Eine Einführung in die Parallele Scala Collections API

Florian Dobener, Betreuer: Prof. Dr. H.P. Gumm

September 27, 2012

Einleitung

Für die Programmiersprache Scala existiert wie in vielen anderen Sprachen auch eine Collections API um verschiedene komplexe Strukturen in einer Bibliothek zu vereinen. Dazu zählen beispielsweise Maps, Sets, Bags, sowie Arrays und Listen.

Um der fortschreitenden technologischen Entwicklung hin zu Mehrkernprozessoren Rechnung zu tragen, hat sich das Scala-Entwicklungsteam dazu entschlossen diese Strukturen automatisch zu parallelisieren. Dazu wurde eine parallele Collections API erschaffen, die vorhandene Datenstrukturen automatisch auf mehrere Prozessorkerne verteilen kann.

Diese Einführung dient dazu die Funktionsweise der parallelen Collections zu verstehen und erste Anwendungsfälle zu diskutieren. Darüber hinaus werden verschiedene Anwendungen und Übungen dargestellt um das Wissen zu festigen. Schlussendlich werden noch Ergebnisse verschiedener Benchmarks und deren Code vorgestellt um das Verhalten und den Geschwindigkeitsvorteil der Parallelisierung verstehen und in konkreten Fällen beurteilen zu können.

Da derzeit sehr stark an den parallelen Collections gearbeitet wird, ist für diese Abhandlung mindestens die *Scala Version 2.10-M6* [1] nötig, die bisher noch nicht für den produktiven Betrieb freigegeben ist. Als Leitfaden für diese Einführung wurde die Arbeit "On A Generic Parallel Collection Framework" [2] verwendet und kann für weitere Informationen zu Rate gezogen werden.

1 Funktionsweise & Ansatz

Die parallelen Collections sind an die normalen Collections angelehnt und verwenden generische Überklassen, die für beide Ansätze funktionieren. Daher ist auch eine einfach Konversion zwischen den sequentiellen und parallelen Strukturen möglich.

In Bild 1 sind die Abhängigkeiten der verschiedenen Klassen dargestellt. Wie bereits von der sequentiellen API bekannt, ist die Oberklasse der Trait *Traversable*. Davon erbt der Trait *Iterable*, aus dem die bekannten Strukturen erzeugt werden können. Durch die abstrakte Herangehensweise kann der Code leicht gewartet und Updates schnell eingespielt werden. Von Verbesserungen in diesen Klassen profitiert sowohl die parallele als auch die sequentielle API.

Besonders erwähnenswert ist der Trait *Splitter*, der zusammen mit dem Trait *Combiner* die Grundlage der Parallelisierung bildet. Der Übersicht wegen werden wir uns auf diese beiden zusätzlichen Typen zur Erklärung der parallelen API beschränken[2].

[1] Der aktuelle Milestonerelease ist unter `http://www.scala-lang.org/downloads#Milestones` zu finden

[2] Zusätzliche Informationen zum Aufbau der kompletten parallelen API findet sich unter `http://docs.scala-lang.org/overviews/parallel-collections/architecture.html`.

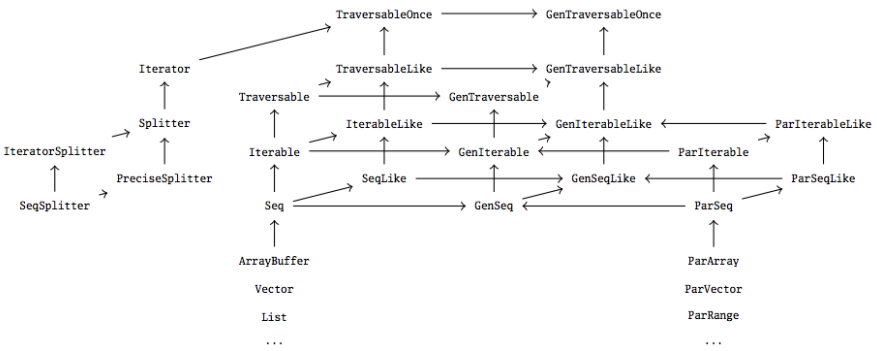

Figure 1: Aufbau der Scala Collections API [1]

1.1 Verwendung der API

Die API ist so gestaltet, dass der Programmierer nicht sehr viel an seinem Programmierstil ändern muss und eine einfache Integration in den normalen Arbeitsablauf gewährleistet ist. Die Benutzung der parallelen Strukturen erfolgt analog zu den sequentiellen Vorbildern und bietet den gleichen Funktionenumfang wie eben diese. Weiterhin gibt es die beiden Funktionen *.par* und *.seq* die verwendet werden können um zwischen parallelen und sequentiellen Versionen umzuschalten. Um beispielsweise einen einfachen Test auf alle Primzahlen zwischen 2 und 10000 durchzuführen kann eine einfache Scala Liste durch das *.par* Keyword zur parallelen Verarbeitung vorbereitet werden.

```
val list = (2 to 10000).toList
list.par.filter{ n =>
  for(i <- 2 until n if n % i == 0)
    return false
  true
}
```

Allerdings sollte in diesem Fall aus Performance Gründen die Liste direkt parallel initialisiert werden. Für das Aufrufen von *.par* wird in diesem Fall $O(n)$ Komplexität beansprucht. Diese Komplexität trifft für die meisten Strukturen zu, nur wenige sind $O(1)$. Daher sollte hier direkt die Liste mit

```
val list = new ParSeq((2 to 10000):_*)
```

initialisiert werden. Damit wird das Aufrufen der *.par* Funktion überflüssig und die Filter Funktion kann direkt auf die Liste angewendet werden.

1.1.1 Probleme und Nichtdeterminismus

Da die parallelen Collections sich sehr ähnlich zu den normalen Verhalten kann man leicht die parallele Verarbeitung vergessen. Beim Verwenden der API sollte man immer darauf achten Seiteneffekt und Nicht-Assoziative Operatoren zu vermeiden.

```
var sum = 0
val list = (1 to 100).toList.par
list.foreach(sum += _)
```

Der obige Code wird jedes mal eine andere Zahl heraus geben. Dies liegt daran, dass es bei der parallelen Verarbeitung dazu kommt, dass gleichzeitig auf die gleiche Variable zugegriffen wird und dabei zwangsweise eine der Aktualisierungen verloren geht. Daher sollten Seiteneffekte zwingend vermieden werden bzw. sichergestellt werden, dass die parallele Verarbeitung nichts am Ergebnis verändert.

```
list.reduce(_-_)
```

Obiger Code wird ebenfalls nichtdeterministisch ausgeführt werden. Da jede Sequenz in kleinere Untergruppen aufgeteilt wird die in einem eigenen Thread ausgeführt werden und diese unabhängig voneinander ausgeführt werden und die Ergebnisse dieser Threads am Ende

wieder verrechnet wird erhalten wir hier das falsche Ergebnis, da man das Aufteilen in Untergruppen als eine Art Klammersetzung betrachten kann. Da diese allerdings nach einer Heuristik ausgeführt wird bzw. auf jeden Fall in kleinere Untergruppen klammert erhalten wir bei diesem nichtassoziativen Operator ein nichtdeterministisches Verhalten.

1.2 Splitter & Combiner

Die beiden Traits *Splitter* und *Combiner* bilden die Grundlage der parallelen Bibliothek. Wie der Name schon andeutet sorgen diese beiden Strukturen dafür, dass die durchlaufbaren Strukturen in kleinere Pakete aufgeteilt werden, was durch den *Splitter*-Trait geschieht, und nach der Ausführung des Codes mit dem *Combiner*-Trait wieder zusammen geführt werden. Durch die Scala typische hohe Abstraktionsebene dieser Traits kann eine eigene parallele Struktur mit wenig Aufwand konstruiert werden.

1.2.1 Splitter

Der Trait *Splitter* wird dazu verwendet die Datenstrukturen in kleinere Elemente aufzuteilen. Dazu stellt der Trait die Methode *split* bereit, die eine Sequenz von neuen Splittern zurück gibt. Der alte Splitter ist danach nicht mehr benutzbar. Ein weiteres Merkmal ist außerdem, dass der Splitter in den parallel Collections als Iterator verwendet wird. Dies gewährleistet, dass das Durchlaufen durch die Elemente ebenfalls parallelisiert wird.

```
trait Splitter[T] extends Iterator[T] {
  def split: Seq[Splitter[T]]
}
```

Das Konzept funktioniert analog wie die sequientiellen Iteratoren. Die verwendete Klasse sollte hier vom Trait *ParIterable* erben, der es erfordert die Funktion *seq*, die eine Konversion in die analoge lineare Klasse durchführt, und die Funktion *size*, die die Anzahl der Elemente zurück gibt, zu definieren. Die wichtigste Funktion ist allerdings *splitter*, die den entsprechenden Splitter der Klasse zurück liefert. Diese ersetzt die Funktion *iterator* der normalen Collections vollständig und ist deren paralleles Gegenüber.

1.2.2 Combiner

Nachdem man eine Klasse mit einem Splitter ausgestattet hat ist es zwar möglich diese zu durchlaufen, etwa mit Funktionen wie foreach o.ä., allerdings erhält man bei Funktionen, wie filter, die den Datentyp als Rückgabewert liefern sollten, die übergeordnete Klasse zurück. Hier kommt nun der *Combiner* ins Spiel, der dafür sorgt, dass eine Struktur wieder zusammen gefügt werden kann. Er ist das parallele Gegenstück zum *Builder*.

```
trait Combiner[T, Coll]
      extends Builder[T, Coll] {

  def combine(other: Combiner[T, Coll])
              :Combiner[T,Coll]
}
```

Die wichtigste Funktion *combine* des Standard-Combiners sorgt dafür, dass die aktuelle Struktur mit dem übergebenen Combiner *other* zusammengefügt wird. Als Rückgabewert erhält man dann einen Combiner, der alle Elemente beider alten Combiner erhält. Sowohl der aktuelle, wie auch der übergebene Combiner other werden dabei invalidiert.

1.3 Erstellen von eigenen Datenstrukturen

Das Erstellen von eigenen Datenstrukturen wird anhand eines Beispiels beschrieben. Wir wollen dazu eine doppelt verkettete Liste *ParList* erstellen, die vollständig in die Scala Collection API integriert werden soll. Die Implementierung orientiert sich dabei stark an dem *ParString* Beispiel aus der Scala Dokumentation[3]. Dazu werden natürlich die Traits Splitter und Combiner benutzt und in diesem Beispiel als *ParListCombiner* und *ParListSplitter* implementiert.

```
class ParList[T](elems:T*)
 extends ParSeq[T]
 with GenericParTemplate[T,ParList]
```

```
with ParSeqLike[T,ParList[T], List[T]]{

private var default:T = _
private var root =
  new ParListElem(null, null,default)
root.next = root
root.prev = root

var cur = root
for(e <- elems){
  cur.next =
    new ParListElem(root,cur,e)
  cur = cur.next
  root.prev = cur
}

class ParListElem(
      var next:ParListElem,
      var prev:ParListElem,
      var data:T){}
...
```

Wir verwenden hier eine ParSeq, da die doppelte verkettete Liste eine normale Sequenz ist und dieser Trait bereits viele Funktionen zum Durchlaufen und Bearbeiten der Liste mitbringt. Der Trait GenericParTemplate signalisiert, dass die Klasse ein Companion-Objekt besitzt, dass bestimmte Funktionen bereit stellt. Dies wird näher bei der Erklärung des Companion-Objekts ausgeführt. ParSeqLike wird dazu verwendet um darzustellen welche Objekttypen durch Funktionen, welche eine neue Struktur zurück geben, erzeugt werden. Außerdem wird dadurch angezeigt welches das sequentielle Gegenstück, hier eine einfache List, zur Klasse ist.

Als Ausgangsanker wird der Zeiger root verwendet, der ein ParListElem-Objekt ist, dabei zeigt der Zeiger *prev* immer auf das letzte Element der Liste und der Zeiger *next* auf das erste Element. Die for-Schleife wird dazu verwendet um die Initialisierungselemente mit Komplexität $O(n)$ in die Liste zu schreiben.

Diese drei Traits verlangen die Implementierungen der Funktionen *length*, *apply* und *seq*. *length* und *apply* liefern dabei die Länge der Liste bzw. das Element an der entsprechenden Stelle zurück. Auf die direkte Implementierung wird wegen Platzgründen an dieser Stelle verzichtet[3]. Interessant ist hier die neue Funktion seq, die das entsprechende sequentielle Gegenstück unsere Liste zurück geben soll. In diesem Fall ist das die normale Scala-List.

```
def seq = {
  var ret = List[T]()
  for(i <- iterator) ret :+= i
  ret
}
```

Hier wird einfach der Iterator verwendet um die aktuelle Struktur zu durchlaufen und die entsprechenden Elemente an die Liste anzuhängen. Der Iterator wird durch die ParListSplitter Klasse implementiert. Dazu muss die Funktion *splitter* definiert werden.

```
def splitter =
  new ParListSplitter(0,length)
```

Der ParListSplitter erbt vom Trait SeqSplitter, der neben den normalen Splitter-Methoden *hasNext*, *next* und *split* auch noch die zusätzlichen Funktionen *remaining*, *dup* und *psplit* definiert haben muss.

```
class ParListSplitter(
      private var pos:Int,
      private var ntl:Int)
      extends SeqSplitter[T]{

  var loc = root.next

  for(i <- 0 until pos)
    loc = loc.next
  ...
```

Der Parameter *pos* liefert die aktuelle Position des Zeigers und *ntl* liefert die aktuelle Endposition. So ist es möglich die Liste aufzuteilen ohne direkt die Elemente zu löschen oder zu verschieben. Zur Initialisierung wird nun der Zeiger *loc* erstellt und auf die aktuelle Position pos gebracht. Das Erstellen der Klasse ist somit also Linear in der Variable pos. Mit den beiden Variablen kann nun die Funktion *hasNext* definiert werden.

[3]Der komplette Quelltext kann unter `https://github.com/domna/ScalaParallelCollections/blob/master/ParList.scala` angesehen werden

```
def hasNext = pos < ntl

def next:T = {
  if(!hasNext)
    throw new ...
  pos += 1
  val d:T = loc.data
  loc = loc.next
  d
}
```

Damit repräsentiert jeder Splitter einen Iterator auf dem Intervall $[pos, ntl[$. Weiterhin brauchen wir die Methoden *remaining*, die die verbleibende Anzahl an Elementen im aktuellen Splitter angibt, sowie *dup*, die eine Kopie des aktuellen Splitters liefert.

```
def remaining = ntl - pos
def dup = new ParListSplitter(pos,ntl)
```

Nun kommen wir zu den eigentlich interessanten Funktionen *split* und *psplit*. split soll in unserer Implementierung die Struktur in zwei etwa gleich große neue Splitter aufteilen. Dem gegenüber liefert psplit eine Anzahl von Splittern mit fest vorgegebenen Größen.

```
def split:Seq[ParListSplitter] = {
  val rem = remaining
  if(rem >= 2)
    psplit(rem / 2, rem - rem / 2)
  else Seq(this)
}
```

Wenn der aktuelle Splitter noch mehr als zwei Elemente besitzt, dann wollen wir die Struktur in einen neuen Splitter mit jeweils der Hälfte aller Elemente aufteilen, dazu wird die Funktion psplit verwendet. Besitzt der Splitter nur noch ein Element wird einfach der Splitter selbst zurück gegeben.

```
def psplit(sizes:Int*)
    :Seq[ParListSplitter] = {

  val splitted =
    new ArrayBuffer[ParListSplitter]

  for(s <- sizes){
    val next = (pos + s) min ntl
    splitted +=
      new ParListSplitter(pos,next)
    pos = next
  }
  if(remaining > 0)
    splitted +=
      new ParListSplitter(pos,ntl)
  splitted
}
```

Der ArrayBuffer wird hier als Sequenz genutzt um die verschiedenen Teile des Aufteilens zu speichern. Dazu werden alle Elemente der sizes Sequenz durchlaufen und die nächste Position ermittelt. Mit den neuen Werten kann nun ein neuer Splitter erstellt werden, der dem ArrayBuffer hinzugefügt wird. Zuletzt wird noch überprüft ob noch Elemente übrig sind und gegebenenfalls ein Restsplitter erstellt.

Damit ist nun der ParListSplitter unserer frisch erstellten Liste vollständig definiert und einfache Funktionen auf die Liste können bereits parallel ausgeführt werden (bspw. paralleles durchlaufen, Funktionen ohne Builder, wie forall etc. können parallel aufgerufen werden). Für eine voll integrierte Liste benötigen wir allerdings noch das Objekt ParListCombiner, der dafür sorgt, dass eine neue ParList aus einzelnen Sequenzen zusammengesetzt werden kann.

```
class ParListCombiner[T]
      extends Combiner[T,ParList[T]]{

  var sz = 0
  var res = new ParList[T]()

  def size = sz
  ...
```

Die Klasse beerbt den Trait Combiner, der zwei Typparameter besitzt. Der erste ist der Inhaltstyp der Kollektion und der zweite ist der Rückgabetyp, den der Combiner liefern wird. Zum Arbeiten mit dem Combiner werden die internen Variablen sz, für die Größe des Combiners und res, in der die Inhalte vorgehalten werden, verwendet. Im obigen Codelisting kann man darüber hinaus auch die erste Funktionsdefinition sehen, die der Trait Combiner verlangt. Für die Größe des Combiners wird einfach die interne Variable sz zurück gegeben,

die bei jedem Update der Klasse verändert wird.

Darüber hinaus verlang der Trait noch vier weitere Funktionen: $+=$ um ein Element hinzu zu fügen, *clear* um den Combiner zu leeren und neu zu initialisieren, *result* gibt den aktuellen Inhalt des Combiners als ParList zurück und *combine* die einen zweiten Combiner entgegen nimmt und diesen mit dem aktuellen verschmilzt. Die Methode hat genauso wie die split Methode die Eigenschaft, dass beide alten Combiner nach dem Aufruf nicht mehr verwendet werden können.

```
def +=(elem:T):this.type = {
  res.append(elem)
  this
}

def clear:Unit = res = new ParList[T]()

def result:ParList[T] = res

def combine[U <: T, NewTo >: ParList[T]]
    (other: Combiner[U,NewTo])
    :Combiner[U,NewTo] = {
  if(other eq this) this
  else {
    val that =
      other.
        asInstanceOf[ParListCombiner[T]]
    sz += that.sz
    for(i <- that.res) this += i
    this
  }
}
```

Die Methode $+ =$ fügt dabei ein Element mit der Hilfsfunktion *append* in die ParList Struktur ein. Die append Funktion realisiert das hinzufügen eines Elements durch das umbiegen des last-Zeigers der Liste, die Komplexität beträgt also immer $O(1)$. Allerdings wird dabei der an sich unveränderbare Datentyp durch umbiegen der Zeiger verändert. Da diese Funktion allerdings private deklariert ist und somit kein Zugriff von außen möglich ist ist diese Funktion in diesem Kontext geschützt und die schnellste Möglichkeit.

Die Methode result liefert einfach die ParList zurück, die intern verwendet wird.

combine testet zunächst ob der übergebene Combiner gleich mit dem aktuellen ist und gibt dann den aktuellen Combiner zurück. Dies ist die Scala Konvention zur Combine Methode und sollte bei jedem eigens erstellten Combiner eingehalten werden. Sofern die beiden sich unterscheiden wird zuerst die übergebene Struktur als ParListCombiner instanziert, damit die Größe sz gesetzt werden kann. Nach dem Setzen der Größe werden schlicht die Elemente des übergebenen Combiners in den aktuellen geschrieben und anschließend die Klasse durch this zurück gegeben.

2 Benchmarks

Um die Geschwindigkeitsverbesserung der Parallelisierung beurteilen zu können wurden zwei Benchmarks durchgeführt. Dazu wurde die Ausführungszeit in Abhängigkeit der Elementanzahl bei verschiedenen Parallelitätsebenen[4] untersucht. Da wie bereits schon beschrieben die Parallelisierung durch die Aufteilung in verschiedene Threads einen Overhead produziert kommt der Laufzeitvorteil erst bei vielen Elementen und großem Rechenaufwand in jedem Schritt zum tragen.

2.1 Durchführung

Für die Benchmarks wird der *Benchmark* Trait verwendet (scala.testing.Benchmark). Die wichtigste Methode ist dabei *run*, die vom Test definiert werden muss und den Rückgabewert Unit sowie keine Parameter besitzt. Sie enthält den Programmcode, der für die Geschwindigkeitsmessung zu Rate gezogen wird. Jedweder Code außerhalb dieser Funktion wird nicht mit getestet. Der Aufruf des Programms wird mit zwei Kommandozeilenparametern gestaltet, von denen einer optional ist. Der erste gibt an, wie oft die Funktion *run* pro Durchlauf aufgerufen werden soll. Der

[4]Die Parllelitätsebene bestimmt die Anzahl der Aufteilungen, sprich Aufrufe von *split*, der entsprechenden Liste.

zweite optionale Parameter gibt an wie oft die Benchmark komplett ausgeführt werden soll.

Die mehrfache Ausführung hat zwei Hauptgründe[5]: Erstens wird damit das Aufrufen des GarbageCollectors amortisiert, da nach jedem Aufruf von *run* eine garbage collection erzwungen wird. Durch mehrfaches Ausführen fällt diese Umgebungsbedingung also nicht mehr ins Gewicht.

Der zweite Grund ist die statistische Schwankung der Ergebnisse. Da der Computer natürlich nicht nur die Benchmark ausführt sondern noch ein komplettes Betriebssystem am laufen halten muss schwanken die Ergebnisse natürlich ein wenig, da bei unterschiedlichen Ausführungen auch unterschiedlich viel Priorität auf die Benchmark gelegt wird. Daher werden gleich mehrere Durchläufe gestartet und später dann der Mittelwert gebildet.

Weiterhin spielen auch die Optimierungen des Java Codes durch die JVM eine Rolle. Aus diesem Grund wird für die Benchmarks die Server Version der JVM verwendet, da diese aggressivere Optimierungen als die Client Version durchführt. Ein Aufruf des Programms kann dann durch

```
java -server
-cp .:$SCALA_HOME/lib/scala-library.jar
-Dlength=10000 -Dpar=4 <Class-File>
```

gestartet werden. Dabei wäre die Länge der Liste dann 10000 und die Parallelitätsebene 4.

Der nachfolgende Test führt jeweils zehn mal die Funktion *run* aus und die ganze Benchmark wird ebenfalls zehn mal wiederholt. Summa summarum erhalten wir also 100 Gesamttests. Diese Tests werden jeweils mit unterschiedlicher Gesamtzahl der Elemente und verschiedenen Parallelitätsstufen (1, 2, 4, 8) durchgeführt. Außerdem wird der Test natürlich auch sequentiell durchgeführt.

2.2 Sqrt Benchmark

In diesem Test wird eine TrieMap bzw. ParTrieMap mit allen Elementen von 1 bis n gefüllt, dabei ist n Abhängig von den Startbedingungen des Programms. Nun wird mithilfe des Heron-Verfahrens jede Wurzel von 1 bis n bestimmt, so dass die Map die Abbildung $n \to \sqrt{n}$ enthält. Als Konvergenzbedingung wird benutzt, dass der vorherige Wert nicht mehr als $0,001$ vom aktuellen Wert abweicht.

Die Implementierung[6] der *run*-Methode erfolgt wie hier dargestellt:

```
for((n,s) <- testing){
  var rt = s.toDouble
  var rtOld = 0.0
  while(math.abs(rt - rtOld) > 0.001){
    rtOld = rt
    rt = 0.5 * (rtOld + n/rtOld)
  }
  testing(n) = rt
}
```

Für die parallelen und sequentiellen Durchläufe wurden jeweils nur die Initialisierungen geändert. Für die lineare Bearbeitung wird ein TrieMap Objekt erzeugt und mit den Elementen von 1 bis *len* initialisiert. *len* ist dabei ein Kommandozeilenparameter, mithilfe dessen die Länge der Map von außerhalb gesetzt werden kann.

```
TrieMap[Int,Double]
       ((0 until len) zip ((0 until len)
       map (_.toDouble)): _*)
```

Für die parallel Version wird dies analog initialisiert. Darüber hinaus wird noch die Parallelitätsebene durch

```
new ForkJoinTaskSupport(
    new ForkJoinPool(parLvl))
```

an das Feld *tasksupport* der ParTrieMap zugewiesen. Dabei kennzeichnet parLvl natürlich die Parallelitätsebene, die wieder extern gegeben wird.

[5] Die Tests wurden auf Basis von `http://docs.scala-lang.org/overviews/parallel-collections/performance.html` konstruiert

[6] Der komplette Quelltext kann unter `https://github.com/domna/ScalaParallelCollections/blob/master/SqrtBench.scala` abgerufen werden.

Die Ergebnisse des Tests können in Bild 3 für zwei Kerne und in Bild 2 für einen Kern betrachtet werden. Dort sind jeweils die Anzahl der TrieMap Elemente gegen die Ausführungszeit eines Durchlaufs in ms dargestellt. Die verschiedenen Graphen bezeichnen damit die verschiedenen Parallelitätsebenen (1, 2, 4, 8) bzw. Seq kennzeichnet den sequentiellen Durchlauf.

2.3 isPrim Benchmark

Dieser Test verwendet die Seq bzw. ParSeq Struktur der Scala collections API. Dabei werden alle Zahlen von 1 bis n in die entsprechende Datenstruktur eingetragen und durch einen einfachen Teilbarkeitstest auf Primzahlen überprüft. Durch die *filter*-Funktion werden nur die Zahlen in der Struktur behalten, die tatsächlich Primzahlen sind.

```
prims = testing filter { n =>
  var teilbar = false
  for(i <- 2 until n if n % i == 0)
    teilbar = true
  !teilbar
}
```

Diese Implementierung ist zwar sehr ineffizient, liefert aber für den Test ein stabiles Laufzeitverhalten von $O(n)$. Der restliche Quelltext wurde analog zum Sqrt Benchmark implementiert[7]. Die Ergebnisse sind in Bild 5 für zwei Kerne sowie in Bild 4 für einen Kern festgehalten. Hier sind wieder die Anzahl der Elemente gegen die Ausführungszeit in ms aufgetragen. In den verschiedenen Graphen sind die Durchläufe mit verschiedenen Parallelitätsebenen (1, 2, 4, 8) bzw. dem sequientiellen Durchlauf Seq dargestellt.

2.4 Fazit

Man erkennt deutlich, dass die parallele Collections API einen Geschwindigkeitsvorteil auf mehreren Prozessorkernen bietet. Dabei muss allerdings die Größe der Struktur einen bestimmten Wert überschreiten um den Overhead der Thread Erzeugung aufzuwiegen. Außerdem hängt die Verbesserung der Geschwindigkeit auch mit dem Arbeitsaufwand der Berechnung zusammen. Im Vergleich der Sqrt zur isPrim Benchmark fällt auf, dass die Bestimmung der Primzahlen ein sehr viel stärkeren Vorteil erbringt, da die Berechnung gerade bei Großen Zahlen sehr langwierig ist.

Weiterhin kann man erkennen, dass bei nur einem Kern ein enormen Geschwindigkeitsnachteil entstehen kann, wenn die Parallelitätsebene zu hoch gewählt wird. Da allerdings die Ebene normalerweise automatisch durch Scala gesetzt wird indem die Anzahl der Prozessorkerne abgerufen wird kann man diesen Faktor vernachlässigen. Demnach fällt bei einem Einkernprozessor nur die Parallelitätsstufe 1 ins Gewicht, da diese automatisch gewählt würde. Im direkten Vergleich kann man feststellen, dass der Overhead relativ gering ist und der Geschwindigkeitsnachteil der parallelen API auf nur einem Kern nicht so stark ins Gewicht fällt. Insbesondere liegt die Geschwindigkeit der parallelen Berechnung hier sogar über der der sequentiellen Bearbeitung (isPrim Benchmark auf einem Kern ab ca. 4500 Sequenzelementen). Es ist also ohne Bedenken möglich die parallelen Strukturen allgemein einzusetzen und damit Mehr- wie auch Einkernprozessoren zu bedienen.

[7] Der volle Quelltext ist unter `https://github.com/domna/ScalaParallelCollections/blob/master/isPrimBench.scala` zu finden.

Referenzen

[1] Aleksandar Prokopec, Phil Bawgell, Tiark Rompf, and Martin Odersky. http://docs.scala-lang.org/resources/images/parallel-collections-hierarchy.png.

[2] Aleksandar Prokopec, Phil Bawgell, Tiark Rompf, and Martin Odersky. http://infoscience.epfl.ch/record/165523/files/techrep.pdf.

[3] Aleksandar Prokopec and Heather Miller. http://docs.scala-lang.org/overviews/parallel-collections/custom-parallel-collections.html.

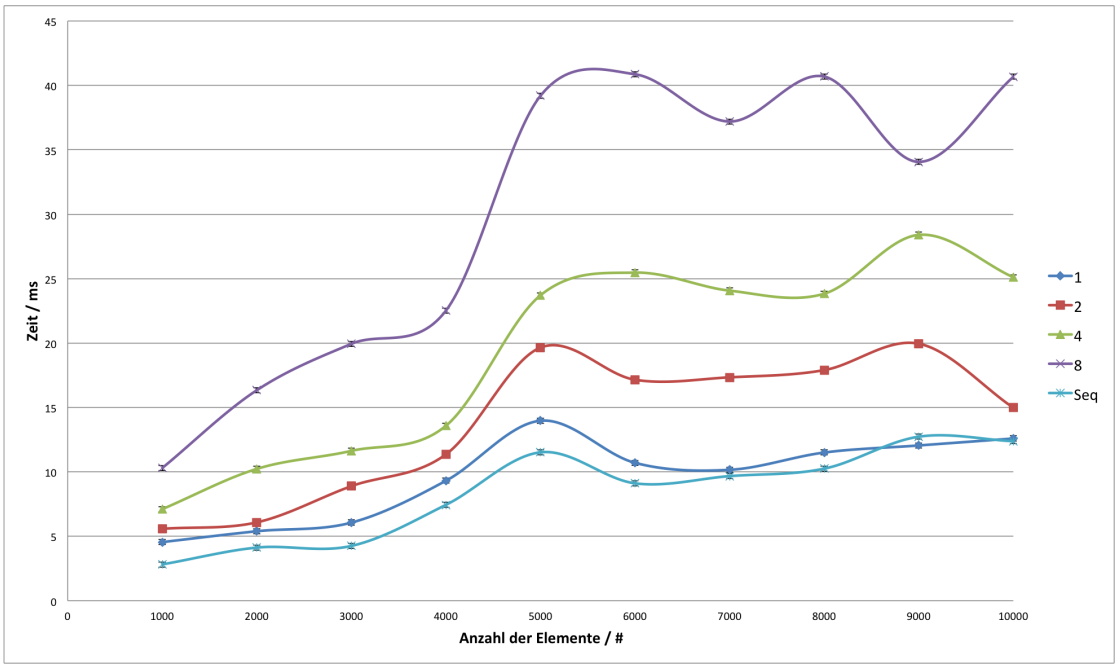

Figure 2: Sqrt Benchmark auf einem Intel Celeron 2.2 Ghz (1 Kern, Java 1.6)

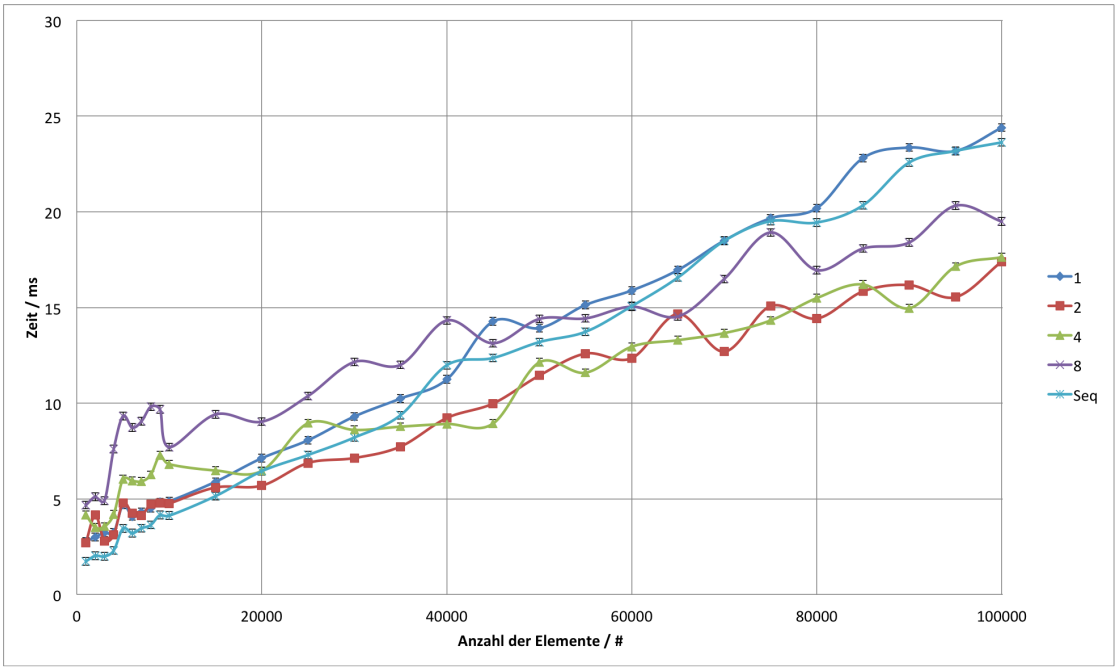

Figure 3: Sqrt Benchmark auf einem Intel Core i5 2.3 Ghz (2 Kerne, Hyperthreading, Java 1.7)

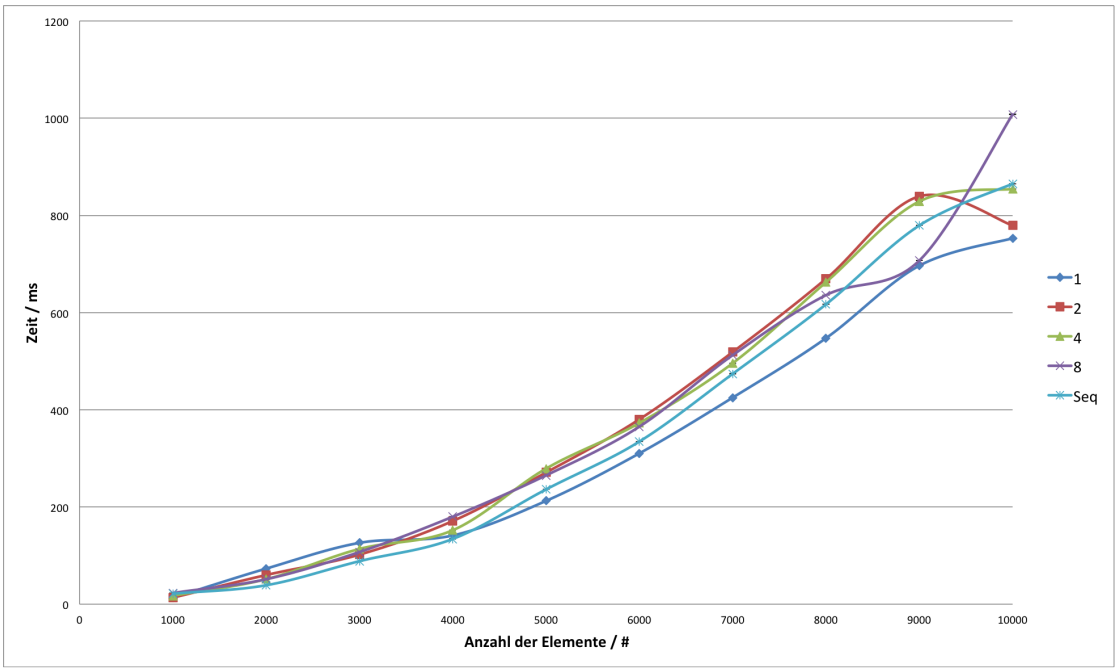

Figure 4: isPrim Benchmark auf einem Intel Celeron 2.2 Ghz (1 Kern, Java 1.6)

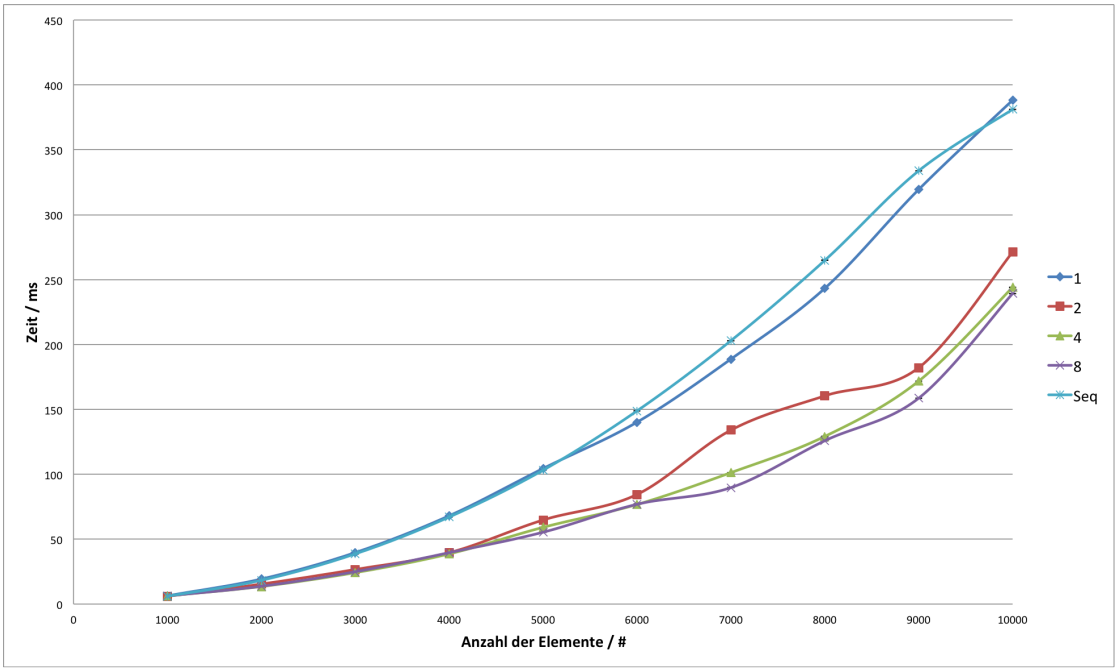

Figure 5: isPrim Benchmark auf einem Intel Core i5 2.3 Ghz (2 Kerne, Hyperthreading, Java 1.7)