Untersuchung der Skalierbarkeit von parallelem Sortieren auf einem Multicore-Prozessor

Bachelorarbeit

Studiengang: Informatik

Bearbeiter: Leon Zoerner

21. Dezember 2025

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Ziel dieser Arbeit ist es, die Grenzen von Threads und Parallelisierung aufzuzeigen. Dabei soll insbesondere untersucht werden, wie groß der Overhead durch Threads ist und welchen Performanceunterschied es macht, bereits initialisierte Workerthreads zu verwenden, im Vergleich zur Erstellung neuer Threads. Da sich für diese Untersuchungen ein geeigneter, leicht verständlicher und programmierbarer Anwendungsfall anbietet, habe ich mich für Sortieralgorithmen entschieden, die sich zudem sehr gut parallelisieren lassen.

1.2 Zielsetzung und Forschungsfrage

Ziel dieser Bachelorarbeit ist die systematische Analyse der Laufzeitentwicklung paralleler Sortierverfahren. Dabei soll untersucht werden, wie sich parallele Implementierungen von Quicksort und Mergesort im Vergleich zu ihren sequentiellen Varianten verhalten. Im Fokus stehen insbesondere folgende Punkte:

- der Einfluss verschiedener Threadingstrategien auf die Laufzeit,
- die Frage, ab welcher Eingangsgröße und bei welcher Anzahl von Threads ein messbarer Geschwindigkeitsvorteil entsteht,
- sowie die Identifikation von Thread-Management-Techniken, die für Sortieralgorithmen die besten Laufzeiten erzielen.

Aus diesen Aspekten ergibt sich die zentrale Forschungsfrage dieser Arbeit:
Unter welchen Bedingungen liefern parallele Sortieralgorithmen anhand von Quicksort und Mergesort einen signifikanten Laufzeitvorteil gegenüber der sequentiellen Ausführung, und welche Threadingstrategien führen dabei zur besten Laufzeit?

2 Theoretische Grundlagen

2.1 Sortieralgorithmen: Quicksort und Mergesort

Sowohl **Quicksort** als auch **Mergesort** basieren auf dem *Teile-und-Herrsche*-Prinzip und sind rekursive Sortieralgorithmen. Dabei wird das zu sortierende Array wiederholt in kleinere Teilprobleme zerlegt, die unabhängig voneinander verarbeitet werden.

Mergesort

Das Grundprinzip von **Mergesort** besteht darin, aus zwei bereits sortierten Teillisten eine neue sortierte Liste zu erzeugen. In dieser Arbeit wird die unsortierte Ausgangsliste rekursiv in zwei möglichst gleich große Hälften geteilt, bis jede Teilliste nur noch aus einem einzelnen Element besteht. Da eine Liste mit einem Element per Definition sortiert ist, beginnt anschließend der sogenannte *Merge-Schritt*. In diesem Schritt werden jeweils zwei sortierte Teillisten zu einer neuen sortierten Liste zusammengeführt.
Hierfür werden beide Teillisten sequenziell durchlaufen und die Elemente verglichen, wodurch pro Merge-Schritt n Vergleiche sowie $2n$ Lese- und Schreibzugriffe erforderlich sind. Die Laufzeit von Mergesort lässt sich durch die Rekurrenzgleichung

$$T(n) = 2 \cdot T(n/2) + n$$

beschreiben, wobei der Term $+n$ den Aufwand des Merge-Schritts repräsentiert. Daraus ergibt sich eine Gesamtlaufzeit von

$$T(n) = n \cdot \log_2(n)$$

bzw. in asymptotischer Notation $O(n \log n)$.

Quicksort

Quicksort ist im Grundaufbau ähnlich strukturiert, unterscheidet sich jedoch wesentlich im Ablauf. Die Liste wird nicht zwingend in zwei gleich große Hälften geteilt. Stattdessen wird zunächst ein sogenanntes *Pivot-Element* gewählt, anhand dessen die Liste in einen kleineren und einen größeren Teil partitioniert wird. Dieser Partitionierungsschritt erfolgt *vor* den rekursiven Selbstaufrufen, weshalb sich Quicksort auch als iterative Variante formulieren lässt. Beim Partitionieren wird die Liste so umsortiert, dass alle Elemente, die kleiner als das Pivotelement sind, links davon stehen und alle Elemente, die größer sind, rechts davon stehen. Dabei werden die Elemente auf beiden Seiten entsprechend getauscht.

Die Laufzeit von Quicksort hängt stark von der Qualität der Partitionierung ab. Im **Worst-Case**, beispielsweise bei ungünstiger Pivot-Wahl, beträgt die serielle Laufzeit

$$T(n) = \frac{1}{2} \cdot (n^2 + n),$$

$$O(T(n)) = n^2.$$

Im **Best-Case** sowie im **Average-Case** ergibt sich hingegen ebenfalls die Rekurrenzgleichung

$$T(n) = 2 \cdot T(n/2) + n,$$

woraus wiederum eine Laufzeit von $O(n \log n)$ folgt. Damit ist Quicksort im Durchschnitt asymptotisch genauso effizient wie Mergesort, aber in der Praxis ist Quicksort oft doppelt so schnell wie Mergesort, doch dazu später mehr.

2.2 Grundlagen der Parallelisierung

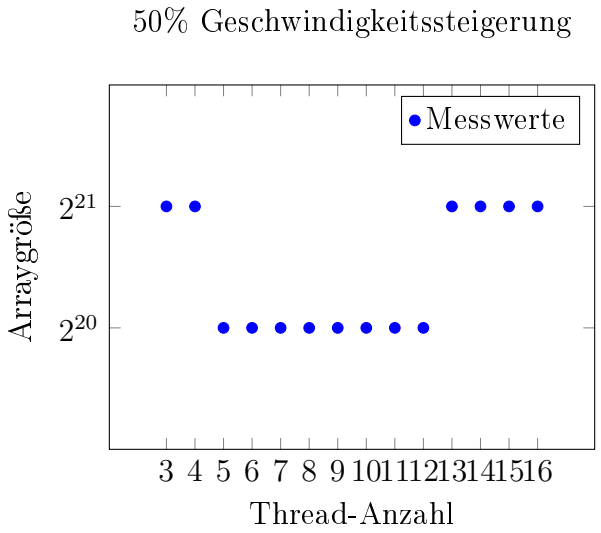
Parallelisierung beschreibt die gleichzeitige Ausführung mehrerer Programmteile mit dem Ziel, die Gesamtlaufzeit einer Berechnung zu reduzieren. Dabei wird eine ursprünglich serielle Aufgabe in mehrere Teilaufgaben zerlegt, die parallel auf mehreren Recheneinheiten verarbeitet werden können.
Der maximal erreichbare Geschwindigkeitsgewinn durch Parallelisierung ist jedoch begrenzt. Nach dem Amdahlschen Gesetz hängt die theoretische Beschleunigung davon ab, welcher Anteil eines Programms parallelisiert werden kann. Serielle Programmanteile sowie zusätzlicher Verwaltungsaufwand, beispielsweise durch Thread-Erzeugung, Synchronisation und Kommunikation, begrenzen die Skalierbarkeit.
In der Praxis führt Parallelisierung daher nicht zwangsläufig zu einer linearen Beschleunigung, insbesondere bei steigender Anzahl von Threads.
Einfach ausgedrückt bedeutet dies, dass Code mit seriellen Abhängigkeiten auch bei mehreren Threads nicht schneller ausgeführt wird. Daher sollte nur der Teil des Codes parallelisiert werden, der keine solchen Abhängigkeiten enthält.

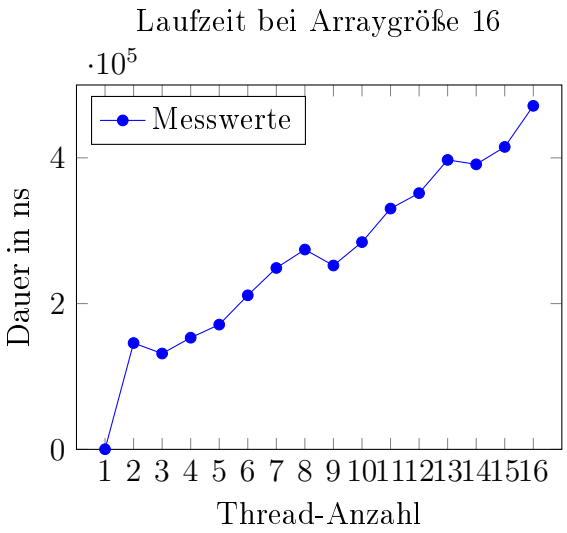
2.3 Thread-Modelle, Overheads und Skalierungsgrenzen

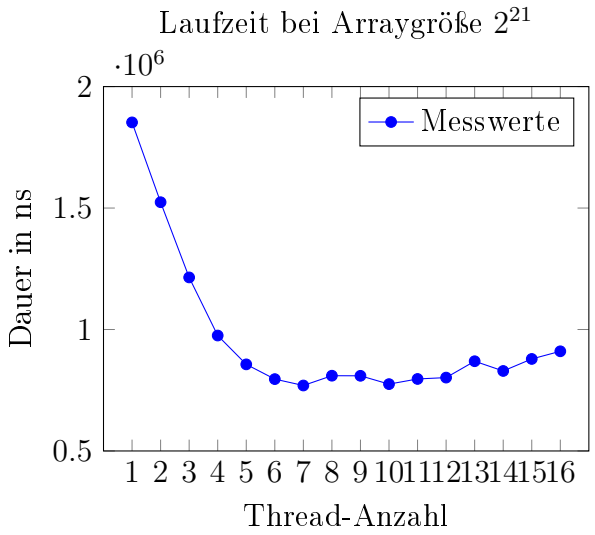
In der Praxis verursachen Threads verschiedene Overheads, die verhindern, dass eine theoretisch ideale Zeitersparnis (linearer Speedup) erreicht wird. Zu diesen Overheads zählen primär die Initialisierungs- und Join-Zeiten, die Laufzeiten von Destruktoren sowie die notwendige Synchronisation bei Abhängigkeiten zwischen Threads. Um die Datenkonsistenz zu gewährleisten, müssen Mechanismen wie Sperren (Mutexe) oder Barrieren (Synchronisationspunkte) eingesetzt werden, welche zusätzliche Wartezeiten und Verwaltungsoverheads verursachen. Parallel dazu setzen Hardware-Limitierungen der Skalierung Grenzen. Hierbei beeinflussen Context-Switching-Zeiten bei einer Überbelegung der Kerne (Oversubscription), die begrenzte Anzahl physischer Kerne (im Testsystem 8 physische

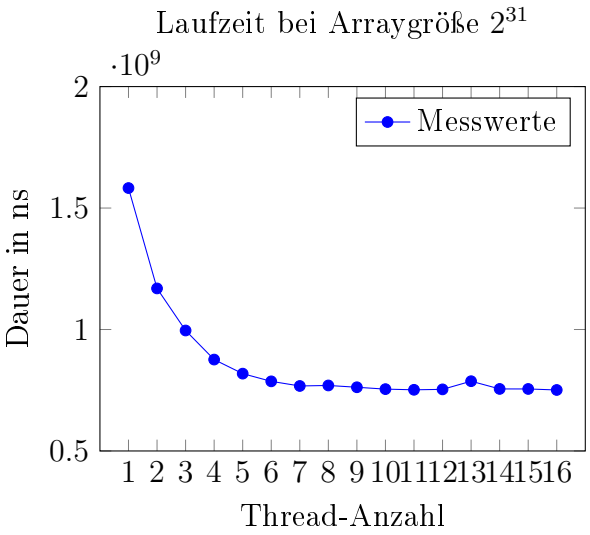
bzw. 16 logische Prozessoren) sowie eine erhöhte Rate an Cache-Misses bei steigender Thread-Anzahl die Performance negativ. Letzteres führt dazu, dass vermehrt Daten aus dem RAM geladen werden müssen, wodurch das System je nach Anwendungsfall eher durch die Bandbreite des Speichercontrollers (Memory Bound) als durch die Rechenleistung der CPU-Kerne begrenzt wird. Hierbei ist zudem zwischen physischen und logische Prozessoren zu differenzieren, da Letztere aufgrund geteilter Hardware-Ressourcen weniger effizient skalieren.
Eine weitere wesentliche Hardware-Grenze stellt die CPU selbst dar. Moderne CPUs sind durch eine maximale Leistungsaufnahme (Thermal Design Power, TDP) begrenzt. Eine höhere Auslastung aller Kerne führt daher nicht zwangsläufig zu einer proportional höheren Leistung. Beispielsweise weist die genutzte CPU einen Single-Core-Boost-Takt von 4,75 GHz auf, jedoch nur einen All-Core-Takt von 4,6 GHz, was bedeutet, dass einzelne Threads bei geringer Auslastung performanter laufen können. Zudem wird ein Großteil der TDP als Abwärme freigegeben, die effizient abgeführt werden muss. Bei unzureichender Kühlung oder Überschreitung der thermischen Grenzen kommt es zur automatischen thermischen Drosselung (Thermal Throttling) der CPU, wodurch der Takt des jeweiligen Kerns temporär erheblich reduziert wird. Diese Faktoren beeinflussen die Performance ebenfalls negativ.
Eine weitere Skalierungsgrenze stellt die Datensatzgröße dar. Übersteigt der Speicherbedarf die Kapazität des RAMs, muss das Betriebssystem Teile des Speichers auslagern (Page-Out). Da sowohl die Zugriffslatenzen als auch die Datenübertragungsraten von Sekundärspeichern (wie SSDs) signifikant schlechter sind als die des Arbeitsspeichers, führt dies zu massiven Performance-Einbußen.
Hinsichtlich der Implementierung existieren verschiedene Ansätze. Die simpelste Methode besteht darin, jeden nicht-sequentiellen Codeabschnitt in einen neuen Thread auszulagern. Dies ist jedoch oft kontraproduktiv, da ein Übermaß an Threads zu Performance-Verlusten durch Context Switching und hohen Speicherverbrauch führt. In der Praxis wird die Thread-Anzahl daher meist limitiert.
Eine Optimierung stellt die Nutzung von Worker-Threads dar. Hierbei werden Threads einmalig initialisiert und verbleiben über die gesamte Laufzeit aktiv, um kontinuierlich neue Aufgaben abzuarbeiten, anstatt nach jeder Aufgabe zerstört zu werden. Eine weiterführende Strategie ist das Dynamic Scheduling (oder Work Stealing Ansätze), bei dem Aufgaben nur dann zugewiesen werden, wenn Ressourcen frei sind. Sollten alle Worker-Threads belegt sein, kann der aufrufende Thread die Aufgabe direkt selbst bearbeiten, um Wartezeiten zu minimieren. Die Vor- und Nachteile dieser Strategien werden im Kapitel der Messungen detailliert analysiert.

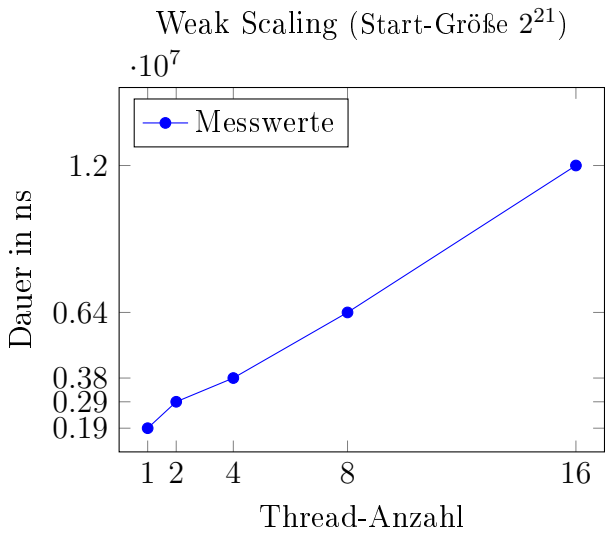
2.4 Threading anhand des einfachen Beispiels Inkrement-Array erklärt

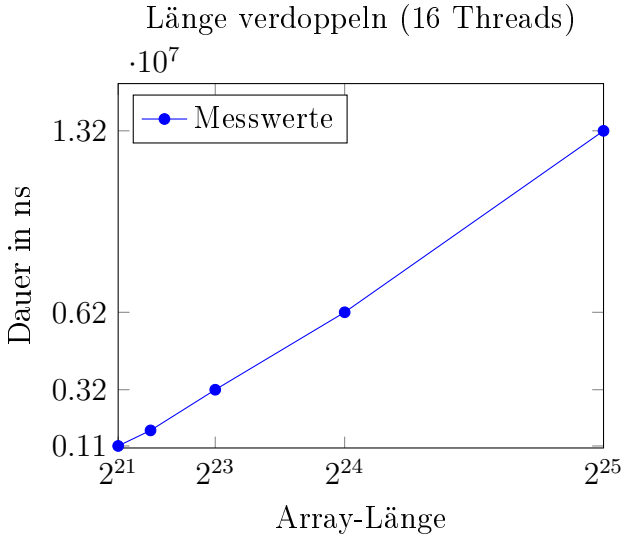












3 Methodik und Versuchsaufbau

3.1 Messumgebung und Hardware

Die relevanten Hardwarekomponenten werden im Anhang detailliert aufgelistet. Zusammenfassend wurden die Tests auf einem System mit einer 8-Kern-CPU durchgeführt (8 physische Kerne bzw. 16 logische Prozessoren) sowie mit 32 GB Arbeitsspeicher.
Als Betriebssystem kam Windows 10 in der Version 22H2 zum Einsatz. Der Code ist in der Programmiersprache `C++` implementiert und wurde mittels CMake mit dem MSVC-Compiler kompiliert. Die Ausführung der Tests erfolgte im integrierten Terminal von Visual Studio Code in einer Release-Konfiguration. Sollte hiervon abgewichen worden sein, wird dies an entsprechender Stelle gesondert angegeben.

3.2 Implementierungsvarianten

Es wurden folgende Implementierungsvarianten umgesetzt:

- rekursiv sequenzielle Variante,
- rekursive Variante mit neu erzeugten Threads bis zu einer Ebene mit einer unterstützten Thread-Anzahl von 2^N,
- Worker-Thread-Variante nach einem Work-Stealing-Ansatz mit einer unterstützten Thread-Anzahl von N, bei Quicksort als iterative Version umgesetzt.

3.3 Messmethodik

Zur Laufzeitmessung wurden zufällig erzeugte Listen verwendet, die stets mit demselben Seed initialisiert wurden. Dadurch sind alle Messungen reproduzierbar und miteinander vergleichbar.
Die Zeitmessung erfolgte mithilfe der Bibliothek `chrono`. Die gemessenen Zeiten besitzen eine Auflösung von 100 ns und wurden entsprechend in Nanosekunden gespeichert sowie in den Diagrammen dargestellt. Zur Einordnung gilt: $1\,\mathrm{s} = 10^3\,\mathrm{ms} = 10^9\,\mathrm{ns}$.
Die Messung begann unmittelbar vor dem Aufruf des zu untersuchenden Sortieralgorithmus und endete direkt nach dessen Abschluss. Die Zeit für die Initialisierung der Testlisten wurde dabei selbstverständlich nicht mitgemessen.
Nach den meisten Messungen wurde überprüft, ob die resultierende Liste korrekt sortiert ist, um die funktionale Korrektheit der Implementierung sicherzustellen.

4 Ergebnisse und Analyse

4.1 Grundlegende Laufzeiten abhängig von der Arraygröße

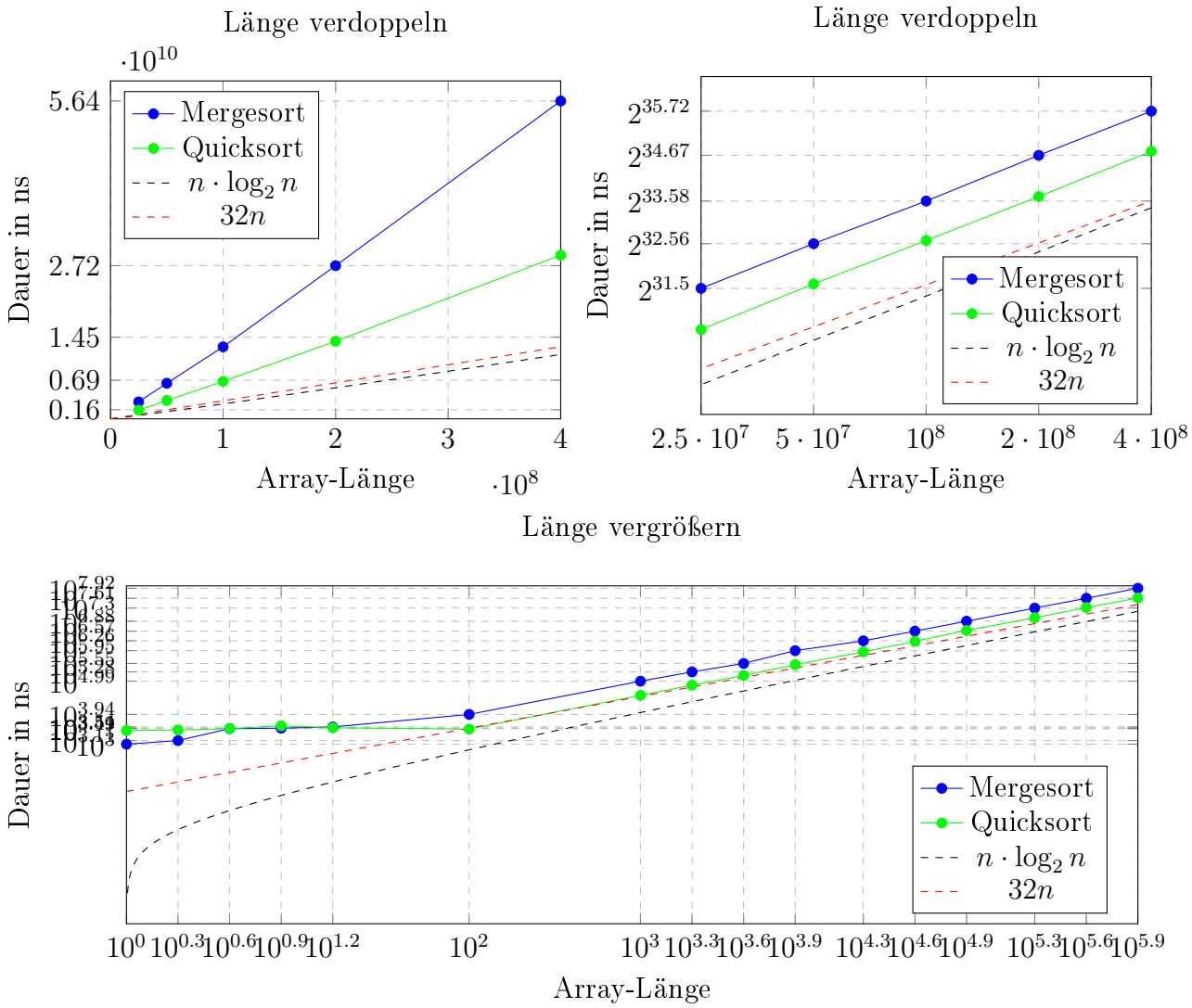
4.1.1 Messziel

Das Messziel besteht darin, die Abhängigkeit des seriellen Algorithmus von der Arraygröße grafisch darzustellen. Dadurch können diese Ergebnisse später mit den nicht-seriellen Varianten verglichen werden. Gleichzeitig dient dies als einfacher Einstieg in das Thema.

4.1.2 Erwartung

Da die durchschnittliche Laufzeit $O(n \log n)$ beträgt, erwarte ich eine logarithmische Laufzeiterhöhung bei wachsender Arraygröße.

4.1.3 Diagramm



4.1.4 Analyse und Interpretation

4.2 Einfluss des Listentyps

4.2.1 Messziel

4.2.2 Erwartung

4.2.3 Diagramm

4.2.4 Analyse und Interpretation

4.3 Einfluss der Arraygröße im Detail

4.3.1 Messziel

4.3.2 Erwartung

4.3.3 Diagramm

4.3.4 Analyse und Interpretation

4.4 Tiefenbasierte Thread-Erzeugung

4.4.1 Messziel

4.4.2 Erwartung

4.4.3 Diagramm

4.4.4 Analyse und Interpretation

4.5 Workerthreads

4.5.1 Messziel

4.5.2 Erwartung

4.5.3 Diagramm

4.5.4 Analyse und Interpretation

4.6 Vergleich der Threading-Methoden

4.6.1 Messziel

4.6.2 Erwartung

4.6.3 Diagramm

4.6.4 Analyse und Interpretation

4.7 Einfluss des Datentyps der Liste

4.7.1 Messziel

4.7.2 Erwartung

4.7.3 Diagramm

4.7.4 Analyse und Interpretation

5 Diskussion und Fazit

5.1 Interpretation aller Ergebnisse

5.2 Beantwortung der Forschungsfrage

5.3 Zusammenfassung

6 Anhang

6.1 Hardware-Spezifikationen

Zur besseren Einordnung der Leistungsfähigkeit der verwendeten Hardware befindet sich unter folgendem Link ein Benchmark:
https://www.userbenchmark.com/UserRun/70984567
Ich liste aber jetzt hier auch nochmal die relevanten Hardwarekomponenten auf.
CPU: AMD Ryzen 7 5800X, 8C/16T, 3.80-4.70GHz
CPU Kühler: be quiet! Dark Rock Pro 4)
RAM: G.Skill Aegis UDIMM 16GB Kit, DDR4-3200, CL16-18-18-38 (2 Kits, insgesamt 4x8 GB = 32 GB)
Betriebssystem: Windows 10 Version 22H2

6.2 Code

Der gesamte Quellcode dieser Arbeit ist öffentlich unter folgendem Link verfügbar:
https://github.com/Leon333M/Sortierverfahren