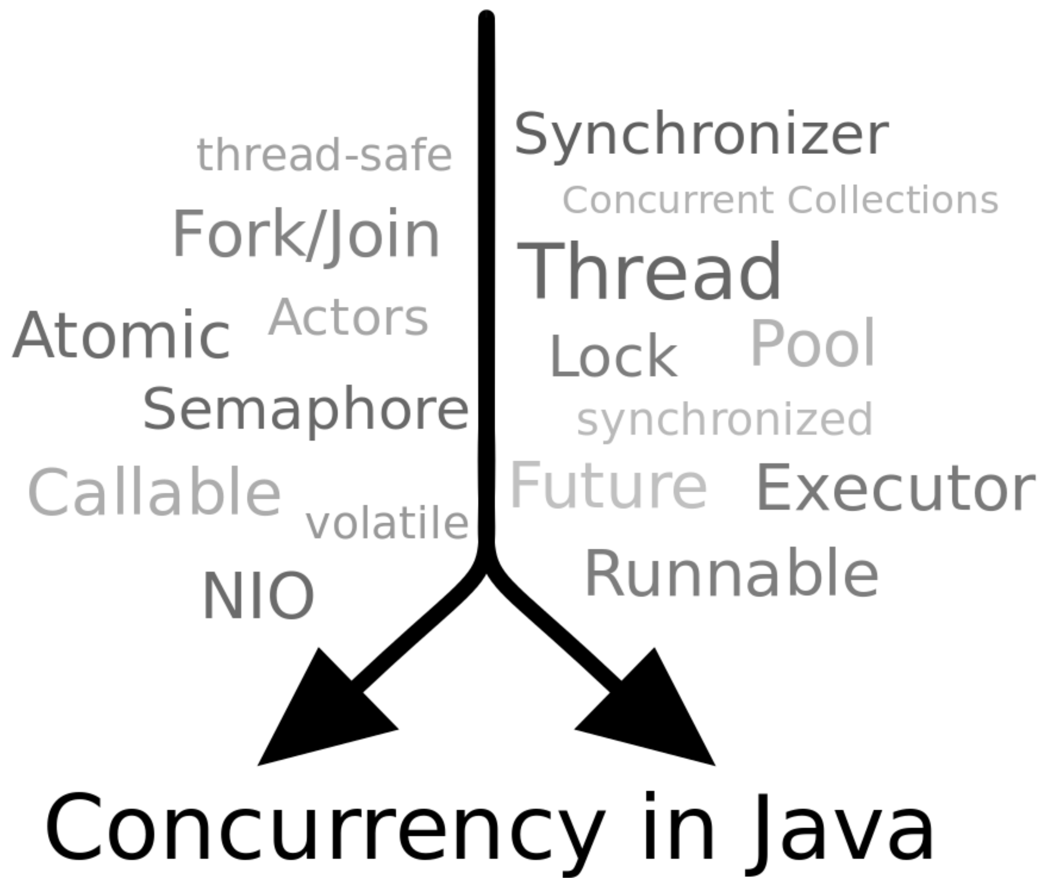

Seminar Softwaretechnik

Sebastian Henneberg

Universität Passau

Lehrstuhl für Softwaresysteme

12. Februar 2011

Inhaltsverzeichnis

1 Motivation **2**
1.1 Anwendungsbereiche . . . 2
1.2 Probleme . . . 2
1.3 Prozess vs. Thread . . . 3

2 Nebenläufigkeit bis Java 1.2 **3**
2.1 Thread (`java.lang.Thread`) . . . 3
2.2 Schlüsselwörter . . . 4
2.2.1 `synchronized` . . . 4
2.2.2 `volatile` . . . 5
2.2.3 `final` . . . 5
2.3 Zusammenfassung . . . 5

3 Nebenläufigkeit in Java 1.4 **5**
3.1 New I/O (`java.nio`) . . . 5

4 Nebenläufigkeit in Java 1.5 **6**
4.1 Atomics (`java.util.concurrent.atomic`) . . . 6
4.2 Locks (`java.util.concurrent.locks`) . . . 7
4.3 Synchronizers (`java.util.concurrent`) . . . 9
4.4 Collections (`java.util.concurrent`) . . . 10
4.5 Executors (`java.util.concurrent`) . . . 11
4.6 Zusammenfassung . . . 12

5 externe Frameworks **12**
5.1 MapReduce . . . 12
5.2 Aktoren . . . 13

6 Nebenläufigkeit in Java 1.7 **14**
6.1 Fork/Join Framework (`java.util.concurrent`) . . . 14
6.2 ParallelArray (`java.util.concurrent.ParallelArray`) . . . 16
6.3 ThreadLocalRandom (`java.util.concurrent.ThreadLocalRandom`) . . . 17

7 Zusammenfassung **17**

1 Motivation

Java ist seit Jahren eine der beliebtesten Programmiersprachen. Sowohl in der Wissenschaft als auch der Wirtschaft [1] [2] finden sich kaum ausgeschrieben Stellen, die keine Java-Kentnisse verlangen. Die Java-Plattform findet Anwendung in vielen Domänen, in denen ein höherer Abstraktionslevel als in C bzw. C++ gewünscht ist. Der TIOBE-Index [3] bestätigt die anhaltende Popularität der Java-Plattform in der Software-Entwicklung.
Zudem werden heutige Computer mit immer mehr Prozessorkernen ausgestattet, die durch die Java Virtual Machine in Form von Threads direkt genutzt werden können. Um skalierbare, effiziente und ruckelfreie Anwendungen zu schreiben, lohnt sich ein Blick in die Entwicklung von nebenläufigen Anwendungen. Diese Ausarbeitung behandelt die nebenläufigen Konstrukte und Konzepte der Java-Plattform in chronologischer Reihenfolge.

1.1 Anwendungsbereiche

An vielerlei Stellen kommt man heutzutage in der Java-Entwicklung nicht mehr umhin, die nebenläufigen Konstrukte der Sprache zu benutzen. Dazu gehört z.B. die Programmierung mit grafischen Toolkits oder Anwendungen, die über das Netzwerk kommunizieren. Durch die parallele Ausführung mehrerer Kontrollflüsse werden reaktive Systeme trotz blockierender Operationen ermöglicht. Es gibt allerdings auch speziellere Domänen [4] in denen nebenläufige Berechnungen Einzug hielten und mittlerweile unverzichtbar sind. Dazu gehören vor allem zeitkritische Anwendungen, die in Echtzeit oder in nahezu-Echtzeit reagieren müssen. Weiter findet Nebenläufigkeit in Programmen Einsatz, in denen viele Operationen auf dem Speicher ausgeführt werden. Ein klassisches Beispiel dafür sind Datenbanken, welche zum Indizieren, für Bulk-Operationen oder für die Verarbeitung der Queries einige Parallelisierungsmechanismen nutzen können. Im wissenschaftlichen Kontext findet Nebenläufigkeit in Simulationen oder rechenintensiver Software ihre Anwendung.
Nicht zuletzt sind die Veränderungen der Hardware-Welt ein schlagkräftiges Argument sich intensiv mit nebenläufiger Programmierung zu beschäftigen. Aufgrund der physikalischen Grenzen haben wir seit Jahren eine oberes Limit der Taktzahlen für Prozessoren erreicht. Stattdessen werden nun immer mehr Prozessorkerne verbaut, die es effizient zu nutzen gilt. Da es sich dabei um eine deutlich komplexere Angelegenheit handelt als bei Programmen mit einem einzigen aktiven Kontrollfluss, lohnt sich die Suche nach Lösungen um die vorhandenen Ressourcen bestmöglich zu nutzen.

1.2 Probleme

Nebenläufige Programmierung ist aber nicht nur wegen der größtenteils domänenunabhängigen Verwendung interessant. Durch die Verwendung von Threads und Prozessen können einige Probleme auftauchen, die man aus der Welt der *single-threaded* Programme nicht kennt. Dazu gehören zahlreiche Liveness-Probleme, bei denen der Fortschritt einer oder mehrerer nebenläufiger Entitäten stagniert. Eines der bekanntesten Probleme dieser Art ist der Deadlock [5]. Dieser tritt auf, wenn zwei oder mehr nebenläufige Aktionen sich derart behindern, dass keinerlei Fortschritt mehr gemacht werden kann. Dies wäre z.B. der Fall, wenn zwei Prozesse zum gleichen Zeitpunkt auf den Fortschritt des jeweils anderen warten. Bei der leichteren Variante namens Livelock bleiben die beteiligten Entitäten zwar aktiv, machen allerdings auch keinerlei Rechenfortschritt mehr. Starvation oder auch ein Fairness-Problem tauchen auf, wenn ein oder mehr Prozesse zwar noch Fortschritt machen, sie aber durch andere Prozesse derart behindert werden, dass dieser Fortschritt minimal ist. Der Grund dieses Problems liegt unter anderem in dem nicht-deterministischen Scheduling. Sofern der Entwickler nicht selbstständig durch irgendeine Art und Weise für Synchronisation und Fairness sorgt, besteht die Gefahr von Starvation.
Ein weiteres und zudem sehr bekanntes Problem ist die Verwendung von gemeinsamen Speicher. Dieses Problem ist primär im Bereich der Threads vorhanden, da sich Threads, die unter dem

gleichen Prozess laufen, ihren Speicherbereich teilen. Ein bekanntes Problem, das bei Verwendung von gemeinsamen Speicher auftreten kann, ist aus dem Kontext der Datenbanken bzw. Transaktion bekannt (Abbildung 1). Dabei teilen sich zwei Entitäten eine gemeinsame Ressource. Durch die Überlagerung der Leseoperation der einen Entität mit der Schreiboperation der Anderen, wird der zuvor gelesene Wert mit einem Neuen überschrieben. Dies kann nach der zweiten Schreiboperation zu Inkonsistenzen führen. Ob das Szenario wirklich eine Gefahr darstellt, hängt von der konkreten Aufgabe ab. In den meisten Fällen ist dieses Verhalten jedoch nicht erwünscht.

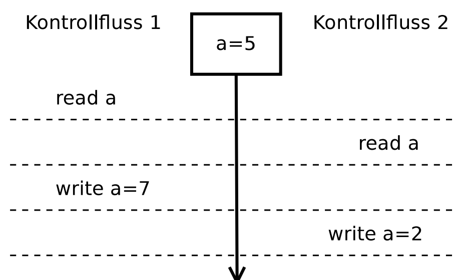

Abbildung 1: nicht-synchronisierte Transaktion

Prozesse bleiben allerdings nicht komplett von dieser Problematik verschont. Speziell im Bereich der verteilten Systeme spielt die Synchronisierung von gemeinsam genutzen Ressourcen eine große Rolle, da die Latenzzeiten zwischen einem lesenden und einem schreibenden Zugriff durch das Netzwerk deutlich höher sind. Um das Transaktionsproblem zu lösen, wäre ein wechselseitiger Ausschluss möglich. Dieser würde dafür sorgen, dass zu jedem Zeitpunkt immer maximal eine Entität auf die Ressource zugreifft.

1.3 Prozess vs. Thread

Java bietet einige Möglichkeiten, aus der virtuellen Maschine heraus neue Prozesse zu starten. Dazu gehören die `exec()`-Methoden der `Runtime`-Klasse als auch die `ProcessBuilder`-Klasse. Letztere ermöglicht das Starten von fremden Prozessen mit Schnittstelle zur Konfiguration von Argumenten und dem Arbeitsverzeichnis sowie Zugriff auf das Environment. Innerhalb der Java-Community hat sich im Laufe der Zeit die Programmierung mit Threads durchgesetzt. Dies zeigt sich vor allem durch die API-Erweiterungen und die entstandenen Frameworks. Bei Verwendung von Threads ist es nicht nötig Interprozess-Kommunikation zu implementieren, da alle Threads des gleichen Prozesses Zugriff auf den gleichen Speicherbereich haben. Trotz des Fokus auf der Programmierung mit Threads wird im Abschnitt 5.1 ein Framework vorgestellt, dass in einem Cluster angewendet wird und somit auf das Erzeugen von Prozessen angewiesen ist.

2 Nebenläufigkeit bis Java 1.2

Gleich vom ersten Tag an war die Möglichkeit der asynchronen Ausführung von Java-Programmen möglich. Das `Runnable`-Interface sowie die `Thread`-Klasse, welche `Runnable` implementiert, waren bereits in der ersten Java-Version vorhanden. Des Weiteren spielen `ThreadGroup` und `Object` eine Rolle in der nebenläufigen Programmierung mit Threads. Alle aufgezählten Klassen befinden sich im Paket `java.lang`. Ein eigenes Paket für Nebenläufigkeits-Werkzeuge gab es zu diesem Zeitpunkt noch nicht.

2.1 Thread (`java.lang.Thread`)

Die Klasse `Thread` stellt den Kern für nebenläufige Ausführung dar. Sie bietet Methoden zum Starten, Stoppen, Anhalten, Priorisieren, Gruppieren und zum Daemonisieren. Threads werden

(sofern möglich) von der virtuellen Maschine in native Threads des Betriebssystems überführt.
Mit Version 2 der Java-Plattform änderte sich die Schnittstelle der `Thread`-Klasse maßgeblich. Die Methoden `destroy()`, `stop()`, `suspend()` und `resume()` wurden als *deprecated* markiert. Aus Gründen der Rückwärtskompatbilität bleiben diese Methoden weiterhin aufrufbar, jedoch wird dringend davon abgeraten [6]. Grund dafür ist die Funktionsweise dieser Methoden. Der Aufruf von `Thread.stop()` wirft eine `ThreadDeath`-Exception, welche zur Folge hat, dass alle momentan bezogenen Monitore, die über das `synchronized`-Schlüsselwort bezogen wurden, abgegeben werden. Das Resultat können inkonsistente Zustände kritischer Ressourcen sein, die zu komplett unvorhersehbaren Verhalten führen können. Der Aufruf von `Thread.suspend()` ist überaus anfällig für Deadlocks, da bereits bezogene Monitore von anderen Ressourcen während des Suspendierens nicht freigegeben werden. Würde also Thread A den Thread B suspendieren und anschließend auf eine von Thread B blockierte Ressource zugreifen wollen, so käme es schließlich zum Deadlock. Folgendes Programmfragment zeigt exemplarisch, wie man eigene Threads denoch stoppen kann.

```
1  public class StopThread {
2
3    private static volatile boolean stopRequested;
4
5    public static void main(String[] args) throws InterruptedException {
6
7      Thread backgroundThread = new Thread(new Runnable() {
8        public void run() {
9          int i = 0;
10         while (!stopRequested)
11           i++;
12         }
13       });
14
15     backgroundThread.start();
16     Thread.sleep(1000);
17     stopRequested = true;
18   }
19 }
```

Die Verwendung des `volatile`-Schlüsselwortes ist unverzichtbar. Ansonsten "optimiert" der Java-Compiler mittels *Loop Invariant Code Motion* [7] [8] die Schleifenbedingung in eine `if`-Abfrage vor die Schleife. Das Ergebnis wäre ein nicht-terminierendes Programm.

2.2 Schlüsselwörter

2.2.1 synchronized

Das Schlüsselwort `synchronized` taucht in Java gleich an zwei verschiedenen Stellen auf. Zum einen als Modifier von Methoden und zum anderen als Statement. Die identische Bezeichnung ist jedoch kein Zufall, da intern auf gleiche Ressourcen zugegriffen wird. Jedes Objekt in Java (auch Class-Objekte) besitzen einen Intrinsic Lock. Durch das `synchronized`-Schlüsselwort wird Zugriff auf diesen Lock genommen. Solange niemand im Besitz des Locks ist, kann man den Besitz des Locks beanspruchen um exklusiven Zugriff zu bekommen. Sofern der Lock bereits an einen Thread vergeben ist, muss solange gewartet werden, bis der Lock zugeteilt wird. Folglich kann zu jedem Zeitpunkt immer nur maximal ein Thread innerhalb der `synchronized`-Blöcke aktiv Berechnungen durchführen.
Die Verwendung von `synchronized` als Modifier von Methoden sieht folgendermaßen aus:

```
public synchronized void accessCriticalResource() {
    // ...
}
```

Die Verwendung von `synchronized` als Statement sieht wie folgt aus:

```
synchronized(someObject) {
```

```
    someObject.doSomething();
}
```

Um Stau durch Blockierung der Locks zu verhindern, besteht die Möglichkeit über `Object.wait()` den Lock temporär innerhalb eines `synchronized`-Blocks abzugeben, bis der Zustand des wartenden Objekts sich verändert hat. Hierzu ruft der Thread, welcher den Zustand geändert hat `Object.notify[All]()` auf. In Folge dessen wird der Lock des wartenden Threads erneut bezogen und die Ausführung wird fortgesetzt. Es empfiehlt sich innerhalb einer Schleife zu überprüfen, ob der gewünschte Zustand sich verändert hat. Andernfalls ruft man `Object.wait()` erneut auf.

2.2.2 volatile

Wie bereits in Abschnitt 2.1 angesprochen, verhindert das `volatile`-Schlüsselwort Compiler-Optimierungen, die zu Terminierungsproblemen in nebenläufigen Anwendungen führen können. Vielmehr garantiert das `volatile`-Schlüsselwort eine globale Ordnung der Lese- und Schreibzugriffe [9]. Dies impliziert, dass jede Leseoperation den momentan aktuellen Wert ließt, bevor die nächste Operation angewendet wird. Ein Caching auf Thread-Ebene wird dadurch ausgeschlossen. Die Verwendung des `volatile`-Modifiers geschieht folgendermaßen:

```
private volatile int counter;
```

2.2.3 final

Das `final`-Schlüsselwort bringt man intuitiv weniger in Kontakt mit Nebenläufigkeit. Jedoch wirkt sich die Verwendung dieses Schlüsselworts positiv in Anwendungen mit parallelen Zugriffen auf eine als `final` deklarierte Variable aus. Grund ist die Beschränkung des Zugriffs auf Lesen. Da keine Schreiboperationen stattfinden dürfen, können keine veralteten Werte ausgelesen werden. Die Verwendung des Schlüsselworts als Modifier sieht beispielsweise wie folgt aus:

```
private final int someImmutable;
```

2.3 Zusammenfassung

Der Thread steht im Mittelpunkt des nebenläufigen Geschehens in frühen Zeiten der Java-Plattform. Zusätzliche Werkzeuge die mehr als nur wechselseitigen Ausschluss durch Blockieren anbieten sucht man vergebens. Folglich kann die Geschwindigkeit mancher Anwendungen die parallel auf den gleichen Instanzen arbeiten teilweise sehr zu wünschen übrig lassen. Da Mehrkernprozessoren noch keine große Rolle zu dieser Zeit spielen, gibt man sich damit zufrieden. Dazu ein Zitat von Brian Goetz (Senior Staff Engineer bei Sun):

> *"In those days, threads were used primarily for expressing asynchrony, not concurrency, and as a result, these mechanisms were generally adequate to the demands of the time."* [10]

3 Nebenläufigkeit in Java 1.4

3.1 New I/O (java.nio)

Mit Version 4 der Java-Plattform wurden vorerst keine Änderungen an den Nebenläufigkeits-Komponenten vorgenommen. Die mit der Sprache und Standardbibliothek ausgelieferten Konstrukte blieben vorerst unberührt. Jedoch gab es eine Erweiterung der API, die den Entwurf von Programmen mit vielen I/O-Operationen verändert hat. Konkret handelt es sich um das neue Paket `java.nio`. Dieses Paket ermöglicht eine äußerst effiziente Verarbeitung von Daten auf der Festplatte sowie von Netzwerkverbindungen. Die Effizienzsteigerung wird durch Asynchronität, Multiplexing und nicht-blockierende Methoden ermöglicht. Statt der Benutzung von Streams und Sockets führt das neue Paket Channels ein. Diese verwalten je einen Eingabe- und

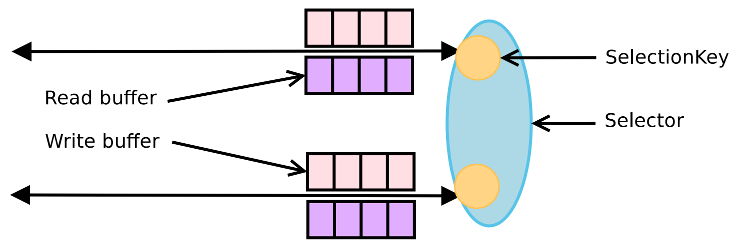

Abbildung 2: Zusammenhang der Schlüsselbegriffe von Java NIO

einen Ausgabe-Puffer. Aufgrund des niedrigeren Abstraktionslevels arbeiten die Puffer auf Byte-Ebene, weshalb durch weniger Kopiervorgänge bereits eine erhöhte Effizienz [11] erreicht wird. Die Architektur nebenläufiger Programme wird schließlich durch die `Selector`-Klasse sowie die `SelectionKey`-Klasse beeinflusst. Jedem SelectionKey ist genau ein Channel zugeordnet. Mithilfe des SelectionKeys kann man das Interesse verschiedener Ereignisse des Channels anmelden. Der Selector kommt ins Spiel um mehrere SelectionKeys gruppiert zu verwalten. Durch die `select()`-Methode der gleichnamigen Klasse besteht die Möglichkeit, auf verschiedene Arten auf ein interessiertes Ereignis der in dem Selector verwalteten Channels aufmerksam gemacht zu werden. Dies kann einerseits blockierend (mit oder ohne max. Wartezeit) als auch schlichtweg über regelmäßiges Anfragen geschehen. Der Selector bietet also die Möglichkeit zahlreiche Dateien oder Netzwerkverbindungen nicht-blockierend in einem Thread zu verwalten. Dabei kann auf den Channels individuell Interesse für spezifische Ereignisse angemeldet werden.
Mit Java NIO verändert sich insbesondere die Architektur von Server-Anwendungen die unter hoher Last stehen oder an einer großen Anzahl an offenen Verbindungen nicht zusammenbrechen dürfen. Durch die Gruppierung von Channels kann die Zahl der Threads minimiert [11] werden. Das Motto *thread-per-socket* ist damit überholt. Alternativ kann Skalierung nun deutlich feingranularer stattfinden. Eine Synchronisation wird durch den Selector ermöglicht. Gemeinsamer Speicher kann durch eine geschickte Aufteilung der dem Selector zugeordneten Channels eliminiert werden.

4 Nebenläufigkeit in Java 1.5

Die mit Abstand umfangreichste Erweiterung der Java API für nebenläufige Anwendungen fand in Version 1.5 mit dem neu eingeführten Paket `java.util.concurrency` statt. Es bietet zahlreiche Werkzeuge um nebenläufige Anwendungen deutlich eleganter ausdrücken zu können. Dabei bedient das Paket ganz verschiedene Anforderungen und kann bei richtiger Verwendung Fehler reduzieren und Entwicklungszeit sparen. Die manuelle Implementierung von Threads sowie die Koordination mehrerer paralleler Aufgaben, wird nun durch eine angenehm zu bedienende API ersetzt. Für Standardsituationen bietet die Bibliothek genug funktionsbereite Implementierungen. Eigene Erweiterungen lassen sich relativ einfach integrieren. Aus diesem Grund wird im Folgenden nun detaillierter auf die 5 wichtigsten Komponenten des `concurrency`-Pakets eingegangen.

4.1 Atomics (`java.util.concurrent.atomic`)

Das Unterpaket `java.util.concurrent.atomic` enthält ein Dutzend Klassen um atomare Operationen auf Variablen zu ermöglichen. Beispielsweise um eine Schreiboperation zu tätigen, die sich auf eine vorhergehende Leseoperation der gleichen Variable bezieht. Analog dazu ist das Transaktions-Problem (Abbildung 1), welches zugleich die Problematik von nicht-atomaren Operationen veranschaulicht. Mithilfe des `volatile`-Schlüsselwortes kann man dies nicht erreichen. Dieses stellt nur sicher, dass jeder Thread der auf eine gemeinsam genutze Variable zugreift, auch den aktuellsten Inhalt zu sehen bekommt. Bisher blieb also nur die Möglichkeit, mithilfe des `synchronized`-Schlüsselwortes Atomizität auszudrücken. Der Nachteil war allerdings, dass

immer nur ein Thread in der Lage war den Monitor der gemeinsam genutzen Variable zu beanspruchen. Alle Threads die während dessen den Monitor anfragten, wurden vorerst blockiert. Letzteres führte folglich zu schlechterer Performanz.
Durch die neuen Klassen `Atomic(Boolean|Integer|Long|Reference<V>)`, werden atomare Operationen angeboten, die *thread-safe* und im Allgemeinen frei von Locks sind. Es gibt jedoch keine Garantie auf Lock-Freiheit. Dies ist abhängig von der darunter liegenden Plattform. Die JVM nutzt für die Klassen des `atomic`-Pakets nämlich erweiterte Befehlssätze, mit denen beispielsweise bedingtes Schreiben ermöglicht wird. Dazu wird der Befehl *Compare and swap* [12] benutzt, welcher auch in Form der Methode `compareAndSet()` in einigen Klassen zu finden ist. Im Grunde genommen handelt es sich schlichtweg um eine `set()`-Methode, die nur dann ausgeführt wird, wenn der momentane Wert einer Variablen gleich dem angenommenen Wert entspricht. Ob die Operation erfolgreich war, darüber informiert der Rückgabewert. Das Setzen bleibt denoch auch ohne Bedingung mittels `set()` möglich. Des Weiteren besteht die Möglichkeit auf `Atomic[Integer|Long]` atomar zu Addieren, Inkrementieren oder Dekrementieren (sowohl `i++` als auch `++i`). Das Paket enthält außerdem noch die Klassen `Atomic(Integer|Long|Reference)Array`, welche sich analog benutzen lassen. Da es sich allerdings gleich um ganze Arrays handelt, werden die meisten Operationen um einen weiteren `int`-Parameter ergänzt, um den Array-Index zu spezifizieren.
Anhand des folgenden Beispiels soll gezeigt werden, wie man einen Generator für Seriennummern mithilfe von `AtomicLong` implementieren kann. Ohne jegliche Verwendung von zusätzlichen Synchronisations-Werkzeugen wird garantiert, dass niemals eine Seriennummer doppelt generiert wird.

```
class SerialNumber {
  private AtomicLong serialNumber = new AtomicLong(0);
  public long next() {
    return serialNumber.getAndIncrement();
  }
}
```

4.2 Locks (`java.util.concurrent.locks`)

Den neu eingeführten Locks in `java.util.concurrent.locks` wurde ebenfalls ein eigenes Unterpaket gewidmet. Mittels dieser Locks werden verschiedene Möglichkeiten geboten um zu blockieren und zu warten. Bei konsequenter Verwendung kann man auf das `synchronized`-Schlüsselwort verzichten und erhält dadurch eine deutlich höhere Flexibilität. Infolgedessen werden auch Alternativen für die Benutzung von `Object.wait` und `Object.notify[All]` angeboten. Ziel dieses Pakets ist es, mittels individueller Locks mehr Funktionalität für verschiedene Anforderungen zu bekommen und bestimmte Szenarien, die auf Locks setzen, zu beschleunigen.
Das zentrale Interface namens `Lock` beschreibt eine Schnittstelle, die einen Lock auf verschiedene Art und Weisen anfordern kann. So besteht die Möglichkeit über die `lock()`- und `unlock()`-Methode ein `synchronized`-Statement zu simulieren. Das Verhalten von `lock()` ist analog zum Eintritt in den `synchronized`-Block und `unlock()` verhält sich analog zum Austritt aus dem Block. Zusätzlich gibt es mit `tryLock()` noch eine Variante die nicht blockiert, allerdings auch nicht in jedem Fall erfolgreich ist. Letztere Funktionalität gibt es auch mit einer eigens spezifizierten maximalen Wartezeit. Als letztes bietet `lockInterruptibly()` die Möglichkeit, solange zu blockieren, bis der Lock zugewiesen wird oder der aktuelle Thread *interrupted* wird. Die wohl am öftesten verwendete Implementierung des `Lock`-Interfaces ist die `ReentrantLock`-Klasse. Sie bietet die zuvor beschrieben Funktionalität mit der Möglichkeit, eine faire Vergabe des Locks in Reihenfolge der Anforderungen zu ermöglichen. Falls man sich nicht für den Fairness-Flag entscheidet, erhält man einen leicht effizienteren Lock. Dies liegt an der fehlenden Warteschlange für wartende Threads. Im fairen Fall kann man über `getQueuedThreads()` die eingereihten Threads ermitteln und über `getOwner()` den aktuellen Besitzer des Locks. Bei der Verwendung von Locks sollte man in jedem Fall daran denken, dass es zu Exceptions kommen kann. Um Liveness zu gewährleisten sollte daher das `lock()`- und `unlock()`-Statement wie im folgenden

Codefragment in einem `try-finally`-Statement verwendet werden. Andernfalls kann ein Lock für ewig blockiert werden.

```
Lock l = ...;
l.lock();
try {
    // access locked resource
} finally {
   l.unlock();
}
```

Das Interface `ReadWriteLock` stellt eine einfache Kaskade für zwei Locks dar. Das Interface spezifiert lediglich die Methoden `readLock()` und `writeLock()`, welche das jeweilige `Lock`-Objekt zurück geben. Die Klasse `ReentrantReadWriteLock` implementiert dieses Interface mit einer Lock-Funktionalität, die analog zum bereits beschriebenen `ReentrantLock` ist. Ein Lese-Lock kann von beliebig vielen Threads gleichzeitig bezogen werden, während ein Schreib-Lock nur exklusiv vergeben werden darf. Zusätzlich darf während ein Schreib-Lock vergeben ist kein Read-Lock vergeben sein. Die Verwendung dieser Klasse bietet sich an, falls man sehr viele parallele Zugriffe auf eine Variable/Instanz hat, auf der größtenteils Leseoperationen ausgeführt werden.
Das letzte Interface des Pakets namens `Condition` bietet eine Alternative zum *wait-notify*-Mechanismus der `Object`-Klasse. Dazu bietet das `Condition`-Interface die Schnittstellen `await()`, `signal()` und `signalAll()` an. Die Vielfalt der `await()`-Methoden ist analog zu `Lock.lock()`. Anstatt nun die `wait()`-Methode auf der eigenen Instanz aufzurufen, muss man zusätzlich ein Condition-Attribut verwalten und darauf `await()` aufrufen. Die gleiche Condition-Instanz muss in mindestens einem anderen Thread verfügbar sein, um von einem noch laufenden Thread irgendwann `signal()` oder `signalAll()` aufrufen zu können.
Das folgende Codefragment zeigt eine ineffiziente Variante der neuen Klasse `SynchronousQueue` (Abschnitt 4.4). Es handelt sich um eine Datenstruktur, die maximal ein Element speichern kann und dazu verwendet wird, ein Objekt von einem Thread an einen anderen auszuliefern. Dazu wird ein Lock benutzt, der beim Eintritt in die `put()`- oder `take()`-Methode bezogen werden muss. Dem Lock sind zwei Conditions assoziiert. Die beiden `Conditions` werden nun dazu benutzt, den Lock temporär freizugeben bis sie von einem Update durch den Aufruf von `signal()` aufgeweckt werden. Es handelt sich hierbei um eine nicht-faire Variante, da die `signal()`-Methode nicht zangsweise den Thread aufweckt, der bereits am längsten wartet.

```
class SynchronousQueue<E> {
  private final Lock lock = new ReentrantLock();
  private final Condition empty    = lock.newCondition();
  private final Condition notEmpty = lock.newCondition();
  private E temp = null;

  public void put(E x) throws InterruptedException {
    lock.lock();
    try {
      while (temp != null) {
        empty.await();
      }
      temp = x;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }

  public Object take() throws InterruptedException {
    lock.lock();
    try {
      while (temp == null) {
        notEmpty.await();
      }
      E x = temp;
```

```
      temp = null;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```

Durch die neuen Lock-Klassen bekommt der Java-Programmierer eine deutlich erhöhte Flexbilität um in ganz individuellen Anwendungsfällen Locks einsetzen zu können. Jedoch verbindet sich damit auch eine größere Gefahr, sofern man sich vor der Verwendung nicht genug Gedanken gemacht hat. So ist man nicht mehr dazu gezwungen, die Locks in umgekehrter Reihenfolge zurück zu geben, wie man sie bezogen hat. Es sind nun beliebige Anordnungen möglich, welche die Gefahr von Liveness-Problemen durchaus erhöhen können.

4.3 Synchronizers (`java.util.concurrent`)

Mithilfe der neu eingeführten Synchronisierer ist man in der Lage nebenläufige Aktionen deutlich besser zu koordinieren. Da Synchronisation nicht zwangsweise etwas mit wechselseitigem Ausschluss bzw. exklusiven Locks zu tun haben muss, bieten diese Klassen einige Alternativen an.
So besteht die Möglichkeit den Zugriff auf eine bestimmte Ressource durch Benutzung der `Semaphore`-Klasse zu beschränken. Bei der Konstruktion der Semaphore, die im Kontext von Nebenläufigkeit weit bekannt ist, gibt man die Anzahl der parallel erlaubten Zugriffe an. Den Zugriff gewährt der Semaphore über die `acquire()`-Methode. Sofern keine weiteren Zugriffe zum momentanen Zeitpunkt möglich sind, blockiert die Methode bis dies der Fall ist. Über die `release()`-Methode wird die zuvor erhaltene Erlaubnis zurück gegeben.
Die Klasse `CountDownLatch` ermöglicht eine Art "Checkpoint" bei der parallelen Verarbeitung mehrerer Threads einzuführen. Analog zu Semaphore wird dem Konstruktor die Anzahl der teilnehmenden Threads mitgeteilt. Anschließend wird jedem Thread die Latch-Instanz übergeben. Am Checkpoint ruft jeder Thread auf dem Latch die Methode `await()` auf. Solange nicht alle Threads die `await()`-Methode aufgerufen haben, blockiert der Aufruf bei allen anderen. In Abbildung 3 ist dies abstrakt für 3 Threads illustriert. Sobald der letzte Thread `await()` aufruft, wird die Ausführung parallel in allen Threads fortgesetzt.

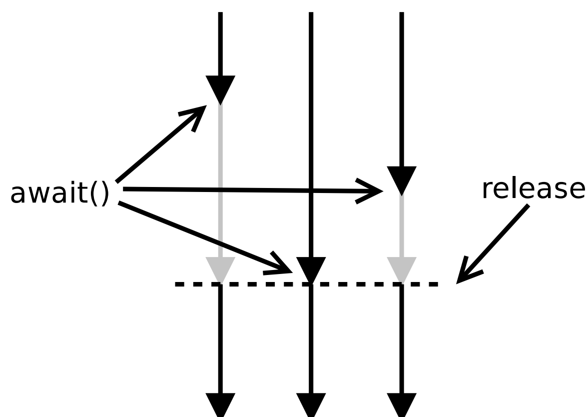

Abbildung 3: Schematische Funktionsweise des `CountDownLatch`

Eine Erweiterung des `CountDownLatch` stellt die `CyclicBarrier`-Klasse dar. Diese erweitert den `await`-Mechanismus im Bezug darauf, dass mehrere Checkpoints mit der gleichen Instanz verwaltet werden können. Weiterhin kann eine `Runnable` spezifiziert werden, dass beim Lösen der Blockierungen eine Aufgabe ausführt. Da weder CountDownLatch noch CyclicBarrier die Threads vor dem Aufruf von `await()` kennen, können diese Klassen sehr flexibel eingesetzt werden. Beide Synchronizer zählen lediglich die Anzahl der `await()`-Aufrufe, anhand derer sie entscheiden,

wann die Ausführung in den Threads fortgeführt werden darf.
Des Weiteren steht die `Exchanger<V>`-Klasse zur Verfügung, um Synchronisationspunkte einzuführen, an denen zwei Threads jeweils Objekte austauschen wollen. Man kann sich die Funktionsweise wie eine blockierende bidirektionale Producer-Consumer-Anordnung vorstellen. Diese eignet sich beispielsweise zum Entwurf von Pipelines. Die einzige Einschränkung ist, dass beide Objekte vom gleichen Typ sein müssen. Dieses Problem lässt sich allerdings durch *Wrapping* lösen.

4.4 Collections (`java.util.concurrent`)

Im Zuge der Nebenläufigkeits-Erweiterung wurden auch eine Reihe von bereits bekannten Collections aus dem Paket `java.util` eingeführt. Das Verhalten spezieller Methoden wurde dazu angepasst oder es wurden neue Schnittstellen geschaffen. Das sicherlich einfachste Beipsiel sind die Klassen `(Array|Linked|Priority)BlockingQueue<E>`, bei denen im Falle einer vollen Queue die `put()`-Operation sowie im Falle einer leeren Queue die `take()`-Operation blockiert. Abbildung 4 veranschaulicht das beschriebene Verhalten. Alle erwähnten Klassen implementieren das Interface `BlockingQueue<E>`. Eine `BlockingQueue<E>` findet überall dort Einsatz, wo es Aufgaben gibt, die in ihrer Auftragsreihenfolge z.B. von einem ThreadPool abgearbeitet werden sollen. Eine Collection die gänzlich auf Speicher verzichtet ist die `SynchronousQueue<E>`. Sie verhält sich ähnlich zur vorher beschriebenen `BlockingQueue`. Der Unterschied besteht darin, dass eine Einfügeoperation solange blockiert bis eine Entfernungsoperation durchgeführt wird. Das Verhalten ist analog für die umgekehrte Anordnung der Operationen. Mithilfe der `SynchronousQueue<E>` können klassische Producer-Consumer-Szenarien bedient werden. Standardmäßig handelt es sich um eine nicht-faire Implementierung, welche aber durch Verwendung eines anderen Konstruktors erreicht werden kann.

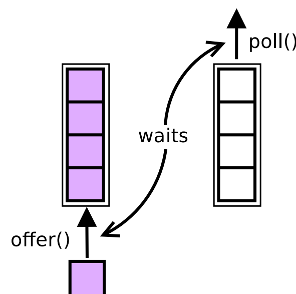

Abbildung 4: Blockierende Operationen der `BlockingQueue`

Die Klasse `ConcurrentHashMap<K,V>` bietet eine *thread-safe* Implementierung einer Hash-Tabelle mit parallelen Schreibzugriffen. Dies wird durch lokale Locks ermöglicht, die nicht die ganze Tabelle blockieren. Die Granularität der Locks bzw. Intensität der Nebenläufigkeit kann zusätzlich durch einen erweiterten Konstruktor konfiguriert werden. Die `ConcurrentHashMap<K,V>` wirft keine `ConcurrentModificationException`, was die Benutzung deutlich vereinfacht. Folglich besteht die Möglichkeit in `foreach`-Schleifen über die Hash-Tabelle, Schreiboperationen auf selbiger ausführen zu können. Die gleichen Vorteile bietet `ConcurrentLinkedQueue<E>` für Anwendungszwecke, in denen statt einer Hash-Tabelle eine FIFO-Struktur gewünscht ist.
Auch die Klassen `CopyOnWriteArray(List|Set)<E>` bieten den Vorteil, dass man sich zumindest um `ConcurrentModificationException` keine Sorgen mehr machen muss. Wie der Name bereits erahnen lässt, legt diese Datenstruktur intern eine Kopie von sich selbst an, sobald ein Schreibzugriff erfolgt. In der neu erzeugten Kopie wird die Schreiboperation angewendet und anschließend der interne Zeiger auf das neue Array gesetzt. Jeder Iterator zeigt zum Zeitpunkt seiner Erzeugung auf das aktuelle Array. Sofern während der Iteration eine Schreiboperation auf dem `CopyOnWriteArray` ausgeführt wird, veraltet der Iterator, bleibt aber weiterhin benutzerbar. Abbildung 5 stellt eine Schreiboperation sowie den daraus resultierenden internen Kopiervorgang

dar.

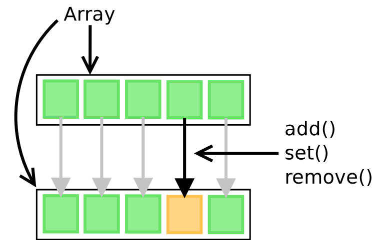

Abbildung 5: Kopiervorgang eines `CopyOnWriteArrays` in Folge eines Schreibzugriffs

4.5 Executors (`java.util.concurrent`)

Mit den Executors wird die Isolation von Aufgabe und Ausführung weiter voran getrieben. Anstatt auf Threads setzt man auf `Runnable` sowie das neue Interface `Callable<E>`. Letzteres funktioniert wie ein `Runnable` das einen anderen Rückgabetyp als `void` in seiner `run()`-Methode ermöglicht. Ziel der Executor ist neben *seperation of concerns* vor allem ein Werkzeug für asynchrone und nebenläufige Ausführung vieler Aufgaben anzubieten. Dabei soll möglichst wenig Overhead erzeugt werden und die Ressourcen sollen zur Laufzeit verwaltet werden können.
Das Interface `Executor` bietet lediglich eine abstrakte Schnittstelle um `Runnable`-Instanzen auszuführen. Erst durch das `ExecutorService`-Interface werden Methoden angeboten, die `Runnables`, `Callables` oder ganze Collections eines Typs entgegen nehmen können. Zudem wird durch die `shutdown()`-Methoden eine Möglichkeit angeboten, den Dienst herunter zu fahren. Eine Erweiterung des `ExecutorService` ist der `ScheduledExecutorService`, welcher eine verzögerte und wiederholte Ausführung der übergebenen Aufgaben ermöglicht. Die Übergabe von Aufgaben an einen der beiden ExecutorServices wird stehts mit einem `Future<V>` beantwortet. Über dieses Future-Objekt bekommt der Aufrufer die Möglichkeit, den Zustand seiner Aufgabe zu ermitteln. So lässt sich in Erfahrung bringen, ob die Aufgabe noch auf Ausführung wartet (verzögert ausgeführt wird oder sich in der Task Queue befindet), momentan berechnet wird oder bereits fertig ausgeführt wurde. Das Future-Objekt ermöglicht zusätzlich den Rückgabewert eines `Callable` zu beziehen. Die Architektur eines abstrakten Executors ist in Abbildung 6 illustriert.

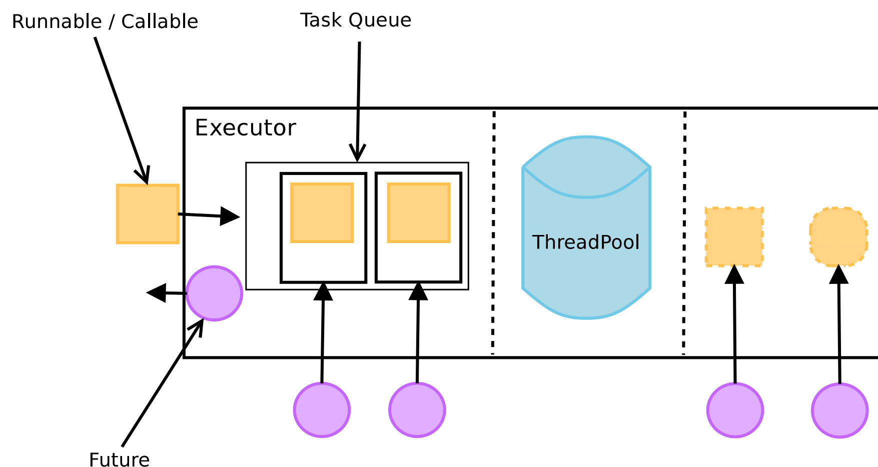

Abbildung 6: Schematische Architektur eines `Executors`

Es gibt durch `[Scheduled]ThreadPoolExecutor` genau zwei konkrete Klassen, welche die zuvor erläuterten Interfaces implementieren. In den meisten Fällen reicht es aber den Executor durch die Factory `Executors` erzeugen zu lassen. Für die meisten Anwendungsfälle finden sich passende Factory-Methoden. Wer etwas individuellere Executor benötigt, hat mit den bestehenden

Implementierungen erhebliche Anpassungs-Möglichkeiten. So lassen sich eigene Task Queues (beschränkt, priorisiert) einbinden um benutzerdefiniertes Scheduling zu ermöglichen. Weiter kann die Größe des ThreadPools bestimmt werden und gegebenenfalls zur Laufzeit angepasst werden, ohne bereits laufende Aufgaben abzubrechen. Unter Umständen kann es sinnvoll sein eine eigene Implementierung des `ThreadFactory`-Interfaces zu spezifizieren. Durch die Future-Objekte besteht zudem die Möglichkeit bereits an den Executor übergebene Aufgaben abzubrechen.
Die Möglichkeiten der Individualisierung sind vielfältig und erlauben eine sehr flexible Benutzung. Durch die Verwendung von ThreadPools und die durchgehende Verwendung von `Runnable` bzw. `Callable` können Threads für mehrere Aufgaben wiederverwendet werden. Dies spart kostbare Zeit beim Erzeugen von Threads, reduziert den Speicherverbrauch sowie die I/O-Belastung aufgrund der geringeren Anzahl an Threads und vermindert zusätzlich noch unnötige Kontextwechsel, welche ebenfalls viel Zeit kosten können.

4.6 Zusammenfassung

Die Einführung des Pakets `java.util.concurrency` sowie die damit verbundenen Unterpakete bieten eine reichhaltige Auswahl an Werkzeugen für die Entwicklung von asynchronen und nebenläufigen Anwendungen. Der parallele Zugriff auf eine Collection muss nicht nicht mehr mit vielen `synchronized`-Schlüsselwörtern anotiert werden, was neben schlechter Übersichtlichkeit zudem noch eine Gefahr für Fehler darstellt und eine schlechte Performanz bietet. Auch die manuelle Implementierung von Synchronisation hatte ein Ende, indem einige leicht verständliche, aber effektiv zu bedienende Klassen diese Arbeit übernehmen. Die alten Konzepte wie Threads oder wait-notify verschwinden aus dem Rampenlicht, da sie einerseits viele Probleme mit sich brachten und andererseits oft eine gewisse Flexibilität vermissen ließen. Dazu ein Zitat von Joshua Bloch (Chief Java Architect bei Google):

> *„In summary, using wait and notify directly is like programming in "concurrency assembly language," as compared to the higher-level language provided by java.util.concurrent"* [13]

5 externe Frameworks

5.1 MapReduce

Das MapReduce-Framework geht einen Schritt weiter als die bisher bekannten Möglichkeiten, welche nebenläufige Berechnungen auf die virtuelle Maschine beschränken. MapReduce [14] weitet parallele Verarbeitung von riesigen Datenmengen auf ein Cluster aus. Dazu müssen im Falle von Java über *Remote Procedure Call* oder *Remote Method Invocation* Prozesse auf entfernten Rechnern erzeugt und mit Daten gespeist werden. Wie der Name schon verrät, charakterisiert sich dieses Verfahren durch die Schritte Abbilden und Reduzieren. Die erste Veröffentlichung zu MapReduce stammt von Google aus dem Jahr 2004. Bei Google wird MapReduce angewendet, um die im Web gesammelten Inhalte zu indizieren.
Der Arbeitsablauf von einer MapReduce-Berechnung beginnt mit dem Aufruf des Masters. Dieser kennt die zu verarbeitenden Daten, die Map-Funktion, die Reduce-Funktion sowie die im Cluster verfügbaren Worker. Letztere werden zu Beginn in Map-Worker und Reduce-Worker unterteilt, indem ihnen die jeweilige Funktion mitgeteilt wird. Nachdem die Eingabedaten vom Master aufgeteilt wurden, teilt dieser jedem Map-Worker mit, welchen Teil der Eingabedaten er zu verarbeiten hat. Jeder Map-Worker speichert die Ergebnisse seiner Berechnungen lokal, die infolgedessen von den Reduce-Workern über das Netzwerk gelesen und anschließend weiter verarbeitet werden. Die Ergebnisse der Reduce-Worker ergeben schlussendlich aggregiert das Ergebnis der MapReduce-Berechnung. Der Ablauf wird in Abbildung 7 veranschaulicht. Ein gutes Beispiel für solch eine Berechnung ist das Zählen der Wörter in einer großen Menge von Dokumenten. Dabei könnte ein Map-Worker die Aufgabe bekommen, in einer Menge von Dokumenten die einzelnen Wörter zu zählen, während die Reduce-Worker die Zähl-Ergebnisse der einzelnen

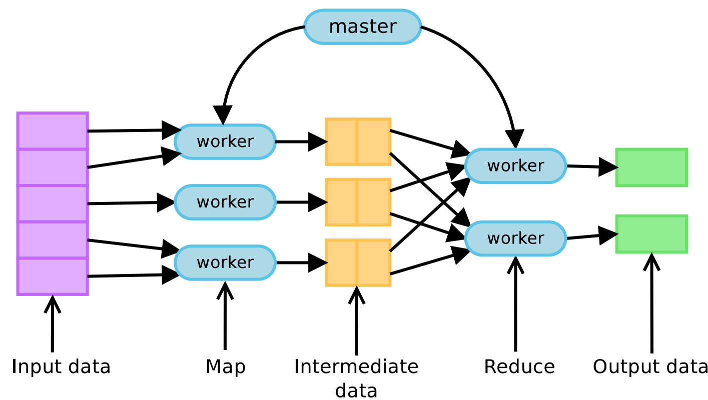

Abbildung 7: Funktionsweise von MapReduce

Map-Worker aufsummieren.
Der de facto Standard für MapReduce im OpenSource-Bereich ist das Hadoop-Projekt[1]. Mittlerweile gibt es immer mehr Web Services, die eine MapReduce-Umgebung gegen Bezahlung zur Verfügung stellen. Darunter auch die Amazon Web Services[2], welche ebenfalls Hadoop einsetzen. MapReduce ist ein Modell, welches sich für die Verarbeitung von extrem großen Datenmengen anbietet. Da einige Vorbereitung stattfinden muss, ist es für kleinere Datenmengen aufgrund des hohen Overheads ungeeignet. Zudem muss gewährleistet sein, dass die Verarbeitung parallelisiert werden kann. Es ist nicht zwingend erforderlich, dass man an zwei Stellen eine parallele Berechnung durchführen kann. So trägt beispielsweise bei Anwendung der Suche im Text (grep) lediglich die Map-Phase zum Ergebnis bei. Die Aufgabe von Reduce ist lediglich die Projektion der Zwischenergebnisse. Der Effizienzgewinn entsteht durch Ausführung in einem Cluster. Da letzteres mit viel Kosten und Aufwand verbunden ist, gibt es mittlerweile zahlreiche Angebote die individuelle Berechnungen in fremden Infrastrukturen ermöglichen.

5.2 Aktoren

Mithilfe von Aktoren lässt sich Nebenläufigkeit auf eine andere Art und Weise implementieren. Kern der Aktoren sind die leichtgewichtigen Prozesse, in denen ähnlich wie in Threads der Kontrollfluss abläuft. Jeder Prozess besitzt eine ID über die er eindeutig identifiziert wird. Sofern ein Prozess die ID eines anderen kennt, ist er in der Lage dem anderen Prozess eine Nachricht zu senden. Diese landet dann in der *Process Mailbox* und kann zu gegebener Zeit abgeholt werden. Sobald ein Prozess `receive` aufruft, erhält er die Nachricht, die sich bereits am längsten in seiner Mailbox befindet (Queue). Auf diese Nachricht kann er durch Senden neuer Nachrichten, Erzeugen neuer Prozesse oder einer Kombination beider reagieren. Die Prozesse besitzen entgegen von Threads keinen gemeinsamen Speicher. Die Kommunikation zwischen den verschiedenen Prozessen wird also ausschließlich über *message passing* ermöglicht. Abbildung 8 veranschaulicht das Aktoren-Modell anhand einiger Prozesse, die sich gegenseitig Nachrichten senden.
Weitläufig bekannt wurde das Modell durch die funktionale Programmiersprache Erlang [15], die den Versand von Nachrichten nicht nur innerhalb der eigenen virtuellen Maschine ermöglicht, sondern auch innerhalb eines Clusters. Durch den erfolgreichen Einsatz von Erlang-Programmen in der Telekommunikationsindustrie wurden Aktoren bekannter und beeinflussten andere Sprachen. Eine Erweiterung der Standardbibliothek von Scala [16] bietet auch hier die Möglichkeit Aktoren [17] zu benutzen. Da Scala-Programme auf der JVM lauffähig sind, wird der Aufruf von Java-Code aus dem Scala-Kontext ermöglicht. Die umgekehrte Richtung ist ebenso möglich. Somit besteht die Möglichkeit, Aktoren indirekt durch den Aufruf von Scala-Code innerhalb von Java zu nutzen.

[1] http://hadoop.apache.org/mapreduce/
[2] http://aws.amazon.com/elasticmapreduce/

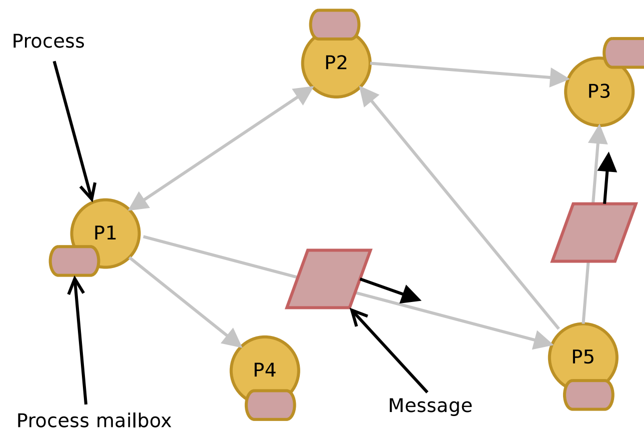

Abbildung 8: Funktionsweise von Aktoren

Die Popularität der Aktoren hat allerdings auch viele Java-Frameworks entstehen lassen. Ein de facto Standard ist derzeit noch nicht absehbar, zudem viele Frameworks noch recht umständlich zu bedienen sind. Momentan existieren folgende Projekte: Akka[3], Ateji PX[4], Korus[5], Kilim[6], ActorFoundry[7] und Jetlang[8].
Das Aktor-Modell bietet gegenüber Threads einige Vorteile. Durch das Senden von Nachrichten in die Process Mailbox sowie das Empfangen über das blockierende `receive`-Statement, besteht für die virtuelle Maschine eine gute Möglichkeit die Prozesse zu schedulen. Gleichzeitig gibt es ein konsistentes Konzept zur Kommunikation der einzelnen Ausführungsstränge. Der Verzicht auf gemeinsamen Speicher verhindert Probleme beim Zugriff auf Variablen und sorgt für ein klareres Konzept. Allerdings erfordert das Aktor-Modell eine andere Denkweise und Struktur der nebenläufigen Programmierung. Daher kann der Umstieg von Threads zu Aktoren anfangs etwas mühsam sein.

6 Nebenläufigkeit in Java 1.7

Die nächste Java-Version bringt viele Neuerungen mit sich. Einige sind leider auf die übernächste Version verschoben worden. Trotzdem bietet Java 1.7 auch für nebenläufige Anwendungen einige neue Klassen und Konzepte. Dazu gehören primär das Fork/Join-Framework sowie das `ParallelArray`, welches eine spezielle Anwendung von Fork/Join mit einer praktisch zu bedienenden Schnittstelle ist. Eine kleinere Erweiterungen ist `ThreadLocalRandom`, welches jedem Thread einen eigenen Zufallszahlengenerator zur Verfügung stellt. Des Weiteren wird ein neuer Synchronizer angeboten, der gegenüber `CountDownLatch` und `CyclicBarrier` (Abschnitt 4.3) über mehr Flexibilität, eine umfangreichere Konfiguration sowie die Möglichkeit von Hierarchien zur Verfügung stellt. Dieser neue Synchronizer wurde auf den Namen `Phaser` getauft.

6.1 Fork/Join Framework (`java.util.concurrent`)

Das Fork/Join-Framework ist inspiriert durch *divide & conquer*-Algorithmen, welche rekursiv ein großes Problem in mehrere kleine Probleme zerteilen. Diese kleineren Probleme können dann abhängig von ihrer Größe erneut aufgeteilt oder direkt (sequentiell) abgearbeitet werden. Durch das Erzeugen von unabhängigen kleineren Problemen besteht die Möglichkeit diese parallel zu

[3] http://akkasource.com/
[4] http://www.ateji.com/px
[5] http://code.google.com/p/korus/
[6] http://kilim.malhar.net/
[7] http://osl.cs.uiuc.edu/af/
[8] http://code.google.com/p/jetlang/

verarbeiten.
Das Konzept ist sehr grob mit dem bereits vorgestellten MapReduce (Abschnitt 5.1) vergleichbar. Allerdings gibt es einige klare Unterschiede. MapReduce beschränkt das Aufteilen der Eingabedaten auf einen einzigen Vorgang, welcher im Master zur Initialisierung stattfindet. Ein erheblicher Unterschied besteht auch in der eingesetzten Infrastruktur. Während das Fork/Join-Framework Nebenläufigkeit auf einer JVM anbietet, arbeitet MapReduce gleich auf ganzen Clustern. In beiden Ansätzen muss allerdings die Voraussetzung geschaffen sein, dass sich die Berechnungen parallelisieren lassen. Es darf also nicht jeder Berechnungsschritt vom jeweils zuvor kommenden Schritt abhängig sein. Das Fork/Join-Konzept ist in Grafik 9 dargestellt. Die grünen Kreise stellen die Probleminstanzen dar, welche rekursiv in kleinere Probleminstanzen aufgeteilt werden. Nachdem die Ergebnisse der kleineren Probleminstanzen berechnet wurden, ermittelt die größere Probleminstanz die Ergebnisse und kombiniert sie in geeigneter Weise. Dieses Verhalten setzt i.d.R. mehrmals rekursiv fort.

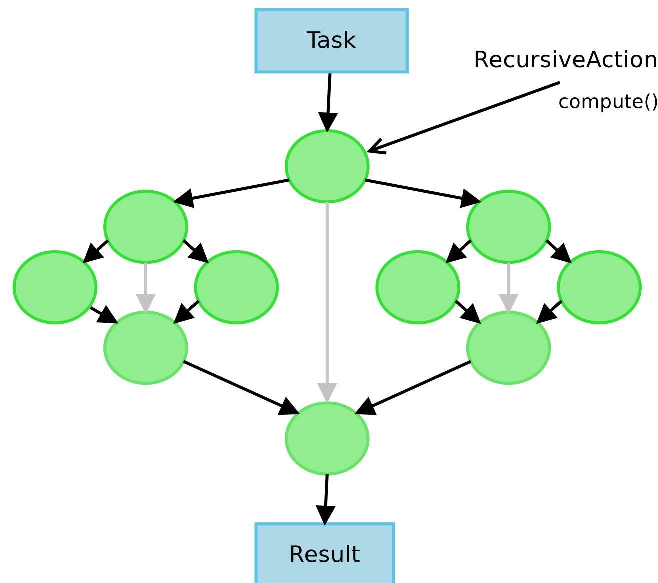

Abbildung 9: Berechnungsabfolge Fork/Join

Das Fork/Join-Framework baut auf den umfangreichen Erweiterungen aus Java 1.5 auf, da es sich bei der zentralen Klasse, dem `ForkJoinPool`, um eine Subklasse von `ExecutorService` handelt. In Berührung kommt man mit dem Pool aber lediglich, um seine Probleminstanzen mittels der `invoke()`-Methode in Auftrag zu geben. Die Standardimplementierung von `ForkJoinPool` kümmert sich bereits um alles weitere wie dem Blockieren, dem Beziehen von Locks und dem Scheduling. Die eigentliche Implementierungsarbeit muss bei der Probleminstanz geleistet werden. Dabei handelt es sich um `RecursiveAction` oder `RecursiveTask<V>`. Die Wahl ist abhängig von der gewünschten Implementierung, da die Ergebnisse aus `RecursiveAction` selbst extrahiert werden müssen, während `RecursiveTask<V>` das Ergebnis der Berechnung direkt in der `compute()`-Methode zurück gibt. Der Rückgabetyp wird durch den generischen Typparameter festgelegt (analog zu `Callable`).
Das folgende Beispiel zeigt eine exemplarische Implementierung, welche von `RecursiveTask<V>` erbt. Die `compute()`-Methode stellt dabei das Herzstück der `ArraySum`-Klasse dar, welche zur Ermittlung der Summe eines `int`-Arrays dient. Die erste `if`-Abfrage prüft, ob das Problem bereits klein genug ist um sequentiell berechnet zu werden. Falls dies nicht der Fall ist, wird das Problem durch Halbierung des Indize-Intervalls in zwei kleinere Probleme `left` und `right` aufgeteilt. Mittels der `left.fork()`-Methode wird das erste der beiden kleineren Probleme zur asynchronen Ausführung an den `ForkJoinPool` gereicht. Der Aufruf von `left.join()` liefert das Ergebnis der `compute()`-Methode, sobald die Berechnung abgeschlossen ist. Mittel `right.compute()` wird selbstständig der rekursive Aufruf getätigt.

```java
class ArraySum extends RecursiveTask<Integer> {
  int low, high;
  int[] values;

  ArraySum(int[] values, int low, int high) {
    this.values = values;
    this.low = low;
    this.high = high;
  }

  protected Integer compute() {
    if(high - low <= 5000) {
      int sum = 0;
      for(int i=low; i < high; ++i) {
        sum += values[i];
      }
      return sum;
    } else {
      int mid = low + (high - low) / 2;
      ArraySum left  = new ArraySum(values, low, mid);
      left.fork();
      ArraySum right = new ArraySum(values, mid, high);
      return rightSum.compute() + leftSum.join();
    }
  }
}
```

Die Arbeit mit dem Fork/Join-Framework erspart viel Arbeit mit den Nebenläufigkeits-Werkzeugen und stellt eine Methode zur Verfügung, in die sich viele Probleme transformieren lassen. Die eigentliche Implementierungsarbeit beschränkt sich auf die Implementierung einer Subklasse von `RecursiveAction` oder `RecursiveTask<V>`. Während die Implementierung relativ schnell getan ist, kann es durchaus knifflig [10] sein, einen geeigneten Grenzwert zu finden. Mithilfe des Grenzwerts wird bestimmt, ab wann ein Problem nicht weiter aufgeteilt wird und sequentiell berechnet werden soll. Wählt man den Grenzwert zu groß, profitiert man nur mäßig von Parallelisierung. Wählt man den Grenzwert zu klein, entsteht ein Overhead durch das Erzeugen zu vieler Probleminstanzen.
Das Fork/Join-Framework bietet eine sehr elegante und ohne viel Aufwand zu nutzende Möglichkeit hohe Nebenläufigkeit auszudrücken. Jedoch sollte man sich bewusst sein, dass bei Wahl eines falschen Grenzwerts die Performanz der Anwendung sogar schlechter sein kann als in der sequentiellen Version.

6.2 ParallelArray (`java.util.concurrent.ParallelArray`)

Die nächste größere Erweiterung für nebenläufige Anwendungen in der kommenden Java-Version ist das ParallelArray. Diese Collection-ähnliche Klasse dient zur Verwaltung größerer Datenmengen von homogenen Instanzen, aus denen besimmte Informationen extrahiert werden sollen. Die auf dem ParallelArray möglichen Operationen sind aus dem Datenbankbereich bekannt. Dabei handelt es sich um Projektionen, Abbildungen, Filter und verschiedene Aggregationsfunktionen. Die Nebenläufigkeit auf dem ParallelArray wird mittels des Fork/Join-Frameworks realisiert. Somit stellt das ParallelArray eine konkrete Ausprägung des Fork/Join-Frameworks für eine besimmte Klasse von Problemen dar, ohne dass der Entwickler sich mit `compute()`-Methoden, `ForkJoinPools` und Grenzwerts auseinander setzen zu muss.
Die Klasse `ParallelArray<T>` bietet eine große Auswahl an Methoden um die gewünschten Datensätze gezielt abzufragen. Darunter befinden sich `allNonIdenticalItems()` zum Löschen von Duplikaten, `removeNulls()` zum Löschen von `null`-Pointern oder `withFilter()` um individuelle Filter anzuwenden. Weiter besteht die Möglichkeit mit `withMapping()` beliebige Operationen wie z.B. Projektionen auf den Elementen des ParallelArray durchzuführen. Mithilfe von `max()`, `min()` oder benutzerdefiniert mit `reduce()`, können die Elemente im ParallelArray aggregiert

werden. Natürlich werden, wie für Java Collections üblich, auch Methoden zum Hinzufügen, Auslesen, Entfernen und Ersetzen angeboten.
Die große Anzahl der Methoden lässt bereits vermuten, dass diese neue Klasse für zahlreiche Probleme nützlich eingesetzt werden kann. Das folgende Code-Fragment zeigt beispielhaft die Verwendung. Das Programm enthält ein `ParallelArray`, welches auf Instanzen von `Student` arbeitet. Ein `Student` fungiert lediglich als Datenablage für die Eigenschaften Name, Abschlussjahr und Notenschnitt. Nachdem das ParallelArray durch Übergabe eines `ForkJoinPools` und des Datenarrays initialisiert ist, können darauf beliebige Operationen angewandt werden. Im Beispiel wird erst ein Filter angewendet, der alle Studenten mit einem Abschluss im aktuellen Jahr auswählt, und anschließend eine Projektion vom Studenten auf seinen Notendurchschnitt durchführt. Die letzte Operation aggregiert die Durchschnittsnoten aller Studenten indem das Minimum gebildet wird. Die Parameter der `withFilter()`- sowie der `withMapping()`-Methode werden als Instanz einer anonymen Klasse implementiert. Dazu wird die Klasse `Ops` i.d.R. statisch importiert um die gewünschte Unterklasse dann zu implementieren.

```
ParallelArray<Student> students = new ParallelArray<Student>(fjPool, data);
double bestGpa = students.withFilter(isSenior)
                         .withMapping(selectGpa)
                         .min();

public class Student {
  String name;
  int graduationYear;
  double gpa;
}

static final Ops.Predicate<Student> isSenior = new Ops.Predicate<Student>() {
  public boolean op(Student s) {
    return s.graduationYear == Student.THIS_YEAR;
  }
};

static final Ops.ObjectToDouble<Student> selectGpa =
  new Ops.ObjectToDouble<Student>() {
  public double op(Student student) {
    return student.gpa;
  }
};
```

Mit der Einführung von *Closures* entfällt der Umstand über die `Ops`-Klasse, da beispielsweise simple Filter oder Abbildungen direkt als Closure ausgedrückt werden können.

6.3 ThreadLocalRandom (`java.util.concurrent.ThreadLocalRandom`)

Eine kleinere aber durchaus interessante Erweiterung der API stellt die Klasse `ThreadLocalRandom` dar. Sie bietet eine Alternative zu `Math.random()` um Zufallszahlen ohne Overhead möglichst effizient zu generieren. `ThreadLocalRandom` bietet im Vergleich zur `Random`-Klasse den Vorteil, dass untere und obere Grenze der zu generierenden Zahl gleich beim Aufruf angegeben werden kann. Dies musste man bisher durch Streckung (Multiplikation) und Verschiebung (Addition) selbstständig vornehmen. Folgendes Statement zeigt exemplarisch die Benutzung:

```
int random = ThreadLocalRandom.current().nextInt(23, 42);
```

7 Zusammenfassung

Innerhalb der letzten 15 Jahre hat sich an der Java-Plattform im Bezug auf Nebenläufigkeit einiges geändert: Klassen wurden hinzugefügt, Methoden als veraltet deklariert, ganze Pakete neu eingeführt und die objektorientierten Konzepte in der API verfeinert. In den frühen Zeiten von Java war es eher mit einer Qual verbunden nebenläufige Anwendungen zu programmieren.

Ganz abgesehen von der Fehleranfälligkeit einiger Methoden der `Thread`-Klasse, was schließlich in Version 1.2 erkannt wurde. Jedoch darf man den Stand der Hardware zu dieser Zeit nicht vergessen. Echte Nebenläufigkeit in Form von paralleler Ausführung war wenn überhaupt in Forschungskreisen interessant.
Die Entwicklung der Hardware beeinflusste maßgeblich auch die Weiterentwicklung der Java-Plattform. Ganz besonders sieht man dies an der bisher größten API-Erweiterung in Java 1.5. Vor allem Technologien wie Mehrkernprozessoren oder Hyperthreading motivierten zu einem eigenen Paket für Nebenläufigkeit. Zudem brachten neue Befehlssätze die Möglichkeit für atomare Operationen. Da der Overhead von übermässiger Thread-Erzeugung und vielen Kontextwechseln erkannt wurde, fanden die Executors bzw. Thread-Pools gleichermaßen Einzug. Die zahlreichen Werkzeuge die nun zur Verfügung gestellt wurden, boten eine erhöhte Effizienz und deutlich mehr Flexibilität.
Der nächste Schritt in der Evolution der Nebenläufigkeit sind die zahlreichen Frameworks. Diese basieren auf teilweise sehr verschiedenen Modellen. Jedoch haben alle ein paar Dinge gemeinsam. Zum einen sollen die Frameworks möglichst unabhängig von der vorhandenen Hardware sein. Zum anderen versuchen sie viele Anwendungsbereiche abzudecken indem das Modell möglichst abstrakt gehalten ist. In jedem Fall geht es darum dem Entwickler die Arbeit deutlich zu vereinfachen, indem er meist nur wenige Methodenrümpfe implementieren muss. Die Frameworks übernehmen selbstständig das Scheduling, welches aufgrund eines vorhanden Modells meist deutlich besser funktioniert als eine "von Hand" implementierte Version.
Allgemein lässt sich feststellen, dass die Klasse `Thread` aus dem Fokus gerückt wurde. Stattdessen werden `Runnable` und `Callable` als Container der parallel auszuführenden Programmlogik verwendet. In Version 1.7 werden es dann schließlich `RecursiveAction` und `ForkJoinTask<V>` oder sogar das `ParallelArray<T>` sein. Mit jedem Schritt schwindet etwas Komplexität für den Software-Entwickler. Zudem sinkt die aufzuwendende Zeit drastisch. Zum einen durch deutlich weniger Implementierungsaufwand, da Synchronisation, Locks und Ressourcenaufteilung bzw. -management automatisiert stattfinden. Zum anderen durch die starke Reduzierung von Fehlern, welche speziell in nebenläufiger Programmierung schlecht erfasst werden kann und daher mehr Aufwand bedeutet.
Wo man früher noch Sprachfeatures nutzte um Synchronität oder einen Mutex zu implementieren, so kategorisiert man heute sein Problem um anschließend ein möglichst passendes Framework darauf anzuwenden. Ein Patentrezept für alle Probleme gibt es nicht. Vielmehr muss an rechenintensiven Stellen jedes mal erneut geprüft werden, ob und welche Strategie am meisten Nutzen bringt. Mithilfe des Aktoren-Modell, des Fork/Join-Framework oder des `ParallelArray` in der zukünftigen Version, werden teilweise schon relativ abstrakte Mechanismen zur Verfügung gestellt. Für speziellere Zwecke bietet das Paket `java.util.concurrent` ausreichende Werkzeuge für individuelle Anwendungen. Letzteres erfordert jedoch mehr Zeit und erhöht die Anfälligkeit von Fehlern. In jedem Fall bleibt es spannend, welche Konzepte und Paradigmen in Zukunft Einzug in die Java-Welt finden.

Literatur

[1] heise jobs. Verfügbar unter http://www.heise.de/jobs/.

[2] Golem stellenmarkt. Verfügbar unter http://jobs.golem.de/.

[3] Tiobe programming community index. Verfügbar unter http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[4] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[5] Deadlock (the java™ tutorials > essential classes > concurrency). Verfügbar unter http://download.oracle.com/javase/tutorial/essential/concurrency/deadlock.html.

[6] Java thread primitive deprecation. Verfügbar unter http://download.oracle.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html.

[7] Peter Haggar. *Practical Java(TM) Programming Language Guide (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, February 2000.

[8] Volatile and loop invariant code motion. Verfügbar unter http://www.vijaykandy.com/2010/07/volatile-and-loop-invariant-code-motion/.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[10] Java theory and practice: Stick a fork in it, part 1. Verfügbar unter http://www.ibm.com/developerworks/java/library/j-jtp11137.html.

[11] Kenneth L. Calvert and Michael J. Donahoo. *TCP/IP Sockets in Java, Second Edition: Practical Guide for Programmers*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2008.

[12] Java theory and practice: Going atomic. Verfügbar unter http://www.ibm.com/developerworks/java/library/j-jtp11234/.

[13] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2008.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, January 2008.

[15] Erlang programming language. Verfügbar unter http://www.erlang.org/.

[16] The scala programming language. Verfügbar unter http://www.scala-lang.org/.

[17] Scala actors: A short tutorial. Verfügbar unter http://www.scala-lang.org/node/242.