1. Erweiterungen des Bounce-Beispiels

1. Ball immutable gemacht
2. Die Liste an Bällen zu einer ConcurrentHashMap gemacht
    a. Um die Identität der Bälle in die Map aus zu lagern
    b. Um die atomaren Operationen der ConcurrentMap zu verwenden
3. BallMover als Runnable Implementiert
4. BallMover benutzt die compute methode der ConcurrentMap um einen Ball atomar durch den bewegten Ball zu ersetzen.
5. Buttons für schneller langsamer und Stop zum Gui hinzugefügt
6. Schneller und langsamer ersetzen die bälle durch “replace all” (jedes einzelne ersetzen ist atomar, aber das ersetzen aller ist nicht atomar)
7. Stop button interupted alle Threads

`de.hs_augsburg.nlp.one.ball.BounceBallAdjustable`

Die Schleife die einen Ball bewegt:

```
for (int i = 1; i <= 1000; i++) {
    if (Thread.currentThread().isInterrupted())
        return;
    ballAtom.compute(ballIdentity, (key, ball) -> ball.move());
    Thread.sleep(5);
    box.repaint();
}
```

Alle Bälle schneller machen:

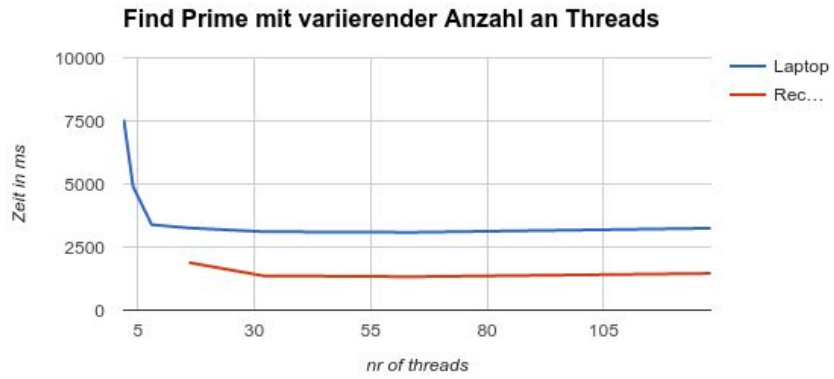
```
allBalls.replaceAll((k, v) -> v.adjustSpeed(2));
```

2. Parallelisierung von isPrime()

Ergebnis ist `de.hs_augsburg.nlp.one.prime.TaskedPrimeCheck`

- Der Bereich der Teiler wird in 1,000,000 “Task[s]” eingeteilt, minimale größe eines Tasks ist 500
- Die Bereiche werden mit parallel map in einen Boolean umgewandelt und mit anyMatch terminiert

PMapPrimeCheck benutzt parallel map, aber auf einzelnen potentiellen Teilern. Damit ist der einzelne Task zu klein und es entsteht mehr Overhead.



3. Miller-Rabin-Algorithmus

Unsere Implementierung ist eine adaption von:

- https://rosettacode.org/wiki/Miller%E2%80%93Rabin_primality_test#Python:_Proved_correct_up_to_large_N
- https://rosettacode.org/wiki/Miller%E2%80%93Rabin_primality_test#Java
- https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html#isProbablePrime-int-

Unsere implementation ist langsamer als die Implementation im JDK.
Unsere parallel version ist nicht schneller als unsere synchrone version.
Zeiten von 1_000_000_000_000_000_000L bis 1_000_000_000_000_000_006L:
- 2 Core Laptop:
Simple: 7340

Implementation	Laptop Zeit in ms	Rechentier Zeit in ms
Simple	7340	24159
MilerRabin(Sync)	53	44
MilerRabin(JDK)	36	6
MilerRabin(Parallel)	42	52
TaskedParallel	3263	3854
ParallelMap	3619	1080

Unsere fertige Implementation ist
`de.hs_augsburg.nlp.one.prime.MillerRabinPrimalityTestParallel`

4. Parallelisierung in Clojure

Die Clojure variante:

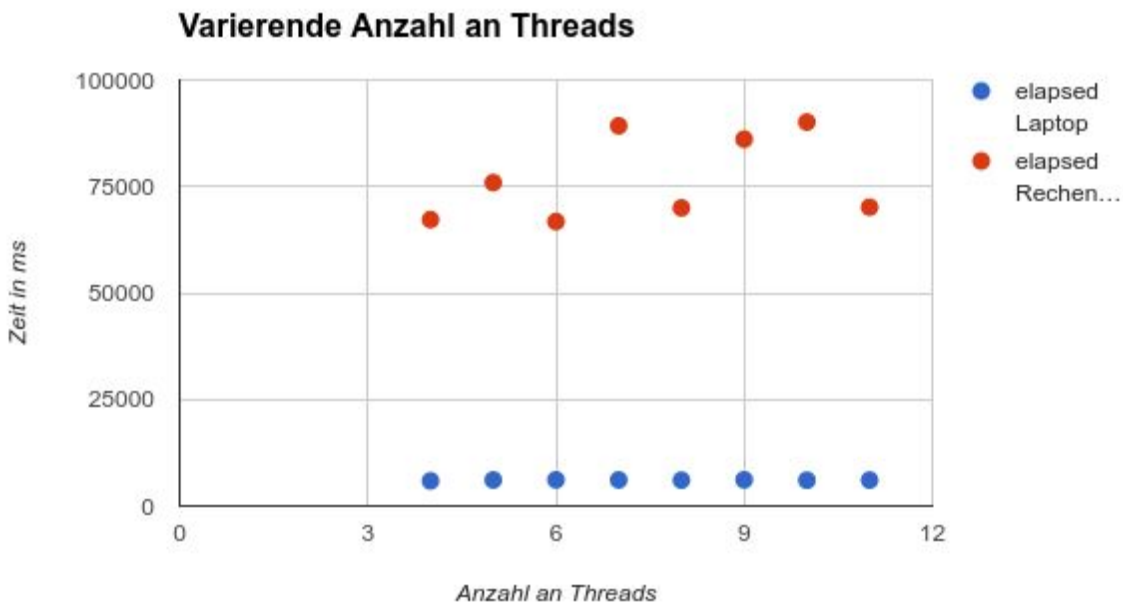
- hat weniger zeilen code
    - Java: 98 Zeilen
    - Clojure: 46 Zeilen
- hat weniger externe dependencies weil alle primitiven Operationen in der Core Library sind
- Ist schneller bei implementation mit gleichen gendanken
    - Suche von 1_000_000_000_000_000_000L bis 1_000_000_000_000_000_006L
    - 4 kern desktop cpu (kein hyperthreading)
    - Java: 2018 ms
    - Clojure: 1661 ms

5. Das Konto-Beispiel

Tabellen:

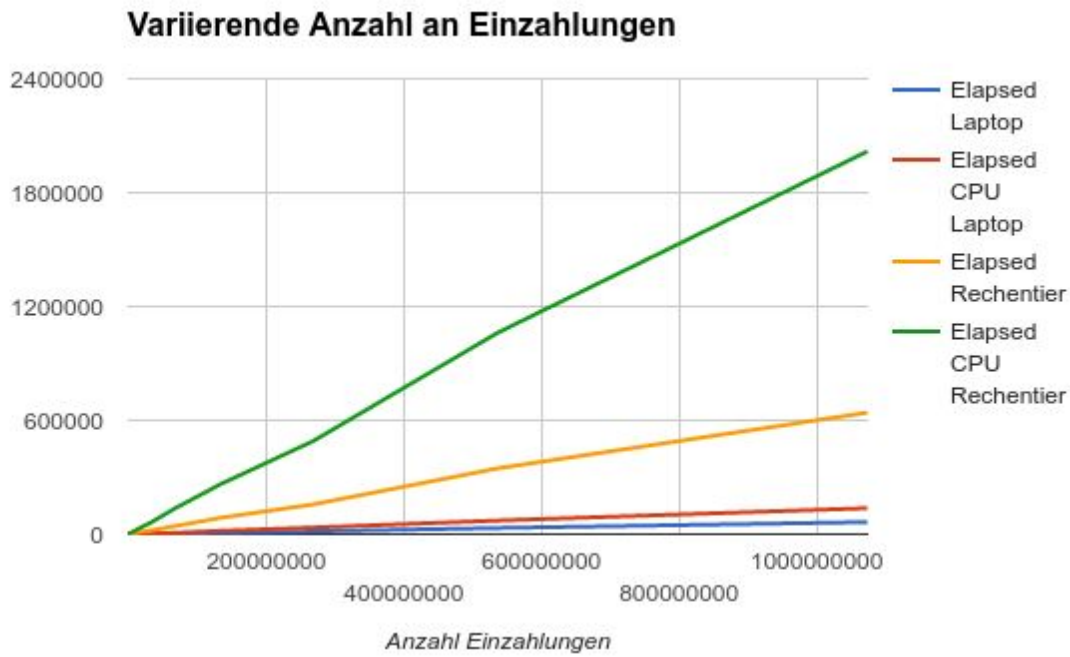
Anzahl an Threads variiert:

customers	elapsed Laptop	elapseld cpu Laptop	elapsed Rechentier	elapsed cpu rechentier
4	5898	12373	67227	216452
5	6112	14295	75918	250795
6	6133	13200	66769	216873
7	6117	13827	89239	308856
8	6059	13308	69978	224138
9	6145	13668	86105	294653
10	6043	13792	90117	312730
11	6073	13558	70141	225541



Anzahl an Einzahlungen variiert:

Anzahl Einzahlungen	Elapsed Laptop	Elapsed CPU Laptop	Elapsed Rechentier	Elapsed CPU Rechentier
32768	16	0	50	139
65536	15	45	94	264
131072	14	15	100	311
262144	13	0	179	561
524288	29	46	328	998
1048576	59	170	679	2203
2097152	115	232	1356	4451
4194304	217	437	2629	8436
8388608	408	717	5429	16654
16777216	918	1998	10973	33233
33554432	1807	3327	20364	63607
67108864	3890	7170	42413	135248
134217728	7549	17654	86362	264514
268435456	15118	34983	156854	490904
536870912	32565	71935	347092	1062420
1073741824	63737	137092	640603	2017437



Beobachtungen:

Zeit ist konstant bei änderung an der Zahl der Threads.
Zeit und CPU-Zeit steigt linear zur Anzahl der Einzahlung.

Lässt sich die Synchronisation im Hinblick auf diese einfache Leseoperation vielleicht relaxieren?
Integer schreiben ist atomar. Daher wird getBalance niemals einen inkorrekten Kontostand sehen. Es ist nur nicht sichergestellt ob der vorherige Einzahlungsvorgang schon abgeschlossen ist. Es müssen also nur Deposit und Withdraw synchronized sein. Die Sichtbarkeit kann mit Volatile erreicht werden.

6. Reduktion von Lock-Contention

Das lock auf dem Konto kann in Zwei kleinere Locks geteilt werden:

- ein Feld mit der Summe der Einzahlungen
- ein Feld mit der Summe der Auszahlungen
- zum auszahlen braucht man schreibzugriff auf Auszahlungen und Lesezugriff auf Einzahlungen
- zum einzahlen braucht man nur schreibzugriff auf Einzahlungen
- So kann gleichzeitig eine Einzahlung und eine Auszahlung gemacht werden

7. Konto-Beispiel noch einmal

Bei nur schreibendem Zugriff (Einzahlung) :

- Atomic ist Schneller als Monitor
- Lock facade ist auch schneller als Monitor
- Bei geringer Contention ist die Lock Facade langsamer als Atomic, bei Hoher Contention ist die echtzeit ähnliche aber Atomic hat höhere CPU zeit
- Monitor und LessMonitor sind gleich

Bei nur lesendem Zugriff:

- Lockfacade ist nur meistens besser als Monitor
- LessMonitor ist besser als Monitor weil er nur einen volatile read macht und kein lock hat
- Atomic grob genau so schnell wie LessMonitor

Lese- und Schreibzugriffe:

- Less Monitor doppelt so schnell wie Monitor
- Lockfacade ist, ab 4 Threads, schneller als Monitor und LessMonitor
- Bei hoher Contention ist Atomic besser als Lockfacade

8. Das Konto-Beispiel mit Akkumulator

Akkumulator kann zwar Addition atomar bietet aber keine möglichkeit größere Funktionen atomar aus zu führen. Deswegen ist ein Akkumulator alleine nicht ausreichen um das Konto mit der selben Semantik auszführen. Wenn zwei Threads gleichzeitig etwas abheben kann es sein das das Konto überzogen wird.
Wir haben einen LongAdder verwendet um die Summe der Einzahlungen zu verwalten und einen AtomicLong für die Summe der Auszahlungen.
Performance Vergleich:
Jeweils 10.000.000 Operationen.
Bei **nur lesendem Zugriff** sind Atomic, Adder und AtomicAdder sehr ähnlich.
Bei **nur einzahlung** sind Adder und AtomicAdder klar besser:

noCustomers	time Laptop	cputime Laptop	time rechentier	cputime Rechentier	impl
64	337	1252	2617	75339	ATOMIC
64	76	187	42	420	ADDER
64	84	179	41	585	ATOMIC_ADDER

Bei **lesen und einzahlen**: Bei geringer Contention ist Atomic selten einmal besser, aber bei hoher sind Adder und AtomicAdder besser.

noCustomers	time Laptop	cputime Laptop	time Rechentier	cputime Rechentier	impl
2	165	188	649	1140	ATOMIC
2	179	205	602	1046	ADDER
2	232	337	708	1234	ATOMIC_ADDER
64	209	735	1149	27856	ATOMIC
64	125	390	626	16473	ADDER
64	148	409	658	17131	ATOMIC_ADDER

Bei **einzahlen und auszahlen**: Die Implementation von Adder ist für diesen fall nicht correct weil der Kontostand negativ ist, aber zum vergleich sind die Zahlen trotzdem aufgelistet.
Hier ist der AtomicAdder etwas Besser als das AtomicAccount, weil AtomicAdder beim einzahlen die performance, vom AdderAccount übernimmt. Die Auszahlung greift auf AtomicLong zurück.

noCustomers	time Laptop	cputime Laptop	time Rechentier	cputime Rechentier	impl
4	580	1598	2852	9623	ATOMIC
4	167	558	1327	4030	ADDER
4	356	846	1839	5389	ATOMIC_ADDER
64	695	2654	13839	378164	ATOMIC
64	170	579	1043	27739	ADDER
64	513	1940	12706	367207	ATOMIC_ADDER

Diese Daten sind auch in Java/doc/one/aufgabe 7.pdf/csv verügbar.