Grundsätzliches zur nebenläufigen und parallelen Programmierung

Kernschwemme

Johannes Link

Gängige Programmiersprachen und -paradigmen setzen zumeist sequenzielle Abläufe um und nutzen nicht die parallele Rechenpower heutiger Prozessorarchitekturen. Was muss sich also ändern, damit Entwickler Software schreiben können, die aufgrund zunehmender Anforderungen nicht zu längeren Laufzeiten führt, sondern die Rechenleistung effektiv nutzt?

Es ist zum gängigen Klischee geworden, dass Softwareentwickler ihren Programmierstil grundlegend ändern müssen, wenn ihre Programme künftig die verfügbaren Computer auch nur annähernd in Anspruch nehmen sollen. Als Grund hierfür wird seit etwa fünf Jahren die veränderte Dynamik der Hardwareentwicklung angeführt. Während vorher Moore's Law dazu führte, dass sich alle 18 bis 24 Monate die Geschwindigkeit der Prozessoren verdoppelte, stagniert seitdem die Taktfrequenz. Stattdessen wächst die Anzahl der Rechenkerne (Cores) in ähnlichem Maße. Heute sind Rechner mit vier oder acht Cores, in absehbarer Zeit – so die Vermutung – mit 64 oder gar 1024 Kernen zu kaufen.

Die verbreiteten Programmiersprachen und -paradigmen gehen überwiegend von sequenziellen Abläufen aus und lassen die verfügbare parallele Rechenpower außen vor. Was muss sich ändern, damit Entwickler auch in Zukunft Software schreiben können, deren zunehmende Komplexität nicht zu längeren Laufzeiten führt, sondern die dann vorhandene Rechenleistung effektiv zum Einsatz bringt.

Sollten alle Softwareentwickler die Kunst der „Multi-Core-Programmierung" erlernen? Oder wird es den Framework-Gurus gelingen, die Komplexität der Parallelisierung so zu verstecken, dass der gemeine Anwendungsentwickler auch weiterhin in seiner sequenziellen Illusion leben kann? Die Software-Community kennt die Antwort (noch) nicht. Darum sollte jeder, der mitreden möchte, die grundlegenden Begriffe, Probleme, Lösungsansätze und Paradigmen kennen.

Verwandt, aber nicht verheiratet

Eine kleine Begriffsverwirrung ist zu Beginn zu klären: nebenläufig ist nicht gleich parallel. Während „parallele" Programmausführung auf das Vorhandensein mehrerer gleichzeitig arbeitender Recheneinheiten angewiesen ist, steckt hinter dem Begriff „Nebenläufigkeit" (Concurrency) die abstrakte Idee, dass mehrere Aufgaben (Tasks) quasi nebeneinander bestehen und Fortschritte machen können. Das kann durch tatsächliche Parallelität geschehen, aber auch durch häufiges Wechseln zwischen mehreren Aufgaben (Task Switching), wie es seit Jahrzehnten Time-Slicing- und Multi-Thread-Systeme praktizieren.

Nebenläufigkeit ist somit mehr als parallele Programmausführung. Sie erlaubt es beispielsweise, dass Entwickler an einem Text weiterschreiben können, während dieser „gleichzeitig“ auf der Festplatte gespeichert wird. Und sie ermöglicht es, dass im Browser der Anfang einer Seite zu sehen ist, während im Hintergrund noch der Rest aus dem Netz geladen wird. Nebenläufigkeit ist damit für jeden Entwickler interessant, auch wenn er Programme für einen einzigen Rechenkern entwickelt.

Parallele und nebenläufige Programmierung teilen sich jedoch einen großen Teil der Techniken, Abstraktionen und Probleme. Beide Bereiche lassen sich mit Multi-Threading und Shared-Memory angehen – und Entwickler handeln sich damit bei beiden ähnliche Probleme ein. Doch nicht immer muss Nebenläufigkeit auf dem gleichen Prinzip wie Parallelität beruhen. Als Gegenbeispiel soll die Idee der „non-blocking IO“ dienen. Hierbei warten nicht etwa mehrere Threads gleichzeitig darauf, dass ihre jeweilige I/O-Ressource für das Schreiben beziehungsweise Lesen bereitsteht, sondern ein einziger Thread fragt in einer Schleife ständig alle interessanten I/O-Quellen nach ihrer Bereitschaft für Lese- und Schreibzugriffe ab. Erklärt sich eine Quelle als bereit, wird umgehend erledigt, was auch immer zu lesen, zu schreiben oder zu verarbeiten ist. Der Ansatz ist oft spürbar performanter als Thread-basiertes Warten und skaliert über die Zahl verfügbarer Threads hinaus. Daher kommt er immer dort zum Einsatz, wo es auf maximalen Datendurchsatz ankommt. Microsoft hat mit seiner Async-Bibliothek nebenläufige I/O gar zum festen Bestandteil der .Net-Plattform gemacht [a]. Nicht blockierende, asynchrone I/O ist somit eine Technik der Nebenläufigkeit, jedoch nicht der parallelen Programmierung.

Unabhängig davon, ob man nun Nebenläufigkeit oder Parallelität beobachtet, gibt es einige Herausforderungen zu meistern:

- Zerlegung: Um überhaupt nebenläufige Programmausführung einsetzen zu können, ist ein sequenzieller Algorithmus in mehrere, möglichst unabhängige Teilaufgaben zu zerlegen. Das liegt manchmal auf der Hand (z. B. das asynchrone Lesen einer Datei), ab und an erfordert die Zerlegung eine vollständig andere Herangehensweise an ein Problem. „Parallele Algorithmen“ stellen ein eigenes Forschungsgebiet dar, oft kommt man aber mit einfachen Hilfsmitteln recht weit. Dazu später mehr.
- Koordination: Wenn bekannt ist, in welche Teilaufgaben ein Problem zerlegt wird, sind diese Aufgaben anzustoßen und – meist – am Ende auch wieder zusammenzuführen. Im einfachsten Fall wartet der Entwickler auf das Ende aller Teile, oft muss er jedoch die Teilergebnisse in ein Gesamtergebnis einbringen.
- Kommunikation und Synchronisation: Wenn Teilaufgaben nicht vollständig voneinander unabhängig sind, bedarf es während ihrer Ausführung noch einer Abstimmung. Das kann durch Zugriff auf gemeinsamen Speicher geschehen („Shared Mutable State“, siehe unten) oder durch den Austausch von Nachrichten. Die Kommunikation und Synchronisation ist es, die zu typischen Problemen wie Race Conditions und Deadlocks führt.

Diese Herausforderungen gehen diverse Nebenläufigkeitsansätze völlig unterschiedlich an.

Die oft einfachste Variante der Nebenläufigkeit ist der Start eines neuen Prozesses. Unter Unix-kompatiblem System erreicht der Entwickler das beispielsweise, indem er auf der Kommandozeile eine Anweisung mit „&“ abschließt. Damit startet er ein Kommando oder ein Programm als eigenen Prozess im Hintergrund. Da ein Prozess viele Betriebssystemressourcen benötigt, eignet sich dieses Verfahren für grobgranulare Nebenläufigkeit. Zudem sind die Kommunikationsmechanismen zwischen Prozessen relativ aufwendig und langsam; außer Pipes existiert in der Regel nur die Implementierung eines eigenen Protokolls über TCP-Sockets oder gar HTTP.

Der Vorteil von Parallelisierung mit Prozessen besteht vor allem darin, dass die einzelnen Programme intern nicht für nebenläufige Verwendung ausgelegt sein müssen. Zudem kann man selbstständige Prozesse beinahe ebenso leicht auf einem anderen Rechenknoten starten wie auf dem lokalen Rechner; die Fähigkeit zum Scaling Out bekommt man quasi umsonst dazu.

Warum Groovy-Codebeispiele?

Die meisten Beispiele im Artikel sind in Groovy geschrieben. Das hat zwei Gründe: Groovy erlaubt sehr kompakten und dennoch lesbaren Code. Zudem ist die Groovy-Basissyntax für einen Java- oder C#-Entwickler leicht zu verstehen. Ein paar Dinge sollte man jedoch wissen, um die Beispiele verstehen zu können:

- Jegliche Typinformation ist in Groovy optional. Bei der Deklaration von Variablen oder Methoden ersetzt man den Typ durch das Keyword *def*, in den meisten anderen Fällen kann man den Typ einfach weglassen. Einige Beispiele:

```
def alter = 13
def addiere(a, b) {
  return a + b
}
```

- Klammern können manchmal weggelassen werden. *addiere(a, b)* lässt sich daher auch *addiere a, b* schreiben.
- Groovy erlaubt das Erzeugen anonymer Methoden, sogenannter Closures. Sie entsprechen den Lambda-Ausdrücken in C#, in Java kommen anonyme innere Klassen eines Interface mit einer Methode der Idee von Closures am nächsten. Closures lassen sich genau wie andere Objekte als Parameter und Rückgabewerte verwenden. Zwei Beispiele:

```
def aktion = {println "Ich bin eine Closure"}
aktion()
def iterator = {zahl -> println "Die Zahl heißt $zahl"}
(1..10).each(iterator)
```

Auf esoterische Sprachfeatures verzichtet der Autor weitgehend – oder er erklärt diese an Ort und Stelle.

Einfache Nebenläufigkeit mit Threads

Benötigt ein Entwickler Nebenläufigkeit innerhalb desselben Prozesses, etwa weil er als Reaktion auf ein GUI-Event eine lang laufende Aktion, beispielsweise einen Dateiexport, starten, aber trotzdem dem Nutzer die Weiterarbeit erlauben will, kommen fast automatisch Threads ins Spiel. Sie sind im Vergleich zu Prozessen hinsichtlich ihres Speicherverbrauchs leichtgewichtig, darüber hinaus können sie gemeinsam auf denselben Speicherraum zugreifen – Segen und Fluch gleichermaßen.

Jede ernst zu nehmende Programmiersprache bietet heutzutage Konstrukte an, um eine Aktion in einem neuen Thread zu starten. In Groovy (siehe Kasten „Warum Groovy-Codebeispiele?“) sieht das wie folgt aus:

```
Thread.start { println "Ich laufe nebenläufig" }
println "Ich nicht"
```

Der Code startet einen eigenen Thread, der *"Ich laufe nebenläufig"* auf der Konsole ausgibt. Ob die Zeile oder der Text *"Ich nicht"* zuerst erscheint, kann je nach Timing und Ausführungsumgebung variieren. Würde die Laufzeitumgebung nicht explizit dafür sorgen, dass *println (= java.lang.System.out.println)* bei der Ausgabe einer Zeile nicht unterbrochen wird, ließen sich gar die einzelnen Buchstaben der beiden Texte vermischen. Damit hat der Leser bereits eine wichtige Eigenschaft nebenläufiger Programme kennen gelernt: Ohne zusätzliche Synchronisation kann man nicht wissen, in welcher Reihenfolge der Code unterschiedlicher Threads zur Ausführung kommt. Ein nebenläufiges Programm ist daher inhärent nichtdeterministisch. Ob der Nichtdeterminismus zum Problem wird oder nicht, hängt von vielen Randbedingungen ab; dazu später mehr.

Die etwas leichteren Threads: Tasks

Heutige Betriebssysteme unterstützen Threads nativ. Sie stellen unter anderem Mittel bereit, mit denen sich Threads untereinander synchronisieren lassen. Es kann auch durchaus passieren, dass mehrere Threads gleichzeitig blockiert sind – weil sie auf ein I/O-Ereignis oder ein anderes Event warten; in dieser Zeit beanspruchen sie keine Rechenzeit. Sobald das Event eintritt, sorgt das Betriebssystem dafür, dass der jeweilige Thread wieder Rechenzeit erhält.

Threads sind auch aus anderen Gründen nicht das ideale Mittel, um einzelne Codestücke zu parallelisieren: Sie verbrauchen immer noch recht viel Ressourcen (eigener Stack plus Verwaltungsdaten), ein Thread-Wechsel auf dem gleichen Rechenkern dauert relativ lange und die Anzahl nativer Threads ist auf allen gängigen Betriebssystemen beschränkt (meist auf wenige Tausend). Möchte der Entwickler im eigenen Programm daher eine große Anzahl von Aufgaben parallelisieren, muss er auf andere Mittel zurückgreifen: Task Parallelism. Ein Task ist eine kleine Aufgabe, deren Ausführung zeitlich unabhängig von anderen Tasks stattfinden kann.

In Java – und damit auch in Groovy – steht der *ExecutorService* für diesen Zweck zur Verfügung, in C# erledigt der *TaskScheduler* Ähnliches. Beide nehmen Aufträge zum Ausführen der Tasks entgegen. Im Hintergrund arbeitet ein Pool von Threads, in dem die einzelnen Tasks (in Java durch das Interface *Callable* repräsentiert) meist in der Reihenfolge ihrer „Beauftragung" abgearbeitet werden. Liefert eine solche Aufgabe einen Rückgabewert, kann man sich diesen in einem *Future*-Objekt zurückgeben lassen. Der Name legt es nahe: Ein „Zukunftsobjekt" liefert den Rückgabewert erst, wenn man danach fragt – *future.get()* –, und blockiert die Ausführung, falls die Berechnung zu dem Zeitpunkt noch nicht beendet ist.

Das folgende Beispiel berechnet das Quadrat der Zahlen 1 bis 10, indem es jede einzelne Berechnung an einen Thread-Pool übergibt:

```
ExecutorService executorPool = Executors.newFixedThreadPool(4);
(1..10).each { Integer zahl ->
   Callable task = new Callable() { def call() {zahl * zahl} }
   Future quadrat = executorPool.submit(task)
   println quadrat.get()
}
```

Größe und andere Eigenschaften des Pools sind konfigurierbar; die Wahl der optimalen Parameter hängt von zahlreichen Bedingungen ab, unter anderem von der Zahl der verfügbaren Rechenkerne und vom Aufgabentyp. Führt der Entwickler CPU-intensive Berechnungen durch, sollte er etwa so viele Threads nutzen, wie er Rechenkerne zur Verfügung hat. Bei blockierenden Zugriffen auf externe Ressourcen (Netzwerk, Dateien, Grafikkarte) erlauben mehr Threads häufig schnellere Antwortzeiten.

Auch Tasks greifen, da sie unter der Haube Threads benutzen, auf einen gemeinsamen Hauptspeicherbereich zu und treffen daher auf ähnliche Synchronisationsprobleme. Hinzu kommt, dass derzeitige Betriebssysteme Tasks noch nicht nativ unterstützen, sondern diese in externen Bibliotheken implementieren. Kombiniert man daher Task-Parallelismus mit Thread-Synchronisation, kann es zu folgender Situation kommen: Zwei Lese- warten auf einen dritten Schreib-Task; wenn die beiden Lese-Tasks jedoch ihre Threads blockieren und der Thread-Pool lediglich zwei Threads enthält, dann warten sie für immer. Task-Parallelismus benötigt aus dem Grund eigene Mechanismen zur gegenseitigen Synchronisation und kommt insbesondere mit blockierender, synchroner I/O nur schlecht zurecht.

Das Problem könnte sich mit der nativen Integration von Task-Parallelismus ins Betriebssystem verbessern. Apples OS X hat mit Grand Central Dispatch [b] einen Anfang gemacht, den die Konkurrenz derzeit aufgreift.

Shared Mutable State: Die Nebenläufigkeit wird zum Problem

Die meisten Programme, die von sich behaupten, „parallel“ oder gar „Multi-Core-fähig“ zu sein, benutzen den gleichen Kommunikationsmechanismus: Die einzelnen Threads oder Tasks greifen auf einen gemeinsamen Zustand (Shared State) in Objekten zu und sorgen (meist) mit Locks dafür, dass sie sich trotz gleichzeitiger Ausführung nicht in die Quere kommen. Ist der Programmierer beim Einsatz – oder beim Vermeiden – der Locks nur geschickt genug, erreicht er die drei Hauptziele nebenläufiger Programmierung: Safety, Liveness und Performance. Das sei an folgendem Beispiel verdeutlicht.

Das fachliche Problem ist einfach: In einem Regal (*Shelf*) kann man Produkte (*Product*) einlagern (*putIn*) und wieder hervorholen (*takeOut*). Das Regal hat jedoch nur eine bestimmte Kapazität; ist das Regal voll, wird beim Hineinlegen eine *StorageException* geworfen. Das Lager (*Storehouse*) ist eine Sammlung von Regalen und erlaubt neben dem Aufstellen neuer Regale (*newShelf*) das Bewegen von Produkten aus einem Regal in ein anderes. Alle Operationen sollen in parallelen Threads sicher ausführbar sein. Betrachtet sei zunächst die Shelf-Klasse (Listing 1).

Sowohl die lesenden Zugriffe (*isEmpty*, *isFull*) als auch die verändernden Operationen (*putIn*, *takeOut*) sind durch einen Lock zu schützen, um Race Conditions auszuschließen beziehungsweise um für die Sichtbarkeit veränderter Werte über den lokalen Thread hinaus zu sorgen. *lock {}* ist ein Konstrukt, das Groovy durch ein wenig Metaprogramming im Nachhinein hinzugefügt wurde [8]. Es führt im Hintergrund nicht nur *lock.lock()* und *lock.unlock()* aus, sondern sorgt auch im Falle einer Exception für die korrekte Freigabe des Lock. Race Condition nennt man den Fall, wenn es bei einer bestimmten nebenläufigen Konstellation – ein Thread überholt beim Rennen einen anderen – zu Inkonsistenzen in der Fachlogik kommen kann. Bei einer nicht atomaren oder durch Locks geschützten Put-in-Operation könnte zwischen der *isFull*-Überprüfung und dem tatsächlichen Einfügen des Produkts ein anderer Thread bereits das Regal füllen. Das hätte zur Folge, dass das Regal auf einmal mehr Produkte enthält, als sein Fassungsvermögen zulässt.

Listing 1: Shelf mit expliziten Locks

```
class Shelf {
    final int capacity
    final private products = [] // Erzeugt unsynchronisierte
                               // leere Liste
    final lock = new ReentrantLock()
    Shelf(capacity) {
        this.@capacity = capacity
    }
    List getProducts() {
        lock {
            return this.@products
        }
    }
    void putIn(Product product) {
        lock {
            if (isFull())
                throw new StorageException("shelf is full.")
            products << product
        }
    }
    boolean takeOut(Product product) {
        lock {
            return products.remove(product)
        }
    }
    boolean isEmpty() {
        lock {
            return products.isEmpty()
        }
    }
    boolean isFull() {
        lock {
            return products.size() == capacity
        }
    }
}
```

Dem Zweck der Thread-übergreifenden Sichtbarkeit dient auch der *final*-Modifier an den öffentlichen Members – in Groovy *properties* genannt. Was zunächst wie eine Eigenart des Java-Memory-Modells aussieht, findet man in veränderter Form auf allen anderen Plattformen wieder. Es ist dafür Sorge zu tragen, dass Veränderungen im lokalen Cache eines Thread auf den „echten“ Hauptspeicher übertragen werden. Auf Prozessorebene spricht man oft von Speicherbarrieren (Memory Barriers), die in den Code an passender Stelle einzufügen sind; sowohl Locks als auch das *final*-Schlüsselwort sorgen im Beispiel dafür. Das Problem liegt unter anderem darin, dass Speicherbarrieren und Locks das Programm potenziell verlangsamen; der Zugriff auf den Hauptspeicher benötigt ein bis zwei Größenordnungen mehr Rechenzyklen als der Zugriff auf einen Level-1-Cache des gerade aktiven Rechenkerns. Nichts ist umsonst.

Betrachtet sei nun die Storehouse-Implementierung (siehe Listing 2). Zwei Dinge fallen auf. Es wurde *shelves* als *ConcurrentHashMap* erzeugt; die Verwendung eines Datentyps, der inhärent Thread-sicher ist, ermöglicht den Verzicht auf zusätzliches Locking in der *getAt*-Methode. Thread-sichere Datentypen gibt es für alle gängigen Plattformen, und man sollte sich mit ihnen vertraut machen (siehe auch den Artikel auf Seite 34). Häufig sind diese ohne blockierende Synchronisation implementiert und haben daher ein recht gutes Laufzeitverhalten.

Um jedoch *move* so zu implementieren, dass ein Entwickler nicht in die oben beschriebene Race Condition gerät, müsste er auf die Lock-Objekte der darunterliegenden Shelf-Instanzen zugreifen. Dieser Doppel-Lock birgt jedoch die Gefahr einer Verklemmung (Dead Lock). Treffen beispiels-

weise zwei Threads aufeinander, bei dem der eine Produkte von Regal A zu Regal B bewegt und der andere in die entgegengesetzte Richtung, dann kann es passieren, dass der erste Thread den Lock für Regal A sichert, während gleichzeitig der zweite Thread den Lock für Regal B ergattert. Von nun an müssen beide bis in alle Ewigkeit auf den Lock für das jeweils andere Regal warten. Diesem klassischen Lock-Ordering-Problem lässt sich abhelfen, indem man bei allen Mehrfach-Locks des Programms für eine eindeutige, deterministische Lock-Reihenfolge sorgt. Etwa so:

```
class Storehouse...
  boolean move(Product product, String from, String to) {
    final locks =
      [shelves[from].lock, shelves[to].lock].sort {System.identityHashCode(it)}
    locks[0] {
      locks[1] {
        if (!shelves[to].isFull() && shelves[from].takeOut(product)) {
          shelves[to].putIn(product)
          return true
        }
        return false
      }
    }
  }
```

Das war harte Arbeit. Trotz der leicht verständlichen Fachlogik musste einiges an Kontextwissen bemüht werden, um Thread-Sicherheit zu erreichen, Verklemmungen zu vermeiden und dennoch die parallele Ausführbarkeit der Fachlogik zu erhalten. Dabei kann es passieren, dass schon die nächste fachliche motivierte Änderung zusätzlichen Synchronisationsaufwand erfordert oder gar zum Wechsel der Synchronisationsstrategie zwingt. Viele Entwickler, die sich näher mit dem Thema beschäftigen – zum Beispiel indem sie das Buch von Brian Goetz [1] lesen –, kommen daher zur Einsicht, dass die systemweite, korrekte Verwendung von Locking als Hauptmechanismus für Thread-sichere Programmierung ihre persönlichen Fähigkeiten sprengt. Wer noch zweifelt, möge sich an der Thread-sicheren Variante der in [2] beschriebenen Observer-Klasse versuchen.

Das Problem mit dem Locking

Der Hauptgrund für die Schwierigkeiten beim vorgestellten Programmieransatz ist die mangelnde Komponierbarkeit Thread-sicherer Objekte der gezeigten Art: Nur weil ein Objekt – *Shelf* – für sich alleine betrachtet Thread-sicher ist, lässt es sich nicht einfach in einem anderen Objekt – *Storehouse* – benutzen und kann dessen Thread-Sicherheit „erben“. Der Entwickler muss über den Synchronisationsmechanismus von *Shelf* Bescheid wissen und darüber hinaus das Wissen über die Gesamtstrategie, die geordnete Lock-Reihenfolge, im Rest des Systems verteilen. Brian Goetz [3] nennt das Kind beim Namen: „Locking is not composable.“ Damit bricht die wunderbare Abstraktion vom gekapselten Objekt, das man zu anderen komplexeren Objekten zusammenbauen kann, ohne sich über dessen Implementierung Gedanken zu machen, beim Einsatz von Locks zusammen. Was ist nun zu tun?

Listing 2: Storehouse mit Deadlock-Gefahr

```
class Storehouse {
    final shelves = [:] as ConcurrentHashMap
    Shelf newShelf(String name, int size) {
        shelves[name] = new Shelf(size)
    }
    Shelf getAt(String name) {
        return shelves[name]
    }
    boolean move(Product product, String from, String to) {
        def lockFrom = shelves[from].lock
        def lockTo = shelves[to].lock
        return lockFrom {
            lockTo {
                if (!shelves[to].isFull() && shelves[from].take
                                                    Out(product)) {
                    shelves[to].putIn(product)
                    return true
                }
                return false
            }
        }
    }
}
```

Isolierte Nebenläufigkeit

In vielen Fällen hilft die konsequente Anwendung eines grundlegenden Softwaredesignprinzips: Separation of Concern. Ebenso wie man jedem anderen fachlichen und technischen Aspekt seine eigene Komponente spendiert, macht man das für seine nebenläufige Aspekte. So wird nicht einfach ein neuer Thread oder Task innerhalb der Fachlogik abgespalten, sondern der Entwickler delegiert die Task-Erzeugung beispielsweise an ein Dispatcher-Objekt. Ein solcher Dispatcher ist für sich genommen einfach zu implementieren und erlaubt dem Programmierer darüber hinaus, für Unit-Tests auf sequenzielles, deterministisches Verhalten zu wechseln.

In ähnlicher Form muss er den Zugriff auf Shared State nicht immer im gesamten System verteilen, sondern kann ihn in einem oder wenigen Modulen verstecken. Auf die Thread-Sicherheit dieser Module muss der Entwickler dann erhöhte Aufmerksamkeit verwenden, der Rest seiner Software bleibt „sequenziell einfach“. Venkat Subramaniam beschreibt die Idee ausführlich [4].

Erreicht man mit dieser Vereinfachungsstrategie nicht das Ziel – das verbesserte Antwortverhalten der GUI oder die Beschleunigung einer Berechnung –, lohnt sich der Blick auf Programmierparadigmen, die dem eingesessenen OO-Programmierer ungewohnt erscheinen. Zum Großteil sind sie altbekannt, fast vergessen und erleben nun im Multi-Core-Zeitalter ihre Renaissance.

Transaktionaler Speicher

Ein denkbarer Ausweg aus dem Dilemma der fehlenden Komponierbarkeit ergibt sich, wenn man eine zentrale Idee aus der Welt der Datenbanken auf den Hauptspeicher überträgt: der Heap wird zur transaktionalen Datenmenge. Man erlaubt die Deklaration einer atomaren Codesequenz, deren Ausführung ähnliche Charakteristika haben soll wie eine ACID-Datenbanktransaktion:

- Atomic: Entweder tritt jede Zustandsänderung in Kraft oder keine.
- Consistent: War der Zustand vor der Ausführung konsistent, ist er es danach auch.
- Isolated: Die Auswirkungen der Codesequenz auf den Programmzustand bekommen andere Threads erst nach erfolgreichem Abschluss zu Gesicht.
- Durable: Dauerhaft im Sinne einer Datenbank sind atomare Operationen im Hauptspeicher nicht; dafür sollten sie für alle Threads vollständig sichtbar sein (safely published).

Call for Papers für parallel 2012

Am 23. und 24. Mai 2012 findet in der IHK in Karlsruhe die parallel 2012 statt, eine neue Softwarekonferenz für Parallel Programming, Concurrency und Multicore-Systeme. Die Veranstalter sind heise Developer, *iX* und der dpunkt.verlag. Die Konferenz will Softwarearchitekten, Entwicklern, Projektleitern und IT-Strategen Grundlagen sowie wesentliche Aspekte paralleler Softwareentwicklung und nebenläufiger Programmierung vermitteln. Die Ausrichter fordern ausgewiesene Experten zur Parallelprogrammierung auf, sich bis 31. Dezember 2011 mit Vorträgen und Tutorials auf **www.parallel2012.de** zu bewerben. Gesucht werden Vorträge zu beispielsweise:

- etablierten Multithreading- und Synchronisationsmechanismen
- modernen Programmiermodellen und Parallelisierungsstrategien
- Entwurfsmustern für Parallelprogrammierung
- Erfahrungen mit verbreiteten Programmierplattformen wie Java, .Net und C/C++
- Erfahrungen mit Sprachen wie Ada, Erlang, F#, Fortran, Scala, Clojure und Groovy
- grundsätzlichen architektonischen Entscheidungen für den Einsatz parallelisierter Software
- Erfahrungsberichten von laufenden oder abgeschlossenen Projekten aus unterschiedlichen Industrien
- wichtigen Tools und Bibliotheken im Bereich der Parallelprogrammierung
- parallelen Beschleunigern wie GPUs (CUDA, OpenCL und OpenGL)

Gewünscht sind entweder Langvorträge bis zu 90 Minuten oder kurze Sessions bis zu 40 Minuten. Tutorials sind als ganztägige Veranstaltungen geplant.

Ergänzt der Leser diese Eigenschaften um Schachtelbarkeit, optimistisches Rollback und automatische Wiederholung der atomaren Sequenz im Falle einer Kollision, erhält er ein intuitives Programmiermodell, das Verklemmungen ausschließt und spürbar einfacher zu verwenden ist als explizite Locks. Listing 3 zeigt einen denkbaren Ausschnitt *Shelf* und *Storehouse* in Pseudo-Groovy – man beachte das neue Schlüsselwort *atomic*.

Listing 3: Shelf und Storehouse mit STM-Ansatz

```
class Shelf...
    void putIn(Product product) {
        atomic {
            if (isFull())
                throw new StorageException("shelf is full.")
            products << product
        }
    }
    boolean takeOut(Product product) {
        atomic {
            return products.remove(product)
        }
    }
class Storehouse...
    boolean move(Product product, String from, String to) {
        atomic {
            if (!shelves[to].isFull() && shelves[from].takeOut
                                                    (product)) {
                shelves[to].putIn(product)
                return true
            }
            return false
        }
    }
```

Der entscheidende Unterschied zur Locking-Lösung ist, dass das *Storehouse*-Objekt nichts darüber wissen muss, wie Synchronisation in den verwendeten *Shelf*-Instanzen funktioniert. Der Entwickler gewinnt dadurch die Komponierbarkeit der lieb gewonnenen Objekte zurück. Doch leider bekommt er die Bequemlichkeit nicht geschenkt. Mehrere Nachteile transaktionalen Speichers lassen sich aufzählen:

- Der Ansatz bietet keine Hilfestellung bei der Parallelisierung der Algorithmen und Programme.
- Innerhalb atomarer Codeteile dürfen keinerlei Seiteneffekte (z. B. Schreiben in Datenbank oder Dateisystem, GUI-Ausgabe) auftreten, da man nicht weiß, wie oft der Code einer Transaktion auszuführen ist, bevor diese sich erfolgreich beenden lassen.
- Verklemmungen sind zwar ausgeschlossen, Fortschritte im Programmfluss ohne Fortschritte in der Programmlogik, sogenannte „live locks“, sind jedoch immer noch möglich.

Dann ist da noch die Frage der Praxistauglichkeit. Hardwarebasiertes TM (Transactional Memory) existiert bislang nur in den Forschungslabors der Chip-Hersteller. Effiziente Implementierungen von STM (Software Transactional Memory) sind seit geraumer Zeit Forschungsthema, was die Vermutung nahelegt, dass es sich um ein nicht triviales Problem handelt. Insbesondere ist der Versuch, jede Variablenverwendung in die Transaktion einzubeziehen, äußerst schwierig. Aus dem Grund gehen die meisten in der Praxis verwendeten STM-Systeme einen anderen Weg: Veränderliche Referenzen sind explizit zu kennzeichnen und dürfen nur innerhalb von Transaktionen geändert werden. Packt man nun noch unveränderliche Datentypen hinzu (siehe nächster Abschnitt), ergibt sich ein praxistaugliches Programmiermodell, das sich allerdings grundlegend vom gewohnten OO-Vorgehen unterscheidet. Der Lisp-Dialekt für die JVM, Clojure, ist der zurzeit bekannteste Vertreter dieser STM-Spielart. Mehr über STM bei Clojure [c] kann man im Artikel auf Seite 16 lesen.

Immutable State: Alles wird einfacher

Locking und andere Synchronisationskonzepte benötigt der Entwickler, weil er von unterschiedlichen Threads aus auf gemeinsamen veränderlichen Zustand zugreifen will. Warum also nicht einfach auf veränderlichen Zustand völlig verzichten und damit zahlreiche Nebenläufigkeitsprobleme mit einem Schlag aus dem Weg räumen? Diese Philosophie der Immutability, die die meisten funktionalen Sprachen als Standardansatz unterstützen, klingt für den OO-Entwickler wie ein Taschenspielertrick. Schließlich haben alle nicht trivialen Programme einen internen Zustand, den man irgendwo lesen und verändern können muss.

Der entscheidende Kniff bei Systemen, die (fast) ausschließlich mit unveränderlichen Werten arbeiten, ist die konzeptionelle Unterscheidung zwischen der Identität eines Objekts und dem aktuellen Zustand des Objekts. Während jede „normale“ Operation mit unveränderlichen Objekten arbeitet, existiert für Entitäten – Objekte, deren Zustand sich über die Zeit ändert – ein standardisierter Weg, um ihren Zustand zu aktualisieren. Was sich in der Theorie schwierig anhört, ist in der Praxis leichter verständlich; man passt das Beispiel so an, dass es nur noch an einer Stelle mit veränderlichem Zustand zu tun hat und ansonsten mit unveränderlichen Wert-Objekten arbeitet. Eine unveränderliche Shelf-Klasse sieht in Groovy wie in Listing 4 aus. (Prinzi-

piell lässt sich der Ansatz auch in Java und C# umsetzen. Allerdings ist die korrekte Implementierung unveränderlicher Wertobjekte in klassischen OO-Sprachen oft mühsam, siehe z. B. [5].)

Die Annotation *@Immutable* sorgt während der Kompilierung dafür, dass alles im Bytecode hinzukommt, was die JVM für ein ordentliches unveränderliches Objekt benötigt. Der große logische Unterschied zur veränderlichen Shelf-Variante ist, dass Methoden, die den Zustand verändern (*takeOut* und *putIn*), jetzt ein neues Shelf-Objekt zurückgeben, das bis auf die Produktliste selbst eine Kopie des ursprünglichen Objekts darstellt. „Kopie" ist hierbei lediglich als Konzept zu verstehen; aus Effizienzgründen setzt man häufig auf andere Implementierungen, wie der Autor weiter unten erläutert.

Um später feststellen zu können, aus welcher Version eines Regals die veränderte Instanz hervorgegangen ist, spendiert man der Klasse einen Versionszähler:

```
@Immutable class Shelf...
  long version
  Shelf incVersion() {
    return new Shelf(capacity, products, version + 1)
  }
```

Die *Shelf*-Klasse ist somit eine reine Werte-Klasse; das Verwalten der Shelf-Identitäten überlässt man dem *Storehouse* (siehe Listing 5). Übrig bleibt eine einzige synchronisierte Methode, nämlich die, die den Update von Shelf-Zuständen ermöglicht. Als Parameter nimmt sie eine *Map* entgegen, jeweils mit dem Namen des Regals als *Key* und der neuen Regalinstanz als *Value*. Die Methode überprüft zunächst, ob sich eine der Versionen verändert hat, was auf eine nebenläufige Änderung schließen lässt. Wenn nicht, ersetzt sie den Zustand aller übergebenen Shelf-Instanzen und erhöht deren Versionszähler. Das Vorgehen erlaubt die gleichzeitige und atomare Zustandsänderung mehrerer Werte, wie man sie

Listing 4: Unveränderliches Shelf

```
@Immutable class Shelf...
    int capacity
    List products
    Shelf putIn(Product product) {
        if (isFull())
            throw new StorageException("shelf is full.")
        return cloneWith(new ArrayList(products) << product)
    }
    Shelf takeOut(Product product) {
        if (!products.contains(product))
            return this
        return cloneWith(new ArrayList(products).minus(product))
    }
    private Shelf cloneWith(products) {
        return new Shelf(capacity, products, version)
    }
```

Listing 5: Storehouse mit Update-Methode für Shelves

```
class Storehouse {
    final shelves = [:] as ConcurrentHashMap
    final lock = new ReentrantLock()
    Shelf newShelf(String name, int size) {
        shelves[name] = new Shelf(size, [], 0)
    }
    boolean update(Map names2shelves) {
        lock {
            if (names2shelves.any {name, shelf ->
                    shelves[name].version != shelf.version
                }) return false
            names2shelves.each {name, shelf ->
                shelves[name] = shelf.incVersion()
            }
            return true
        }
    }
```

beim Verschieben von Produkten benötigt (siehe Listing 6). Wieder erkennt der Leser das Muster optimistischer Operationen: Sie werden so lange wiederholt, bis sie erfolgreich sind, was bei wenigen zu erwartenden Kollisionen fast nie notwendig sein wird, bei vielen verändernden Zugriffen aber durchaus die Performanz negativ beeinflussen kann.

Der große Nachteil von Systemen, die auf unveränderliche Werte setzen, ist neben dem für OO-Entwickler ungewohnten Design die Performance – zumindest dann, wenn man den Klonvorgang bei verändernden Funktionen naiv als Deep Copy implementiert. In dem Fall benötigt der Entwickler nicht nur übermäßig viel Speicher, den anschließend der Garbage Collector wieder freigeben muss, sondern das Kopieren an sich braucht spürbar mehr Prozessorzyklen als eine „normale" Zustandsänderung. Tatsächlich lässt sich Cloning oft ressourcenschonend programmieren, indem man lediglich die Referenz auf den Vorgänger und zusätzlich das Zustandsdelta speichert. Sprachen, die auf Immutability als Hauptparadigma setzen, benutzen daher effizient implementierte unveränderliche Datenstrukturen, zum Beispiel Clojures persistente Datenstrukturen.

Actors: Die wahren Objekte?

Ein weiteres Paradigma der nebenläufigen Entwicklung erlebt zurzeit eine Renaissance: Aktoren (Actors). Aktoren sind Objekte, die sich gegenseitig asynchrone und unveränderliche Nachrichten hin- und herschicken und weitgehend auf einen gemeinsam verwendeten Zustand verzichten. Ein Aktor kann einen anderen um Zustandsinformationen bitten und bekommt diese dann im Gegenzug zurückgeschickt. Jeder Aktor hat seinen eigenen Briefkasten, aus dem er die Nachrichten nacheinander entnimmt und abarbeitet. Ein leichtgewichtiges Threading- oder Prozess-Modell sorgt dafür, dass viele Aktoren parallel existieren und sich die verfügbaren Ressourcen effizient teilen können. Manche behaupten sogar, dass Aktoren genau das sind, was Alan Kay im Kopf hatte, als er das Wort „objektorientiert" erfand.

Listing 6: Optimistische Update-Operation

```
class Storehouse...
    boolean move(Product product, String from, String to) {
        while(true) {
            def shelfTo = shelves[to]
            def shelfFrom = shelves[from]
            if (shelfTo.isFull()) return false
            def newShelfFrom = shelfFrom.takeOut(product)
            if (shelfFrom == newShelfFrom) return false
            shelfTo = shelfTo.putIn(product)
            def shelvesToUpdate = [(from): newShelfFrom, (to):
                                                    shelfTo]
            if (update(shelvesToUpdate)) return true
        }
    }
```

Onlinequellen

[a]	Visual Studio Asynchronous Programming	msdn.microsoft.com/en-us/vstudio/gg316360
[b]	Grand Central Dispatch	developer.apple.com/technologies/mac/core.html#grand-central
[c]	Clojure	clojure.org
[d]	GPars	gpars.codehaus.org
[e]	Go	golang.org
[f]	Erlang	www.erlang.org

Bekannt wurden Aktoren durch die Sprache Erlang [f]; dort nennt man sie Agenten. Erlang nimmt für sich in Anspruch, mit dem Aktoren-Konzept hochverfügbare und fehlertolerante Systeme bauen zu können. Insider berichten beispielsweise von 99,9999999 Prozent erreichter Verfügbarkeit bei einem Telefon-Switch der Firma Ericsson. Bibliotheken für die Aktoren-Programmierung gibt es sowohl für Java als auch für .Net. Interessanter sind jedoch Programmiersprachen, die das Konzept direkt unterstützen, neben Erlang etwa Scala oder Groovy mit der GPars-Bibliothek.

Aktoren erleben zurzeit auch deshalb ihren zweiten Frühling, weil manche sie als das Silver-Bullet der nebenläufigen Programmierung verkaufen: Sie fühlen sich beinahe an wie die lieb gewonnenen Objekte und sind darüber hinaus unabhängig, skalierbar und verteilbar. Mit ihnen lassen sich Race Conditions und Verklemmungen einfacher vermeiden. In der Praxis ist leider auch diese Wiese nicht so grün, wie sie von Weitem aussieht: Die Skalierbarkeit im Großen hängt stark von der darunterliegenden Plattform ab. Deadlocks und Co. sind zwar seltener, aber weiterhin möglich, und das Festlegen auf asynchrone Kommunikation gestaltet manche Implementierung aufwendiger, als sie sein müsste. Der größte Fallstrick ist jedoch, dass Aktoren dann versagen, wenn man mit ihnen Aufgaben angehen möchte, die einen Konsens über gemeinsam verwendeten Zustand erfordern – beispielsweise eine Transaktion über mehrere Nachrichten hinweg. Der Artikel auf Seite 21 behandelt das Thema ausführlich.

Parallele Verarbeitung gleichförmiger Daten

Bislang hat sich der Autor auf das Vermeiden von Verklemmungen und Race Conditions konzentriert. Die andere Seite der Medaille ist keineswegs einfacher: Wie parallelisiert man das Beispielprogramm so, dass Nebenläufigkeit überhaupt möglich ist, oder gar so, dass man mit einer nennenswerten Performanceverbesserung durch den Einsatz mehrerer Kerne rechnen kann?

Grundsätzlich steht eine Reihe von Möglichkeiten zur Verfügung: Hat der Entwickler es mit einem Standardproblem zu tun – wie der Berechnung der Zahl Pi (;-)) – fahndet er im Netz nach einem parallel ausführbaren Algorithmus. In den anderen Fällen versucht er sich selbst an der Parallelisierung, meist nach einer der beiden Grundstrategien:

- Parallelisierung des Kontrollflusses: Indem der Entwickler die zeitlichen Vor- und Nachbedingungen aller Berechnungsschritte betrachtet, ermittelt er die Teile des Programms, die nicht in einer zeitlichen Abhängigkeit zueinander stehen, und startet dann an den entsprechenden Stellen einen neuen Task. Dabei können ihn Modelle (z. B. Petri-Netze) unterstützen. Dadurch erreicht er zwar unter Umständen eine spürbare Beschleunigung des Programms, eine für große Mengen verfügbarer Rechenkerne skalierbare Lösung findet er so jedoch meist nicht.
- Datenparallelisierung: Häufiger ist die Situation, dass der Programmierer es mit einer größeren Menge gleichartiger Daten zu tun hat, von denen jedes einzelne Datum auf die gleiche Weise durch das Programm zu schleusen ist. In dem Fall steht ihm plötzlich ein weiterer Weg offen: Er zerteilt die Daten in kleine Häppchen und wirft jedes Häppchen einem Rechenkern zum Fraß vor.

Der Gedanke paralleler Verarbeitung gleichförmiger Daten lässt sich mit der erwähnten Groovy-Bibliothek GPars leicht

umsetzen. Hier ein einfaches Beispiel, das die Quadratzahlen aller Elemente einer Zahlenliste bildet:

```
def numbers  = [1, 2, 3,  4,  5,  6]
def expected = [1, 4, 9, 16, 25, 36]
groovyx.gpars.GParsPool.withPool {
  def squares = numbers.collectParallel { it * it }
  assert squares == expected
  assert squares.sumParallel() == 91
}
```

GPars stellt die entsprechenden **Parallel*-Methoden zur Verfügung, die im Hintergrund einen Thread-Pool benutzen. Den einen oder anderen könnte überraschen, dass es auch eine *sumParallel*-Methode gibt. Sie funktioniert so, dass jeweils zwei Summanden in einem eigenen Task addiert werden und das Ergebnis der Addition anschließend mit einem anderen Berechnungsergebnis wieder „gepaart" wird, bis schließlich die Gesamtsumme ermittelt wurde.

Diese Art der Parallelisierung funktioniert nur effizient, wenn sich die jeweilige Berechnungen tatsächlich unabhängig voneinander durchführen lassen und nicht etwa auf gemeinsame veränderliche Daten zugreifen. Das würde nämlich wieder explizite Synchronisation (z. B. Locking) erfordern, mit allen ihren negativen Auswirkungen auf Performance und Thread-Sicherheit. Eine spezielle Form der parallelen Datenverarbeitung stellt der MapReduce-Ansatz dar. Diesen erläutert der Artikel auf Seite 26.

Dataflow: Nebenläufigkeit wird wieder deterministisch

Sobald nebenläufige Aufgaben komplexere Abhängigkeiten haben, gerät man mit parallelen Mengenoperationen und MapReduce-Ansätzen in Schwierigkeiten, da bei ihnen lediglich am Ende der Berechnungskette eine Gesamtsynchronisation vorgesehen ist. Zu Hilfe kommt das Dataflow-Konzept; ein Satz aus [6] beschreibt die Idee gut: „Dataflow abstraction consists of concurrently run tasks communicating through single-assignment variables with blocking reads." Hier nur ein kleines GPars-Beispiel, um den Leser auf den Geschmack zu bringen:

```
import groovyx.gpars.dataflow.*
import groovyx.gpars.dataflow.*
import static groovyx.gpars.dataflow.Dataflow.task
final dichte = new DataflowVariable()
final gewicht = new DataflowVariable()
final volumen = new DataflowVariable()
task { dichte < gewicht.val / volumen.val }
task { gewicht < 10.6 }
task { volumen < 5.0 }
assert dichte.val == 2.12
```

Mit *task {}* kann man in GPars asynchron Aufgaben starten. *dichte.val* in der *assert*-Zeile wird daher so lange blockiert, bis die anderen Tasks ihre Wertzuweisungen mittels < erledigt haben.

Dataflows haben die Eigenschaft, dass sie sich auch in einer nebenläufigen Umgebung deterministisch verhalten. Das bedeutet unter anderem, dass ein Deadlock, wenn er denn auftritt, immer auftritt, auch wenn man die einzelnen Aufgaben sequenziell startet. Damit wird das Erstellen aussagekräftiger Unit-Tests plötzlich wieder einfach. Darüber hinaus reagieren Dataflow-Variablen in der Praxis wie unveränderliche Werte und benötigen somit keinen Synchronisationscode – Race Conditions und Live Locks sind nicht möglich. Den größten Nachteil stellt wohl auch hier die geänderte Sicht auf den Lösungsraum dar. Entwickler sind nicht gewohnt, in Datenflüssen zu denken und müssen das (wieder) erlernen, wenn sie den Nutzen aus dieser mächtigen Abstraktion ziehen wollen.

Der Artikel auf Seite 31 geht detailliert auf die Eigenschaften des Dataflow-Ansatzes ein.

Fazit

Der heilige Gral der Multi-Core-Programmierung ist noch lange nicht gefunden. Die verbreitetste Idee besteht in der Lock-Synchronisation von Shared Mutable State. Dieser Ansatz ist jedoch in der Praxis nur schwer zu zähmen, weil er der freien Komponierbarkeit von Objekten zuwiderläuft. In der Praxis müssen Entwickler daher je nach Aufgabenstellung und Kontext aus der Menge der bekannten Programmierparadigmen einen geeigneten Ansatz auswählen.

Benötigt man echtes nebenläufiges, transaktionales Verhalten über beliebige Objekte hinweg, muss man den Preis dafür zahlen. Entweder in Form der fehlerträchtigen manuellen Synchronisation, oder als noch (?) nicht ausgereifte STM-Technik oder in der ungewohnten Variante versionierter unveränderlicher Zustandsobjekte. Eines jedoch ist klar: Je weniger Shared Mutable State ein System besitzt, desto einfacher ist Thread-Sicherheit zu erreichen. Der Entwickler ersetzt daher zustandsbehaftete durch Wertobjekte, wo immer er kann. In vielen Fällen erledigen die eingesetzten Frameworks den Rest. Wer an der vorderen Front der Multi-Core-Programmierung mitmischen oder mitlauschen möchte, kommt um eine gehörige Portion Mehrsprachigkeit nicht herum. Das beweisen auch die übrigen Artikel dieses Sonderhefts zur Multi-Core-Programmierung. (ane)

JOHANNES LINK

ist freiberuflicher Coach für agile Softwareentwicklung. Mit dem Thema Multi-Core-Programmierung beschäftigt er sich intensiv, seit er feststellen musste, dass sein neuer Rechner weniger Gigahertz hatte als sein vorheriger.

Literatur

[1] Brian Goetz; Java Concurrency in Practice; Addison-Wesley, 2006
[2] Edward A. Lee; The Problem with Threads; www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf
[3] Brian Goetz; Concurrency: Past and Present; www.infoq.com/presentations/goetz-concurrency-past-present
[4] Venkat Subramaniam; Programming Concurrency on the JVM. Pragmatic Programmers, 2011
[5] Joshua Bloch; Effective Java; Addison-Wesley, 2008
[6] Vaclav Pech; Flowing with the Data; www.jroller.com/vaclav/entry/flowing_with_the_data
[7] Herb Sutter; The Free Lunch is Over; www.gotw.ca/publications/concurrency-ddj.htm
[8] Johannes Link; Simplified Use of Locks in Groovy; blog.johanneslink.net/2011/10/25/simplified-use-of-locks-in-groovy

Alle Links: www.ix.de/ix1116006