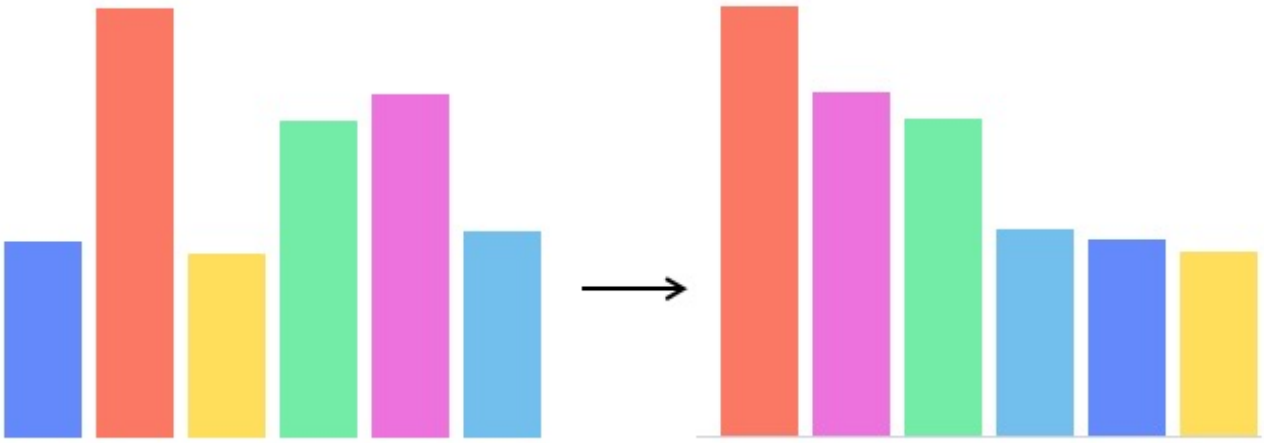
Projektbericht - Sortieralgorithmen

Fortgeschrittene Programmierung

Von Elias Pfeiffer, WDS123



Inhaltsverzeichnis

1. Einleitung

1.1 Vorwort

Der Code und die Vorarbeit des Projekts wurde auf einem Github Repository dokumentiert. Dieses kann unter folgendem Link erreicht und eigesehen werden:
https://github.com/eliaspfe/Projektbericht

1.2 Einleitung

Ein Computer besteht aus vielen verschiedenen Komponenten, die er benötigt, um richtig funktionieren zu können. Damit dieser in Anwendungen und Spielen eine gute Erfahrung bietet, müssen die leistungstragenden Komponenten, wie Grafikkarte und Prozessor genug Performance bereitstellen können. Ich besitze einen Computer, der schon etwas in die Jahre gekommen ist. Daher erreiche ich in Spielen und beim Arbeiten nicht mehr die Leistung, die ich eigentlich gerne haben möchte. Dies hat damit zu tun, dass Anwendungen immer mehr Leistung benötigen und so ältere Komponenten nicht mehr in der Lage sind, diese bereit zu stellen. Es wird also Zeit, einige dieser Komponenten auszutauschen. Doch damit stehen wir vor einem Problem. Das Angebot an Grafikkarten und Prozessoren wirkt auf den ersten Blick ziemlich überwältigend. Welche Kombination aus Grafikkarte und Prozessor ist nun am besten und bietet die beste Preis- Leistung?
Zu diesem Problem habe ich einen Datensatz erstellt. Für meine Projektarbeit werden später die Spalten „verwendete Grafikkarte“ und „Gesamtpreis“ relevant.

Meine Aufgabe ist es nun, die Daten zuerst nach Grafikkarte zu gruppieren und anschließend nach der Spalte „Gesamtpreis“ absteigend zu sortieren. Für den Sortieralgorithmus gelten folgende Bedingungen: Es sollen mindestens eine Milliarde aggregierte Zeilen sortiert werden. Außerdem soll er theoretisch in der Lage sein, die Sortierung parallel zu verarbeiten.
Um den richtigen Sortieralgorithmus für die gestellten Bedingungen zu wählen, werde ich die verschiedenen Algorithmen im Folgenden miteinander vergleichen.

2. Vorstellung Sortieralgorithmen

2.1 Selection Sort

Das Grundprinzip von Selection Sort besteht darin, das kleinste Element aus dem unsortierten Teil der Liste zu finden und dem sortierten Teil anzufügen. Dabei wird zu Beginn der Index auf Low gesetzt und der Algorithmus durchsucht die ganze Liste nach dem kleinsten Element. Ist das kleinste Element gefunden, wird dieses mit dem Wert am Index getauscht. Anschließend wird der Index inkrementiert und der Algorithmus sucht aufwärts vom Index erneut nach dem kleinsten Element. Dieser Vorgang wird solange wiederholt, bis die Liste sortiert ist. Die Zeitkomplexität von Selection Sort wird mit $O(N^2)$ angegeben, während die Speicherkomplexität mit O(1) beschrieben wird. Die Zeitkomplexität $O(N^2)$ bedeutet, dass bei einer schlechten CPU auch eine schlechte Performance erzielt wird, die Speicherkomplexität O(1) hingegen funktioniert gut mit wenig RAM. Wichtig zu erwähnen ist, dass der Selection Sort Algorithmus nicht stabil ist.
Außerdem ist er nicht in der Lage die Sortierung parallel zu verarbeiten. Daher und aufgrund der Zeitkomplexität $O(N^2)$ ist der Selection Sort Algorithmus nicht für unsere Problemstellung geeignet.

2.2 Bubble Sort

Der stabile Bubble Sort Algorithmus verschiebt das größte Element einer unsortierten Liste in ihren sortierten Teil. Er beginnt ebenfalls beim Low Index und durchläuft die Liste bis er am High Index angelangt ist. Ist das zu vergleichende Element größer als das Element mit dem nächst größeren Index, wird das zu vergleichende Element getauscht. Das Element mit dem größten Wert wandert also in der Liste nach hinten, dort befindet sich auch der sortierte Teil des Arrays. Der High Index wird, sobald das höchste Element des unsortierten Teils an der Stelle des High Index steht, dekrementiert. Dann wird dieser Vorgang so lange wiederholt, bis eine sortierte Liste vorliegt. Die Zeitkomplexität wird, gleich wie beim Selection Sort, durch $O(N^2)$ angegeben und die Speicherkomplexität von O(1) ist ebenfalls identisch. Da der Bubble Sort bei einer Milliarde Einträge durch die Zeitkomplexität $O(N^2)$ nur langsam performen würde und ebenfalls nicht parallel sortieren kann, ist er ebenfalls für unsere Problemstellung ungeeignet.

2.3 Insertion Sort

Beim Insertion Sort Algorithmus werden die Elemente einer unsortierten Liste nach und nach genommen und an der richtigen Stelle im sortierten Teil wieder eingefügt. Das erste Element ist beim Insertion Sort bereits sortiert, daher wird der „sortedHighIndex“ auf Low + 1 gesetzt. Danach wird der unsortierte Teil der Liste durchlaufen, bei dem das nächste Element des unsortierten Teils gewählt wird. Es findet also ein Vergleich mit jedem Element im sortierten Teil statt. Ist das Element am aktuellen Index kleiner, als das am vorherigen Index, werden die beiden Elemente getauscht. Zusätzlich wird der „sortedHighIndex“ dekrementiert. Dieser Vorgang wird solange wiederholt, bis das Element seinen Platz gefunden hat. Danach findet die Einordnung mit dem nächsten Element des unsortierten Teils statt. Die Zeitkomplexität kann bei einer bereits sortierten Liste O(N) betragen, im Average- und Worst-Case beträgt sie allerdings $O(N^2)$. Somit ist dieser Algorithmus bei großen Datensätzen ebenfalls sehr langsam. Die Speicherkomplexität beträgt O(1). Der Insertion Sort Algorithmus zählt zu den stabilen Sortierverfahren, ist allerdings aufgrund schlechter Performance bei großen Datenmengen und der fehlenden Fähigkeit parallel zu sortieren leider auch für unser Problem ungeeignet.

2.4 Quick Sort

Während die ersten drei Algorithmen einfach zu implementieren sind, wird der Quick Sort Algorithmus rekursiv implementiert. Er zählt zu den nicht stabilen Sortieralgorithmen. Das grobe Prinzip von Quick Sort besteht darin, ein beliebiges Element als Pivotelement zu definieren und anschließend die Zahlen, welche größer als das Pivotelement sind, rechts davon einzuordnen. Die kleineren Zahlen hingegen werden links vom Pivotelement eingeordnet. Dann wird zunächst der linke Teil der Liste sortiert und anschließend der rechte Teil. Dabei werden erneut Pivot Elemente für jeden Teil der Liste definiert und wieder werden die Zahlen, die größer als das Pivotelement sind auf der rechten Seite angeordnet. Dieser Vorgang wird solange wiederholt aufgerufen, bis alle Teile der Liste sortiert sind. Da der Quick Sort, wie bereits erwähnt, rekursiv implementiert wird, gibt es hier einen Base Case. Dieser tritt ein, wenn nur noch ein Element übrig ist.

Im Pre Recurse Teil wird die Liste in linke und rechte Teile unterteilt und im Recurse Teil werden diese erst links und dann rechts sortiert. Die Zeitkomplexität im Best- und Average- Case beträgt O(N log N), im Worst Case allerdings noch immer $O(N^2)$.
Das bedeutet, dass der Quick Sort schon einmal deutlich besser für größere Datenmengen geeignet ist und nur noch im Worst Case eine schlechte Performance liefert. Die Speicherkomplexität wird mit O(1) angegeben. Das bedeutet unser Quick Sort Algorithmus benötigt sehr wenig Arbeitsspeicher. Dennoch ist der Quick Sort Algorithmus nicht für unsere Problemstellung geeignet, da auch dieser die Sortierung nicht parallel bearbeiten kann.

2.5 Merge Sort

Ähnlich wie der Quick Sort wird der Merge Sort rekursiv implementiert. Beim Merge Sort Algorithmus wird eine Liste wiederholt in der Mitte aufgeteilt. Diese vielen kleinen Teil Listen werden dann von links nach rechts sortiert und anschließend wieder zusammen gefügt. Dies geschiet solange, bis die Liste ihre ursprüngliche Länge wieder erreicht hat.
Da der Merge Sort rekursiv implementiert ist, gibt es hier wieder einen Base Case. Dieser tritt ein, wenn nur ein Element übrig ist. Der Pre Recurse Teil beinhaltet die Aufteilung des Arrays in linke und rechte Teile. Beim Recurse werden dann diese Teillisten sortiert und im Post Recurse erneut zusammen gefügt. Hier sehen wir einen deutlichen Unterschied zu den bisher vorgestellten Algorithmen. Die Aufgeteilte Liste kann von mehreren Geräten parallel sortiert werden, d.h. mehrere Geräte können bei der Sortierung einer Liste gleichzeitig mitwirken. Der Merge Sort zählt zu den stabilen Sortieralgorithmen und besitzt für den Worst-, Average-, und Best-Case die Zeitkomplexität O(N log N) und die Speicherkomplexität O(N). Das bedeutet der Algorithmus ist auch bei großen Datenmengen zuverlässig schnell, bringt allerdings als Nachteil einen höheren Speicherbedarf mit sich. Das liegt daran, dass unsere Liste in viele kleinere Listen aufgespalten wird und anschließend wieder zusammen gefügt werden muss. Dennoch ist dieser Algorithmus der am Besten geeignete der hier beschriebenen. Er ist auch der einzige, der eine Lösung für unsere Problemstellung bietet. Ziel ist es, eine Milliarde Einträge zu sortieren, was ebenfalls parallel umgesetzt werden kann. Der Merge Sort erfüllt hierfür alle Bedingungen, weshalb dieser im nächsten Schritt implementiert wird.

3. Implementierung

Die gesamte Implementierung basiert auf einem vorgegebenen Klassendiagramm.

3.1 Row & AggregatedRow

Die Klassen Row und AggregatedRow sind sehr simpel und einfach implementiert.
Ich habe meinen Datensatz auf die Attribute „grafikkarte“ und „gesamtpreis“ reduziert.
Dabei stellt eine Row bzw. AggregatedRow eine PC-Konfiguration dar.
Beide Klassen enthalten zusätzlich noch einen Konstruktor.

```
public class Row {

    public Row(String grafikkarte, int gesamtPreis) {
        this.grafikkarte = grafikkarte;
        this.gesamtPreis = gesamtPreis;
    }

    public String grafikkarte;

    public int gesamtPreis;
}
```

```
public class AggregatedRow {

    public AggregatedRow(String grafikkarte, int gesamtPreis) {
        this.grafikkarte = grafikkarte;
        this.gesamtPreis = gesamtPreis;
    }


    public String grafikkarte;

    public int gesamtPreis;
}
```

3.2 Aggregator

Die Klasse Aggregator gruppiert die instanziierten Row Objekte nach Grafikkarte und findet für jede Grafikkarte den höchsten Gesamtpreis. Die ArrayList aggregatedRows beinhaltet also nur die Grafikkarte mit dem höchsten Gesamtpreis. Die Methode „findGreatestPriceForEveryGPU“ durchsucht dabei die Row Objekte nach einem größeren Gesamtpreis und achtet darauf, dass nur identische Grafikkarten Modelle miteinander verglichen werden. Im Folgenden Schritt wird der größte Gesamtpreis einer Grafikkarte und das passende Modell in AggregatedRows gespeichert. Durch wiederholtes Vergleichen mit der ArrayList „seenGraphicsCards“ wird verhindert, dass ein Grafikkarten Modell mehrmals der AggregatedRows Liste hinzugefügt wird.

```
public class Aggregator {

    public ArrayList<AggregatedRow> aggregate(ArrayList<Row> rows){
        ArrayList<AggregatedRow> aggregatedRows = new ArrayList<>();
        ArrayList<String> seenGraphicsCards = new ArrayList<>();

        for(Row row : rows){
            findGreatestPriceForEveryGPU(aggregatedRows, row);
            if(seenGraphicsCards.contains(row.grafikkarte) == false){
                aggregatedRows.add(new AggregatedRow(row.grafikkarte, row.gesamtPreis));
                seenGraphicsCards.add(row.grafikkarte);
            }
        }

        return aggregatedRows;
    }


    private void findGreatestPriceForEveryGPU(ArrayList<AggregatedRow> aggregatedRows, Row row){
        for(AggregatedRow aggregatedRow : aggregatedRows){
            if(aggregatedRow.grafikkarte.equals(row.grafikkarte)){
                if (row.gesamtPreis > aggregatedRow.gesamtPreis){
                    aggregatedRow.gesamtPreis = row.gesamtPreis;
                }
            }
        }
    }
}
```

3.3 Sorter

Die Sorter Klasse implementiert den Merge Sort Algorithmus. Dieser ist in mehrere Unterfunktionen unterteilt. Die Methode „mergeSort“ übernimmt die Unterteilung der Liste in kleinere Listen, welche parallel sortiert werden können. Ganz am Ende findet der Aufruf der „merge“ Methode statt. Diese Beginnt die Teil Listen wieder in der richtig geordneten Reihe zusammenzufügen. Die Methode „sort“ wird in der Main Methode verwendet um der „mergeSort“ Methode die zu sortierende Liste zu übergeben.

```
public class Sorter {

    public static ArrayList<AggregatedRow> mergeSort(ArrayList<AggregatedRow> rows) {
        int length = rows.size();
        if (length < 2) {
            return rows;
        }
        int middleIndex = length / 2;
        ArrayList<AggregatedRow> leftSide = new ArrayList<>(rows.subList(0, middleIndex));
        ArrayList<AggregatedRow> rightSide = new ArrayList<>(rows.subList(middleIndex, length));
        leftSide = mergeSort(leftSide);
        rightSide = mergeSort(rightSide);
        return merge(leftSide, rightSide);
    }

    public static ArrayList<AggregatedRow> merge(ArrayList<AggregatedRow> leftSide,
ArrayList<AggregatedRow> rightSide) {
        ArrayList<AggregatedRow> mergedArrayList = new ArrayList<>();
        int leftIndex = 0;
        int rightIndex = 0;
        while (leftIndex < leftSide.size() && rightIndex < rightSide.size()) {
            if (leftSide.get(leftIndex).gesamtPreis >= rightSide.get(rightIndex).gesamtPreis) {
                mergedArrayList.add(leftSide.get(leftIndex));
                leftIndex++;
            } else {
                mergedArrayList.add(rightSide.get(rightIndex));
                rightIndex++;
            }
        }
        while (leftIndex < leftSide.size()) {
            mergedArrayList.add(leftSide.get(leftIndex));
            leftIndex++;
        }
        while (rightIndex < rightSide.size()) {
            mergedArrayList.add(rightSide.get(rightIndex));
            rightIndex++;
        }
        return mergedArrayList;
    }

    public static ArrayList<AggregatedRow> sort(ArrayList<AggregatedRow> rows) {
        return mergeSort(rows);
    }
}
```

4. Fazit

Die Problemstellung bestand darin, eine Milliarde Zeilen zu aggregieren und diese anschließend performant zu sortieren. Der Algorithmus sollte die Möglichkeit bieten, diesen Vorgang parallel verarbeiten zu lassen. Verglichen wurden hierfür die Algorithmen Selection Sort, Bubble Sort, Insertion Sort, Quick Sort und Merge Sort.
Von den genannten Algorithmen war ausschließlich der Merge Sort für die Problemstellung geeignet. Dies liegt daran, dass der Merge Sort gut mit großen Datenmengen zurecht kommt, was bei den anderen oben genannten Algorithmen nicht immer der Fall ist. Zudem bietet der Merge Sort als Einziger die Möglichkeit, die Sortierung parallel zu verarbeiten.
Die Implementierung des Merge Sort Algorithmus findet in vier verschiedenen Klassen statt. Die Klassen „Row“ und „AggregatedRow“ stellen in meinem Fall eine PC Konfiguration dar, welche auf meinem selbst erstellten Datensatz basiert. Sie unterscheiden sich lediglich im Klassennamen. Dahingegen gruppiert die „Aggregator“ Klasse meinen Datensatz nach den verschiedenen Grafikkarten Modellen und filtert die Konfiguration mit dem höchsten Gesamtpreis heraus. Die Kombination mit dem höchsten Gesamtpreis eines Modells wird anschließend einer Liste von AggregatedRows hinzugefügt. In der vierten Klasse, der „Sorter“ Klasse, wird die Liste der AggregatedRows absteigend nach Gesamtpreis sortiert. In dieser Klasse findet sich die Logik eines Merge Sort Algorithmus.
Jeder Sortieralgorithmus bringt seine eigenen Vor- und Nachteile mit sich. Bei der Wahl eines Algorithmus kommt es aufgrund der unterschiedlichen Funktionsweise sehr auf den Anwendungsbereich an.
Das Ergebnis meiner Arbeit ist ein funktionierender Merge Sort Algorithmen der alle Anforderungen der Problemstellung erfüllt.

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
Alle sinngemäß und wörtlich übernommenen Textstellen aus fremden Quellen wurden kenntlich gemacht.

Hirrlingen, den 20.05.2024

Elias Pfeiffer