

Assignment 01: Distributed Sort

Hinweis: Alle Teilnehmer haben, soweit nicht anders gekennzeichnet, zu gleichen Teilen an diesem Dokument mitgewirkt.

Lukas Johannes Foerner
Hochschule München
Fakultät 07, IF
München, Deutschland
lukas.foerner@hm.edu

Robin Grellner
Hochschule München
Fakultät 07, IF
München, Deutschland
robin.grellner@hm.edu

Simon Symhoven
Hochschule München
Fakultät 07, IS
München, Deutschland
simon.symhoven@hm.edu

Abstract—Der verwendete Quellcode ist in einem GitHub Repository verwaltet [1].

I. RANDOM DATA

Mit Hilfe der Methode `createFiles()` werden Zufallszahlen erzeugt – beginnend bei 10.000 Stück. Diese werden auf dem Dateisystem in Form einer Textdatei gespeichert. Jede Runde erhöhen wir die Anzahl der Datenpunkte um 10.000, solange bis die Größe von 1.000.000 Zufallszahlen erreicht ist. Die Ergebnisse der Messungen für die später verwendeten Algorithmen werden in einer csv-Datei gespeichert. Diese zeichnet die Anzahl der Zufallszahlen, die Zeit für den Merge-Sort, sowie die Zeit für den parallelisierten Merge-Sort auf.

II. DER MERGE-SORT

Die Methode `processFiles()` startet den eigentlichen Merge-Sort. Die zuvor erzeugten Textdateien werden in einer Schleife eingelesen und jeweils die beiden Merge-Algorithmen angewendet, die Zeiten gestoppt und anschließend ein Durchschnitt über alle Runden berechnet. Die Methode `sort()` erwartet dabei eine Liste von Zahlen und eine Anzahl der zu sortierenden Zahlen n . Ist die $n < 2$, so ist der Basisfall erreicht. Andernfalls wird die Mitte kalkuliert und zwei neue Listen erzeugt, die jeweils die Elemente 0 bis $n/2$, bzw. $n/2$ bis n enthalten. Nun wird `sort()` rekursiv auf die beiden Listen aufgerufen:

```
public void sort(List<Integer> a, int n) {
    if (n < 2) {
        return;
    }
    int mid = n / 2;
    List<Integer> l =
        new ArrayList<Integer>();
    List<Integer> r =
        new ArrayList<Integer>();

    for (int i = 0; i < mid; i++) {
        l.add(i, a.get(i));
    }
    for (int i = mid; i < n; i++) {
```

```
        r.add(i - mid, a.get(i));
    }
    sort(l, mid);
    sort(r, n - mid);
    merge(a, l, r, mid, n - mid);
}
```

Die sortierten Teillisten werden dann wieder zusammengesetzt:

```
public void merge(List<Integer> a,
    List<Integer> l, List<Integer> r,
    int left, int right)
{
    int i = 0, j = 0, k = 0;
    while (i < left && j < right) {
        if (l.get(i) <= r.get(j)) {
            a.set(k++, l.get(i++));
        } else {
            a.set(k++, r.get(j++));
        }
    }
    while (i < left) {
        a.set(k++, l.get(i++));
    }
    while (j < right) {
        a.set(k++, r.get(j++));
    }
}
```

III. DER PARALLELISIERTE MERGE-SORT

Das Sortieren der Teillisten kann parallelisiert werden, in dem die rekursiven Aufrufe auf `sort()` in jeweils einem eigenen Thread ausgeführt werden. Sobald beide Threads beendet sind, wird `merge()` aufgerufen. Dazu wurde die Methode `sort()` angepasst:

```
public void sortParallel(List<Integer> a,
    int n)
{
    // Basisfall pruefen
    // Teillisten erzeugen
```

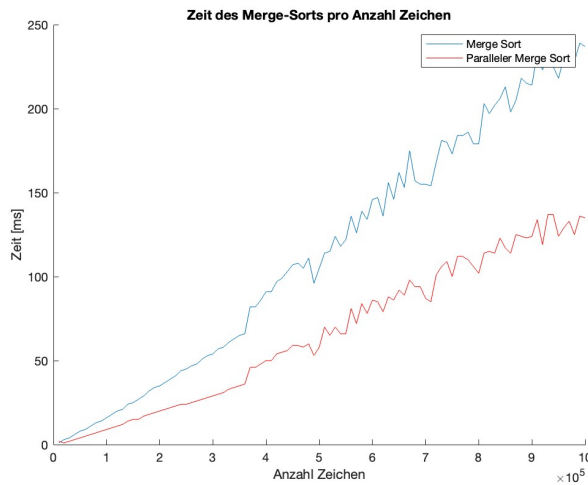
```

Thread t1 = new Thread(() -> {
    sort(l, mid);
});
Thread t2 = new Thread(() -> {
    sort(r, n - mid);
});
t1.start();
t2.start();
t1.join();
t2.join();

// Merge
}

```

IV. ANALYSE



Wie erwartet, hat sich die Ausführungszeit durch die Aufteilung in zwei Threads ungefähr halbiert. Im kleineren Bereich ($< 0.2 \times 10^5$) ist der Overhead der Threads allerdings größer als der Zeitvorteil durch die Parallelisierung.

V. ALGORITHMUS AUF VERTEILTEN SYSTEMEN

Können nicht alle Dateien bzw. deren Inhalte auf einem einzigen System gespeichert werden, müssen diese auf mehrere Knoten verteilt werden. Dazu muss die Liste der Zufallszahlen zunächst zu gleichen Teilen auf alle Knoten verteilt werden. Auf jedem dieser Knoten läuft dann der übliche Merge-Sort Algorithmus. Am Ende werden dann die vorsortierten Listen der jeweiligen Knoten im Zip-Lock-Verfahren (Reißverschluss) zusammengefügt. Dafür wird das jeweils erste Element der jeweiligen Liste mit den anderen verglichen. Das kleinste wird auf den Speicher des ersten Knoten geschrieben, sofern Platz vorhanden ist, ansonsten auf den nächsten Knoten. Dieses Verfahren wird für jeden Index der Listen wiederholt, bis alle Elemente der Ausgangsliste sortiert und auf mehreren Knoten verteilt ist.

REFERENCES

- [1] L. J. Foerner, R. Gellner, und S. Symhoven, "simonsymhoven/big-data-hm-group-j", GitHub, URL: <https://github.com/simonsymhoven/big-data-hm-group-j>, März 2022.