Technische Hochschule Rosenheim

Die nebenläufige und verteilte Verarbeitung am Beispiel des Aktorenmodells

Maximilian Bundscherer
Fakultät für Informatik

SoSe 2020

Die Taktraten von Prozessoren ist in den letzten Jahren nicht mehr signifikant gestiegen. Durch die Annäherung an physikalische Grenzen wachsen die Kosten bei der Entwicklung und Herstellung, bis zur Einstellung derselben. Da viele Anwendungen, zum Beispiel Wetter- oder Windkanalsimulationen in der Forschung, hohe Rechenleistungen benötigen, um in einer absehbaren Zeit zu einem Ergebnis zu gelangen, liegt es nahe, einen Teil der Berechnungen zu parallelisieren, d.h. ein Teil der Berechnungen läuft gleichzeitig über zum Beispiel mehrere Prozessor-Kerne oder Computer ab.

Viele Anwendungen laufen heutzutage in der "Cloud", d.h. die Anwendungen laufen in einem Rechenzentrum über mehrere Computer verteilt und meistens sehr stark von der eigentlichen Hardware abstrahiert. Das bedeutet für die Applikationen, dass diese nicht alle Vorteile der Hardware bzw. der zugrundeliegenden Architektur nutzen (können), im Gegensatz zu Anwendungen, die zum Beispiel in C++ oder Go systemnah bzw. hardwarenah implementiert worden sind.

Da die Parallelisierung von Berechnungen kein triviales Unterfangen darstellt und das vor allem die Softwareentwicklung betrifft, möchte diese Arbeit anhand des Aktorenmodells erläutern, wie man über nebenläufige Programmierung die Laufzeit einer Anwendung reduziert und wie man Anwendungen über die Verteilung auf mehrere Computer ausfallsicherer gestalten kann. Diese Arbeit geht aus den oben genannten Gründen nicht auf die Hardware(-Parallelisierung) ein und richtet sich daher eher an Softwareentwickler im Cloud-Umfeld als an Entwickler, die hardwarenah entwickeln.

Inhaltsverzeichnis

Abkürzungsverzeichnis

FIFO First In – First Out

JVM Java Virtual Machine

GC Garbage Collection

DSL Domain Specific Language

ES Event Sourcing

1 Einleitung mit Motivation

Die Taktraten von Prozessoren ist in den letzten Jahren nicht mehr signifikant gestiegen. Durch die Annäherung an physikalische Grenzen wachsen die Kosten bei der Entwicklung und Herstellung, bis zur Einstellung derselben [Ben15, VII].

Da viele Anwendungen, zum Beispiel Wetter- oder Windkanalsimulationen in der Forschung, hohe Rechenleistungen benötigen, um in einer absehbaren Zeit zu einem Ergebnis zu gelangen [Rau12, S.1] [Vog12, S.13], liegt es nahe, einen Teil der Berechnungen zu parallelisieren, d.h. ein Teil der Berechnungen läuft gleichzeitig über zum Beispiel mehrere Prozessor-Kerne oder Computer ab.

Folglich lässt sich der Trend erklären, dass viele Prozessor-Hersteller die heutigen Prozessoren mit mehreren Kernen ausstatten, um zusammen mit anderen Maßnahmen, wie zum Beispiel eine Verbesserung der Pipeline-Anordnung(en) oder eine Vergrößerung der internen Prozessor-Caches (L1, L2, L3), den nicht mehr signifikanten Anstieg der Taktraten zu kompensieren.

Viele Anwendungen laufen heutzutage in der "Cloud", d.h. die Anwendungen laufen in einem Rechenzentrum über mehrere Computer verteilt. Die Sicht auf die eingesetzte Hardware, wie Prozessoren, wird deutlich abstrakter, da viele Anbieter von Rechenzentren nur gegen Aufpreis den Einsatz von bestimmter Hardware garantieren und daher zum Beispiel nicht an den Kunden weitergeben, welche Prozessoren eingesetzt werden. Manche moderne Programmiersprachen wie Java oder Scala laufen außerdem selbst oft innerhalb einer virtuellen Maschine, im Beispiel Java in der Java Virtual Machine. In der Cloud ist es auch nicht unüblich, dass die Anwendungen in einem Container laufen, zum Beispiel Docker. Das bedeutet für die Applikationen, dass diese sehr viel abstrakter auf der Hardware laufen und daher nicht alle Vorteile der Hardware bzw. der zugrundeliegenden Architektur nutzen (können), im Gegensatz zu Anwendungen, die zum Beispiel in C++ oder Go systemnah bzw. hardwarenah implementiert worden sind.

Da die Parallelisierung von Berechnungen kein triviales Unterfangen darstellt und das vor allem die Softwareentwicklung betrifft [Uel19, S.1], möchte diese Arbeit anhand des Aktorenmodells erläutern, wie man über nebenläufige Programmierung die Laufzeit einer Anwendung reduziert und wie man Anwendungen über die Verteilung auf mehrere Computer ausfallsicherer gestalten kann. Diese Arbeit geht aus den oben genannten Gründen nicht auf die Hardware(-Parallelisierung) ein und richtet sich daher eher an Softwareentwickler im Cloud-Umfeld als an Entwickler, die hardwarenah entwickeln.

1.1 Einführende Begriffe

Die Begriffe aus diesem Umfeld müssen zunächst abgegrenzt und definiert werden, da es auf die folgenden Begriffe unterschiedliche Sichten gibt, die sich auch im Laufe der Zeit verändert haben und je nach Kontext unterschieden werden: So stellt sich beispielsweise ein Softwareentwickler im Cloud-Umfeld unter dem Begriff Verteilung die Verteilung einer Anwendung über mehrere Server oder virtuelle Maschinen vor; der Entwickler, der hardwarenah entwickelt, versteht darunter die Verteilung der Berechnungen über mehrere Prozessor-Kerne.

1.1.1 Nebenläufig, Parallel und Verteilt

Wenn viele Dinge gleichzeitig passieren, nennt man ein System **nebenläufig**. Dabei gibt es Vorgänge, die **echt parallel** ausgeführt werden können. Manche Vorgänge sehen aber nur parallel aus, werden aber in Wirklichkeit nur schnell hintereinander ausgeführt; dieses Verhalten bezeichnet man als **quasi-parallel**. Auf Software übertragen bedeutet das, dass die gleichzeitige Abarbeitung von Programmen und die Nutzung von Ressourcen nebenläufig ist, aber das Betriebssystem unter Berücksichtigung der verbauten Hardware, zum Beispiel Anzahl der Prozessoren bzw. Prozessor-Kerne, vorgibt, ob Teile dieser Abarbeitung auch echt parallel stattfinden [Ull16, 15.1].

Die **verteilte Verarbeitung**, auch Distributed Computing genannt, beschäftigt sich mit der Koordination von Computern, zum Beispiel innerhalb eines Netzwerks, die eine gemeinsame Aufgabe erledigen. Die eingesetzten Techniken bzw. Werkzeuge, wie Hardware oder Betriebssysteme der einzelnen Computer können dabei sehr stark variieren [Ben15, S.25], wie es bei angemieteten (ohne Aufpreis für spezielle Hardware) Cloud-Umgebungen häufig der Fall ist.

1.1.2 Programme, Threads, Prozesse und Scheduler

Viele moderne Betriebssysteme suggerieren dem Benutzer, dass verschiedene Anwendungen gleichzeitig ausgeführt werden:

- Bei Computern mit nur einem Prozessor bzw. Prozessor-Kern: Das Betriebssystem wechselt mit der Abarbeitung der Teile zum Beispiel alle paar Millisekunden. Die Ausführung ist hierbei nebenläufig, aber nicht echt parallel.
- Bei einem Computer mit mehreren Prozessoren bzw. Prozessor-Kernen: Die Programmteile können echt parallel ausgeführt werden.

Der Teil des Betriebssystem, der diese Umschaltung vornimmt, wird als **Scheduler**, auch Steuerprogramm genannt, bezeichnet [Ull16, 15.1.1]. Auch in Cloud-Umgebungen sind häufig ein oder mehrere Scheduler anzutreffen, da meistens verschiedene Berechnungen auf ein und derselben Maschine oder in einem Rechenzentrum stattfinden. Es gibt verschiedene Arten von Scheduler, die nicht nur auf einem Computer "umschalten", zum Beispiel der Linux-Scheduler *Completely Fair Scheduler*[1], sondern auch in Cloud-Umgebungen Berechnungen einplanen, zum Beispiel das sogenannte *Job Scheduling*[2] von der Firma Amazon Web Services (AWS). In dieser Arbeit wird nicht weiter auf dieses Thema eingegangen, da dies sonst den Rahmen sprengen und nicht weiter zum Verständnis beitragen würde.

Ein **Programm** besteht aus einem oder mehreren Prozessen. Ein **Prozess** setzt sich aus dem Programmcode und den Daten zusammen und besitzt einen eigenen Adressraum. Die Adressräume der einzelnen Prozesse werden durch die virtuelle Speicherverwaltung des Betriebssystems getrennt, wodurch es nicht möglich ist, dass ein Prozess in den Speicherraum eines anderen Prozesses eingreift, da das Betriebssystem das Programm in diesem Fall beenden würde [Ull16, 15.1.1].

1 siehe `https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html`
2 siehe `https://docs.aws.amazon.com/batch/latest/userguide/job_scheduling.html`

Bei modernen Betriebssystemen gehört zu jedem Prozess mindestens ein **Thread**, auch Ausführungsstrang genannt. Nach dieser Definition werden nicht mehr die Prozesse nebenläufig ausgeführt, sondern die Threads. Die Threads innerhalb eines Prozesses teilen sich den gleichen Adressraum und können untereinander auf ihre öffentlichen Daten zugreifen [Ull16, 15.1.2].

1.1.3 Race Conditions und Deadlocks

Eine **Race Condition**, auch kritische Wettlaufsituation genannt, bezeichnet eine Konstellation, bei der das Ergebnis einer Operation vom zeitlichen Ablauf bestimmter Einzeloperationen abhängt [Ben15, S.129] [Rau12, S.149]. Bei der nebenläufigen Programmierung sind diese Situationen von besonderer Relevanz, da schwer auffindbare, nichtdeterministische Fehler entstehen können. Erschwerend kommt meistens hinzu, dass durch (lokales) Debugging, zum Beispiel mithilfe eines Loggers und bei nur einem Prozessor, die Fehler nicht reproduzierbar und daher schwer aufzufinden sind. Im Abschnitt 2.4 wird dies an einem Beispiel verdeutlicht. Manche Entwickler sprechen in diesem Fall scherzhaft von einem sogenannten *Heisenbug*, was eine Zusammensetzung aus dem Namen des Physikers Werner Heisenberg und dem Bug ist.

Ein **Deadlock**, auch Verklemmung genannt, beschreibt einen Zustand, bei dem eine zyklische Wartesituation zwischen mindestens zwei Ausführungssträngen auftritt. Die Ausführungsstränge blockieren sich dabei selbst, d.h. die Anwendung friert beispielsweise ein. Grundsätzlich sollte bereits bei der Planung bzw. Implementierung einer Anwendung darauf geachtet werden, diesen Zustand zu vermeiden, da eine Auflösung nicht trivial und meistens auch gar nicht möglich ist und das Programm beendet werden muss. In Cloud-Umgebungen sind diese Situationen von besonderer Relevanz, da ein Deadlock auch zu einem totalen Ausfall führen kann und zur Folge hat, dass die Anwendung evtl. nicht mehr (von außen) erreichbar ist. Im Abschnitt 2.5 wird erläutert, wie man mit Deadlocks in Cloud-Umgebungen umgehen kann.

1.2 Leistungsmaße

Um die Reduktion der Laufzeit messbar machen zu können, können verschiedene Leistungsmaße herangezogen werden. Diese Arbeit beschränkt sich auf die *Laufzeit*, die *Overheadzeit* und auf den *Speedup*.

Hinweis: Diese Definitionen beziehen sich auf die Anzahl der Prozessoren bzw. Prozessor-Kerne, können aber im Cloud-Umfeld analog auf die Anzahl der Computer oder die Anzahl der Threads, die gemeinsam an einer Aufgabe arbeiten, angewendet werden.

1.2.1 Laufzeit und Overheadzeit

Um die Ausführung nebenläufiger Anwendungen bewerten zu können, kann die **Laufzeit** eines nebenläufiges Programms, das zum Teil auch parallel verarbeitet wird, wie folgt angegeben werden [Ben15, S.339]:

$$T_p(n)$$

Hierbei steht n für die Problemgröße. Das Problem wird auf p Prozessoren bzw. Prozessor-Kernen aufgeteilt. Die Laufzeit eines nebenläufigen und damit evtl. auch parallel verarbeitenden Programms setzt sich aus folgenden Punkten zusammen [Ben15, S.339]:

- **Rechenzeit** (T_{CPU}): Die Zeit für die Durchführung der eigentlichen Berechnung(en).

- **Kommunikationszeit** (T_{COM}): Die Zeit für den Austausch von Daten zwischen den Prozessoren bzw. Prozessor-Kernen.
- **Wartezeit** (T_{WAIT}): Die Zeit, die beispielsweise ein Prozessor bzw. ein Prozessor-Kern auf einen anderen wegen ungleicher Lastverteilung warten muss.
- **Synchronisationszeit** (T_{SYN}): Die Zeit für die Synchronisation aller beteiligten Prozessoren bzw. Prozessor-Kerne.
- **Platzierungszeit** (T_{Place}): Die Zeit für die Allokation.
- **Startzeit** (T_{Start}): Die Zeit, die für das Starten auf allen Prozessoren bzw. Prozessor-Kernen benötigt wird.

Zur Reduktion der Laufzeit muss die **Overheadzeit**, zusammengesetzt aus der *Kommunikationszeit*, der *Wartezeit* und/oder der *Synchronisationszeit*, reduziert werden. [Ben15, S.340]:

$$T_{CWS} = T_{COM} + T_{WAIT} + T_{SYN}$$

Hierbei zu beachten ist, dass die Overheadzeit zunimmt, desto mehr Prozessoren bzw. Prozessor-Kerne an einem System beteiligt sind. Das liegt zum Beispiel daran, dass die Lastverteilung komplexer wird, was sich unmittelbar auf die Wartezeit T_{WAIT} auswirkt. Auch die Synchronisationszeit T_{SYN} erhöht sich, da die Prozessoren bzw. Prozessor-Kerne normalerweise häufiger miteinander synchronisiert werden müssen, spätestens wenn die Berechnungen zusammenlaufen.

1.2.2 Speedup

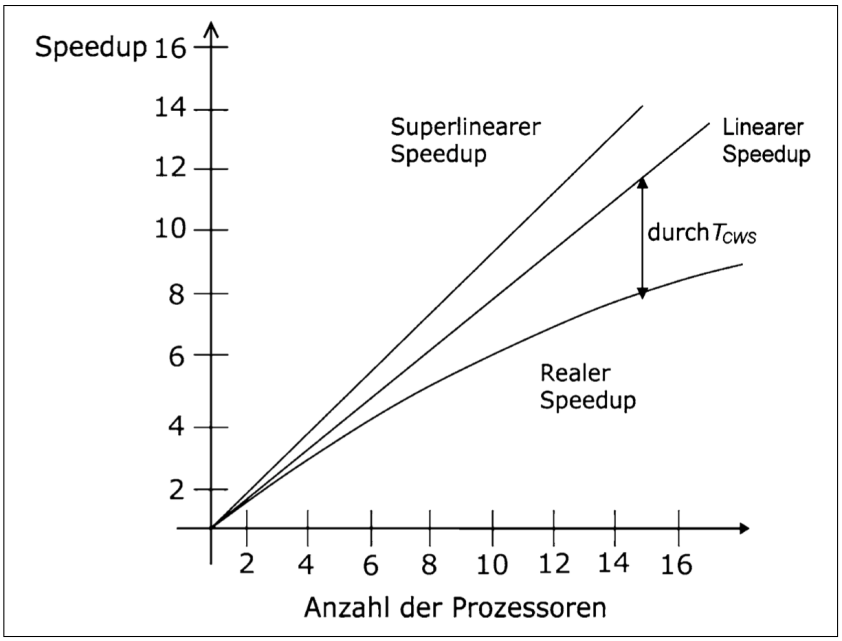

Abbildung 1: Abstrakte Darstellung von Arten des Speedups [Ben15, S.341]

Die Reduktion der Laufzeit für das Gesamtproblem einer nebenläufigen Anwendung, die zum Teil auch parallel ausgeführt wird, gibt der **Speedup**, auch Leistungssteigerung genannt,

an [Ben15, S.340]:

$$S_p(n) = \frac{T'(n)}{T_p(n)}$$

Hierbei steht n für die Problemgröße. Das Problem wird auf p Prozessoren bzw. Prozessor-Kerne aufgeteilt. $T'(n)$ steht für die Laufzeit der schnellsten bekannten sequentiellen Ausführung, d.h. nicht parallelen Ausführung und $T_p(n)$ für die Laufzeit des nebenläufigen und damit evtl. auch parallelen Programms. Dabei gilt $T_1(n) \neq T'(n)$, da die nebenläufige und parallele Ausführung immer mit einer Overheadzeit T_{CWS} verbunden ist.

Der Speedup ist normalerweise nach oben beschränkt, durch die Anzahl der Prozessoren bzw. Prozessor-Kerne [Ben15, S.340]:

$$S_p(n) \leq p$$

Vereinfacht gesagt bedeutet das, dass die Geschwindigkeitssteigerung von der Anzahl der Prozessoren bzw. Prozessor-Kerne abhängt und beispielsweise zwei Prozessoren bzw. Prozessor-Kerne nicht den Geschwindigkeitsvorteil von dreien bieten können.

Die Abbildung 1 stellt exemplarisch verschiedene Speedup-Arten dar: Ist $S = p$, dann spricht man von einem **linearen Speedup**. Dieser Fall stellt einen Idealfall dar und tritt in der Praxis durch den Overhead T_{CWS} nicht auf. Durch den Overhead T_{CWS} ergibt sich der **reale Speedup**. Ist $S > p$, dann spricht man von **superlinearem Speedup** [Ben15, S.341]. Diese Arbeit geht nicht weiter auf den superlinearen Speedup ein, da dieser in der Praxis selten vertreten ist und nicht zum Verständnis beitragen würde. Mehr zu diesem Thema findet man aber unter [Ris16].

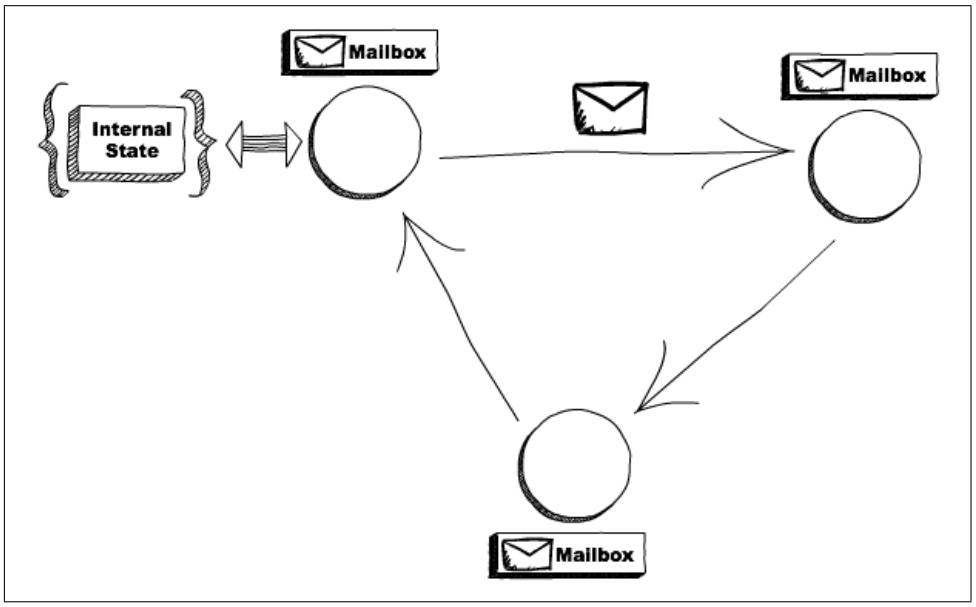

Abbildung 2: Abstrakte Darstellung von drei Aktoren in einem Aktorensystem [Imga]

2 Nebenläufige und verteilte Verarbeitung am Beispiel des Aktorenmodells

Wie bereits in der Einleitung erläutert, möchte diese Arbeit anhand des Aktorenmodells erläutern, wie man über nebenläufige Programmierung die Laufzeit einer Anwendung reduziert und wie man Anwendungen über die Verteilung auf mehrere Computer ausfallsicherer gestalten kann.

Im Abschnitt 1.1.1 und 1.1.2 wurde erläutert, dass es einen Unterschied zwischen nebenläufig und parallel gibt: Der Softwareentwickler kann eine Anwendung nebenläufig implementieren, die Parallelisierung erfolgt hierbei durch das Betriebssystem in Anbetracht der eingesetzten Hardware.

Grundsätzlich lässt sich sagen, dass nicht alle Teile einer nebenläufigen Anwendung parallelisiert werden können, da zum Beispiel manche Berechnungen voneinander abhängen. Als Beispiel wird hierfür die Berechnung der Fibonacci-Folge[3] angeführt: Die Werte aus der Fibonacci-Folge werden aus ihren Vorgängern berechnet, d.h. die vorherigen Werten müssen erst alle berechnet werden, um die nächsten Werte zu bestimmen.

2.1 Das Aktorenmodell

Das **Aktorenmodell** ist ein Modell aus der Informatik für die nebenläufige Programmierung. Das Programm wird dabei in Aktoren unterteilt. Diese Aktoren werden in einem Aktorensystem verwaltet. Aktoren kommunizieren ausschließlich über unveränderbare Nachrichten. Der Zustand eines Aktors ist von außen nicht direkt sichtbar und kann auch nur über Nachrichten abgefragt und modifiziert werden [Hew73, S.235]. Das Modell wurde 1973 das erste Mal von Carl Hewitt, Peter Bishop und Richard Steiger beschrieben [Hew73] und ist bei funktionalen

3 siehe `http://www.mathematik.uni-muenchen.de/~forster/v/zth/inzth_01.pdf`

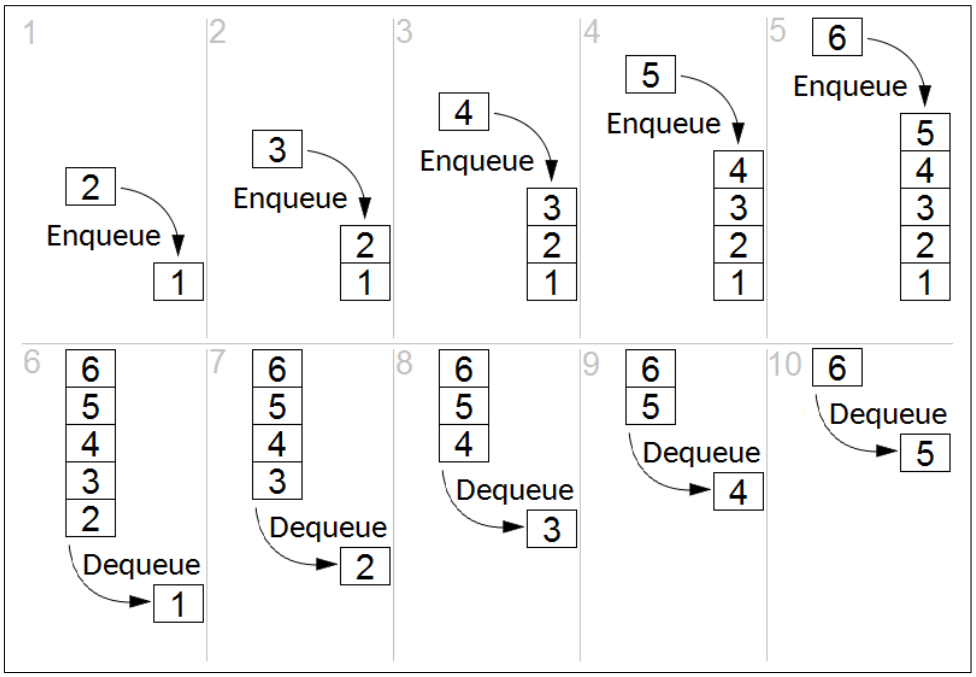

Abbildung 3: Abstrakte Darstellung des FIFO-Prinzips [Imgb]

Programmiersprachen wie zum Beispiel Scala oder Erlang stark verbreitet. Die Abbildung 2 zeigt eine exemplarische Darstellung von drei Aktoren in einem Aktorensystem.

Beschreibung eines Aktors

Ein **Aktor** ist eine kleine Verarbeitungseinheit in einem System, dessen Zustand von außen nicht direkt einsehbar oder veränderbar ist. Um mit einem Aktor interagieren zu können, um zum Beispiel dessen Zustand einzusehen oder zu verändern, wird ausschließlich in Form von unveränderbaren Nachrichten mit diesem kommuniziert [Hew73, S.235]. Ein Aktor kann Nachrichten empfangen und selbst versenden. Eingehende Nachrichten werden zunächst in dem Postfach des jeweiligen Aktors hinterlegt.

Der Aktor arbeitet sequentiell die eingegangenen Nachrichten aus seinem Postfach ab. Das Postfach verwaltet die Nachrichten in Form einer Warteschlange [Roe16, S.107]. Daher arbeitet ein Aktor nach dem First In – First Out (FIFO)-Prinzip [Hew73, S.236]. Bei dem FIFO-Prinzip werden Nachrichten in der Reihenfolge abgearbeitet, in der diese eingegangen sind. Die Abbildung 3 visualisiert dieses Prinzip.

2.2 Scala

Scala ist eine funktionale und objektorientierte Programmiersprache für die Java Virtual Machine (JVM)[4] [Sca20a]. Seit 2001 wird Scala an der École Polytechnique Fédérale de Lausanne (EPFL) vom einem Team unter der Leitung von Martin Odersky entwickelt [Pie10, S.30].

4 siehe https://docs.oracle.com/javase/6/docs/technotes/guides/vm/index.html?intcmp=3170

In Scala lässt sich im Vergleich zu Java mit weniger Code deutlich mehr ausdrücken. Zum einen benötigt Scala deutlich weniger Schlüsselwörter, zum anderen war das Designziel von Scala, eine knappe, elegante und typsichere Programmierung zu ermöglichen [Pie10, S.30] [Bra10, S.2]. Als Beispiel wird hierfür das **Pattern Matching** in Scala herangezogen. Das Pattern Matching in Scala stellt eine Verallgemeinerung der aus C-ähnlichen Programmiersprachen bekannten `switch-case` Anweisung dar [Bra10, S.114], ein Beispiel:

```
val t = "Ich bin ein String"

t match {

  case r: String => println("String: " + r)
  case r: Int => println("Int: " + r)

}
```

Mit der `match` Anweisung lassen sich hierbei nicht nur Typen prüfen, sondern auch der Wert einer Variablen.

Im Gegensatz zu vielen modernen Programmiersprachen ist Scala statisch typisiert. Das bedeutet, der Typ aller Ausdrücke wird zur Kompilierzeit überprüft und nicht erst zur Laufzeit, wie bei dynamisch typisierten Sprachen. Das Typsystem von Scala ist sehr ausgreift und lässt neben generischen Klassen und polymorphen Methoden auch Varianz-Annotationen, Upper und Lower Bounds zu. Ein weiteres relevantes Merkmal von Scala ist die einfache Erweiterbarkeit. Damit ist Scala für die Erstellung von Domain Specific Language (DSL)s gut geeignet [Bra10, S.2].

Es ist möglich, Java-Komponenten innerhalb von Scala zu benutzen, da Scala-Bytecode mit Java-Bytecode kompatibel ist. Als Bytecode wird eine Sammlung von Befehlen für eine virtuelle Maschine bezeichnet. Programmiersprachen wie Java und Scala werden nicht zu einem direkten Maschinencode kompiliert, sondern zu einem Zwischencode, dem Bytecode[5]. Java- und Scala-Bytecodes sind innerhalb der JVM lauffähig und miteinander kompatibel. Dadurch können Scala-Komponenten von Java-Komponenten benutzt werden und andersrum [Sca20b].

Hinweis: Diese Kompatibilität ist in der Praxis häufig mit Problemen verbunden. So kann Beispielsweise in Scala ein `Object`, eine Singleton-Abstraktion[6], im Package-Pfad direkt mit dem Operator `$` angesprochen werden, zum Beispiel `de.maxbundscherer.example.Main$`. Das ist innerhalb von Java nicht möglich, d.h. das würde bei einem Aufruf innerhalb von Java zu Problemen führen.

2.2.1 Nebenläufigkeit und Seiteneffekte

Nachdem Scala und Java zueinander kompatibel und beide Programmiersprachen in der JVM lauffähig sind, können innerhalb von Scala die Java-Techniken, zum Beispiel Java-Threads und Java-Threadpools, verwendet werden. In dieser Arbeit werden Scala Threads und Java

5 siehe https://www.mi.fu-berlin.de/inf/groups/ag-pr/Lehrveranstaltungen/swpue-2013/Bytecode_2013-04-18.pdf

6 siehe https://docs.scala-lang.org/tour/singleton-objects.html

Threads synonym verwendet, da diese sich technisch nicht unterscheiden[7].

In Scala bietet sich aber eine sehr elegante Abstraktionsmöglichkeit für die nebenläufige Programmierung: das Aktorenmodell, vergleiche Abschnitt 2.1 [Bra10, S.193]. Das Aktorenmodell wurde in Scala als *Akka Actors* implementiert und wird im folgenden Abschnitt 2.2.2 genauer erläutert.

Da Scala auch eine funktionale Sprache ist, versucht man, bereits über das Sprachdesign Seiteneffekte zu reduzieren[8]. Ein **Seiteneffekt** liegt dann vor, wenn eine Funktion beispielsweise den Wert einer globalen Variablen ändert. Dies kann insbesondere bei nebenläufiger Ausführung zu Race Condtions führen, vergleiche Abschnitt 1.1.3. So sind in Scala beispielsweise die Collections, wie z.B. `List` oder `Map` immutable (unveränderlich).

2.2.2 Akka Actors

Akka ist ein Open-Source Toolkit für die Erstellung von parallelisierten, verteilten, ausfallsicheren und nachrichtengesteuerten Anwendungen in Scala und Java [Akk20]. Akka implementiert mit **Akka Actors** das Aktorenmodell, vergleiche Abschnitt 2.1. Akka ist für den Einsatz innerhalb der JVM konzipiert und implementiert.

Synchronisieren von Aktoren

Nach dem Aktorenmodell besitzen die Aktoren jeweils ihren eigenen Zustand, der von außen nicht direkt einsehbar oder veränderbar ist. Um mit einem Aktor interagieren zu können, um zum Beispiel dessen Zustand einzusehen oder zu verändern, wird ausschließlich in Form von unveränderbaren Nachrichten mit diesem kommuniziert, vergleiche Abschnitt 2.1.

In Scala stellt jeder Thread auch einen Akka Actor dar. Die Umkehrung gilt nicht, da nicht jeder Akka Actor einen eigenen Thread benötigt [Bra10, S.194]. Das heißt die Aktoren müssen gegebenenfalls über Nachrichten miteinander synchronisiert werden. Hierfür gibt es beispielsweise zwei Möglichkeiten:

- **Fire and Forget**[9]: Ein Aktor A_a sendet eine Nachricht an einen anderen Aktor A_b und wartet nicht auf die Antwort von A_b. Der Aktor A_a **blockiert nicht**, während Aktor A_b die Nachricht verarbeitet. Dies wird mit dem Aufruf von `aktorB ! message` innerhalb des Aktors A_a realisiert.
- **Ask**[10]: Ein Aktor A_a sendet eine Nachricht an einen anderen Aktor A_b und wartet auf die Antwort von A_b. Der Aktor A_a **blockiert**, während Aktor A_b die Nachricht verarbeitet. Dies wird mit dem Aufruf von `aktorB ? message` innerhalb des Aktors A_a realisiert. Dieser Ausdruck liefert ein `Future` zurück, das einen Platzhalter für ein noch nicht existierendes Objekt darstellt. Die Zeit, die der Aktor A_a auf den Aktor A_b wartet, muss zum

7 siehe https://alvinalexander.com/scala/differences-java-thread-vs-scala-future/
8 siehe https://www.informatik-aktuell.de/entwicklung/programmiersprachen/funktionale-programmierung-mit-scala-herangehensweisen-und-konzepte.html
9 siehe https://doc.akka.io/docs/akka/current/typed/interaction-patterns.html#fire-and-forget
10 siehe https://doc.akka.io/docs/akka/current/typed/interaction-patterns.html#request-response-with-ask-between-two-actors

Beispiel über den Aufruf von `implicit val timeout = Timeout(5 seconds)` vorher definiert werden.

Moderne Betriebssysteme unterstützen Threads direkt, so bildet die JVM die Thread-Verwaltung in der Regel auf das Betriebssystem ab. Ob die Laufzeitumgebung native Threads nutzt, steht in der Spezifikation der jeweiligen JVM. Die 1-zu-1 Abbildung ermöglicht eine einfache Verteilung auf mehrere Prozessoren bzw. Prozessor-Kerne [Ull16, 15.1.2].

Ein Beispiel

An der folgenden exemplarischen Implementierung[11] wird verdeutlicht, wie man mit einem Akka Aktor arbeitet. Der `TestActor` in dieser Implementierung besitzt einen Zustand in Form eines ganzzahligen Werts (`Int`).

Zunächst wird der Zustand `State(...)` des Aktors und
die Nachricht `IncreaseBalance(...)`, die vom Interface (in Scala als `Trait` bezeichnet) `Request` erbt, deklariert:

```
//Declare request wrapper and internal state
sealed trait Request
private case class State(accountBalance: Int)

//Declare concrete request
case class IncreaseBalance(amount: Int) extends Request
```

Anschließend wird der Zustand des Aktors bei der Initialisierung definiert (`accountBalance = 0`) und das Verhalten auf eingehende Nachrichten vom Typ `IncreaseBalance(...)` definiert: Der Aktor wird bei eingehenden Nachrichten von diesem Typ den Wert seines Zustands um den in der Nachricht definierten Wert inkrementieren und anschließend per Logger den Wert seines Zustand, vor und nach der Veränderung, ausgeben:

```
//Default state is idle (define internal state)
def apply(): Behavior[Request] = applyIdle(State(accountBalance = 0))

//Process messages in state idle
private def applyIdle(state: State): Behavior[Request] =
    Behaviors.receive { (context, message) =>

  message match {

    case cmd: IncreaseBalance =>

      val newState: State = state.copy(accountBalance =
          state.accountBalance + cmd.amount)

      context.log.info(s"Old balance was (${state.accountBalance}) / New
          balance is (${newState.accountBalance})")

      applyIdle(newState)
```

11 siehe https://github.com/maxbundscherer/scala-akka-prime-speedup/blob/master/src/main/scala/de/maxbundscherer/akka/scala/prim/examples/MainExample.scala

```
  }

}
```

Abschließend wird noch das Aktorensystem mit dem `TestActor` gestartet und es werden dem Aktor zwei Nachrichten vom Typ `IncreaseBalance(...)` zugestellt:

```
//Init actor system and actor
private val actorSystem: ActorSystem[TestActor.Request] =
   ActorSystem(TestActor(), "actorSystem")

//Fire and forget two requests
actorSystem ! TestActor.IncreaseBalance(amount = 10)
actorSystem ! TestActor.IncreaseBalance(amount = -5)
```

Dies führt zu folgender Konsolenausgabe:

```
Old balance was (0) / New balance is (10)
Old balance was (10) / New balance is (5)
```

Hinweis: Da die Ausführungsreihenfolge nichtdeterministisch ist, ist es möglich, dass die Nachrichten in unterschiedlichen Reihenfolgen zugestellt werden, vergleiche Abschnitt 1.1.3.

Kompatibilität mit Computercluster(n)

Ein **Computercluster**, auch Rechnerverbund genannt, bezeichnet eine Anzahl von vernetzten Computern. Die Computer werden hierbei häufig über das TCP/IP-Protokoll miteinander verbunden. Computercluster sind häufig im Cloud-Umfeld anzutreffen, siehe Erläuterung zur *verteilten Verarbeitung* im Abschnitt 1.1.1.

Akka unterstützt sowohl den Betrieb auf einem Computer als auch den Betrieb auf Computerclustern:

- Bei Betrieb auf einem Computer: Es ist keine zusätzliche Konfiguration oder Implementierung notwendig, da dies der Standardkonfiguration entspricht.
- Bei Betrieb auf einem Computercluster: Es ist es zum Beispiel notwendig, die Nachrichten über das TCP/IP-Protokoll zu transportieren.

Akka bietet mit **Akka Remote**[12] eine Abstraktion für den Entwickler, womit Aktoren über das TCP/IP-Protokoll miteinander kommunizieren können. *Akka Remote* sollte aber innerhalb eines Produktivsystems nicht ohne **Akka Cluster**[13] verwendet werden, da diese Erweiterung *Akka Remote* beinhaltet und noch zusätzliche Funktionen wie *Service Discovery*[14]

12 siehe `https://doc.akka.io/docs/akka/current/remoting.html`
13 siehe `https://doc.akka.io/docs/akka/current/typed/cluster.html`
14 siehe `https://doc.akka.io/docs/akka/current/discovery/index.html`

oder automatische *Health Checks*[15] mit sich bringt. Auf *Akka Remote* und *Akka Cluster* wird im Rahmen dieser Arbeit nicht mehr eingegangen, da dies sonst den Rahmen sprengen würde.

Hinweis: Standardmäßig verwendet Akka die Java-Standardserialisierung (Java Object Serialization), um Nachrichten serialisieren bzw. wieder deserialisieren zu können. Diese wird aber nicht empfohlen, da sie zwischen unterschiedlichen Java-Versionen und Systemen nicht binärkompatibel ist[16].

2.3 Filtern von Primzahlen als Beispielimplementierung

Um den Speedup, vergleiche Abschnitt 1.2.2, greifbar machen zu können, wurde eine Anwendung in Scala nach dem Aktorenmodell implementiert basierend auf *Akka Actors*, die Primzahlen aus einem vorher definierten Bereich filtert.

2.3.1 Aufbau der Anwendung

Die Beispielimplementierung[17] filtert Primzahlen aus einem vorher definiertem Bereich, misst dabei die benötigte Zeit und dokumentiert diese in einer CSV-Datei. Die Anwendung kann die Filterung auf mehrere Threads verteilen, damit auch über mehrere Prozessoren bzw. Prozessor-Kerne, beachte Hinweis aus Abschnitt 1.2. Um gegen Messabweichungen zum Beispiel wegen Garbage Collection (GC)-Zeiten, IO-Zugriffszeiten oder Cache-Stufen vorzugehen, können die Läufe mehrfach automatisch wiederholt werden. Die Abbildung 4 stellt groben den Ablauf der Anwendung als Sequenzdiagramm dar.

Die Parameter der Anwendung[18] lassen sich wie folgt beschreiben:

- **maxWorkersPerRun**: Gibt an, auf wie viele Threads die Filterung aufgeteilt werden soll und wird im Folgenden als Verteilungsstufe bezeichnet. Beispielsweise gibt der Wert $1, 2$ hier an, dass die Filterung erst auf einem Thread stattfindet und anschließend auf zwei Threads verteilt wird.
- **repeatRun**: Gibt an, wie oft die Läufe pro Verteilungsstufe (*maxWorkers* aus *maxWorkersPerRun*) wiederholt werden sollen. Der Wert dieses Parameters sollte bei starken Messabweichungen erhöht werden. Beispielsweise gibt der Wert 5 an, dass die Messung fünfmal pro Verteilungsstufe wiederholt werden soll.
- **to**: Gibt an, bis zu welchem Wert die Primzahlen gefiltert werden sollen. Dieser wird, falls nicht anders angegeben, automatisch aus den Verteilungsstufen (*maxWorkersPerRun*) berechnet: Da die Aufteilung der Last gleichmäßig sein soll, stellt dieser Wert das kleinste gemeinsame Vielfache der Verteilungsstufen dar. Beispielsweise wird der Wert aus den Verteilungsstufen 1 und 2 berechnet: Das kleinste gemeinsame Vielfache ist hierbei 2.

15 siehe `https://doc.akka.io/docs/akka-management/current/healthchecks.html`
16 siehe `https://doc.akka.io/docs/akka/2.5.31/serialization.html`
17 siehe `https://github.com/maxbundscherer/scala-akka-prime-speedup`
18 siehe `https://github.com/maxbundscherer/scala-akka-prime-speedup/blob/master/src/main/scala/de/maxbundscherer/akka/scala/prim/Main.scala`

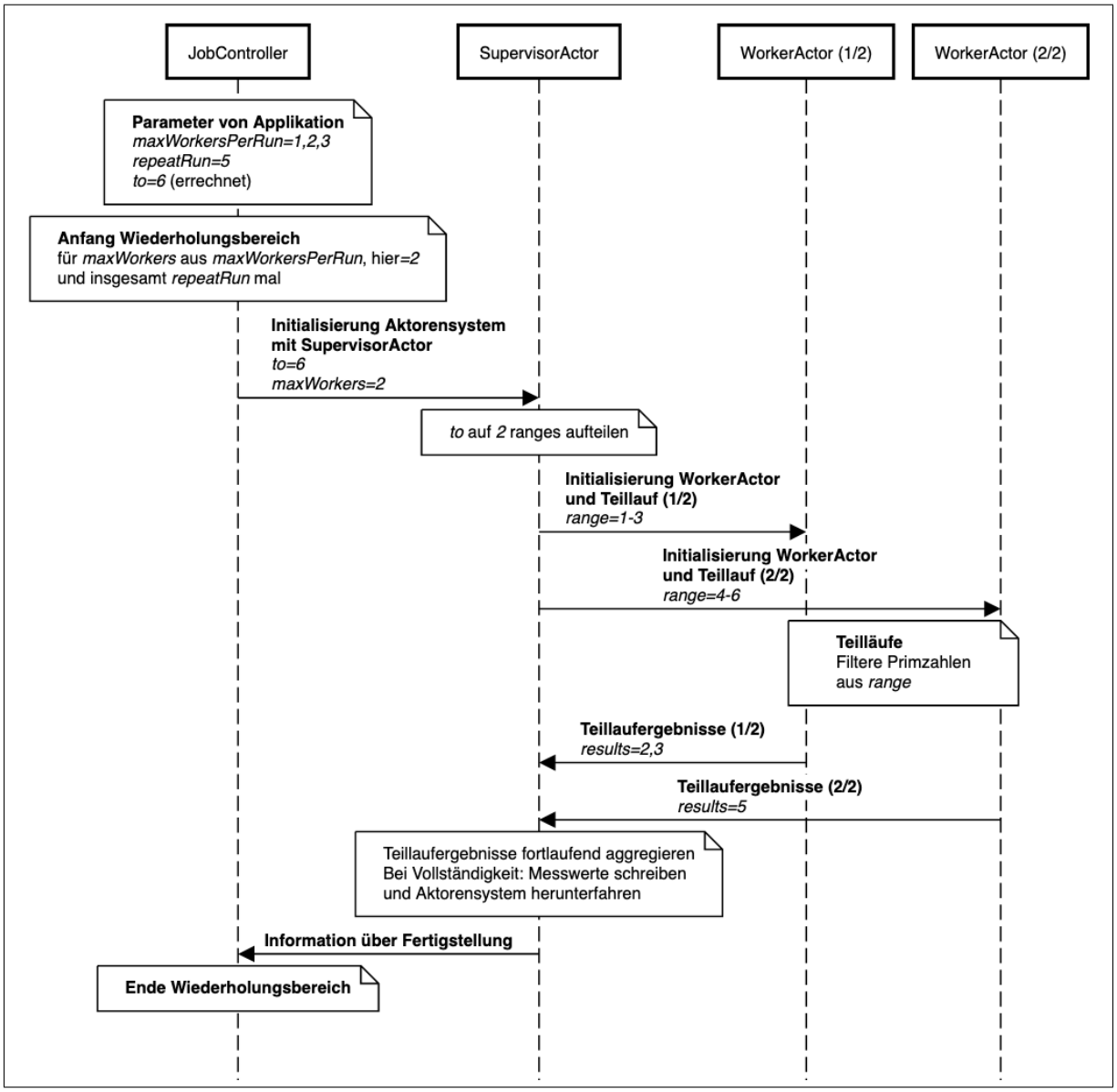

Abbildung 4: Sequenzdiagramm Filtern von Primzahlen als Beispielimplementierung

Hinweise zur Filterung und Verteilung

Die Zahlen werden in der Beispielanwendung mithilfe dieser sehr trivialen Implementierung[19] gefiltert:

```
def isPrime(n: Int): Boolean = ! ((2 until n-1) exists (n % _ == 0))
```

Im Rahmen dieser Arbeit ist dies ausreichend, da nicht die effiziente Filterung von Primzahlen im Vordergrund steht und alle Läufe diese Methodik[20] verwenden.

Die Verteilung[21] erfolgt ziemlich rudimentär; das bedeutet beispielsweise bei einem Zielbereich von $0 - 100$ und zwei Aktoren, dass der Zielbereich auf $0 - 50$ und auf $51 - 100$ "gleichmäßig" aufgeteilt wird. Diese Strategie führt aber dazu, dass die Bereiche mit niedrigeren Werten schneller abgearbeitet werden können als die Bereiche mit höheren Werten. Im Rahmen dieser Arbeit ist dies ausreichend, dem Leser sollte aber bewusst werden, dass diese Verteilungsstrategie Auswirkungen auf die Laufzeit hat, da sich nach der Abarbeitung eines Zielbereichs der Aktor herunterfährt und wieder Ressourcen auf der Maschine frei werden.

2.3.2 Referenzsystem und die gewählten Parameter

Die Messungen wurden auf einem eigenem Server durchgeführt. Das Referenzsystem weist folgende Eigenschaften auf:

- **Anbieter**: Amazon Web Services (AWS)
- **Betriebssystem**: Ubuntu Server 20.04 LTS (HVM) (ami-0b90a8636b6f955c1)
- **Festplatte**: SSD mit 8 GB
- **Prozessor**: 4vCPU
- **Arbeitsspeicher**: 16 GB

Um die Installation auf dem genannten Referenzsystem zu vereinfachen und zu automatisieren, wurde ein Skript erstellt, dieses lässt sich dem Abschnitt A.3.1 entnehmen. Um die Läufe auf dem genannten Referenzsystem zu vereinfachen und zu automatisieren, wurde ein Skript erstellt, dieses lässt sich dem Abschnitt A.3.2 entnehmen.

Die Parameter wurden wie folgt gewählt:

- **repeatRun**: 5
- **maxWorkersPerRun**: $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16$
- **to** (errechnet): 720720

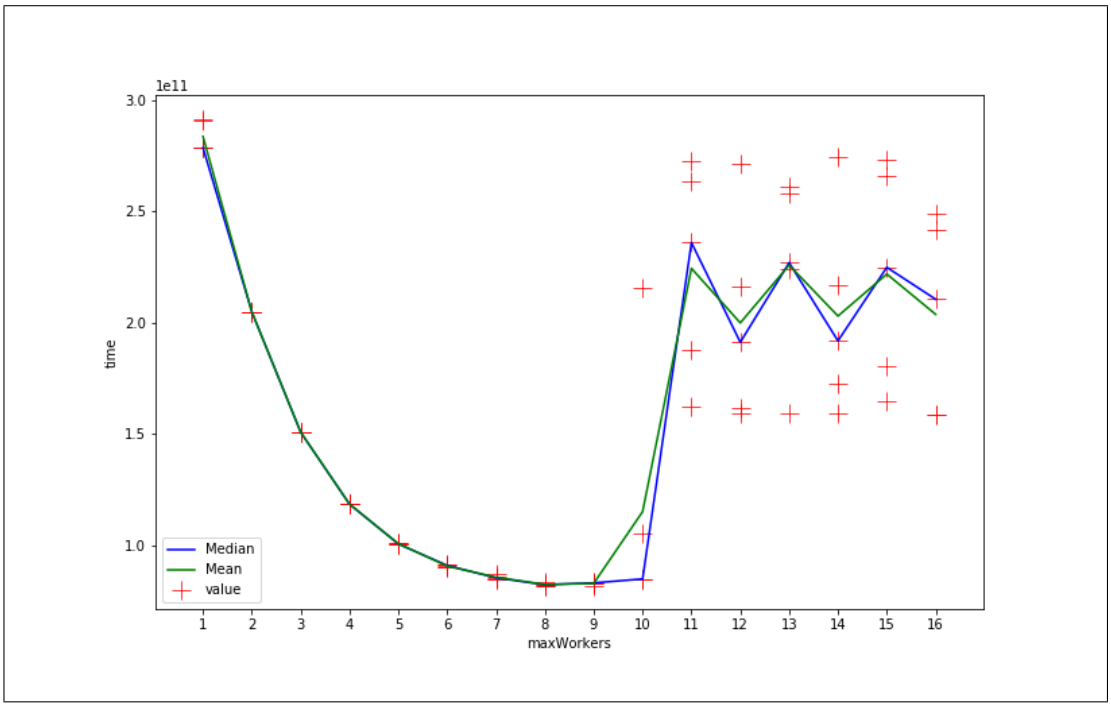

Abbildung 5: Darstellung der Laufzeit anhand einer festen Problemgröße und variierenden Verteilungsstufen (maxWorkers) der Beispielimplementierung. Die einzelnen Messungen werden zusätzlich über die Durchschnitte (Mean) und die Mittelwerte (Median) pro Verteilungsstufe angegeben.

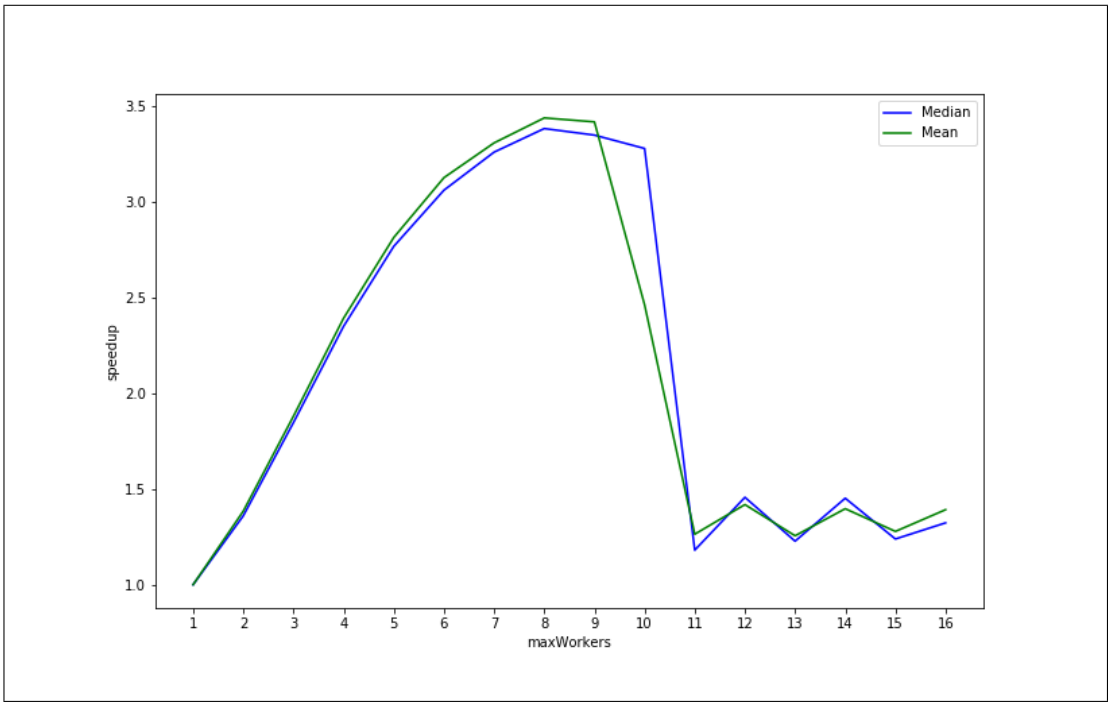

Abbildung 6: Darstellung des Speedups anhand einer festen Problemgröße und variierenden Verteilungsstufen (maxWorkers) der Beispielimplementierung. Die einzelnen Messungen werden über die Durchschnitte (Mean) und die Mittelwerte (Median) pro Verteilungsstufe angegeben.

2.3.3 Auswertung von Laufzeit und Speedup

Zur Auswertung der Läufe wurde ein *Jupyter Notebook*[22] entwickelt[23]. Die Grafiken wurden mithilfe der Python-Bibliothek *Matplotlib*[24] erstellt. Die Ergebnisse der Läufe lassen sich im Abschnitt A.1 einsehen. Das Log der Messung lässt sich im Abschnitt A.2 einsehen.

Die Daten wurden zunächst eingelesen und ausgewertet: Es wurden insgesamt 80 Messungen durchgeführt, wobei pro Verteilungsstufe $1 - 16$ die Messungen 5 mal wiederholt worden sind. Damit ergibt sich ein berechneter Zielbereich von $0 - 720720$, vergleiche *to* aus Abschnitt 2.3.1. In diesem Bereich wurden 58084 Zahlen als Primzahlen klassifiziert, alle Messungen stimmen mit diesem Ergebnis überein. Dem Log lässt sich entnehmen, dass der erste Lauf um 08:21:19 Uhr begonnen hat und der letzte Lauf um 11:57:26 Uhr endete, der gesamte Vorgang dauerte folglich insgesamt ungefähr 3 Stunden und 36 Minuten.

Hinweis: Wie bereits im Abschnitt 1.2 beschrieben, wird hier die Anzahl der Prozessoren bzw. der Prozessor-Kerne analog auf die Anzahl der Threads, die gemeinsam an einer Aufgabe arbeiten, angewendet.

Auswertung der Laufzeit

Anschließend wurde die Laufzeit $T_p(n)$, vergleiche Abschnitt 1.2.1, ausgewertet. Da die Problemgröße n, hier durch *to* definiert, innerhalb der Läufe konstant ist, wurde die Anzahl der Threads variiert, hier durch *maxWorkers* (Verteilungsstufe). Da die einzelnen Messungen pro Verteilungsstufe variieren, werden diese über den Mittelwert und dem Durchschnitt angegeben. Die Abbildung 5 stellt die Laufzeit in Relation zu der Anzahl der verwendeten Threads dar.

Den Messungen und der Grafik lässt sich entnehmen, dass die Verteilung über mehrere Threads, damit auch die Verteilung auf mehrere Prozessoren bzw. Prozessor-Kerne, einen Einfluss auf die Laufzeit hat. In diesem Fall wurde die Filterung von Primzahlen aus einem vorher vorgegeben festen Bereich auf mehrere Threads verteilt. Die Geschwindigkeitssteigerung pro zusätzlichem Thread nimmt mit der Anzahl der Threads deutlich ab. Das ist aber nicht weiter verwunderlich, da die Overheadzeit auch mit Anzahl der beteiligten Threads zunimmt und die Laufzeit sich mitunter aus der Overheadzeit zusammensetzt, vergleiche Abschnitt 1.2.1.

Den Messungen und der Grafik lässt sich auch entnehmen, dass ab der Verteilungsstufe 8 die Laufzeit wieder zunimmt, d.h. der Server konnte die Berechnungen nicht mehr effizient und sinnvoll über die (virtuellen) Prozessoren bzw. Prozessor-Kerne verteilen. Die Gründe dafür lassen sich nur schwer ermitteln, zum Beispiel mit dem Einsatz von Software-Profilern,

19 siehe `https://github.com/maxbundscherer/scala-akka-prime-speedup/blob/master/src/main/scala/de/maxbundscherer/akka/scala/prim/utils/Calculator.scala`
20 siehe `https://github.com/maxbundscherer/scala-akka-prime-speedup/blob/master/src/main/scala/de/maxbundscherer/akka/scala/prim/actors/WorkerActor.scala`
21 siehe `https://github.com/maxbundscherer/scala-akka-prime-speedup/blob/master/src/main/scala/de/maxbundscherer/akka/scala/prim/actors/SupervisorActor.scala`
22 siehe `https://jupyter.org/`
23 siehe `https://github.com/maxbundscherer/scala-akka-prime-speedup/blob/master/reports/report.ipynb`
24 siehe `https://matplotlib.org/`

im Fall von Scala kann *Java VisualVM*[25] herangezogen werden. Auch das Monitoring von Betriebssystem und Hardware kann bei der Aufklärung unterstützen. Im Rahmen dieser Arbeit wurden die genauen Gründe hierfür nicht ermittelt, da diese sich wahrscheinlich nicht mit einem Mehrwert für den Leser auf andere Betriebssysteme und Hardware übertragen lassen. Vermutet wird aber, dass der Server sich im Grenzbereich befindet bzw. künstlich durch den Cloud-Anbieter limitiert wird, wodurch zum Beispiel die Befehle in einer (virtuellen) Prozessoren-Pipeline häufiger verworfen werden (pipeline flush) und der Server häufiger Daten aus dem Arbeitsspeicher auf die Festplatte auslagern muss (swapping). Diese Effekte werden zusätzlich durch den Einsatz von Scala innerhalb der JVM verstärkt, da durch die Abstraktion die eigentliche Hardware nicht effizient genutzt werden kann. Zu beachten gilt auch, dass die Verteilung der Zielbereiche durch die Beispielimplementierung erfolgt, vergleiche Abschnitt 2.3.1. Beim Einsatz von anderer Hardware, zum Beispiel von Prozessoren mit mehr Kernen, tritt dieser Effekt erst bei höheren Verteilungsstufen ein.

Auswertung des Speedups

Abschließend wurde der Speedup $S_p(n) = \frac{T'(n)}{T_p(n)}$, vergleiche Abschnitt 1.2.2, ausgewertet. Da die Problemgröße n, hier durch *to* definiert, innerhalb der Läufe konstant ist, wurde die Anzahl der Threads variiert, hier durch *maxWorkers* (Verteilungsstufe). Zu beachten gilt hierbei, dass $T_1(n) = T'(n)$ gesetzt worden ist, da hier nur die Läufe miteinander verglichen werden sollen und die Ergebnisse außerhalb der Läufe keine Aussagekraft besitzen. Da die einzelnen Messungen pro Verteilungsstufe variieren, werden diese über den Mittelwert und den Durchschnitt angegeben. Die Abbildung 6 stellt den Speedup in Relation zur Anzahl der verwendeten Threads dar.

Da der Speedup die Reduktion der Laufzeit angibt, lassen sich die Aussagen von der Auswertung zur Laufzeit auch auf diese Auswertung anwenden. Den Messungen und der Grafik lässt sich auch entnehmen, dass der Speedup nicht linear zunimmt, vergleiche Aussagen zu linearem und realem Speedup aus Abschnitt 1.2.2. D.h. die Geschwindigkeitssteigerung nimmt pro zusätzlicher Verteilungsstufe ab. Das bedeutet für die Applikation, dass die Verwendung von beispielsweise zwei Prozessoren die Laufzeit nicht um die Hälfte reduziert.

Fazit über beide Auswertungen

Die Auswertungen zeigen, dass die Verteilung der Berechnungen über mehrere Threads und damit über mehrere Prozessoren bzw. Prozessor-Kerne eine Geschwindigkeitssteigerung, also eine Reduktion der Laufzeit, bedeutet. Die Geschwindigkeitssteigerung nimmt pro zusätzlicher Verteilungsstufe ab, da der Overhead immer stärker zunimmt, der sich direkt auf die Laufzeit auswirkt. Ab einer bestimmten Verteilungsstufe nimmt sogar die Laufzeit wieder zu, was sich negativ auf den Speedup auswirkt.

Auf andere Applikationen übertragen bedeutet das, dass die Reduktion der Laufzeit über die Verteilung der Berechnungen auf mehrere Prozessoren bzw. Prozessor-Kerne möglich ist, unter Berücksichtigung dessen, dass der Overhead, zum Beispiel durch die Synchronisierung von Prozessor-Kernen zunimmt. Das bedeutet auch, dass das Hinzufügen von weiteren Prozessoren bzw. Prozessor-Kernen sich nicht linear auf die Verbesserung der Laufzeit auswirkt;

25 siehe `https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/profiler.html`

so führt das Hinzufügen eines weiteren Prozessors nicht zu einer Reduktion der Laufzeit um 50%.

Es liegt an den Entwicklern, die Anwendung sinnvoll nebenläufig zu implementieren, um eine sinnvolle Parallelisierung von Berechnungen zu ermöglichen und nicht die "gewonnene" Zeit durch die Steigerung des Overheads zu verlieren. Es sollte auch beachtet werden, dass es Vorgänge gibt, die nicht parallelisiert werden können, vergleiche Hinweis zur Fibonacci-Folge aus Abschnitt 2.

2.4 Race Condition an einem Beispiel

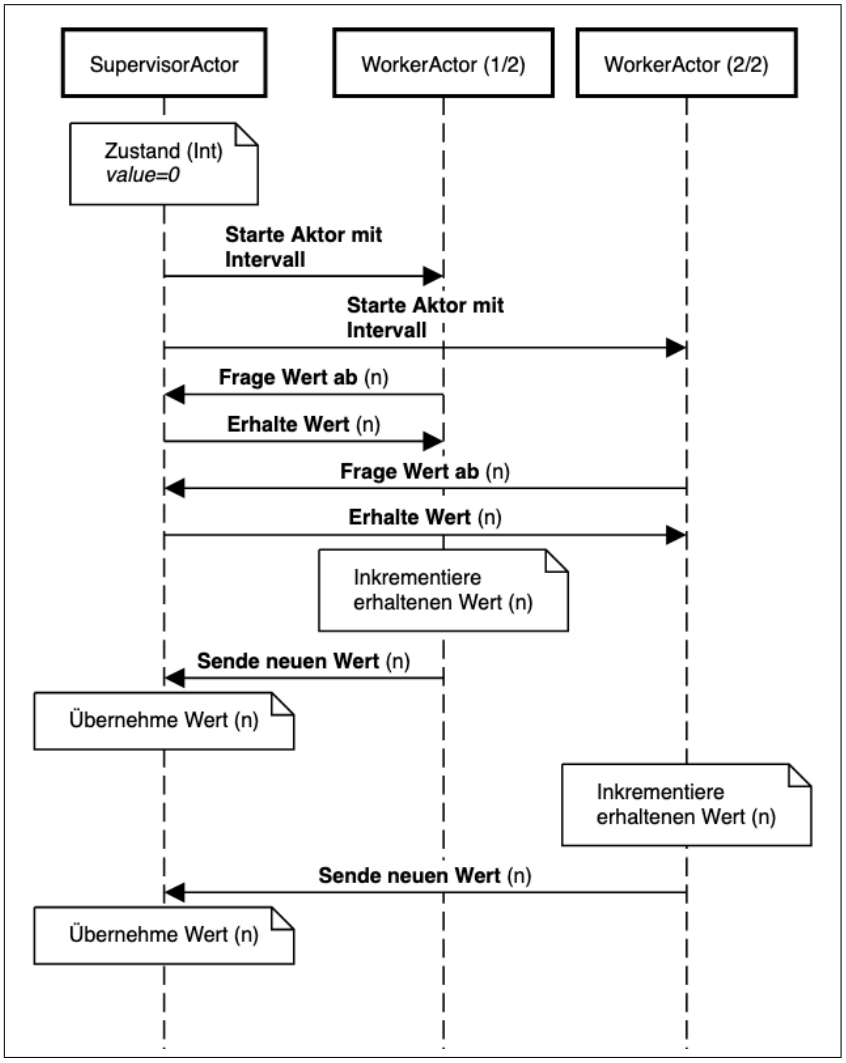

Abbildung 7: Sequenzdiagramm Race Condition als Beispielimplementierung

Im Abschnitt 1.1.3 wurde erläutert, dass eine Race Condition eine Konstellation bezeichnet, bei der das Ergebnis einer Operation vom zeitlichen Ablauf bestimmter Einzeloperationen abhängt und Fehler aus diesen häufig schwer reproduzierbar und damit schwer zu finden sind.

Um diese Aussage etwas greifbarer machen zu können, wurde eine Konstellation implemen-

tiert[26], bei der es zu einer Race Condition kommt. Die Beispielimplementierung startet zunächst einen Aktor (`SupervisorActor`) mit einem Zustand in Form eines ganzzahligen Werts (`Int`). Anschließend startet dieser Aktor zwei Aktoren (`WokerActor`), die in einem festen Intervall den Wert des Zustands vom `SupervisorActor` abfragen, inkrementieren und anschließend dem `SupervisorActor` zurücksenden. Dieser übernimmt den Wert und gibt den neuen und alten Wert per Logger aus. Abbildung 7 stellt grob den Ablauf der Anwendung als Sequenzdiagramm dar.

Das führt zunächst zur erwartenden Konsolenausgabe:

```
[11,188] [SupervisorActor$] - Start run with initial State(0)
[11,192] [WorkerActor$] - Start periodic timer
[11,192] [WorkerActor$] - Start periodic timer
[11,719] [SupervisorActor$] - OldState State(0) / NewState State(0)
[11,719] [SupervisorActor$] - OldState State(1) / NewState State(1)
[12,204] [SupervisorActor$] - OldState State(1) / NewState State(1)
[12,204] [SupervisorActor$] - OldState State(2) / NewState State(2)
[12,704] [SupervisorActor$] - OldState State(2) / NewState State(2)
[12,705] [SupervisorActor$] - OldState State(3) / NewState State(3)
[13,204] [SupervisorActor$] - OldState State(3) / NewState State(3)
```

Die ersten in eckigen Klammern angegeben Werte, z.B.: `[13,204]` stehen dafür, dass die Zeile in der Sekunde 13 und Millisekunde 204 geloggt worden ist. Wie man an der Ausgabe sehen kann, sind die geloggten Ergebnisse zunächst wie erwartet: Die Aktoren `WokerActor` erhalten jeweils den gleichen Wert vom `SupervisorActor`, inkrementieren diesen, anschließend wird dieser wieder vom `SupervisorActor` übernommen. Der Wert wird also immer doppelt inkrementiert, bis auf die erste Inkrementierung, da die Aktoren hier noch stärker zeitlich versetzt arbeiten (Platzierungszeit ist unterschiedlich). Doch nach kurzer Zeit:

```
[06,206] [SupervisorActor$] - OldState State(110) / NewState State(110)
[06,705] [SupervisorActor$] - OldState State(110) / NewState State(110)
[06,705] [SupervisorActor$] - OldState State(111) / NewState State(111)
[07,206] [SupervisorActor$] - OldState State(111) / NewState State(111)
[07,206] [SupervisorActor$] - OldState State(112) / NewState State(112)
[07,705] [SupervisorActor$] - OldState State(113) / NewState State(113)
[07,706] [SupervisorActor$] - OldState State(114) / NewState State(114)
[08,206] [SupervisorActor$] - OldState State(114) / NewState State(114)
[08,206] [SupervisorActor$] - OldState State(115) / NewState State(115)
[08,706] [SupervisorActor$] - OldState State(115) / NewState State(115)
```

In den Berechnungen hat sich nach kurzer Zeit bereits ein Fehler eingeschlichen: Hier wurde der Wert `112` nicht wie erwartend doppelt inkrementiert. Das kann zum Beispiel passieren, wenn der Thread von einem `WorkerActor` vom Scheduler kurzzeitig pausiert wird, vergleiche Abschnitt 1.1.2. Das führt dazu, dass der "pausierte" `WorkerActor` den nicht-inkrementierten Wert nicht "mitbekommt" und daher den bereits inkrementierten Wert noch einmal inkrementiert.

26 siehe `https://github.com/maxbundscherer/scala-akka-prime-speedup/blob/master/src/main/scala/de/maxbundscherer/akka/scala/prim/examples/RCExample.scala`

2.5 Ausblick Steigerung der Ausfallsicherheit

Diese Arbeit möchte noch einen Ausblick darüber geben, wie man Anwendungen über die Verteilung auf mehrere Computer ausfallsicherer gestalteten kann. Der Begriff **Ausfallsicherheit** ist nicht eindeutig definiert, der Autor der Arbeit versteht darunter, eine Applikation durch die Verteilung auf mehrere Computer, zum Beispiel Server, also durch den Einsatz von Redundanzen, gegen die Nicht-Erreichbarkeit zu schützen.

Verwiesen wird zunächst auf das CAP Theorem, auch Brewers Theorem genannt: Nach diesem Theorem sind nur zwei von drei möglichen Eigenschaften in einem verteilen System möglich [Gil02]:

- **Konsistenz** (**C**onsistency): Die gesamten gespeicherten Daten sind konsistent.
- **Verfügbarkeit** (**A**vailability): Anfragen an das System werden beantwortet.
- **Partitionstoleranz** (**P**artition tolerance): Das System arbeitet auch weiter, wenn zum Beispiel einzelne Server innerhalb eines Clusters nicht mehr erreichbar sind.

Da bei einer Verteilung über mehrere Server gewährleistet werden muss, dass das System auch weiterarbeiten kann, wenn einzelne Server nicht mehr erreichbar sind, muss auf jeden Fall die Eigenschaft *Partition tolerance* erfüllt sein. Dem Entwickler steht nun frei, das System mit einer weiteren Eigenschaft zu implementieren:

- **C+P**: Die Daten sind global gesehen immer konsistent, das System ist evtl. nicht zu jedem Zeitpunkt erreichbar. Als Datenbank könnte hierfür beispielsweise *Redis*[27] oder *MongoDB*[28] verwendet werden.
- **A+P**: Die Daten sind global gesehen nicht immer konsistent, das System ist aber immer erreichbar. Als Datenbank könnte hierfür *Apache Cassandra*[29] verwendet werden. In diesem Kontext fällt häufig der Begriff **Eventual Consistency**[30], auf diesen kann aber im Rahmen der Arbeit nicht mehr eingegangen werden, da dies sonst den Rahmen der Arbeit sprengen würde.

Es sollte hierbei erwähnt werden, dass es sich bei dem CAP-Theorem um theoretische Eigenschaften handelt, so kann beispielsweise die Availability nicht gewährleistet werden, wenn die Internet-Anbindung des Rechenzentrums ausfällt.

Die Verteilung von Akka Aktoren über mehrere Server kann zum Beispiel über den Einsatz von Akka Cluster, vergleiche Abschnitt 2.2.2, gewährleistet werden. Akka Cluster gibt nicht vor, ob es sich um ein System mit den Eigenschaften C+P oder A+P handelt. Diese Eigenschaften zeichnen sich zum Beispiel durch Verwendung einer geeigneten Datenbank und der Implementierung selbst ab.

Des weiteren sollte darauf geachtet werden, dass Daten nicht verloren gehen, wenn ein Computer aus dem Cluster nicht mehr reagiert und neugestartet wird: Um den Zustand eines Akka

27 siehe `https://redis.io/`
28 siehe `https://www.mongodb.com/`
29 siehe `https://cassandra.apache.org/`
30 siehe `https://dl.acm.org/doi/10.1145/1435417.1435432`

Aktors nach dem Beenden wiederherstellen zu können, kann **Akka Persistence**[31] verwendet werden. Akka Persistence ist eine Erweiterung für Akka, die es ermöglicht, den Zustand eines Akka Aktors mithilfe der Speicherung und Verwaltung von Events (die durch die eingehenden Nachrichten definiert werden) und optionalen Momentaufnahmen (Snapshots) auch nach dem Beenden der Aktoren wiederherzustellen.

Damit ist *Akka Persistence* eine Implementierung des Event Sourcing (ES)-Ansatzes. Beim ES werden alle Veränderungen des Zustands eines Systems in Form von Events abgebildet [Pac18, S.115]. Durch diese Architekturentscheidung ist es möglich, das komplette System zu jedem Zeitpunkt wiederherzustellen. Das unterstützt nicht nur bei der Fehlersuche, sondern ermöglicht es auch, besser zu verstehen, wie mit dem System gearbeitet wird.

Im Abschnitt 1.1.3 wurde erwähnt, dass zum Beispiels Deadlocks für Cloud-Umgebungen von besonderer Relevanz sind, da diese zu einem Totalausfall führen könnten. Ein Deadlock könnte beispielsweise entstehen, wenn ein Akka Aktor auf einen anderen wartet, der aber nicht antwortet und evtl. selbst auf einen Aktor wartet. Um dies zu vermeiden, kann beispielsweise ein Akka Aktor auf einen anderen Aktor nur eine maximale Zeit warten, weshalb bei `ask`, vergleiche Abschnitt 2.2.2, diese Zeit definiert werden musste. Falls in der vorgegebenen Zeit keine Antwort eingetroffen ist, lässt Akka den Aktor abstürzen und neustarten. Dieses Verhalten wird in der *Supervisor Strategie*[32] von Akka definiert.

31 siehe `https://doc.akka.io/docs/akka/current/persistence.html`
32 siehe `https://doc.akka.io/docs/akka/current/fault-tolerance.html`

A Anänge

A.1 Ergebnisse der Messungen

Die Ergebnisse der Messungen lassen sich unter `https://github.com/maxbundscherer/scala-akka-prime-speedup/blob/master/results.csv` einsehen:

```
to,resultsSize,startTime,maxWorkers,time
720720,58084,1076321827730,1,290840815916
720720,58084,1367225151264,1,291387615803
720720,58084,1658643477235,1,278438275264
720720,58084,1937127234899,1,278406359526
720720,58084,2215566355452,1,278412696104
720720,58084,2494027662397,2,204721940674
720720,58084,2698783298022,2,204715441299
720720,58084,2903529989798,2,204732259299
720720,58084,3108289888033,2,204738063774
720720,58084,3313072275031,2,204776110272
720720,58084,3517888044136,3,150731133757
720720,58084,3668658094731,3,150720842650
[...]
```

Hinweis: Die Ergebnisse wurden mit `[...]` gekürzt.

A.2 Log der Messläufe

Das Log zu den Messungen lässt sich unter `https://github.com/maxbundscherer/scala-akka-prime-speedup/blob/master/log.txt` einsehen:

```
[info] welcome to sbt 1.3.12 (Private Build Java 13.0.3)
[info] loading project definition from
   /home/ubuntu/scala-akka-prime/project
[info] loading settings for project scala-akka-prime from build.sbt ...
[info] set current project to scala-akka-prime (in build
   file:/home/ubuntu/scala-akka-prime/)
[info] Compiling 1 Scala source to
   /home/ubuntu/scala-akka-prime/target/scala-2.13/classes ...
[info] Done compiling.
[info] running de.maxbundscherer.akka.scala.prim.Main
[2020-07-17 08:21:19,681] [INFO] [akka.event.slf4j.Slf4jLogger] -
   Slf4jLogger started {}
[2020-07-17 08:21:19,706] [INFO]
   [de.maxbundscherer.akka.scala.prim.actors.SupervisorActor$] - Start
   run (StartRunCmd(720720,1,results.csv))
   {akkaAddress=akka://actorSystem, akkaSource=akka://actorSystem/user,
   sourceActorSystem=actorSystem}
[2020-07-17 08:26:10,552] [INFO]
   [de.maxbundscherer.akka.scala.prim.actors.SupervisorActor$] -
   Finished run (StartRunCmd(720720,1,results.csv))
   {akkaAddress=akka://actorSystem, akkaSource=akka://actorSystem/user,
   sourceActorSystem=actorSystem}
[...]
```

Hinweis: Das Log wurde mit `[...]` gekürzt.

A.3 Skripte für AWS EC2 Instanzen mit Ubuntu Server 20.x

Für die Messungen wurde ein Skript für die Installation und ein Skript für den eigentlichen Lauf geschrieben.

A.3.1 EC2 Installation

Der Skript lässt sich unter `https://github.com/maxbundscherer/scala-akka-prime-speedup/blob/master/ec2-install.sh` einsehen:

```
echo "JAVA INSTALL"

sudo apt-get update
sudo apt install openjdk-13-jre-headless

echo "SBT INSTALL"
echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a
   /etc/apt/sources.list.d/sbt.list
curl -sL "https://keyserver.ubuntu.com/pks/lookup?op=get&search=[...]" |
   sudo apt-key add
sudo apt-get update
sudo apt-get install sbt

echo "GIT CLONE"
git clone https://github.com/maxbundscherer/scala-akka-prime
```

Hinweis: Das Skript wurde mit `[...]` gekürzt.

A.3.2 EC2 Lauf

Der Skript lässt sich unter `https://github.com/maxbundscherer/scala-akka-prime-speedup/blob/master/ec2-run.sh` einsehen:

```
echo "SET SBT OPTS"
export SBT_OPTS="-Xss2M -Xms4G -Xmx15G"

echo "RUN SBT"
sbt run >> log.txt

echo "SHUTDOWN SERVER"
sudo shutdown now
```

Hinweis: Das Skript wurde mit `[...]` gekürzt.

Literatur

[Akk20] Akka: Offizieller Internetauftritt. https://akka.io/, 2020. Abgerufen am 14.07.2020.

[Ben15] G. Bengel. In *Masterkurs Parallele und Verteilte Systeme (978-3-8348-2151-5)*. Springer, 2015.

[Bra10] O. Braun. In *Scala (978-3-446-42399-2)*. Hanser, 2010.

[Gil02] S. Gilbert. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. https://dl.acm.org/doi/10.1145/564585.564601, 2002. Heruntergeladen von https://users.ece.cmu.edu/~adrian/731-sp04/readings/GL-cap.pdf - Abgerufen am 25.07.2020.

[Hew73] C. Hewitt. A universal modular ACTOR formalism for artificial intelligence. https://dl.acm.org/doi/10.5555/1624775.1624804, 1973. Heruntergeladen von https://www.ijcai.org/Proceedings/73/Papers/027B.pdf - Abgerufen am 14.07.2020.

[Imga] Brianstorti - The actor model in 10 minutes - Bild Exemplarische Darstellung Aktorenmodell. https://www.brianstorti.com/the-actor-model/. Abgerufen am 14.07.2020.

[Imgb] Wikimedia Commons - Bild Exemplarische Darstellung FIFO. https://commons.wikimedia.org/wiki/File:Fifo_queue.png. Abgerufen am 14.07.2020.

[Pac18] V. Pacheco. In *Microservice Patterns and Best Practices (9781788471206)*. Packt Publishing, 2018.

[Pie10] L. Piepmeyer. In *Grundkurs funktionale Programmierung mit Scala (978-3-446-42416-6)*. Hanser, 2010.

[Rau12] T. Rauber. In *Parallele Programmierung (978-3-642-13604-7)*. Springer, 2012.

[Ris16] S. Ristov. Superlinear Speedup in HPC Systems: why and when? http://dx.doi.org/10.15439/2016F498, 2016. DOI: 10.15439/2016F498 - Abgerufen am 12.07.2020.

[Roe16] R. Roestenburg. In *Akka in Action (978-1617291012)*. Manning Publications, 2016.

[Sca20a] Scala: Offizieller Internetauftritt. https://www.scala-lang.org/, 2020. Abgerufen am 14.07.2020.

[Sca20b] Scala: Seamless integration with Java. https://www.scala-lang.org/old/node/25, 2020. Abgerufen am 14.07.2020.

[Uel19] M. Uelschen. In *Software Engineering paralleler Systeme (978-3-658-25343-1)*. Springer, 2019.

[Ull16] C. Ullenboom. Java ist auch eine Insel. http://openbook.rheinwerk-verlag.de/javainsel/, 2016. Abgerufen am 12.07.2020.

[Vog12] C. Vogt. In *Nebenläufige Programmierung (978-3-446-43201-7)*. Hanser, 2012.