Hochschule Luzern, Informatik

Auftraggeber/Betreuer: HSLU I / Prof. Dr. Ruedi Arnold
Experte: Roland Christen
Wirtschaftsprojekt von Nico Bosshard

Nebenläufige Programmiermodelle

```
2   main = do
3       channel <- atomically $ newTChan
4       guard <- newQSem 10
5       let seed = 0
6       forkIO (forM_ [1..] $ \i -> do
7               waitQSem guard
8               let currentSeed = seed + (0x9E3779B97F4A7C15 * i) :: Word64
9               let z1 = (currentSeed `xor` (currentSeed `shiftR` 30)) * 0xBF58476D1CE4E5B9
10              let z2 = (z1 `xor` (z1 `shiftR` 27)) * 0x94D049BB133111EB
11              let result = (z2 `xor` (z2 `shiftR` 31)) `shiftR` 31
12              atomically $ writeTChan channel result
14          )
15      forM_ [1..100] $ \_ -> do
16          threadDelay (1000)
17          signalQSem guard
18          x <- atomically $ readTChan channel
19          putStrLn $ show x
```

Rotkreuz, 28. Mai 2021

Wirtschaftsprojekt an der Hochschule Luzern – Informatik

Titel: Nebenläufige Programmiermodelle

Studentin/Student: Nico Bosshard

Studiengang: BSc Informatik

Jahr: 2021

Betreuungsperson: Prof. Dr. Ruedi Arnold

Expertin/Experte: Roland Christen

Auftraggeberin/Auftraggeber: HSLU I

Codierung / Klassifizierung der Arbeit:
☒ A: Einsicht (Normalfall)
☐ B: Rücksprache (Dauer: Jahr / Jahre)
☐ C: Sperre (Dauer: Jahr / Jahre)

Eidesstattliche Erklärung
Ich erkläre hiermit, dass ich/wir die vorliegende Arbeit selbständig und ohne unerlaubte fremde Hilfe angefertigt haben, alle verwendeten Quellen, Literatur und andere Hilfsmittel angegeben haben, wörtlich oder inhaltlich entnommene Stellen als solche kenntlich gemacht haben, das Vertraulichkeitsinteresse des Auftraggebers wahren und die Urheberrechtsbestimmungen der Fachhochschule Zentralschweiz (siehe Merkblatt «Studentische Arbeiten» auf MyCampus) respektieren werden.

Ort / Datum, Unterschrift Oberwil, 28.5.2021 N. Bosshard

Ort / Datum, Unterschrift _____

Ausschliesslich bei Abgabe in gedruckter Form:
Eingangsvisum durch das Sekretariat auszufüllen

Rotkreuz, den _____ Visum: _____

Abstrakt

Mit der Zunahme der verfügbaren CPU-Cores gewinnt die parallele Programmierung immer mehr an Bedeutung. Das Ziel in dieser Arbeit ist es, eine Übersicht über die aktuellen nebenläufigen Programmiermodelle und nebenläufigen Kontrollkonzepte zu erstellen. Die Analyse und der Vergleich werden mit einer Auswahl verschiedener Programmiersprachen realisiert. Welche Sprachen welche parallelen Programmiermodelle und Nebenläufigkeitskonzepte unterstützen wird mit Codebeispielen analysiert, demonstriert, verglichen, kommentiert und zusammenfassend in tabellarischer Form festgehalten. Es werden die Programmiersprachen Java, Golang, C#, Erlang, Kotlin und Haskell untersucht. Für die nebenläufigen Kontrollkonzepte werden Atomics, Mutex, Software transactional Memory, Channels und Messages angeschaut. Bei den nebenläufigen Programmiermodellen werden Thread Pools, Futures/Tasks, Asynchrone Programmierung, eventbasierte Parallelisierung / Reactive, Continuation, Coroutines, Fibers, Message passing Channels und Actor Concurrency analysiert und verglichen. Sämtliche Codebeispiele für alle Programmiersprachen können in einer einzigen Entwicklungsumgebung geöffnet und ausgeführt werden.

Inhalt

1 Fragestellung

Bei einer nebenläufigen Programmierung werden mehrere Befehle oder Anweisungen gleichzeitig ausgeführt. Diese parallele Ausführung, englisch concurrency genannt, gewinnt mit den heute üblichen Mehrkernprozessoren immer mehr an Bedeutung, obwohl eine nebenläufige Programmierung auch bei Single-Core-Prozessoren verwendet wird. Das Gegenstück ist die sequentielle Programmierung, wo eine Folge von Befehlen nacheinander ausgeführt wird. In dieser Arbeit sollen verschiedene nebenläufige Programmiermodelle analysiert und verglichen werden.

2 Einführung und Begriffserklärungen

In dieser Arbeit werden fachspezifische Begriffe und Konzepte verwendet. Teilweise werden in den verschiedenen Programmiersprachen unterschiedliche Begriffe verwendet. Um die Vergleichbarkeit zu gewährleisten werden überall die gleichen Begriffe, welche nachfolgend beschrieben werden, verwendet.

Race Condition: Greifen mehrere Threads genau gleichzeitig auf dieselbe Speicherregion zu und ist eine dieser Speicheroperationen schreibend, so kommt es zu einer race condition. Das Resultat einer race condition ist undefiniert. Race conditions müssen für fehlerfreie parallele Programmierung vermieden oder behandelt werden. Race conditions sind nicht auf Speicherzugriffe limitiert, sondern existieren auch bei Dateizugriffen und API- Aufrufen.

Deadlocks: Bleibt ein Thread in einer kritischen Region stecken oder vergisst er den Mutex wieder freizugeben, so können keine weiteren Threads mehr diese kritische Region betreten. Das Programm kommt zu einem Stillstand. Dasselbe passiert in einer Situation, in der alle Threads vom Fortschreiten eines anderen Threads abhängig sind. Durch lockfreie Programmierung können Deadlocks verhindert werden.

Atomics: Atomare Datentypen können von mehreren Threads gleichzeitig gelesen und geschrieben werden, ohne dass ein Mutex benötigt wird. Dies wird intern über Konzepte wie test-and-set oder compare-and-swap realisiert. Programme, welche ausschliesslich Atomics nutzen, werden als lockfrei bezeichnet und haben die Eigenschaft, dass sie den physikalischen Ausfall eines CPUs verkraften können. Viele sicherheitskritische Systeme müssen lockfrei

programmiert werden. Atomare Datentypen sie sind der Grundbaustein und somit die schnellsten aller threadsicheren Datentypen.

Mutex: Ein Mutex ermöglicht das Definieren einer kritischen Region, welche gleichzeitig nur von einem Thread ausgeführt werden kann. Mutex werden manchmal auch als Locks bezeichnet. Der einzige Unterschied zwischen einem Mutex und einem Lock ist, dass ein Mutex nicht zwingend an einen Prozess gebunden sein muss sondern auch Systemweit implementiert werden kann.

Guard: Ein Guard ist ähnlich wie ein Mutex mit dem Unterschied, dass anstelle eines einzigen Threads eine definierte Anzahl von Threads gleichzeitig die kritische Region betreten können. Guards werden oft durch Semaphoren oder bounded Channels implementiert. Der Begriff Guard ist einer Semaphore gleichzusetzen.

Software transactional Memory (STM): Eine Transaktion ist ein Codeausschnitt welcher entweder erfolgreich ausgeführt wird oder es wird der Zustand vor der Transaktion wiederhergestellt. Das STM ist das optimistische Äquivalent zum pessimistischen Mutex.

Channels: Mit Concurrent Queues, Channels und Messages können Nachrichten zwischen verschiedenen Threads ausgetauscht werden. Anders als Channels sind Messages immer unlimitiert von der Anzahl zu speichernden Nachrichten (unbounded). Messages beschränken sich nicht auf einen einzigen Computer, sondern sie können auch über das Internet versendet werden.

Thread: Jeder Thread wird einem eigenen OS Thread zugeordnet. Diese werden wiederum vom OS Scheduler auf CPU Cores verteilt. Jeder OS Thread hat eigene CPU Register, einen eigenen Programm Counter, einen eigenen Stack sowie einen Prozesssteuerungsblock. Durch ein Kontextwechsel wird vom OS Scheduler zwischen OS Threads gewechselt. Bei jedem Kontextwechsel müssen die Register in den Prozesssteuerungsblock gespeichert und die Register des neuen Threads aus dem Prozesssteuerungsblock wiederhergestellt sowie die CPU Pipeline verworfen werden.

Fiber: Ein leichtgewichtiger virtueller Thread, welcher beinahe in beliebiger Menge erstellt werden kann und wesentlich schnelleren Kontextwechsel erlaubt. Tausende von Fibers können miteinander einen OS Thread teilen. Fibers werden durch den in der Programmiersprache integriertem Scheduler und nicht dem OS Scheduler verwaltet.

Thread Pools: Ein Pool von wiederverwendbaren Threads. Hat man keine Fibers, so ist dies eine gute Alternative, da dadurch der hohe Ressourcenaufwand vom ständigen Erstellen/Löschen von Threads vermieden wird.

Asynchrone Programmierung: Durch Asynchrone Programmierung kann der Code sequentiell geschrieben werden und Datenabhängigkeiten können markiert werden. Es ist Sache des Compilers sich um die Parallelisierung des Programms zu kümmern.

Eventbasierte Parallelisierung: Mit der eventbasierten Parallelisierung kann auf bestimmte Ereignisse über einen asynchronen Task reagiert werden. Reactive ist eine Teilmenge der eventbasierten Parallelisierung.

Continuation: Eine unterbrechbare Funktion, welche genau an der unterbrochenen Stelle wieder fortgesetzt werden kann. Continuations werden als Bausteine für Coroutines sowie moderne Enumeratoren benötigt.

Coroutines: In dieser Arbeit werden Coroutinen als Continuation mit vollautomatisiertem Threadmanagement definiert. Goroutinen und Coroutinen werden als äquivalent betrachtet.

Actor Concurrency: Bei Actor Concurrency wird das Programm in mehrere, unabhängige Aktoren aufgeteilt, welche durch Messages miteinander kommunizieren. Dadurch werden geteilte Variablen und somit race conditions vermieden.

3 Konzept

Für die zu untersuchenden Programmiersprachen programmiere ich ausführbare Beispielcodes zu den verschiedenen nebenläufigen Programmiermodellen. Dabei wird Java immer als Referenz für den Vergleich mit den nebenläufigen Programmiersprachen verwendet. Es werden die Vor- und Nachteile sowie die unterschiedliche Umsetzung der Programmiersprachen aufgezeigt.

3.1 Nebenläufige Programmiermodelle

In dieser Arbeit werden folgende Programmiermodelle / Konzepte untersucht:

- Atomics ..... (nebenläufiges Kontrollkonzept)
- Mutex ..... (nebenläufiges Kontrollkonzept)
- Software transactional Memory ..... (nebenläufiges Kontrollkonzept)
- Channels / Messages ..... (nebenläufiges Kontrollkonzept)
- Thread Pools ..... (nebenläufiges Programmiermodell)
- Futures/ Promises/Tasks ..... (nebenläufiges Programmiermodell)
- Asynchrone Programmierung ..... (nebenläufiges Programmiermodell)
- Eventbasierte Parallelisierung / Reactive ..... (nebenläufiges Programmiermodell)
- Continuation ..... (nebenläufiges Programmiermodell)
- Coroutines / Goroutines ..... (nebenläufiges Programmiermodell)
- Message passing Channels ..... (nebenläufiges Programmiermodell)
- Actor Concurrency ..... (nebenläufiges Programmiermodell)

3.2 Programmiersprachen

Untersucht wird die Nebenläufige Programmierung an folgenden Programmiersprachen:

- Java
- Golang
- C#
- Erlang
- Kotlin
- Haskell

4 Vorstellung der Codebeispiele

Um die Übersicht zu behalten werden nur die drei folgenden Beispielcodes für den Vergleich der Programmiersprachen und den parallelen Programmierkonzepten besprochen. Das erstes Codebeispiel Count ist ein Zähler mit race condition. Das zweite Beispiel ist ein asynchroner HTTP-Request und das dritte Codebeispiel ist ein nebenläufiger Pseudozufallszahlen-generator. Sämtliche Codebeispiele sind auf GitLab unter dem Link: https://gitlab.enterpriselab.ch/cc/examples verfügbar. Der Ordner kann mit Visual Studio Code geöffnet werden.

4.1 Count

Bei dem Beispielprogramm Count soll der Umgang der verschiedenen Programmiersprachen mit race conditions demonstriert werden. Dazu wird ein Zähler mit mehreren Threads gleichzeitig hoch- und wieder heruntergezählt. Ist der Test erfolgreich, muss der Zähler am Ende wieder auf null sein. Beispielsweise zählen 23 verschiedene Threads gleichzeitig von null auf eine Million in Einerschritten hoch und danach wieder in Einerschritten herunter. Eine race condition tritt auf, wenn zwei Threads genau gleichzeitig in die gleiche Speicheradresse schreiben oder gleichzeitig lesen und schreiben wollen. Eine race condition führt zu einem undefinierten Ergebnis. Es ist darum ein Hauptanliegen in der parallelen Programmierung diese race conditions zu verhindern.

Beispiel als Pseudocode:

```
Counter = 0
do parallel:
        for 0 to 1000000:
                Counter++
        for 0 to 1000000:
                Counter--
```

4.2 Asynchroner HTTP-Request

Eine häufige Anwendung ist asynchron einen HTTP Request zu senden, auf eine Antwort zu warten und diese Antwort dann dem Main Thread mitzuteilen. Da der Main Thread nicht auf den langsamen Verbindungsaufbau und die Antwort warten muss, wird dies asynchron, das heisst parallel ausgeführt.

Bei diesem Test wird parallel ein http get request auf eine Webseite (http://www.nicobosshard.ch/Hi.html) gesendet, währenddem der Main Thread eine Aufgabe, wie beispielsweise eine 2^{256} bit grosse Primzahl suchen, löst. Der durch den response erhaltenen html-Code wird dem Main Thread mitgeteilt. Ist der Main Thread mit der Rechenaufgabe vorher fertig, so wartet er auf die html-Rückmeldung. Der Test ist erfolgreich, wenn der Main Thread den korrekten html-Code erhält.

Bei gewissen Beispielen werden 10 Requests gesendet und 10 Antworten erwartet um parallele Programmiermodelle zu zeigen. Dies, weil bei einigen Modellen wie beispielsweise Threadpools, es keinen Sinn macht mit nur einem parallelen Thread zu arbeiten.

Beispiel als Pseudocode:

```
Do asynchronous:
        result = get "http://www.nicobosshard.ch/Hi.html"
Do some work
Wait for asynchronous task
assert result == "<html><body>Hi!</body></html>"
```

4.3 Pseudozufallszahlengenerator (PRNG)

In diesem Beispiel geht es um einen pseudozufälligen Zufallsgenerator. Dieser wird mit einem Seed gestartet und läuft parallel zum Main Thread. Der Main Thread hat jederzeit die Möglichkeit eine zuvor vom Zufallsgenerator berechnete Zufallszahl abzurufen. Ist keine vorhanden, so wird der Main Thread auf die nächste Zufallszahl warten. Der Zufallsgenerator berechnet immer eine bestimmte Anzahl Zufallszahlen und stellt diese dem Main Thread zur Verfügung. Wird vom Main Thread eine Zufallszahl abgeholt, so wird der Zufallsgenerator eine neue Zufallszahl generieren um den Vorrat wieder aufzufüllen. Der Pseudozufallsgenerator

basiert auf C++ splitmix PRNG von Arvid Gerstmann (Gerstmann, 2018). Er musste für die Verwendung in alle getesteten Programmiersprachen übersetzt werden.

Beispiel als Pseudocode:

```
Channel of capacity 10
Do asynchronous
        Repeat forever:
                Generate random number
                Wait for channel to have capacity left:
                        Put the random number in the channel
for 0 … 100:
        Wait for channel to contain something:
```

5 Nebenläufige Kontrollkonzepte

Die Umsetzungen des Codebeispiels Count, siehe Kapitel 4.1, mit verschiedenen Nebenläufigen Kontrollkonzepten in Java werden in den nachfolgenden Unterkapiteln beschrieben.

5.1 Unsafe

Das Keyword *volatile* in Java reicht nicht aus, um das Problem von race conditions zu lösen, da ein Inkrement wie ++i nicht in eine einzige Instruktion übersetzt wird. Es werden 3 Instruktionen erzeugt. Die erste lädt den Zustand der Variablen in ein Register, die zweite erhöht diese um Eines und die letzte speichert das Resultat zurück in die Variable. Wird zwischen diesen Instruktionen der Thread gewechselt sieht der andere Thread noch den alten Zustand der Variablen, was in diesem Beispiel dazu führt, dass gewisse Inkrement- und Dekrement-Operationen durch andere überschrieben werden. Dieses Beispiel ist nicht threadsave und das Resultat ist deswegen undefiniert und je nach Scheduler Timing auch jedes Mal anders. (Javin, 2020)

```
1   public final class UnsafeCounter implements Counter {
2       private volatile int i;
3       public UnsafeCounter() {
4           i = 0;
5       }
6       @Override public void increment() {
7           ++i;
8       }
12      @Override public int get() {
13          return i;
14      }
15  }
```

5.2 Atomic

Um atomaren Zugriff auf Variablen zu ermöglichen müssen zwei Dinge sichergestellt werden:

1. Die Variable wird nicht gecached. Dies geschieht in Java über das Keyword *volatile*
2. Ein Thread-Wechsel während einer atomaren Operation muss erkannt und darauf reagiert werden. Dazu gibt es mehrere Möglichkeiten, welche jedoch alle Hardware-Unterstützung vom CPU erfordern.

a. Die einfachste ist test-and-set. Jeder der auf die Variablen zugreifen will, landet in einem Spinlock. Es wird so lange eine vom CPU bereitgestellte atomare test-and-set Operation durchgeführt bis diese erfolgreich ist.
b. Eine weitere Option ist fetch-and-add. Hier wird eine Variable in demselben Speicherzyklus eingelesen, erhöht und wieder zurückgeschrieben.
c. Die mächtigste atomare Operation ist compare-and-swap. Es wird der Funktion compare-and-swap der Speicherbereich, der erwartete alte Wert und der gewünschte neue Wert übergeben. Die Operation ist nur erfolgreich, falls der erwartete aktuelle Zustand der Variablen mit dem tatsächlichen Zustand der Variablen übereinstimmt. Wenn dies zutrifft, wird der neue Wert in den Speicherbereich geschrieben.

Atmoare Operationen sind die Bausteine auf welche lock und lockfreie Konzepte aufbauen.

Durch atomare Klassen wie AtomicInteger kann in Java ein Integer Objekt erstellt werden, welches atomare Operationen implementiert.

Folgende Atomaren Klassen existieren seit Java 7:

AtomicBoolean, AtomicInteger, AtomicIntegerArray, AtomicIntegerFieldUpdater<T>, AtomicLong, AtomicLongArray, AtomicLongFieldUpdater<T>, AtomicMarkableReference<V>, AtomicReference<V>, AtomicReferenceArray<E>, AtomicReferenceFieldUpdater<T,V>, AtomicStampedReference<V>. (Oracle, Package java.util.concurrent.atomic, 2020)

Weiterführende Wissensquellen:

What-is-atomicinteger-class-and-how-it-works-internally. [1]
Test-and-set. [2]
Spinnlock. [3]
Fetch-and-add. [4]
Compare-and-swap (Wikipedia). [5]
Compare-and-swap (Jenkov). [6]

[1] https://www.javacodemonk.com/what-is-atomicinteger-class-and-how-it-works-internally-1cda6a56 (Carvia, 2019)
[2] https://en.wikipedia.org/wiki/Test-and-set (Wikipedia, Test-and-set, 2020)
[3] https://en.wikipedia.org/wiki/Spinlock (Wikipedia, Spinlock, 2021)
[4] https://de.wikipedia.org/wiki/Fetch-and-add (Wikipedia, Fetch-and-add, 2020)
[5] https://de.wikipedia.org/wiki/Compare-and-swap (Wikipedia, Compare-and-swap, 2019)
[6] http://tutorials.jenkov.com/java-concurrency/compare-and-swap.html (Jenkov, 2019)

```
1   public final class AtomicCounter implements Counter {
2       private final AtomicInteger i;
3       public AtomicCounter() {
4           i = new AtomicInteger(0);
5       }
6       @Override public void increment() {
7           i.incrementAndGet();
8       }
12      @Override public int get() {
13          return i.get();
14      }
15  }
```

Codebeschrieb:

Zeile 2: Ein atomarer Integer wird erstellt.

Zeile 7: Eine atomare Increment-/Decrement-Operation wird ausgeführt. Diese wird von Java im Hintergrund mit compare-and-swap ausgeführt. Es wird sichergestellt, dass die Operation so lange wiederholt wird bis sie erfolgreich ist.

Zeile 13: Der Wert des atomaren Integers wird ausgelesen und als int zurückgegeben.

Achtung: Durch Falschbenutzen eines AtomicIntegers können auch hier race conditions auftreten. Würde man beispielsweise i.set = i.get() + 1 schreiben, hätte man wieder genau dasselbe Problem, da dies nicht mehr eine atomare Operation wäre.

5.3 Synchronized

Durch die zuvor behandelten atomaren Operatoren werden von den meisten Programmiersprachen Locks implementiert und dem Programmierer bereitgestellt. Ein Lock kann immer nur von einem Thread genommen werden, während alle anderen Threads auf die erneute Freigabe des Locks warten müssen. Dadurch können kritische Codeausschnitte, welche race conditions erhalten, in nicht-parallelerweise ausgeführt werden, während der Rest des Programmes parallel ausgeführt werden kann. Da dabei die Parallelität des Programms vermindert wird, sollen kritische Abschnitte so kurz wie möglich gehalten werden. Ansonsten wird das Programm nach einer bestimmten Anzahl CPU-Cores nur noch einen sehr geringen Performancegewinn durch weitere Cores erhalten.

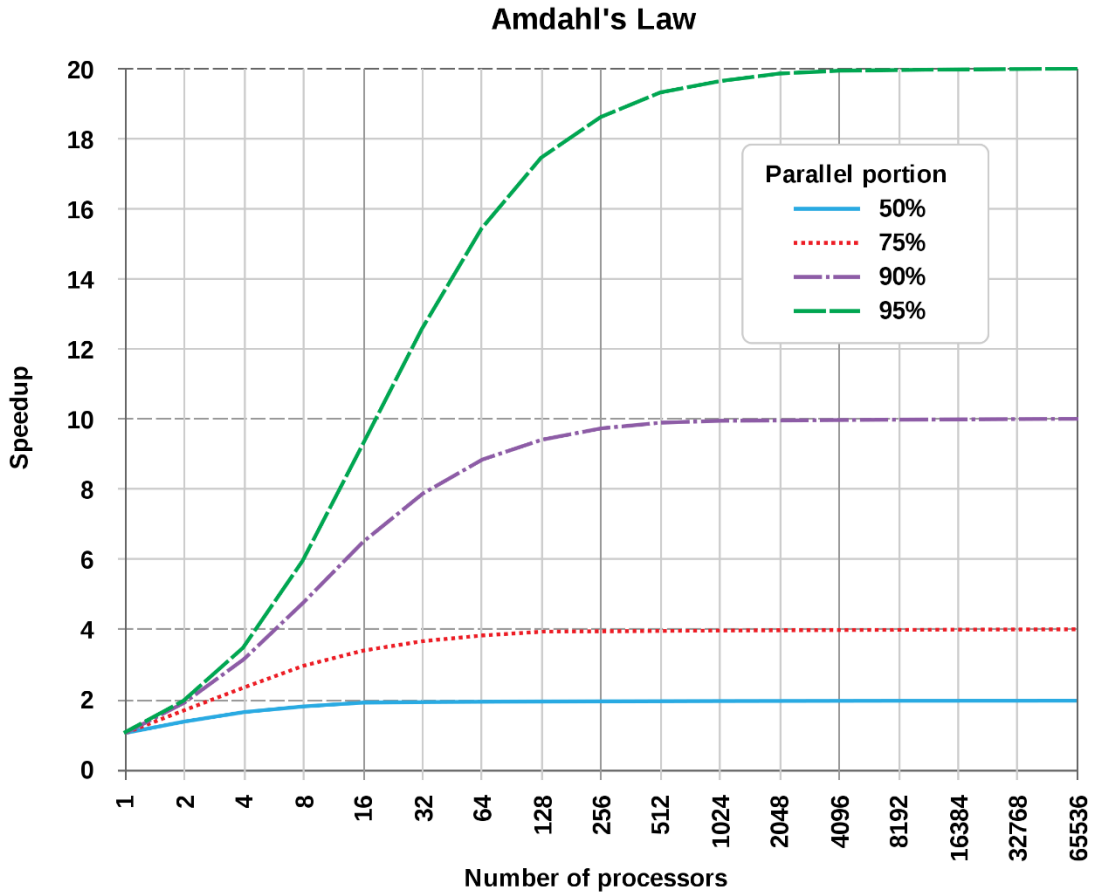

Abbildung 1: Amdahl's Gesetz
Quelle: https://en.wikipedia.org/wiki/Amdahl%27s_law

Eine weitere Gefahr durch Locks sind sogenannte Deadlocks. Bleibt ein Thread aus irgendeinem Grund in einer kritischen Region stecken oder vergisst das Lock wieder freizugeben, bleibt das ganze Programm hängen. Dies ist in sicherheitsrelevanten Systemen unakzeptabel und es muss deswegen lock-free programmiert werden. Um das Konzept von Monitors zu implementieren werden aber nicht nur Locks sondern auch Synchronisationskonzepte wie wait/notify von einer Programmiersprache gefordert, welche durch den Scheduler implementiert werden.

Weiterführende Wissensquellen:

Amdahl's law. [7]

Gustafson's law. [8]

[7] https://en.wikipedia.org/wiki/Amdahl%27s_law (Wikipedia, Amdahl's law, 2021)

[8] https://en.wikipedia.org/wiki/Gustafson%27s_law (Wikipedia, Gustafson's law, 2021)

5.3.1 Synchronisierte Methoden

Um kritische Methoden in Java zu markieren kann das synchronized Keyword in die Methodendefinition geschrieben werden. Bei jedem Methodenaufruf wird auf das Lock des aktuellen Objekts gewartet und dieses bis zum Verlassen der Methode genommen. Dies bedeutet, dass keine dieser synchronisierten Methoden, egal ob dieselbe oder eine andere desselben Objekts, von verschiedenen Threads gleichzeitig ausgeführt werden kann.

```
1   public final class SynchronizedMethodsCounter implements Counter {
2       private volatile int i;
3       public SynchronizedMethodsCounter() {
4           i = 0;
5       }
6       @Override public synchronized void increment() {
7           ++i;
8       }
12      @Override public synchronized int get() {
13          return i;
14      }
15  }
```

Codebeschrieb:

Zeile 2: Ein uncachebarer int wird erstellt. Uncachebar da mit einem Cache auch Locks nichts helfen. Dies weil nicht sichergestellt wird, dass der Wert der Variable nicht immer zwischen allen Threads aktuell gehalten wird.

Zeile 6, 12: Hier wird der Lock auf das aktuelle Objekt genommen

Zeile 7, 13: Hier befinden wir uns in der kritischen Region. Es kann nie mehr als ein Thread auf einer oder mehreren dieser Zeilen sein.

Zeile 8, 14: Hier wird das Lock auf das aktuelle Objekt automatisch wieder freigegeben.

Achtung: Ein Lock über alle synchronized Methoden hat oft einen grossen Einfluss auf den anteilig parallelisierbaren Programmanteil, was nach Amdahl's law einen negativen Einfluss auf die Skalierbarkeit und Performance hat.

5.3.2 Synchronisiertes this

Dieser Code ist sehr ähnlich zu dem im vorherigen Kapitel *Synchronisierte Methoden* beschriebenen Beispiel. Anders, als im vorherigen Beispiel wird jetzt synchronized(this)

verwendet. Dadurch wird das Lock des aktuellen Objekts nur für die Dauer dieses synchronized Blocks genommen. Dies hilft den Anteil nebenläufigen Codes zu erhöhen, da die kritische Region dadurch verkleinert werden kann.

```
1   public final class SynchronizedThisCounter implements Counter {
2       private volatile int i;
3       public SynchronizedThisCounter() {
4           i = 0;
5       }
6       @Override public void increment() {
7           synchronized(this) {
8               ++i;
9           }
10      }
16      @Override public int get() {
17          synchronized(this) {
18              return i;
19          }
20      }
21  }
```

Codebeschrieb:

Zeile 2: Ein uncachebarer int wird erstellt.

Zeile 7, 17: Hier wird das Lock auf das aktuelle Objekt genommen

Zeile 8, 18: Hier befinden wir uns in der kritischen Region. Es kann nie mehr als ein Thread auf einer oder mehreren dieser Zeilen sein.

Zeile 9, 19: Hier wird das Lock auf das aktuelle Objekt automatisch wieder freigegeben.

Achtung: Es muss darauf geachtet werden, dass die synchronized(this) Blöcke möglichst kurz sind, um den prozentual parallelisierbaren Code möglichst hoch zu halten.

5.3.3 Synchronisationsobjekte

Anders als bei *Synchronisierte Methoden* und *Synchronisiertes this* wird hier nicht das Lock des eigenen Objekts genommen, sondern ein eigenständiges Objekt für das Lock erstellt. Dies ist in der Regel übersichtlicher, da man so das Lock-Objekt sinnvoll benennen kann. Auch ist dies die einzige Option, falls man in einer Klasse mehr als nur ein Lock braucht.

```
1   public final class SynchronizedObjectCounter implements Counter {
2       private volatile int i;
3       private Object lock = new Object();
4       public SynchronizedObjectCounter() {
5           i = 0;
6       }
7       @Override public void increment() {
8           synchronized(lock) {
9               ++i;
10          }
11      }
17      @Override public int get() {
18          synchronized(lock) {
19              return i;
20          }
21      }
22  }
```

Codebeschrieb:

Zeile 2: Ein uncachebarer int wird erstellt.

Zeile 3: Ein neues, leeres Objekt wird erstellt um als Lock benutzt zu werden.

Zeile 8, 18: Hier wird das Lock des Lock-Objekts genommen.

Zeile 9, 19: Hier befinden wir uns in der kritischen Region. Es kann nie mehr als ein Thread auf einer oder mehreren dieser Zeilen sein.

Zeile 10, 20: Hier wird das Lock des Lock-Objekts automatisch wieder freigegeben.

Achtung: Es muss darauf geachtet werden, dass die synchronized(lock) Blöcke möglichst kurz sind um den prozentual parallelisierbaren Code möglichst hoch zu behalten.

Hinweis: Es sollte den Lock Objekten sinnvolle Namen gegeben werden und nicht an Lock-Objekten gespart werden, um gut lesbaren Code zu schreiben. Niemals sollte dasselbe Lock-Objekt für unterschiedliche Zwecke wiederverwendet werden, da dies beinahe immer zu Fehlern führt, die in Deadlocks enden.

5.4 Software Transactional Memory

Die Begriffserklärung und die Beschreibung des STM erfolgte im Kapitel 2.

5.4.1 Multiverse STM

Multiverse ist die wahrscheinlich bekannteste und auch schnellste Software Transactional Memory Implementierung für Java. Leider ist sie mittlerweile etwas in die Jahre gekommen. Der letzte Release war am 7. April 2012 und auch auf GitHub tut sich seit 9 Jahren nichts mehr. Grund für den Entwicklungsstopp waren fundamentale Probleme mit starvation von längeren Transaktionen durch kürzere Transaktionen, welche die längeren durch eine race condition immer zu einem Rollback zwingen. Dennoch funktioniert Multiverse auch noch mit Java 16, der momentan neusten Java Version, bestens. Das Multiverse STM wird in vielen älteren Applikationen verwendet und ist bis heute am schnellsten. Der Release des letzten Multiverse Updates war weit vor dem Release von Project Lambda, dennoch funktioniert Lambda problemlos mit Multiverse. (Veentjer, 2013)

```
1   public final class TransactionalCounterMultiverse implements Counter {
2       private final TxnInteger i;
3       public TransactionalCounterMultiverse() {
4           i = newTxnInteger(0);
5       }
6       @Override public void increment() {
7           atomic(() -> {
8               i.increment();
9           });
10      }
16      @Override public int get() {
17          return atomic(() -> {
18              return i.get();
19          });
20      }
21  }
```

Codebeschrieb:

Zeile 2, 4: Ein transaktioneller Integer wird erstellt und initialisiert.

Zeile 7, 17: Beginn einer Transaktion. Ab hier wird alles transaktionell ausgeführt. Bei einer race condition wird genau der Zustand wo der transaktionelle Block betreten wurde, wiederhergestellt.

Zeile 8, 18: Hier werden Operationen auf dem transaktionellen Integer ausgeführt Dies geschieht intern auf einem Weg, wo alles im Fall einer race condition rückgängig gemacht werden kann. In einem transaktionellen Block dürfen nur transaktionelle Variablen verwendet werden und keine anderen zustandsändernden Operationen, wie beispielsweise IO Operationen, durchgeführt werden.

Zeile 9, 19: Hier endet der transaktionelle Block. Ist bis anhin keine race condition aufgetreten wird hier die Transaktion irreversibel auf alle transaktionellen Variablen in diesem Block angewendet.

5.4.2 Scala STM

Scala STM ist eine von Haskell und Clojure inspirierte Software Transaction Memory Bibliothek, welche für Scala entwickelt wurde. Scala STM wurde in Scala geschrieben. Glücklicherweise ist es dennoch möglich diese in Java zu nutzen. Scala STM ist bis heute in aktiver Entwicklung. Der letzte Release war am 2. April 2021 und somit weniger als ein Monat alt. Scala STM ist mächtiger als Multiverse und nicht an vorgegebene, transaktionelle Datentypen gebunden. Es können selbst transaktionelle Datentypen erstellt werden. Scala STM unterstützt mehrere Software Transactional Memory Implementierungen. Unerwartete Fehler führen, wie erwartet, zu einem Rollback und einem Rethrow. Während bei behandelten Exceptions durchaus auch ein Commit und Rethrow stattfinden kann. Wie auch Multiverse ist Scala STM sehr einfach zu lernen und anzuwenden. Scala STM ist zwar etwas langsamer als Multiverse, aber sie ist bei weitem die bessere Bibliothek. Durch die Verwendung von Scala STM in der Programmierung spart man sich sehr viel Zeit, ohne dass es ein grosser Geschwindigkeitsverlust in der Programmausführung gibt, wie man in den nachfolgenden Diagrammen sehen kann.

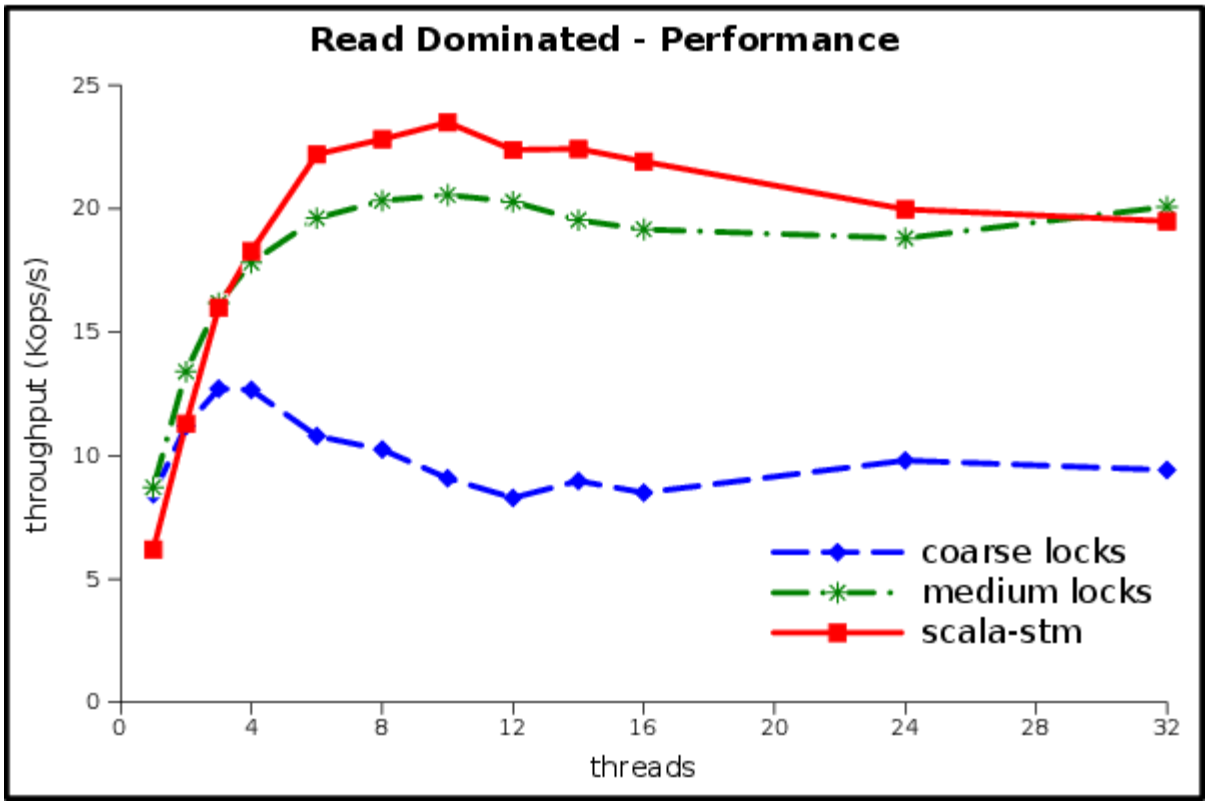

Abbildung 2: Schreibgeschwindigkeit von Scala STM

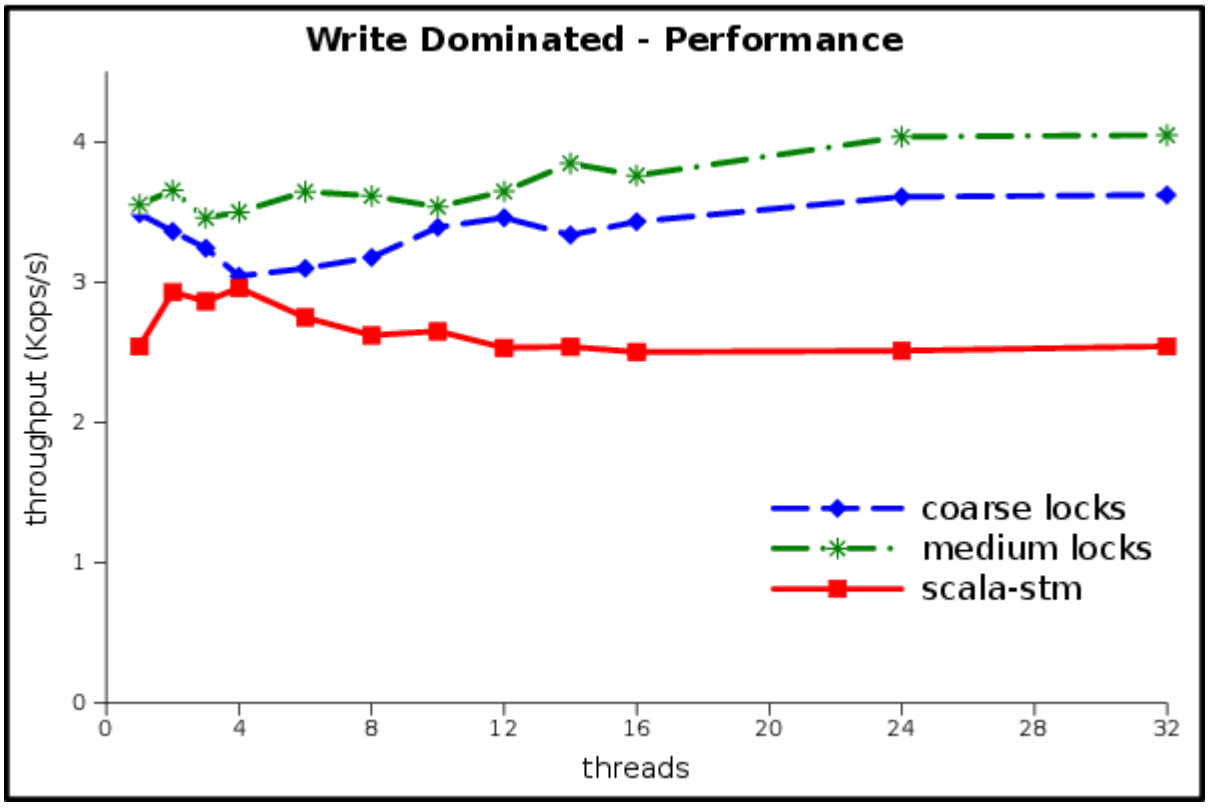

Abbildung 3: Lesegeschwindigkeit von Scala STM
Quelle: https://nbronson.github.io/scala-stm/benchmark.html

Weiterführende Wissensquellen:

Scala-STM. [9]

Scala-STM, GitHub. [10]

Scala-STM, benchmark. [11]

```
1   public final class TransactionalCounterScala implements Counter {
2       private Ref.View<Integer> i = newRef(0);
3       public TransactionalCounterScala() {
4           i = newRef(0);
5       }
6       @Override public void increment() {
7           atomic(() -> {
8               i.set(i.get() + 1);
9           });
10      }
17      @Override public int get() {
18          return i.get();
19      }
20  }
```

Codebeschrieb:

Zeile 2, 4: Ein transaktioneller Integer wird erstellt und initialisiert.

Zeile 7: Beginn einer Transaktion. Ab hier wird alles transaktionell ausgeführt. Bei einer race condition wird genau der Zustand wo der transaktionelle Block betreten wurde, wiederhergestellt.

Zeile 8: Hier werden Operationen auf dem transaktionellen Integer ausgeführt.

Zeile 9: Hier endet der transaktionelle Block. Ist bis anhin keine race condition aufgetreten wird hier die Transaktion irreversibel auf alle transaktionellen Variablen in diesem Block angewendet.

Zeile 18: Um zu lesen ist bei ScalaSTM keine Transaktion erforderlich.

[9] https://nbronson.github.io/scala-stm/index.html (Scala STM Expert Group, SCALA SMT, 2017)

[10] https://github.com/scala-stm/scala-stm/ (Bronson, 2021)

[11] https://nbronson.github.io/scala-stm/benchmark.html (Scala STM Expert Group, Benchmarking, 2017)

5.4.3 Narayana STM

Narayana STM ist die Software Transactional Memory Bibliothek für Profis. Es ist bei weitem die mächtigste, aber auch komplizierteste STM Bibliothek für Java. Dies ist keine Bibliothek für Hobbyprogrammierer, sondern fokussiert sich auf Enterprise Kunden und ist dementsprechend gut getestet. Red Hat bietet professionellen Support und Training. Die Bibliothek ist absolut riesig. Es werden keine transaktionellen Objekte bereitgestellt, sondern sie müssen selbst implementiert werden. Es wird sowohl optimistischer wie auch pessimistischer STM unterstützt. Atomic actions sind eigene Objekte und können viel mehr als nur Transaktionen zu beginnen oder beenden. Ein Rollback kann explizit angefordert werden. Auch kann die atomare Aktion pausiert und fortgesetzt werden. Transaktionen können beliebig ineinander verschachtelt werden. Der einzige Nachteil von Narayana ist, dass durch seine Komplexität auch die Performance schlechter als bei ScalaSTM ist. Dies spielt jedoch in Realität oft keine Rolle, ausser man hat Millionen von Transaktionen pro Sekunde. Ich empfehle Narayana STM für jedes grössere Java Projekt zu nutzen. Die Zeit, die man sich spart, indem man sich nicht um Locks kümmern muss, ist der Mehraufwand wert. Auf der offiziellen Webseite wird Narayana wie folgt beworben:

"With over 30 years of expertise in the area of transaction processing, Narayana is the premier open source transaction manager." (Red Hat, 2021)

"The Narayana transaction suite protects businesses from data corruption by guaranteeing complete, accurate business transactions for Java based applications" (Red Hat, 2021)

Weiterführende Wissensquelle:

Narayana, premier open source transaction manager. [12]

```
1  public final class TransactionalCounterNarayana implements Counter {
2      private Atomic atomicObj;
3      public TransactionalCounterNarayana() {
4          atomicObj = new Container<Atomic>().create(new NarayanaCounter());
5      }
6      @Override public void increment() {
7          AtomicAction atomicAction = new AtomicAction();
8          atomicAction.begin();
9          atomicObj.increment();
```

[12] https://narayana.io/index.html (Red Hat, 2021)

```
10          atomicAction.commit();
11      }
18      @Override public int get() {
19          return atomicObj.get();
20      }
21      @Transactional @Pessimistic
22      public interface Atomic
23      {
24          public void increment();
25          public void decrement();
26          public int get();
27      }
28
29      public class NarayanaCounter implements Atomic {
30          private int state;
31          @Override @ReadLock
32          public int get() {
33              return state;
34          }
35          @Override @ReadLock @WriteLock
36          public void increment() {
37              ++state;
38          }
43      }
44  }
```

Codebeschrieb:

Zeile 8: Beginn einer Transaktion. Ab hier wird alles transaktionell ausgeführt. Bei einer race condition wird genau der Zustand vor der Transaktion wieder hergestellt.

Zeile 9: Hier werden Operationen auf dem transaktionellen Integer ausgeführt.

Zeile 10: Ist bis anhin keine race condition aufgetreten, wird hier die Transaktion irreversibel auf alle transaktionellen Variablen in diesem Block angewendet

Zeile 19: Um zu lesen ist bei Narayana STM keine Transaktion erforderlich

Zeile 21 bis 27: Spezifizieren des Interfaces des transactional Integers. Durch @Transaction wird spezifiziert, dass es sich um einen transaktionellen Datentyp handelt und durch @Pessimistic wird spezifiziert, dass viele race conditions zu erwarten sind. Normalerweise gehen Software transactional Memory Implementierungen von einem eher optimistischen Ergebnis aus.

Zeile 29 bis 43: Implementieren eines transaktuellen Integers. Durch @ReadLock und @WriteLock wird spezifiziert wie diese Methode des transaktionellen Objekts auf die Daten zugreift.

5.5 Channels

Ein essentielles nebenläufiges Kontrollkonzept um zwischen verschiedenen Threads zu kommunizieren sind Concurrent Queues, Channels und Messages. Diese Datenstrukturen dienen als Grundbausteine und Ergänzung einer Vielzahl nebenläufiger Parallelitätskonzepte, beispielsweise bei ThreadPools Continuations, Coroutinen und Actor Concurrancy. Dies sind FIFO-Listen auf welche mehrere Threads gleichzeitig zugreifen können. Man hat also einer oder mehrere Produzenten welche Werte in diesen Channel schreiben und einen oder mehrere Konsumenten welche Werte aus diesem Channel lesen und konsumieren. Channels sind eine Spezialisierung von Concurrent Queues welche genau für dieses parallele Producer/Consumer Pattern optimiert wurden und in der Regel auch besseren Syntax bereitstellen. Alles was man durch Channels machen kann, ist auch durch Concurrent Queues möglich, jedoch nicht zwingend umgekehrt. Eine Message ist recht ähnlich wie ein Channel jedoch ist diese immer unbounded, also ein Channel ohne die Möglichkeit eine Kapazitätsgrenze zu definieren. So werden Deadlocks verunmöglicht. Jedoch ist deswegen bei Messages oft Flow Control nötig, da diese ansonsten beim Produzenten niemals blockieren würden. Ohne Flow Control ist es bei Messages möglich, dass der Produzent den Konsumenten mit Nachrichten überflutet, bis schliesslich der Arbeitsspeicher irgendwann voll wird. Ist bekannt, dass der Konsument immer schneller ist als der Produzent, so kann auch auf Flow Control verzichtet werden da sich so nie Nachrichten aufstauen können. Es gibt viele Möglichkeiten solche Flow Control Systeme zu implementieren. Beispielsweise kann der Konsument dem Produzenten über eine Message metteilen, wenn er mit Nachrichten überflutet wird. Eine weitere Möglichkeit ist, dies durch Guards zu implementieren wo beispielsweise eine Semaphore genutzt wird und der Produzent nur so lange Nachrichten senden darf bis die Semaphore auf null ist. Es muss jedoch beachtet werden, dass je nach Flow Control Mechanismus hier wieder die Möglichkeit von Deadlocks gegeben sein kann, was ein grosser Vorteil der Actor Concurrancy zunichtemachen würde. Somit muss darauf geachtet werden, dass durch den Flow Control keine Deadlocks entstehen können. Anders als Queues können Messages auch über das Internet versendet werden. Dies ist weshalb Actor Concurancy zum Erstellen von verteilten Systemen verwendet werden kann. Dies zeigen auch andere für verteile Systeme weit verbreitete Bibliotheken wie Message Passing Interface oder ZeroMQ.

Weiterführende Wissensquelle:

An Introduction to System.Threading.Channels. [13]

What's the Difference between Channel and ConcurrentQueue in C#? [14]

Message Passing Interface. [15]

ZeroMQ. [16]

[13] https://devblogs.microsoft.com/dotnet/an-introduction-to-system-threading-channels/ (Toub, 2019)

[14] https://jeremybytes.blogspot.com/2021/02/whats-difference-between-channel-and.html (Sedro-Woolley, 2021)

[15] https://de.wikipedia.org/wiki/Message_Passing_Interface (Wikipedia, Message Passing Interface, 2019)

[16] https://zeromq.org/get-started/ (ZeroMQ, 2021)

6 Nebenläufige Programmiermodelle

In diesem Kapitel werden die verschiedenen Code-Beispiele der Programmiersprachen betreffend Nebenläufigkeit aufgezeigt, kommentiert und die dazugehörenden Bibliotheken beschrieben. Des Weiteren werden die Nebenläufigkeitskonzepte erläutert und gleichsprachliche Beispiele werden miteinander verglichen.

6.1 Java

Java ist eine bekannte objektorientierte, imperative Programmiersprache, welche in einer virtuellen Maschine (JVM) ausgeführt wird. Die Abfallsammlung übernimmt Java selbst. Java wird nicht direkt in Maschinensprache, sondern in einer Zwischensprache kompiliert. Diese wird in der JVM ausgeführt. Diese Programmiersprache wird in dieser Arbeit als Referenzsprache verwendet.

6.1.1 Asynchroner HTTP-Request

6.1.1.1 Thread Pools

Durch ein Thread Pool werden mehrere OS Threads in einem Pool zusammengefasst. Dadurch ist es möglich diese für mehrere Aufgaben wiederzuverwenden und so den Overhead vom Erstellen und Löschen von OS Threads zu vermeiden. In Java bietet ein ThreadPoolExecutor dem Programmierer eine Abstraktion über das manuelle Erstellen und Verwalten von Threads. Durch einen CompleationService kann auf die von den Tasks zurückgegebenen Werte zugegriffen werden. Es gibt sehr viele Arten von Thread Pools in Java wobei sich diese hauptsächlich durch das Thread Caching Verhalten unterscheiden. Dazu mehr in den folgenden Unterkapiteln.

Thread Pools sind ein relativ altes und in vielen Programmiersprachen genutztes Konzept. Durch Sprachunterstützung von Fibers sind Thread Pools nicht mehr nötig und werden bei Project Loom nur zur Rückwärtskompatibilität in dieser Art verwendet. Dies weil durch Fibers keine OS Threads mehr erstellt werden und somit Threads beinahe keinen Overhead mehr haben. Die Implementierung von Fibers in eine Programmiersprache ist oft ein in die Sprache integriertes globales Thread Pool welches durch Fibers abstrahiert wird.

Falls man sich für die Implementierung eines Thread Pools interessiert, ist in der nachfolgenden Wissensquelle eine sehr einfache und weit verbreitete C++ Implementierung mit weniger als 100 Zeilen verlinkt.

Weiterführende Wissensquelle:

ThreadPool. [17]

6.1.1.1.1 Fixed ThreadPool

Die einfachste Art eines Thread Pools ist das fixe Thread Pool. Dabei werden bei der Erstellung des Thread Pools im Hintergrund eine spezifizierte Anzahl Threads erstellt, welche bis zum Löschen des Thread Pools aktiv gelassen werden. Ein Nachteil davon ist, dass so oftmals mehr Threads als benötigt laufen was eine Ressourcenverschwendung ist. Auch besteht die Gefahr mehr Threads als notwendig zu erstellen und so unnötigen Overhead zu erzeugen oder die Anzahl Threads auf aktuelle Hardware zu beschränken und so seine Anwendung weniger skalierbar zu machen. Dies sollte nur verwendet werden, wenn die Anzahl notwendiger Threads ganz klar ist. Ist es nicht offensichtlich wie viele Threads benötigt werden oder gibt es Zeiten, wo viel Threads über längere Zeit ungenutzt sind, sollte stattdessen ein Cached Thread Pool verwendet werden, sodass sich der Programmierer nicht Gedanken darüber machen muss. Wobei auch dieses alles andere als perfekt ist.

```
1   public static void run() {
2       ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(4);
3       CompletionService<String> completionService = new ExecutorCompletionService<String>(executor);
4       for (int i = 0; i < 10; ++i) {
5           completionService.submit(() -> {
7               return new BufferedReader(new InputStreamReader(new URL("http://www.nicobosshard.ch/Hi.html")
8                           .openStream())).lines().collect(Collectors.joining("\n"));
9           });
10      }
12      BigInteger.probablePrime(256, new Random());
14      for (int i = 0; i < 10; ++i) {
16          System.out.println(completionService.take().get());
20      }
21  }
```

[17] https://github.com/progschj/ThreadPool/blob/master/ThreadPool.h (Progsch, 2014)

Codebeschrieb:

Zeile 2: Ein ThreadPoolExecutor mit einem FixedThreadPool welches immer 4 Threads enthält wird erstellt.

Zeile 3: Es wird ein CompletionService, welcher Strings als Rückgabewerte akzeptiert, auf dem zuvor erstellten ThreadPoolExecutor erstellt.

Zeile 4 bis 9: Es werden 10 Tasks erstellt und über den CompletionService ausgeführt. Diese laden asynchron HTML-Daten herunterladen und speichern das Ergebnis im CompletionService.

Zeile 12: Es wird eine 256-Bit Primzahl mit berechnet um Arbeit auf dem Main Thread zu simulieren.

Zeile 14 bis 20: Es wird über alle dem CompletionService gesendeten Tasks iteriert und deren Resultat ausgegeben. Ist ein Task noch nicht abgeschlossen wird auf ihn gewartet.

6.1.1.1.2 Cached ThreadPool

Ein gecachetes Thread Pool beinhaltet immer nur so viele Threads wie nötig. Gestartet wird mit null Threads und es existiert kein Limit wie viele Threads laufen können. Wird ein weiterer Thread benötigt und keiner wartet auf Arbeit so wird ein neuer Thread erstellt. Wird ein Thread über längere Zeit nicht mehr genutzt wird dieser gelöscht. Dadurch muss sich der Programmierer nie mehr Gedanken darüber machen wie viele Threads er überhaupt braucht. Leider ist ein gecachtes Thread Pool auch sehr gefährlich da keine maximale Anzahl von Threads angegeben werden kann. Die einzige Möglichkeit Threads zu beschränken ist auf Ausführungsebene. Dies geschieht beispielsweise durch ein Thread Pool Executor. Dabei ist es jedoch sehr einfach Fehler zu machen was massive negative Performanceauswirkungen auf das Programm haben kann. Auch wird das Programm dadurch anfällig gegenüber Denial of Service Attacken. Dies da auch falls der CPU schon voll ausgelastet ist weitere Threads erstellt werden und somit alle Threads länger und länger dauern, bis es zu einem beinahe kompletten Stillstand kommt. Deswegen ist oftmals ein Fixed Thread Pool mit der Grösse der Anzahl realer CPU Threads die bessere Option, wobei dann auch wieder Ressourcen verschwendet werden. Schlussendlich sind klassische Thread Pools in Java einfach nicht eine gute Idee und sollten,

sobald Project Loom veröffentlicht wird mit einem Thread Pool welches Fibers als Backend nutzt, ersetzt werden.

Weiterführende Wissensquelle:

CachedThreadPool. [18]

```
1   public static void run() {
2       ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newCachedThreadPool();
3       CompletionService<String> completionService = new ExecutorCompletionService<String>(executor);
4       for (int i = 0; i < 10; ++i) {
5           completionService.submit(() -> {
7               return new BufferedReader(new InputStreamReader(new URL("http://www.nicobosshard.ch/Hi.html")
                .openStream())).lines().collect(Collectors.joining("\n"));
8           });
9       }
11      BigInteger.probablePrime(256, new Random());
13      for (int i = 0; i < 10; ++i) {
15          System.out.println(completionService.take().get());
19      }
20  }
```

Codebeschrieb:

Zeile 2: Ein ThreadPoolExecutor mit einem CachedThreadPool erstellt. Dieses beinhaltet immer so viele Threads wie nötig. Ungenutzte Threads werden nach einer bestimmten Zeit gelöscht.

Zeile 3: Es wird ein CompletionService, welcher Strings als Rückgabewerte akzeptiert, auf dem zuvor erstellten ThreadPoolExecutor erstellt.

Zeile 4 bis 20: Siehe Codebeispiel Fixed ThreadPool

[18] http://dev.bizo.com/2014/06/cached-thread-pool-considered-harmlful.html (Bizo dev team, 2014)

6.1.1.1.3 ForkJoinPool

Das ForkJoinPool wurde in Java 7 eingeführt und dient zum Parallelisieren von Teile- und Herrschealgorithmen. Anders als andere ThreadPools nutzt dieses einen Algorithmus bei dem jeder freie Thread versucht anderen Threads Arbeit abzunehmen. Dies geschieht so, dass die aufwändigen Tasks priorisiert werden. Dies funktioniert da bei einem Teile -und Herrsche-Algorithmus die Tasks in der Regel immer kleiner und somit weniger aufwändig werden, je später in den Pool submitted.

Ein gutes Bespiel für einen solchen Algorithmus wäre bitonisches Sortieren – ein beliebter paralleler Sortieralgorithmus welcher in worst case O(n(log^2(n)) paralleler Laufzeit läuft und somit wesentlich schneller als worst case O(n log n) von klassischen sequenziellen Sortieralgorithmen ist.

Weiterführende Wissensquelle:

Bitonic sorter, wikipedia. [19]

Ein ForkJoinPool funktioniert sehr gut für Teile -und Herrschealgorithem bei denen keine race condition der Teile auf gleicher Stufe auftreten, sollte aber anders als im unterstehenden Beispiel nicht für normale Algorithmen verwendet werden. Auch ist es relativ komplex ein Teile -und Herrschealgorithmus mit einem ForkJoinPool korrekt zu parallelisieren und sobald Project Loom erscheint, sollte auch bei Teile -und Herrschealgorithmen auf Fibers gewechselt werden.

Weiterführende Wissensquelle:

Guide to the Fork/Join Framework in Java. [20]

[19] https://en.wikipedia.org/wiki/Bitonic_sorter (Wikipedia, Bitonic sorter., 2021)

[20] https://www.baeldung.com/java-fork-join (Baeldung, 2020)

```
1   public static void run() {
2       ForkJoinPool executor = new ForkJoinPool(4);
3       CompletionService<String> completionService = new ExecutorCompletionService<String>(executor);
4       for (int i = 0; i < 10; ++i) {
5           completionService.submit(() -> {
7               return new BufferedReader(new InputStreamReader(new URL
                ("http://www.nicobosshard.ch/Hi.html").openStream()))
               .lines().collect(Collectors.joining("\n"));
8           });
9       }
11      BigInteger.probablePrime(256, new Random());
13      for (int i = 0; i < 10; ++i) {
15          System.out.println(completionService.take().get());
19      }
20  }
```

Codebeschrieb:

Zeile 2: Ein ForkJoinPool welches immer 4 Threads enthält wird erstellt.

Zeile 3: Es wird ein CompletionService, welcher Strings als Rückgabewerte akzeptiert, auf dem zuvor erstellten ForkJoinPool erstellt.

Zeile 4 bis 20: Siehe Codebeispiel Fixed ThreadPool

6.1.1.1.4 SingleThreadExecutor

Der SingleThreadExecutor erstellt ein ThreadPool mit einem Thread. Er ist äquivalent zu newFixedThreadPool(1), ausser dass man ihn nicht konfigurieren kann mehr als ein Task zu nutzen. Er garantiert Tasks in derselben Reihenfolge wie man sie submitted abzuarbeiten. Es wird empfohlen anstelle eines newSingleThreadExecutor() ein newFixedThreadPool(1) zu verwenden.

Achtung: Es kann nicht davon ausgegangen werden, dass immer derselbe Thread verwendet wird, weil falls ein Thread crasht ein neuer für den nächsten Task erstellt wird.

Weiterführende Wissensquelle:

Interface ExecutorService. [21]

[21] https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ExecutorService.html (Oracle, Interface ExecutorService, 2015)

```
1   public static void run() {
2       ExecutorService executor = Executors.newSingleThreadExecutor();
3       CompletionService<String> completionService = new ExecutorCompletionService<String>(executor);
4       for (int i = 0; i < 10; ++i) {
5           completionService.submit(() -> {
7               return new BufferedReader(new InputStreamReader(new URL("http://www.nicobosshard.ch/Hi.html")
                .openStream())).lines().collect(Collectors.joining("\n"));
8           });
9       }
11      BigInteger.probablePrime(256, new Random());
13      for (int i = 0; i < 10; ++i) {
15          System.out.println(completionService.take().get());
19      }
20  }
```

Codebeschrieb:

Zeile 2: Ein ExecutorService eines SingleThreadExecutor wird erstellt.

Zeile 3: Es wird ein CompletionService, welcher Strings als Rückgabewerte akzeptiert, auf dem zuvor erstellten ExecutorService erstellt.

Zeile 4 bis 20: Siehe Codebeispiel Fixed ThreadPool

6.1.1.1.5 SingleThreadScheduledExecutor

Der SingleThreadScheduledExecutor ist ein spezieller Thread Pool welcher ein gegebener Task zu bestimmten Zeiten ausführen kann. Beispielsweise kann damit ein Task generiert werden welcher alle Sekunden ausgeführt werden soll. Dies wird nicht durch Warten im Thread, sondern zum Erstellungszeitpunkt des Thread Pools definiert. Anstelle dem häufig verwendeten *scheduleAtFixedRate* kann auch *scheduleWithFixedDelay* verwendet werden. Dabei wird, keine *java.util.concurrent.RejectedExecutionException* ausgelöst, falls ein Task länger geht als der Abstand zwischen den Tasks. Jedoch hat dies den Nachteil, dass die Dauer des Tasks nicht mitgezählt wird und somit die Tasks nicht mehr regelmässig ausgeführt werden, da nur der Abstand zwischen Tasks gegeben ist. Solche Exceptions können auch über ThreadPoolExecutor.DiscardPolicy unterdrückt werden.

SingleThreadScheduledExecutor ist eine elegante Lösung für eine oft auftretende Anforderung welche jedoch leider oft vergessen geht.

Weiterführende Wissensquelle:

Interface ExecutorService. [22]

```
1   static BlockingQueue<String> blockingQueue = new LinkedBlockingDeque<>();
2   public static void run() throws InterruptedException {
3       ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor();
4       final ScheduledFuture<?> promise = executor.scheduleWithFixedDelay(() -> {
7           blockingQueue.add(new BufferedReader(new InputStreamReader(new URL("http://www.nicobosshard.ch/Hi.html")
            .openStream())).lines().collect(Collectors.joining("\n")));
11      }, 1, 1, TimeUnit.SECONDS);
12      executor.schedule(() -> promise.cancel(false), 1, TimeUnit.MINUTES);
13      for (int i = 0; i < 10; ++i) {
15          BigInteger.probablePrime(256, new Random());
17          System.out.println(blockingQueue.take());
18      }
20  }
```

Codebeschrieb:

Zeile 1: Erstellt eine neue blockingQueue für die Interthreadkommunikation

Zeile 3: Ein neuer ScheduledExecutorService wird erstellt

Zeile 4 bis 11: Erstellt einen Task welcher alle Sekunden HTML-Daten herunterlädt und der blockingQueue hinzufügt

Zeile 13: Nach einer Minute soll der ScheduledExecutorService gestoppt werden.

Zeile 13 bis 18: Der Main Thread generiert 10-mal eine 256-Bit grosse Primzahl um Arbeit zu simulieren und nimmt danach ein Element aus der blockingQueue. Ist die blockingQueue leer so wird gewartet.

6.1.1.2 AkkaHTTP

AkkaHTTP ist eine auf Java Akka und Java Akka Streams basierende Java Bibliothek. AkkaHTTP ermöglicht es sowohl HTTP Server wie auch HTTP Clients mit APIs von unterschiedlichen Abstraktionslevels zu implementieren. Diese reichen von lowlevel über request level und connection level bis host level. Dabei wird alles über Aktor Parallelität realisiert. Dadurch ist es möglich, sehr grosse Server Systeme mit niedriger Komplexität zu entwickeln. AkkaHttp ist

[22] https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ExecutorService.html (Oracle, Interface ExecutorService, 2015)

riesig und bietet Dinge wie DSL-Routing, Caching, Server-sent Events, DNS, JSON, XML, Komprimierung, WebSocket, Multipart, HTTPS, HTTP/2 und vieles mehr. Ich werde mich in dieser Arbeit auf clienteseitiges HTTP mit der ConnectionLevel und HostLevel API beschränken.

6.1.1.2.1 ConnectionLevel

Auf der ConnectionLevel Abstraktionsstufe wird pro HTTP Verbindung ein Aktor erstellt. Dieser kann für alle Requests und Responses dieser Verbindung wiederverwendet werden. Die Connection Level API bietet ein gutes Mittelmass zwischen Abstraktion und Verständlichkeit. Auch ist es die neben der Host Level API eine der wahrscheinlich weit verbreitetsten.

```
1  public static void main(String[] args) {
2      ActorSystem actorSystem = ActorSystem.create("client");
3      AkkaHTTPDemo client = new AkkaHTTPDemo(actorSystem);
4      client.connectionLevel(Optional.of("Hi_connectionLevel"), success ->
5              success.entity()
6                      .getDataBytes()
7                      .runForeach(byteString -> System.out.println("[ConnectionLevel]: "
                       + byteString.utf8String()),
8                              client.getSystem())
9      ).whenComplete((success, throwable) -> client.close());
10 }
11
12 public static class AkkaHTTPDemo {
13     private final ActorSystem system;
14     private final Flow<HttpRequest, HttpResponse, CompletionStage<OutgoingConnection>> connectionFlow;
15     public AkkaHTTPDemo(ActorSystem sys) {
16         system = sys;
17         connectionFlow = Http.get(system).outgoingConnection(ConnectHttp.toHost(
           "http://www.nicobosshard.ch/", 80));
18     }
22     public void close() {
24         Http.get(system).shutdownAllConnectionPools().whenComplete((s, f) -> system.terminate());
25     }
29     public <U> CompletionStage<Done> connectionLevel(Optional<String> s,
30             Function<HttpResponse, CompletionStage<Done>> responseHandler) {
31         return Source.single(HttpRequest.create().withUri(getUri(s))).via(connectionFlow)
32                 .runWith(Sink.head(), system).thenComposeAsync(responseHandler);
33     }
34 }
```

Codebeschrieb:

Zeile 2: Erstellt eine neue ActorSystem namens «client».

Zeile 3: Das zuvor erstellte ActorSystem wird nun verwendet um einen AkkaHTTPDemo Aktor zu erstellen.

Zeile 4 bis 8: Auf dem nun erstellten Aktor wird die AkkaHTTP connectionLevel API verwendet um alle Daten einer HTML Seite herunterzuladen und auszugeben.

Zeile 9: Das Aktor System wird nach der Aktion geschlossen. Egal ob diese erfolgreich war oder ein Fehler aufgetreten ist

Zeile 13 bis 18: Im Konstruktor des AkkaHTTPDemo Aktors wird das übergebene ActorSystem als Klassenvairable abgespeichert sowie ein neue HTTP-Verbindung aufgebaut. Diese kann für alle mit diesem Aktor durchgeführten Requests wiederverwendet werden weswegen dies ein ConnectionLevel Actor ist.

Zeile 22 bis 25: Um das AktorSystem korrekt schliessen zu können muss hier der Verbindungsabbau sowie das Terminieren des ActorSystems gehandelt werden. Durch in AttpAkka integrierten Funktionen ist dies jedoch recht einfach. Man ruft erst shutdownAllConnectionPools() auf und nutzt den whenComplete Event um terminate() aufzurufen.

Zeile 29 bis 33: Dieser Codeabschnitt führt den eigentlichen HTTP Request aus und sendet das Ergebnis dem responseHandler.

6.1.1.2.2 HostLevel

Auf der HostLevel Abstraktionsstufe wird pro Host ein Aktor erstellt. Dieser nutzt ein gecachtes Verbindungspool um mehrere HTTP Verbindungen zum gleichen Host zu verwalten. So muss sich der Programmierer nicht mehr um einzelne HTTP sondern nur noch um Hosts kümmern.

```
1   public static void main(String[] args) {
2       ActorSystem actorSystem = ActorSystem.create("client");
3       AkkaHTTPDemo client = new AkkaHTTPDemo(actorSystem);
4       client.hostLevel(Optional.of("Hi_hostLevel"), success ->
5               success.first().get().entity()
6                       .getDataBytes()
7                       .runFold(ByteString.emptyByteString(), ByteString::concat, client.getSystem())
8                       .handle((byteString, f) -> {
16      ).whenComplete((success, throwable) -> client.close());
17  }
18
19  public static class AkkaHTTPDemo {
20      private final ActorSystem system;
21      private final Flow<Pair<HttpRequest, Integer>, Pair<Try<HttpResponse>, Integer>,
        HostConnectionPool> poolClientFlow;
22      public AkkaHTTPDemo(ActorSystem sys) {
23          system = sys;
24          poolClientFlow = Http.get(system).cachedHostConnectionPool(ConnectHttp.toHost
            ("http://www.nicobosshard.ch/", 80));
25      }
36      public <U> CompletionStage<U> hostLevel(Optional<String> s,
37              Function<Pair<Try<HttpResponse>, Integer>, CompletionStage<U>> responseHandler) {
38          return Source.single(Pair.create(HttpRequest.create().withUri(getUri(s)), 42)).via
            (poolClientFlow)
39                  .runWith(Sink.head(), system).thenComposeAsync(responseHandler);
40      }
41  }
```

Da bei der HostLevel API vieles ähnlich wie bei der ConnectionLevel API ist, habe ich mich entschieden nur die Codezeilen zu beschrieben welche sich klar unterscheiden.

Codebeschrieb:

Zeile 4 bis 8: Anders als bei der ConnectionLevel API wird hier die HostLevel API auf den Aktor angewendet. Diese unterscheidet sich erheblich aber macht schlussendlich das gleiche.

Zeile 21 und 24: Anstelle einer Verbindung wird hier ein gecachtes Verbindungspool auf den Host www.nicobosshard.ch erstellt.

Zeile 36 bis 40: Anstelle der Verbindung wird auch hier das gecachte Verbindungspool für den eigentlichen HTTP Request genutzt.

6.1.1.3 Future

Um Futures in Java zu zeigen habe ich mich für die asyncHttpClient Bibliothek entschieden auf welche man auf einfachste Weise asynchrone HTTP Requests ausführen kann.

6.1.1.3.1 Einfaches Future

Um einfach anzufangen hier ein Beispiel eines einfachen asyncHttpClient Futures um HTML-Daten hernunterzuladen und auf den Main Thread zu übertragen.

```
1  public static String run() throws InterruptedException, ExecutionException {
2      Future<Response> blockingFuture = asyncHttpClient().prepareGet("http://www.nicobosshard.ch/Hi.html")
       .execute();
4      BigInteger.probablePrime(256, new Random());
6      return blockingFuture.get().getResponseBody();
7  }
```

Codebeschrieb:

Zeile 2: Ein neues Future wird erstellt und mit execute() gestartet. Seine Aufgabe ist das Herunterladen von HTML-Daten.

Zeile 4: Eine 256-Bit Primzahl wird berechnet um Arbeit des Main Threads zu simulieren.

Zeile 6: Durch .get() kann das Ergebnis des Futures genommen werden. Ist es noch nicht fertig wird gewartet. Durch die Methode getResponseBody() wird der HTML-Response extrahiert.

6.1.1.3.2 Completable Future

Die asyncHttpClient bietet nicht nur normale Futures, sondern auch CompletableFutures. Durch toCompletableFuture() liefert asyncHttpClient ein CompletableFuture auf welches alle von Java bereitgestellten CompletableFuture Methoden angewendet werden können. CompletableFuture erlaubt in Java asynchrone Streamverarbeitung und kann als Alternative zu async/await in anderen Programmiersprachen gesehen werden.

```
1   public static String run() {
2       var completableFuture = asyncHttpClient().prepareGet("http://www.nicobosshard.ch/Hi.html").execute()
3           .toCompletableFuture().whenComplete((result, ex) -> {
5           });
7       BigInteger.probablePrime(256, new Random());
9       return completableFuture.join().getResponseBody();
10  }
```

Codebeschrieb:

Zeile 2: Ein neues Future wird erstellt und mit execute() gestartet.

Zeile 3: Durch toCompletableFuture() wird das Future in ein CompletableFuture umgewandelt. Durch Standard CompletableFuture Java Befehle wie whenComplete kann das Resultat weiterverarbeitet und auf bestimmte Events reagiert werden.

Zeile 7: Eine 256-Bit Primzahl wird berechnet um Arbeit des Main Threads zu simulieren.

Zeile 9: Durch .join() kann das Ergebnis des Futures genommen werden. Ist es noch nicht fertig wird gewartet. Durch die Methode getResponseBody() wird der HTML-Response extrahiert.

6.1.1.3.3 Listenable Future

Die asyncHttpClient bietet nicht nur normale Futures, sondern auch ListenableFutures. Ein ListenableFuture ist ein Future, auf welches Listener hinzugefügt werden welche nach Abschluss des Download-Tasks auf einem bestimmten ExecutorService ausgeführt werden. Es gibt keine Einschränkungen wie viele Listeners man hinzufügen kann und auf welchen ExecutorServices man diese laufen lässt. Das ListenableFuture ist als Event basierende Alternative zu .toCompletableFuture().whenComplete() zu sehen.

```
1   public static String run() throws InterruptedException, ExecutionException {
2       ListenableFuture<Response> whenResponse = asyncHttpClient().prepareGet("http://www.nicobosshard.ch/Hi.html")
3               .execute();
4       ExecutorService executorService = Executors.newSingleThreadExecutor();
5       whenResponse.addListener(
6               () -> System.out.println("[ListenableFutureDemo] Listener: " + Thread.currentThread().getName()),
7               executorService);
9       BigInteger.probablePrime(256, new Random());
11      return whenResponse.get().getResponseBody();
12  }
```

Codebeschrieb:

Zeile 2 bis 3: Ein neues ListenableFuture wird erstellt und mit execute() gestartet.

Zeile 4: Ein ExecutorService wird erstellt auf welchem das Event ausgeführt wird.

Zeile 5 bis 7: Hinzufügen eines Listener welcher ausgeführt wird sobald das ListenableFuture fertig ist.

Zeile 9: Eine 256-Bit Primzahl wird berechnet um Arbeit des Main Threads zu simulieren.

Zeile 11: Auch von einem ListenableFuture kann durch .get()das Ergebnis des Futures genommen werden und durch die Methode getResponseBody() der HTML-Response extrahiert werden.

6.1.1.4 Asynchronous Handlers

Asynchronous Handlers ist ein Event basiertes Konzept um mit asynchronen Aufgaben umzugehen. Dabei kann anders als beim ListenableFuture nicht nur auf das Endergebnis subscribed werden, sondern auf jeden Teilschritt der asynchronen Aufgaben. Diese Events werden im gleichen asynchronen Thread wie der eigentliche asynchrone Task ausgeführt. Durch diese Events kann Einfluss auf den asynchronen Task genommen werden. Durch Setzen von Zuständen in Teilaufgaben kann Einfluss auf zukünftige Teilaufgaben genommen werden. Durch Rückgabeparameter kann nach jedem Event entschieden werden ob der asynchrone Task fortgesetzt oder abgebrochen werden soll. Asynchronous Handlers ist ein sehr mächtiges Konzept, wenn man auf eine asynchrone Aufgabe wie ein HTTP Request Einfluss nehmen will ohne gleich auf Reactive wechseln zu müssen. Während es recht ähnlich wie Reactive ist und die meisten seiner Vorteile abdeckt, kann hier direkt auf die Events subscribed werden und es muss kein eigenes Publisher/Subscriber-System aufgebaut werden. Dies minimiert die Komplexität und den benötigten Code gegenüber Reactive und ist die bessere Option falls nicht mehrere Subscriber auf die Events subscriben können müssen.

Die asyncHttpClient Bibliothek bietet mit AsyncHandler genau dies. Im nachfolgenden Beispiel wird gezeigt, wie auf die Events onStatusReceived, onHeadersReceived, onBodyPartReceived, onCompleted und onThrowable reagiert wird.

```
1  public static String run() throws InterruptedException, ExecutionException {
2      Future<String> asyncHandler = asyncHttpClient().prepareGet("http://www.nicobosshard.ch/Hi.html")
3          .execute(new AsyncHandler<String>() {
4              private Charset charset = StandardCharsets.UTF_8;
5              private StringBuilder htmlSource = new StringBuilder();
6              @Override public State onStatusReceived(HttpResponseStatus statusCode) throws Exception {
8                  return (statusCode.getStatusCode() == 200) ? State.CONTINUE : State.ABORT;
9              }
10             @Override public State onHeadersReceived(HttpHeaders headers) throws Exception {
12                 Charset specifiedCharset = HttpUtils
13                         .extractContentTypeCharsetAttribute(headers.get("Content-Type"));
14                 if (specifiedCharset != null) charset = specifiedCharset;
16                 return State.CONTINUE;
17             }
18             @Override public State onBodyPartReceived(HttpResponseBodyPart bodyPart) throws Exception {
20                 htmlSource.append(new String(bodyPart.getBodyPartBytes(), charset));
21                 return State.CONTINUE;
22             }
23             @Override public String onCompleted() throws Exception {
25                 return htmlSource.toString();
26             }
31         });
33     BigInteger.probablePrime(256, new Random());
35     return asyncHandler.get();
36 }
```

Codebeschrieb:

Zeile 2 bis 3: Ein neuer Future<String> wird erstellt und mit execute gestartet. Anders als bei den vorhergehenden Beispielen wird hier jedoch dem execute ein AsyncHandler<String> mitgegeben weswegen diesmal auch ein Future<String> anstelle eines Futures<Response> verwendet werden kann.

Zeile 4 bis 5: Durch Variablen wie «charset» und «htmlSource» können eventübergreifende Zustände gespeichert werden. Da alle Events im selben Thread laufen wird es zwischen diesen nicht zu race conditions kommen. Dies ist auch das einzige Beispiel welches verschiedene Charsets korrekt behandelt, was in vielen andern parallelen Programmiermodellen nur schwer zu machen ist.

Zeile 6 bis 9: Durch das Event onStatusReceived kann der HTTP Response Status geprüft werden. In diesem Beispiel nutzen wir den Status Code um die Ausführung der asynchronen Aufgabe bei einem Status Code von 200 (OK) durch die Rückgabe von State.CONTINUE fortzusetzen und ansonsten mit State.ABORT frühzeitig abzubrechen. So kann sichergestellt werden, dass nur erfolgreiche Antworten verarbeitet werden.

Zeile 10 bis 17: Durch das onHeadersReceived Event können wichtige Informationen über die Antwort wie das verwendete Charset ermittelt und in Variabeln als eventübergreifenden Zustand gespeichert werden. Wie bei jedem Event muss auch hier durch die Rückgabe von State.CONTINUE die Weiterverarbeitung der asynchronen Aufgabe bestätigt werden.

Zeile 18 bis 22: Durch das onBodyPartReceived Event werden die eigentlichen HTML-Daten empfangen. Je nachdem wie gross die HTML-Daten sind kann dieses Event mehrere Male ausgelöst werden. Deswegen wird ein StringBuilder genutzt, um alle so erhaltenen HTML-Daten in einen String zusammenzusetzen. Durch das im onHeadersReceived Event ermittelte Charset kann korrektes Encoding sichergestellt werden. Wie bei jedem Event muss auch hier durch die Rückgabe von State.CONTINUE die Weiterverarbeitung der asynchronen Aufgabe bestätigt werden.

Zeile 23 bis 26: Das onCompleted Event wird ausgelöst sobald alle Teilaufgaben sowie alle Events der asynchronen Aufgabe abgeschlossen wurden. In diesem Beispiel wird hier der durch den in den onBodyPartReceived Events zusammengebaute komplette HTML-String zurückgegeben.

Zeile 27 bis 30: Auch Error Handling ist ein sehr wichtiges Thema. Durch das onThrowable event können alle in dieser asynchronen Aufgabe auftretenden Fehler behandelt werden.

Zeile 33: Eine 256-Bit Primzahl wird berechnet um Arbeit des Main Threads zu simulieren.

Zeile 35: Hier kann der von onCompleted zurückgegebene String von dem Future<String> durch .get() entnommen werden.

6.1.1.5 Reactive Streams

Im Letzten Kapitel über asynchronous Handlers wurde das Konzept der eventbasierenden Verarbeitung von asynchronen Teilaufgaben schon gezeigt. Durch Reactive wird dieses Konzept durch ein vollständiges Publisher/Subscriber-System erweitert. Anstelle direkt auf die Events im Handler zu subscriben und dort die gewünschte Manipulation der asynchronen Teilaufgaben zu erledigen, werden nun im Event onStream alle Daten vom Publisher zu den Subscribern gestreamt. Durch onNext kann der Subscriber die vom Publisher gestreamten Daten weiterverarbeiten. Durch weitere Events wie onComplete weiss der Subscriber wann der Publisher alle Daten fertig gestreamt hat. Anders als bei asynchronous Handlers kann man hier schlecht direkt Einfluss auf die asynchrone Aufgabe nehmen, sondern mehr auf die von

ihr gestreamten Daten, welche weiterverarbeitet und als eigentliches Resultat verwendet werden können. Das Schöne an diesem Konzept ist, dass dadurch nicht die asynchrone Aufgabe an sich manipuliert wird und somit mehrere unabhängige Subscriber den von der asynchronen Aufgabe generierten Dertenstream asynchron weiterverarbeiten können. Dies geschieht nicht im gleichen Thread wie der Handler, sondern jeder Subscriber kriegt einen eigenen Thread und muss somit seine Ressourcen nicht mit dem Handler teilen. Da die Subscriber nie die eigentliche asynchrone Aufgabe an sich beeinflussen können, gibt es hier auch keine Probleme wegen race conditions. Der Händler hat die Möglichkeit Daten von seinen Subscribern über getter Methoden zurückzuholen und danach dem Main Thread zurückzusenden. Des Weiteren hat der Handler immer noch alle in Kapitel über asynchronous Handlers beschriebenen Möglichkeiten, um die asynchrone Aufgabe zu beeinflussen. Reactive Programming ist ein sehr mächtiges Konzept jedoch auch nicht ganz einfach zu implementieren. Bevor man voll auf Reactive Programming setzt solle überlegt werden, ob nicht asynchronous Handlers alleine ausreichend ist. Braucht es nicht mehrere Subscriber, müsste asynchronous Handlers alles abdecken können.

```
1   public static String run() throws InterruptedException, ExecutionException, Throwable {
2       ListenableFuture<MyHandler> future = asyncHttpClient().prepareGet("http://www.nicobosshard.ch/Hi.html")
3           .execute(new MyHandler());
5       BigInteger.probablePrime(256, new Random());
7       byte[] result = future.get().getBytes();
8       return new String(result, "UTF-8");
9   }
10  static protected class MyHandler implements StreamedAsyncHandler<MyHandler> {
11      private final MySubscriber<HttpResponseBodyPart> sub;
12      MyHandler() {
13          this(new MySubscriber<>());
15      }
16      MyHandler(MySubscriber<HttpResponseBodyPart> sub) {
19          this.sub = sub;
20      }
21      @Override public State onStream(Publisher<HttpResponseBodyPart> pub) {
23          pub.subscribe(sub);
24          return State.CONTINUE;
25      }
42      @Override public MyHandler onCompleted() {
44          return this;
45      }
46      public byte[] getBytes() throws Throwable {
48          List<HttpResponseBodyPart> parts = sub.getElements();
```

```
49          ByteArrayOutputStream bytes = new ByteArrayOutputStream();
50          for (HttpResponseBodyPart item : parts) {
51              bytes.write(item.getBodyPartBytes());
52          }
53          return bytes.toByteArray();
54      }
55  }
56  static protected class MySubscriber<T> implements Subscriber<T> {
57      private final List<T> elements = Collections.synchronizedList(new ArrayList<>());
58      private final CountDownLatch latch = new CountDownLatch(1);
59      private volatile Subscription subscription;
60      private volatile Throwable error;
61      @Override public void onSubscribe(Subscription subscription) {
63          this.subscription = subscription;
64          subscription.request(1);
65      }
66      @Override public void onNext(T t) {
68          elements.add(t);
69          subscription.request(1);
70      }
76      @Override public void onComplete() {
78          latch.countDown();
79      }
80      public List<T> getElements() throws Throwable {
83          latch.await();
89          return elements;
91      }
92  }
```

Quelle: Lose basierend auf https://github.com/AsyncHttpClient/async-http-client/blob/master/client/src/test/java/org/asynchttpclient/reactivestreams/ReactiveStreamsDownloadTest.java

Wegen der Länge dieses Beispiels wurde es massiv gekürzt! Grosse Teile des Handlers und das ganze Error Handling und vieles mehr wurden entfernt. Falls Interesse an der vollständigen Version besteht empfehle ich das Programmierbeispiel anzuschauen.

Codebeschrieb:

Zeile 2 bis 3: Ein neues Future <MyHandler> wird erstellt und mit execute gestartet. Dem execute wird eine neue Instanz von MyHandler mitgegeben.

Zeile 5: Eine 256-Bit Primzahl wird berechnet um Arbeit des Main Threads zu simulieren.

Zeile 6 bis 7: Durch getBytes vom Handler können vom Subscriber alle HTML-Daten durch get() als Rohdaten empfangen werden. Diese werden per UTF-.8 encodiert und in einen String umgewandelt welcher gleich wie das Endergebnis aller anderen async http Beispiele ist.

Zeile 10 bis 55: Ein Handler welcher StreamedAsyncHandler implementiert und dazu dient Publisher und Subscriber miteinander zu verbinden. Dazu wird ein Subscriber initialisiert, und im onStream Event mit dem Publisher verbunden. Bei onCompleted wird diesmal der Handler

dem Main Thread weitergegeben, sodass dieser über getBytes vom Handler die Daten des Subscribes kriegen kann. Mehr über asynchronous Handlers kann im vorgehenden Kapitel nachgelesen werden.

Zeile 46 bis 55: Diese Funktion ist offiziell nicht Teil eines StreamedAsyncHandlers. Sie wird in diesem Beispiel nur dazu verwendet um Daten vom Subscriber über getElements zu erhalten, weiterzuverarbeiten und vom Main Thread aus zugreifbar zu machen. Da die getElements Funktion blockiert bis der Subscriber das onComplete Event enthält, ist auch diese Funktion blockierend. Dies ist jedoch kein Problem, da nur der Main Thread blockiert wird.

Zeile 56 bis 91: Ein Subscriber, welcher das Subscriber Interface Implementiert.

Zeile 61 bis 65: Durch den onSubscribe Event wird durch die Subscription die Verbindung zum Publisher registriert. Diese wird in onNext benötigt um dem Publisher zu signalisieren, dass man alle Events verarbeitet hat und bereit auf weitere Events ist.

Zeile 66 bis 70: Der onNext wird ausgeführt sobald der Publisher dem Subscriber HTML-Daten sendet. In diesem Beispiel werden diese einfach in einer Liste gespeichert und das Event danach als empfangen bestätigt.

Zeile 76 bis 69: Das onComplete Event signalisiert dem Subscriber, dass der Publisher fertig ist und keine weiteren Daten mehr zu erwarten sind. Dieses wird genutzt um durch ein CountDownLatch alle in der getElements Funktion wartenden Threads ihre Arbeit machen zu lassen.

Zeile 80 bis 91: Die getElements Funktion erlaubt es anderen Threads wie dem Handler, auf die vom Subscriber empfangenen und weiterverarbeiteten Daten zuzugreifen. Dieser Zugriff wird durch ein CountDownLatch so lange blockiert, bis das onComplete-Event dem Subscriber das Ende der asynchronen Aufgabe signalisiert.

6.1.2 Pseudozufallszahlengenerator

6.1.2.1 AkkaPRNG

Objektorientierte Programmierung war ursprünglich als Aktor-Modell mit Message Passing gedacht und nicht wie in modernen Programmiersprachen, für Klassen und Vererbungen gedacht. Java Akka bringt Actor Concurrency nach Java. (Cheung, 2018)

Anders als viele andere Parallelitätskonzepte verhindert Actor Concurrency race conditions nicht, sondern vermeidet diese. Dadurch ist Actor Concurrancy lock free. Dies hat den riesigen Vorteil, dass auch Deadlocks vermieden werden. Java Akka bietet zudem einen Thread Starvation Detector, welcher dem Dispatcher Warnungen bei verhungernden Threads liefern kann. Dadurch löst Java Akka alle grossen Probleme der parallelen Programmierung. Actor Concurrancy scheint die Zukunft der parallelen Programmierung zu sein und ist in beinahe allen Aspekten den anderen paralellen Programmierkonzepten überlegen. Programmierung mit Actor Concurrancy ist anders als normale parallele Programmierung einfach zu erlernen, gut verständlich und vorstellbar. Zudem vermeidet es die meisten durch parallele Programmierung verursachten Fehler schon auf Konzeptebene. In Aktoren getrennte Programmstücke, welche je seriell ablaufen und durch Nachrichten miteinander kommunizieren ist viel einfacher vorstellbar paralleler Code.

Java Akka bietet hervorragende Fehlertoleranz. Es ist mit Java Akka möglich selbstheilende Systeme zu entwickeln. Dies ist möglich, da bei Actor Concurrancy keine direkte Kopplung zwischen den verschiedenen Aktoren herrscht. Aktoren können sich also gegenseitig überwachen und auf Fehler reagieren und so das System automatisch wieder stabilisieren. Dies ist vor allem bei verteilten Systemen von besonderem Interesse. Ich werde dieses Thema in dieser Arbeit nicht behandeln aber bei Interesse kann man dies nachlesen auf Akka-in-action, chapter 4/22. [23]

Eine in dieser Arbeit nicht behandeltes aber in Hochleistungsrechnen weit verbreitetes Framework für parallele Programmierung ist das Message Passing Interface welches ein Subset des Actor Models ist. Mit Message Passing Interface werden Rechenaufgaben auf allen CPU-Cores und allen im Rechenzentrum vorhandenen Servern verteilt. Java Akka baut auf dieser Idee auf und macht sie ohne grosse Vorkenntnisse nutzbar. Java Akka ist nicht auf eine

[23] https://livebook.manning.com/book/akka-in-action/chapter-4/22 (Manning, 2020)

einzelne Maschine limitiert, sondern kann wie Message Passing Interface auch für verteilte Systeme genutzt werden. Wie auch Message Passing Interface bietet Java Akka beliebige Netzwerk Topologien. Ich werde mich in dieser Arbeit auf Java Akka für parallele Programmierung und nicht verteilte Systeme fokussieren. Wenn man sich für Java Akka für verteile Systeme interessiert kann man dies auf dies nachlesen auf Akka-in-action, chapter 6/1. [24]

Eine weit verbreitete Bibliothek welche Java Akka als backend nutzt ist AkkaHTTP. Diese wurde in einem eigenen Kapitel dieser Arbeit behandelt. Andere Module im Akka Framework sind: Akka Streams, Akka Cluster, Cluster Sharding, Distributed Data, Akka Persistence, Akka Projections, Akka Management, Alpakka, Alpakka Kafka und Akka gRPC. Wie man sieht ist Java Akka riesig. Deswegen beschränke ich mich in diesem Kapitel auf parallele Programmierung auf einem einzelnen System mit Java Akka Actors und beschränke mich auf die einfachste Art damit ein Actor System aufzubauen.

Main Thread:

```
1  public static void main(String[] args) {
2      final ActorSystem<MyRandom.printRandomNumbers> actorSystem = ActorSystem.create(MyRandom.create(),
   "AkkaGenerator");
3      actorSystem.tell(new MyRandom.printRandomNumbers(0, 100));
4  }
```

Codebeschrieb:

Zeile 1: Ein Actor welcher Nachrichten des Typs «printRandomNumbers» akzeptiert, wird mit dem System mit dem MyRandom-Actor erstellt

Zeile 2: Dem MyRandom-Actor wird eine Nachricht vom Typ «printRandomNumbers» gesendet in welcher er angefordert wird 100 Pseudozufallszahlen, startend mit dem Seed 0 zu generieren.

[24] https://livebook.manning.com/book/akka-in-action/chapter-6/1 (Manning, 2020)

“MyRandom” Actor:

```
5   public static class MyRandom extends AbstractBehavior<MyRandom.printRandomNumbers> {
6       public static class printRandomNumbers {
7           public final long seed;
8           public final int randomNumbersToGenerate;
9           public printRandomNumbers(long seed, int randomNumbersToGenerate) {
10              this.seed = seed;
11              this.randomNumbersToGenerate = randomNumbersToGenerate;
12          }
13      }
14      private final ActorRef<Generator.Generate> prng;
15      public static Behavior<printRandomNumbers> create() {
16          return Behaviors.setup(MyRandom::new);
17      }
18      private MyRandom(ActorContext<printRandomNumbers> context) {
19          super(context);
20          prng = context.spawn(Generator.create(), "generator");
21      }
22      @Override public Receive<printRandomNumbers> createReceive() {
23          return newReceiveBuilder().onMessage(printRandomNumbers.class, this::onPrintRandomNumbers).build();
24      }
25      private Behavior<printRandomNumbers> onPrintRandomNumbers(printRandomNumbers command) {
26          ActorRef<Receiver.RandomNumber> replyTo = getContext().spawn(Receiver.create(), "receiver");
27          prng.tell(new Generator.Generate(command.seed, command.randomNumbersToGenerate, replyTo));
28          return Behaviors.stopped();
29      }
30  }
```

Quelle: Lose basierend auf AkkaQuickstart Beispiel

Codebeschrieb:

Zeile 6 bis 13: Die «printRandomNumbers» Klasse spezifiziert wie eine Nachricht an den MyRandom-Actor auszusehen hat. In diesem Fall wird als erstes Argument beim Konstruieren dieser Nachricht ein Seed vom Datentyp long und ein Integer mit der Anzahl zu generierenden Zufallszahlen erwartet. Die einzige Aufgabe des Konstruktors ist es diese Variablen im Objekt zu speichern, sodass diese vom Empfänger der Nachricht gelesen werden können.

Zeile 14: Es wird eine leere Actorreferenz auf den Generator erstellt, sodass nach spawnen dieses Actors mit diesem über diese Actorreferenz kommuniziert werden kann.

Zeile 15 bis 17: Jeder Actor benötigt eine create () Methode durch welche eine neue Instanz dieses Actors erstellt werden kann.

Zeile 18 bis 21: Im Konstruktor des Actors muss immer der super constructor aufgerufen werden. Danach werden üblicherweise die zuvor definierten Actor Referenzen durch spawnen aller benötigten untergeordneten Actoren gefüllt. In diesem Beispiel wird also eine Instanz des Generators gespawnt und in der zuvor definierten prng Actor-Reverenz gespeichert.

Zeile 22 bis 24: Die empfangene Nachricht wird hier über den ReceiveBuilder für alle Handler wie in diesem Beispiel dem onPrintRandomNumbers-Handler vorbereitet und dann in Subscription-Reihenfolge aufgerufen.

Zeile 25 bis 29: Der onPrintRandomNumbers reagiert auf die vom Main Thread gesendete Nachricht um 100 Zufallszahlen startend bei Seed 0 zu generieren. Da bis anhin nur ein Generate-Actor erstellt wurde, wird nun ein Receiver-Actor gespawnt. Dem Generator wird nun eine Nachricht gesendet welche Seed, Anzahl zu generierenden pseudo Zufallszahlen sowie eine Actor Reverenz zum gerade eben gespawnten Receiver-Actor enthält. Durch Rückgabe von Behaviors.stopped() wird dem ActorSystem mitgeteilt, dass der MyRandom Actor gestoppt werden soll.

“Receiver” Actor:

```
31  public static class Receiver extends AbstractBehavior<Receiver.RandomNumber> {
32      public static final class RandomNumber {
33          public final long randomNumber;
34          public RandomNumber(long randomNumber) {
35              this.randomNumber = randomNumber;
36          }
37      }
38      public static Behavior<RandomNumber> create() {
39          return Behaviors.setup(Receiver::new);
40      }
41      private Receiver(ActorContext<RandomNumber> context) {
42          super(context);
43      }
44      @Override public Receive<RandomNumber> createReceive() {
45          return newReceiveBuilder().onMessage(RandomNumber.class, this::onGenerated).build();
46      }
47      private Behavior<RandomNumber> onGenerated(RandomNumber command) {
48          getContext().getLog().info("Received {}", command.randomNumber);
49          return this;
50      }
51  }
```

Codebeschrieb:

Zeile 32 bis 36: Die «RandomNumber» Klasse spezifiziert wie eine Nachricht einer generierten Zufallszahl an den Receiver-Actor auszusehen hat. Als einziges Element wird hier ein Wert vom Typ lang, also die eigentliche Zufallszahl erwartet. Diese wird im Konstruktor für den Empfänger im Objekt gespeichert.

Zeile 38 bis 40: Jeder Actor benötigt eine create () Methode durch welche eine neue Instanz dieses Actors erstellt werden kann.

Zeile 41 bis 43: Im Konstruktor des Actors muss immer der super constructor aufgerufen werden. Weiteres ist hier nicht notwendig da der Receiver-Actor keine untergeordneten Aktoren besitzt.

Zeile 44 bis 46: Die empfangene Nachricht wird hier über den ReceiveBuilder für den onGenerated Handler vorbereitet.

Zeile 47 bis 50: Die vom Generator generierte und hier vom Receiver empfangene pseudo Zufallszahl wird nun ausgegeben. Die Rückgabe von this ist äquivalent zu Behaviors.same() um dem ActorSystem mitzuteilen, dass das letzte bekannte Behaviour wiederverwendet werden soll.

“Generator” Aktor:

```
52  public static class Generator extends AbstractBehavior<Generator.Generate> {
53      public static final class Generate {
54          public final long seed;
55          public final int randomNumbersToGenerate;
56          public final ActorRef<Receiver.RandomNumber> replyTo;
57          public Generate(long seed, int randomNumbersToGenerate, ActorRef<Receiver.RandomNumber> replyTo) {
58              this.seed = seed;
59              this.randomNumbersToGenerate = randomNumbersToGenerate;
60              this.replyTo = replyTo;
61          }
62      }
63      public static Behavior<Generate> create() {
64          return Behaviors.setup(Generator::new);
65      }
66      private Generator(ActorContext<Generate> context) {
67          super(context);
68      }
69      @Override public Receive<Generate> createReceive() {
70          return newReceiveBuilder().onMessage(Generate.class, this::onDoGenerate).build();
71      }
```

```
72      private Behavior<Generate> onDoGenerate(Generate command) {
73          // Basieren auf C++ splitmix Generator von Arvid Gerstmann.
74          long seed = command.seed;
75          final long ulong1 = Long.parseUnsignedLong("11400714819323198485");
76          final long ulong2 = Long.parseUnsignedLong("13787848793156543929");
77          final long ulong3 = Long.parseUnsignedLong("10723151780598845931");
78          for (int i = 0; i < command.randomNumbersToGenerate; ++i) {
79              seed += ulong1;
80              long z = seed;
81              z = (z ^ (z >> 30)) * ulong2;
82              z = (z ^ (z >> 27)) * ulong3;
83              long randomNumber = (z ^ (z >> 31)) >> 31;
84              getContext().getLog().info("Generated {}!", randomNumber);
85              command.replyTo.tell(new Receiver.RandomNumber(randomNumber));
86          }
87          return Behaviors.stopped();
88      }
89  }
```

Codebeschrieb:

Zeile 53 bis 62: Die «Generate» Klasse spezifiziert wie eine Nachricht vom MyRandomActor an den Generate Actor auszusehen hat. Das erste Argument beim Konstruieren dieser Nachricht muss ein Seed vom Datentyp long, das zweite die Anzahl zu generierenden Zufallszahlen vom Typ int und das dritte muss eine Actor Reverenz zu einem Receiver Actor sein. Der Konstruktor speichert diese Argumente für den Empfänger im Objekt ab.

Zeile 63 bis 65: Jeder Actor benötigt eine create () Methode durch welcher eine neue Instanz dieses Actors erstellt werden kann.

Zeile 66 bis 68: Im Konstruktor des Actors muss immer der super constructor aufgerufen werden. Weiteres ist hier nicht notwendig, da der Generator Actor keine untergeordneten Actoren besitzt.

Zeile 69 bis 71: Die empfangene Nachricht wird hier über den ReceiveBuilder für den onDoGenerate Handler vorbereitet.

Zeile 72 bis 88: Hier wird die eigentliche Zufallszahl über den Splitmatrix Algorithmus von Arvid Gerstmann generiert. Diesen habe ich von C++ zu Java portiert weshalb ich für die vorzeichenlosen 64-bit Ganzzahlen parseUnsignedLong verwenden musste. Über das als Argument erhaltene Generate Objekt kann auf die benötigten Parameter wie Seed, Anzahl zu generierenden Zufallszahlen und wem diese gesendet werden sollen, zugegriffen werden. Nach jedem Generieren einer Zufallszahl wird diese gleich asynchron dem Receiver gesendet.

6.1.3 Java Project Loom Preview

Mit dem Projekt Loom ist es endlich möglich, in Java Fibers zu verwenden. Fibers sind leichtgewichtige virtuelle Threads von welchen sich mehrere einen realen OS Thread teilen können. Dadurch wird das Erstellen und Löschen von Threads beinahe kostenlos.

Viele funktionale Programmiersprachen wie Erlang und Haskell kennen das Prinzip von Fibers schon seit Jahrzenten. Bei älteren prozeduralen Programmiersprachen wurde Fibers bis anhin eher vernachlässigt während neuere Sprachen wie beispielsweise Golang gar nichts anderes als Fibers kennen. Es ist höchste Zeit für Java mit Project Loom endlich auch Fibers einzuführen. Fibers lösen alle Probleme von Thread Pools wie beispielsweise das im Kapitel Thread Pools behandelte Problem mit dem Versenden von Ressourcen durch offengelassene Threads bei dem fixed Thread Pool so wie auch das denial of service Problem beim cached Thread Pool. Eigentlich kann man bei Fibers nicht mal mehr von einem Thread Pool reden, da Threads nicht mehr gecacht werden. Zur einfachen Migration auf Fibers und um erfahrenen Java Programmierern den Umstieg so einfach wie möglich zu machen, wurde die Thread Pool Syntax bei Project Loom beibehalten.

Programmierer mit Grafikkartenprogrammiererfahrung wo Threads aus Hardwaregründen kostenlos sind, hätten wahrscheinlich lieber eine GPU Kernel ähnlichen Syntax gehabt. Aber sogar dies lässt sich durch invokeAll und nCopies einfach realisieren. Die Frage ist eher, wie effizient und intuitiv dies ist.

Project Loom unterstützt Cancellation. Das heisst eine Fiber kann bei jedem yield-Punkt abbrechen. Auch bestehende Java Funktionen wurden auf Fibers portiert und blockierende Aufrufe können nun sicher unterbrochen werden.

"A crucial Loom contribution, making interruption more practical is the ability to cancel (interrupt) a fiber on any yield point. Combined with the retrofitted Java APIs, “blocking” calls can now be safely interrupted, as opposed to the current situation." (Warski, 2020)

Eine weitere für die parallele Programmierung sehr bedeutende Neuerung, welche vom Project Loom eingeführt wird, sind die Continuations. Durch Continuations ist es möglich, Codeausführung direkt durch die Programmiersprache ohne OS Scheduler zu unterbrechen. Dies ist eine riesige Leistung und erlaubt das Umschreiben vieler blockierenden Funktionen in nicht blockierende Funktionen. Dies nicht nur Code und auf Bibliotheksebene, sondern auch

viele in Java integrierte Funktionen können dadurch neu in nicht blockierender Art implementiert werden. Continuations sind technisch gesehen ähnlich wie Kotlins suspendable Functions. Sie kosten aber nur falls die Funktion tatsächlich durch yield unterbrochen wird und nicht wie momentan in Kotlin wo jeder Aufruf einer suspendable Funktion von einer anderen suspendable Function was kostet. Somit sind sie fundamental besser als Kotlins suspendable Functions.

Durch Continuations und Fibers wird es mit Project Loom möglich sein, in Java goroutinenähnliche Konzepte zu implementieren. Konzepte die besser sind als das, was die auf Thread Pool basierenden Kotlin Coroutinen momentan bieten können.

Project Loom wird die parallele Programmierung in Java revolutionieren. Viele Prallelitätskonzepte und Bibliotheken, welche momentan Thread Pools nutzen, werden auf Fibers umsteigen. Auch eröffnen Loom mit Fibers und Tail-Call Elimination das Tor zu besseren funktionalen Programmierkonzepten in Java, während mit Continuations eine sehr gute Alternative zu iterierbaren Objekten geschaffen wird.

Trotz aller Freude löst Loom nicht alle Probleme von Java. Channels werden auch in Zukunft über BlockingQueue's implementiert werden müssen, was alles andere als schön ist, wenn man es mit anderen Programmiersprachen wie Kotlin oder Golang vergleicht. Auch gibt es trotz Loom noch keine offiziellen Coroutinen für Java, was definitiv eine verpasste Chance war, da sich dies bestens hätte kombinieren lassen. Nach Loom werden sicherlich Bibliotheken mit Coroutinen ähnlichen Konzepten für Java kommen und ohne Standard ist es schwer zu entscheiden, welche Bibliothek für ein grösseres Projekt benutzt werden soll. Auch bei Continuation vermisse ich das von C# bekannte yield return wo mit yield gleich eine Variable übergeben werden kann. In Loom muss dies noch manuell über geteilte Variablen gemacht werden.

Weiterführende Wissensquelle:

Will-project-loom-obliterate-java-futures. [25]

[25] https://blog.softwaremill.com/will-project-loom-obliterate-java-futures-fb1a28508232 (Warski, 2020)

6.1.3.1 LoomFiberCounter

```
1   public static void main(String[] args) {
2       AtomicInteger count = new AtomicInteger();
3       ExecutorService pool = Executors.newVirtualThreadExecutor();
4       for (int i = 0; i < 1000000; ++i) {
5           pool.submit(() -> { count.incrementAndGet(); });
6       }
7       pool.shutdown();
16      System.out.println(count.get());
17  }
```

Codebeschrieb:

Zeile 2: Ein atomarer Integer wird erstellt um das parallele Hochzählen zu demonstrieren.

Zeile 3: Eine ExecutorService von welcher Fibers als Backend nutzt wird erstellt.

Zeile 4 bis 6: Eine Million Tasks werden submitted welche je in einer eigenen Fiber gestartet werden.

Zeile 7: Das Beenden des Pools wird angefordert und auf das Beenden aller Fibers gewartet.

Zeile 8: Das erwartete Resultat «1000000+ wird ausgegeben.

Durch verwenden von Collections gibt es jedoch eine aus Sicht von Programmierern mit Grafikkartenprogrammiererfahrung folgende schönere Art obigen Code zu programmieren:

```
1   public static void main(String[] args) throws InterruptedException {
2       AtomicInteger count = new AtomicInteger();
3       ExecutorService pool = Executors.newVirtualThreadExecutor();
4       pool.invokeAll(Collections.nCopies(1000000, count::incrementAndGet));
5       pool.shutdown();
6       System.out.println(count.get());
7   }
```

Codebeschrieb:

Zeile: 4: Anders als auf **Zeile 4 bis 6** vom letzten Beispiel werden hier gleich eine Million Instanzen des gleichen Tasks auf einer einzelnen Zeile gestartet. Dieses Konzept ist sehr populär auf Single Instruction Multiple Data (SIMD) Architekturen wie Grafikkarten.

6.1.3.2 LoomContinuationPRNG

Dieses Beispiel zeigt, wie sich in Java mit Project Loom durch Continuations und virtuellen Threads ein schöner Pseudozufallszahlgenerator erstellen lässt welcher in einer separaten Fiber läuft.

Hier Fibers anstelle von Threads zu nutzen spielt in diesem Beispiel nicht so eine Rolle aber in Realität wie beispielsweise in einem Vidoegame hat man in der Regel sehr viele solcher Pseudozufallszahlgeneratoren, welche andauernd erstellt und wieder abgebaut werden, da oft jedes Objekt seinen eigenen Zufallsgenerator hat und Objekte durch Bewegung des Players andauernd geladen und entladen werden. Dennoch sollen Pseudozufallszahlgeneratoren nicht auf demselben Thread wie der Rest des Codes des Objekts laufen. Genau deswegen gibt es Fibers, welche sich beinahe kostenlos erstellen lassen.

Durch Continuations wird die Funktion welche die Zufallszahlen dem Main Thread sendet von dem eigentlichen Zufallsgenerator getrennt. Dadurch lässt sich wesentlich saubereren und besser verständlicheren Code schreiben als wenn beide in derselben Funktion wären. Es zeigt gut, wie Continuations Iterables in solchen Szenarien ablösen werden.

Wie schon in der Project Loom Einführung erwähnt, gibt es mit Loom keine neuen Interthreadkommunikationsmöglichkeiten, weshalb hier leider noch eine BlockingQueue anstelle richtiger Channels verwendet werden muss.

Beispiele für einen PRNG ohne Continuations oder Continuations ohne Fibers befinden sich in den Codebeispielen. Da dieses Beispiel jedoch beide Konzepte miteinander verbindet ist es das einzige, auf welches ich in meiner Arbeit genauer eingehen werde.

```
1   public static void main(String[] args) {
2       BlockingQueue<Long> queue = new ArrayBlockingQueue<>(10);
3       var fiber = Thread.startVirtualThread(() -> {
4           AtomicLong result = new AtomicLong();
5           var scope = new ContinuationScope("LoomPRNG");
6           var continuation = new Continuation(scope, () -> {
7               var seed = 1;
8               final long ulong1 = Long.parseUnsignedLong("11400714819323198485");
9               final long ulong2 = Long.parseUnsignedLong("13787848793156543929");
10              final long ulong3 = Long.parseUnsignedLong("10723151780598845931");
11              while(true) {
12                  seed += ulong1;
```

```
13                  long z = seed;
14                  z = (z ^ (z >> 30)) * ulong2;
15                  z = (z ^ (z >> 27)) * ulong3;
16                  result.set((z ^ (z >> 31)) >> 31);
18                  Continuation.yield(scope);
19              }
20          });
21
22          while (true) {
24              continuation.run();
26              queue.put(result.get());
30          }
31      });
32      for(int i = 0; i < 100; ++i) {
34          System.out.println(queue.take());
38      }
40  }
```

Codebeschrieb:

Zeile: 2: Hier wird die BlockingQueue erstellt welche als nicht so schöne Java Alternative eines Channels angesehen werden kann.

Zeile: 3: Hier wird eine neue Fiber gestartet.

Zeile: 5: Ein neues ContinuationScope namens «LoomPRNG» wird erstellt.

Zeile: 6 bis 20: Mit dem auf **Zeile 5** erstellten ContinuationScope wird ein neuer, mit yield unterbrechbarer Bereich erstellt.

Zeile: 7 bis 19: Hier befindet sich der eigentliche Pseudozufallszahlgenerator basierend auf dem Splitmatrix Algorithmus von Arvid Gerstman. Diesen habe ich von C++ zu Java portiert, weshalb ich für die vorzeichenlosen 64-bit Ganzzahlen parseUnsignedLong verwenden musste.

Zeile: 18: Durch yield kann in die Elternfunktion gesprungen werden.

Zeile: 22 bis 30: Hier wird über continuation.run() die Continuation ausgeführt und das so über die geteilte Variable erhaltene Resultat über eine BlockingQueue an den Main Thread gesendet.

Zeile: 32 bis 38: Es wird hier auf 100 Zufallswerte gewartet, diese ausgegeben und das Programm danach beendet. Alle Fibers werden mit dem Programmende automatisch geschlossen. Dies funktioniert so gut da Project Loom Cancellation unterstützt.

6.2 Golang

Golang ist eine auf Nebenläufigkeit spezialisierte moderne, imperative, streng typisierte Programmiersprache mit einem Garbage Collector. Sie wurde durch Google Mitarbeiter entwickelt und die erste stabile Version wurde 2012 veröffentlicht. Golang ist somit eine verhältnismässig moderne Programmiersprache und wurde mit dem Ziel entwickelt, Nebenläufigkeit durch native Sprachelemente zu unterstützen. Golang wird direkt in Maschinensprache kompiliert und ist somit beinahe so schnell wie C++. Die Sprache wurde aber so konstruiert, dass sie sehr einfach zu kompilieren ist und somit wesentlich schneller kompiliert als C++. Golang ist relativ einfach zu erlernen und man kann mit wenig Programmieraufwand recht viel erreichen. Die Fehlerbehandlung in Golang geschieht über Rückgabewerte was für Programmierer, welche von Programmiersprachen wie Java mit try/catch error Handling herkommen, etwas gewöhnungsbedürftig ist. Denkt man aber etwas darüber nach, ist dies eine viel elegantere Art der Fehlerbehandlung und entspricht viel mehr der Fehlerbehebung in der realen Welt und ist weniger komplex als try/catch. Eine weitere Spezialität von Golang ist das eingebaute dependency Management. Es können einfach GitHub URLs angegeben werden und die benötigten Abhängigkeiten werden automatisch heruntergeladen. Dies ist im Vergleich zu externen dependency Management Tools wie beispielsweise Maven viel komfortabler, da keine separate Konfigurationsdatei heruntergeladen werden muss.

Die grosse Besonderheit durch parallele Programmierung in Golang sind Goroutinen und Channels. Dadurch wird Nebenläufigkeit durch mehrere miteinander kommunizierende Threads realisiert. Offiziell heisst dieses parallele Programmiermodel *Communicating Sequential Processes*. Ähnlich wie beim Actor Modell gibt es auch hier keine race conditions, da keine geteilten Variablen verwendet werden. Anders als das Actor Modell bieten *Communicating Sequential Processes* jedoch keine Fehlertoleranz oder Selbstheilung und somit auch nicht eine so gute Ausfallssicherheit/Verfügbarkeit. Anders als das Actor Modell bieten Goroutinen keinen fundamentalen Schutz vor Deadlocks. Es ist Sache vom Programmierer durch ein garantiert deadlockfreies Kommunikationsmuster wie Client/Server-Kommunikation das Programm deadlockfrei zu gestalten. Der Hauptunterschied liegt darin, dass die Nachrichten bei Goroutinen synchroner und an den Prozess gekoppelt sind, während beim Actor Modell diese asynchron und komplett vom Actor getrennt sind. Dies

da bei Golang die Kapazität der Channels immer limitiert ist, während sie im Actor Modell unlimitiert ist. Das hat aber beim Actor Modell auch erhebliche Nachteile. Oft ist Flow Control notwendig ist da Buffers nicht endlos gross sein können und man ansonsten irgendwann kein Arbeitsspeicher mehr hat. Mit Flow Control hat dann das Actor Modell wieder ähnliche Nachteile wie Goroutinen.

Anders als das Actor Modell sind Goroutinen jedoch viel einfacher zu erlernen und weniger komplex. Auch muss man zum Erstellen von Goroutinen viel weniger schreiben als um einen vollständigen Actor zu erstellen. Goroutinen erlauben es also Programmierern mit wenig Erfahrung in paralleler Programmierung und keiner Erfahrung des Actor Modells dennoch die meisten Vorteile des Actor Modells zu nutzen. Auch für erfahrene Programmierer machen Goroutinen für kleinere Programme durchaus Sinn, da sie die Menge an notwendigem Code vermindern und somit die Lesbarkeit erhöhen. Auch erlauben Goroutinen dasselbe in viel weniger Zeit zu erreichen. Allgemein geht es bei der Programmierung mit Golang immer darum möglichst schnell möglichst viel zu erreichen und so die Effizienz eines Programmierers zu steigern ohne dass die Performance darunter leidet. Für grössere Projekte sollte aber auch in Golang Actor concurrancy verwendet werden, da Goroutinen mit wachsender Komplexität des Systems irgendwann an ihre Grenzen stossen werden und man mit Deadlocks zu kämpfen hat. Dazu kann die protoactor-go Bibliothek verwendet werden. Allem in allem sind Goroutinen ein revolutionäres paralleles Programmierkonzept, welches sich vor allem für kleinere und mittelgrosse Projekte enorm eignet. Golang erlaubt es durch Goroutinen mit wenigen Zeilen Code die CPU aller Cores voll auszunutzen ohne grosses Wissen über parallele Programmierung zu erfordern. Dies ist eine riesige Leistung einer sehr gelungenen Programmiersprache.

Weiterführende Wissensquellen:

Go (Programmiersprache), Wikipedia. [26]

Actor vs goroutine. [27]

[26] https://de.wikipedia.org/wiki/Go_(Programmiersprache) (Wikipedia, Go (Programmiersprache), 2021)

[27] http://gqlxj1987.github.io/2019/03/20/actor-vs-goroutine/ (DevilKing, 2019)

6.2.1 Asynchroner HTTP-Request

In den nachfolgenden Beispielen zeige ich die Verwendung verschiedener komplexer Goroutinen für das asynchrone HTTP Beispiel.

6.2.1.1 Goroutines

6.2.1.1.1 Einfache Goroutine

Als Einführung in Goroutinen dient folgendes Beispiel, welches absichtlich so einfach wie möglich gestaltet wurde. Es findet keine Fehlerbehandlung statt und Ressourcen werden in sequenzieller Reihenfolge manuell freigegeben.

```
1   func main() {
2       result := make(chan string, 1)
3       url := "http://www.nicobosshard.ch/Hi.html"
4       go func(url string) {
5           response, _ := http.Get(url)
6           bodyBytes, _ := ioutil.ReadAll(response.Body)
7           result <- string(bodyBytes)
8           response.Body.Close()
9       }(url)
10      fmt.Println("Task läuft...")
11      value := <-result
12      close(result)
14  }
```

Codebeschrieb:

Zeile: 2: Hier wird ein Channel erstellt um zwischen der Goroutine und dem Main Thread zu kommunizieren. Dieser ist auf maximal einen String beschränkt.

Zeile: 3: Hier wird eine variable definiert welche den URL enthält.

Zeile: 4: Durch das Keyword go wird eine Goroutine erstellt, welche asynchron zum Main Thread läuft. Der in Klammer definierte Parameter ist nötig sodass die Goroutine auf den ausserhalb der Goroutine definierte URL zugreifen kann.

Zeile: 5 bis 8: Ein HTTP Request auf die zuvor definierte URL wird ausgeführt und die eigentlichen HTML-Daten über einen Stream extrahiert, welcher danach geschlossen wird. Fehler werden durch discard Variablen ignoriert.

Zeile: 7: Das Resultat wird über den Channel dem Main Thread gesendet.

Zeile: 9: Hier wird die ausserhalb der Goroutine definiert URL der Goroutine als Argument mitgegeben.

Zeile: 10: Diese Zeile läuft parallel zur Goroutine

Zeile: 11 bis 12: Es wird gewartet bis die Goroutine ihr Resultat in den Channel schreibt und dieses danach empfangen und der Channel danach geschlossen.

6.2.1.1.2 Goroutine mit Fehlerbehandlung

Dieses Beispiel zeigt die gleiche Goroutine wie im Beispiel zuvor diesmal aber mit Fehlerbehandlung und defer um sicherzustellen, dass am Ende der Goroutine korrekt aufgeräumt wird.

```
4   go func(url string) {
5       response, err := http.Get(url)
6       if err != nil {
7           log.Fatal("HTTP get error: ", err)
8       }
9       defer response.Body.Close()
10      bodyBytes, err := ioutil.ReadAll(response.Body)
11      if err != nil {
12          log.Fatal("HTTP ReadAll error: ", err)
13      }
14      result <- string(bodyBytes)
15  }(url)
```

Codebeschrieb:

Zeile: 5 bis 8: Ein HTTP Request auf die zuvor definierte URL wird ausgeführt.

Tritt auf **Zeile 5** ein Fehler auf so wird dieser auf **Zeile 7** gehandelt. In diesem Fall indem das Programm mit einer Fehlermeldung terminiert.

Zeile: 9: Das defer bedeutet, dass auch falls auf **Zeile 12** abgebrochen wird, aufgeräumt wird.

Zeile: 10 bis 13: Hier werden die eigentlichen HTML-Daten über einen Stream extrahiert.

Tritt auf **Zeile 10** ein Fehler auf so wird dieser auf **Zeile 15** gehandelt. In diesem Fall indem das Programm mit einer Fehlermeldung terminiert.

6.2.1.1.3 Mehrfachparallele Goroutine

Bis anhin hatten alle Golang Beispiele nur eine Goroutine gestartet. Um die Skalierbarkeit von Golang zu demonstrieren ändere ich hierfür mein Standardbeispiel und anstelle eines asynchronen HTTP Downloads werde ich in diesem Beispiel 10 asynchrone http Downloads zeigen, wobei der Server nicht überlastet werden soll, weswegen die Anzahl paralleler Downloads limitiert werden soll. Dazu wird ein Guard durch Channels erstellt. Dies ist möglich da Channels blockierend sind, wenn sie voll sind. Sie können deswegen genutzt werden, um das Starten von zu vielen parallelen Goroutinen zu verhindern, indem vor jedem Starten einer Goroutine ein dummy Objekt in den Channel geschrieben wird. Nach dem Beenden einer Goroutine wird das dummy Objekt wieder aus dem Channel entfernt. Das Konzept in Golang mit den Guards durch Channels entspricht den Semaphore in anderen Programmiersprachen.

Diese mit Dummy Objekten gefüllten Channels zeigen wie Golang mit möglichst wenig nativen Sprachkonzepte auskommen will, um den Einstieg für Anfänger so einfach wie möglich zu machen. Anstatt, dass in Golang das Konzept einer Semaphore einzuführen wurde, werden hier Channels wiederverwendet. Eine kontroverse Entscheidung für welche es sicher von beiden Seiten viele gute Argumente gibt.

Guards sind in der parallelen Programmierung mit Golang sehr wichtig. Ansonsten werden hunderttausende von Fibers erstellt, was Webserver überlasten und bei IO Operationen sogar das Dateisystem beschädigen kann. Dies habe ich aus eigener Erfahrung schon erlebt und verlor alle Daten auf dieser SSD Partition, welche ich glücklicherweise extra für dieses Golang script angelegt habe und somit keine wichtigen Daten verloren habe.

```
1   func main() {
2       tasks := 10
3       maxGoroutines := 4
4       guard := make(chan struct{}, maxGoroutines)
5       result := make(chan string, tasks)
7       for i := 0; i < tasks; i++ {
8           guard <- struct{}{}
9           go func(url string) {
19              result <- string(bodyBytes)
20              <-guard
21          }(url)
22      }
24      for i := 0; i < tasks; i++ {
25          value := <-result
27      }
29  }
```

Um das Beispiel so einfach wie möglich zu halten wurden alle zu Guards irrelevanten Teile rausgekürzt. Siehe dazu vorgängiges Beispiel oder die Codebeispiele.

Codebeschrieb:

Zeile: 4: Hier wird dar Guard Channel definiert. Er kann maximal 4 dummy Objekte speichern und dient zum Limitieren der Anzahl laufender Goroutinen.

Zeile: 5: Hier wird ein Channel erstellt um zwischen der Goroutine und dem Main Thread zu kommunizieren. Dieser ist auf 10 beschränkt um Platz für alle Tasks zu haben. Eine kleinere Einschränkung wäre hier möglich jedoch sollte diese grösser als die maximalen parallelen Goroutinen sein da dadurch ansonsten die Parallelität eingeschränkt wird.

Zeile: 7 bis 22: In dieser for-Schleife werden 10 Goroutinen gestartet.

Zeile: 8: Hier wird das dummy-Objekt in den Channel geschrieben. Es wird eine leere struct verwendet um keinen Arbeitsspeicher zu verschwenden.

Zeile: 9 bis 21: Hier befindet sich die eigentliche Goruntine welche 10-mal gestartet wird.

Zeile: 19: Das Resultat wird wie zuvor in den «result»-Channel gesendet

Zeile: 20: Am Ende der Goroutine wird ein dummy-Objekt aus dem Channel genommen. Dies sollte in der Regel mit defer gemacht werden auf was hier aufgrund der Verständlichkeit verzichtet wurde. Auch ist dies hier nicht nötig da jeder mögliche Error das Programm abbricht. Siehe nächstes Beispiel um das defer des Guards zu sehen.

Zeile: 24 bis 27: Über einen Loop werden alle 10 Resultate ausgelesen. Ist eine Goroutine noch nicht fertig so wird auf hier diese gewartet. Siehe nächstes Kapitel falls dies nicht so gewollt ist.

6.2.1.1.4 Mehrfachparallele Goroutine mit WaitGroup

Da man bei Golang nur dann auf eine Goroutine warten muss, wenn ihr Resultat verwendet wird, ist sehr nützlich. Dies ist in vielen Anwendungen auch die empfohlene Art um Goroutinen zu nutzen. Dies ist jedoch nicht immer möglich, vor allem wenn es um API-Aufrufe oder IO Operationen geht, da es sonst zu race conditions ausserhalb des eigentlichen Golang programms kommen könnte. Es muss somit eine Möglichkeit geben auf alle Goroutinen eines Blocks zu warten. Genau dies bietet Golang mit WaitGroups. Zusätzlich reduzieren WaitGroups die Komplexität, da danach das Programm in der Regel in einem klar definierten Zustand ist. Leider erhöhen WaitGroups auch das Risiko von Deadlocks erheblich. WaitGroups sind etwa das Gegenteil einer Guard/Semaphore. Es wird für jede Goroutine um eins hochgezählt und beim Abschluss einer Goroutine hinuntergezählt. Der Main Thread wartet dann bis die WaitGroup wieder bei null ist. Die Verwendung genau solcher WaitGroups werde ich in diesem Beispiel zeigen.

```
1   func main() {
2       var wg sync.WaitGroup
8       for i := 0; i < tasks; i++ {
9           guard <- struct{}{}
10          wg.Add(1)
11          go func(url string) {
12              defer func() { <-guard }()
13              defer wg.Done()
23              result <- string(bodyBytes)
24          }(url)
25      }
27      wg.Wait()
28      for i := 0; i < tasks; i++ {
29          value := <-result
31      }
33  }
```

Um das Beispiel so einfach wie möglich zu halten, wurden alle zu WaitGroup irrelevanten Teile herausgekürzt. Siehe dazu vorgängiges Beispiel oder die Codebeispiele.

Codebeschrieb:

Zeile: 2: Hier wird eine neue WaitGroup erstellt.

Zeile: 8: Dieser Loop erstellt wie schon im letzten Beispiel 10 Instanzen dieser Goroutine.

Zeile: 9: Wie schon im letzten Beispiel wird ein Dummy-Element in den Guard stream gespeichert um die Anzahl parallellaufender Goroutinen zu beschränken.

Zeile: 10: Die WaitGroup wird vor dem Starten der Goroutine um eines erhöht, um zu wissen wie viele Goroutinen momentan laufen.

Zeile: 12: Am Ende der Goroutine wird ein Dummy-Objekt aus dem Channel genommen. Dazu wird defer verwendet da ansonsten diese Aktion bei einem Error nicht ausgeführt werden würde. Im letzten Beispiel wurde dies um die Verständlichkeit zu erhöhen weggelassen, da der Syntax etwas gewöhnungsbedürftig ist. Golang erlaubt über defer nur Funktionen aufzurufen. Somit muss diese Zeile in eine Funktion umgewandelt werden. Am Ende der Funktion ist () notwendig da es eine Funktion ohne Argumente ist.

Zeile: 13: Per defer wg.Done() wird definiert, dass egal was passiert, die WaitGroup beim Verlassen diese Goroutine um eins verringert werden muss.

Zeile: 23: Das Resultat wird wie zuvor in den Result Channel gesendet

Zeile: 27: Hier wird gewartet bis die WaitGroup wieder bei 0 ist und somit alle Goroutinen abgeschlossen wurden.

Zeile: 28 bis 31: Über einen Loop werden alle 10 Resultate ausgelesen. Da alle Goroutinen schon fertig sind, muss hier auch nicht gewartet werden.

Achtung: Würde der Result Channel in diesem Beispiel kleiner als 10 sein, so hätte man ein Deadlock, da sich der Result Channel füllen wird. Dies da sich der Konsument nach der WaitGroup befindet und somit nie vor Abschliessen aller Goroutinen erreicht werden kann. Es ist gut ersichtlich wie WaitGroups die Gefahr von Deadlocks erheblich erhöhen. Sowas wäre bei Actor Parallelisierung niemals passiert da dort Chanels keine Limitierungen haben.

6.2.1.2 Asynchrone Programmierung

Es ist mit Golang sehr einfach möglich, dass in anderen Programmiersprachen so gelobte async/await zu implementieren. Dies ist sehr erstaunlich, da dies bei anderen Programmiersprachen massive Änderungen im Kompiler erforderte. Dies ist möglich da Goroutinen und allgemein *Communicating Sequential Processes* alle für async/await notwendigen Konzepte implementien und somit ohne Kompileränderungen zum Laufen bringen. Bei mehr Interesse zu diesem Thema unbedingt mein Codebeispiel async.go anschauen, das die ganze async/wait Implementierung enthält. Das Codebeispiel basiert auf hackernoon.com [28]

Auf dieser Webseite ist die Implementierung auch schon bestens erklärt. Deswegen werde ich in dieser Arbeit nicht weiter darauf eingehen.

Es gibt keinen Grund async/await zu nutzen, wenn man Goroutinen zur Verfügung hat, da sie das mächtigere parallele Programmierkonzept sind. Durch das Erstellen neuer paralleler Programmierkonzepte kann jedoch die parallele Programmierung nach eigenen Wünschen selbst weiter abstrahiert werden. Mächtigere parallele Programmierkonzepte können oft weniger mächtige parallele Programmierkonzepte implementieren, wenn auch nicht mit so schöner Syntax. Wie schon in früheren Beispielen gesehen bietet Golang keine Semaphoren, da sich diese durch das mächtigere parallele Programmierkonzept von Channels sehr einfach selbst implementieren lassen. Selbes gilt in Golang für viele parallele Programmierkonzepte. Der einzige Grund dies in meiner Arbeit zu erwähnen ist zum Zeigen, dass man nicht durch die von der Programmiersprache vorgegebenen parallelen Programmierkonzepte limitiert ist. Wenn gewünscht kann man immer seine eigenen entwickeln. Dabei muss jedoch beachtet werden, dass es manchmal unmöglich ist, gewisse parallele Programmierkonzepte zu implementieren, da gewisse Bausteine wie beispielsweise Fibers oder unterbrechbare Funktionen fehlen. Deswegen ist es auch von so grosser Bedeutung, dass Projekt Loom bald in die offiziellen JVM integriert werden wird.

[28] https://hackernoon.com/asyncawait-in-golang-an-introductory-guide-ol1e34sg (Ahad, 2020)

```
1   func asyncHttpRequest() string {
3       response, err := http.Get(url)
7       defer response.Body.Close()
8       bodyBytes, err := ioutil.ReadAll(response.Body)
12      return string(bodyBytes)
13  }
14
15  func main() {
16      future := async(func() interface{} {
17          return asyncHttpRequest()
18      })
19      fmt.Println("Task läuft...")
20      val := future.Await()
21      fmt.Println(val)
22  }
```

Hier ein Beispiel wie Goroutinen durch async und await weiter abstrahiert werden können. Der Abstraktionscode befindet sich weiter oben in async.go. Fehlerbehandlung, sowie weitere in den früheren Golang Beispielen besprochene Konzepte wurden rausgekürzt da die Betonung auf der Implementierung von async/await liegen soll.

Codebeschrieb:

Zeile: 1 bis 13: Die Funktion asyncHttpRequest wird asynchron ausgeführt

Zeile: 12: Da dieses Beispiel async/await nutzt, werden keine Channels für die Kommunikation zwischen asynchroner Aufgabe und Main Thread verwendet, sondern ein return Statement.

Zeile: 26 bis 18: Hier wird die asynchrone Aufgabe gestartet.

Zeile: 19: Diese Zeile läuft parallel zur asynchronen Aufgabe

Zeile: 20: Durch await wird auf die parallele Aufgabe gewartet und ihr Resultat in eine Variable gespeichert.

Zeile: 21: Das von der asynchronen Aufgabe erhaltene Resultat wird hier ausgegeben.

Die Funktion asyncHttpRequest wird auf Zeile 59-61 asynchron ausgeführt. Zeile 62 läuft parallel zum asynchronen Task. Zeile 63 wartet auf den asynchronen Task.

6.2.2 Pseudozufallszahlengenerator

Auch Generatoren lassen sich mit Goroutinen problemlos implementieren. Nicht nur das, Golang ist meiner Meinung nach sogar die schönste Programmiersprache um diesen zu implementieren. Wie schon in den meisten anderen Programmiersprachen zeige ich in diesem Beispiel eine Golang Implementierung des auf dem von Arvid Gerstmann erfundenen Pseudozufallsgenerator splitmix. Dazu habe ich diesen von C++ nach Golang portiert.

```
2   func random(seed uint64, result chan uint64) {
3       for {
4           seed += 0x9E3779B97F4A7C15
5           var z uint64 = seed
6           z = (z ^ (z >> 30)) * 0xBF58476D1CE4E5B9
7           z = (z ^ (z >> 27)) * 0x94D049BB133111EB
9           result <- (z ^ (z >> 31)) >> 31
10      }
11  }
13  func main() {
14      cache := 10
15      tasks := 100
16      result := make(chan uint64, cache)
17      go random(0, result)
18      for i := 0; i < tasks; i++ {
19          value := <-result
20          fmt.Println(value)
21      }
23  }
```

Eigentlich gibt es dazu nicht mal mehr viel zu sagen, da alle hier verwendeten Konzepte in früheren Kapiteln schon beschrieben wurde. Golang ist eine relativ einfach zu lernende Programmiersprache und somit wird das meiste schon klar sein. Dennoch werde ich auch diesen Code nochmals beschreiben.

Codebeschrieb:

Zeile: 2 bis 11: Der eigentliche splitmix Algorithmus implementiert in Golang.

Zeile: 3: Durch for {} macht man in Golang eine Endlosschleife – ich mag die Syntax.

Zeile: 9: Das vom Pseudozufallsgenerator generierte Ergebnis wird in den Channel gelegt. Ist dieser schon voll wird gewartet.

Zeile: 16: Hier wird der Channel mit den vom Pseudozufallsgenerator generierten Zufallszahlen und einer Kapazität von 10 erstellt.

Zeile: 17: Der Zufallsgenerator wird als Goroutine mit einem Seed von 0 gestartet. Der Result Channel muss übergeben werden, da ich mich in diesem Beispiel entschieden habe keine lokale Funktion zu erstellen. So unterscheiden sich die Beispiele etwas und man hat verschiedene Konzepte kennengelernt.

Zeile: 18 bis 21: Hier werden 100 Zufallszahlen aus dem Channel abgeholt und ausgegeben. Danach wird das Programm mitsamt noch laufender Goroutine terminiert, was in Golang erlaubt ist.

6.3 C#

6.3.1 Asynchroner HTTP-Request

6.3.1.1 AsyncLargeFileDownload

Eines der besten parallelen Programmierkonzepte in C# ist die asynchrone Programmierung mit async und await. Die asynchrone Programmierung ist eine Abstraktion der parallelen Programmierung. Der Code wird sequenziell geschrieben ohne sich gross Gedanken über parallele Programmierung zu machen. Das Einzige, um das sich der Programmierer kümmern muss, ist das Markieren von Abhängigkeiten durch await. Der Compiler kümmert sich um die Parallelisierung.

Während andere parallele Programmiermodelle sich auf parallelisierbare Bereiche fokussieren, fokussiert sich asynchrone Programmierung auf die eigentlichen Aktionen, die Tasks. Durch asynchrone Programmierung ist es möglich, mit nur sehr wenig Code parallel zu programmieren. Sprachen wie C# haben sehr viele asynchrone Funktionen schon integriert, die nur noch genutzt werden müssen. Die Fehlerbehandlung kann in der asynchronen Programmierung gleich wie in der seriellen Programmierung erfolgen. Try-Catch Blöcke sind absolut kein Problem.

Await bietet eine nicht blockierende Möglichkeit um einen Task zu starten, sodass bis zum Abschluss dieses Tasks andere Arbeit erledigt werden kann. Wenn der Task fertig ist, wird die Ausführung so fortgesetzt als ob nie etwas dazwischen gemacht wurde. Await sollte erst auf den Task aufgerufen werden, wenn der vom asynchronen Task erzeugte Wert auch wirklich gebraucht wird. So werden dem Compiler die Abhängigkeiten signalisiert.

Wie die meisten parallelen Programmierkonzepte ist auch asynchrone Programmierung nicht perfekt. Es korrekt zu lernen braucht viel Zeit, da es sich fundamental von anderen parallelen Programmierkonzepten unterscheidet. Obwohl asynchrone Programmierung wie sequenzieller Code aussieht, ist er dies nicht. Falsche Nutzung kann in race conditions und Deadlocks enden, welche schwerer zu debuggen sind als bei gewöhnlicher paralleler Programmierung. Der vom Compiler durch asynchrone Programmierung automatisch generierte Code ist oft sehr hässlich und ineffizient, was man vor allem durch die höhere Anzahl Speicherzuweisung und Säuberungen sieht. Asynchrone Programmierung ist eines der besten parallelen Program-

mierkonzepte und in den meisten Fällen überwiegen die Vorteile und eine Nutzung ist absolut zu empfehlen.

Entscheidet man sich, für sein Projekt asynchrone Programmierung zu nutzen, wird sie sich durch den ganzen Code verbreiten, da await asynchrone Funktionen voraussetzt und selbst nur in asynchronen Funktionen vorkommen kann. Dies ist gut, da so ein konsistentes paralleles Programmierkonzept für das ganze Projekt verwendet wird. Aber vor allem bei grösseren bereits existierenden Projekten mit viel legacy Code ist es deswegen ratsam, auf andere parallele Programmierkonzepte zu setzen, ausser man hat die Zeit, die asynchrone Programmierung über das ganze Projekt anzuwenden und alle so entstehende Probleme zu lösen.

Weiterführende Wissensquelle:

Asynchronous programming with async and await. [29]

```
1   static async Task Main(string[] args) {
2       var result = Channel.CreateBounded<byte[]>(20);
3       var task = Task.Run(async () => {
5           using (HttpClient client = new HttpClient()) {
6               const string url = "http://www.nicobosshard.ch/Hi.html";
7               using (HttpResponseMessage response = await client.GetAsync(url, HttpCompletionOption.ResponseHeadersRead))
8               using (Stream streamToReadFrom = await response.Content.ReadAsStreamAsync()) {
9                   int count = 0;
10                  do {
11                      var buffer = new byte[10];
12                      count = await streamToReadFrom.ReadAsync(buffer, 0, 10);
14                      _ = result.Writer.WriteAsync(buffer[0..count]);
15                  } while (count > 0);
16              }
17          }
19      });
21      System.Threading.Thread.Sleep(100);
23      while (true) {
24          var chunk = await result.Reader.ReadAsync();
26          if (chunk.Length == 0) break;
27          Console.WriteLine(Encoding.UTF8.GetString(chunk));
28      }
29      await task;
30  }
```

[29] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/ (Wagner, 2020)

Codebeschrieb:

Zeile: 1: Sogar das Main ist jetzt async – Dies zeigt schön, wie sich asynchrone Programmierung zwingend über das ganze Projekt hinweg ausbreitet.

Zeile: 2: C# hat Channels! Hier wird ein Channel mit bis zu 20 byte-Array Objekten erstellt.

Zeile: 3: Hier wird der asynchrone Code gestartet. Im asynchronen Code selbst werden Tasks durch async/await selbst verwaltet.

Zeile: 5: Using ist die C# Art um Dinge sauber abzuräumen. Egal ob der Code erfolgreich ist oder ein Fehler auftritt. Dies ist viel schöner als das finally in java. Das finally in Java geht oftmals vergessen und falls im finally selbst ein Fehler ausgelöst wird, existiert oftmals kein weiteres try-catch-finally um dennoch sauber aufzuräumen. Mit using hat C# eine saubere Lösung für dieses Problem.

Zeile: 7: Hier wird ein asynchroner HTTP GET request ausgeführt. Mit await wird signalisiert, dass das Resultat auf der nächsten Zeile benötigt wird. Die Codeausführung wird nach dem Fertigstellen dieses Tasks auf dieser Zeile mit dem Zustand vor dem ausführen des Tasks fortgesetzt.

Zeile: 8: Hier wird eine asynchrone Serialisierung der durch den GET Request erhaltenen Daten durchgeführt und ein serialisierter Datenstrom zurückgegeben. Durch await wird signalisiert, dass dieser Datenstrom auch gleich benötigt wird.

Zeile: 12: Hier werden asynchron 10 bytes vom serialisierten HTTP Datenstrom gelesen. Durch await wird signalisiert, dass diese Daten auch gleich benötigt werden

Zeile: 14: Hier wird das Resultat asynchron in den Channel, welcher die Daten dem Main Thread sendet, geschrieben. Da es keinen Grund gibt, sofort auf diese Aktion zu warten wird kein await davorgesetzt. Auch wollen wir nicht später per await auf diese Aktion warten, weswegen wir den Task nicht in eine echte Variable, sondern in eine discard Variable speichern. Dies könnte auch weggelassen werden, was aber in einer CS4014 Warnung enden würde, da die Methode nicht ohne auf diese Zeile zu warten fortgesetzt wird, was in der Regel ungewollt ist, falls man den Task nicht explizit in einer Variablen speichert. Achtung: Dadurch wird die Reihenfolge in welcher die Chunks in den Channel geschrieben werden undefiniert. Falls eine definierte Reihenfolge benötigt wird, soll hier await verwendet werden.

Zeile: 21: Es wird 100 ms gewartet um parallel zum asynchronen Task laufende Arbeit zu simulieren.

Zeile: 23 bis 28: Die Nachrichten werden vom Channel empfangen. Dies geschieht durch await da die Nachricht sofort benötigt wird. Durch das Senden einer leeren Nachricht signalisiert der asynchrone Task dem Main Thread, dass er fertig ist. Die erhaltenen Nachrichten werden UTF-8 encodiert ausgegeben.

Zeile: 29: Durch await wird hier auf den asynchronen Task gewartet. Eigentlich sollte er ja sowieso schon fertig sein, da er dies über den Channel dem Main Thread schon mitgeteilt hat aber asynchrone Tasks ohne await sind nicht schön und führen zu eine CS1998 Warnung. Hier sieht man auch die mit asynchroner Programmierung bestehende Gefahr von Deadlocks. Hätte der Main Thread nichts aus dem Channel genommen und der HTTP Stream den Channel überfüllt, und würde anstelle der discard Variable mit await die Leerung des Channels durch einen Consumer warten, würde hier ein Deadlock entstehen. Jedoch sieht man auch, dass man sehr viel falsch machen muss um mit asynchroner Programmierung tatsächlich ein Deadlock zu erzeugen und asynchrone Programmierung das Deadlock Risiko somit sehr verringert.

6.3.2 Pseudozufallszahlengenerator

6.3.2.1 Continuations

Durch Continuations ist es möglich, Funktionen an beliebiger Stelle über yield return zu unterbrechen und der übergeordneten Funktion einen Wert zu liefern. Zum Kapitel über Continuation in Java Project Loom wurde dieses Konzept schon erklärt und ich werde hier deswegen nur die Unterschiede Zu Java mit Project Loom genauer erläutern. Der Hauptunterschied ist, dass anders als bei Continuations in Project Loom C# per yield return Werte an die übergeordnete Funktion zurückliefern kann. Deswegen sind keine geteilten Variablen mehr notwendig. Ein weiterer Unterschied ist, dass sich Continuations in C# auf Funktionen und nicht Bereiche beziehen. Die Rückgabe von yield return ist vom Objekt IEnumerable, was schon die Verwandtschaft mit Enumeratoren zeigt. Dadurch lässt sich durch foreach über solche Continuations loopen. Allem in allem sind Continuations in C# deswegen etwas schöner als in Java mit Project Loom, dennoch sind sie von dem was damit machbar ist etwa gleichwertig.

Dieses Beispiel zeigt asynchrone Programmierung kombiniert mit Continuations. Was asynchrone Programmierung ist und wie diese funktioniert wurde im letzten Kapitel erklärt, deswegen wird Verständnis über asynchrone Programmierung in diesem Kapitel

vorausgesetzt. Der Grund, weshalb Continuations ohne asynchrone Programmierung für parallele Programmierung nutzlos sind, ist dass diese anders als Goroutinen keine Parallelität integriert haben. Deswegen nenne ich sie auch Continuations und nicht Coroutinen, obwohl technisch gesehen Parallelität keine Anforderung einer Coroutine ist.

In diesem Beispiel wird gezeigt wie asynchron ein Generator Manager gestartet wird, welcher über Continuations mit dem eigentlichen Generator kommuniziert. Durch die Trennung des Generator Managers, dessen Hauptaufgabe die Kommunikation mit dem Main Thread und dem eigentlichen Generator ist, wird der Code viel leserlicher da eine Funktion nur noch eine einzige Aufgabe hat und somit das Clean Code Konzept *Separation of Concerns* einhält.

```
1   private static IEnumerable<ulong> prng(ulong seed) {
3       while (true) {
4           seed += 0x9E3779B97F4A7C15;
5           ulong z = seed;
6           z = (z ^ (z >> 30)) * 0xBF58476D1CE4E5B9;
7           z = (z ^ (z >> 27)) * 0x94D049BB133111EB;
9           yield return (z ^ (z >> 31)) >> 31;
10      }
11  }
```

Codebeschrieb:

Zeile: 1: Die Funktion prng ist vom Typ IEnumerable<ulong> so dass yield die generierte Zufallszahl vom Typ long der übergeordneten Funktion zurückliefern kann.

Zeile: 3 bis 10: Der eigentliche Pseudozufallszahlgenerator basierend auf dem splitmix PRNG von Arvid Gerstmann, welcher ich von C++ auf C# portiert habe.

Zeile: 9: Mit yield return wird die gerade eben generierte Zufallszahl der übergeordneten Funktion zurückgeliefert und die aktuelle Funktion pausiert. Funktionen können also ohne OS-Scheduler angehalten werden. Dies ist eine riesige Leistung von C#.

```
13  public static async Task Main() {
14      var result = Channel.CreateBounded<ulong>(20);
15      _ = Task.Run(async () => {
16            foreach (var randomNumber in prng(0)) {
17                await result.Writer.WriteAsync(randomNumber);
20            }
22        });
24      System.Threading.Thread.Sleep(100);
26      for (int i = 0; i < 100; ++i) {
27          var randomNumber = await result.Reader.ReadAsync();
29      }
30  }
```

Codebeschrieb:

Zeile: 13: Wie schon im Kapitel über asynchrone Programmierung erwähnt, muss auch die Main Methode asynchron sei, da sich asynchrone Programmierung über das ganze Projekt hinweg ausbreitet.

Zeile: 14: Hier wird ein Channel mit bis zu 20 byte-Array Objekten erstellt.

Zeile: 15: Der Generator Manager und Generator sollen in einem eigenen asynchronen Task ausgeführt werden. Wir speichern diesen in einer discard Variable, da wir niemals auf ihn warten wollen, da er uns für den Rest des Programms Pseudozufallszahlen liefern soll.

Zeile: 16: Da eine Continuation ein Objekt vom Typ IEnumerable zurückliefert, kann mit foreach über diesen endlosen Pseudozufallsgenerator geloopt werden.

Zeile: 17: Hier wird das Resultat asynchron in den Channel, welcher die Daten dem Main Thread sendet, geschrieben. Da es keinen Sinn macht, weitere Zufallszahlen zu generieren ohne die Bestehenden zu speichern, wird hier durch await gewartet.

Zeile: 24: Es wird 100 ms gewartet um parallel zum asynchronen Task laufende Arbeit zu simulieren.

Zeile: 26: Wir wollen 100 Zufallszahle, also loopen wir bis 100.

Zeile: 27: Die Nachrichten werden vom Channel empfangen. Dies geschieht durch await, da die Nachricht sofort benötigt wird.

Hinweis: Ein Beispiel mit nur Tasks ohne Continuations, ohne async und mit ConcurrentQueue anstelle von Channels das Codebeispiel TasksRandom anzuschauen, habe ich in meiner Arbeit nicht genauer beschrieben, da sich darin nicht wirklich etwas Neues/Interessantes befindet.

6.4 Erlang

Erlang ist eine funktionale, dynamisch typisierte, von Prolog beeinflusste Programmiersprache. Sie läuft auf der Middleware Erlang/OTP wobei OTP für *The Open Telecom Platform* steht. Sie wurde im Jahr 1987 veröffentlicht und ursprünglich für die Telekomindustrie geschaffen. Die Anforderungen an die Programmiersprache waren:

- Parallelität
- Verfügbarkeit
- Fehlertoleranz
- Änderungen zur Laufzeit

Dies sind genau die im Kapitel über Java Akka besprochenen Vorteile des Actor Models. Somit ist nur schon durch die Anforderung an die Programmiersprache klar, dass das einzige für diese Programmiersprache in Frage kommende Modell Actor Concurrancy ist. Würde man eine Programmiersprache mit gleichen Anforderungen entwickeln, würde man auch heute noch auf das Actor Model setzen, da bisher keine besseren Alternativen bekannt sind.

Die mit Erlang geschriebenen Telefonsysteme wurden in der Regel hierarchisch aufgebaut. Bei einem Ausfall fielen nur gewisse Äste dieses hierarchischen Baumes aus und es kam in der Regel nie zu einem grösseren Ausfall. Aktoren, welche abgestürzt sind oder sich unerwartet verhalten, werden über Selbstheilungsmechanismen ihrer übergeordneten Aktoren automatisch neu gestartet und deren Verfügbarkeit wird in wenigen Sekunden wiederhergestellt. Geht beispielsweise eine Maschine defekt, welche mehrere Aktoren hostet, können diese einfach über Selbstheilung auf einer anderen Maschine gespawnt werden. Es werden also maximal einige wenige Telefonanrufe abgebrochen, aber das System als solches wird kaum beeinträchtigt. Da jeder Aktor unabhängig von den anderen Aktoren läuft, kann beliebig skaliert werden und es ist eine beinahe beliebige Parallelität möglich. Dadurch dass Aktoren voneinander unabhängig sind, können auch Teile des Systems beliebig aktualisiert werden, ohne dass das ganze System neu gestartet oder durch ein neues System ersetzt werden muss. Solche live Updates sind essentiell für einen Telefon Service, welcher immer verfügbar sein muss.

Bis heute wird Erlang als eine der wenigen funktionalen Programmiersprachen aktiv in der Industrie eingesetzt. Viele Softwaresysteme, welche wir täglich nutzen sind mindestens

teilweise in Erlang geschrieben. Dazu zählen unter anderem: AnyDesk, Facebook, GitHub, Telekom Deutschland (und viele weitere Telekomunternehmen), WhatsApp und Discord. (Wikipedia, Erlang (Programmiersprache), 2021)

Es ist erstaunlich wie das Actor Modell schon so lange bekannt ist aber bis anhin nur sehr wenig Verbreitung fand, da es hauptsächlich von funktionalen Programmiersprachen unterstützt wurde. Erst mit der Verbreitung von Message Passing Interface in Hochleistungsrechnern wurden bekannte prozedurale Programmiersprachen mit einem ähnlichen Modell ausgestattet. Mittlerweile entstehen für die meisten Programmiersprachen recht gute Actor Concurancy Bibliotheken wie beispielsweise das zuvor besprochene Java Akka. Dennoch werden diese leider viel zu wenig genutzt. Wahrscheinlich ist das so, weil sich die wenigsten Programmierer den Vorteilen des Actor Models bewusst sind, die dadurch entstehende Komplexität fürchten oder einfach keine Zeit haben, viel Zeit in ein ihnen noch unbekanntes paralleles Programiermodell zu stecken.

Ich werde in diesem Kapitel nicht weiter auf die Funktionsweise sowie Vor- und Nachteile des Actor Models eingehen, da dies schon im Kapitel über Java Akka besprochen und im Kapitel über Goroutinen mit diesen verglichen wurde.

Weiterführende Wissensquelle:

Erlang (Programmiersprache), Wikipedia. [30]

6.4.1 Count

Durch Actor concurrancy ist es einfach einen threadsicheren Zähler zu implementieren.

```
4   send_msgs(_, 0) -> true;
5   send_msgs(Counter,
6             Count) ->
7       Counter ! {inc, 1},
8       send_msgs(Counter, Count - 1).
9   run() ->
11      Counter = spawn(counter, counter, [0]),
12      send_msgs(Counter, 1000000),
13      Counter.
```

[30] https://de.wikipedia.org/wiki/Erlang_(Programmiersprache) (Wikipedia, Erlang (Programmiersprache), 2021)

Codebeschrieb:

Zeile: 4 bis 8: send_msgs Rekursion welche 1'000'000 Nachrichten an den «counter»-Actor sendet. Auf **Zeile 4** befindet sich die Abbruchbedingung und auf **Zeile 8** der rekursive Aufruf.

Zeile: 7: Es wird eine Inkrement Message mit an den «counter»-Actor gesendet.

Zeile: 9: Die Hauptfunktion welche beim Ausführen des Erlang-Programms gestartet wird.

Zeile: 11: Es wird ein neuer «counter»-Actor gespawnt.

Zeile: 13: Die oben beschriebene send_msgs rekrsion welche 1000000 Nachrichten an den «counter»-Actor sendet wird aufgerufen. Der zuvor gespawnte «counter»-Actor wird als Argument übergeben.

```
1  -module(counter).
2  -export([counter/1]).
3  counter(Value) ->
4      receive
5          value -> io:fwrite("Done! Counter is ~w!~n", [Value]);
6          {inc, Amount} -> counter(Value + Amount)
7      end.
```

Codebeschrieb:

Zeile: 1: Module können als Aktoren betrachtet werden. Somit ist dies der «counter»-Actor. Der Modulname wird zum Spawnen von Aktoren verwendet.

Zeile: 2: Durch «export» werden die für andere Module sichtbare Funktionen definiert. definiert die für andere Module sichtbare Funktionen. In diesem Fall ist die counter Funktion sichtbar, sodass Inkrement Nachrichten diesem Aktor gesendet werden können.

Zeile: 7: counter ist eine endlos rekursive Funktion, welche Nachrichten vom Hauptaktor empfängt.

Zeile: 4 bis 7: Alle sich in der Mailbox befindenden empfangenen Nachrichten werden versucht an die sich in diesem receive-Block befindenden Funktionen zu matchen.

Zeile: 5: Hier werden die empfangenen Nachrichten ausgegeben.

Zeile: 6: Hier werden die Nachrichten empfangen und die Rekursion mit dem neuen Zählerstand ausgeführt.

6.4.2 Pseudozufallszahlengenerator

Der splitmix Pseudozufallsgenerator von Arvid Gerstmann lässt sich ähnlich wie in Java Akka auch in Erlang implementieren. Während dazu in Java Akka jedoch relativ viel Code notwendig war, ist dies in Erlang in wenigen Zeilen Code realisierbar. Dies weil Erlang speziell für Actor Concurrancy entwickelt wurde. Es zeigt auch, dass obwohl Java Akka eine sehr gute Bibliothek ist, eine native Sprachunterstützung in der Regel einer Bibliothek vorzuziehen ist. Dies da so der Syntax viel einfacher gestaltet werden kann. So wird dem Programmierer viel Programmierarbeit durch das Design der Programmiersprache abgenommen. Auch führt eine native Sprachunterstützung durch bessere Syntax einfacher zu lesebarem Code.

Da Erlang eine funktionale Programmiersprache ist und somit keine mutierbaren Variablen hat, können für den splitmix-Algorithmus keine Variablen wiederverwendet werden. Ob dies positive oder negative Auswirkung auf die Lesbarkeit des Codes hat kann individuell entschieden werden.

```
4   start(_StartType, _StartArgs) ->
5       spawn(prng, prng, [0]),
```

Codebeschrieb:

Zeile: 4: Die «start» Methode wird beim Starten der Erlang Applikation automatisch ausgeführt

Zeile: 5: Das Einzige was die «start» Methode macht ist ein «prng»-Actor zu spawnen. Dies um ähnlich wie das Java Akka Beispiel zu sein, wo auch ein «generator»-Actor und «consumer»-Actor verwendet wird. In diesem Beispiel entspricht der «generator»-Actor vom Java Akka Beispiel dem «prng»-Actor. Der «MyRandom»-actor ist nicht notwendig, da der Hauptaktor den prng-Actor schon wie gewollt spawnen kann welcher den «consumer»-Actor spawnen kann. Somit ist kein übergeordneter Aktor mehr erforderlich wie der «MyRandom»-Actor im Java Akka Beispiel.

```
1   -module(prng).
2   -export([prng/1]).
3   send_msgs(_, 0, _) -> true;
4   send_msgs(Consumer,
5             Count,
6             Seed) ->
7       SeedNew = Seed + 1140071481932319848 5,
8       Z1 = (SeedNew bxor (SeedNew bsr 30)) * 13787848793156543929,
9       Z2 = (Z1 bxor (Z1 bsr 27)) * 10723151780598845931,
10      Result = (Z2 bxor (Z2 bsr 31)) bsr 31,
11      Consumer ! {rand, Result},
12      send_msgs(Consumer, Count - 1, SeedNew).
13  prng(Seed) ->
15      Consumer = spawn(consumer, consumer, []),
16      send_msgs(Consumer, 100, Seed).
```

Codebeschrieb:

Zeile: 1: Dies ist der «prng»-Actor.

Zeile: 2: Die Funktion prng/1 ist für andere Aktoren sichtbar.

Zeile: 3 bis 12: Die send_msgs ist eine rekursive Funktion, welche eine Aktorreferenz zum «consumer»-Actor hat, die Anzahl verbleibenden Seeds, welche noch generiert werden müssen sowie den aktuelle Seed als Argument nimmt.

Zeile: 7 bis 10: Hier befindet sich der eigentliche splitmix-Algorithmus. Da Erlang eine funktionale Programmiersprache ist und somit keine veränderbaren Variablen besitzt, wird hier SeedNew, Z1 und Z2 als unveränderbare Variable verwendet

Zeile: 13 bis 16: Im Konstruktor des «prng»-Actors wird ein «consumer»-Actor gespawnt und die rekursive send_msgs Funktion aufgerufen. Dieser wird dem zuvor gespawnten «consumer»-Actor übergeben, 100 als Anzahl zu generierenden Pseudozufallszahlen und der dem «prng»-Actor Konstruktor übergebenen Seed.

```
1  -module(consumer).
2  -export([consumer/0]).
3  consumer() ->
4      receive
5          {rand, RandomNumber} -> io:fwrite("Recieving ~w!~n", [RandomNumber]), consumer()
6      end.
```

Codebeschrieb:

Zeile: 1: Dies ist der «consumer»-Actor.

Zeile: 2: Die Funktion consumer/0 ist für andere Aktoren sichtbar

Zeile: 4 bis 6: Alle sich in der Mailbox befindenden empfangenen Nachrichten werden versucht an die sich in diesem receive-Block befindenden Funktionen zu matchen.

Zeile: 5: Hier werden die vom «prng»-Actor empfangenden Pseudozufallszahlen empfangen und in der Konsole ausgegeben.

6.5 Kotlin

Kotlin ist eine statisch typisierte Programmiersprache welche in Java Bytecode kompiliert wird. Eine Kompilierung nach JavaScript oder Maschinencode ist auch möglich. Die erste Version von Kotlin wurde 2011 und die erste stabile Version 2016 veröffentlicht. Da Kotlin auch auf der JVM aufbaut, ist Kotlin interoperabel mit Java Code. Somit können problemlos Java Bibliotheken in Kotlin verwendet werden. Google hat Kotlin als bevorzugte Programmiersprache für Android Appentwicklung erklärt und viele Android Apps werden mittlerweile in Kotlin geschrieben. Kotlin hat viele Vorteile gegenüber Java wie unter anderem Suspending functions, Extension Functions, Nullsicherheit, String Templates mit Ausdrücken, Smart Casts, Lazy Initialization und Operator Overloading.

Während Java noch auf Project Loom wartet, hat Kotlin schon vieles ohne direkte JVM Unterstützung auf Sprachebene implementiert. Obwohl die JVM an sich noch keine Continuations unterstützt, gelang es Kotlin diese schon auf Sprachebene zu implementieren. Continuations sind der zentrale Baustein von Coroutinen. Während Continuations technisch gesehen ausreichen um Coroutinen zu implementieren, wird für eine sinnvolle Anwendung in der Praxis jedoch auch Fibers benötigt. Da es ohne JVM Änderungen unmöglich ist Fibers zu implementieren, wird im Hintergrund ein ForkJoinPool verwendet. Ein ThreadPool bietet ähnliche Eigenschaften wie Fibers in dem Threads gecached werden. Durch die work stealing Eigenschaft eines ForkJoinPool eignet dieses sich besonders gut um Coroutinen zu implementieren. Dennoch bleibt leider der Nachteil, dass man anders als beispielsweise in Golang nicht hunderttausende von Kotlin Coroutinen gleichzeitig ausführen kann und durch die ganzen Threads ein erheblicher Overhead entsteht. Dies ist jedoch nur eine temporäre Lösung bis es Project Loom in die JVM schafft. Während vieles ohne JVM Änderungen technisch möglich ist, sind Implementierungen auf Sprachebene in der Regel oft nur durch in Kauf nehmen schlechter Performance möglich. Dies sieht man auch an Kotlins Continuations. Der Aufruf jeder suspendable Function ist recht teuer und sollte somit nur falls nötig verwendet werden. Continuations in Projekt Loom hingegen kosten nur etwas falls auch wirklich unterbrochen wird. Dennoch bin ich der Meinung, dass es Sinn macht immer das absolut Neuste in seine Programmiersprache zu integrieren. Die Performance kann nach Implementierung der für gute Performance notwendigen Änderungen in der JVM beispielsweise durch Project Loom mit einem einfachen Update erhöht werden. Für den Programmierer ist es also möglich, Code zu schreiben welcher in Zukunft ohne Arbeit nur durch ein Kotlin Update effizient gemacht

werden kann. Dies ist viel besser als beispielsweise in Java, wo der Programmierer gezwungen wird alte parallele Programmierkonzepte zu verwenden und beim Release von Project Loom seinen ganzen parallel Code manuell aktualisieren müsste. Dennoch finde ich es schade, dass Kotlin als Sprache soweit vor JVM ist. Es wäre sicherlich sinnvoller, wenn JVM sich etwas schneller entwickeln würde. Leider ist dies wegen der Grösse der JVM und der Garantie nach rückwärts Kompatibilität nicht möglich.Für genauere Vergleiche zwischen Project Loom und der Kotlin Implementierung empfehle ich das Kapitel über Project Loom zu lesen, in welchem ich die beiden detaillierter miteinander verglichen habe.

Auch speziell ist, dass in Kotlin Coroutinen nur als Bibliotheken oder durch eigene Implementierung verwendet werden können. Viele Programmierer nutzen die offizielle kotlinx.coroutines Bibliothek um Coroutinen zu verwenden. Da sich Coroutinen jedoch so einfach in Kotlin implementieren lassen, entschied ich mich, die in den offiziellen Kotlin Coroutines Beispielen implementierte Coroutine zu nutzen. Dies da hier Coroutinen in nur 5 kleinen Kotlin Dateien implementiert wurden. So kann eine Coroutine Implementierung für Interessierte betrachtet und verstanden werden, was bei kotlinx.coroutines aufgrund der Grösse dieser Bibliothek eher schwierig wäre. Auch mag ich persönlich den Goroutinen ähnliche Syntax dieser Implementierung.

Falls Interesse an der Kotlin Implementierung von Coroutinen besteht, empfehle ich in den Beispielen den Code unter *lib_coroutines* anzuschauen. *go.kt* ist die Hauptdatei welche nur Funktionen in den anderen aufruft. In *channel.kt* wird gezeigt wie Golang ähnliche Channels in Kotlin implementiert werden können. *Launch.kt*, *pool.kt* und *runBlocking.kt* welche die eigentlichen Coroutinen implementieren, lassen sich zusammen problemlos in weniger als 100 Zeilen Code schreiben.

Weiterführende Wissensquellen:

Kotlin (Programmiersprache), Wikipedia. [31]
Kotlin vs Java. [32]
Kotlin / kotlinx.coroutines, GitHub. [33]
Kotlin/coroutines-examples, GitHub. [34]

---

[31] https://de.wikipedia.org/wiki/Kotlin_(Programmiersprache) (Wikipedia, Kotlin (Programmiersprache), 2021)

[32] https://www.spaceotechnologies.com/kotlin-vs-java/ (Patolia, 2021)

[33] https://github.com/Kotlin/kotlinx.coroutines (GitHub, Kotlin / kotlinx.coroutines, 2021)

[34] https://github.com/Kotlin/coroutines-examples (GitHub, Kotlin/coroutines-examples, 2021)

6.5.1 Asynchroner HTTP-Request

Die Kotlin Coroutinen Beispiele sind sehr ähnlich zu den Goroutinen Golang Beispielen. Die Erklärungen über die Beispiele an sich wurden hier absichtlich kurzgehalten. Ich empfehle das Kapitel zu Goroutinen in Golang zuerst zu lesen.

6.5.1.1 Coroutinen

Wie schon in der Einleitung zu Kotlin erklärt wurde, sind Coroutinen in Kotlin möglich. Dieses Beispiel zeigt eine einfache Kotlin Coroutine. Zur Kommunikation zwischen Coroutinen und dem Main Thread werden Channels verwendet.

```
1  suspend fun httpGetSingle(result: SendChannel<String>) {
2      result.send(URL("http://www.nicobosshard.ch/Hi.html").readText())
3  }
4  suspend fun coroutines(): String {
5      val result = Channel<String>()
6      go { httpGetSingle(result) }
7      return(result.receive())
8  }
```

Codebeschrieb:

Zeile: 1 bis 3: Dies ist eine Corourtine. Dies ist auch der Grund, weshalb hier eine unterbrechbare Funktion erstellt wird. Den für das Resultat zu verwendenden Channel wird der Coroutine als Argument übergeben.

Zeile: 2: HTML-Daten werden heruntergeladen und in den result Channel geschrieben.

Zeile: 4 bis 8: Dies ist die Hauptfunktion. Da eine unterbrechbare Funktion nur von einer unterbrechbaren Funktion aufgerufen werden kann, ist diese auch unterbrechbar.

Zeile: 5: Damit die unterbrechbare Funktion mit der Hauptfunktion kommunizieren kann, wird hier ein Channel erstellt welcher String Objekte akzeptiert. Dieser kann in diesem Beispiel eine beliebige Grösse annehmen.

Zeile: 6: Hier wird die Coroutine gestartet. Da sich die verwendete Coroutinen-implementierung sehr an Goroutinen anlehnt wird ähnlich wie in Golang der go-Befehl verwendet.

Zeile: 7: Hier wird das von der Coroutine durch den result-Channel erhaltene Resultat gelesen. Ist die Coroutine zur Zeit der Ausführung dieser Zeile noch nicht fertig so wird hier auf die Coroutine gewartet.

6.5.1.2 Mehrere Coroutinen

Auch mehrere parallell ausgeführte Kotlin Coroutinen sind analog zu Goroutinen in Golang möglich. Als Guard, um die Anzahl parallellaufender Coroutinen zu limitieren, werden Semaphoren verwendet.

```
1   suspend fun httpGetMultible(result: SendChannel<String>, guard: Semaphore) {
2       result.send(URL("http://www.nicobosshard.ch/Hi.html").readText())
3       guard.release()
4   }
5   suspend fun multibleCoroutines() {
6       val tasks = 10
7       val maxCoroutines = 4
8       val guard = Semaphore(maxCoroutines)
9       val result = Channel<String>(tasks)
10      for (i in 1..tasks) {
11          guard.acquireUninterruptibly()
12          go { httpGetMultible(result) }
13      }
14      for (i in 0..tasks-1) {
15          var result: String = result.receive()
16          println("$i: $result")
17      }
18  }
```

Codebeschrieb:

Zeile: 1 bis 4: Die Coroutine ist gleich wie im letzten Beispiel ausser, dass am Ende die Semaphore wieder freigegeben werden muss. Den für das Resultat zu verwendenden Channel sowie die als Guard benutze Semaphore wird der Coroutine als Argument übergeben.

Zeile: 5 bis 18: Auch an der Hauptfunktion an sich änderte sich nichts.

Zeile: 6: Wir werden 10 Tasks starten.

Zeile: 7: Davon können 4 parallel zueinander ablaufen.

Zeile: 8: Um zu verhindern, dass zu viele Coroutinen gestartet werden wird ein Guard verwendet. Die Guard Datenstruktur von Golang ist äquivalent zu einer Semaphore aber durch Channels realisiert. Da Kotlin Semaphoren hat, wird hier eine Semaphore verwendet.

Zeile: 9: Wie schon im letzten Beispiel wird auch hier ein Channel verwendet um Daten zwischen den Coroutinen und dem Main Thread auszutauschen. Anders als im letzten Beispiel ist dieser nun aber auf die Anzahl Tasks limitiert. Dies spielt jedoch nicht wirklich eine Rolle.

Zeile: 10 bis 13: Durch diesen for loop werden 10 Tasks in Coroutinen gestartet.

Zeile: 11: Vor dem Starten einer Goroutine muss die Semaphore erhöht werden.

Zeile: 12: Hier wird die Goroutine gestartet.

Zeile: 14 bis 17: Nach dem Ausführen der Coroutinen wollen wir diese ausgeben. Dazu nutzen wir diesen for-Loop.

Zeile: 15: Hier wird das von der Coroutine durch den result-Channel erhaltene Resultat gelesen. Ist die Coroutine zur Zeit der Ausführung dieser Zeile noch nicht fertig so wird hier auf die Coroutine gewartet.

Zeile: 16: Das so erhaltene Reultat wird ausgegeben.

6.5.1.3 Mehrere blockierende Coroutinen

Analog zur WaitGroup in Golang existiert in dieser Coroutinen Implementierung mainBlocking um vor Weiterführung des Programms auf das Abschliessen aller Coroutinen zu warten. Für weitere Informationen über WaitGroups empfehle ich das Kapitel über WaitGroups in Golang zu lesen.

```
5   suspend fun multibleCoroutinesBlocking() {
10      mainBlocking {
11          for (i in 1..tasks) {
12              guard.acquireUninterruptibly()
13              go { httpGetMultible(result) }
14          }
15      }
20  }
```

Dieses Beispiel ist ausser mainBlocking gleich wie das vorherige Beispiel. Codeausschnitte, welche unabhängig von mainBlocking sind, wurden deswegen rausgekürzt.

Codebeschrieb:

Zeile: 10 bis 15: Es wird gewartet bis alle in diesem Block gestarteten Coroutinen abgeschlossen sind bevor dieser Block verlassen wird. So ist sichergestellt, dass alle die Funktionen, welche die von den Coroutinen erzeugten Dateien weiterverarbeiten, nicht durch nicht abgeschlossene Coroutinen blockiert werden. Dadurch verringert sich die Komplexität. Durch diesen manuelle Pipelinestall, kommt es in der Regel aber auch zu Performance Verlusten. mainBlocking in Kotlin sollten also wie WaitGroups in Golang nur falls unbedingt nötig eingesetzt werden. Es sollte eingesetzt werden, wenn es durch IO-Zugriffe zu race conditions kommt, oder der empfangene Thread nicht blockiert werden darf, oder die Komplexität ohne mainBlocking zu gross werden würde. oder die Komplexität des Systems zu gross werden würde.

Zeile: 11 bis 13: An der for-Schleife um die Coroutinen zu starten hat sich im Vergleich zum letzten Beispiel nichts geändert.

6.5.2 Pseudozufallszahlengenerator

Wie schon für Golang gezeigt, lassen sich durch Coroutinen schöne Pseudozufallsgeneratoren schreiben. Wie schon bei allen anderen Programmiersprachen wurde auch hier den C++ splitmix PRNG Algorithmus von Arvid Gerstmann nach Kotlin portiert. Da Kotlin anders als Java keine Probleme mit 64-bit vorzeichenlosen Ganzzahlen hat, ging dies problemlos.

6.5.2.1 Coroutinen

```
2   suspend fun random(seed_arg: ULong, c: SendChannel<ULong>) {
3       var seed = seed_arg;
4       while (true) {
5           seed += 0x9E3779B97F4A7C15u
6           var z = seed
7           z = (z xor (z shr 30)) * 0xBF58476D1CE4E5B9u
8           z = (z xor (z shr 27)) * 0x94D049BB133111EBu
10          c.send((z xor (z shr 31)) shr 31)
11      }
12  }
```

Codebeschrieb:

Zeile: 2 bis 12: Die unterbrechbare Funktion ist der eigentliche Pseudozufallsgenerator implementiert als Kotlin Coroutine. Als Argument werden Seed sowie den für das Resultat zu verwendenden Channel übergeben. Da diese Funktion eine Endlosschleife ohne Abbruchbedingung enthält terminiert sie erst beim Programmende.

Zeile: 3 bis 11: Dies ist der C++ splitmix PRNG Algorithmus von Arvid Gerstmann portiert nach Kotlin. Man sieht auch schön wie Kotlin anders als Java keine Probleme mit vorzeichenlosen 64-bit Ganzzahlen hat.

Zeile: 10: Hier wird jede vom Zufallsgenerator erhaltene Pseudozufallszahl in den Channel geschrieben. Ist der Channel voll wird hier auf einen Konsumenten gewartet.

```
14  suspend fun coroutinesRandom() {
15      val cache = 10
16      val tasks = 100
17      val c = Channel<ULong>(cache)
18      go { random(0u, c) }
19      for (i in 0..tasks) {
20          println(c.receive())
21      }
23  }
```

Codebeschrieb:

Zeile: 15 und 17: Es wird ein Kanal mit einer Kapazität von 10 erstellt, dies bedeutet, dass immer 10 Zufallszahlen vorausberechnet werden.

Zeile: 16: Wir wollen 100 Zufallszahlen generieren lassen.

Zeile: 18: Da nur ein Pseudozufallsgenerator benötigt wird, muss auch nur eine Coroutine gestartet werden. Dies hat auch den Vorteil, dass dadurch in diesem Beispiel der Nachteil von Kotlin wegen den fehlenden Fibers nicht wirklich eine Rolle spielt da nur ein Thread erstellt werden muss. Als Parameter wird der Seed sowie der für die generierten Zufallszahlen zu verwendenden Channel erstellt.

Zeile: 19 bis 21: In diesem Loop werden 100 in der Coroutine generierte Zufallszahlen ausgegeben.

Zeile 20 wartet falls der Channel der generierten Zufallszahlen leer ist.

6.5.2.2 Coroutinen mit Yield

Dieses Codebeispiel unterscheidet sich von dem vorherigen indem die Coroutine durch Continuations in dem eigentlichen Generator und dem mit dem Main Thread kommunizierenden Teil getrennt wurde. Dadurch ergibt sich eine schöne Trennung was die Lesbarkeit des Codes verbessert. Das Clean Code Prinzip *separation of concerns* wird eingehalten.

```
2   fun randomYield(seed_arg: ULong) = sequence {
4       while (true) {
10          yield((z xor (z shr 31)) shr 31)
11      }
12  }
13
14  suspend fun randomYieldCaller(seed_arg: ULong, c: SendChannel<ULong>) {
15      randomYield(seed_arg).forEach {
16          c.send(it)
17      }
18  }
```

Da dieses Beispiel sehr nahe dem vorherigen Beispiel ist, wurde der Main Thread Teil, sowie alle irrelevanten Teile des Pseudozufallsgenerators rausgekürzt.

Codebeschrieb:

Zeile: 2 bis 12: In dieser Funktion befindet sich der eigentlich Pseudozufallsgenerator. Als Argument wird der Seed erwartet. Da sie selbst keine weiteren unterbrechbaren Funktionen mehr aufruft ist das suspend Keyword nicht notwendig. Da diese Funktion eine Endlosschleife ohne Abbruchbedingung enthält terminiert sie erst beim Programmende.

Zeile: 10: Durch das yield-Keyword wird der aktuelle Zustand gespeichert und die Kontrolle zurück an den randomYieldCaller übergeben.

Zeile: 14 bis 18: In dieser Funktion werden die vom Pseudozufallsgenerator generierten Pseudozufallszahlen erhalten und über den als Argument übergebene Channel dem Main Thread gesendet. Das seed Argument wird direkt an die randomYield-Funktion weitergegeben.

Zeile: 15 bis 17: Über eine Continuation kann wie über ein Iterable Object geloopt werden. Da randomYield eine Endlosschleife ohne Abbruchbedingung hat, wird diese Iteration auch niemals vor Programmende enden.

Zeile: 16: Die von randomYield durch yield enthaltenen werden hier in den Channel geschrieben, um diese dem Main Thread zu senden.

6.5.3 Vergleich der Programmiermethoden und Sprachen

Vergleich zwischen Pseudozufallszahlengenerator - Code in Golang (links) und Kotlin (rechts).

```go
//Basieren auf C++ splitmix PRNG von Arvid Gerstmann.
func random(seed uint64, result chan uint64) {
    for {
        seed += 0x9E3779B97F4A7C15
        var z uint64 = seed
        z = (z ^ (z >> 30)) * 0xBF58476D1CE4E5B9
        z = (z ^ (z >> 27)) * 0x94D049BB133111EB
        fmt.Println("Generated!")
        result <- (z ^ (z >> 31)) >> 31
    }
}

func main() {
    cache := 10
    tasks := 100
    result := make(chan uint64, cache)
    go random(1, result)
    for i := 0; i < tasks; i++ {
        value := <-result
        fmt.Println(value)
    }
    fmt.Println("Done")
}
```

```kotlin
//Basieren auf C++ splitmix PRNG von Arvid Gerstmann.
suspend fun random(seed_arg: ULong, c: SendChannel<ULong>) {
    var seed = seed_arg;
    while (true) {
        seed += 0x9E3779B97F4A7C15u
        var z = seed
        z = (z xor (z shr 30)) * 0xBF58476D1CE4E5B9u
        z = (z xor (z shr 27)) * 0x94D049BB133111EBu
        println("Generated!")
        c.send((z xor (z shr 31)) shr 31)
    }
}

suspend fun coroutinesRandom() {
    var cache = 10
    var tasks = 100
    val c = Channel<ULong>(cache)
    go { random(1u, c) }
    for (i in 0..tasks) {
        println(c.receive())
    }
    println("Done")
}
```

Wie man sieht wurde durch das Verwenden von Coroutinen-Hilfsfunktionen in Kotlin der Code ähnlich wie der Code in Golang. Die Sprachen unterscheiden sich nur noch etwas in der Syntax. Beide machen exakt dasselbe. Die Programmierung der Nebenläufigkeit bei diesem Beispiel ist die Gleiche. Der Code ist Zeile für Zeile in die jeweils andere Sprache übersetzbar. Technisch gesehen passieren im Hintergrund jedoch unterschiedliche Vorgänge. Beispielsweise leichtgewichtige Threads bei Golang und schwergewichtige, gecachte Threads in Kotlin. Durch die Abstraktion ist für den Programmierer davon jedoch nichts sichtbar.

6.6 Haskell

Haskell ist eine rein funktionale statisch typisierte Programmiersprache welche auf dem Lambda-Kalkül basiert und 1990 veröffentlicht wurde. Als reine funktionale Programmiersprache haben Funktionen keine Nebeneffekte. Haskell kennt keine imperativen Sprachkonzepte und nutzt Monads um Ein- und Ausgaben funktional zu realisieren. Variablen sind unveränderbar. Alle Variablen sind somit Konstanten.

Durch diese Eigenschaften eignet sich Haskell besonders gut für die parallele Programmierung. Ohne veränderbaren Variablen sind race conditions nicht möglich. Alle Datentypen die dennoch etwas ähnliches wie veränderbare Variablen unterstützen sind in der Regel thread sicher. Dies wird entweder durch atomare Operationen und Locks oder Software transactional Memory realisiert. Oftmals hat der Programmierer die Wahl, ob er die eher pessimistische atomare oder gelockten Operation, oder die eher optimistische Software transactional Memory Implementierung einer Datenstruktur nutzen will. Da die Nutzung der auf der Software transactional Memory basierten Datenstrukturen Lock-Free ist, können keine Deadlocks auftreten. Haskell unterstützt natives STM. Dieses ist in der Regel etwa so schnell wie Locks, wenn nicht schneller. Siehe dazu das referenzierte Paper *Lock Free Data Structures using STMs in Haskell*.

Im Vergleich zu Erlang erlaubt Haskell durch Monads wesentlich kompaktere Programmierung. Auch können in Haskell problemlos beliebig viele Aktoren in dieselbe Datei geschrieben werden ohne dass dadurch der Code unübersichtlich wird. Anders als bei Golang bei welchem Channels als Guards verwendet werden müssen, unterstützt Haskell echte Semaphoren, welche als Guards genutzt werden können. Dies ist besonders wichtig da Haskell keine bounded channels kennt und Semaphoren somit die einzig einfache Möglichkeit sind um Guards zu implementieren.

Anders als bei Erlang nutzt Haskell statische Typisierung. So kann beispielsweise mit *:: word64* eine vorzeichenlose 64-bit Ganzzahl definiert werden.

Während Erlang sich mehr auf verteilte Systeme fokussiert, eignet sich Haskell, wenn genau dies nicht benötigt wird und auf einem System durch parallele Programmierung möglichst viel Performance herausholt werden soll. Durch statische Typisierung und direkte Kompilierung in Maschinencode hat Haskell gegenüber Erlang bei nicht verteilten Systemen in der Regel einen

Performancevorteil, wobei aber die run-time Optimierung von Erlang sehr gut ist. Auch benötigt parallele Programmierung in Haskell weniger Code und ist weniger komplex als in Erlang.

Actor Concurrancy ist in Haskell einfach und ähnlich wie in Erlang. Actor Concurancy kann beispielsweise durch *Control.Concurrent.Actor* oder *Control.Concurrent.CHP* realisiert werden. Ich werde mich in dieser Arbeit jedoch auf Actor Concurrancy ähnliche parallele Programmierung durch *Message Passing Channels* fokussieren. Message passing channels ist gleich wie Actor Concurrancy aber anstelle Aktoren werden über Channels kommunizierenden Threads verwendet. Von der Programmierung her, sind diese ähnlich wie Coroutinen. Dennoch haben sie die meisten Vorteile der Actor Concurrancy, da Haskell Nebeneffekte verhindert und Channels unlimitiert gross sein können. Zudem sind alle threads in Haskell Fibers. Durch TChains sind lockfreie Channels möglich. TChans sind anders als normale Channels durch Software transactional Memory implementiert und somit lock free. So ist in Haskell lockfreie Programmierung mit dem coroutinenartigen *Message Passing Channels* möglich. Dies hat alle Vorteile der Actor concurrancy ohne die Komplexität der eigentlichen Actor concurrancy. Auch zeigt dies sehr gut den riesigen Vorteil von Software Transactional Memory implementierten Datenstrukturen gegenüber nicht lockfrei implementierten Datenstrukturen.

Weiterführende Wissensquellen:

Haskell (Programmiersprache), Wikipedia. [35]

Lock Free Data Structures using STM in Haskell. [36]

Control-Concurrent-Aktor. [37]

Haskell for multicores. [38]

[35] https://de.wikipedia.org/wiki/Haskell_(Programmiersprache) (Wikipedia, Haskell (Programmiersprache), 2021)

[36] https://www.microsoft.com/en-us/research/wp-content/uploads/2006/04/2006-flops.pdf (Discolo, Harris, Marlow, Jones, & Singh, 2006)

[37] https://hackage.haskell.org/package/thespian-0.999/docs/Control-Concurrent-Actor.html (Marlow & Simon, 2011)

[38] https://wiki.haskell.org/Haskell_for_multicores#Message_passing_channels (HaskellWiki, 2020)

6.6.1 Count

6.6.1.1 Counter mit Software Transactional Memory

Haskell ist bekannt für seine performanten lockfreien, durch Software transactional Memory realisierten Datenstrukturen. Eine davon ist Control.Concurrent.STM.TMVar. Es ist das durch Software transactional Memory implementierte Äquivalent von dem durch Locks implementierte Control.Concurrent.MVar.

T wurde wie folgt definiert und ist somit ein Synonym von Control.Concurrent.STM:

```
import qualified Control.Concurrent.STM as T
```

```
1   type Counter = T.TMVar Integer
2   newCounter :: Integer -> T.STM (Counter)
3   newCounter value = T.newTMVar value
4   getCounter :: Counter -> T.STM Integer
5   getCounter counter = do
6       value <- T.takeTMVar counter
7       return value
8   incrementCounter :: Counter -> T.STM ()
9   incrementCounter counter = do
10      oldValue <- T.takeTMVar counter
11      let newValue = (oldValue + 1)
12      T.putTMVar counter newValue
13  forkThread :: IO () -> IO (MVar ())
14  forkThread proc = do
15      handle <- newEmptyMVar
16      _ <- forkFinally proc (\_ -> putMVar handle ())
17      return handle
18  main :: IO ()
19  main = do
20      counter <- T.atomically (newCounter 0)
21      threads <- forM [1..1000000] $ \_ -> do
22          forkThread $ T.atomically (incrementCounter counter)
23      mapM_ takeMVar threads
24      value <- T.atomically $ getCounter counter
25      print $ value
```

Codebeschrieb:

Zeile: 1: Durch das «type» Keyword wird «Counter» als Synonym von «T.TMVar Integer» definiert.

Zeile: 2 bis 3: Diese Funktion dient um einen neuen Counter zu erstellen. Das Erstellen des Counters geschieht durch newTMVar. Als Argument wird ein Integer übergeben und als Rückgabewert einem Counter übergeben.

Zeile: 4 bis 7: Diese Funktion dient zum Auslesen des aktuellen Werts des Counters. Das Auslesen des Counters geschieht durch takeTMVar. Als Argument wird ein Counter übergeben und als Rückgabewert wird der ausgelesene Integer übergeben.

Zeile: 8 bis 12: Diese Funktion dient zum Erhöhen des Counters. Dazu wird zuerst der aktuelle Zustand über takeTMVar ausgelesen. Danach wird die so erhaltene Variable um eines erhöht und schliesslich mit putTMVar zurück in den Counter geschrieben.

Zeile: 13 bis 17: Um zu demonstrieren, dass durch Software transactional Memory race conditions korrekt gehandelt werden, müssen viele Fibers erstellt werden. Auch sollte das Programm auf den Abschluss aller Fibers warten bevor das finale Resultat ausgegeben wird. Dazu wurde die Funktion forkThread erstellt welche über forkFinally eine neue Fiber erstellt und durch putMVar den Abschluss der Fiber signalisiert. Dies wäre mit dem Control.Concurrent.Async packet schöner lösbar, ich wollte mich jedoch in dieser Arbeit auf Standardpakte beschränken. Quelle zu diesem Codeabschnitt: (Stackoverflow, 2015)

Zeile: 18 bis 25: Dies ist die Hauptfunktion welche beim Starten des Haskell Programms ausgeführt wird.

Zeile: 20: Hier wird ein neuer Counter erstellt. Durch atomically wird diese Funktion in einer atomaren Transaktion ausgeführt.

Zeile: 21 bis 22: Durch das forM Monad wird ein for-loop erstellt welcher eine Million Mal ausgeführt wird. In diesem werden durch die zuvor erklärte forkThread Funktion eine Million Fibers erstellt welche alle über eine atomare Transaktion den Counter über den Aufruf der zuvor erklärten incrementCounter Funktion um Eines erhöhen.

Zeile: 23: Durch mapM_ wird auf den Abschluss aller Fibers gewartet. Der Unterschied zwischen mapM und mapM_ besteht darin, dass mapM_ die erhaltenen Resultate verwirft. Dies weil in diesem Beispiel MVar nur wegen seinen Nebeneffekten genutzt wird.

Zeile: 24 bis 25: Um zu überprüfen ob der transaktionelle Counter tatsächlich 1000000 hat und somit alle race conditions richtig gehandelt hat, lesen wir diesen durch die zuvor erklärte getCounter Funktion über eine atomare Transaktion aus und geben das so erhaltene Resultat aus.

6.6.2 Pseudozufallszahlengenerator

6.6.2.1 Actors

6.6.2.1.1 Chan

```
2   random channel seed = do
3       let currentSeed = seed + 0x9E3779B97F4A7C15 :: Word64
4       let z1 = (currentSeed `xor` (currentSeed `shiftR` 30)) * 0xBF58476D1CE4E5B9
5       let z2 = (z1 `xor` (z1 `shiftR` 27)) * 0x94D049BB133111EB
6       let result = (z2 `xor` (z2 `shiftR` 31)) `shiftR` 31
7       writeChan channel result
9       random channel currentSeed
```

Codebeschrieb:

Zeile: 2 bis 9: Diese Funktion beinhaltet den eigentlichen Zufallszahlengenerator

Zeile: 3 bis 6: Hier befindet sich der eigentlichen Pseudozufallsgenerator basierend auf dem C++ splitmix PRNG von Arvid Gerstmann welcher nach Haskell portiert wurde. Durch :: Word64 war das Portieren recht einfach. Das Verwenden von unveränderbaren Variablen sind schon vom Erlang-Beispiel bekannt. Die Notation der Bitoperationen wie beispielsweise `xor` war jedoch sehr gewöhnungsbedürftig und in anderen Programmiersprachen wesentlich schöner gelöst.

Zeile: 7: Hier wird die vom Pseudozufallszahlengenerator generierte Zahl in den Channel geschrieben um diesen dem Main Thread zu senden.

Zeile: 9: Hier wird die random Funktion rekursiv aufgerufen um durch funktionale Programmierung eine Endlosschleife zu realisieren. Der neue Seed wird der rekursiven Funktion als Argument mitgegeben.

```
11  main = do
12      channel <- newChan
13      forkIO $ random channel 0
14      forM_ [1..100] $ \_ -> do
15          threadDelay (1000)
16          print =<< readChan channel
```

Zeile: 12: Hier wird ein neuer regulärer durch MVar implementierter Channel erstellt. Dieser ist per offizieller Dokumentation anfällig auf Deadlocks und es wird empfohlen stattdessen TChan zu nutzen. Siehe dazu das TChan Beispiel weiter unten.

Zeile: 13: Hier wird die Fiber mit dem eigentlichen Zufallszahlengenerator gestartet.

Zeile: 14 bis 16: Dieser durch der forM_ monad realisierte for-loop wird 100-mal ausgeführt. Durch threadDelay werden 1000 Mikrosekunden also 1 Millisekunde gewartet. Dadurch ist gut ersichtlich wie der Generator den Konsumenten durch Aufstauen von tausenden, vorausgenerierten Pseudozufallszahlen überflutet. Anhand von diesem Beispiel sieht man auch, dass durch die unlimitierten Channels ähnliche flow control Probleme wie in Actor concurrancy auftreten.

6.6.2.1.2 MVar

Das im letzten Beispiel auftretende Problem, dass der Channel vom Generator zum Main Thread mit Zufallszahlen überschwemmt wird kann durch die Nutzung von MVar anstelle eines TChan-Channels gelöst werden. Dadurch ist jedoch kein caching mehr möglich da eine MVar nur einen Wert speichern kann. Falls der Konsument mehrere Zufallszahlen auf einmal braucht muss er auf den Generator warten. Während diese Methode das Problem zwar löst, ist sie alles andere als optimal.

```
2   random channel seed = do
7       putMVar channel result
9       random channel currentSeed
10  main = do
11      channel <- newEmptyMVar
13      forM_ [1..100] $ \_ -> do
15          print =<< takeMVar channel
```

Codebeschrieb:

Zeile: 7: Hier wird putMVar anstelle writeChan verwendet da neu in eine MVar und nicht in ein Chan geschrieben werden muss.

Zeile: 11: Hier wird newEmptyMVar anstelle von newChan verwendet da in diesem Beispiel eine MVar und nicht ein Chan erstellt werden soll.

Zeile: 15: Hier wird takeMVar um das vom Generator erhaltene Resultat aus der MVar zu lesen. Im Beispiel mit Chan wurde an dieser Stelle readChan verwendet.

6.6.2.1.3 TChan

Ein weiteres im Beispiel Chan besprochenes Problem waren die Chans an sich. Chans basieren auf MVars und sind somit nicht lock free und können zu races und Deadlocks führen. Auch scheinen race conditions in diesem Beispiel eher selten weswegen der optimistische Umgang mit race conditions durch Software transactional Memory zu einer besseren Performance führen würde. Genau deswegen bietet Haskell mit TChan eine mit Software transactional Memory implementierte alternative zu Chan. Das Problem mit dem Überfluten des Channals durch den Generator bleibt jedoch vorhanden. Eine Lösung dieses Problems wird im nächsten Beispiel gezeigt.

```
2   random channel seed = do
7       atomically $ writeTChan channel result
9       random channel currentSeed
10  main :: IO ()
11  main = do
12      channel <- atomically $ newTChan
14      forM_ [1..100] $ \_ -> do
16          x <- atomically $ readTChan channel
```

Codebeschrieb:

Zeile: 7: Hier wird atomar writeTChan aufgerufen anstatt writeChan zu verwenden. Dies um die generierte Pseudozufallszahl in den TChan zu schreiben.

Zeile: 12: Hier wird atomar newTChan aufgerufen anstatt newChan zu verwenden.

Zeile: 16: Hier wird atomar readTChan aufgerufen um das vom Generator erhaltene Resultat aus dem TChan zu lesen. Im Beispiel mit Chan wurde an dieser Stelle readChan verwendet.

6.6.2.1.4 Bounded TChans

Um das letzte Problem mit dem Haskell Pseudozufallszahlengenerator zu lösen muss ein Flow Control Mechanismus eingefügt werden. Dazu müssen Guards verwendet werden. Diese können in Haskell durch Semaphoren implementiert werden. Haskell kennt verschiedene Arten von Semaphoren. In diesem Beispiel wurde Control.Concurrent.QSem verwendet da diese die einfachste aller Sempahoren in Haskell ist und für die Implementierung eines Guards völlig ausreichend ist. Auch ist diese in der Standardbibliothek von Haskell vorhanden.

Andere Semaphoren in Haskell sind beispielsweise:
Control.Concurrent.STM.SSem, Control.Concurrent.MSem, Control.Concurrent.SSem

Dieses Beispiel ist eine gute Demonstration wie schön man in Haskell parallel programmieren kann. Richtig schade, dass sich Haskell als Programmiersprache bis anhin in der Industrie kaum durchgesetzt hat.

```
2   main = do
3       channel <- atomically $ newTChan
4       guard <- newQSem 10
5       let seed = 0
6       forkIO (forM_ [1..] $ \i -> do
7               waitQSem guard
8               let currentSeed = seed + (0x9E3779B97F4A7C15 * i) :: Word64
9               let z1 = (currentSeed `xor` (currentSeed `shiftR` 30)) * 0xBF58476D1CE4E5B9
10              let z2 = (z1 `xor` (z1 `shiftR` 27)) * 0x94D049BB133111EB
11              let result = (z2 `xor` (z2 `shiftR` 31)) `shiftR` 31
12              atomically $ writeTChan channel result
14          )
15      forM_ [1..100] $ \_ -> do
16          threadDelay (1000)
17          signalQSem guard
18          x <- atomically $ readTChan channel
19          putStrLn $ show x
```

Anders als in den vorherigen Beispielen wurde dieses kaum gekürzt um nochmals einen Überblick über eine gute Haskell Pseudozufallsgeneratorimplementierung zu geben. Für das volle Verständnis dieses Beispiels sind die Erklärungen aus den vorherigen Haskell Zufallszahlengeneratorbeispielen erforderlich.

Codebeschrieb:

Zeile: 3: Hier wird ein neuer TChan erstellt.

Zeile: 4: Hier wird ein neues Guard durch eine Semaphore erstellt. Durch die angegebene Grösse wird die Semaphore mit 10 gestartet. So wird die Anzahl vorausgenerierter Pseudozufallszahlen auf 10 limitiert.

Zeile: 6 bis 14: Durch forkIO einer forM_ [1..] Endlosschleife und eine inline Funktion werden in diesem Beispiel im Vergleich zu früheren Haskell Pseudozufallsgeneratorbeispielen einiges schöner geschrieben. Der Grund, dass dies erst jetzt gemacht wurde ist, da für Programmierer ohne Haskellerfahrungen diese schwerer zu verstehen sind. forM_ [1..] ersetzt beispielsweise

die ganze zuvor verwendete Endlosrekursion durch ein Monad. Dennoch wollte ich diese Art auch noch zeigen. Ansonsten änderte sich im eigentlichen Zufallsgenerator kaum etwas verglichen mit den anderen Haskell Pseudozufallsgeneratorbeispielen.

Zeile: 7: Hier wird die Semaphore um eines verringert. Ist die Semaphore leer so wird gewartet bis der Konsument einen weiteren Wert aus dem TChan liest und die Semaphore um eines erhöht. Dadurch dass dies eine inline Funktion ist, können auf Variablen der äusseren Funktion direkt zugegriffen werden. So muss diese nicht als Argument übergeben werden.

Zeile: 12: Hier wird wie schon im letzten Beispiel atomar writeTChan aufgerufen um die generierte Pseudozufallszahl in den TChan zu schreiben.

Zeile: 15 bis 19: Dieser durch der forM_ monad realisierte for-loop wird 100-mal ausgeführt.

Zeile: 16: Durch threadDelay werden 1000 Mikrosekunden, also 1 Millisekunde gewartet. Dadurch ist ersichtlich, dass anders als in vorherigen Haskell Pseudozufalls-generatorbeispielen hier nicht tausende von Zufallszahlen vorausgeneriert werden, sondern nach 10 auf den Konsumenten gewartet wird.

Zeile: 17: Durch signalQSem wird die Semaphore wieder um Eines erhöht, sodass der Generator eine weitere Pseudozufallszahl vorausgenerieren darf.

Zeile: 18: Hier werden die generierten Zufallszahlen aus dem TChan genommen und somit konsumiert.

Zeile 19: Hier werden die ersten 100 vom Pseudozufallszahlengenerator erhaltenen Pseudozufallszahlen ausgegeben.

7 Tabellenvergleich der Programmiermethoden und Sprachen

In den nachfolgenden Tabellen wird eine Übersicht der behandelten Programmiersprachen betreffend dem parallelen Datenaustausch und den nebenläufige Programmiermodellen aufgelistet.

7.1 Übersicht Nebenläufiger Kontrollkonzepte in verschiedenen Programmiersprachen

Konzept/Sprache	Java	Golang	C#	Erlang	Kotlin	Haskell
Atomics	java.util.concurrent.atomic AtomicCounter	sync.atomic	System.Threading.Interlocked	:atomics atomics_ref()	AtomicReference java.util.concurrent.atomic	Data.Atomics
Mutex	SynchronizedMethodsCounter SynchronizedObjectCounter SynchronizedThisCounter	sync.Mutex	System.Threading: Monitor (Enter, Wait, Pulse, Exit), Mutex, Semaphore, CountdownEvent	Mit Actor Concurancy	Lock.withLock(action) ReentrantReadWriteLock @Synchronized	Control.Concurrent.MVar: takeMVar putMVar MVar Chan bounded_Chan
Software transactional memory	TransactionalCounterMultiverse TransactionalCounterNarayana TransactionalCounterScala	https://github.com/anacrolix/stm	Shielded STMNet Sasa.TM	https://github.com/ian-plosker/stm_erl	Über Java Libraries	TransactionalCounter TChans bounded_TChans
Concurrent Queues, Channels und Messages	java.util.concurrent.BlockingQueue LoomPRNG LoomContinuationPRNG SingleThreadScheduledExecutorDemo	GoroutinesSimple Goroutines MultibleGoroutines MultibleGoroutinesBlocking GoroutinesRandom async	System.Collections.Concurrent.ConcurrentQueue System.Threading.Channels. Channel<T> AsyncLargeFileDownload TasksRandom CoroutinesRandom	!-Operator count/actors prng/actors	kotlinx.coroutines.channels.Channel or https://raw.githubusercontent.com/Kotlin/coroutines-examples/master/examples/channel/channel.kt Coroutines MultibleCoroutines MultibleCoroutinesBlocking CoroutinesRandom CoroutinesRandomYield	Control.Concurrent.Chan or Control.Concurrent.STM.TChan Chan TChans bounded_chan bounded_TChans

Direkt von der Sprache unterstützt

Bibliothek in gleicher Sprache

Zukünftig direkt von der Sprache unterstützt

Bibliothek in anderer Sprache

Möglich durch höhere Nebenläufige Programmiermodelle

Nicht unterstützt

Text: Referenzierter Programmcode

7.2 Übersicht Nebenläufiger Programmiermodelle in verschiedenen Programmiersprachen

Konzept/Sprache	Java	Golang	C#	Erlang	Kotlin	Haskell
Thread Pools	FixedThreadPoolDemo CachedThreadPoolDemo ForkJoinPoolDemo SingleThreadExecutorDemo SingleThreadScheduledExecutor	Ja (mit Gorutines oder Library) https://github.com/shettyh/threadpool	System.Threading.ThreadPool	https://github.com/devinus/poolboy https://github.com/g-andrade/taskforce	Gleich wie in Java	Control.ThreadPool Control.Concurrent.Async.Pool
Futures/Tasks	AsyncHandlers BlockingFuture	Ja (über Gorutines und Channels) https://appliedgo.net/futures/	System.Threading.Tasks	https://github.com/gleber/erlfu	Gleich wie in Java	Control.Concurrent.Future
Futures with completion logic/Promises	CompletableFuture CompletableFutureBranchless ListenableFutureDemo	Ja (über Gorutines und Channels oder async)	Ja (Task oder Async/Await)	https://github.com/gleber/erlfu	Gleich wie in Java	Control.Concurrent.Future
Async/Await		async	AsyncLargeFileDownload	https://github.com/redink/task (Native mit Elixir)		Control.Concurrent.Async
Reactive	ReactiveX (RxJava) ReactiveStreams ReactivePRNG	ReactiveX (RxGo)	ReactiveX (Rx.NET)	Functional Reactive Programming	ReactiveX (RxKotlin)	Functional Reactive Programming
Continuation	LoomContinuationPRNG LoomSingleThreadedContinuationPRNG	Ja (https://bbengfort.github.io/2016/12/yielding-functions-for-iteration-golang/)	CoroutinesRandom	Nur in Elixir mit Continuation Passing Style	CoroutinesRandomYield	Continuation Passing Style
Coroutines	Mit Continuation und Fibers manual machbar	Coroutines CoroutinesSimple MultibleCoroutines MultibleCoroutinesWaitgroup CoroutinesRandom	Mit Continuation und Tasks manuell machbar – Eingebaut in der Unity C# Game Engine	Mit Erlang processes	Coroutines MultibleCoroutines MultibleCoroutinesBlocking CoroutinesRandom	Control.Monad.Coroutine
Fibers	LoomFiberCounter LoomPRNG	Siehe Coroutines Beispiele	Nein aber Async/Await	Light-weight process	Mit Project Loom	Threads sind Fibers
Actor Concurrency	AkkaPRNG AkkaHTTP_ConnectionLevel AkkaHTTP_HostLevel	Ja (Native und über Akka.NET kompatibles protoactor-go) https://github.com/AsynkronIT/protoactor-go	Akka.NET	count/actors prng/actors	https://akka.io/	Control.Concurrent.CHP Control.Concurrent.Actor

Direkt von der Sprache unterstützt

Bibliothek in gleicher Sprache

Zukünftig direkt von der Sprache unterstützt

Bibliothek in anderer Sprache

Möglich durch höhere Nebenläufige Programmiermodelle

Nicht unterstützt

Text: Referenzierter Programmcode

7.3 Übersicht der Begriffe von nebenläufigen Programmiermodellen in verschiedenen Programmiersprachen

Konzept/Sprache	Java	Golang	C#	Erlang	Kotlin	Haskell
Thread Pools	**Thread Pools** https://www.baeldung.com/thread-pool-java-and-guava https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html https://www.geeksforgeeks.org/thread-pools-java/	**Goroutines** https://gobyexample.com/goroutines https://tour.golang.org/concurrency/1 https://medium.com/technofunnel/understanding-golang-and-goroutines-72ac3c9a014d	**Thread Pools** https://www.dotnetperls.com/threadpool https://docs.microsoft.com/en-us/dotnet/api/system.threading.threadpool?view=net-5.0		**Thread Pools** Siehe Java	**ThreadPool Pool** https://hackage.haskell.org/package/Control-Engine-1.1.0.1/docs/Control-ThreadPool.html http://hackage.haskell.org/package/async-pool-0.9.1/docs/Control-Concurrent-Async-Pool.html
Futures/Tasks	**Future** https://www.baeldung.com/java-future https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html	**Goroutines** https://appliedgo.net/futures/	**Tasks** https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=net-5.0		**Future** https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.concurrent/-future/	
Futures with completion logic/Promises	**CompletableFuture** https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/CompletableFuture.html https://stackoverflow.com/questions/14541975/whats-the-difference-between-a-future-and-a-promise	**Goroutines** https://levelup.gitconnected.com/use-go-channels-as-promises-and-async-await-ee62d93078ec	**TaskCompletionSource** https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcompletionsource-1?view=net-5.0		**CompletableFuture** Siehe Java	
Async/Await		**Asynchronous programming** https://medium.com/@gauravsingharoy/asynchronous-programming-with-go-546b96cd50c1	**Asynchronous programming** https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/			
Reactive				**Functional Reactive Programming** https://www.infoq.com/presentations/frp-erlang-grisp/		**Functional Reactive Programming** http://wiki.haskell.org/Functional_Reactive_Programming
Continuation	**Continuation** https://openjdk.java.net/projects/loom/ https://cr.openjdk.java.net/~rpressler/loom/Loom-Proposal.html	**Continuation-Passing style** Yielding Functions https://medium.com/@meeusdylan/continuation-passing-style-in-go-fa06a0ca00a2 https://bbengfort.github.io/2016/12/yielding-functions-for-iteration-golang/	**yield contextual keyword** https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/yield		**Continuation** suspending functions https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.coroutines/-continuation/ https://kotlinlang.org/docs/composing-suspending-functions.html	**Continuation** https://wiki.haskell.org/Continuation
Coroutines	**Coroutines** https://blog.frankel.ch/project-loom-reactive-coroutines/	**Goroutines** https://tour.golang.org/concurrency/1		**Coroutines** https://en.wikipedia.org/wiki/Coroutine	**Coroutines** https://kotlinlang.org/docs/coroutines-overview.html https://github.com/Kotlin/coroutines-examples	
Fibers	**Fibers** Lightweight threads https://openjdk.java.net/projects/loom/ https://cr.openjdk.java.net/~rpressler/loom/Loom-Proposal.html	**Goroutines** https://yourbasic.org/golang/goroutines-explained/		**Light-weight process** http://erlang.org/doc/efficiency_guide/processes.html https://en.wikipedia.org/wiki/Erlang_(programming_language)#Concurrency_and_distribution_orientation	**Fibers** Lightweight threads Siehe Java	**Threads** https://hackage.haskell.org/package/base-4.15.0.0/docs/Control-Concurrent.html https://hackage.haskell.org/package/base-4.15.0.0/docs/Control-Concurrent.html#v:forkIO
Actor Concurrency		**Actors** https://slcjordan.github.io/posts/actors/		**Actors** https://www.infoworld.com/article/2077999/understanding-actor-concurrency--part-1--actors-in-erlang.html		**Actors** https://hackage.haskell.org/package/hactors-0.0.3.1/docs/Control-Concurrent-Actor.html

7.4 Übersicht der Begriffe Nebenläufiger Kontrollkonzepte in verschiedenen Programmiersprachen

Konzept/Sprache	Java	Golang	C#	Erlang	Kotlin	Haskell
Atomics	**Atomic Variables** Atomic Operations https://www.baeldung.com/java-atomic-variables	**Atomic Memory Primitives** https://golang.org/pkg/sync/atomic/	**Interlocked** https://www.dotnetperls.com/interlocked	**Atomic Functions** https://erlang.org/doc/man/atomics.html	**Atomic References** https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.native.concurrent/-atomic-reference/ https://kotlinlang.org/docs/mobile/concurrent-mutability.html	**Atomic Memory** https://hackage.haskell.org/package/atomic-primops-0.8.4/docs/Data-Atomics.html
Monitors (Mutual exclusion & Synchronisation)	**Monitors** Locks Guarded Blocks https://stackoverflow.com/questions/3362303/whats-a-monitor-in-java https://winterbe.com/posts/2015/04/30/java8-concurrency-tutorial-synchronized-locks-examples/ https://www.baeldung.com/java-concurrent-locks https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html	**Monitors** Mutexes https://medium.com/dm03514-tech-blog/golang-monitors-and-mutexes-a-light-survey-84f04f9b7c09	**Monitor** Locks https://docs.microsoft.com/en-us/dotnet/api/system.threading.monitor?view=net-5.0 https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/lock-statement		**Mutual exclusion** Locks https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.sync/-mutex/index.html https://blog.egorand.me/concurrency-primitives-in-kotlin/	**Synchronising variables** MVar (gesprochen: "em-var") https://hackage.haskell.org/package/base-4.3.1.0/docs/Control-Concurrent-MVar.html
Software transactional memory						**Atomic transaction** https://en.wikipedia.org/wiki/Concurrent_Haskell
Concurrent Queues, Channels und Messages	**Blocking Queue** https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html	**Channels** https://tour.golang.org/concurrency/2	**Channels** Concurrent Queue https://devblogs.microsoft.com/dotnet/an-introduction-to-system-threading-channels/ https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentqueue-1?view=net-5.0	**!-Operator** https://erlang.org/doc/reference_manual/expressions.html	**Channels** https://kotlinlang.org/docs/channels.html	**Chan/TChan** https://hackage.haskell.org/package/base-4.15.0.0/docs/Control-Concurrent-Chan.html https://hackage.haskell.org/package/stm-2.5.0.1/docs/Control-Concurrent-STM-TChan.html

8 Fazit

Als Zusammenfassung und zur groben Übersicht werden nachfolgend die wichtigsten Eigenschaften der behandelten nebenläufigen Programmiermodelle beschrieben und eine persönliche Einschätzung und Empfehlung abgegeben.

Nebenläufige Kontrollkonzepte / Concurrency control:
Diese Konzepte wurden ausführlich im Kapitel 5 *Concurrency control* behandelt.

Atomics: Atomare Datentypen sind lockfrei. Durch optimistische, auf Hardware implementierte Operatoren können diese verändert werden. Sie sind der Grundbaustein aller threadsicheren Datentypen. Atomics sind die schnellsten threadsicheren Datentypen und sollen immer verwendet werden, wenn nichts Komplexeres benötigt wird.

Mutex: Ein Mutex wird für das Markieren einer kritischen Region verwendet. In einer kritischen Region kann sich immer nur ein Thread befinden. Dadurch verringert sich der parallelisierbare Teil des Programms. Gemäss Amdahl's law stagniert der Performance-Gewinn nach einer bestimmten Anzahl CPU-Cores. Um diesen Effekt zu reduzieren muss die kritische Region so kurz wie möglich gehalten werden. Mutex sind somit eine pessimistische Art um race conditions zu verhindern. Locks können zu Deadlocks führen. Locks sollten möglichst vermieden werden und durch ein anderes nebenläufiges Kontrollkonzept ersetzt werden.

Software Transactional Memory: Das STM ist das optimistische Äquivalent zum pessimistischen Mutex. Kritische Regionen werden in einer atomaren Transaktion ausgeführt. Tritt eine race condition auf, so wird der Zustand vor der Transaktion wieder hergestellt und die Transaktion wiederholt. Da keine Locks verwendet werden, kann es auch keine Deadlocks geben. Das ganze Programm bleibt parallellsierbar. Treten wenig race conditions auf, so ist STM schneller als ein Mutex. Da Prozessorhersteller bis heute keinen fehlerfreien transaktionellen Speicher in ihrer Hardware herstellen konnten, werden die Transaktionen bis heute per Software emuliert. Dies verursacht leider einen overhead. Generell wäre STM einem Mutex vorzuziehen.

Channels: Über Concurrent Queues, Channels und Messages können Nachrichten zwischen verschiedenen Threads ausgetauscht werden. Sie werden in der Regel für ein Producer / Conumer-System verwendet. Messages sind unbounded, das heisst, wenn der Produzent

schneller als der Konsument muss ein Flow Control -System implementiert werden. Messages beschränken sich nicht auf einen einzigen Computer, sondern sie können auch über das Internet zum Erstellen verteilter Systeme genutzt werden.

Nebenläufige Programmiermodelle:
Diese wurden detailliert im Kapitel 6 *Nebenläufige Programmiermodelle* behandelt.

Thread Pools: Ein Thread Pools fasst mehrere OS Threads zusammen. In diesen Pool können Aufgaben gesendet werden. Sie werden an die im Pool vorhandenen Threads verteilt. Dadurch wird der Overhead vom Erstellen und Löschen von OS Threads vermieden. Unterstützt eine Programmiersprache Fibers so gibt es keinen Grund Thread Pool zu verwenden.

Fibers: Fibers sind leichtgewichtige Threads. Mehrere Fibers werden durch einen, in der Programmiersprache integrierten Scheduler auf einen OS Thread gemappt. Dadurch können beliebig viele Fibers ohne grossen Overhead erstellt und gelöscht werden.

Asynchrone Programmierung: Die asynchrone Programmierung ist eine Abstraktion der parallelen Programmierung. Der Code wird sequenziell geschrieben und der Compiler kümmert sich um die Parallelisierung. Await bietet eine nicht blockierende Möglichkeit, um einen Task zu starten und beim Abschluss des Tasks an der aktuellen Stelle fortzufahren. Falsche Nutzung kann in race condition und Deadlocks enden. Der vom Compiler generierte Code ist ineffizient und schwer zu debuggen. In den meisten Fällen überwiegen die Vorteile.

Eventbasierte Parallelisierung: Mit der eventbasierten Parallelisierung kann auf bestimmte Ereignisse über einen asynchronen Task reagiert werden. Durch Zustände können Daten über Ereignisse ausgetauscht werden. Für die Realisierung von reactive Programmierung werden die Datenstreams an die Observer weitergeleitet. Eventbasierte Parallelisierung eignet sich besonders gut für Webanwendungen und grafische Benutzeroberflächen.

Coroutines: Coroutines kommunizieren über Channels miteinander. Durch das Vermeiden von gemeinsamen Variablen sind keine race conditions möglich. Da die Kapazität der Channels limitiert ist, kann sowohl das Senden wie auch das Empfangen von Daten blockierend sein, wodurch Deadlocks entstehen können. Dies kann durch ein deadlockfreies Kommunikationsmuster verhindert werden. Der Code ist einfach zu schreiben und übersichtlich, was die Effizienz steigert. Coroutinen eignen sich bestens für kleinere Programme, für grössere Projekte wird Actor Concurrency empfohlen.

Message passing channels: Message passing channels ist ähnlich wie actor concurrency, aber mit anonymen Threads anstelle von Aktoren. Durch das Verwenden von lockfreien, beispielsweise durch Software transactional Memory implementierten, unboundet Channels (Messages) für die Kommunikation zwischen den Threads, bleiben, anders als bei Coroutinen, viele Vorteile der Aktor Concurrancy erhalten. Dadurch entsteht ein Paralleles Programmierkonzept, welches ohne grossen Programmieraufwand, die meisten Vorteile der Actor Concurrancy besitzen. Diese Art der parallelen Programmierung eignet sich für alle Projektarten, bei sehr grossen Projekten ist Actor Concurrancy jedoch übersichtlicher.

Actor Concurrency: Bei Actor Cocurrency wird das Programm in mehrere, unabhängige Aktoren aufgeteilt, welche durch Messages miteinander kommunizieren. Dadurch werden geteilte Variablen und somit race condicion vermieden. Da Actor Concurrency lockfrei ist gibt es keine Deadlocks. Dadurch werden alle grösseren Probleme der parallelen Programmierung vermieden. Sie ist einfach zu erlernen und gut verständlich. Sie bietet hervorragende Fehlertoleranz und es können selbstheilende Systeme erstellt werden. Programmteile können während der Laufzeit ausgetauscht werden. Verteilte Systeme über das Internet mit beliebiger Topologie sind möglich. Pro Aktor ist einiges an Code zu schreiben, wodurch sich Actor Concurrency für grösser Projekte sehr gut eignet. Es ist eines der besten parallelen Programmierkonzepte.

9 Ausblick

Erlang gibt es schon seit 1986, dennoch hat sich Actor Concurrency in den imperativen Programmiersprachen bis heute nicht als Standard durchgesetzt. Der Grund dafür liegt sehr wahrscheinlich darin, dass in dieser Programmiersprache sehr viel Code pro Aktor geschrieben werden muss. Solange es keinen Actor Concurrency spezifischen Syntax in imperativen Programmiersprachen gibt, wird sich dieses Konzept kaum durchsetzen. Die Zukunft der parallelen Programmierung könnte bei Message passing channels liegen, da diese die meisten Vorteile der Aktor Concurancy mit wenig Code ermöglichen. Software transactional Memory wird heute eher unterschätzt, denn als lockfreie Alternative zu Mutex hat sie grosses Potenzial. Es ist vorstellbar, dass es in Zukunft Prozessoren mit echtem transaktionellem Speicher gibt, wodurch die Performancenachteile bei STM eliminiert werden. Aktuell ist auch die asynchrone Programmierung eine gute Möglichkeit parallel zu programmieren.

10 Abbildungsverzeichnis

11 Literaturverzeichnis

Bizo dev team. (24. Juni 2014). *Executors.newCachedThreadPool() considered harmful*. Von http://dev.bizo.com/2014/06/cached-thread-pool-considered-harmlful.html abgerufen

Scala STM Expert Group. (Dezember 2017). *Benchmarking*. Von https://nbronson.github.io/scala-stm/benchmark.html abgerufen

Ahad, H. (28. November 2020). *Async/Await in Golang: An Introductory Guide*. Von Hackernoon: https://hackernoon.com/asyncawait-in-golang-an-introductory-guide-ol1e34sg abgerufen

Baeldung. (24. April 2020). *Guide to the Fork/Join Framework in Java*. Von Baeldung.com: https://www.baeldung.com/java-fork-join abgerufen

Bronson, N. (15. Mai 2021). *Scala-stm/scala-stm*. Von https://github.com/scala-stm/scala-stm/ abgerufen

Carvia, T. (3. August 2019). *What is AtomicInteger class and how it works internally*. Von javacodemonk.com: https://www.javacodemonk.com/what-is-atomicinteger-class-and-how-it-works-internally-1cda6a56 abgerufen

Cheung, B. (18. August 2018). *Actor model vs object oriented model*. Von stackoverflow.com: https://stackoverflow.com/questions/41215734/actor-model-vs-object-oriented-model abgerufen

DevilKing. (30. Juli 2019). *Actor vs goroutine*. Von DevilKing's blog: http://gqlxj1987.github.io/2019/03/20/actor-vs-goroutine/ abgerufen

Discolo, A., Harris, T., Marlow, S., Jones, S., & Singh, S. (April 2006). *Lock Free Data Structures using STM in Haskell*. Von Microsoft.com: https://www.microsoft.com/en-us/research/wp-content/uploads/2006/04/2006-flops.pdf abgerufen

Gerstmann, A. (2. Juli 2018). *Better C++ Pseudo Random Number Generator*. Von arvid.io: https://arvid.io/2018/07/02/better-cxx-prng/ abgerufen

GitHub. (22. Mai 2021). *Kotlin / kotlinx.coroutines*. Von GitHub: https://github.com/Kotlin/kotlinx.coroutines abgerufen

GitHub. (10. Februar 2021). *Kotlin/coroutines-examples*. Von GitHub: https://github.com/Kotlin/coroutines-examples abgerufen

HaskellWiki. (23. Juli 2020). *Haskell for multicores*. Von HaskellWiki: https://wiki.haskell.org/index.php?title=Haskell_for_multicores&oldid=63369 abgerufen

Javin, P. (April 2020). *Javarevisted*. Von Difference between atomic, volatile and synchronized in Java?: https://javarevisited.blogspot.com/2020/04/difference-between-atomic-volatile-and-synchronized-in-java-multi-threading.html abgerufen

Jenkov, J. (22. Juni 2019). *Compare and Swap*. Von Jenkov.com: http://tutorials.jenkov.com/java-concurrency/compare-and-swap.html abgerufen

Manning. (2020). *Akka in Action*. Von livebook.manning.com: https://livebook.manning.com/book/akka-in-action/chapter-4 abgerufen

Marlow, & Simon. (10. März 2011). *Control.Concurrent.Actor*. Von Haddock: https://hackage.haskell.org/package/thespian-0.999/docs/Control-Concurrent-Actor.html abgerufen

Oracle. (März 2015). *Interface ExecutorService*. Von API Document, Java Platform Standard Ed. 6: https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ExecutorService.html abgerufen

Oracle. (März 2015). *Interface ScheduledExecutorService*. Von API Document, Java Platform Standard Ed. 6:

https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ScheduledExecutorService.html abgerufen

Oracle. (2020). *Package java.util.concurrent.atomic*. Von Java™ Platform, Standard Edition 7: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html abgerufen

Patolia, A. (11. März 2021). *Kotlin vs Java – Which is Better for Your Android App Development Project?* Von spaceotechnologies.com: https://www.spaceotechnologies.com/kotlin-vs-java/ abgerufen

Progsch, J. (26. September 2014). *ThreadPool*. Von GitHub: https://github.com/progschj/ThreadPool/blob/master/ThreadPool.h abgerufen

Red Hat. (April 2021). *Narayana*. Von Premier open source transaction manager: https://narayana.io/index.html abgerufen

Scala STM Expert Group. (Dezember 2017). *SCALA SMT*. Von https://nbronson.github.io/scala-stm/index.html abgerufen

Sedro-Woolley, J. (9. Februar 2021). *What's the Difference between Channel and ConcurrentQueue in C#?* Von Jeremy Bytes: https://jeremybytes.blogspot.com/2021/02/whats-difference-between-channel-and.html abgerufen

Stackoverflow. (Januar 2015). *How to force main thread to wait for all its child threads finish in Haskell*. Von stackoverflow.com: https://stackoverflow.com/questions/28169297/how-to-force-main-thread-to-wait-for-all-its-child-threads-finish-in-haskell abgerufen

Toub, S. (11. Dezember 2019). *An Introduction to System.Threading.Channels*. Von Devblogs.microsoft.com: https://devblogs.microsoft.com/dotnet/an-introduction-to-system-threading-channels/ abgerufen

Veentjer, P. (2013). *Github*. Von Multiverse: https://github.com/pveentjer/Multiverse abgerufen

Wagner, B. e. (06. April 2020). *Asynchronous programming with async and await*. Von Microsoft: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/ abgerufen

Warski, A. (28. Januar 2020). *Will Project Loom obliterate Java Futures?* Von SoftwareMill: https://blog.softwaremill.com/will-project-loom-obliterate-java-futures-fb1a28508232 abgerufen

Wikipedia. (23. Dezember 2019). *Compare-and-swap*. Von Wikipedia, Die freie Enzyklopädie: https://de.wikipedia.org/w/index.php?title=Compare-and-swap&oldid=195172616 abgerufen

Wikipedia. (18. August 2019). *Message Passing Interface*. Von Wikipedia, Die freie Enzyklopädie: https://de.wikipedia.org/w/index.php?title=Message_Passing_Interface&oldid=191472952 abgerufen

Wikipedia. (02. Januar 2020). *Fetch-and-add*. Von Wikipedia, Die freie Enzyklopädie: https://de.wikipedia.org/w/index.php?title=Fetch-and-add&oldid=195445947 abgerufen

Wikipedia. (18. Dezember 2020). *Test-and-set*. Von Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/w/index.php?title=Test-and-set&oldid=994900064 abgerufen

Wikipedia. (06. Mai 2021). *Amdahl's law*. Von Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/wiki/Amdahl%27s_law abgerufen

Wikipedia. (23. März 2021). *Bitonic sorter.* Von Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/w/index.php?title=Bitonic_sorter&oldid=1013749550 abgerufen

Wikipedia. (9. Mai 2021). *Erlang (Programmiersprache)*. Von Wikipedia, Die freie Enzyklopädie: https://de.wikipedia.org/w/index.php?title=Erlang_(Programmiersprache)&oldid=211785938 abgerufen

Wikipedia. (23. April 2021). *Go (Programmiersprache)*. Von Wikipedia, Die freie Enzyklopädie.: https://de.wikipedia.org/w/index.php?title=Go_(Programmiersprache)&oldid=211222644 abgerufen

Wikipedia. (16. Mai 2021). *Gustafson's law*. Von Wikipedia: https://en.wikipedia.org/wiki/Gustafson%27s_law abgerufen

Wikipedia. (31. März 2021). *Haskell (Programmiersprache)*. Von Wikipedia, Die freie Enzyklopädie: https://de.wikipedia.org/w/index.php?title=Haskell_(Programmiersprache)&oldid=210424726 abgerufen

Wikipedia. (15. März 2021). *Kotlin (Programmiersprache)*. Abgerufen am 27. 5 2021 von Wikipedia, Die freie Enzyklopädie: https://de.wikipedia.org/w/index.php?title=Kotlin_(Programmiersprache)&oldid=209821302

Wikipedia. (14. April 2021). *Spinlock*. Von Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/w/index.php?title=Spinlock&oldid=1017713599 abgerufen

ZeroMQ. (2021). *ZeroMQ Documentation*. Von zeromq.org: https://zeromq.org/get-started/ abgerufen