Dennis Krüger ()
Martin Thurau (612016)

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 und 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)[1] ausgeführt. Die Ausführung erfolgte dabei jeweils für Server und jeden Client getrennt auf einer eigenen „Micro"-Instanz. Die Durchführung der Testreihen wurde mit Hilfe von Python, Fabric[2] und boto[3] automatisiert und ausgeführt.

1.2. Durchführung der Messung

Die Performance-Messung 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, das 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 `Colletions.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 Messwerte für das lokale Sortieren gegeben. Die folgenden Messwerte sind dabei angegeben:

- Blockgröße: Die Anzahl der Blöcke, die beim Iterieren über die sortierten Daten auf einmal vom Client zum Server übertragen werden.
- # Clients: Die Anzahl der Clients, die am Sortieren beteiligt waren
- t_{add}: Die Zeit für das Einlesen der Daten und das Einfügen in eine lokale, unsortierte Liste benötigt hat.

[1] `http://aws.amazon.com/ec2/`
[2] `http://fabfile.org`
[3] `http://code.google.com/p/boto/`

- $t_{distribute}$: Die Zeit, 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}: die Zeit, die der Server für das Iterieren über die sortierten Daten benötigt hat. Die Iterationszeit umfasst dabei auch die Kommunikationskosten - hier wirkt sich also die Blockgröße unmittelbar aus.
- t_{total}: Die gesamte Ausführungszeit. Die gesamte Ausführungszeit 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 eine 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 geschieht an späterer Stelle im Zusammenhang mit den anderen Sortierverfahren.

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

Tabelle 1: Ergenisdaten: 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*

Dennis Krüger ()
Martin Thurau (612016)

Wie zu erwarten, besteht beim Mergesort ein direkter Zusammenhang zwischen Blockgröße und der Gesamt-Ausführungszeit: Je höher die Blockgröße ist, desto kleiner ist die Gesamtzeit (vgl. Abbildung 1). Dies ist jeweils auf den kleineren Zeitwert t_{iter} 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 zu bemerken, dass der Unterschied zwischen der Blockgröße von 1000 und 10000 nicht so groß ausfällt, wie zwischen 1000 und den kleineren Blockgrößen. Offenbar liegt der Wert für die optimale Blockgröße (für diese Menge an zu sortierenden Daten) irgendwo zwischen diesen beiden Werten.

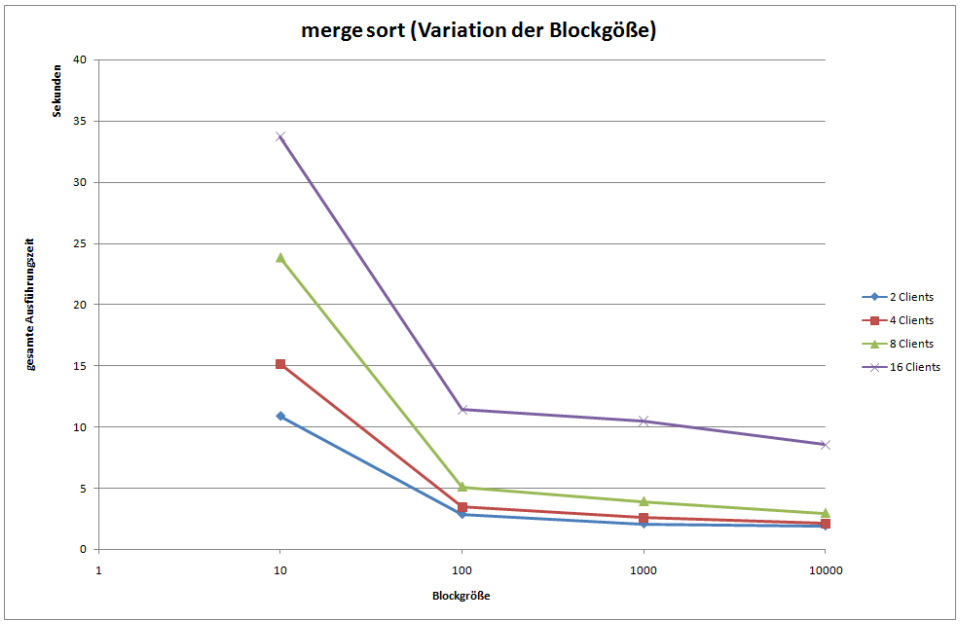

Abbildung 1: Ausführungszeit unter Variation der Blockgröße (merge sort)

Ferner 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 Verwaltungsoverhead verbunden, der durch das ausgelagerte Sortieren nicht wieder rein geholt werden kann (vgl. Abbildung 2).

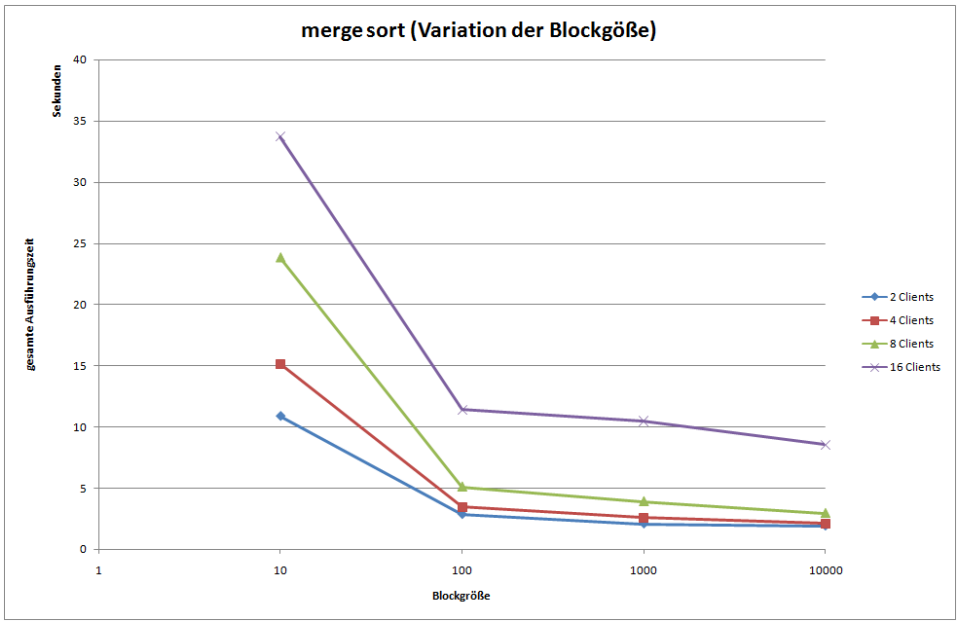

Abbildung 2: Ausführungszeit unter Variation der Client-Anzahl (merge sort)

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: Ergebnisdaten: Verteiltes *distribution sort*

Auch beim Distributionsort 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. Abbildung 3 zeigt diesen Umstand.

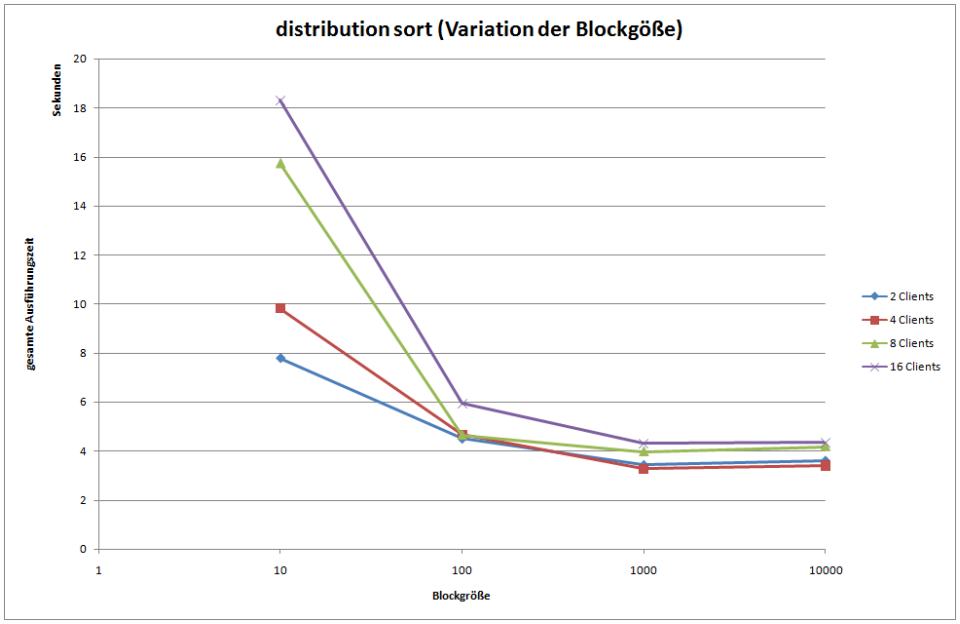

Abbildung 3: Ausführungszeit unter Variation der Blockgröße (distribution sort)

Die optimale Client-Anzahl liegt in diesem Fall offenbar bei 4 Clients, wobei die Werte für 2, 4 und 8 Clients insgesamt dichter bei einander liegen, als dies bei Mergesort der Fall war (vgl. Abbildung 4). Offenbar skaliert Distributionsort im Gegensatz zu Mergesort bei steigender Client-Anzahl 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.

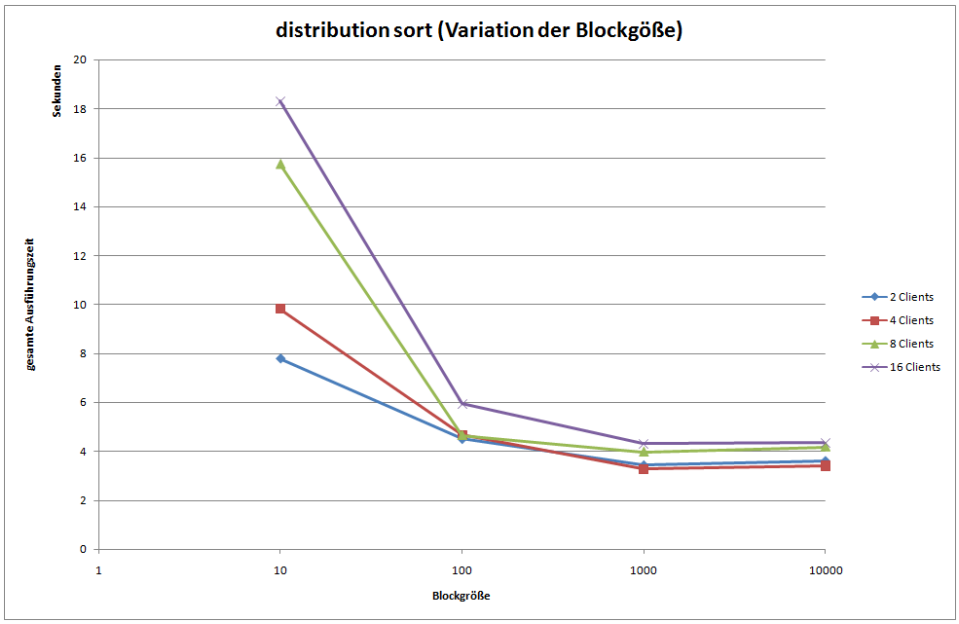

Abbildung 4: Ausführungszeit unter Variation der Client-Anzahl (distribution sort)

1.4.3. Vergleich der verteilten Verfahren mit lokalem Sortieren

Vergleicht man die beiden verteilten Sortierverfahren mit den Ergebnissen des lokalen Sortierens wird die völlige Unterlegenheit der verteilten Verfahren ersichtlich (cgl. Abbildung 5). Dieses Ergebnis ist auch nicht weiter verwunderlich. Zwar lässt sich lokales und auch verteiltes Sortieren im Wesentlichen mit der Komplexität von $O(n \log n)$ abschätzen, jedoch hat das verteilte Sortieren einen gravierenden Nachteil: Denn für das verteiltes Sortieren ist es zunächst einmal erforderlich die Daten komplett an die Clients zu übertragen. Der damit verbundene (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 wäre, ist es unwahrscheinlich, dass das verteilte Sortieren das lokale je schlagen könnte. Zu beachten ist jedoch, dass dies nur gilt, solange die Daten im vorne herein übertragen werden müssen.

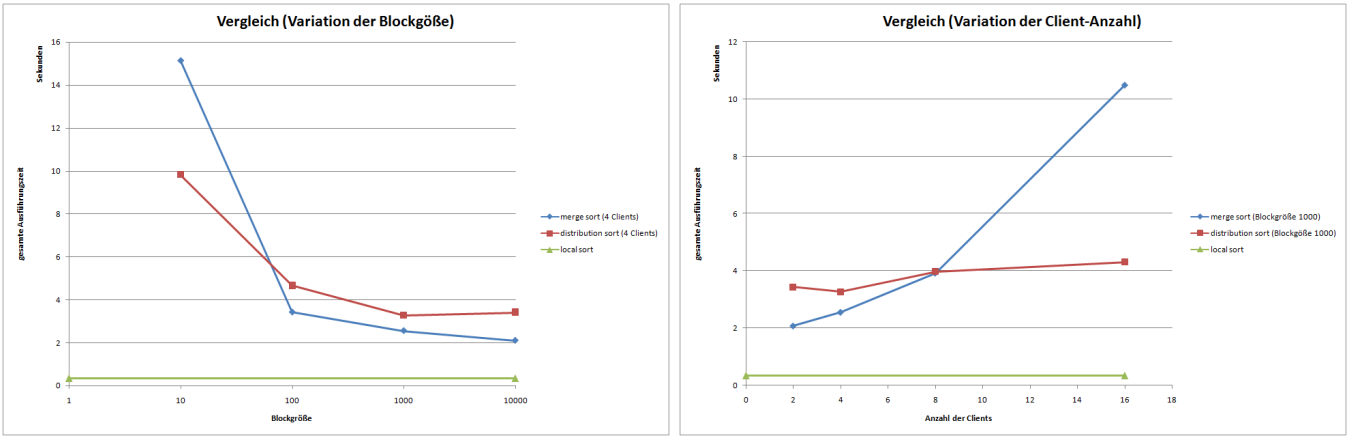

Abbildung 5: Vergleich des verteilten Sortierens mit lokalem

2. Verteilte Join Berechnung

2.1. Umgebung

Die Durchführung der Messungen erfolgte, wie in 1.1 beschrieben mittels einiger virtueller Instanzen auf Amazon Elastic Compute Cloud. Dabei wurden (je nach Join-Art) 2 bzw. 3 Instanzen verwendet.

2.2. Durchführung der Messung

Wie auch beim verteilten Sortieren wurde die Performancemessung ausschließlich auf dem Server durchgeführt (vgl. 1.2). Die folgenden Messgrößen wurden dabei erfasst:

- $t_{prepare}$: Die Zeit, die der Server zur Vorbereitung der Daten benötigt hat bevor sie versendet werden (z.B. Erstellung des Bitvektors).
- $t_{localjoin}$: Lokaler Joinaufwand (sei es zur Nachbearbeitung oder weil nur lokal gejoint wird). Sofern kein lokaler Join nötig war, ist dieser Wert 0.
- $t_{remotejoin}$: Verteilter Joinaufwand (inklusive der benötigten Kommunikationszeit vom Server zum Client und zurück).
- t_{total}: Gesamte Ausführungszeit (beinhaltet die anderen Zeiten).

2.3. Parameter der Messungen

Die Messreihen wurden jeweils mit 1 oder 2 Join-Clients und einem Server durchgeführt. Im Falle des Bitvektor-Joins wurden dabei Vektorgrößen von 10, 100 und 1000 getestet. Als Testdaten für das Joinen wurden dabei 3 unterschiedliche Datensätze verwendet, deren Einzelwerte jeweils aus einem zufällig generierten, numerischen Key und einem festen String bestehen.

Listing 1 zeigt einen Ausschnitt aus diesen Daten. Die Keys wurden randomisiert erzeugt. Allerdings wurde beim erzeugen der Testdaten darauf geachtet, dass es zwischen den einzelnen Datensätzen genügend Überschneidungen für einen sinnvollen Join gibt.

Listing 1: Auszug aus den Join-Testdaten

```
1773, DataA 0
1911, DataA 1
1551, DataA 2
34, DataA 6
[...]
```

2.4. Auswertung

Auffällig ist, dass der Fetch-as-needed Join der mit Abstand langsamste Join-Typ ist. Dies dürfte insbesondere mit der deutlich höheren Anzahl an RMI-Aufrufen zusammenhängen und insbesondere mit der damit verbundenen Latenz.

Beim Bitvektor-Join ist auffällig, das die Größe des Bitvektors offenbar kaum Einfluss auf die Performance dieses Join-Typs hat. Womit dies zusammen hängt ist unklar.

Join-Typ	Blockgröße	$t_{prepare}$	$t_{localjoin}$	$t_{remotejoin}$	t_{total}	# RMI calls
Lokal (nested loop)	-	809	809	809	809	0
Ship-Whole	-	0	757	0	943	2
Fetch-as-needed	-	0	2496	0	2496	4000
Semi-Join	-	0	557	176	735	1
Semi-Join (parallel)	-	0	981	226	1209	2
Semi-Join (sequential)	-	0	999	293	1294	2
Bitvektor	10	12	593	105	1075	1
Bitvektor	100	9	588	103	1058	1
Bitvektor	1000	14	587	99	1058	1

Tabelle 4: Ergebnisdaten: Verteiltes Joinen

Insgesamt betrachtet, liegen die einzelnen Join-Arten (bis auf den Fetch-as-needed Join) sehr dicht zusammen. Dies ist in sofern überraschend, da die Join-Arten sehr unterschiedliche Charakteristika aufweisen.

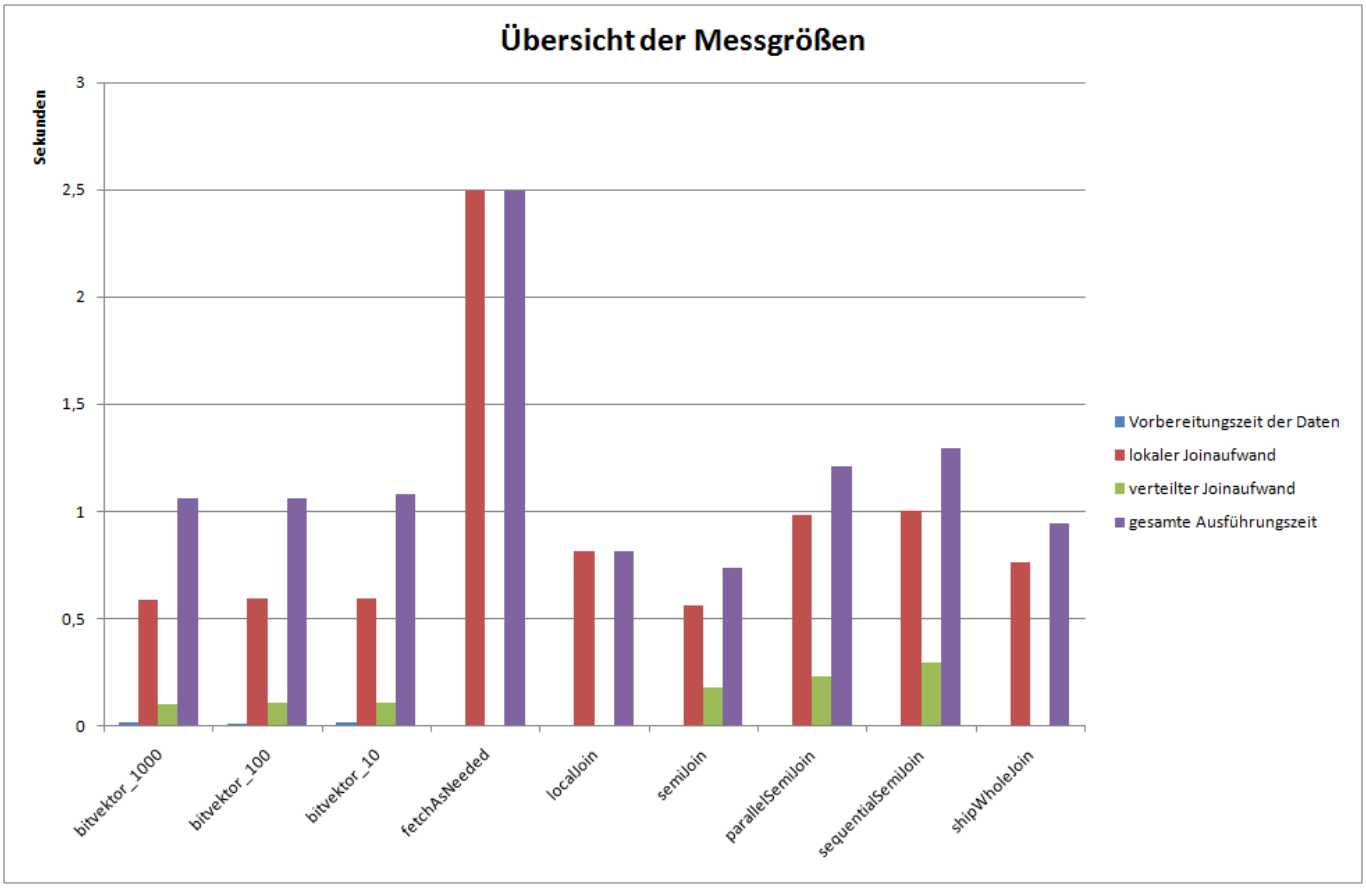

Abbildung 6: Ausführungszeit der einzelnen Join-Arten im Vergleich

A. Klassendiagramme

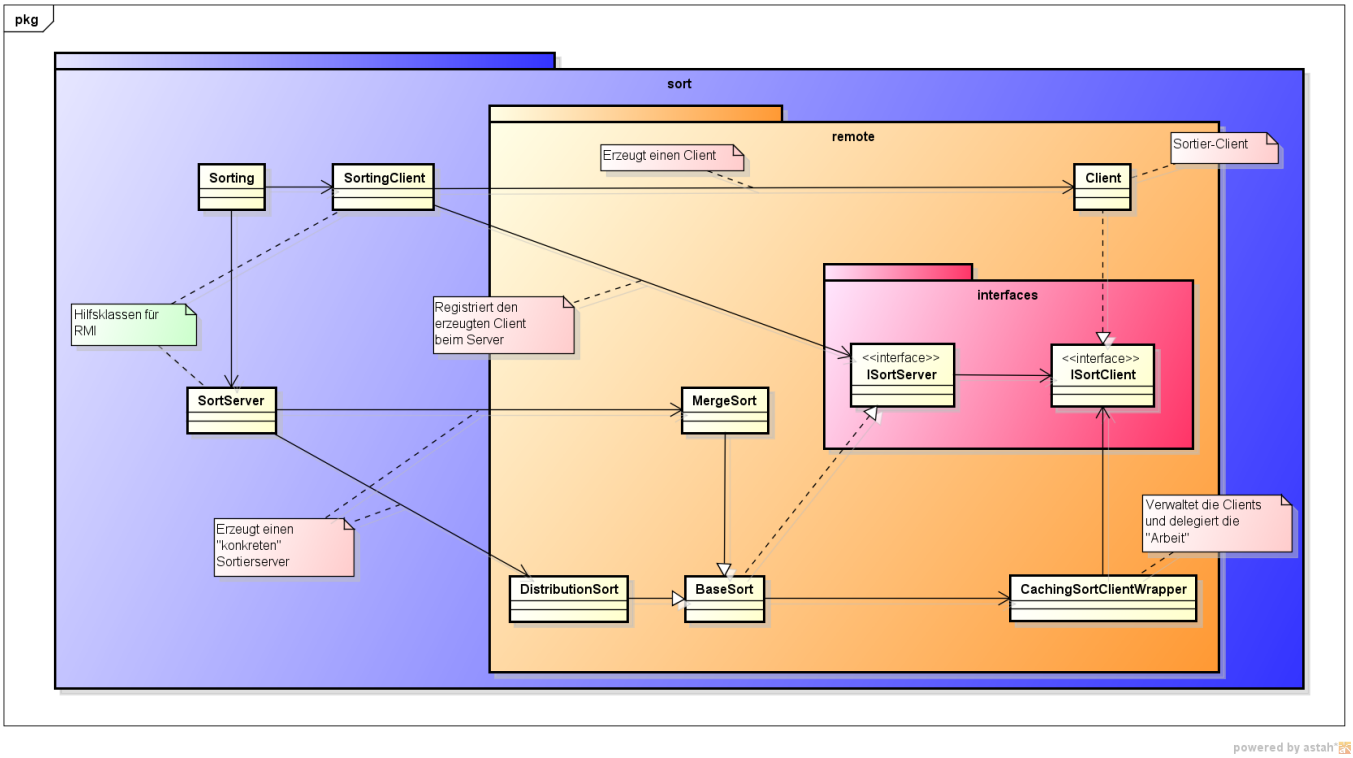

Abbildung 7: Klassenstruktur: Sortierclient und -server

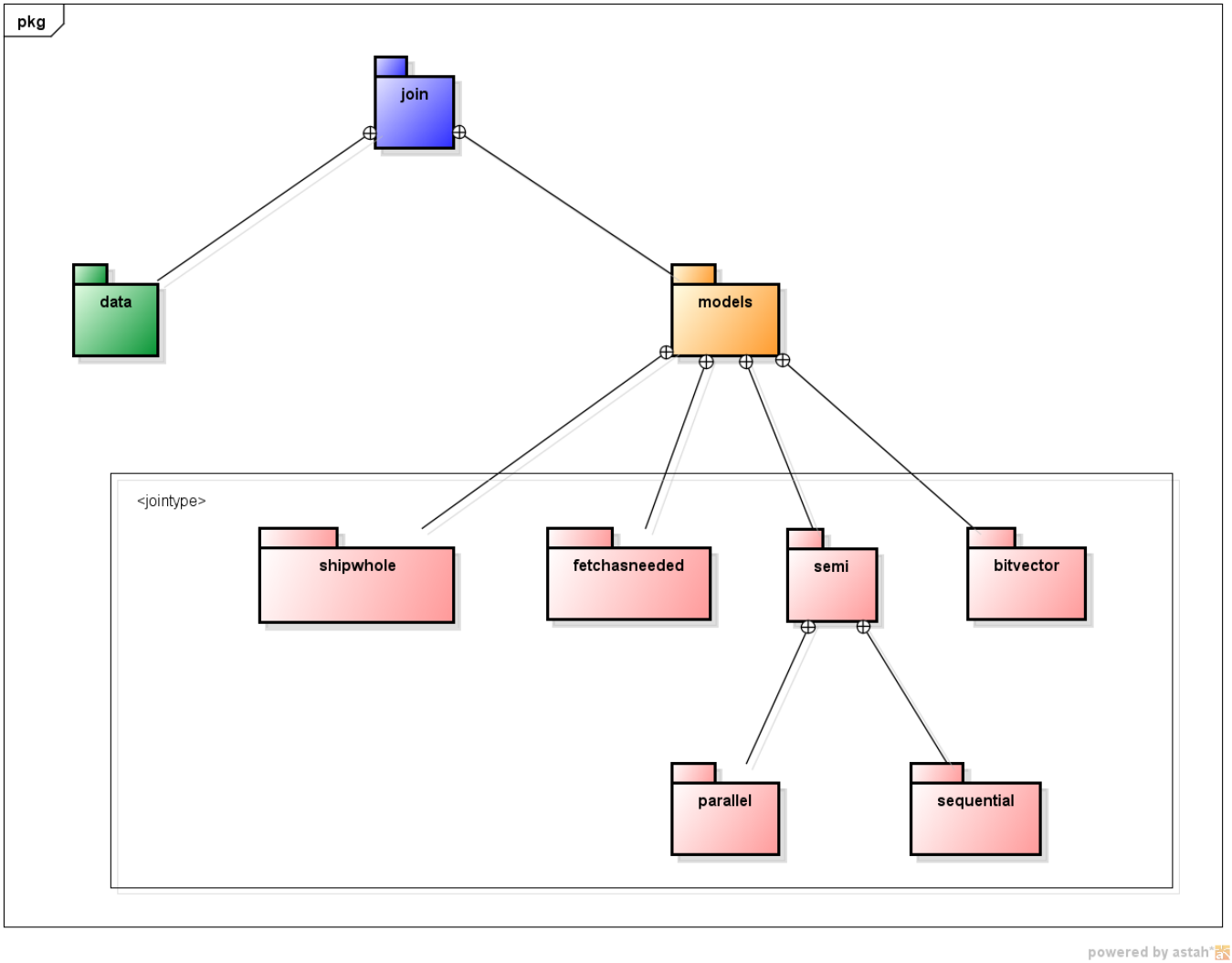

Abbildung 8: Paketstruktur: Joinclient und -server

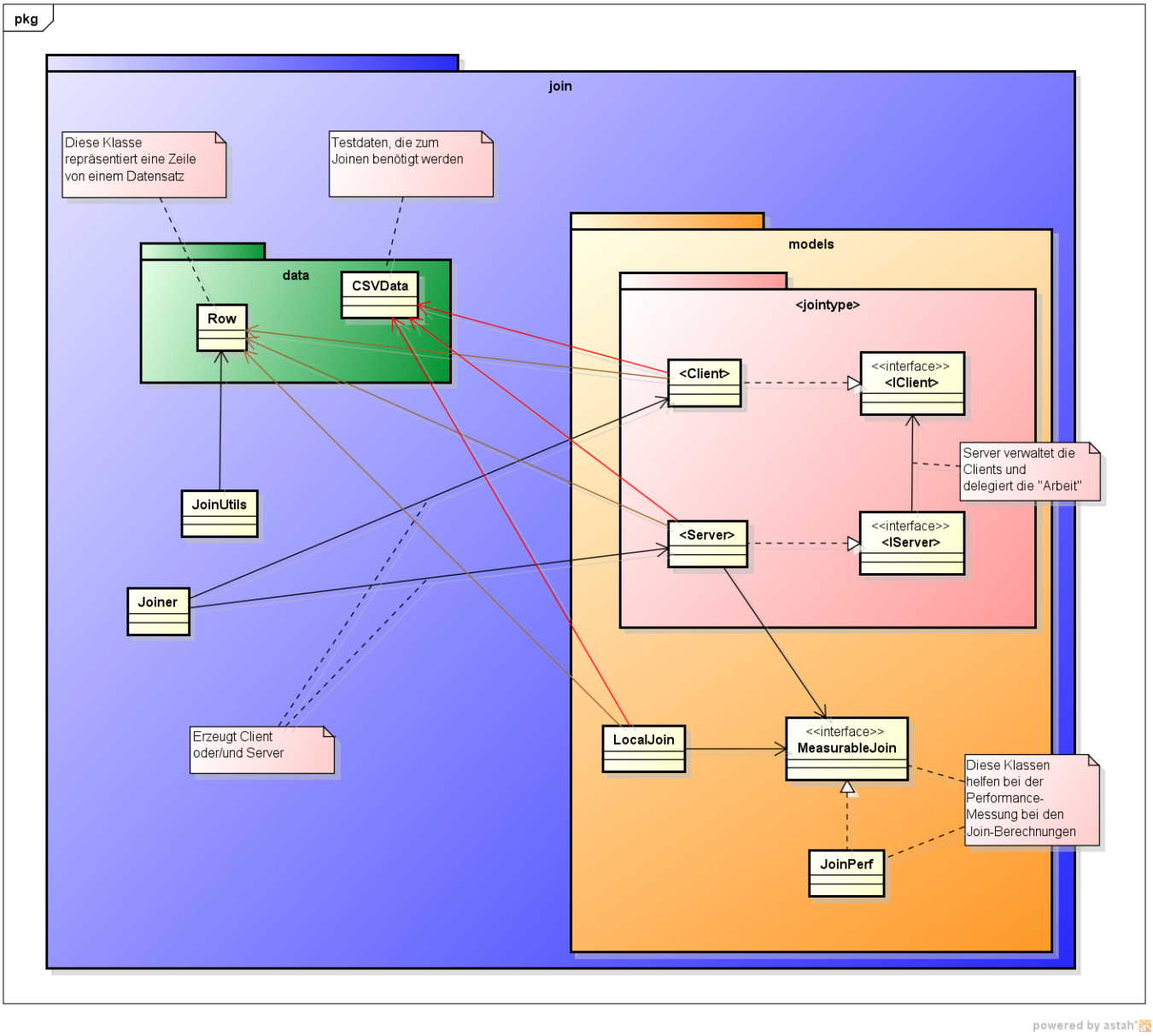

Abbildung 9: Klassenstruktur: Joinclient und -server