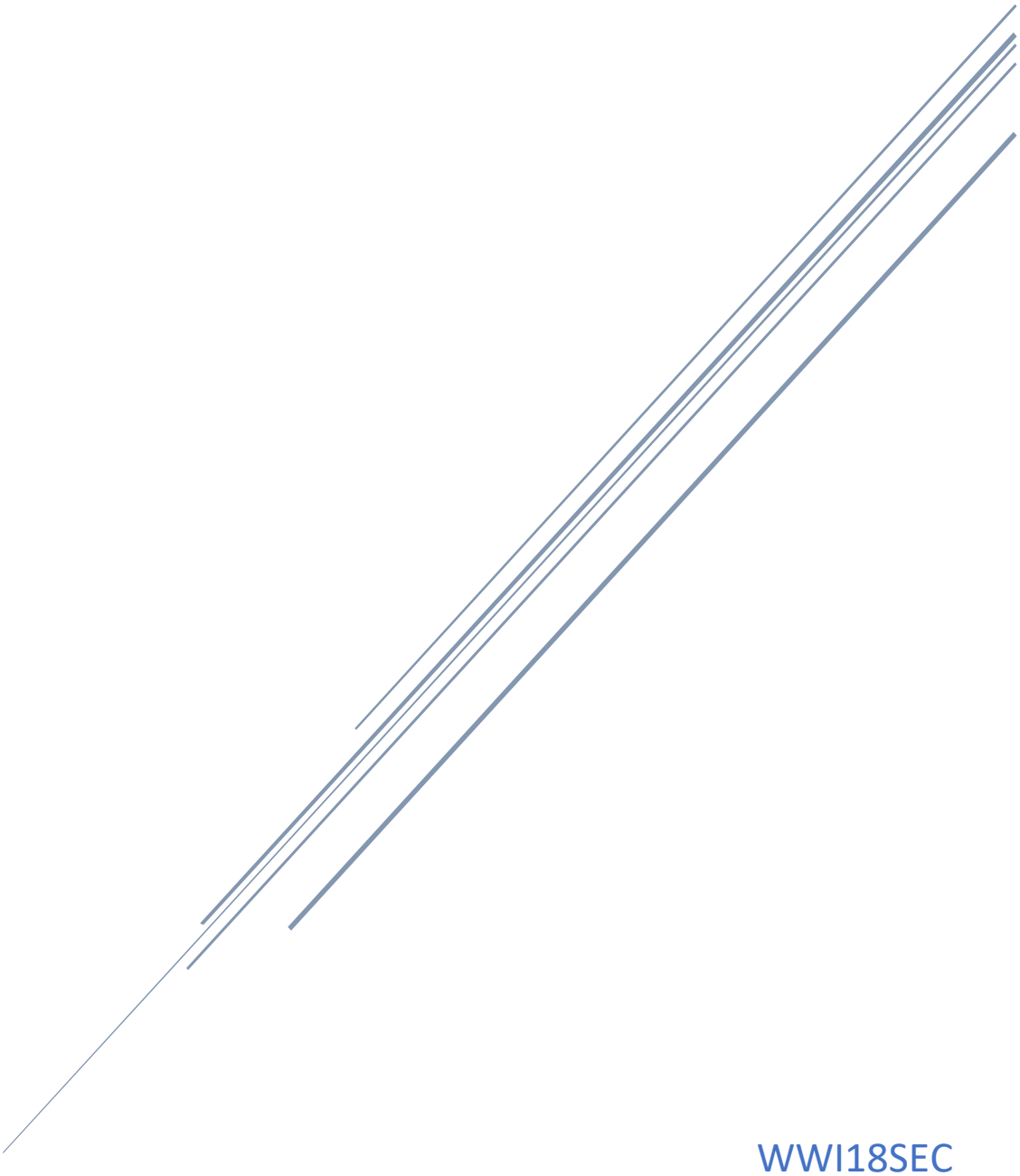
MERGESORT

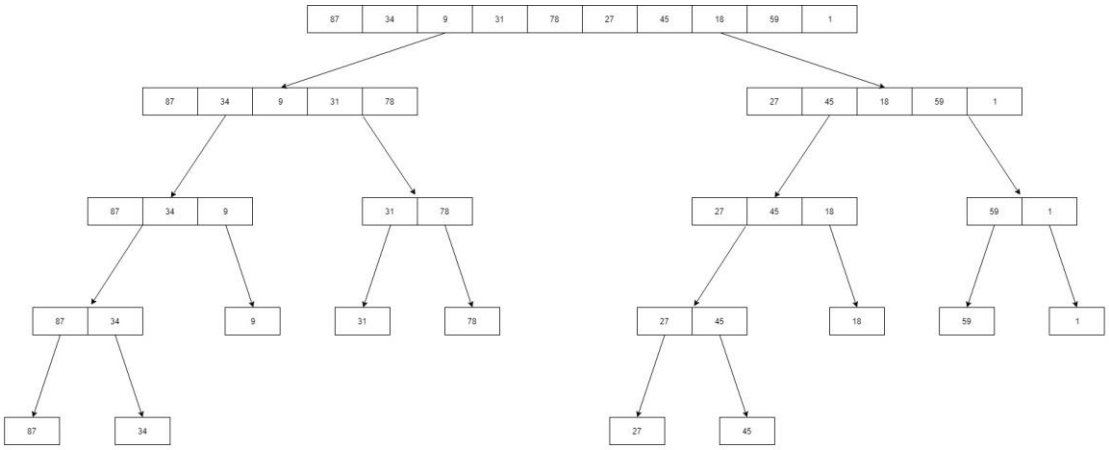
Dokumentation im Rahmen der Prüfungsleistung für
Moderne Programmierkonzepte

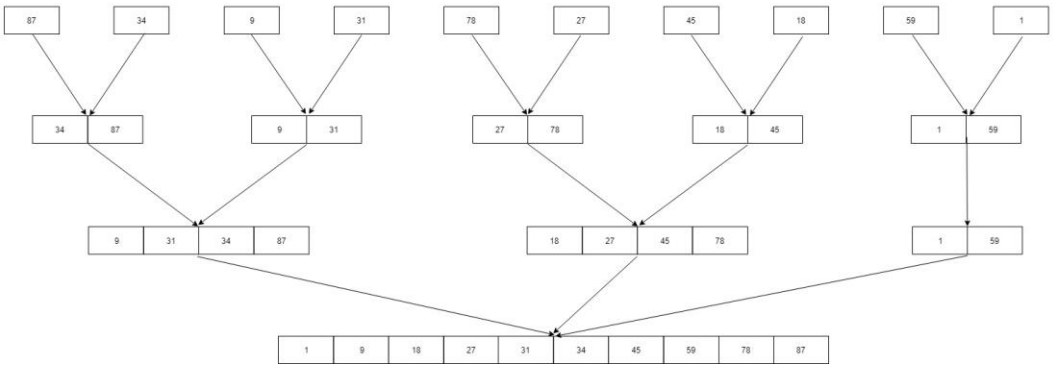


WWI18SEC
Florian Klinke, Jeannine Bertrand, Lilli Michaelis,
Ruben Ruhland, Tom Lindner

Was ist der Mergesort?

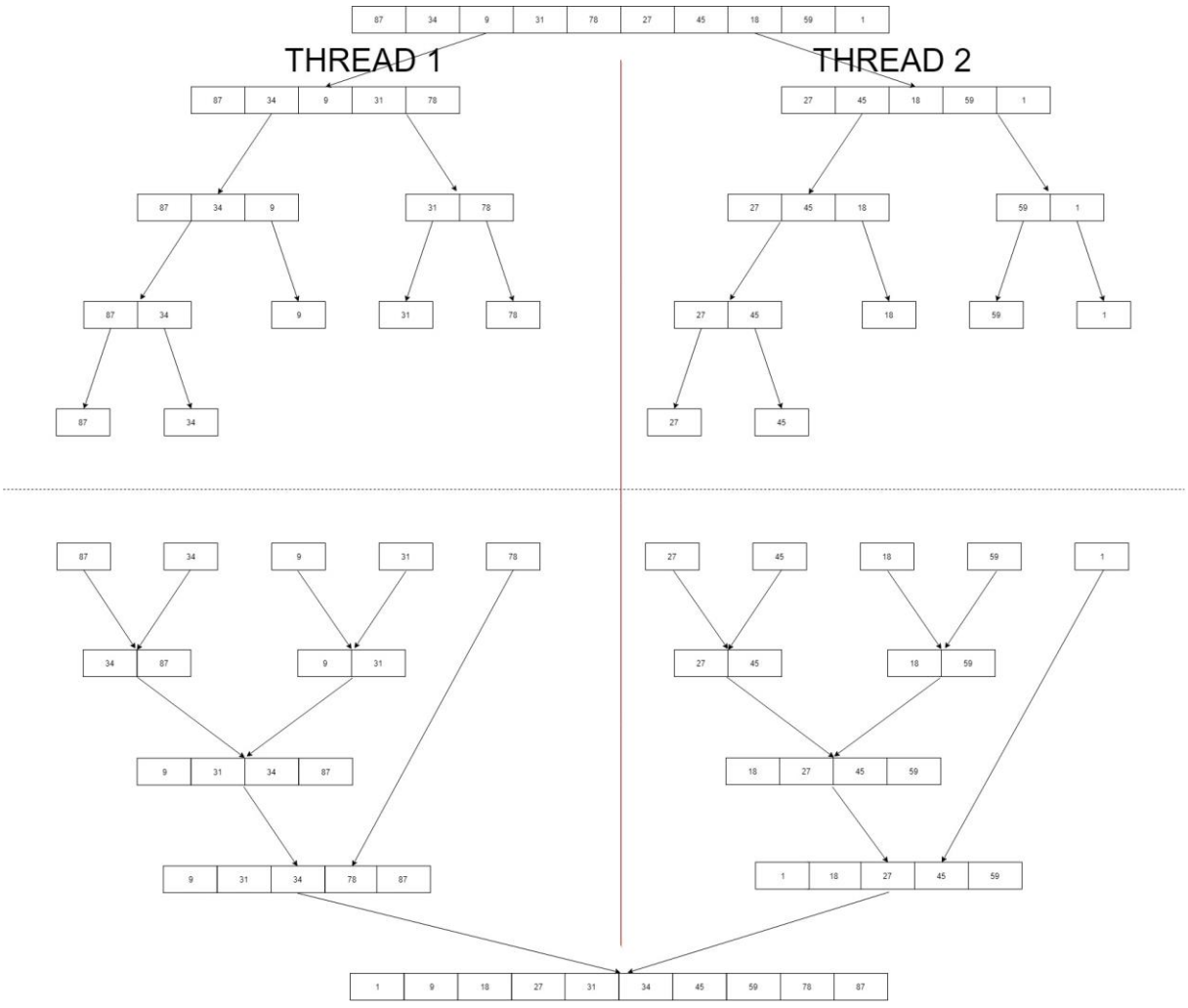
Der Mergesort ist ein bekannter Sortier-Algorithmus. Bei diesem Verfahren soll eine Kette von zufälligen Zahlen der Größe nach, mit Hilfe von Aufteilung, Vergleichen und wieder zusammensetzen, sortiert werden. Die Zahlenkette wird so lange geteilt, bis sie in Pakete, aus einzelnen Zahlen, aufgeteilt ist. Nun werden nach und nach immer zwei Pakete verglichen und sortiert zusammengefügt. Dies wiederholt sich so lange, bis alle Pakete wieder eine lange, sortierte Zahlenkette bilden.





Unser Ziel?

Unser Ziel war es, Teile der Programmierung funktional darzustellen und sowohl den Teilungs-, als auch den Sortiervorgang auf zwei parallellaufende Threads aufzuteilen, um eine bessere Performance zu erzielen.



Unser Vorgehen:

Zu allererst ließen wir uns ein Array mit Zahlen zwischen 1 und 100 mittels Math.random() in einer for-Schleife generieren und Variablen zuteilen, ohne sie selbst immer festlegen zu müssen, damit wir einen besseren Überblick über das Verfahren während der Implementierung haben.

➔ Wie legen wir Bezeichnungen für die einzelnen Elemente fest?

Die erste Idee eines dynamischen Variablennamens war für uns nicht umsetzbar. So legten wir eine Array-List an, die mit den einzelnen Arrays während des Teilungsvorgangs befüllt wurde.

Der zweite Schritt war nun das Zerteilen des originalen Arrays in zwei Elemente, welche jeweils eine Hälfte der Zahlenkette widerspiegeln sollten. Mit Hilfe einer if-Bedingung testeten wir, ob die Länge des originalen Arrays durch 2 teilbar ist, um eine richtige Aufteilung im Falle einer geraden oder ungeraden Anzahl an Elementen sicherstellen zu können.

Ein kurzer Zwischentest zeigte nun, dass dies bis hier hin auch unabhängig von der Länge der Zahlenkette funktioniert.

➔ Wie bekommen wir genug Arrays zum Speichern der geteilten Elemente? Stellenwertzuweisung?

Da wir die Arraylist nicht so neugestalten oder umsortieren konnten, wie wir es gerne hätten, entschieden wir uns dafür das ganze durch Rekursion zu ersetzen.

Um die Arrays richtig in zwei Teilarrays aufteilen zu können, definierten wir rekursiv *split()* mit einer Abbruchbedingung bei einer Arraylänge von 1. *Split()* wird so lange durchgeführt, bis die Arrays nur noch ein Element beinhalten und die Aufteilung somit abgeschlossen ist.

Da bis hierher alles funktionierte, begannen wir nun die Sortierung *merge()* zu implementieren. Wir ordneten die einzelnen Arrays ihrer Position „links“ und „rechts“ mit Hilfe von Vergleichen zu. Ist das zweite betrachtete Element größer als das erste wird es „links“ zugeordnet, wenn es kleiner ist „rechts“. Auf diese Weise werden nach und nach alle Elemente der Größe nach aufsteigend geordnet und miteinander verglichen. Das Endresultat war ein wieder erneut zusammengesetztes sortiertes Array.

Zur Anpassung des Programms an den Benutzer, nutzten wir ein Scanner-Objekt, welches erlaubt die Eingabeparameter für den Wertebereich und die Größe des Arrays variabel über eine manuelle Eingabe anzupassen.

Nun kam der Zeitpunkt, an dem wir das ganze parallelisieren wollten. Wir fertigten extra Klassen für die zwei zur parallelen Sortierung benötigten Threads an. Manuell splitteten wir die Ursprungsarrays, um die jeweiligen Hälften auf die Threads zu übergeben.

Bei der Ausführung trat das Problem auf, dass in unserer main-Klasse die Ergebnisse unsere Threads nicht mit unserer Erwartung gleichkam. Schließlich stellte sich heraus, dass die Threads die Arrays nicht rechtzeitig zurückgaben und somit die main-Klasse schon vor der Rückgabe durchgelaufen ist.

➔ Wie realisieren wir den Return der sortierten Arrays aus *run()* in die *main()* rechtzeitig?

Als Lösung hierfür stellte sich *join()* heraus. Nun übernahmen beide Threads jeweils die Hälfte der geteilten Arrays und fingen an diese weiter zu teilen, bis sie schließlich nur noch Arrays mit einzelnen Zahlen in ihrer Funktion beinhalten. Des Weiteren setzten die Threads die Arrays auch wieder zusammen, um als Ergebnis die sortierte Hälfte der Ursprungsliste zu speichern. Manuell wurde in der Main-Klasse durch *merge()* die beiden sortierten Listen in einem Array sortiert wieder zusammen gefasst.

Zum Vergleich von paralleler und einfacher Ausführung initialisierten wir eine Stoppuhr, die die Performance der unterschiedlichen Varianten einfacher vergleichen ließ. Es stellte sich wie erwartet heraus, dass die parallele Verarbeitung um einiges schneller arbeitet als die einfache Verarbeitung.

Nachdem wir nun das Programm zunächst „einfach“ programmiert haben, ging es darum Teile durch funktionale Programmierung zu ersetzen, um Performance- oder Implementierungsvorteile oder Nachteile zu entdecken.

Fazit:

Schon während der Implementierung der einfachen Programmierung stellten wir fest, dass wir mit dem Befehl „stream“ viel schneller die Array-Liste auslesen können.

Statt einer forEach-Schleife zum Befüllen des Arrays mit den generierten Zufallszahlen bat sich .map als eine kürzere, einfachere Alternative an. Dieser funktionale Teil machte es ebenfalls einfacher die Liste zu erweitern und bearbeiten.

Auch die Ausgabe der sortierten Listen der beiden Threads ließ sich durch funktionale Programmierung vereinfachen und abkürzen.

Die größte Herausforderung war die Aufteilung der Liste auf die zwei Threads funktional zu programmieren. Hier benötigten wir einige Anläufe, änderten zunächst jedoch die Art die Arrays zu teilen in *Arrays.copyofRange()*. Dies vereinfachte nicht nur den Code, sondern war der Anhaltspunkt für die Funktionale Programmierung. Somit lagerten wir Arrays.copyofRange() in einen eigenen Lampda -Ausdruck aus, um so das Teilen der Arrays durchzuführen.

Quellen:

http://www.oberstufeninformatik.de/info12/Stoppuhr.pdf (03.04.19)

https://www.youtube.com/watch?v=KF2j-9iSf4Q (03.04.19)

https://stackoverflow.com/questions/9148899/returning-value-from-thread (04.04.19)

https://www.youtube.com/watch?v=2XcfA6WKLPw (24.05.19)

Subramaniam, Venkat (2014): Functional programming in Java. Harnessing the power of Java 8 Lambda expressions. Frisco, TX: The Pragmatic Programmers (Pragmatic programmers). Online verfügbar unter http://proquest.tech.safaribooksonline.de/9781941222690.