Languages

When choosing a framework, one might make his decision depending on the programming language he favors. In this section there will be no discussion if one programming language outperforms some other language or is shorter, easier to learn etc. This is because choosing a programming language is often only a matter of taste as long as the languages are equivalent in expressiveness and their performance does not differ dramatically.

Akka is available for the languages Java and Scala. Since Java should be known only Scala is introduced briefly. Scala is a object-functional programming language for the JVM. That means, Scala is mainly designed to be a functional language but it still offers techniques from object-oriented languages. Moreover, it is fully compatible with Java as you can call Java jars from Scala. In general Scala code is usually "reduced by a factor of two to three when compared to an equivalent Java application" [10]. Referring to Akka, the Scala API offers some additional support than the Java API, for instance dataflow concurrency is only available for Scale.

If you need many different languages in one project, Vert.x might meet that desire. It currently offers Java, Ruby, Python, Groovy, JavaScript and CoffeeScript. For all of these languages an API documentation exists except for CoffeeScript (as the time of writing). Additionally, support for Clojure is on the way. To offer Ruby, Vert.x uses JRuby which is an interpreter of Ruby for the JVM. JavaScript and CoffeeScript is available through Rhino which can compile these languages to Java bytecode or interpret them dynamically on the JVM. Python can be used through Jython which also compiles Python-code to Java Bytecode.

Just like Akka, GPars offers two different languages. It can be employed with the languages Java and Groovy which gave Groovy Parallel Systems its name. Groovy is a dynamic programming language which "attempts to combine the convenience of scripting with the functionality of Java [23]." Thus, the syntax of Groovy is a lot more compact than Java but its expressiveness is not limited as in other scripting languages. When using Java in GPars, the documentation recommends to use certain features directly from Java libraries such as Parallel Collections,
8. Qualitative Comparison

Fork/Join, Asynchronous functions (executor service). "The other parts of GPars can be used from Java just like from Groovy, although most will miss the Groovy DSL capabilities" [20].

8.2. Approaches to Concurrency

In this section, the different concepts of Vert.x, GPars and Akka for handling concurrency are explained and discussed.

8.2.1. Approach of Vert.x

Vert.x does not provide multiple models to handle concurrency. Instead it offers one approach. This is executing code only single-threaded on top of an event-driven architecture. In 4.2, we have already discussed the reactor pattern, one possibility to implement this architecture. Vert.x employs this pattern but it uses multiple reactors instead of one. They call this extension "multi-reactor-pattern" and one reactor is referred to as "event loop". The number of event loops, running on a thread pool, equals the number of available cores in the initial settings to exploit them all. In combination with an asynchronous programming model, this has advantages relating to performance we discuss now.

The main idea behind asynchronous programming is, synchronous APIs often lead threads to block, for instance when reading from a socket. In consequence, one would need many threads for each blocking operation, so that non-blocking threads are still available to execute other parts of the system. But handling many threads is costly due to overhead such as context switching. Thus, referring to the small number of event loops, Vert.x has "The Golden Rule - Don’t block the event loop!"[13]. Blocking one or even several event loops would cause the whole application to stop. Though, sometimes it is inevitable to do work which is blocking, for instance querying a database which does not support non-blocking calls. For that reason, Vert.x offers a special thread pool called "worker pool". This pool has usually more threads than there are event loops, thereby it shall be guaranteed that you do not run out of non-blocked threads during execution.

Both pools, the event loop pool and the worker share one eventbus. From that, they read and dispatch events to their clients and probably push new events on the bus. In Vert.x these clients are called verticles. One verticle is an instance of a class or script depending on the programming language used. Once an verticle is created, it is ultimately assigned to one event loop. From this moment on, the event loop is responsible to execute the logic defined by this verticle. This logic, consists mainly of handlers reacting on events. Generally, one event loop has to handle multiple verticles which do not need to be defined by the same class or script. Another type of verticle is called "worker verticle". In contrast to normal verticles, they are not assigned to a single thread. Instead every thread of the workerpool can execute it. The only constraint is, that one worker verticle is never executed simultaneously by more than one worker. Hence, it is still thread safe, because it is ensured that on thread sees the changes of a verticle made before by other threads1. Optionally, you can allow multiple threads to execute one verticle simultaneously

1 Answered in Google Group Vert.x by founder Tim Fox
but then you need to ensure thread safety on your own. All verticles together form a single verticle instance which is running on its own JVM.

Programming in Vert.x means implementing verticles. More precisely, most of the time you implement handlers and register them to the event bus. Handlers react to events, possible send new events or change the state of a verticle. Later, you can unregister the handler again. The address to register a handler is just a String, thus its free to the programmer to decide what formalism he uses to name events. Pushing events on the event bus can be done by calling send(message, <replyHandler>) or by publish(message, <replyHandler>) on the eventbus. When using send only the handler of one verticle will receive the event, whereas publish indicates that all handlers registered to the event will receive it. The optional <replyHandler> will receive the answer if a verticle replies to the receive of an event. At this point, it should be stressed, that events in Vert.x are not arbitrary Java objects. Instead they are restricted to simple types such as boolean, int, String and more complex ones such as JsonObjects, JsonArrays or Buffers. This is certainly a limitation when running locally because if you want to use normal objects for communication you need to transform them before pushing them on the event bus. On the receiver side, the object has to be recreated from the simple messages. An additional overhead for each message is the result of this trait. However, there is a reason for this since Vert.x is designed for developing distributed systems which imposes that messages anyhow have to be serialized before they are send over the wire. Further details about distribution and scalability are explained in 8.7.

8.2.2. Approaches of Gpars

In contrast to Vert.x, Gpars is a multiparadigm framework offering three different models to tackle concurrency. These are actors, Dataflow and CSP. Despite that, several other concepts are provided. We start looking at the models before we discuss what concepts are available.

Actors

The actor model of Gpars is feasible to implement a whole application as well as forming some subsystem of it. Especially, when asynchronous processing is desired, actors go well with that. In the following we discuss what Gpars actors support. What it misses is postponed to section 8.3 in which the actor implementations of Gpars and Akka are compared.

As we have already stated, the actor model of Gpars is implemented completely in Java with a Groovy DSL on top. When using it from Groovy, the DSL for actors is translated into equivalent Java code. Hence, using Gpars actors from Java or Groovy does not cause different performance. Referring to execution, all actors run on a thread pool. It is possible to group actors by assigning them to different thread pools. But one actor can always belong only to one thread pool. This feature might be useful if you want to customize your threads pools for different use cases. For instance you need a pool with many threads because its actors block frequently.

In general, you can choose between two types of actors. First, there are stateful actors represented by the class DefaultActor. This actor encapsulates an internal state which can be
8. Qualitative Comparison

accessed and modified while processing messages. The second type are the stateless actors represented by the classes DynamicDispatchActor, StaticDispatchActor or ReactiveActor. The DynamicDispatchActor dispatches incoming messages to its handler methods depending on the type of the received message. StaticDispatchActors are determined to receive only one type of message. Finally, the ReactiveActor processes a message with a function, determined at creation. After processing a message it replies with the result of the function applied.

Reading messages can be either done blocking with a special actor class blockingActor or non-blocking by using the other actors we mentioned before. A blockingActor is assigned to a single thread which is blocked every time its actor tries to read a message when none is available. In contrast, non-blocking actors release their thread each time when they read from their empty message queue. According to Pech, the stateless actors are faster than the others. Caring about state needs additional effort in form of continuations. In brief, continuation is the mechanism that the actors can be detached from a thread of a pool and invoked later by another thread. The state of an actor is preserved between detaching and continuing. Since stateless actors do not need continuations they are faster. Furthermore, a DynamicDispatchActor is slower than a StaticDispatchActor. The former has to determine the appropriate method to handle the type of message which costs additional time.

DataFlow

The second concurrency model GPars offers is dataflow. Its main advantages are, deadlocks are deterministic and it comes along with implicit synchronization. The implementation of GPars, however, is not pure dataflow concurrency. It does not relinquish basic concepts of Java such as exceptions or randomness. But the main requirement of real declarative concurrency like dataflow, is any non-determinism is not observable from the outside. Exceptions and random break this constraint. Nevertheless, these concepts are extremely useful. Exception handling is necessary to build robust applications. Implicit exception handling would be required to suit declarative concurrency. But the decision how to handle an exception can hardly be made implicit but sophisticated at the same time because of the variety of possible exceptions. Despite that, the lack of randomness limits the range of applications which can be expressed with dataflow. In particular, security stands in need of randomness to create keys for ciphering. Next, we have a close look on the programming with dataflow.

As described in 4.1.3, one way to realize dataflow can be implementing the producer consumer pattern together with dataflow variables and streams. These data structures are employed to supply implicit synchronization between threads. GPars follows this approach. Two main features are provided, data stores and tools to process the stores. The former consist of DataflowVariables, DataflowQueues and DataflowBroadcasts whereas the latter consists of tasks, operators and selectors. GPars DataflowVariables can be written once and read unlimited times. Yet, partial assignment to variables is not possible. When reading from a DataflowVariable, the reading is blocked until the variable has been set for the first time. DataflowQueues, also called streams, are used to implement the producer consumer pattern. They allow one to one or one to many communication between threads. If there is more than one consumer, they consume all different messages but one message is never processed by more than one consumer. Some other use case could be, to let each consumer process all messages from a queue. For this case, you can leverage the DataflowBroadcast.
To actually produce and consume work, you can use the features of processing data. First of all, there are tasks, small logical units, similar to a Callable or Runnable in Java. They run on the default thread pool or can be grouped in a new one. In order to coordinate, they read and set DataflowVariables and they are capable of returning promises (futures). Writing and reading from DataFlowQueues and DataflowBroadcasts can be done with Operators. Operators have multiple inputs and output queues. Every time, all input queues are hot (contain at least one unprocessed message) they consume one message from each queue, process the message and usually put new messages on the output queues. Other operators are, selectors, prioritySelector and splitter. A selector consumes a single message as soon as one of its input queues is hot. Similar to that, the prioritySelector consumes messages from prioritized input channels first if multiple are simultaneously available. Finally, the splitter copies a single input to all of its output channels.

Another feature of GPars dataflow is lazy evaluation. When creating a DataflowVariable its value is determined initially with a return value of a function. But this function will only execute and calculate the result when the variable is read for the first time. Lazy execution means it is executed when it is actually needed. That might be useful when you want to defer long running operations to prevent all of them be executed simultaneously.

Referring to error handling, exceptions, thrown while creating promises can be either handled directly or be propagated as the result of the promise. Operators can override their listener to handle exceptions or use the default one which prints the error message and terminates the operator. As mentioned before, GPars relaxes the limitations of pure dataflow concurrency at this point. That was a rough overview about the basic functions of GPars dataflow. In section 8.4, you can find the comparison between GPars dataflow and Akka dataflow.

**Communicating Sequential Processes**

Communication Sequential Processes is intended to offer a formalism to describe and implement concurrent applications. Its main advantage is, the correctness of a CSP application can be proofed. The documentation of GPars describes two possibilities to leverage CSP, either by using data flow or by the external library JCSP.

First, we discuss how dataflow can be applied for that. To represent the execution logic of CSP processes, you can use Closures, Runnables or Callables. For modeling synchronous communication through CSP Channels, special SyncDataflowQueues and SyncDataflowBroadcasts are available. Unlike asynchronous Queues and Broadcasts, their synchronous versions blocks writing to the channel as long as there is no reader ready to process the message. Finally, alternatives, non-deterministic choices between incoming events, could be modeled through a selector. As a reminder, a selector is a special operator which starts processing incoming messages as soon as at least one input channel contains work. In conclusion, it is possible to use GPars Dataflow to implement CSP but it misses the original idea to provide automatic checks for correctness. It is still to be shown that this is indeed possible with GPars Dataflow.

The second approach to CSP, recommended by GPars, is to use the external library JCSP. It was developed at the University of Kent at Canterbury and released in early November 1999. The framework is purely Java and provides all primitives required for CSP. According to "Formal
Analysis of Concurrent Java Systems, the implementation of JSCP is correct and generally each JCSP program can be proved formally of its correctness [40]. In the scope of this thesis, we will not look how JCSP works. This is because, there is already a comparison [34] including JCSP with CTJ, another CSP implementation for Java.

Data Parallelism

After we discussed the three concurrency models, we look what additional concepts GPars offers. Processing data in parallel is basically no concept for concurrency (see 2.1). However, what GPars calls Data Parallelism is no framework for parallel programming. Even instead, when you want to use Java then GPars does not supply you with any additional functions for parallelism. For Java users, GPars recommends to use the basic Java libraries jsr-166 including, parallel arrays, fork/join etc. to exploit parallelism. For groovy, it adds features to process collections concurrently which is build on the java Fork/Join framework under the surface [20]. In addition, you can also execute functions asynchronously on a thread pool (like Java executorservice) or perform just a fork/join algorithm. In order to process collections parallel, GPars has a special DSL for map/reduce. Usually, before processing a parallel function on a collection, a parallelArray is constructed. This array is capable to be processed from multiple threads while ensuring consistency. When a function is processed on its elements the result collection is returned. Afterwards the parallelArray is destroyed again. In the map-reduce environment (started by calling collection.parallel) the parallelArray is created only once and the programmer itself can decide when to retrieve the result collection. This increases the speed, when chaining several functions because the parallelArray is constructed only once. To conclude, the mentioned features are useful for Groovy users but already exist for Java. As concepts to protect mutable state GPars offers Agents and Software Transactional Memory.

Agents

Agents can be employed to protect mutable state which is shared between multiple entities of an application. They are mainly inspired by agents from Clojure (see 4.2). GPars agents own a state object which is changed by passing a closure to the agent. The closure acts on the state object and mutates it. If you pass another object than closures to the agent the state object will be exchanged with the passed object. Calls to extract an agents value, can either be performed synchronously or asynchronously. Under the surface, these calls are deferred until all preceding change messages are processed.

STM

Another approach to protect shared mutable state is using Software Transactional Memory (see 4.2). The STM, "currently" available in GPars, is called Multiverse which is also an open source project. "Currently" because in a recent discussion the responsible of GPars decided to look for other STM they could include. The reasons were, that the last update of Multiverse is

2This was discussed in the GPars mailing list
8.2. Approaches to Concurrency

several months ago and with the current version 0.7 some use cases could not be implemented. For instance, handling sets and lists is not supported. Nonetheless, we briefly summarize how Multiverse works and is used. In contrast to other STM implementations, Multiverse relinquishes an optimistic non-blocking approach and employs always locks for writing instead. With the use of a global version clock each transaction is numbered. Each Data field of the STM stores the number of the transaction which caused the last change to the field. Reads need to compare if their transaction number is higher than the one of the fields they got as a result to know if their read was successful. When data fields of the STM are written, the transaction tries to lock them, change the fields and update the version number of the field. More details about this STM implementation can be found in [18]. Programming with Multiverse is straightforward. Refs are pointers to the data fields of the STM. They can be changed with the function `atomic` which represents a single transaction. This function is called with a `Runnable` in which Refs are read and written. Either all writes inside an `atomic` commit or none.

Active Objects

Active objects are suitable when you want to exploit the advantages of asynchronous programming combined with object orientation (see 4.2). In GPars active objects can be utilized by using annotations. You need to mark a class with `@ActiveObject` and methods with `@ActiveMethod`. If an instance is created of this class, additionally a hidden actor is created responsible to execute the marked methods. Calls to methods of the instance are asynchronous returning immediately a `DataflowVariable`. Under the cover, these calls are transformed into messages sent to the internal actor of the object. For each received message the actor invokes the demanded methods and binds the `DataflowVariable` to its result.

8.2.3. Approach of Akka

To encounter concurrency, Akka mainly relies on the actor model. Additionally to actors or also in combination with them, you can use a STM, Agents, and some further support to dataflow but only for scala.

Actors

The actor system of Akka tries to achieve three main properties inspired by Erlang. These are distribution, concurrency and fault tolerance. To accomplish all of them, Akka actors offers much more features compared to GPars. A direct comparison between both actor implementations is available in 8.2.2.

Now we discuss the actual implementation. `ActorSystem` provides the basis for a whole actor application. New actors are created either from inside another actor or directly as a child of the `ActorSystem`. Thereby an actor hierarchy is constructed. That means, each actor has exactly one parent actor and any number of children actors. Furthermore, an actor has a reference to the root `ActorSystem` object. This stores the references of all actors created. Actors can ask it
to search for a reference by passing an path. The path determines where the searched actor is
placed in the hierarchy. Especially, in terms of fault-tolerance the hierarchy is important. Every
actor is a supervisor for its children. More precisely, you can use the basic or set a custom
supervisor strategy for each actor. When an exception is thrown within an body of an actor
the exception is handled based on the given supervisor strategy. For instance, the exception is
forwarded upwards the hierarchy, handled directly or sent somewhere else when other branches
of actors need to be informed. Handling an exception could mean, resume, stop, restart actors or
do something else. Akka comes along with two strategies, the One-For-One Strategy and the
All-For-One Strategy. The former applies it rules only on the failing child actor, whereas the
latter applies it on each child, although only one child failed.

After describing the basis of Akka actors we deal with the actors itself and later with further
features. Two different types of actors exist, untypedActors and typedActors. UntypedActors
process messages (just any Object) from their mailbox. They react on messages by sending new
messages or creating actors while keeping track of an internal state. In contrast, typedActors, are
an implementation to provide active objects (8.2.2). The behaviour of an actor can be changed
during execution, also called hotswap. By Calling the method become, passing the new behavior
as a parameter, the actor will change its behavior to the new one. Changes to behavior can
also be rolled back by calling unbecome. Some other very useful features are routers, configue
dispatching, remote actors and transfer actors. First, routers are special actors which route
incoming messages to their routees (just default untypedactors ) depending on a given strategy.
For example, strategies could be roundRobin, broadcast or loadbalancing. Special about routers
is they are not harmful to scalability because incoming messages bypass the routers mailbox and
are concurrently dispatched to the routees. However, for a programmer a router is just a common
actor with an address you can send messages to and get replies from. Moreover, Akka enables to
choose between the dispatching strategy of messages. In the default dispatch behavior each actor
has its own mail box, and all actors share one underlying thread pool. As for instance, other
configurations might be, actors of the same type share a mail box, each actor has its own thread
or unconsumed work is redistributed from busy to idle. The feature of remote actors is explained
in 8.7 whereas transfer actors in 8.2.3. Although we discussed not all features of Akka actors, the
most important ones were stated. In conclusion, Akka offers an elaborated implementation of
the actor model with many useful features.

Active Objects

The concepts of active objects is supported by so called TypedActors in Akka. It is recommended
to use them "exactly at the touching point where actor systems meet non-actor code"[14]. These
are normal actors as they can send and receive messages. Moreover, they can be supervised by
other actors or be a supervisor. In order to create a TypedActors you need to write an interface,
an implementation of it and pass both to the actor at creation. An TypedActor is then used like a
normal object on which you can call the methods defined in the interface. Under the cover, these
calls are sent as messages to the actor. You should care that methods return either nothing or
Futures. If not, calling a method of a TypedActor would block the thread which made the call.
8.2. Approaches to Concurrency

**STM**

Akka employs the ScalaSTM which is mainly inspired by the STM of Clojure. Like other STMs it intends to protect shared mutable state but Akka also applies it for transactions across several actors. The main idea behind the inspiring STM of Clojure is, a variable has an identity and corresponding values which are time relative. That means, updating a variable adds the latest value to the history of the variable but not override the old ones. This notion derives from pure functional data structures. The ScalaSTM does only retain one old version, not more. For a programmer, the identity is a represented by a Ref whose values can be changed in transactions. Each transaction is started with an optimistic approach, then falls back to pessimistic conflict detection when it could not succeed.

Referring to actors, Akka supports to send coordinated messages to actors. When receiving coordinated messages, each actor can decide if he participates the transaction and dispatches the coordination message to additional actors. Then each actor tries to execute his portion of the transaction which is usually updating a STM Ref. The whole transaction will succeed completely or all changes are rolled backed after a variable duration of time. To facilitate this procedure, Akka offers transactors which have a function for handling normal messages and coordination messages. The process of joining and dispatching coordinated messages is simplified with transactors.

**Agents**

The agents provided by Akka are very similar to the one offered by GPars (see 8.2.2). Each Akka agent protects a storage (plane object). To change the state of the storage you have to pass a function to the Agent which is performed then. Processing different functions is done sequentially and asynchronously. Requests to retrieve the state of an agents are done synchronously. Furthermore, you can include the agent in a STM transaction. When an agent participates in a transaction any messages sent to the agent are deferred until the transaction succeeds. If the transaction fails, these messages are discarded. In contrast to the agent of GPars, it is not possible to change the state object, once the agent has been created. If that is a drawback or not is a matter of taste.

**Dataflow**

The support of Akka to Dataflow is very small compared to GPars. For a summarized comparison between both refer to 8.4. The first limitation is, that dataflow support only exists for Scala. For its use, you need to integrate the Delimited Continuation plugin from Scala which provides continuations required to create asynchronous flow methods. The flow method is a construct to represent a dataflow expression. Inside a flow expression, promises can be used just as usual dataflow variables which means they can be written only once but read multiple times. With the continuation plugin each step of an flow expression, e.g. write to several dataflow variables looks like it is executed synchronously but indeed it is asynchronously and non blocking in the background. The result of each call to flow is a Future containing the results of its execution. That is all, there is no further dataflow specific support.
8. Qualitative Comparison

8.3. Akka actors vs GPars actors

This section is an overall summary of the differences between both actor systems provided by Akka and Gpars. The similarities and differences discussed are summarized in table 8.1.

Let us start with the similarities. Both frameworks offer active objects backed by actors. It is a bit less effort to use active objects from Gpars, since you only need to use annotations, compared to Akka where you have to create and interface plus an implementation suiting the interface. In both actor systems, messages sent to actors can be any object you want. When processing these objects without the need of state inside the actor, Gpars offers special stateless actors to dispatch the messages to the appropriate handling. However, in this thesis it was not regarded whether stateless actors of Gpars are faster than Akka actors. Despite that Akka offers several additional features which are not covered by Gpars. First, Akka offers routers, which can be very useful relating to scalability. For instance you just need to increase the number of routees for each router if your system has untapped power left. Other important features are supervision and monitoring of actors which allows to build a fault-tolerance system in which parts of the system can recover from errors without stopping the rest of the system. In order to update multiple actors in one transaction, Akka offers transactors or coordinated messages. Nevertheless, it is questionable if transactions across several actors suit the notion of the actor model at all. Remote messaging between actors is only in Akka possible which is particularly discussed in 8.7. Also configuring the dispatch strategies of messages is not supported by Gpars. Akka is more flexible in that point. Finally, changing an actors behavior during runtime (hotswap) is directly supported by Akka whereas implementing this behavior in Gpars has to be done by yourself.

In conclusion, the actor system provided by Akka offers a lot more features. Thus, from qualitative perspective it is recommend to prefer the actor system of Akka to the one of Gpars.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Akka</th>
<th>G Pars</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages can be any Object</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Active Objects</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>stateless actors</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>routers</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>transactors</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>remote actors</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>configurable dispatching</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>hotswap</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.1.: Similarities and Differences between Actor Systems

3 reply in Gpars mailing list to the question if transactions are possible across Gpars actors
After we discussed the differences between the actor models, we compare the support for dataflow.

In general, both Akka and GPars implement a pragmatic version of dataflow. As mentioned before, a property of dataflow is its non-observable non-determinism. As an observer you cannot see that the execution is non-deterministic under the cover. This property prohibits the use of constructs like exceptions or randomness because their occurrence introduce observable non-determinism. But in both frameworks, it is allowed to use exceptions as well as random. Hence, their implementations build only on basic ideas of dataflow and none provides a pure dataflow approach. All difference between their support is summarized in table 8.2.

Using dataflow in Akka is only possible from Scala whereas Groovy allows to employ it from Java and Groovy. But as we have learned before, using it from Java requires more effort because the documentation focuses on Groovy and you need to struggle through the API.

The basic approach in Akka is the flow method which represents a dataflow expression. After processing asynchronously multiple commands of an expression, it returns a Future containing the composed results. In GPars, one could model a flow expression with multiple tasks which are wrapped in a pool and their results are composed after processing. Beyond this, Akka does not offer anything more. To be more precise, Akka does not offer dataflow queues nor broadcasts. In contrast to GPars, Akka lacks of operators such as selectors or splitters to process the streams. The concept of lazy evaluation of dataflow variables is only in GPars available. In brief, Akka has only a minimal support to dataflow whereas GPars offers constructs for building channel processing similar to an assembly line. Thus, you should favor GPars to Akka relating to dataflow.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Akka</th>
<th>GPars</th>
</tr>
</thead>
<tbody>
<tr>
<td>strict declarative concurrency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dataflow variables</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>group expressions</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>queues, broadcasts</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>operators, selectors, splitter</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>lazy evaluation</td>
<td></td>
<td>+</td>
</tr>
</tbody>
</table>

*Figure 8.2.:* Similarities and Differences between Dataflow
8. Qualitative Comparison

8.5. Vert.x Multireactor Pattern vs Actor Model

On the first look, Vert.x Multireactor Pattern and the Actor Model seem to be very similar. Vert.x itself says: "As such the Vert.x concurrency model resembles the Actor Model where verticle instances correspond to actors. There are however differences, for example, verticles tend to be more coarse grained than actors [13]." Unfortunately, this is the only difference mentioned in the main manual. Thus, this section aims to figure out more similarities and differences between both. Therefore, we compare the actual implementation of the Vert.x Multireactor Pattern against Akka actors and GPars actors. First of all, let us explain what "verticles tend to be more coarse grained than actors" means. In Akka and GPars all actors run on thread pools. Each thread of a pool is able to run each actor, there is no fixed assignment. In contrast, verticles are executed always by the same event loop (except they are worker verticles). If two or more verticles are assigned to the same event loop, it can never happen that more than one is executed simultaneously.

Thus, the possible concurrency is more coarse grained. Another difference is, actors are addressed directly. Actors communicate among each other by sending messages to these addresses. On the contrary, one verticle can listen to several addresses. For that reason, it registers handlers which are invoked when events (messages) are sent to the addresses it was registered. Multiple verticles can register handlers at the same address. Some other difference is, to create a dialog between two verticles you need to nest multiple response handler using reply to answer. reply is the only way to explicitly reply to the verticle which initially sent the event. With actors you can use a combination of current state, and addresses of actors to determine in which state a conversation between two actors is. All messages in Vert.x can only consist of simple types whereas Akka and GPars allow to send any object as a message but this depends again strongly on the actual implementations. The concept of routers from Akka, implicitly exists in Vert.x. Whenever you use send(address, event) the event is only forwarded to a single handler which is equivalent to the function of a router. The broadcast messages you can sent in Akka, are similar to publish an event in Vert.x. But only all routees of s router will receive the message, not all actors. In Vert.x theoretically every verticle can receive an published event if it has registered a handler for it. Referring to distribution, both models are suitable to incorporate in a network but concrete advantages or disadvantages are highly dependent on the actual implementation (see 8.7).

8.6. Usability

One of the most important properties when making a decision between several frameworks is their usability. Thus, this section tries to answer how difficult it is to get started with each framework.

Installation

Before any line of code is written, each framework need to be set up. A difficult installation might annoy user and let them choose another framework which offers an easier installation. Since the set up of a framework is rather boring, this passage is kept short.
8.6. Usability

To install Akka, you can download the required jars directly\(^4\), use the Typesafe Platform which offers additional support for developing or employ builders to include Akka in your project. Using Akka requires at least Java 1.6. Vert.x requires Java 1.7 and can be either be downloaded directly\(^5\) or cloned from its github repository. Finally, installing Gpars can be done implicitly by installing the programming language Groovy which includes Gpars\(^6\) or using build tools\(^7\).

To sum it up, the installation of all three frameworks is no barrier. It is equally simple and therefore all frameworks get same rating of 5 points on the ordinal scale.

Support

Making a decision between frameworks is often based on how much support there is to learn it as well as help when problems occur. We split it up into support for free through the internet and additional support such as books or training.

During writing this theses, I had contact with each community and asked several questions. If you have any questions on Akka or Vert.x you should contact their google groups of them which are very active which means you mostly get answers to your questions within one day. The same applies to Gpars, but instead of the google group you should subscribe to the community mailing-list. Although this chapter deals with a qualitative comparison we include some figures to illustrate the support. For this reason we look at the number of questions asked on stackoverflow\(^8\). Stackoverflow is a very popular site to question and answer questions on programming. Its rating system of answers ensures that most of the replies have as high quality. The figures of the google groups are also provided. Delivering numbers for the mailing list of Gpars is not done because of its difficulty. You would need to be a subscriber from beginning then or get the data from someone else.

In table 8.3 you can see the frequency of questions asked on stackoverflow marked with the tags of the frameworks. In table 8.4 figures of corresponding google groups are shown. Akka has definitely the highest activity in both cases. Gpars shows lowest both times but to clarify Gpars has no google group. However, from my personal experience the mailing list is much less active than the google groups of the two others\(^9\). Furthermore, searching for problems which have been already answered is not possible with the mailing list. To conclude, each of the frameworks has active communities which should be the first contact when you have some difficulties. Akka has the most active community, followed by Vert.x whereas Gpars shows only small activity.

Another important property is if there is good literature and documentation for the frameworks. We begin with Gpars. Its documentation contains mainly Groovy Code. There is a short section about Java, but if you want to use Gpars from Java you need to bring in a lot effort to go through the API and figure out what you need. Beyond the documentation, there are currently no books available which could help to start with Gpars. Vert.x, however, offers for each language a separate documentation to get a quick start. Beyond the documentation there are also no books which could help to learn about general patterns relating to Vert.x, how to use Vert.x or

\(^4\)\url{http://akka.io/downloads/}
\(^5\)\url{http://vertx.io/downloads.html}
\(^6\)\url{http://groovy.codehaus.org/Installing+Groovy}
\(^7\)\url{http://gpars.codehaus.org/Integration}
\(^8\)\url{http://stackoverflow.com/}
\(^9\)I got about 20 mails on 4 topics in 5 months
8. Qualitative Comparison

<table>
<thead>
<tr>
<th>Tag</th>
<th>Number of Followers</th>
<th>Number of Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpars</td>
<td>11</td>
<td>45</td>
</tr>
<tr>
<td>akka</td>
<td>1117</td>
<td>458</td>
</tr>
<tr>
<td>vert.x</td>
<td>44</td>
<td>32</td>
</tr>
</tbody>
</table>

Figure 8.3.: Shows the figures of stackoverflow relating to the frameworks (as on 1 October 2013)

<table>
<thead>
<tr>
<th>Group</th>
<th>Number of Members</th>
<th>Number of Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Akka User List</td>
<td>2723</td>
<td>4470</td>
</tr>
<tr>
<td>vert.x</td>
<td>1418</td>
<td>1900</td>
</tr>
<tr>
<td>GPars</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 8.4.: Shows the figures of the google groups relating to the frameworks (as on 1 October 2013)

solutions to frequently problems. With the Typesafe Platform behind it, Akka provides training to learn Scala, Akka and also additional tools such as spray which is a webframework build on top of Akka. Like Vert.x, the documentation of Akka is equally detailed for Java and Scala. Additionally, books such as "Composable futures with Akka 2.0"[37], "Akka essentials" [21] or "Akka in Action"[31] (currently early access edition only) could help to get better understanding. In summary, using GPars from Java is not documented well but as its actual intention is to be a Groovy extension to handle concurrency, this fact might never change. Vert.x provides for each language equally documentations but all in all Akka offers the most professional support which is probably caused by its commercial background. Expressing this on my rating between one to five would be GPars gets two Vert.x three and Akka four points. This shall express that GPars offers the fewest support. A bit more support is given by Vert.x, especially its google group is more useful than a mailing list. Akka delivers most with its activity on stackoverflow as well as its google group. Besides that it offers literature to get an overall overview.

8.7. Scalability/Distribution

Whenever you develop an application, scalability is a property which should be reasoned about prematurely. Otherwise, it could cost a lot effort to adjust an application to improve scalability afterwards. This section deals with how each framework is suitable to create scalable applications. Therefore, we distinguish between Scale-up and Scale-out (see 2.1). More precisely, we start discussing the possibilities to create applications which are adaptable to exploit the available resources of a single powerful machine (Scale-out). Then we look on the capability of the frameworks to build applications which can scale out by using distribution.
In Akka, an application is mainly based on the actor model. In the basic settings of Akka, all actors share one thread pool to utilize all cores of a machine. To build a scalable (up) application with actors, you should frequently use routers if possible. The amount of actors (or routees) which belong to one router are determined either in the code or by configuration. It is recommended to do it by configuration because this enables to tune the performance more flexible. For instance, on a machine with four cores, having only two routees is a possible waste of untapped power. Subsequently, you could increase the number of routees in the configuration up to four. However, this fine tuning should only be done if it is really necessary. Back to our example, it could also be that two routees are enough because there is never more work to utilize more than two. On the whole, Akka actors suits well to develop an applications which exploits the power of a single machine. Likewise Akka, it is possible in GPars either with dataflow or actors to scale up but it requires a lot more effort to make it adaptable than in Akka. GPars actors can be grouped in different thread pools to apply fine tuning. However, they lack routers which are required to provide simple changing of the amount of actors used. In general, configuration must be implemented by yourself. Referring to dataflow, GPars has the same problem. If you realize, you could use additional operators at a certain point to increase performance you need to change the code directly. Alternatively, you could implement something to enable configuration for dataflow by yourself. Scaling out with Vert.x is similar to Akka more flexible. Initially, configuration is simply available by passing external configuration files. You can define the number of verticles there, inside an application read the values and deploy the appropriate number of verticles. In contrast to the actors of Akka and GPars, one verticle is only executed by the same thread. This limits the utilization of a whole machine. Whenever two verticles are assigned to the same event loop they cannot run simultaneously. Hence, they would not make use of additional idle processors. In the basic configuration, the number of event loops equals the number of available cores to exploit all cores of a machine. In conclusion, all three frameworks enable to scale up an application. However, in Akka and Vert.x it requires less effort than in GPars to build adaptable configurations.

In the same way, now we discuss how each frameworks supports Scale-out. The current state of GPars does not provide any support for distributed computing. There are plans to implement features for that, but at the moment nothing concrete is in the works. On the contrary, Akka as well as Vert.x are designed to create distributed applications. Therefore, Akka offers remote actors. These can be used like normal actors. That means, they are UntypedActors which run on an ActorSystem. To retrieve the ActorRef of an remote actor, you can use the path which determines its location including the address of the remote system. For example this code snippet would deliver the ActorRef of an remote actor.

```scala
ActorRef remoteActor = getContext().actorFor("akka://RemoteNodeApp@10.102.141.14:2552/user/remoteActor");
```

**Code 8.1:** Shows how to retrieve the address of a remote actor

RemoteNodeApp is the name of the actor system with the given ip and port number. /user/remoteActor is the path where the actor is located (user is always the top in each actor system). Once the reference is accessible, you can send messages to the actor like to any local actor. Additionally, features such as routers can also be employed to distribute the work on several nodes. In the configuration you can determine where the routees of an router are located in the network. Under the surface, messaging between remote actors is realized with Netty on top...
of TCP [21]. But the employed transport mechanism is adaptable through configuration if you want to use any other. With remote actors, you could implement patterns to distribute the work of your application. Besides that, each actor can subscribe to listen to events from remote actors. This makes it possible to react directly on events, for instance errors or forward them in the form of throwing exceptions to supervisors. In the current version of akka 2.2.1 Clusters were introduced to simplify the distribution of an application while improving fault-tolerance. Nodes can join and leave the cluster which is noticed by other member nodes of the network. However, most of the planned features have not been implemented yet. Thus, we do not discuss it more at this point. All in all, Akka tries to provide an easy approach to scale out. In the near future additional support for clustering will be added.

Vert.x provides a more simple approach to distribute an application than Akka. You just need to run your application passing the option -cluster on each node of the network. For instance, vertx run handler.js -cluster starts an handler verticle on one node and vertx run sender.js -cluster starts verticle which could push events on the event bus. -cluster causes the deployed Vert.x instance to from a cluster with other instances on the network. Furthermore, an distributed event bus instead of a local one is used then. Under the covers, Vert.x employs Hazelcast to share the information between the nodes required for clustering. This information consists of the clustered servers, handlers, their network addresses as well as the events each node is subscribed to. The distributed event bus itself is based on TCP and does not use Hazelcast. Inside of a verticle, there is no difference when running on a cluster. The API to register handlers at the event bus as well as sending events on the bus stays the same. Generally, this is more easy to use than remote actors in Akka. This is because, you do not need explicitly create the references to the actors you want to communicate with. Instead, in Vert.x you just send the event to an address on the event bus. Listener to this address can retrieve and answer to the message but it is not required to know where the handlers are located in the network. One limitation in Vert.x is, that shared immutable data is currently only available when running locally, not in cluster mode. In terms of error handling, you should stick to the following rules: “All messages in the event bus are transient, and in case of failure of all or parts of the event bus, there is a possibility messages will be lost. If your application cares about lost messages, you should code your handlers to be idempotent, and your senders to retry after recovery [13].”

In summary, GPars, Akka and Vert.x can be employed to scale-up an application. But in Akka and Vert.x it is more comfortable to create an application which is adaptable to tune through configuration. In terms of Scale-out, GPars has currently no support at all. Akka offers remote actors and intends to improve their support for clustering. Finally, the approach of Vert.x is an distributed event bus which is easy to use but it imposes the programmer to care more about error handling than in Akka. Expressing this on the ordinal scale GPars gets only one point because it does not allow to scale out. Vert.x gets 4 points since it is more simple to scale out but since Akka allows better fault-tolerance it also gains 4 points.

---

11example of a pattern http://www.akkaessentials.in/2012/03/implementing-master-slave-grid.html
12stated by Tim Fox:https://groups.google.com/forum/#!searchin/vertx/hazelcast/vertx/E2q9Auq7gfI/QbZ08MAT8d4gJ
8.8. Summary

Let us have a short review of the most important facts we figured out. The two models to handle concurrency of GPars are dataflow and actors. The latter is also what Akka focuses. Beside that it has support to dataflow but this does not suffice to develop an entire application with it in contrast to the one of GPars. On the other hand Akka actors provide a lot more support than GPars actors. The approach of Vert.x employs the reactor pattern which resembles the actor model. However, this pattern allows only more coarse grained concurrency than the actor implementations. My personal ratings for both ordinal properties, usability and scalability/distribution is summarized in table 8.5. In terms of usability and support, Akka clearly outweighs the others. GPars offers the smallest support since the community of Vert.x is still better. In order to scale up an application, all three frameworks are appropriate. In contrast to that, scaling-out is currently not supported by GPars. The distributed event bus of Vert.x requires less effort than remote actors to distribute an application. However, in terms of fault tolerance Akka beats Vert.x since actors can observe the state of remote actors and react on failures properly. Vert.x does not involve a similar mechanism for verticles.

<table>
<thead>
<tr>
<th>property</th>
<th>Akka</th>
<th>GPars</th>
<th>Vert.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>usability (installation)</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>usability (support)</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>scalability/distribution</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

**Figure 8.5.:** Summary of both properties estimated on ordinal scale
8. Qualitative Comparison
9. Implementation

In this chapter, we will discuss all implementations of the problems specified in 6.2 for each framework. More precisely, the implemented properties of each problem are determined closely. After describing the general requirements of a problem, we discuss the implementations as well as the experiences made during this time. Moreover, it should be mentioned, that every problem is also implemented with pure Java. Within each framework, Java is used as far as possible.

9.1. Requirements and Specifications for Transactions between Spaceships

The precise requirements of the problem "Transactions between Spaceships" are only few. First, each implementation has to do the same amount of work, so that later benchmarks make sense at all. By beaming just one person from each ship to all other ships it is guaranteed that each implementation will have the same amount of transactions which is \( \text{numberOfTransactions} = \text{numberOfShips}^2 \). The initial settings of each ship are determined so that the maximum and minimum capacity will not be exceeded with these transactions, however every case is still implemented so that transactions will fail if they cross the limitations. Without this restriction of capacity, a ship could immediately modify its number of people and send the corresponding change to another ship. Ships would process incoming change events and update the information but there would be no need for any synchronization in which is checked if none of the capacities is exceeded. A single transaction is represented by a Java Object, because in message passing concurrency that is needed anyway. Also with other approaches such as locking, we wrap each transaction as a good manner in terms of object-oriented programming. Finally, it should be possible to measure the execution time of a run, which is not trivial. It would cost a lot of time to include this afterwards. Lastly, all implementations are executed locally. In summarize, there are five requirements:

- determining the transaction in advance (each ships sends one person to any other ship)
- transactions are represented by an object
- transactions regard capacity limitations
- including mechanisms required for benchmarking (execution time, throughput and memory)
- the problem is executed locally on a single machine
This problem was the only which was implemented with pure Java, Vert.x, Akka and GPars. Furthermore, there are several approaches for each possibility, except Vert.x.

### 9.1.1. Implementation with Java

In Java, the problem was solved in two different ways. The first approach was to employ just simple single-threaded Java. There is only one thread and all ships are stored in a list. When starting execution, the thread starts to create all transaction messages and after each creation, it passes the object to the method of the ship to process the transaction. This is absolutely thread safe because there is only one thread. Thus, it is not required to involve synchronization. Determining the end of execution is implicitly. When all transactions are done, the main thread has no more work to do and stops.

The second approach targeted to exploit the potential of parallel execution, preserving the approach of shared state concurrency. For that reason, the ExecutorService of Java was employed. This time, each ship implemented the Callable interface to be able to run on the executor. In addition, every ship stores all references to the other ships. When the call method is invoked by the executor, the corresponding ship starts sending one person to each of the other ships from its references. The process of sending from one ship to another is wrapped in a special method in the class ship. Its special requirement is to avoid to dead lock because at the same time other ships run on the executor and simultaneously do transactions. Hence, the classic problem of a circular wait between to threads can happen. To illustrate it for this use case, imagine, there is ship A and ship B. When A invokes its transfer method and B invokes it vice versa, for instance \texttt{A.transfer(new\ Msg(A,B,1))} and \texttt{B.transfer(new\ Msg(B,A,1))} is invoked simultaneously, then a deadlock can occur. In \texttt{A.transfer(...)} the object A and B is locked before the transaction is done and in \texttt{B.transfer\ vice\ versa}. When there is no total order of locking, it can happen that A is locked by \texttt{A.transfer(...)} and simultaneously \texttt{B.transfer(...)} locks B. Now both methods will wait for an infinity time for each other to get the required second lock. In order to avoid this problem, the implementation employs a total ordering of locks. Each ship got a unique id and in the transfer method, always the ship with the lower id is locked first. This prohibits any possibility for circular waiting.

Determining the end of execution is possible through the futures, returned by the ExecutorService.

### 9.1.2. Implementation with Vert.x

In contrary to pure Java, only one implementation of the transaction problem was developed in Vert.x. The reason for that is Vert.x offers only one approach to handle concurrency as described in 8.2.1. The implementation consists of two verticles, one of the verticles sends all transaction messages and the other receives them while taking care about all ships. Transaction messages are normal Strings because Vert.x does not allow to pass whole Java objects. Thus, the receiver must additionally parse a string to retrieve the required information of each transaction. The following code snippet shows how the transfer handler is implemented within a verticle and how it is registered to listen to an address.
9.1. Requirements and Specifications for Transactions between Spaceships

```java
Handler<Message<String>> transferHandler = new Handler<Message<String>>() {
    public void handle(Message<String> msg) {
        transfer = msg.body();
        split = transfer.split(",");
        idFrom = Integer.parseInt(split[0]);
        idTo = Integer.parseInt(split[1]);
        amount = Integer.parseInt(split[2]);
        transfer(fleet.get(idFrom), fleet.get(idTo), amount);
    }
};

eb.registerHandler("transfer", transferHandler);
```

**Code 9.1:** Shows implementation of transfer handler and its subscription to the event bus

When all transfer messages are sent the sender transmits a message containing the String "sentAll" which we call final message from this moment on. It informs the receiver that all work has been sent. Later, we need to know when execution has finished to measure the execution time and other properties. Anyway, this approach is only possible because when using the event bus locally, events send from one verticle to another arrive in the same order at the receiver. This does not apply for messages pushed on a distributed event bus. At the beginning, there was another idea, to represent each ship by an verticle. But this notion had to be rejected soon. The main problem is, a transaction between two verticles has to be atomic but implementing this is basically not supported in Vert.x. Transactions across several verticles which either commit completely or not at all has to be implemented by yourself. Hence the idea was omitted, because this implementation would be difficult and it is essentially implementing your own transaction protocol. It is essentially implementing your own transaction protocol.

### 9.1.3. Implementation with Akka

Three different implementations of the problem have been created for Akka. Their difference are not tremendously because each is based on the actor model. In all three cases, used actors extend the `UntypedActor.class`. The first solution is very similar to the solution of Vert.x. There is one actor which sends all transaction messages and the receiver, referred as transaction handler now, processes each incoming message by applying the transaction on the ship objects stored in a list. By sending a final message (see 9.1.2), the sender indicates that there are no more transactions to be processed. When the transaction handler reads the message, it is ensured that all work is done and the whole actor system can be shut down. The approach, to mark the end of execution, is only possible because the actor system of Akka ensures for local execution that messages sent by one actor to another arrive in the same order at the receiver. Thus the final message message will really arrive as the last message.

After implementing the first solution the idea came up it could be faster to send the transaction messages from different senders concurrently to the transaction handler. Therefore, several senders split up all transactions among each other. Same as before, when a sender has sent all messages it sends a final message to the transaction handler. This time, the transaction handler must know how many senders exist in advance, to be able to know if he received all final messages.

[^1]: [https://groups.google.com/forum/#!topic/vertx/YTTtV7vIo04](https://groups.google.com/forum/#!topic/vertx/YTTtV7vIo04)
9. Implementation

The last approach with Akka tried to exploit the provided STM combined with the actor system. In detail, there is again one sender dispatching transfer messages to a router which distributes them to its routees. The applied routing strategy is sending messages always to the router which has currently the fewest amount in his mailbox. One ship is represented by one UntypedActor. Each routee is a normal UntypedActor which creates a Coordinated object when it receives a transfer message. After that, several atomic actions are appended to the coordinated object. The first is, the routee sends the coordination including the correct information about the change to both ships. Here some code, to illustrate this part of the routee.

```java
private void doTransaction(TransferMsg msg, Boolean retry) throws Exception {
    final TransferMsg transfer = (TransferMsg) msg;
    final Coordinated coordinated =
        new Coordinated(new Timeout(5, TimeUnit.SECONDS));
    final ActorRef from = transfer.getFrom();
    final ActorRef to = transfer.getTo();
    coordinated.atomic(new Runnable() {
        public void run() {
            to.tell(coordinated.coordinate(new ChangePeople(transfer.getChangeto()), null));
            from.tell(coordinated.coordinate(new ChangePeople(transfer.getChangefrom()), null));
        }
    });
}
```

Code 9.2: Shows how coordinated messages are used to handle transactions

The second and third atomic action is added to the transaction when both ship actors receive the coordinated message. Only when all three atomic actions succeed the transaction succeeds, if not the whole transaction is retried. Since all transfer messages are chosen carefully so that no limits are exceeded I set the number of retries to Integer.MAXVALUE, in order to guarantee that all implementations terminate with the same result. To indicate that all transfer messages has been sent, a final message is forwarded to each routee. When all routees have replied to this message, the sender knows the execution has finished.

9.1.4. Implementation with GPars

With GPars, two different implementations have been created. Like in Akka, one is based on the actor system and the other uses STM. Let us have a look on the approach using actors. Since there is no big difference to the other actor implementations, thus we just state them briefly.

There is one sender and one transaction handler taking care about all ships by applying transaction messages. Transaction messages are also plane objects in GPars, there is no need for any transformation. Actors extend the DefaultActor class. In order to indicate the completion of sending all transactions, the sender transmits a final message exactly like in the other implementations with actors. After receiving the final message the application terminates.

For the second implementation, the STM "Multiverse" was employed. Each ship obtained a STM-reference storing its amount of people on the ship. Like in the Java multithreaded solution, the class ship implements the Callable interface and is executed on a thread pool. When its call
method is invoked, each ship tries transferring from itself to the others. For each transaction the
ship starts an atomic STM transaction in which the STM references of both ships, involved in
the transfer, are updated. When an transaction fails because the STM could not get the required
locks (see 8.2.2), the basic behaviour is to do an automatic retry. The number of retries is initially
set to 1000 which has not been changed because it was high enough to ensure all transactions are
eventually applied.

9.2. Requirements and Specifications for Frequency of Visits

Despite what we determined in section 9.2, some more requirements for implementing the
problem “frequency of visits” exist. Like the fixed sequence of transactions, it is necessary to
have a fixed set of users, thereby ensuring each implementation has to process an equal amount
of work. Therefore, the users are created from plain text files storing the sequence before the
actual execution starts. The visited places are stored in a list for each user. For example, a list
could look like \(l=[1,2,3,4,2,3,1,14,...]\). Each place is represented by a number and might occur
multiple times in each list. As the result of processing all users some sort of map is returned
containing the overall amount how often each place has been visited. Referring to benchmarking,
it is necessary to include the possibilities to estimate required quantitative properties. Lastly, like
specified before, the problem is solved locally. The requirements are summarized below:

- fixed sequences of visited places
- all users in memory before starting execution
- one visited place is represented by a number
- include mechanisms required for benchmarking (execution time, throughput and memory)
- the problem is executed locally on a single machine

This problem is implemented for pure Java, Akka and GPars. There is no implementation for
Vert.x, not because it is not possible, but it imposes a lot additional expense. To dispatch users to
verticles, you would need to transform them into easier types before you can send them. This
additional overhead through transformation, although running on one local JVM, speaks against
using Vert.x here.

9.2.1. Implementation with Java

To have a datum to compare the performance of all other solutions against, the first implementa-
tion in Java is processing all users simply on one thread without introducing any parallelism.
After reading the sequences and creating the user objects, the thread starts to process the list of
users. In one global hashmap the thread stores how often each place has been visited. For each
user, it loops through the sequence and updates the information in the hashmap.
Since the actual properties of the problem allow to process it in parallel, the second solution with Java employs the fork/join framework of Java to exploit all cores of a machine. The fork/join framework is based on a special thread pool in which each thread has its own task queue besides the main queue of the pool. A work stealing algorithm on top is the main trait. When the task queue of an thread gets empty the thread looks in the other task queues if there is some work left it could steal. If there is no work it takes the next task from the main queue [2, 29]. Threads can split tasks into new subtasks and push them to their task queues and let the "mother" task join until the result is available. The fork/join pool can improve performance compared to common executors when they really create subtasks. Then the overhead to handle race condition of threads accessing the main queue occurs less often.

Back to the implementation, the first step is to create a ForkJoinPool and a ConcurrentHashMap for storing how often each place has been visited. Then the initial task is created containing the list of users. When the first task is executed it creates a new subtask for each user of the list. In each subtask the sequence of the passed user is processed and the global ConcurrentHashMap is updated. At this point you need to know that updating a ConcurrentHashMap has its limit referring to concurrency. In the first attempt, for each place in a sequence of an user, the subtask extracted the current value for this place from the hashmap and set the new value to the last incremented by one. However, this was not threadsafe for a hashmap from type <Integer,Integer>. The reason for that is, concurrent hashmaps ensure to be thread safe for writes as well as reads but not for casual dependent reads and writes. That means between reading a value and increment it, other threads can also read the last value which leads to the well known problem called "lost updates". To prevent wrong updates, you could either lock the whole hashmap or impose a synchronized access to each value of the hashmap. The first solution does not scale very well because only one thread could update the hashmap simultaneously which is a contradiction to the notion of a ConcurrentHashMap. Hence, the second option was implemented. Instead of Integers, AtomicInteger were employed as values of the hashmap. The code below shows the critical part were concurrency issues are prevented.

```java
    void computeDirectly(User user) {
        AtomicInteger result;
        for (Integer i : user.getVisitedPlaces()) {
            result = map.putIfAbsent(i, new AtomicInteger(1));
            if (result != null) { // in the mean time another thread set the value
                map.get(i).incrementAndGet();
            }
        }
    }
```

**Code 9.3: Critical part of Fork/Join solution**

Let us have a closer look why the code does not suffer from concurrency issues. the method putIfAbsent is thread safe. It returns null if there was no value stored in the hashmap before and it could successfully set the passed value. If another thread set the value in the meantime the method returns the value. Subsequently, when the method returns a value, we know this value must be incremented, but when null is returned the corresponding thread could initialize the value with one. The trick to guarantee thread safety between retrieving a value and increment it is caused by incrementAndGet. The method atomically increments and returns the AtomicInteger. Therefore, no lost updates can occur while multiple threads can still update different values of the hashmap simultaneously. To be exact, the AtomicIntegers must be placed in different parts of the map to be accessible simultaneously by multiple threads. The main thread invoking the
execution on the ForkJoinPool blocks until the work is done, thus you need no additional effort to determine the finish of execution.

### 9.2.2. Implementation with Akka

The implementation with Akka employs again the actor system. First of all, the actor system is created. On its top an actor, we call master now, is deployed which is a DefaultActor. The master gets the list of user as well as the number how many mappers and reducers he should create. Mappers and reducers are both DefaultActors. Each mapper receives messages which are a user object from the list. They create a `HashMap<Integer, Integer>` in which they store the information of one processed sequence. When one user is processed they send each key-pair (place: visited n times) of the map to one of the reducers. To which reducer they have to send the key, is determined by the key modulo the size of the list where all reducers addresses are stored (see the code below). That guarantees that one receiver gets one specific portion of all possible keys.

```java
for (Entry<Integer, Integer> entry : frequency.entrySet()) {
    reducers.get(entry.getKey() % reducers.size()).tell(entry, null);
}
```

In each reducer, the incoming key-value pairs are summed up in an own hashmap. Once, the master has sent all user objects to the mappers, he sends a final message to the mappers. Mappers react on the receive of this message by forwarding it to all reducers. The reducer knows how many mappers there are and counts the number of received final message. If the number equals the number of mappers all work is done. As the final step, the reducers send their partial results back to the master which puts the disjoint hashmaps together to the final result. The moment, in which the master received from every reducer the result indicates the end of execution after which the actorsystem is shut down.

### 9.2.3. Implementation with GPars

For the first and only time, the language in which the implementation is written is not Java, instead Groovy. The reason for that was, in the documentation there is a section about parallel processing in map reduce style. Since GPars is written in Java under the surface, the first attempt was to find out how to implement the map reduce style by using the underlying Java code. However, this section in the documentation targets only groovy. In the "Java" API, text still targets how to use it with groovy DSL which does not help much. Hence, the decision was made to write it with the groovy DSL plus a small part of real groovy code.

Referring to the length of the solution, you can see that groovy makes code really short. The following method solves the entire problem.

```
def mapreduce(arg, poolSize) {
    GParsPool.withPool(poolSize) { // run on default pool with size
        return arg.parallel.map { // create parallelArray of user list(arg) and apply map
            function for each entry
        }
    }
}
```
9. Implementation

```groovy
it.visitedPlaces.groupBy({it}).collectEntries { key, value -> [key, value.size()] } // group the sequence by their keys, instead of list:value=size of list
	.reduce { a, b -> def newMap = combine( a, b ) }

//collectMany(Closure projection): Projects each item from a source map to a result collection and concatenates (flattens) the resulting collections adding them into a collection.
//inject(Object initialValue, Closure closure) Iterates through the given Map, passing in the initial value to the 2-arg Closure along with the first item (or 3-arg Closure along with the first key and value)

def combine( Map... m ) {
    m.collectMany { it.entrySet() }.inject([:{ }]) { result, e ->
        result << [(e.key):e.value + (result[e.key]?:0)]
    }
}
```

**Code 9.4:** Entire solution with GPars for Frequency of Visits

When the method `mapreduce` is called the user list is converted into a parallel array which is then processed with a fork join pool. For each user in the array, the sequence is grouped by its keys. The grouped keys are then transformed into a hashmap with entries of type (key:sizeOfGroup). In the reduce step, two maps are combined to one until one map is left which constitutes the final result.

9.3. Requirements and Specifications Counting Server and Computation Intensive Server

We start determining the requirements which concern both servers. The first is, both servers use HTTP. When receiving a simple but valid "GET" request they counting server will return the number of the request whereas the computation server will compute PI every time up to a certain approximation. How many decimal places are computed depends on the number of times the Leibniz formula (look at 6.2.1) is executed. This was kept adaptable to ensure it can be changed when doing benchmarks. After computing, a HTTP response is replied from the server containing either PI or the request number. All TCP connections can either be kept “alive” or not. The former could improve the performance, because the client will send multiple request in a row, thus disconnecting and reconnecting after each request would be much overhead. Whether this assumption is confirmed or not is revealed in chapter 10.3 On server side non blocking I/O has to be used. Non blocking I/O (NIO) guarantees that all socket communication is done asynchronously. A dedicated thread pool is looking continuously for new incoming messages, without blocking. Advantages of NIO are, there is less overhead due the lack of costly context switches between threads. Furthermore, it is possible to have more connections simultaneously since one JVM allows an upper limit of threads. All requirements and specifications are summarized below.

Both
9.3. Requirements and Specifications Counting Server and Computation Intensive Server

- HTTP as protocol
- Non blocking I/O
- keep connections alive
- the problem is tested on a small network (one server and one client)

Counter Server

- HTTP response with number of requests

PI server

- HTTP response with computed PI (intense/precision of computation is configurable)

This problem is implemented for pure Java, Akka and Vert.x but not GPars. In contrast to Akka and Vert.x, GPars does not provide features like routers. However, the computing server would require them to dispatch the requests but implementing your own configurable and scalable routers single-handed was omitted due to its effort. Relating to feasibility of the counting server, applying the same solution for GPars like the one for Akka was easily possible. But no additional information whether we use Akka or GPars actors for the counter were expected. Their difference in performance is already estimated through transactions between spaceships. Thus, it was decided to reduce the overall effort by relinquish GPars for the counting server.

9.3.1. Implementation with Java

Both server implementations are based on the netty 4.x framework for NIO blocking network communication. Two different thread pools are responsible for the I/O work. A boss pool listens to bounded sockets and forwards established connections to the worker thread pool. The threads from this pool perform the reading and writing non blocking for all server sockets. On top of each connection a channel is attached. Incoming as well as outgoing messages are passed through the pipeline of the channel. In the pipeline you can register handler which transform the message and forward it to the handler. All of this work is handled by the worker thread pool. The following pipeline was created to transform incoming messages to HTTP objects.

```java
public void initChannel(SocketChannel ch) throws Exception {
    ch.pipeline().addLast("logger", new LoggingHandler(LogLevel.TRACE));
    ch.pipeline().addLast("codec-http", new HttpServerCodec());
    ch.pipeline().addLast("aggregator", new HttpObjectAggregator(65536));
    ch.pipeline().addLast("requesthandler", new HttpRequestHandler(HttpServer.this));
}
```

**Code 9.5:** Shows the pipeline each HTTP request and response passes through

The handler "codec-http" converts incoming bytebuffers into HTTP objects and outgoing HTTP objects to bytebuffers. "aggregator" composes several "chunked" HTTP messages to one Object. Our counter is placed in the HTTP server instance. It is accessed from the HttpRequestHandler synchronously. More precisely, the access to the variable which stores the number of requests
is incremented and retrieved in a synchronized method. After determining the number of the request, it is wrapped in a HTTP response Object, before it is finally sent back to client.

Exactly the same implementation is used for the server calculating PI, except one difference. Instead of retrieving the request number, the HTTPRequestHandler computes PI and sends the answer back.

9.3.2. Implementation with Akka

Originally, it was planned to use the features such as remote actors directly to implement the HTTP servers. Yet, this imposed to implement the HTTP protocol completely or use a third party to provide it. Subsequently, the implemented server from the pure Java solution was reused. Then on top of the server, the actor model is incorporated. This has been the actual solution finally. In contrast to pure Java, the server has additionally an instance of ActorSystem. Each request handler makes use of the actor system. In the case of the counting server, one actor represents the counter. Request handler send a message to this actor and wait for its reply containing the number of the current request. Then this number is sent back in an HTTP response to the client. Similar to this, PI is computed by actors instead of directly by a request handler. The amount of actors to calculate PI are configurable at initialization. To sum it up, all network communication stayed the same as with pure Java, but the actor system is put on top.

9.3.3. Implementation with Vert.x

The main purpose of Vert.x is to create web application. Therefore, the implementation with Vert.x needed much less effort than the others. First, we describe how the counting server was implemented. Inside one verticle a HttpServer instance is created listening to a passed ip and port. Additional, a handler to reply to incoming messages is implemented. Inside the handler, an event is dispatched to the verticle which counts all demands. The response of the counting verticle is processed by another handler. This handler finally sends back the response to the client. To be able to scale up the server, the number of deployed verticles, described above, can be set at initialization. Under the covers, Vert.x deploys netty for its asynchronous network communication.

The server calculating PI has two differences to the counting server. Multiple verticles are deployed to calculate PI, instead of one counting verticle. The exact number is also adaptable to be able to tune performance later. Second, During measurement later, the verticles to compute PI were either deployed as worker verticles or normal verticles to see if this makes any difference. Besides that there are no differences. The request handler retrieve the result of PI like the number of request was obtained in the counting server.
9.4. Summary

As its most important requirement, each implementation has to be implemented, so that it can be evaluated through benchmarks. Transactions between spaceships and frequency of visits is executed locally on a single machine while the server is tested in a small network. To have an basis to compare implementations against each problem is implemented with pure Java. Transactions between spaceships is not programmed with GPars because is lacks support for distribution whereas Vert.x is not employed for frequency of visits since it allows only simple types for messages which imposes only additional effort.
10. Quantitative Comparison

After discussing all implementations, this chapter deals with estimating their performance. First, the inherent difficulties doing benchmarks on the JVM are explained. Then we discuss how the benchmarks were actually performed. Finally, the results are gathered and evaluated.

10.1. Benchmarking on the JVM

There are two different kinds of benchmarks, either micro benchmarks or macro benchmarks. Although, there is no clear definition, the former usually means to evaluate the performance of small specific code snippets whereas the latter usually includes an entire application. In general, writing micro benchmarks is very difficult on the JVM because you need a deep understanding how the Just-In-Time (JIT) compiler of the JVM works [5]. These compilers are able to perform various optimization which is generally desirable but in the artificial environment of a benchmark it can distort all results. Let us take some examples for optimization and explain why they can be harmful to a benchmark. Consider the following method:

```java
public static void main(args[]){
    long start = System.currentTimeMillis();
    calculateSomething(1000000); // return value never used -> candidate for elimination
    System.out.println((System.currentTimeMillis()-start));
}

public static int calculateSomething(int num){
    int result=0;
    for(int i=1; i<num; i++){
        result+=num%num;i;
    }
    return result;
}
```

**Code 10.1:** Example for dead code elimination

Above is an example for dead code elimination which distorts your results. Since the integer returned by `calculateSomething()` is never used, it is possible that the compiler eliminates the whole method because it has no influence on the result of the execution. To prevent dead code elimination you should avoid letting results untapped. Another optimization is applied when methods are called very frequently. Then they are called "hot", thereby their byte code is no more interpreted, instead it is compiled to improve the performance. It is obviously what happens when you compare two methods and one is compiled and the other not. The consequence is, it is hard to determine which method is the faster one, when they are executed differently. This
Quantitative Comparison

is especially a problem because the JVM sometimes undoes compilation and starts interpreting again when keeping the compiled code would cause wrong behavior. The same applies to other optimization such as loop unrolling, method inlining, lock coarsening etc. [6]. As you can imagine, there is much potential to produce distorted results in micro benchmarks. On the contrary, when performing macro benchmarks you try to produce realistic test scenarios for the application in order to get results which describe the applications behavior in real situations. That means you do not worry about optimization and let it just happen. The next section describes what kind of benchmarks was actually employed.

10.2. Actual Benchmarks

All implementations are evaluated with a benchmark which is a mix of micro and macro benchmark. Some basic rules for micro benchmarks, as described on [7], are observed but whole runs of the problem are measured instead of code snippets which is rather a macro benchmark then. One of these basic rules for micro benchmark is measuring performance only after a warm-up phase. During that, the problem is ran several times without measuring any property. This is because the initialization of a whole application requires a lot of time. If you measure directly the first run your results are usually way slower than they were if an application has already completed several runs. Additionally, some JVM flags were used. For instance, the flag “-sever” for the compiler entails more intensive optimization than the "-client" mode. Other flags ensure that each implementation runs with the same heap size in order to ascertain nobody obtains any advantage through different amount of memory. However, other rules were not regarded such as printing compilation or information about garbage collection. This is mainly due to the whole measurement being written to run automatically. As the result of one run after a warm up phase, all measured properties where printed out which was directly consumed by another process. This process collected several measurements to calculate a mean before it continued with a bigger problem size. Any output through garbage collection or compilation would have been disturbing. Furthermore, it was measured so many time that it would have been impossible to regard all the output information. Since printing compilation or garbage collection was turned of, the used benchmark is somewhere between micro and macro. We let the compiler and garbage collector do their work as they would in real situations. Consequently, it is not excluded that both influence the measured run after the warm-up phase which is called timing phase. For instance, in one timing phase the garbage collector starts to run whereas in the second it does not. In consequence, the first timing phase requires more time than the second. Similar to this, other programs running on the same computer simultaneously could influence the measurement. Programs could possibly distort results such as anti virus programs as well as not required network connections were shut down. In order to diminish influences of unexpected events, all results are a mean of multiple measurements. For the web servers the mean consisted of four consecutive measurements for the other problems it depended on the computer used. The laptop employed fewer values because it is slower and it would have taken to much time otherwise. Moreover, the implementation certainly have room for improvement left. Initially, the idea was to implement all problems and ask the communities and developers of the frameworks to give advice where is space left for improvement in the code. Therefore, some problems were put on GitHub to facilitate reading and modifying the implementations for others. Yet, the feedback was not as useful as hoped. Aside from suggestions to employ
specific other tools for that there was no feedback at all after waiting approximately three weeks. Considering all possible influences above, the benchmarks need to viewed as the following. They are benchmarks of implementations, done by someone which programmed the first time with the frameworks. Furthermore, the implementations are measured in a real environment. Hence, it might happen that this environment distorts the results to some degree. If results seem to suffer from an unexpected event the measurement is repeated to check if the results stay the same. For example, when all partial values of a single measurement seem to stick to an obvious behaviour but one stands totally out. Now, we look closer how each problem was evaluated.

The problems were tested on two different computers, a laptop and a desktop computer. In the case of the counting or computation intensive server, the desktop computer was the server while the other was the client. Table A.2 shows details about both computers. As you can see, the desktop computer offers more power than the laptop which should later be noticeable in the gained results.

<table>
<thead>
<tr>
<th>Property</th>
<th>Laptop</th>
<th>Desktop Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>processor</td>
<td>Intel(R) Core(TM) i5 CPU M450 @2.40Ghz</td>
<td>Intel(R) Core(TM) i7-3770 CPU @3.40GHz*8</td>
</tr>
<tr>
<td>working memory</td>
<td>4GB</td>
<td>16GB</td>
</tr>
<tr>
<td>operating system</td>
<td>Windows 7</td>
<td>Ubuntu(12.04)</td>
</tr>
</tbody>
</table>

Figure 10.1.: Basic information about the used hardware

Measurement of Servers

The Counting Server and Computation Intensive Server were measured with the apache benchmark tool \(^1\). It allows to send basic HTTP requests to an address. Furthermore, you can define how many messages are sent, how many sockets are concurrently used and if connections are kept alive or not after each message. The most important properties measured by the tool are the mean processing time, transfer rate and time per request of the server. The precise meaning of the properties is explained in table A.1 Before each server was evaluated a few thousand messages were sent as a warm-up. For the actual test, each run consisted of one hundred thousand messages sent to the counting server. Four consecutive runs with the same amount of concurrently employed sockets were done to determine a mean value. After these four runs the amount of concurrently used sockets was increased. In the case of keeping connections alive 100,000 messages were sent. On the other hand when connections were closed after each message, 20,000 request transmitted because it takes a lot more time. Referring to the calculating server, the number of requests decreased to 2,000 for both cases because it requires even more time for a single request.

Measurement of local running Problems

\(^1\)httpd.apache.org/docs/2.2/programs/ab.html
Transactions between Spaceships and Frequency of Visits were measured both locally on two different computers (see A.2). The same properties were determined for both problems. These were the processing time for a whole run, the throughput and the total memory which was allocated by the JVM. The actual used memory was not determined because it can diversify very strongly depending on how long the last garbage collection is ago. But the total memory amount stays the same after garbage collection. Nevertheless, its information is important since the total memory allocated is no more available for any other processes. More information such as unit and meaning of the measured properties are described in table A.1. In order to determine the course of the curve for a property the size of the problem was systematically increased for each measurement. One measured value is the mean of five consecutive runs except the measuring on the laptop of Frequency of Visits. There it consisted only of two values because it would have taken too much time otherwise. This is because a single measurement includes multiple warm ups. In our case, a whole problem in its given size was executed five times as a warm up before one run was measured. Subsequently, for each mean value an implementation was ran five times six (five warm-ups plus one measured run). As an exception, Frequency of Visits for the laptop used again only two warm ups.

10.3. Results

First of all, the entire results we discuss in this section are placed in the appendix A. In table A.1 you can find what each measured property exactly means. A short description is attached to each figure which describes what can be observed. Hence, we focus on summarizing and interpreting them with regard to their validity. We start with the servers, continue with transition between spaceships and eventually we discuss frequency of visits.

Counting Server and Computing Server

For both cases, the counting server and the computing server, closing connections after each request increased the total time compared to keeping connections alive. Apart from that, the results gathered when connections were closed every time, suffered from high variance. All three courses contain many knees and the ranking which is faster changed often. The cause for this can be ascribed to the inherent variance in opening and closing connections as it does not occur when connections are kept alive. Thus, results with closing connections are not used to make further conclusions. We will infer in the following based on the results made with connections kept alive.

On the counting server, increasing the amount of concurrent connections caused an increase of the total processing time on the server. This is because, the counter is a bottleneck. For every incoming request the counter has to be increased and then demanded concurrently for the number. But it is impossible run the counter in parallel since it requires atomic access. Otherwise lost updates are bound to occur. Altogether, Vert.x and Java showed approximately equal total processing time on the server. Akka tent to be slower (at most it needs 34% more time, absolutely 11 ms among all requests). For a programmer, these results mean, as long as your application is fast enough you should prefer the approach to concurrency you think to be more productive in terms of error-proneness and developing time. But if performance becomes really an issues,
you should try to detect bottlenecks caused by concurrency and try if other approaches may be improve the overall performance.

In contrast to the counting server, there was no bottleneck by concurrency on the server calculating Pi. The results contained two interesting outcomes. First, Vert.x with normal verticles to compute Pi was way slower than Vert.x deploying workerverticles instead. The explanation \(^2\) is, Vert.x employs worker verticles which increases the overall throughput. If standard verticles are deployed, 16 verticles compete for 8 event loops in our setting. If a computing verticle is blocked in a long running computation the other 8 verticles are prevented from serving HTTP requests because there are more verticle instances than available event loops. As the second outcome, the additional effort of utilizing the actor system could not be recognized. In pure Java, the request handlers from Netty directly computed Pi and sent back the reply whereas in Akka the request handler demanded an actor to perform the computation. Akka was even minimally faster than Java. Either this is due to the inherent imprecision of the measurement or similar to Vert.x with workervertciles, including another thread pool for serving the actors improved the overall throughput. All in all, the results definitely prove that fine tuning by increasing the number of employed threads, can have huge impact on performance. Unfortunately, there is no standard procedure for this kind of tuning other than trial and error while observing if performance improves.

Transactions between Spaceships

From all implementations Akka STM exposed the worst performance referring to the total execution time. Second worst, was Vert.x followed by the others employing actors. Among the actors itself it could not be determined whether GPars is faster or Akka. On the desktop GPars actors have been faster while on the laptop Akka actors performed better for many transactions whereas GPars better for few transactions. To figure out which on is really faster, a dedicated micro benchmark would be necessary taking care about any possible influence on the JVM. Besides that, the STM Multiverse which GPars employs, is on the laptop nearly as fast as Java single-threaded. Furthermore, on the desktop computer it is even close to the perfomance of Java multi-threaded which was the fastest of all.

The approach to determine the required memory was less reliable as expected. Non deterministic timing of the garbage collection but also the strategy when to allocate from and return memory back to the operating system caused a variance which was useless for detailed comparisons. Nonetheless, the results indicate that message passing concurrency tends to require more memory since, except Akka STM A.23, all concerning implementations required way more memory than the implementations based on shared state.

In terms of throughput, both Java solutions, GPars STM and Akka STM have an increase of throughput up to a certain maximum. After they reach saturation they constantly keep the throughput. On the contrary, the others entail a decrease of throughput after the maximum throughput was reached. The behaviour is probably related directly to the memory consumption. When more memory is utilized due to more transactions, the time where the garbage collector runs is increased. Consequently, the throughput suffers form that. Moreover, messages are obviously produced faster than they can be processed because the queuing would explain why more transactions cause more memory. If an actor or verticle produces messages these are

\(^2\) stems from a discussion with Tim Fox in Vert.x Google Group
stored until they are processed. Not until after processing, the garbage collector can free the memory again. As a programmer you should be aware of the fact that bottlenecks in processing messages are to the disadvantage of memory in message-passing concurrency.

More interesting facts could be figure out. Two verticals, one producing transactions messages, and one processing them, are slower than two actors independent whether these are GPars or Akka actors. Another outcome is, Akka and GPars actors cannot compete with pure Java. In the absence of locking (Java single-threaded ) as well as with fine grained locks (Java multi-threaded), pure Java was faster than any of actor implementation when many transactions had to be done. For small values GPars actors was sometimes even faster than Java single-thread (see A.16). This changed very likely because of the influence of the memory for bigger problem sizes we discussed before. However, none approach based on messaging could compete with fine grained locks because this allows to execute independent transactions simultaneously. Finally quite surprisingly, the STM Multiverse performed very well. On top of a thread pool it could nearly keep pace with fine grained locking.

**Frequency of Visits**

Like before, we open with summarizing the results followed by an interpretation. GPars required the most time followed by Akka with quite some difference (see A.19 and A.22). Once more pure Java was even faster. On the laptop only within a certain interval Java Fork/Join outperformed Java single-threaded . In contrast to that, it always beat Java single-threaded on the desktop computer except one surprising knee. Unfortunately, there is no good explanation for that knee. It could be caused through unforeseen influences such as other processes on the computer but it is very unlikely because three independent measurements, each a mean of five values, produced the same result. Another possibility is, the access to the concurrent hashmap is so adverse, that multiple threads block each other frequently. In consequence, the circumstances even deteriorate the performance due to the overhead by handling multiple threads.

Observing the memory produced results following nearly the same behaviour as the required time we discussed above. GPars allocated most, followed by Akka. Both Java implementations show quite small differences (see A.21 and A.23) but also required the fewest.

In terms of throughput, the rankings are equal to the one for the required time. On the laptop Java single-threaded and Akka decrease only lightly after they reached their maximum throughput whereas Fork/Join and GPars fall sharply afterwards. On the contrary, the results from the desktop computer are more constant. Java Fork/Join seems to have only just reached its maximum with the last measured value but its course also contains the knee explained before.

What could be actually figured out? The basic aim of each implementation was to exploit multiple cores to improve performance. Neither GPars nor Akka accomplished that goal since both were slower than Java single-threaded. Only Java multi-threaded with fine grained locks yielded an improvement. The results we discussed allow several conclusions to be drawn. First, both GPars and Akka were obviously not implemented optimally. In the reduce step of GPars, all hashmaps representing the places of an user were merged together. It might have been faster to use a data structure suitable for concurrent modification instead of merging them in parallel which creates each time a new hashmap. Also Akka suffered most likely from creating additional hashmaps and sending results as messages between mapper and reducer actors. Subsequently, we can
conclude the problem Frequency of Visits itself was not that appropriate as anticipated. Splitting up the work, through the actor model as well as creating many hashmaps while exploiting several cores did not outperform the simple solutions which relinquishes multiple cores but also the additional overhead for coordination. After splitting up the work, each part should have required still enough effort to be processed so that exploiting multiple cores outweighed additional overhead through coordination. Since the problem was embarrassingly parallel the actual problem size has actually be blamed being to small. The bigger the data, the more we should have gained advantage of multiple cores. Nevertheless, this is a very good example to illustrate how difficult it may be to exploit multicores. Referring to the frameworks, if you do not need to improve speed, you should rather prefer a single-threaded solution before you possibly even worsen performance.

10.4. Summary

We have learned that benchmarks are challenging on the JVM due to the optimization through the JIT compiler during execution. It can happen that elements you want to compare are optimized differently or code seeming to have no change on the result is even deleted. That can distort the artificial environment of a benchmark completely. The benchmark which was actually applied was a mix between a macro and micro benchmark. It stuck to some of the rules for micro benchmarking on the JVM such as doing some warm-up but the JVM was free to optimize as it would in real environments. Next, the most important results are summarized. From the measurement of the servers it could be concluded that using workerverticles instead of normal for long running or blocking operations can cause a tremendous boost of performance. Besides that, Akka actors suits well as part of a server application. However, to employ the actor model throughout the whole server other frameworks should be employed on top \(^3\). Transaction between spaceships showed that fined grained locking with pure Java outperformed actors as well as verticles. Furthermore, it was surprising how well the STM Multiverse, which GPars employs, performs. The STM combined with Akka actors should not be used for high amounts of transactions in short times. Whether GPars actors are faster or Akka actors could not be determined but verticles are slower than both actors because they have to serialize objects before sending them. Some other outcomes originate from frequency of visits. Neither GPars nor Akka makes it simple to exploit multiple cores and actually gain an improvement of performance since both solutions were slower than Java single-threaded. Relating to this, it must be concluded that the computing intensity of the problem itself might have been to small. Splitting and distributing work cost more time than could be regained through the additional power of multiple threads except with Java Fork/Join which used one global ConcurrentHashMap.

\(^3\)http://spray.io/ or http://typesafe.com/platform/runtime/playframework
10. Quantitative Comparison
11. Conclusion

Lastly, we merge the qualitative and quantitative results we made so far to draw some final conclusion about the frameworks. By referring to the characteristics of concurrency problems from chapter 6, it is concluded which framework suits which problems. Before that, we briefly reflect getting started with each framework.

When you start programming with a framework, the time you need to familiarize yourself with it should be as short as possible. Hence, it is essential that the framework provides good usability. Referring to that, Akka outweighs GPars and Vert.x in quantity as well as quality. When implementing the problems, it was mostly faster and easier with Akka to figure out information I missed in that moment. But for which concurrency problems are the frameworks suitable?

The two server, we have discussed, had both a high I/O intensity. Determining Pi for each request had a moderate capability for parallelization. Based on the quantitative results, we could determine that Akka with actors as well as Vert.x are appropriate to implement problems which can be parallelized while having a high I/O and computing intensity. This is because the additional effort of dispatching requests through actors or verticles is negligibly small compared to the time required for I/O as well as computing. Both implementations can keep pace with pure Java. Additionally, they provide implicitly modularisation which is essential for developing maintainable and extendable software. More particular referring to programming, verticles as well as actors can easily be employed to form the logical base of a server application in which multiple cores want to be exploited. Actors from GPars theoretically can accomplish the same but impose additional effort due to the lack of routers. Thus, it is recommended to use either Akka or Vert.x for this type of problem. If you have the need of fault-tolerance for a specific problem, it is a case for Akka whereas Vert.x allows a faster draft of an application.

Compared to the computing server, frequency of visits even has a higher capability for parallelization but less computing intensity. Depending on the actual size of the problem an implementation may required to distributed. If that is the case, GPars discards and I recommend to use Akka because it combines the possibility to scale out but also to scale up. Vert.x also suits but since you can only pass around simple types it can be faster but especially more convenient sending any object as a message as it is possible in Akka. When the problem size is small enough it can be solved locally. In particular, we could learn one lesson from frequency of visits. Although a problem might be embarrassingly parallel, it may happen that you do not gain an improvement when you try exploiting multicores. Instead, you even deteriorate the performance. For you as programmer this means, you have to balance the pros and cons whether it is necessary to exploit multiple cores or to better rely on more robust and simpler single-threaded solutions. However, since the trend is towards more and more cores in a single computer, in near future you will certainly have to focus on concurrency even more than now. But how should we actually implement an embarrassingly parallel local running problem on the JVM then? We could show
that actors as well as concurrently shared data structures can be employed. The former has the advantage it is less error prone in terms of thread safety. On the other hand, actors may mislead to reduce a problem to too many pieces so that you do not gain an improvement at all due to the additional effort of dispatching the pieces of work. However, I think the actor model is natural to describe such kind of problem. Splitting up work into smaller pieces and sending them to specialized actors is way more intuitive as a system than employing concurrent data structures and worrying about thread safety.

The last problem, transactions between spaceships, had a low capability for parallelization but on the other hand more amount of state than the others relating to concurrency. The problem itself represents a situation in which multiple objects are very frequently modified concurrently. In that case, pure native multi-threaded Java beats actors and \texttt{verticles}. Fine grained locking allows to apply multiple modifications simultaneously which is why it outperforms the other approaches. A combination of STM and actors enables to describe the same behavior but resulting performance advises using it only rarely but not for high frequent transactions. Actually the best solution is to employ a STM such as Multiverse which GPars utilizes. Relating to speed, it could keep pace with fine grained locks. The combination of good performance while preventing a programmer from struggling with locks directly plead to use a STM for concurrency problems with theses characteristics.

Finally, we make a small summarize what framework suits which characteristics of concurrency problems. Problems which have a high I/O intensity because they have to be distributed are recommended to be implemented with Akka or Vert.x but not GPars. Embarrassingly parallel problems is a case for the actor model as long as the problem size is big enough. You should also prefer Akka to GPars actors due to the amount of support and offered features. Problems with a high amount of state while running locally is what you should tackle with a STM like Multiverse. Though, a real problem will exist mostly of a combination of these characteristics. Consequently, a mix of the approaches might be the best way to encounter it. A final example shell illustrate that. Imagine, the given problem requires to update multiple objects concurrently and very fast but only at certain points of time. Additionally, some smaller part of your application insists on high robustness whereas other code changes or extends frequently. To implement that it would be reasonable to combine different approaches. For instance, the concurrent updates across entities could be tackled with a STM backed by a thread pool to ensure they are fast. The part of the application demanding to be highly safe can either employ dataflow or even CSP libraries such as JSCP to prevent deadlocks or for the latter allow a proof of correctness. The overall architecture and the modules which alter often could be implemented with actors. This final example intentionally exaggerates a bit by mixing so many different concepts and approaches but it shell clarify one thing. You as a programmer have the agony of choice which approach you take to handle concurrency. The ideas discussed in this thesis hopefully have facilitated that decision for concurrent programming on the JVM.
A. Appendix

<table>
<thead>
<tr>
<th>Property</th>
<th>Unit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Time</td>
<td>Milliseconds [ms]</td>
<td>Time server needed to process all requests or time required to solve one problem</td>
</tr>
<tr>
<td>Transfer Rate</td>
<td>Kilobytes/second</td>
<td>How many Kilobytes/second are received at sender</td>
</tr>
<tr>
<td>Time per Request</td>
<td>Milliseconds [ms]</td>
<td>Time required per request, mean across all concurrent connections</td>
</tr>
<tr>
<td>Throughput</td>
<td>Transition/ms or user/ms</td>
<td>Given problem size divided through required time</td>
</tr>
<tr>
<td>Memory</td>
<td>Megabyte [MB]</td>
<td>Total memory allocated by the JVM (not actual used memory)</td>
</tr>
</tbody>
</table>

**Table A.1.**: Shows the meaning of measured properties

A.1. Settings

<table>
<thead>
<tr>
<th>Property</th>
<th>Laptop</th>
<th>Desktop Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel(R) Core(TM) i5 CPU M450 @2.40Ghz</td>
<td>Intel(R) Core(TM) i7-3770 CPU @3.40GHz*8</td>
</tr>
<tr>
<td>Working memory</td>
<td>4GB</td>
<td>16GB</td>
</tr>
<tr>
<td>Operating system</td>
<td>Windows 7</td>
<td>Ubuntu(12.04)</td>
</tr>
</tbody>
</table>

**Table A.2.**: Basic information about the used hardware

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heapsize</td>
<td>2560 Megabyte</td>
</tr>
<tr>
<td>JVM Compilation</td>
<td>-server</td>
</tr>
<tr>
<td>Warm-Ups</td>
<td>5</td>
</tr>
<tr>
<td>Number of values for mean</td>
<td>5</td>
</tr>
</tbody>
</table>

**Table A.3.**: Configuration for transactions between spaceships on laptop
A. Appendix

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heapsize</td>
<td>14000 Megabyte</td>
</tr>
<tr>
<td>JVM Compilation</td>
<td>-server</td>
</tr>
<tr>
<td>Warm-Ups</td>
<td>5</td>
</tr>
<tr>
<td>Number of values</td>
<td>5</td>
</tr>
<tr>
<td>for mean</td>
<td></td>
</tr>
</tbody>
</table>

**Table A.4.:** Configuration for transactions between spaceships on desktop computer

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Special Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Single Threaded</td>
<td>none</td>
</tr>
<tr>
<td>Java Multi Threaded</td>
<td>ExecutorService with 4 threads used</td>
</tr>
<tr>
<td>Akka Single Actor Sends</td>
<td>• poolsize = 4 (cores) * 3 (standardfactor) = 12</td>
</tr>
<tr>
<td></td>
<td>• 1 sender</td>
</tr>
<tr>
<td></td>
<td>• 1 receiver</td>
</tr>
<tr>
<td>Akka Four Actors Send</td>
<td>• poolsize = 4 (cores) * 3 (standardfactor) = 12</td>
</tr>
<tr>
<td></td>
<td>• 1 sender</td>
</tr>
<tr>
<td></td>
<td>• 1 receiver</td>
</tr>
<tr>
<td>Akka STM</td>
<td>• poolsize = 4 (cores) * 3 (standardfactor) = 12</td>
</tr>
<tr>
<td></td>
<td>• 1 sender</td>
</tr>
<tr>
<td></td>
<td>• 1 receiver</td>
</tr>
<tr>
<td>GPars Single Actor</td>
<td>• poolsize = cores +1 = 5</td>
</tr>
<tr>
<td></td>
<td>• 1 sender</td>
</tr>
<tr>
<td></td>
<td>• 1 receiver</td>
</tr>
<tr>
<td>GPars STM</td>
<td>ExecutorService with 4 threads used.</td>
</tr>
<tr>
<td>Vert.x Single Verticle</td>
<td>• eventloops = 4</td>
</tr>
<tr>
<td></td>
<td>• 1 sender</td>
</tr>
<tr>
<td></td>
<td>• 1 receiver (standard verticle)</td>
</tr>
</tbody>
</table>

**Table A.5.:** Configuration of each implementation for transactions between spaceships on laptop

80
Table A.6.: Configuration of each implementation for transactions between spaceships on desktop computer
Table A.7.: Configuration of servers for different implementations. The size of any used thread pool was not altered from the standard size.

<table>
<thead>
<tr>
<th>Property</th>
<th>Counting Server</th>
<th>Computing Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>deployed verticles (Vert.x)</td>
<td>9 (8 HTTP 1 Counter)</td>
<td>either 16 standard (8 HTTP + 8 for Pi) or 16 (8 HTTP + 8 PI-workers)</td>
</tr>
<tr>
<td>standard size of workpool size</td>
<td>pool not used</td>
<td>20</td>
</tr>
<tr>
<td>standard number of event loops</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>deployed actors (Akka)</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>standard size of actor thread pool</td>
<td>available processors * factor; here 8*3 = 24</td>
<td>24</td>
</tr>
<tr>
<td>standard size of EventLoop-Group of Netty</td>
<td>2*available cores = 16</td>
<td>2*available cores = 16</td>
</tr>
<tr>
<td>heap size of server</td>
<td>14 Gigabyte</td>
<td>14 Gigabyte</td>
</tr>
<tr>
<td>JVM compilation</td>
<td>-server</td>
<td>-server</td>
</tr>
<tr>
<td>number of values for mean</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table A.8.: Configuration for frequency of visits on laptop

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heapsize</td>
<td>2048 Megabyte</td>
</tr>
<tr>
<td>JVM Compilation</td>
<td>-server</td>
</tr>
<tr>
<td>Warm-Ups</td>
<td>2</td>
</tr>
<tr>
<td>Number of values</td>
<td>2</td>
</tr>
<tr>
<td>for mean</td>
<td></td>
</tr>
</tbody>
</table>

Table A.9.: Configuration for Frequency of Visits Desktop Computer

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heapsize</td>
<td>14000 Megabyte</td>
</tr>
<tr>
<td>JVM Compilation</td>
<td>-server</td>
</tr>
<tr>
<td>Warm-Ups</td>
<td>5</td>
</tr>
<tr>
<td>Number of values</td>
<td>5</td>
</tr>
<tr>
<td>for mean</td>
<td></td>
</tr>
</tbody>
</table>
### A.1. Settings

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Special Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Single Threaded</td>
<td>none</td>
</tr>
<tr>
<td>Java Fork/Join</td>
<td>Fork/Join pool with 4 threads</td>
</tr>
<tr>
<td>Akka Actors</td>
<td>- poolsize = 4 (cores) * 3 (standardfactor) = 12</td>
</tr>
<tr>
<td></td>
<td>- 1 master actor</td>
</tr>
<tr>
<td></td>
<td>- 4 mapper</td>
</tr>
<tr>
<td></td>
<td>- 4 reducer</td>
</tr>
<tr>
<td>GPars MapReduce-DSL</td>
<td>pool with 4 threads used</td>
</tr>
</tbody>
</table>

**Table A.10.:** Configuration of each implementation for frequency of visits on laptop

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Special Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Single Threaded</td>
<td>none</td>
</tr>
<tr>
<td>Java Fork/Join</td>
<td>Fork/Join pool with 8 threads</td>
</tr>
<tr>
<td>Akka Actors</td>
<td>- poolsize = 8 (cores) * 3 (standardfactor) = 24</td>
</tr>
<tr>
<td></td>
<td>- 1 master actor</td>
</tr>
<tr>
<td></td>
<td>- 8 mapper</td>
</tr>
<tr>
<td></td>
<td>- 8 reducer</td>
</tr>
<tr>
<td>GPars MapReduce-DSL</td>
<td>pool with 8 threads used</td>
</tr>
</tbody>
</table>

**Table A.11.:** Configuration of each implementation for frequency of visits on laptop
A. Appendix

A.2. Results

**Important!** The results below sometimes lack values for big sizes of a problem. The reason for that is, either it took so much time that it was necessary to measure only less values or it required too much memory, thereby the limits of the given heap size were exceeded. Since other implementations have been already measured up to these sizes, the results have not been discarded afterwards.

![Figure A.1.](image)

**Figure A.1.:** 2000 Requests to compute PI while connections are kept alive. All implementations show very similar performance except Vertx with normal Verticles. An increase of concurrent connections causes a decrease of required time up to a certain saturation. Vert.x with standard verticles reaches the saturation first and Java second. The explanation for that is, Vert.x employs worker verticles which increases the overall throughput. When using standard verticles 16 verticles compete for 8 event loops. When a computing verticle is blocked in a long running computation the other 8 verticles are prevented from serving HTTP requests because there are more verticle instances than available event loops. Obviously, the additional effort of utilizing the actor system cannot not be recognized. For the last measured value for Java two possible explanations exist. Either it is caused by the overall imprecision or similar to Vert.x with workers, the additional thread pool serving the actors even increases the overall throughput.
Figure A.2.: 2000 Requests to compute PI while connections are closed after each request. As assumed, the overall time per request increases when connections are closed after each request. Generally, the behaviour as well as its explanation is the same as in A.1. The penultimate values of Java and Vert.x can either be ascribed to the underlying inaccuracy through the network, unforeseen events such as optimization or influence of other processes. Most probably, closing connections after each request is responsible for the variance.
Concurrent Connections

Transfer rate, received [Kbytes/sec]

Figure A.3.: 2000 Requests to compute PI while connections are kept alive. The results show the same behavior as the ones of A.1 where it is also explained. Pure Java, Akka and Vert.x with workers have nearly equal transfer rates whereas Vert.x with normal Verticles has a decrease for an increase of concurrent connections.
A.2. Results

Figure A.4.: 2000 Requests to compute PI while connections are closed after each request. This behaviour suits to the results of A.2 where it is also explained.
Figure A.5.: 2000 Requests to calculate PI while connections are kept alive. The Mean Total Processing Time grows proportionally to the amount of concurrent connections. This is because each additional socket causes further effort to handle it. The figure illustrates again that Vert.x with standard verticles scales worse than the other three implementations which show very similar performance.
Figure A.6.: 2000 Requests to calculate PI while connections are closed after each request. The results are similar to the ones of A.5. Nevertheless, connections are kept alive or not, the processing time of the server is independent from that. Despite that, the course of the values can be explained as in A.5
Concurrent Connections
Mean Time per Request [ms]

Akka
Java
Vertx

Figure A.7.: 100K requests to retrieve the number of request while connections are kept alive. All implementations show comparable performance. An increase of sockets causes a decrease of time per request. At some point, the saturation is reached, so that more concurrent connections start decreasing the required time again. Akka requires always more time than Java, but the absolute difference is rather small.
Figure A.8.: 20K requests to retrieve the number of request while connections are closed after each request. Like before, closing connections results in lower performance than keeping them alive. The course of Java contains an unexpected jump in the middle. With regard to figure A.7, and the course of Akka and Vert.x which also show unexpected knees for their last values we must conclude that the measurement with closing connections suffers from high variance.
Figure A.9.: 100K requests to retrieve the number of request while connections are kept alive. The courses of Akka, Vert.x and Java are very close to each other. Until the saturation is reached, more sockets result in a higher transfer rate. After that it causes a decrease of the transfer rate. Akka requires always more time than Java, but the absolute difference is quite small.
Figure A.10.: 20K requests to retrieve the number of request while connections are closed after each request. As in figure A.8, the results seem to underlie a very high variance when connections are closed. Thus, nothing more is inferred from them.
Figure A.11.: 100K requests to retrieve the number of request while connections are kept alive. It must be mentioned, the first two values are not shown because the resolution is too coarse. The apache benchmark tool rounded them to zero. All courses stick to the same behaviour which is nearly a straight line. More concurrent connections cause an increase of the total processing time. Since it is a logarithm scale a line indicates exponential growth. Java and Vert.x show nearly equal results whereas Akka requires always a bit more time.
Figure A.12.: 20K requests to retrieve the number of request while connections are closed after each request. The first value of Akka and Vert.x were set to 0.1 although the actual result was zero. Due to the resolution, measured points very close to zero could not be represented anymore. But zero is impossible in reality and also not shown on the logarithm scale. Therefore, a realistic value close to zero was determined to obtain consistent figures. Apart from this, a high variance can be recognized again.
Figure A.13.: Laptop: Transactions from each Ship one persone to each other. In A.2 is explained why some courses stop sooner. All courses have exponential growth since they conform roughly lines on a logarithm scale for big problem sizes. Java multithreaded is the fastest but converges to Java single threaded in the end. GPars STM keeps up well with pure Java. Then Akka with one sending actors follows. Four actors sending simultaneously improved the speed only for small amounts of ships. Afterwards it is even slower. GPars Actors is faster than Akka up to a certain point. Beyond, it requires more time. Next, Vert.x is way slower. As it requires also a lot of memory, there are no more measure points for big sizes of the problem. By far the slowest, is Akka STM.
### A.2. Results

#### Figure A.14.

Laptop: Transactions from each Ship one person to each other. All approaches based on message passing concurrency (except Akka with STM) require much more time than the others. For some implementations, increasing the problem to the next size caused an exceed of the given limit on the heap size. Therefore, some courses stop earlier than others. Generally, all courses look quite chaotic. This is because the absolute memory allocated was measured. Depending on the timing of the garbage collection, it can happen that new memory is allocated although a short time later it would not have been necessary because enough memory was freed through the collection. Furthermore, the amount of memory is measured directly when the problem signals the finish of execution. It is possible that allocated memory was returned to operating system in the meantime. Thus, smaller problem sizes can require less memory in adverse circumstances.
Figure A.15.: Laptop: Transactions from each Ship one person to each other. The ranking of each implementation stays the same as in A.14. We can see, the throughput increases for each implementation with an increase of the problem until it reaches saturation. From then on, Java single-threaded keeps constant like GPars STM. Multi-threaded Java has the highest throughput but then it converges to Java single-threaded. Vert.x and Akka STM reach saturation sooner. Way higher throughput than both have the other implementations which exploit message-passing but they are still slower than Multiverse and pure Java.
A.2. Results

Figure A.16.: Desktop Computer: Transactions from each Ship one person to each other. In A.2 is explained why some courses stop sooner. For Akka STM, an entire second test was run because some results of the required memory were very surprising. Obviously, Akka STM shows a huge variance for small problem sizes. The explanation is, it is non-deterministic whether collisions between transaction occur or not. Each collision requires additional time. Unsurprisingly, the desktop computer returns faster results. However, the ranking changed to the one in figure A.14. This time, GPars actors is faster than Akka and converges to Akka for high problem sizes. GPars STM is nearly as fast as Java multi-threaded. In contrast, Vert.x is again comparatively slow and converges to Akka STM which requires most time.
Figure A.17.: Desktop Computer: Transactions from each Ship one person to each other. As on the laptop, the measured memory suffers from high variance. That is proved by the second time Akka STM was measured. Although, the measurement lacks of validity it indicates that message passing concurrency on the JVM requires more memory than lock based approaches. The reason for that is, work is queuing in the form of messages if it is not processed fast enough.
Figure A.18.: Desktop Computer: Transactions from each Ship one person to each other. Likewise before, the throughput increases until it reaches saturation. Afterwards, Java multi-threaded, Java single-threaded, Multiverse and Akka STM keep their throughput constant. Unlike, the others have a decrease of the throughput. The ranking between each is the same as it was for the required time in figure A.16.
Figure A.19.: Laptop: Determine how often Places have been visited (20K Places per User).
Although the problem increases linear with the number of users all implementations show exponential courses. 
GPars is by far the slowest followed by Akka. From 30 to the knee at 600 users Java multi-threaded is the fastest. 
After the knee, it is slower than Java single-threaded. Since, the initial measurement accidentally used only two threads instead of four (available processors), it was measured another time. Surprisingly, the differences between using two and four threads are very small and the surprising knee occurs again. A conceivable reason to explain the knee would be unforeseen influences through other processes on the computer but this is rather unlikely since the two courses show very similar behaviour. Another possibility is, the access to the concurrent hashmap is so adverse, that multiple threads block each other frequently. In consequence, it even deteriorates the performance due to the overhead by handling multiple threads.
Figure A.20.: Laptop: Determine how often Places have been visited (20K Places per User). Both implementations with pure Java consume nearly an equal amount of memory. Akka requires a bit more and is followed by GPars which needs the most memory.
Figure A.21.: Laptop: Determine how often Places have been visited (20K Places per User). We can assess, that the saturation is reached at about 30 users for Java single-thread, Akka and GPars. Java Fork/Join has its maximum around 300 users before it nosedives.
A.2. Results

Figure A.22.: Desktop Compter: Determine how often Places have been visited (20K Places per User). Generally, the behaviour does not differ from the one measured on the laptop. The implementation with GPars is the slowest, then Akka follows. The difference between Akka and Java single-threaded got smaller than on the laptop. Java Fork/Join is the fastest implementation but like on the laptop an unexpected knee appeared. Two additional measurements should exclude that it was accidental. As stated in A.19 the knee could be caused through unforeseen influences such as other processes on the computer but it is very unlikely because all three measurements show the same result. Another possibility is, the access to the concurrent hashmap is so adverse, that multiple threads block each other frequently. In consequence, it even deteriorates the performance due to the overhead by handling multiple threads.
Figure A.23.: Desktop Computer: Determine how often Places have been visited (20K Places per User). The ranking of used memory is the same as on the laptop. GPars requires most, Akka second most and both Java solutions need similar amounts.
Figure A.24.: Desktop Computer: Determine how often Places have been visited (20K Places per User). GPars has by far the lowest throughput. Java single-thread has a bit higher throughput than Akka. The highest has Java Fork/Join but it also shows the unexpected knee we discussed in A.22.
A.2. Results
Bibliography


[34] SCHALLER, Nan C.; HILDERINK, Gerald H.; WELCH, Peter H.: Using Java for Parallel Computing: JCSP versus CTJ, a Comparison


Erklärung

Ich, Thomas Georg Kühner, Matrikelnummer 737643, erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den 25.10.2012

Thomas Georg Kühner