

## 1 Verteiltes Sortieren

### 1.1 Umgebung

Um das verteilte Sortieren sinnvoll Testen zu können, ist es notwendig die Ausführung auf mehrere Rechner zu verteilen. Eine rein lokale Ausführung würde zum einen keinen realistischen Kommunikationsoverhead bedeuten, zum anderen wäre die Ausführung an die Leistungsfähigkeit der lokalen CPU gebunden.

Die Messreihen wurden daher auf virtuellen Maschinen in der Amazon Elastic Compute Cloud (Amazon EC2)<sup>1</sup> ausgeführt. Die Ausführung erfolgte dabei jeweils für Server und jeden Client getrennt auf einer eigenen „Micro“ Instanz. Zur Durchführung der Testreihen wurde mit Hilfe von Python, Fabric<sup>2</sup> und boto<sup>3</sup> automatisiert und ausgeführt.

### 1.2 Durchführung der Messung

Die Performancemessung wird ausschließlich auf dem Server durchgeführt. Der Server misst dabei, wie viele Millisekunden er für das Einlesen der zu sortierenden Daten, zum Verteilen der Daten an die Clients sowie zum Iterieren über die Sortierten Ergebnisse benötigt. Außerdem wird noch die gesamte Ausführungszeit gemessen. Da es sich bei der Umgebung, in der das Sortieren durchgeführt wird, um virtuelle Maschinen handelt, ist die Ausführungszeit natürlich abhängig von der Auslastung der physikalischen Hosts. Es ist jedoch davon auszugehen, dass aufgrund der Tatsache, dass Amazon die Instanzen über viele Rechner verteilt, ein brauchbarer Mittelwert entsteht.

### 1.3 Parameter der Messungen

Die Messreihen wurden jeweils mit 2, 4, 8 und 16 Sortierclients und einem Server durchgeführt. Dabei wurden jeweils die Blockgrößen 10, 100, 1000 und 10000 getestet. Für die lokale Sortierung gelten diese Parameter natürlich nicht, hierbei wurde nur die Sortierung mittels `collections.sort()` ausgeführt. Als Daten die sortiert werden sollen wurden Goethes Faust I und II ausgewählt, die dem Projekt Gutenberg als Textdatei entnommen wurden. Die Testdatei hat dabei eine Dateigröße von 556 Kilobyte und wird vom Server vor der Verteilung an jedem Whitespace aufgetrennt. Dabei entstehen 81333 einzelne Strings, die dann Sortiert wurden.

### 1.4 Auswertung

Zur weiteren Referenz werden zunächst einmal die Messerte für das lokale Sortieren gegeben. Die folgende Messerte sind dabei angegeben:

- Blockgröße: die verwendete Blockgröße, d.h. wie viele Blöcke beim Iterieren über die sortierten Daten auf einmal vom Client zum Server übertragen werden.
- # Clients: wie viele Clients am Sortieren beteiligt waren
- $t_{add}$ : Wie lange der Server für das Einlesen der Daten und das Einfügen in eine lokale, unsortierte Liste benötigt hat.

---

<sup>1</sup><http://aws.amazon.com/ec2/>

<sup>2</sup><http://fabfile.org>

<sup>3</sup><http://code.google.com/p/boto/>

- $t_{distribute}$ : Wie lange die der Server zum Aufteilen und Verteilen der unsortierten Daten an die Clients benötigt hat. Die Clients beginnen unmittelbar nach dem Übertragen der Daten mit der Sortierung.
- $t_{iter}$ : Wie lange der Server für der Iterieren über die sortierten Daten benötigt hat. Dies umfasst dabei auch die Kommunikationskosten - hier wirkt sich also die Blockgröße unmittelbar aus.
- $t_{total}$ : Die gesamte Ausführungszeit. Dies beinhaltet alle anderen gemessene Zeiten plus den benötigten Verwaltungsoverhead.

Zunächst werden die Messwerte für das lokale Sortieren gegeben. Beim lokalen Sortieren wurden lediglich die Werte für das Einfügen der Daten in einer lokale, unsortierte Liste ( $t_{add}$ ), das Iterieren über die sortierten Ergebnisse ( $t_{iter}$ ) und die Gesamtzeit gemessen, da die restlichen Messgrößen hierbei keine Rolle spielen. Auf die Ergebnisse wird an dieser Stelle nicht eingegangen, dies wird an späterer Stelle, im Zusammenhang mit den anderen Sortierverfahren erledigt.

Blockgröße	# Clients	$t_{add}$ (ms)	$t_{distribute}$ (ms)	$t_{iter}$ (ms)	$t_{total}$ (ms)
0	0	66	0	4	307

Tabelle 1: Ergebnisdaten: Lokales sortieren (mittels `Collections.sort(...)`)

#### 1.4.1 Verteiltes *merge sort*

Blockgröße	# Clients	$t_{add}$ (ms)	$t_{distribute}$ (ms)	$t_{iter}$ (ms)	$t_{total}$ (ms)
10	2	104	533	8960	10908
100	2	103	353	1489	2850
1000	2	103	347	721	2080
10000	2	102	338	559	1904
10	4	103	467	11469	15145
100	4	103	365	2050	3423
1000	4	101	394	957	2557
10000	4	103	337	756	2101
10	8	99	1230	19300	23840
100	8	99	523	2898	5126
1000	8	112	619	1854	3924
10000	8	108	552	964	2961
10	16	105	854	25589	33684
100	16	109	791	3982	11406
1000	16	98	1310	2274	10494
10000	16	108	838	1177	8578

Tabelle 2: Ergebnisdaten: Verteiltes *merge sort*

Wie zu erwarten besteht beim Merge-Sort ein direkter Zusammenhang zwischen Blockgröße und der Gesamt-Ausführungszeit: Je höher die Blockgröße, desto schneller die Gesamtzeit. Dies ist jeweils

auf den kleineren  $t_{iter}$  Wert zurückzuführen. Offenbar ist es Vorteilhaft, möglichst große Blöcke zu übertragen. Dies ist möglicherweise auf die TCP-basierte Kommunikation und den dabei verwendeten Slow-Start-Algorithmus zurückzuführen.

Außerdem ist auffällig, dass die Variante mit nur 2 Clients die kürzeste Gesamt-Ausführungszeit hat. Dies resultiert dabei offensichtlich auf dem niedrigen Wert für  $t_{distribute}$ . Offenbar ist hiermit ein nicht unerheblicher Verwaltungs-Overhead verbunden, der durch der ausgelagerte Sortieren nicht wieder reingeholt werden kann.

Ferner ist zu Bemerken, dass der Unterschied zwischen der Blockgröße von 1000 und 10000 nichts so groß ausfällt, wie zwischen 1000 und den kleinere Blockgrößen. Offenbar liegt der Wert für die optimale Blockgröße (für diese Menge an zu sortierenden Daten) irgendwo zwischen diesen beiden Werten.

#### 1.4.2 Verteiltes *distribution sort*

Blockgröße	# Clients	$t_{add}$ (ms)	$t_{distribute}$ (ms)	$t_{iter}$ (ms)	$t_{total}$ (ms)
10	2	102	501	4844	7792
100	2	103	573	1514	4506
1000	2	100	468	556	3438
10000	2	116	527	665	3614
10	4	99	443	6873	9822
100	4	109	516	1831	4662
1000	4	122	352	483	3278
10000	4	112	501	597	3415
10	8	99	591	12638	15737
100	8	117	451	1676	4658
1000	8	111	577	985	3980
10000	8	102	706	742	4190
10	16	107	611	15080	18305
100	16	110	867	2547	5938
1000	16	103	752	861	4314
10000	16	109	864	760	4351

Tabelle 3: Ergenisdaten : Verteiltes *distribution sort*

Auch beim Distribution-Sort besteht offenbar ein Zusammenhang zwischen Blockgröße und Ausführungszeit. Hierbei ist jedoch auffällig, dass die optimale Blockgröße (abermals bezogen auf diese Datenmenge) sehr dicht an einem Wert von 1000 liegt. Darüber steigt die Ausführungszeit wieder leicht an. **TODO Kann man da noch was rein interpretieren?**

Die optimale Client Anzahl liegt in diesem Fall offenbar bei 4 Clients, wobei die Werte für 2, 4 und 8 Clients insgesamt dichter beieinander liegen, als dies bei Mergesort der Fall war. Offenbar skaliert Distributionsort im Gegensatz zu Mergesort bei steigender Clientanzahl etwas besser. Dabei ist Distributionsort bei der hier vorliegenden Datenmenge im Optimalfall etwa um den Faktor 1,5 langsamer als Mergesort, was allerdings möglicherweise auch auf Implementierungsdetails zurück zu führen ist.

### 1.4.3 Vergleich der verteilten Verfahren mit lokalem Sortieren

Vergleicht man die beiden verteilten Sortierv Verfahren mit den Ergebnissen des lokalen Sortierens wird die völlige Unterlegenheit der verteilten Verfahren ersichtlich. Dieses Ergebnis ist auch nicht weiter verwunderlich. Zwar lässt sich lokales und auch verteiltes Sortieren im Wesentlichen mit Komplexität von  $O(n \log n)$  abschätzen, jedoch hat das verteilte Sortieren einen gravierenden Nachteil. Denn für ein verteiltes Sortieren ist es zunächst einmal erforderlich die Daten komplett an die Clients zu übertragen. Der damit verbundenen (lineare) Aufwand hat allerdings einen, um ein vielfaches höheren Vorfaktor als das lokal der Fall wäre. Sofern das Übertragungsmedium also nicht unglaublich schnell, und der lokale Speicherzugriff unglaublich langsam ist, ist es unwahrscheinlich, dass das verteilte Sortieren das lokale je schlagen kann. Zu Beachten ist jedoch, dass dies nur gilt, solange die Daten im Vorneherein übertragen werden müssen.