Untersuchung der Skalierbarkeit von parallelem Sortieren auf einem Multicore-Prozessor

Bachelorarbeit

Studiengang: Informatik

Bearbeiter: Leon Zoerner

16. 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.

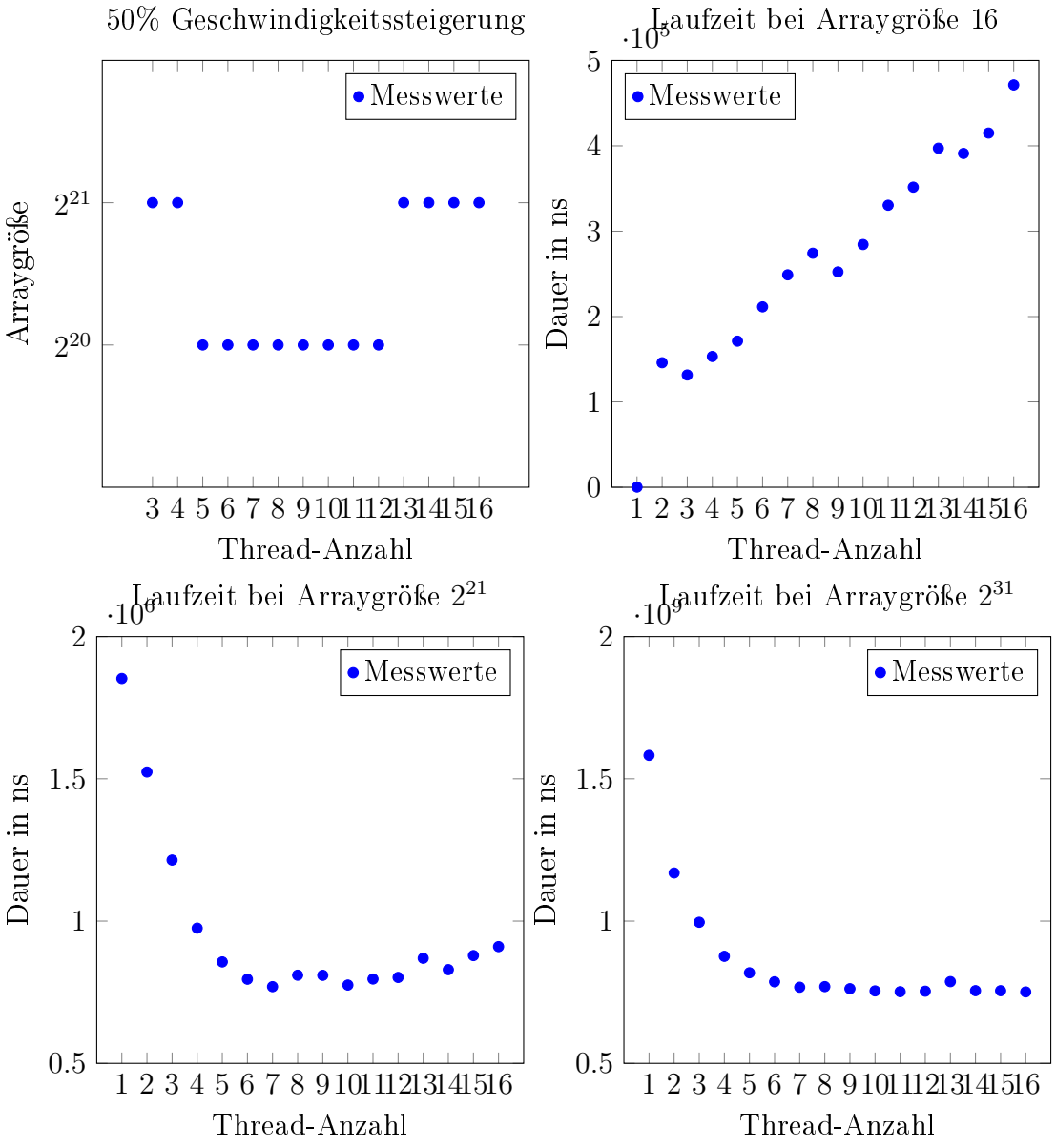
1.2 Zielsetzung und Forschungsfrage

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?

1.3 Threading anhand des einfachen Beispiels Inkrement-Array erklärt



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.

2.1.1 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 $O(n \log n)$.

2.1.2 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

$$O(n^2).$$

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

2.3 Thread-Modelle, Overheads und Skalierungsgrenzen

3 Methodik und Versuchsaufbau

3.1 Messumgebung und Hardware

3.2 Implementierungsvarianten

3.3 Messmethodik

4 Ergebnisse und Analyse

4.1 Grundlegende Laufzeiten abhängig von der Arraygröße

4.1.1 Messziel

4.1.2 Erwartung

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