Nebeneinander statt Durcheinander

Paradigmen der Nebenläufigkeit – Teil 2: Aktoren

Johannes Link

Das gängige Programmiermodell für die nebenläufige Programmierung unter Java versucht, die semantische Korrektheit von Zugriffen auf Objekte, die in mehreren Threads benutzt werden, überwiegend durch den Einsatz expliziter Synchronisation zu erreichen. Die Feinheiten dieses Ansatzes sind schwer zu erlernen und in komplexeren Systemen kaum fehlerfrei zu implementieren. Es existiert jedoch eine Reihe alternativer Programmierparadigmen, die versprechen, den Umgang mit nebenläufigen Problemstellungen einfacher zu machen. In diesem zweiteiligen Artikel werden einige dieser Ansätze vorgestellt und mit Beispielen in Java und dem Groovy-Framework GPars erläutert.

Was bisher geschah

Im ersten Teil dieser Miniserie [Link10] haben wir das fundamentale Problem des klassischen Java-Ansatzes zur Nebenläufigkeit unter die Lupe genommen: Die Technik, in nebenläufigen Threads gemeinsam genutzte Objekte mittels Locking zu schützen und zu synchronisieren, ist aufgrund fehlender Komponierbarkeit in größeren Programmen kaum fehlerfrei umsetzbar. Als erste Alternativen hierzu haben wir *Software Transactional Memory (STM)* und *Immutability* kennengelernt und sind dabei - der leichteren Umsetzbarkeit wegen – von Java auf Groovy umgeschwenkt.

Im zweiten und letzten Teil geht es um weitere Möglichkeiten, sich der Nebenläufigkeit zu widmen: *Aktoren*, *Agenten*, *Parallel Collections* und *Data Flows*. Dabei wird auch die Frage eine Rolle spielen: Wie parallelisiere ich denn meine Programmieraufgabe?

Nachrichtenaustausch

Ein ursprünglich aus der Telefonbranche stammendes Paradigma der nebenläufigen Entwicklung erlebt zurzeit eine Renaissance: Aktoren (*Actors*). Aktoren sind Objekte, die sich gegenseitig asynchrone Nachrichten hin und her schicken und weitgehend auf 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 sehr viele Aktoren parallel existieren und sich die verfügbaren Ressourcen effizient teilen können.

Bekannt wurden Aktoren durch die Sprache Erlang [erlang], die für sich in Anspruch nimmt, mithilfe des Aktoren-Konzepts hochverfügbare und fehlertolerante Systeme bauen zu können. Insider berichten beispielsweise von 99,9999999 % erreichter Verfügbarkeit bei einem Telefon-Switch der Firma Ericsson.

Auch für die JVM gibt es Aktoren: Akka [AKKA] und Jetlang [jetlang] sind Java-Bibliotheken, die Programmiersprache Scala [scala] bringt Aktoren in den Basis-Bibliotheken mit und für Groovy gibt es *GPars* [gpars], ein Projekt, das sich dem Gedanken verschrieben hat, die wesentlichen Nebenläufigkeitsparadigmen in einfacher und intuitiver Form verfügbar zu machen. Neben Aktoren unterstützt GPars auch *Agenten*, *Parallel Collections*, *Map-Reduce*, *Data Flows* und einiges mehr.

Die schwerfälligste, aber für den Java-Entwickler am leichtesten zu verstehende Variante, um in GPars Aktoren zu programmieren, ist es, diese von einer gemeinsamen Oberklasse abzuleiten und dort die `act`-Methode zu implementieren. Nehmen wir die `Bank`-Klasse aus [Link10] und verwandeln sie auf diese Weise in einen `BankActor` - unter Einsatz von GPars, Version 0.10 GA:

```
class BankActor extends AbstractPooledActor...
  void act() {
    loop {
      react { message ->
        switch(message) {
          case CreateAccount:
            createNewAccountFor(message.customer); break
          case GetAccount:
            accountByNumber(message.accountNumber); break
          case GetAllAccountNumbers:
            allAccountNumbers(); break
          case UpdateAccounts:
            update(message.accounts); break
          default:
          println "${this.class}: $message ???"
  } } } }

@Immutable class CreateAccount { String customer }
@Immutable class UpdateAccounts { Account[] accounts }
@Immutable class GetAccount { long accountNumber }
@Immutable class GetAllAccountNumbers {}
```

Hier geschieht in `act()` nichts weiter, als dass je nach Typ der eingehenden Nachricht eine andere fachliche Methode aufgerufen wird. Die verschiedenen Nachrichtentypen sind als eigene Wertklassen definiert, das ist jedoch lediglich eine Möglichkeit von vielen. Die eigentliche Fachlichkeit ist nun so einfach, wie zu erwarten:

```
class BankActor extends AbstractPooledActor...

  private accounts = [:]
  private lastAccountNumber = 0
  void createNewAccountFor(String customer) {
    Account account = ...
```

```
    reply account
  }
  void accountByNumber(long accountNumber) {
    reply accounts[accountNumber]
  }
  void allAccountNumbers() {
    reply accounts.keySet() as List
  }
  void update(Account[] accountsToUpdate) {
    if (accountsToUpdate.any { acc ->
      acc.version != accounts[acc.accountNumber]?.version
    }) {
      reply false
    } else {
      accountsToUpdate.each {acc ->
        accounts[acc.accountNumber] = acc.incrementVersion()
      }
      reply true
  } }
```

Abgesehen vom `reply`, das die asynchrone Antwort eines Aktors auf die Nachricht eines anderen darstellt und so das `return` ersetzt, bleibt alles beim Alten; nun ist keinerlei Synchronisationscode mehr notwendig, denn das Aktorenmodell sorgt für die Sequenzialisierung aller Operationen im Aktor. Ebenso direkt lässt sich `MoneyTransfer` in einen Aktor übersetzen:

```
class MoneyTransferActor...
  BankActor bank
  void act() {
    ...
    switch(message) {
      case TransferMoney: transferMoney(message); break;
  }
  void transferMoney(transfer) {
    while(true) {
      Account source =
        bank.sendAndWait(new GetAccount(transfer.from));
      Account target =
        bank.sendAndWait(new GetAccount(transfer.to));
      if (source.balance < transfer.amount) {
        reply new TransferFailure()
        break
      }
      source = source.withdraw(transfer.amount)
      target = target.deposit(transfer.amount)
      if (bank.sendAndWait(
          new Update(accounts: [source, target]))) {
        reply new TransferSuccess()
        break
  } } }
```

Die `sendAndWait`-Methode ermöglicht einen synchronen Kommunikationsmechanismus auf Basis asynchroner Nachrichten, denn sie blockiert den Aufrufer, bis eine Antwort eintrifft. Was auf den ersten Blick wie ein Hindernis für echte Skalierbarkeit aussieht, ist in Wirklichkeit keines; zwar muss der einzelne `MoneyTransferActor` auf die Antwort vom `BankActor` warten, es hindert uns jedoch niemand daran 2, 20 oder 2000 Überweisungs-Aktoren gleichzeitig zu starten. Allerdings haben wir mit der synchronen Aufrufvariante auch wieder das Tor für Verklemmungen (engl. deadlock) geöffnet; es ist eben nichts perfekt.

Aktoren erleben zurzeit auch deshalb ihren zweiten Frühling, weil sie von manchen als Wundermittel der nebenläufigen Programmierung verkauft werden: Sie fühlen sich beinahe an wie unsere lieb gewonnenen Objekte; sie sind unabhängig, skalierbar und verteilbar; sie machen es leichter, Race Conditions (dt. etwa Wettlaufsituation) und Verklemmungen zu 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; Verklemmungen und Co. sind zwar seltener, aber weiterhin möglich, und die Festlegung auf asynchrone Kommunikation macht manche Implementierung aufwendiger, als sie sein müsste. Der größte Fallstrick ist jedoch, dass Aktoren dann versagen, wenn man mit ihnen Probleme lösen möchte, die einen Konsens über gemeinsam verwendeten Zustand erfordern. Auf unser Beispiel übertragen heißt das: Ein Aktor pro Bankkonto ist vermutlich kein guter Ansatz, wenn wir Transaktionen über Kontogrenzen hinweg sicherstellen wollen, da wir dann in das Problemfeld verteilter Transaktionen geraten.

Streng geheim

Wenn es lediglich um das Schützen des inneren Zustands eines Objekts vor unsynchronisierten Zustandsveränderungen geht, dann macht uns das Konzept der Agenten (engl. *Agent*) das Leben leicht. Ein Agent fungiert als Wächter (engl. *Guard*) eines beliebigen anderen Objekts; Zugriff auf dieses Objekt erhält man nur, indem man den Zugriffscode – in Groovy eine Closure - an den Wächter schickt, um das bewachte Objekt zu manipulieren. Diese Zugriffsfunktionen werden garantiert nacheinander ausgeführt und können sich somit nicht in die Quere kommen. Das Konto lässt sich damit sehr einfach thread-sicher implementieren:

```
class Account...
  private balance = new Agent(0.0)
  void deposit(amount) {
    balance.send { value -> updateValue (value + amount)}
  }
  void withdraw(amount) {
    balance.send { value -> updateValue (value - amount)}
  }
  def getBalance() {
    balance.val
  }
```

So bequem Agenten sind, das Problem der objektübergreifenden Transaktion lösen sie nicht. Man kann zwar im Zugriffscode eines Agenten auch an einen anderen Agenten zustandsverändernde Closures schicken, eine Zusicherung über die Ausführungsreihenfolge bezüglich anderer Aufrufe bekommt man jedoch nicht, und damit auch keine atomare Änderung.

Nur parallel geht's auch schneller

Bislang haben wir uns in diesem Artikel auf die Vermeidung von Verklemmungen und Race Conditions konzentriert. Die andere Seite der Medaille ist keineswegs einfacher: Wie parallelisieren wir unser Programm so, dass Nebenläufigkeit überhaupt möglich ist, oder gar so, dass wir mit einer nennenswerten Performance-Verbesserung durch den Einsatz mehrerer Kerne rechnen können?

Grundsätzlich stehen uns eine Reihe von Möglichkeiten zur Verfügung: Falls wir es mit einem Standardproblem zu tun haben - wie etwa der Berechnung der Zahl ϖ ;-) -, dann fahnden wir im Netz nach einem parallel ausführbaren Algorithmus. In den anderen Fällen versuchen wir uns selbst an der Parallelisierung, meist nach einer der beiden Grundstrategien:

- *Parallelisierung des Kontrollflusses:* Indem wir die zeitlichen Vor- und Nachbedingungen aller Berechnungsschritte betrachten, ermitteln wir die Teile unseres Programms, die nicht in einer zeitlichen Abhängigkeit zueinander stehen, und starten dann an den entsprechenden Stellen unseres Systems eine neue Task. Dabei können uns Modelle (z. B.

Petri-Netze) unterstützen. Dadurch erreichen wir zwar unter Umständen eine spürbare Beschleunigung unseres Programms. Eine für große Mengen verfügbarer Rechenkernen skalierbare Lösung finden wir so jedoch meist nicht. Es sei denn, wir stoßen zufällig auf ein rekursiv zerlegbares Problem, für welches dann der Fork/Join-Ansatz aus dem JSR 166y [JSR166y] greift.

- *Datenparallelisierung:* Häufiger ist die Situation, dass wir es mit einer größeren Menge gleichartiger Daten zu tun haben, von denen jedes einzelne Datum auf gleichartige Weise durch unser Programm geschleust werden muss. In diesem Falle kommen uns weitere Nebenläufigkeitsparadigmen zur Hilfe.

Diese beiden Strategien stellen eine hilfreiche, jedoch stark vereinfachende Sicht auf das Problem der Algorithmenparallelisierung dar. Tatsächlich treffen wir in der Praxis oft auf eine Mischung aus Datenparallelisierung und Kontrollflussparallelisierung oder wir müssen uns zwischen mehreren denkbaren Strategien entscheiden. Eine ausführliche Darstellung der alternativen Muster zur Parallelisierung findet sich bei [Mattson05].

Parallel Collection Processing

Knüpfen wir mit einem Beispiel an die existierende Bank-Konto-Lösung an. Ziel ist die Berechnung der Standardabweichung des Saldos aller „Millionärskonten" von 1.000.000. Ein sequenzielles Programm würde dabei so vorgehen:

- Suche aus allen Konten diejenigen raus, welche einen Saldo von mindesten 1 Million aufweisen.
- Berechne für jeden Saldo dieser Menge: (saldo – 1000000)2 und merke dir alle Werte.
- Summiere alle Werte auf, dividiere das Ergebnis durch die Anzahl der beteiligten Konten und ziehe daraus die Quadratwurzel.

In diesem „Algorithmus" erkennen wir mindestens zwei Schritte, die potenziell von Datenparallelisierung profitieren können: das Herausfiltern der betroffenen Konten sowie die Berechnung der Formel im zweiten Schritt. GPars bietet uns für die typischen Mengen-Operationen wie **collect** und **findAll** parallele Varianten, die nebenläufig über einen Thread-Pool abgearbeitet werden:

```
GParsPool.withPool(numThreads) {
  def allBalances =
      bank.allAccountNumbers().collectParallel { accountNumber ->
    return bank.accountByNumber(accountNumber).balance
  }
  def millionaireBalances = allBalances.findAllParallel { balance ->
    return balance >= MILLION
  }
  def deviations = millionaireBalances.collectParallel { balance ->
    def diffToMean = balance - MILLION
    return (diffToMean * diffToMean)
  }
  count = deviations.size()
  average = Math.sqrt(deviations.sumParallel() / count)
}
```

Überraschend ist dabei, dass es auch eine **sumParallel**-Methode gibt. Dies funktioniert so, dass jeweils zwei Summanden in einem eigenen Thread addiert werden und das Ergebnis dieser Addition mit einem anderen Berechnungsergebnis wieder „gepaart" wird, bis schließlich die Gesamtsumme ermittelt wurde.

Abbildung 1 skizziert das Geschehen. Jede Linie stellt eine Berechnung dar, die von GPars einem Thread aus dem Pool zugewiesen wird. Die Zuordnung von Berechnungsschritten auf Threads erfolgt dabei entweder der Reihe nach über Javas Executor-Framework oder mit optimierten Verteilungsalgorithmen über den Fork/Join-Pool [JSR166y], der mit Java 7 Aufnahme ins JDK finden wird und im Augenblick noch als externe Bibliothek hinzugenommen werden muss.

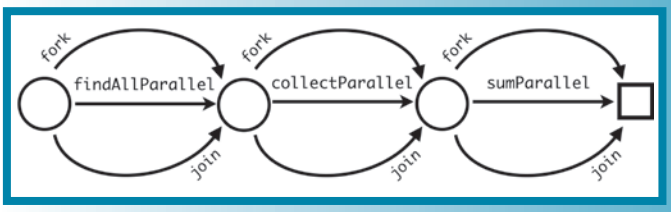

Abb. 1: Parallel Collections

Map-Reduce

An Abbildung 1 erkennt man auch, dass jedes Zwischenergebnis zunächst wieder in einer Ergebnismenge zusammengeführt wird, bevor man die nächste Welle paralleler Operationen starten kann. Dies wird insbesondere dann zum Performance-Engpass, wenn sich die Rechenkerne auf physikalisch verteilten Rechnern befinden und somit die Ergebnissammlung aufwendige Netzwerkkommunikation erfordert. Um dieses Problem zu mindern, hat Google das sogenannten Map-Reduce-Framework veröffentlicht und patentiert, das die Ausführung mehrerer aufeinanderfolgender Berechnungsschritte an einem Datum erlaubt, ohne zwischenzeitliche Konsolidierung des Gesamtergebnisses [map-reduce]. Auch für Map-Reduce stellt GPars Mittel bereit:

```
GParsPool.withPool(numThreads) {
  def deviations =
      bank.allAccountNumbers().parallel.map {accountNumber ->
    return bank.accountByNumber(accountNumber).balance
  }.filter { balance ->
    return balance >= MILLION
  }.map {balance ->
    def diffToMean = balance - MILLION
    return (diffToMean * diffToMean)
  }
  count = deviations.size()
  average = Math.sqrt(deviations.sum()/ count)
}
```

Der Unterschied zum vorigen Beispiel besteht zunächst mal in der Parallelisierung aller Kontonummern durch die Property **parallel**. Danach verwenden wir **map** statt **collectParallel**, **filter** statt **findAllParallel** und **sum** statt **sumParallel**. Der kleine syntaktische Unterschied hat jedoch eine große Wirkung bei der

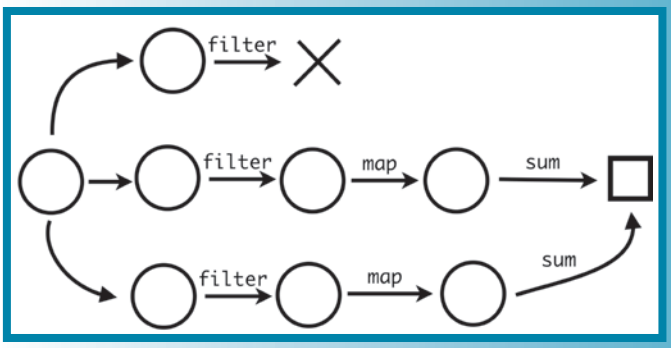

Abb. 2: Map-Reduce

Ausführung, wie Abbildung 2 verdeutlicht. Entscheidend ist, dass zwischen den einzelnen Filter- und Map-Schritten keine Zusammenführung der Ergebnisse in eine gemeinsame Menge notwendig ist und erst der Reduce-Schritt – in unserem Beispiel der `sum`-Befehl – zur Konsolidierung der Ergebnisse führt.

Parallele Collection-Methoden sind einfacher zu verstehen als Map-Reduce und haben auch ein breiteres Einsatzspektrum, da sie auch Zugriff auf andere Elemente der Collection erlauben. Dafür liefert der Map-Reduce-Ansatz eine in der Theorie höhere Parallelität. Ob diese in der Praxis von Bedeutung ist, können nur konkrete Performance-Messungen zeigen.

Data Flows

Sobald nebenläufige Aufgaben komplexere Abhängigkeiten haben, gerät man mit parallelen Mengenoperationen und Map-Reduce-Ansätzen in Schwierigkeiten, da bei ihnen lediglich am Ende der Berechnungskette eine Gesamtsynchronisation vorgesehen ist. Zu Hilfe kommt das Data-Flow-Konzept; ein Satz aus [Pech10] beschreibt die Idee sehr gut: *„Dataflow abstraction consists of concurrently run tasks communicating through single-assignment variables with blocking reads."* Hier nur ein ganz kleines GPars-Beispiel, um den Leser auf den Geschmack zu bringen:

```
def x = new DataFlowVariable()
def y = new DataFlowVariable()
def z = new DataFlowVariable()
task { z << x.val + y.val }
task { x << 40 }
task { y << 2 }
assert 42 == z.val
```

Mit `task {}` kann man in GPars asynchron Aufgaben starten. Die erste `task`-Zeile im Beispiel wird daher solange blockiert, bis die anderen beiden Tasks ihre Zuweisung erledigt haben.

Data-Flows haben die Eigenschaft, dass sie sich auch in einer nebenläufigen Umgebung deterministisch verhalten. Dies bedeutet u. a., dass eine Verklemmung, wenn sie denn auftritt, immer auftritt, auch wenn ich die einzelnen Aufgaben sequenziell starte. Damit wird das Erstellen aussagekräftiger Unit-Tests plötzlich wieder einfach. Darüber hinaus reagieren Data-Flow-Variablen in der Praxis wie unveränderliche Werte und benötigen somit keinen Synchronisationscode; *Race Conditions* und *Live Locks* sind unmöglich. Den größten Nachteil stellt wohl auch hier die geänderte Sicht auf unseren Lösungsraum dar; wir sind nicht gewohnt, in Datenflüssen zu denken und müssen das (wieder) erlernen, wenn wir den Nutzen aus dieser mächtigen Abstraktion ziehen wollen.

Fazit

Der Heilige Gral der Multicore-Programmierung ist noch nicht gefunden. In der Praxis müssen wir je nach Aufgabenstellung und Kontext aus der Menge der Angebote einen passenden Kandidaten auswählen. Für bestimmte Aufgaben eignen sich Fork/Join-Ansätze sehr gut, für andere bieten sich parallele Collections an, für wieder andere stellen Data-Flows ein natürliches Ausdrucksmittel dar; und auch Transactional Memory und Aktoren haben ihre Vorzüge.

Benötigen wir echtes nebenläufiges, transaktionales Verhalten über beliebige Objekte hinweg, dann müssen wir den Preis dafür zahlen. Entweder in Form der fehlerträchtigen manuellen Synchronisation oder als noch nicht ausgereifte STM-Technologie oder in der ungewohnten Variante versionierter unveränderlicher Zustandsobjekte. Eines jedoch ist klar: Je weniger *Shared mutable State* unser System besitzt, desto einfacher ist Thread-Sicherheit zu erreichen. Ersetzen wir daher zustandsbehaftete Objekte durch Wertobjekte, wo immer wir können. In vielen Fällen erledigen die eingesetzten Frameworks für uns den Rest.

Und schließlich noch eine letzte Bemerkung zur Wahl der passenden Programmiersprache. Theoretisch sind die hier gezeigten Konzepte mit reinem Java umsetzbar, schließlich läuft alles auf der JVM. In der Praxis zeigen viele der aktuellen JVM-basierten Sprachen ihre Vorteile gerade im Bereich der Nebenläufigkeit, allen voran Clojure, Scala und Groovy. Dies ist kein Zufall, sondern liegt an Javas unflexiblem Typsystem und der vergleichsweise aufwendigen Zeremonie für grundlegende Dinge wie Closures und interne DSLs. Wer daher an der vorderen Front der Multicore-Programmierung mitmischen oder mitlauschen möchte, kommt zurzeit um eine gehörige Portion Mehrsprachigkeit nicht herum.

Danksagung

Ich danke allen Reviewern des Artikels und Zuhörern meiner Vorträge zum Thema; insbesondere geht mein Dank an Dierk König für seine konstruktive Kritik und der Anregung zu den Grafiken im Teil 2 der Serie.

Literatur und Links

[AKKA] AKKA-Framework, `http://akkasource.org/`
[clojure] `http://clojure.org/`
[erlang] `http://www.erlang.org/`
[gpars] `http://gpars.codehaus.org/`
[groovy] `http://groovy.codehaus.org/`
[jetlang] `http://code.google.com/p/jetlang/`
[JSR166y] `http://artisans-serverintellect-com.si-eioswww6.com/default.asp?W9??`
[Lee06] E. A. Lee, The Problem with Threads, `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf`
[Link10] J. Link, Paradigmen der nebenläufigen Programmierung – Teil 1, in: JavaSPEKTRUM, 4/2010
[map-reduce] `http://labs.google.com/papers/mapreduce.html`
[Mattson05] T. G. Mattson et al., Patterns for Parallel Programming, Addison-Wesley, 2005
[Pech10] V. Pech, Flowing with the Data, `http://www.jroller.com/vaclav/entry/flowing_with_the_data`
[scala] The Scala Programming Language, `http://www.scala-lang.org/`

Johannes Link ist freiberuflicher Coach für Agile Softwareentwicklung. Mit dem Thema Multicore-Programmierung für die JVM beschäftigt er sich intensiv, seit er feststellen musste, dass sein neuer Rechner weniger Gigahertz hatte als sein vorheriger.
E-Mail: business@johanneslink.net.