# Untersuchung der Skalierbarkeit von parallelem Sortieren auf einem Multicore-Prozessor

Bachelorarbeit

Studiengang: Informatik

Bearbeiter: Leon Zoerner

3. Januar 2026

# 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

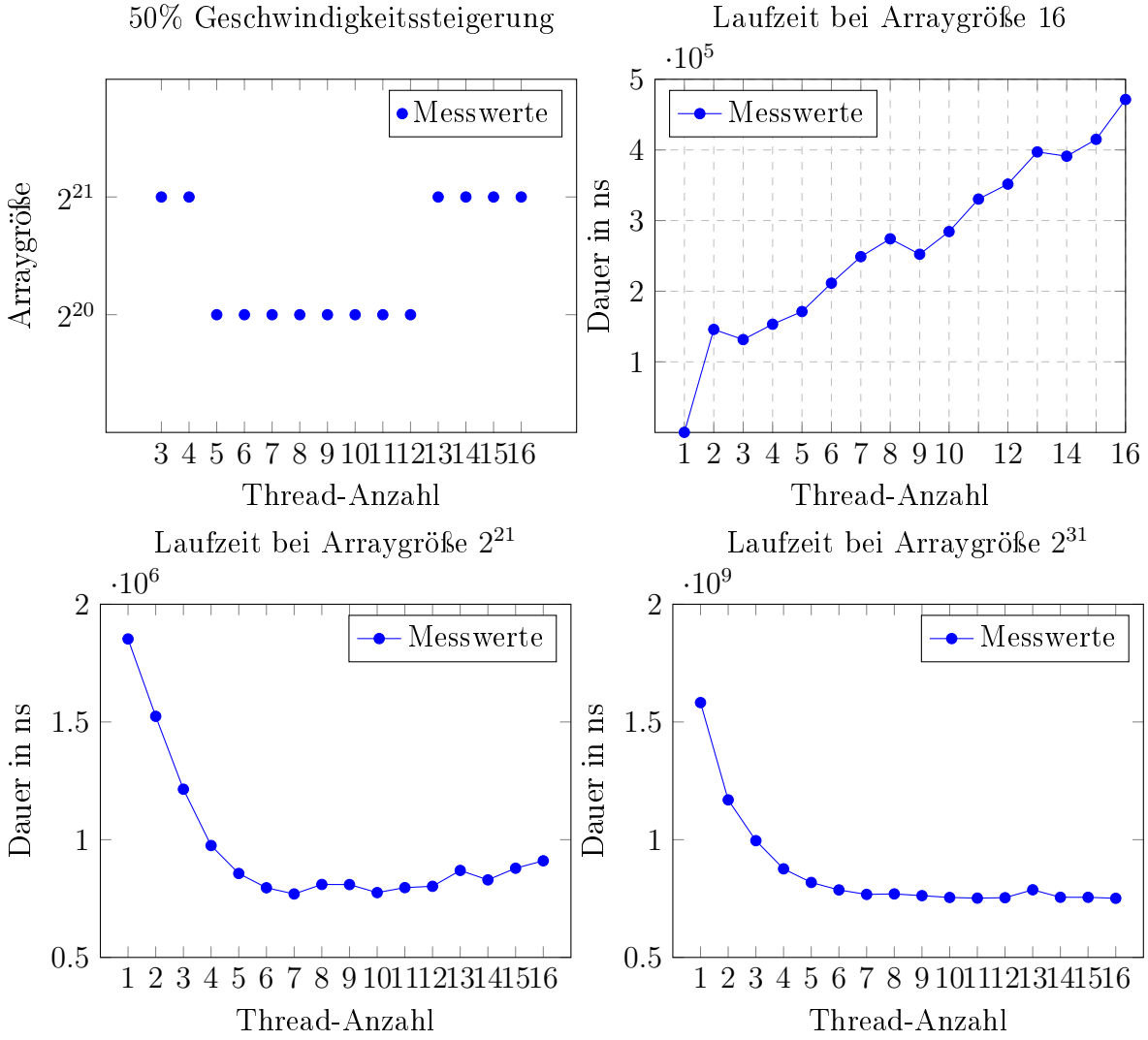
In der Praxis verursachen Threads verschiedene Overheads, die verhindern, dass eine theoretisch ideale Zeitersparnis (linearer Speedup) erreicht wird. Zu diesen Overheads zählen primär die Initialisierungs- und Join-Zeiten, die Laufzeiten von Destruktoren sowie die notwendige Synchronisation bei Abhängigkeiten 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

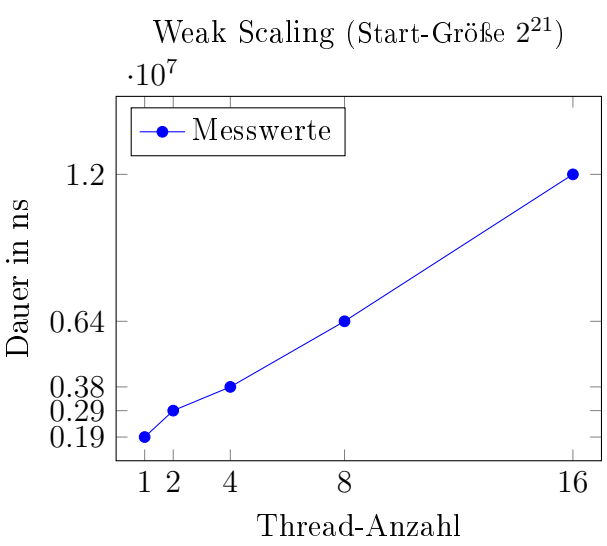
bzw. 16 logische Prozessoren) sowie eine erhöhte Rate an Cache-Misses bei steigender Thread-Anzahl die Performance negativ. Letzteres führt dazu, dass vermehrt Daten aus dem RAM geladen werden müssen, wodurch das System je nach Anwendungsfall eher durch die Bandbreite des Speichercontrollers (Memory 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.

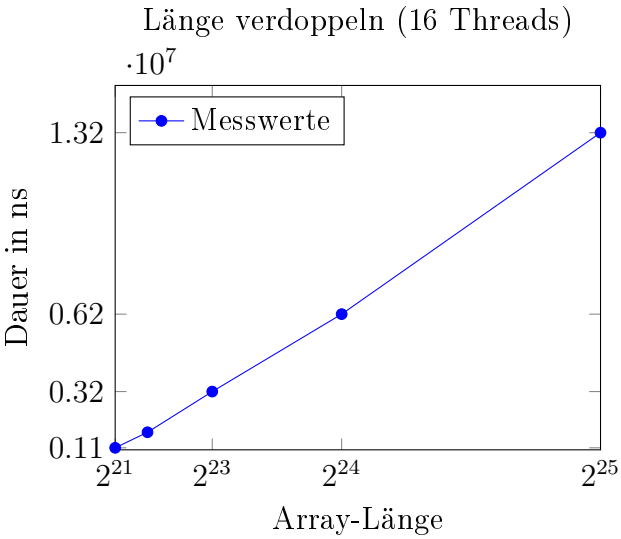
## 2.4 Threading anhand des einfachen Beispiels Inkrement-Array erklärt

Anhand dieses einfachen Beispiels möchte ich zeigen, was Parallelisierung in der Praxis bringt und dass die Praxis nicht immer mit den Erwartungen übereinstimmt. Zudem stellt dieses Beispiel einen guten Einstieg in das Thema Parallelisierung dar. An den Diagrammen kann man deutlich erkennen, dass dieses Beispiel nicht linear mit der Thread-Anzahl skaliert, obwohl es rein theoretisch so sein sollte. Das liegt wahrscheinlich daran, dass wir

am Datenübertragungslimit sind. Dies würde erklären, warum ab einem gewissen Punkt mehr ausgelastete Kerne keinen Performancegewinn mehr bringen. Zudem ist zu erkennen, dass das Array mindestens $2^{20}$ groß sein muss, damit überhaupt eine Thread-Anzahl doppelt so schnell ist wie die serielle Laufzeit. Ich möchte außerdem erwähnen, dass die serielle Laufzeit dieser Funktion normalerweise unter 100 ns liegt. Dies ist auf Compiler-Optimierungen zurückzuführen. Daher habe ich diese Funktion mit `volatile` ausgeführt, was verhindert, dass der Compiler die eigentliche Aufgabe herausoptimiert und sie somit messbar bleibt. Das `volatile`-Schlüsselwort sorgt dafür, dass bei jedem Lesevorgang die Daten aus dem RAM geladen werden müssen. Daher ist es auch das wahrscheinlichste Szenario, dass unser Beispiel durch das Datenratenlimit begrenzt ist. Zusätzlich ist die eigentliche Aufgabe trivial für die CPU und lastet diese daher nicht vollständig aus.




Weak Scaling (Start-Größe $2^{21}$)
$\cdot 10^7$
Messwerte
1.2
0.64
0.38
0.29
0.19
Dauer in ns
1 2 4 8 16
Thread-Anzahl


Länge verdoppeln (16 Threads)
$\cdot 10^7$
Messwerte
1.32
0.62
0.32
0.11
Dauer in ns
$2^{21}$ $2^{23}$ $2^{24}$ $2^{25}$
Array-Länge

# 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.

Für die Variante der tiefenbasierten Thread-Erzeugung kann man von folgender Gleichung ausgehen:

$$
\begin{aligned}
e &= \log_2(\text{Thread-Anzahl}) \\
T(n,e) &= n \cdot 2\left(1 - \frac{1}{2^e}\right) + \frac{n}{2^e} \cdot \log_2\left(\frac{n}{2^e}\right) + \frac{n}{2^e} \\
O(T(n,e)) &= O\left(n + \frac{n}{2^e} \cdot \log_2(n)\right)
\end{aligned}
$$

Man muss jedoch auch beachten, dass für Quicksort im Worst-Case weiterhin $O(n^2)$ gilt. Außerdem kann man nicht mehr Ebenen parallelisieren, als vorhanden sind, was bedeutet, dass $e_{\max} = log_2(n)$ ist. Für die Work-Stealing-Variante lässt sich keine exakte Laufzeit vorhersagen, sondern nur abschätzen. Man kann jedoch gerundet von derselben Laufzeit ausgehen wie bei der tiefenbasierten Thread-Erzeugungs-Variante.

## 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.
Für die Darstellung der theoretischen Laufzeiten in den Diagrammen gehe ich der Übersicht halber von einer direkten 1:1-Zuordnung von Arraygröße zu Dauer in ns aus. Diese Annahme ist jedoch stark vereinfacht, da sie nicht garantiert werden kann. Die erwarteten Werte sollten daher stets kritisch betrachtet werden.

# 4 Ergebnisse und Analyse

**Hinweis zum Umfang der Darstellung** Alle beschriebenen Messungen wurden vollständig durchgeführt und die entsprechenden Rohdaten liegen vor. Aufgrund des begrenzten zeitlichen Rahmens dieser Bachelorarbeit wird jedoch auf eine vollständige grafische Darstellung sowie eine detaillierte Analyse aller Messreihen verzichtet. Stattdessen werden im Folgenden ausgewählte, repräsentative Messungen dargestellt und analysiert, da diese ausreichend sind, um die theoretisch erwarteten Laufzeiteigenschaften der untersuchten Algorithmen zu bestätigen. Weitere Messdaten würden keine zusätzlichen inhaltlichen Erkenntnisse liefern, sondern lediglich bereits beobachtete Effekte wiederholen.

## 4.1 Grundlegende Laufzeiten abhängig von der Arraygröße

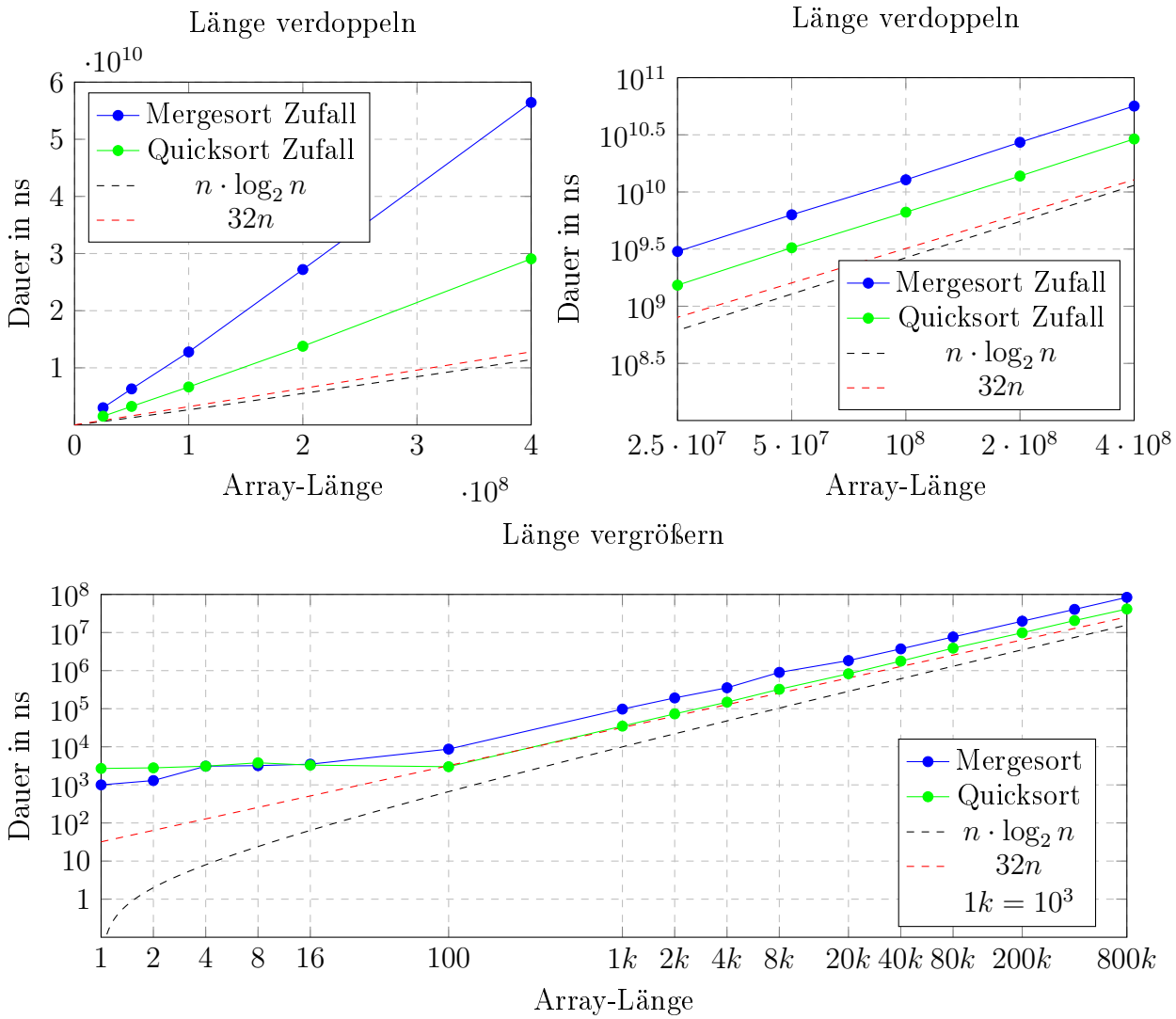
### 4.1.1 Messziel

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

In den ersten zwei Diagrammen ist die Veränderung der Laufzeit zu sehen, wenn die Listengröße fünfmal verdoppelt wird und bei $2.5 \cdot 10^7$ startet. Beim zweiten Diagramm sind die Achsen logarithmisch dargestellt, da dies die Darstellung und den Vergleich der Laufzeiten erleichtert.
Unter diesen beiden Diagrammen befindet sich ein drittes Diagramm, in dem die gemessenen Laufzeiten ebenfalls logarithmisch dargestellt sind und der Größenbereich von 1 bis 800 000 betrachtet wird.
Anhand dieser Diagramme ist deutlich erkennbar, dass sowohl Mergesort als auch Quicksort tatsächlich eine Laufzeit von $O(n \log n)$ besitzen.
Zudem ist erkennbar, dass Mergesort etwa doppelt so lange benötigt wie Quicksort und dass Quicksort näherungsweise eine Laufzeit von $2 \cdot n \log_2(n)$ aufweist.
Da eine lineare Laufzeit auf einen Blick leichter zu interpretieren ist, wurde zusätzlich die

Funktion $32n$ eingezeichnet. Anhand dieser Funktion ist erkennbar, dass sie im untersuchten Zahlenbereich von 1 bis $4 \cdot 10^8$ teilweise sogar eine genauere Abschätzung liefert als $n \log_2(n)$. Daraus folgt, dass von einer gerundeten Mindestlaufzeit von $32n$ ausgegangen werden kann.
Abschließend ist anzumerken, dass alle Messungen mit einer Laufzeit von kleiner oder gleich $10^4$ ns aufgrund der Messtoleranz nur eine eingeschränkte Aussagekraft besitzen. Zwar kann mit `chrono` auf eine Auflösung von 100 ns genau gemessen werden, dennoch verbleiben natürliche Schwankungen, die insbesondere im Bereich von $10^4$ ns einen erheblichen Einfluss auf die Messergebnisse haben.

## 4.2 Einfluss des Listentyps

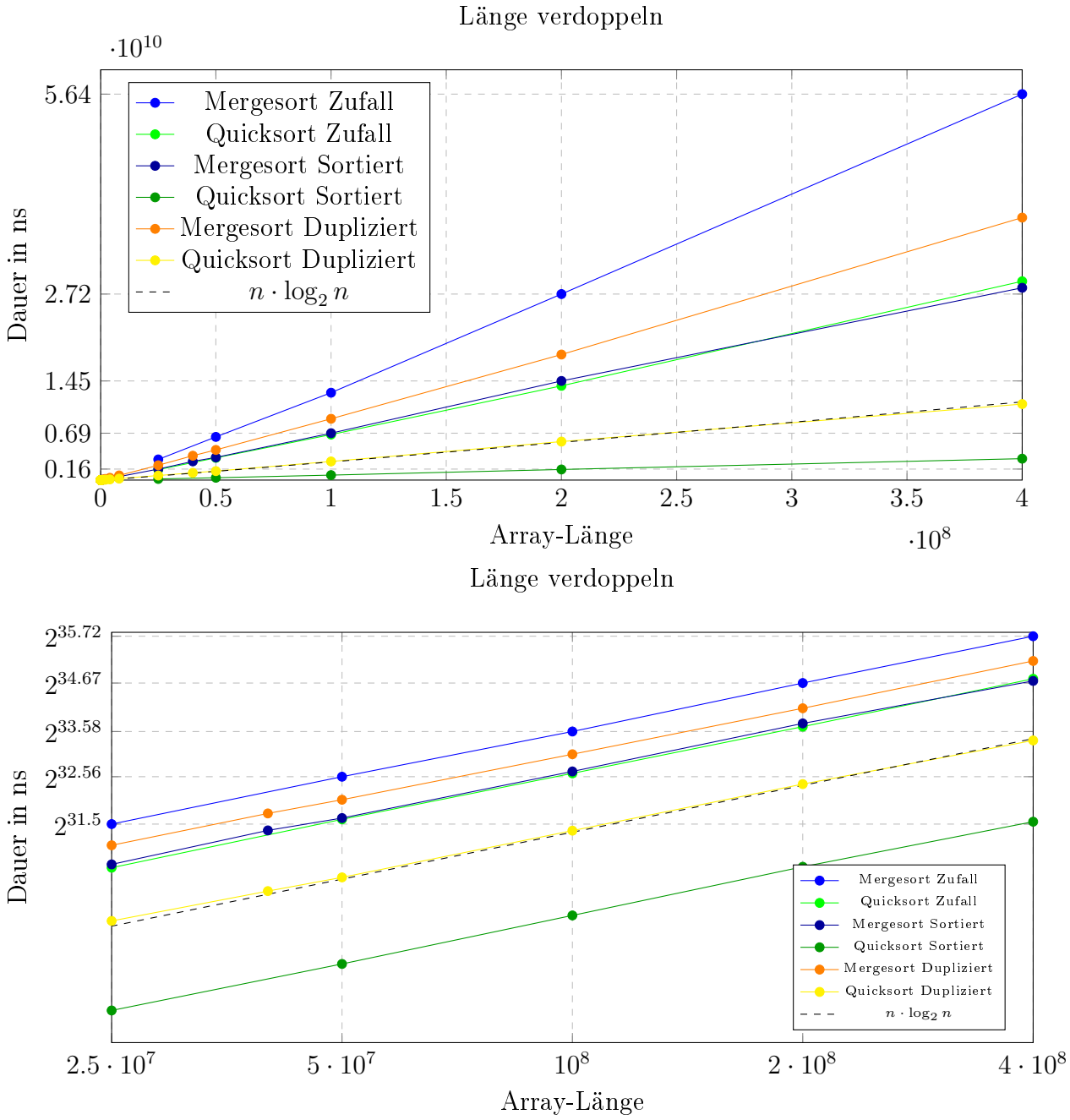
### 4.2.1 Messziel

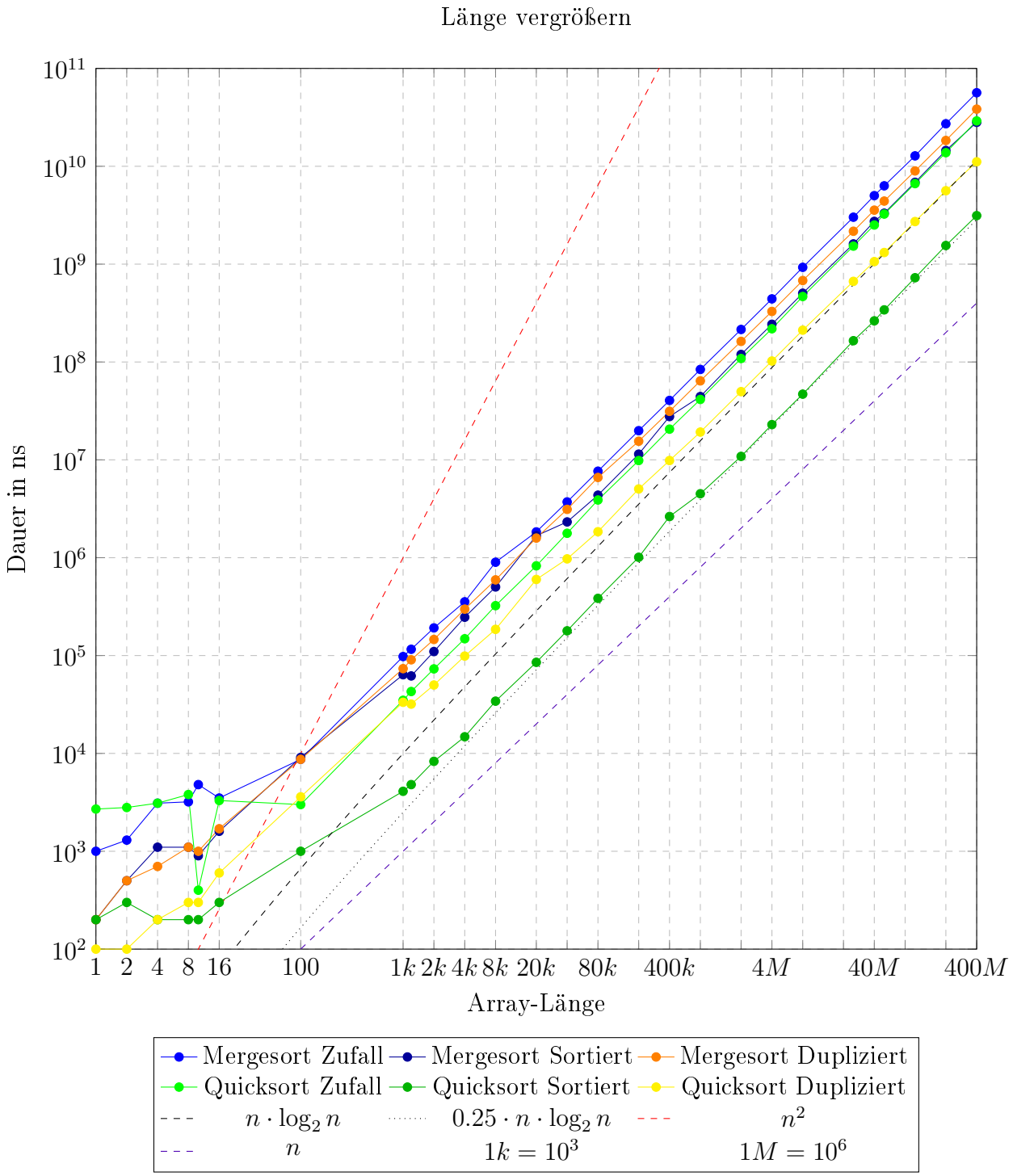
Der Listentyp ist relevant, da es bei Quicksort einen Best-Case sowie einen Worst-Case gibt. Diese hängen vom Inhalt der zu sortierenden Liste ab und somit auch vom Listentyp. Gleichzeitig dient diese Messung der Vollständigkeit, damit die Laufzeiten auch mit anderen, nicht von mir implementierten Sortieralgorithmen gut vergleichbar sind. Auch hierbei werden zunächst nur die seriellen Laufzeiten der Algorithmen gemessen. Zur Vollständigkeit werden hierbei zusätzlich weitere Listentypen außer zufällig und sortiert betrachtet.

### 4.2.2 Erwartung

Ich erwarte, dass der Listentyp bei Mergesort nahezu keinen Einfluss hat und somit nur sehr geringe Laufzeitänderungen zu beobachten sind. Bei Quicksort erwarte ich hingegen bei einer sortierten Liste exakt den Best-Case, da als Pivotelement jeweils das mittlere Element gewählt wird. Ebenso erwarte ich, dass bei Quicksort bei Wahl des jeweils ganz rechten Elements als Pivotelement bei einer sortierten Liste der Worst-Case eintritt.

### 4.2.3 Diagramm






### 4.2.4 Analyse und Interpretation

Die gemessenen Daten zeigen deutlich, dass der Best Case von Quicksort in der Praxis besser als $n \log_2(n)$ ausfällt. Dies ist darauf zurückzuführen, dass die Messungen in der Release-Version mit aktivierten Compiler-Optimierungen durchgeführt wurden. In der Debug-Version (ohne Compiler-Optimierungen) wurde bei einem sortierten Array der Größe von 400 Millionen Elementen hingegen eine Laufzeit von 14 s für Quicksort gemes-

sen, welche wiederum oberhalb der erwarteten Laufzeit von $n \log_2(n)$ liegt. Die Messungen zeigen außerdem, dass sortierte, invertiert sortierte sowie fastsortierte Arrays nahezu identische Laufzeiten erzeugen. Die entsprechenden Kurven würden nahezu aufeinanderliegen, weshalb auf eine separate grafische Darstellung dieser Listentypen zugunsten der Übersichtlichkeit verzichtet wurde.

## 4.3 Tiefenbasierte Thread-Erzeugung

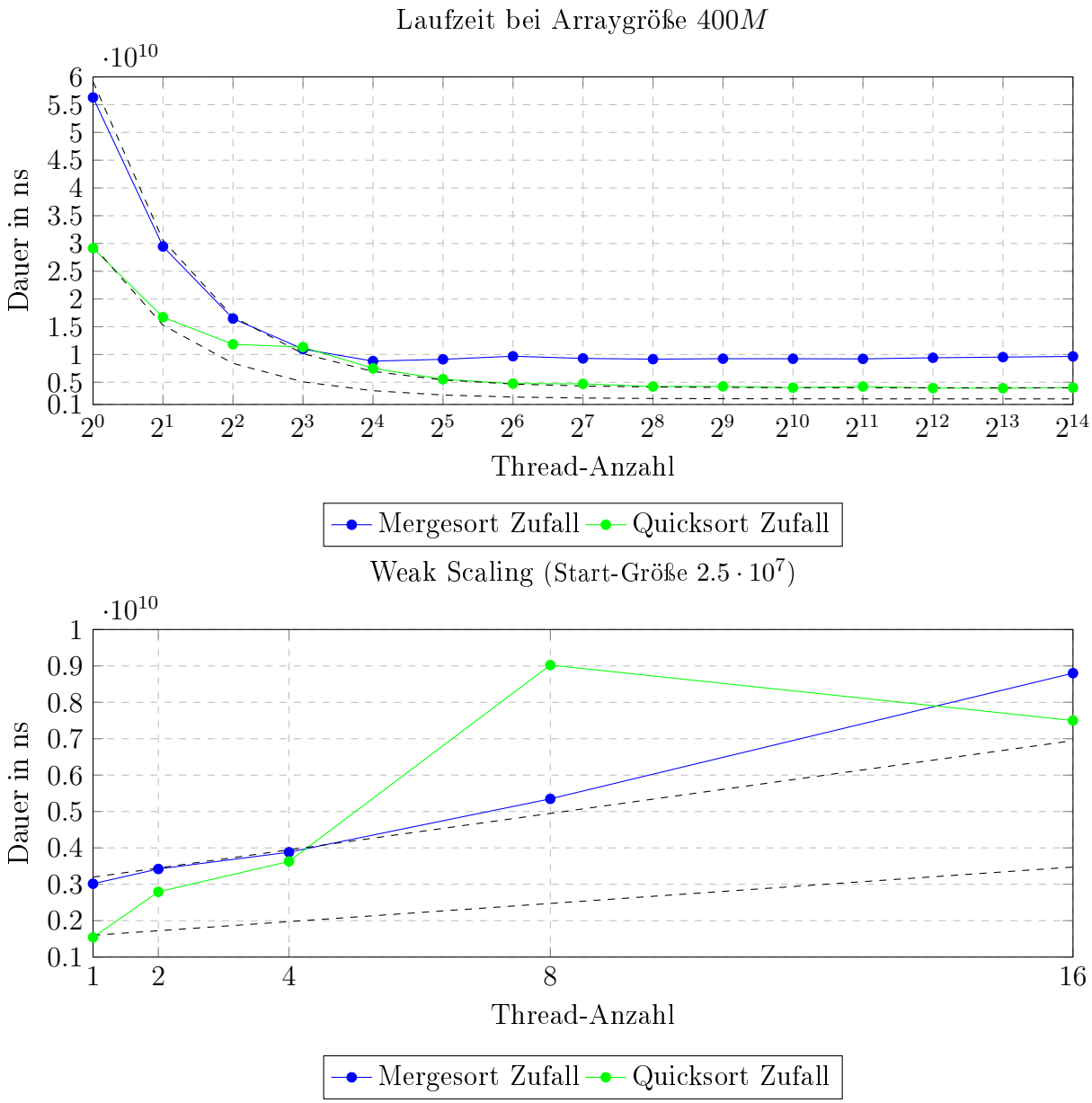
### 4.3.1 Messziel

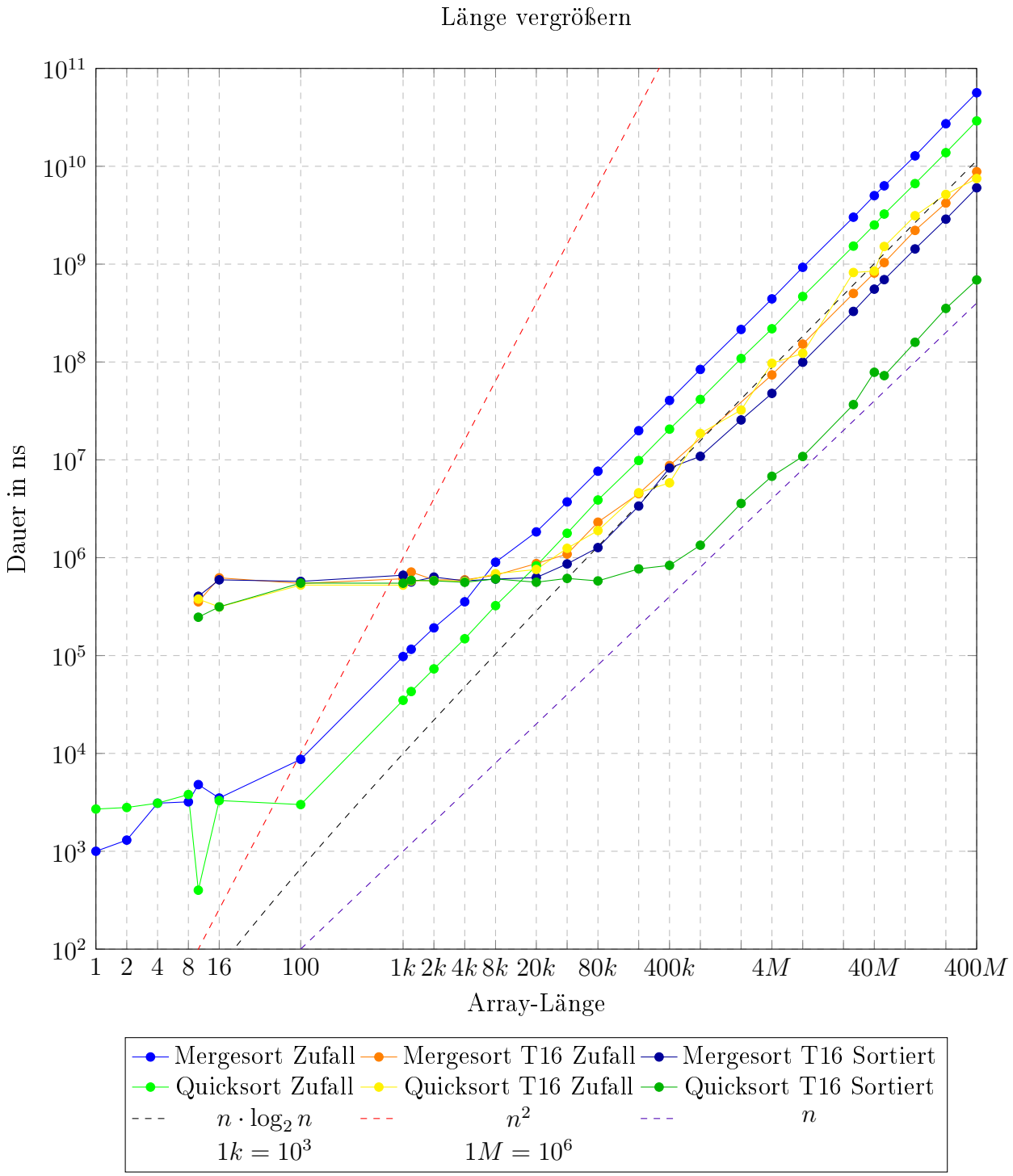
Hier messen wir die rekursive Variante, die einen der beiden rekursiven Selbstaufrufe in einem neuen Thread ausführt. Dabei gibt es jedoch ein Limit, da jeder Thread selbst Speicher benötigt und der RAM nicht unendlich groß ist. Hierbei soll gemessen werden, welchen Performance-Unterschied eine höhere Anzahl an Threads bewirkt.

### 4.3.2 Erwartung

Ich erwarte einen theoretisch linearen Performance-Zuwachs, solange kein Hardware-Limit dazwischenkommt. Zudem erwarte ich in dieser Variante, dass Mergesort besser skaliert als Quicksort, da Quicksort die Liste nicht exakt in der Mitte teilt, sondern nur dies theoretisch im Durchschnitt tut. Der Best Case von Quicksort sollte jedoch immer noch wesentlich besser sein als der von Mergesort, da Quicksort in diesem Fall die Liste immer perfekt in der Mitte teilt. Ich erwarte außerdem, dass sich die Laufzeit deutlich verschlechtert, wenn mehr Threads genutzt werden als logische Prozessoren vorhanden sind, da der Overhead durch Thread-Initialisierung und Context-Switching zunimmt.

### 4.3.3 Diagramm






### 4.3.4 Analyse und Interpretation

Entgegen der ursprünglichen Erwartung konnte kein signifikanter Performance Unterschied zwischen einer vollständig sortierten und einer nahezu sortierten Liste bei Quicksort festgestellt werden. Dieses Verhalten ist vermutlich auf Compiler-Optimierungen zurückzuführen und darauf, dass in der Implementierung nicht systematisch ein besonders ungünstiges Pivot-Element gewählt wurde.

Deutliche Performance-Verbesserungen sind jedoch in allen Bereichen messbar, wie theoretisch zu erwarten war. Für ein unsortiertes Array der Größe 400 Mio. zeigt sich, dass Mergesort mit 16 Threads lediglich etwa 16 % der Laufzeit der seriellen Variante benötigt, während Quicksort 26 % erreicht. Die Laufzeit von Mergesort wächst dabei sehr konstant, während Quicksort im Durchschnitt ebenfalls eine konsistente Steigerung der Laufzeit zeigt, jedoch sher stark schwankt.
Bemerkenswert ist, dass die 16-Thread-Variante von Quicksort nur rund 85 % der Laufzeit von Mergesort erreicht, wodurch Mergesort insgesamt überlegen ist. Außerdem wird deutlich, dass Performance-Vorteile erst ab einer Array-Größe von etwa 20.000 Elementen auftreten, was auf den Overhead der Thread-Erzeugung zurückzuführen ist.
Die starken Laufzeitschwankungen von Quicksort bei verschiedenen Listen sind auf die ungleiche Lastverteilung auf die Threads zurückzuführen. Bei dieser Quicksort-Variante wird die Thread-Last meist ungleich verteilt, was dazu führt, dass die genutzten Threads über die Laufzeit hinweg immer weniger werden und somit die gesamte Hardware nicht mehr vollständig ausgenutzt wird. Dies ist auch deutlich im Strong Scaling zu erkennen. Dort zeigt sich, dass Quicksort bei dieser Liste weiterhin an Performance gewinnt, obwohl die Threadanzahl die Anzahl der logischen Prozessoren übersteigt. Hingegen ist bei Mergesort zu beobachten, dass die gemessene Laufzeitverbesserung nur geringfügig schlechter ausfällt als die theoretisch zu erwartende.
Beim Weak Scaling ist zu erkennen, dass Quicksort dort sehr schlecht abschneidet. Dies liegt ebenfalls an der unausgeglichenen Thread-Auslastung sowie daran, dass nur eine zufällige Liste gemessen wurde und kein Median über mehrere Listen verwendet wird.
Interessant ist auch, dass man nahezu keine Performanceverschlechterung feststellt, wenn man deutlich mehr Threads verwendet, als effektiv genutzt werden können. Erst wenn so viele Threads erstellt werden, dass dafür mehr RAM benötigt wird, als vorhanden ist, lässt sich eine erhebliche Performanceverschlechterung beobachten.

## 4.4 Workerthreads

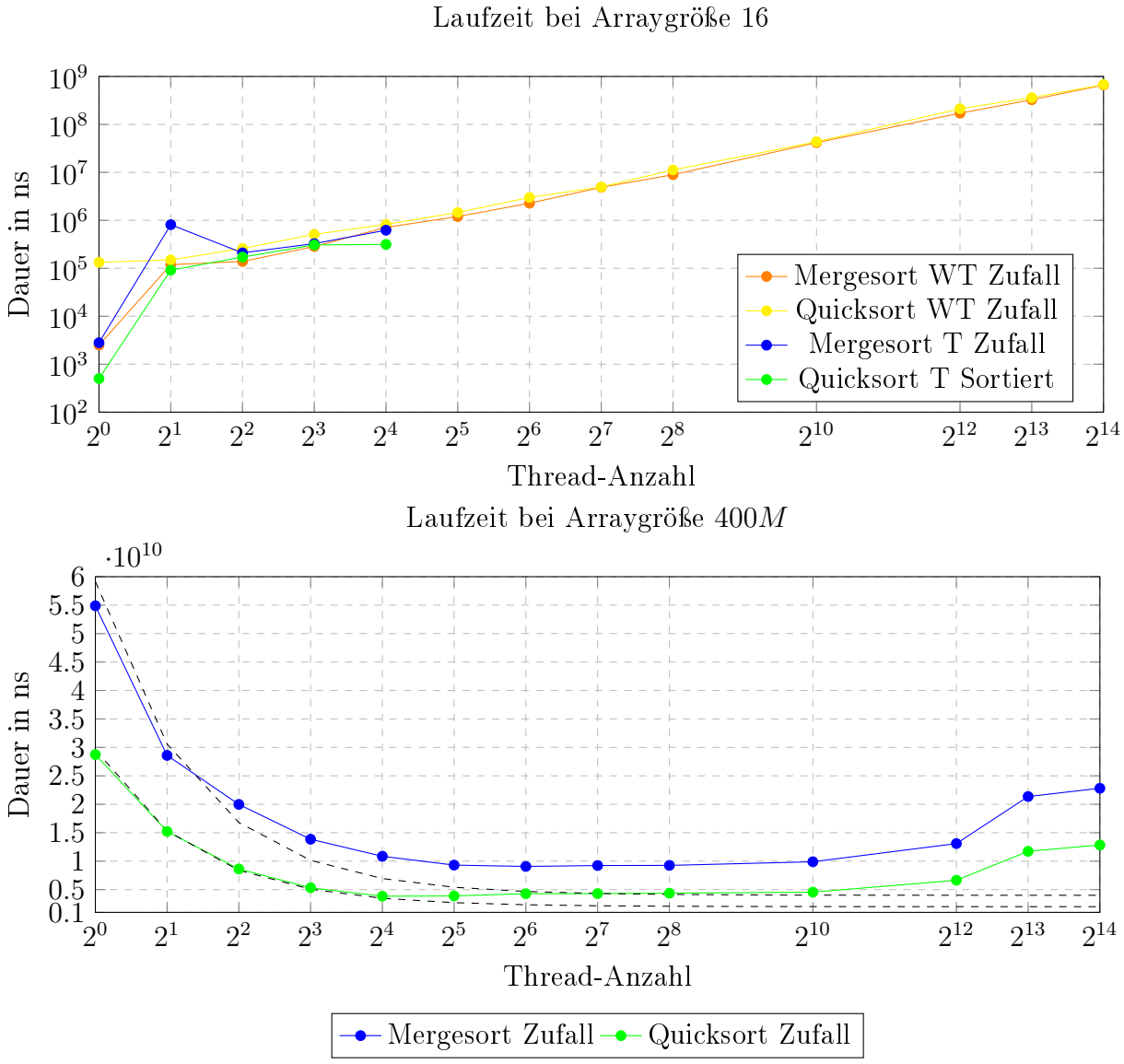
### 4.4.1 Messziel

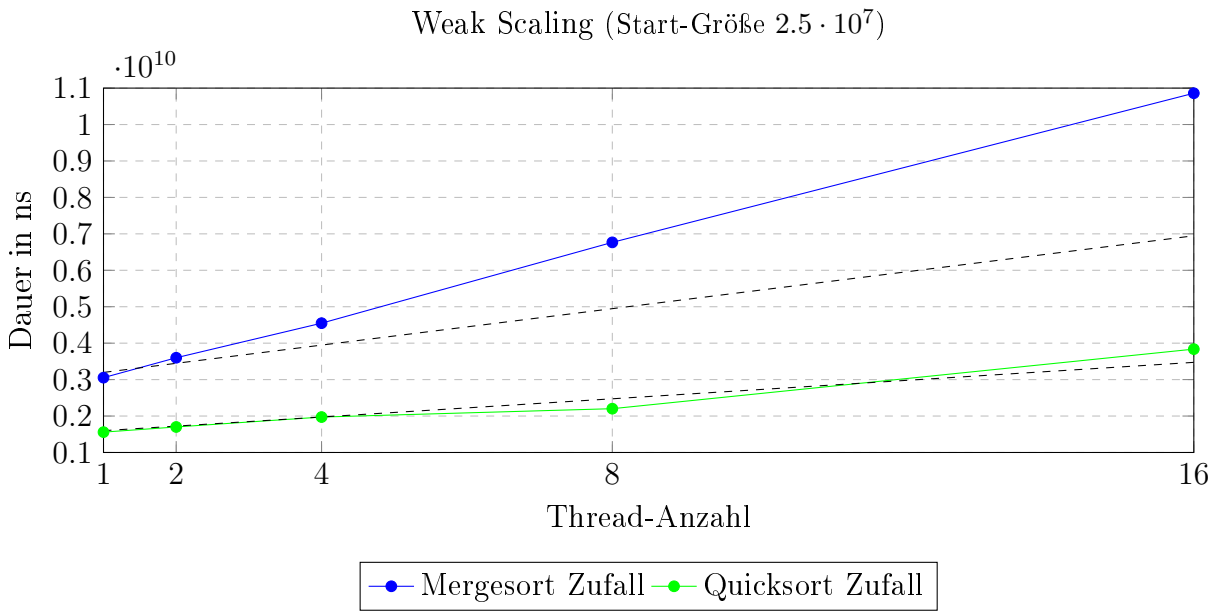
Hier messen wir die Worker-Thread-Variante nach einem Work-Stealing-Ansatz mit einer unterstützten Thread-Anzahl von $N$, wobei Quicksort als iterative Version umgesetzt ist. Zusätzlich messen wir die Zeit, die die Threads für ihre Initialisierung benötigen mit. Da in unserem Fall die Worker-Threads nach dem Sortieren nicht wiederverwendet werden, ist diese Messung nicht vollständig fair.
In der Praxis würde man die Threads weiterverwenden, wodurch dieser Overhead geringer ausfallen würde. Bei der Worker-Thread-Variante wurde außerdem eine Mindestgröße von 4.000 Elementen für neue Threads eingeführt. Dies geschieht aus dem schlichten Grund, dass sich andernfalls kein Performance-Vorteil durch zusätzliche Threads ergibt, da der entstehende Overhead sonst zu groß ist.

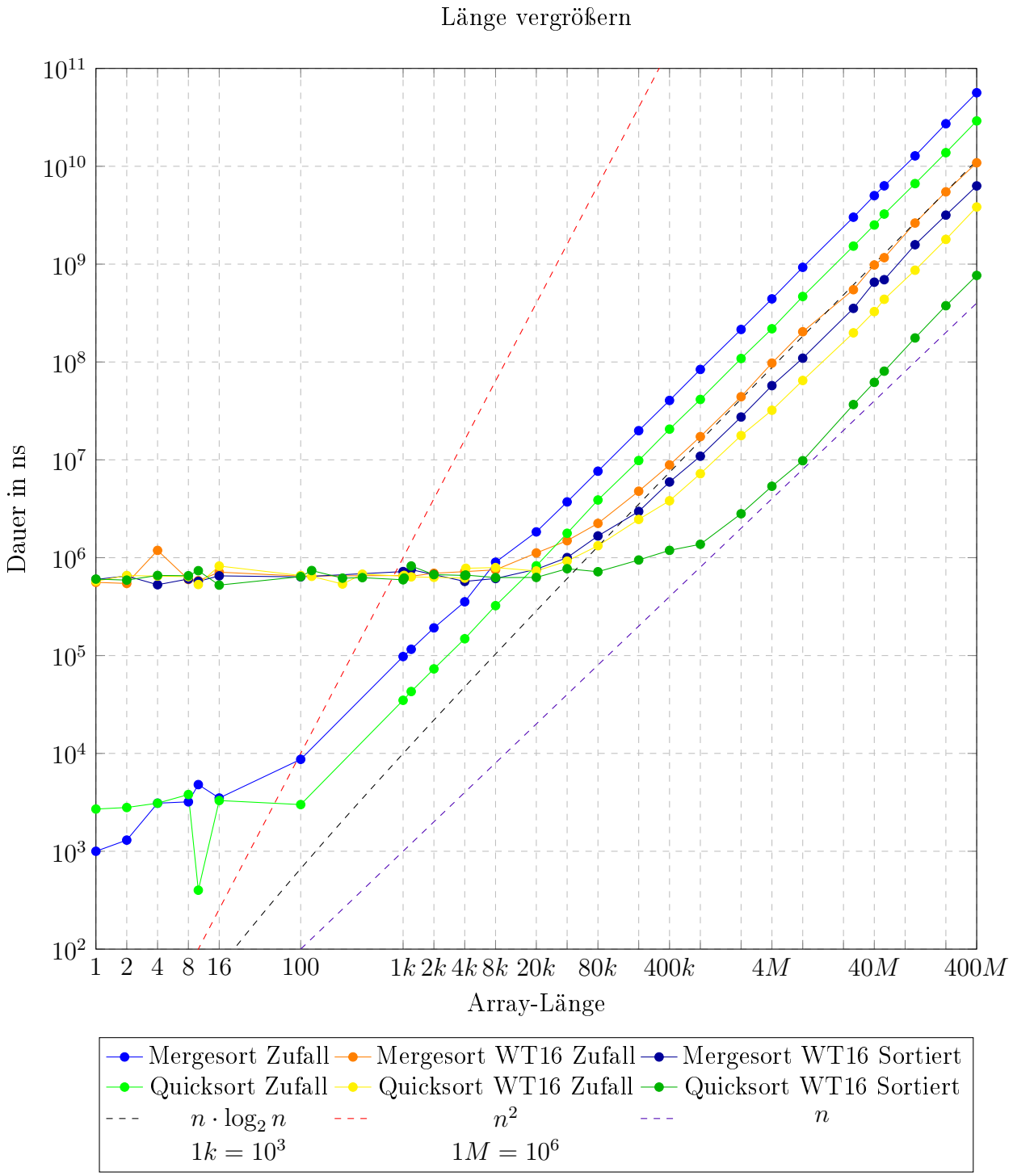
### 4.4.2 Erwartung

Es wird erwartet, dass Mergesort in der Worker-Thread-Variante schlechtere Ergebnisse erzielt als bei der tiefenbasierten Thread-Erzeugung. Dies ist darauf zurückzuführen, dass sich die Arbeit bei Mergesort in dieser Variante nur selten gleichmäßig auf die verfügbaren Threads verteilt.
Ebenso wird erwartet, dass der Best-Case von Quicksort im Vergleich zur tiefenbasierten Variante schlechter ausfällt, da auch hier keine optimale Arbeitsaufteilung erreicht wird. Für den Average-Case von Quicksort wird hingegen eine bessere Performance erwartet, da der Work-Stealing-Ansatz eine gleichmäßigere Lastverteilung über mehrere Threads ermöglicht.

### 4.4.3 Diagramm




Weak Scaling (Start-Größe $2.5 \cdot 10^7$)
$\cdot 10^{10}$
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
Dauer in ns
1
2
4
8
16
Thread-Anzahl
Mergesort Zufall
Quicksort Zufall




### 4.4.4 Analyse und Interpretation

Die Messergebnisse bestätigen die zuvor formulierten Erwartungen. Zu beachten ist, dass die Mergesort-Worker-Thread-Variante erst ab einer Array-Größe von 64.000 Elementen alle 16 Threads nutzt. Dies liegt am Mindestlimit von 4.000 Elementen für die Erzeugung neuer Threads. Man muss beachten, dass Mergesort hier langsamer ist als bei der tiefenbasierten Thread-Erzeugung, da die Last nicht mehr optimal auf alle Threads verteilt ist.

Bei Quicksort hingegen zeigt sich eine deutlich bessere Leistung, da die Last hier sehr gut auf alle Threads verteilt ist. Dies zeigt sich auch im Strong Scaling und Weak Scaling, da hier Quicksort nur minimal schlechter als die theoretisch erwartete Laufzeit ausfällt, während Mergesort im Vergleich zur anderen Variante schlechter abschneidet. Wenn man jetzt die Quicksort-Worker-Thread-Variante mit der Mergesort-tiefenbasierten Thread-Erzeugungsvariante vergleicht, stellt man immer noch fest, dass Quicksort im Durchschnitt doppelt so schnell ist wie Mergesort und im Bestfall sogar zehnmal schneller ist. Man sieht außerdem, dass beim Strong Scaling die Performance bei sehr vielen Threads wieder abnimmt. Das liegt daran, dass ich in dieser Variante Sperren für die Thread-Arbeitsverteilung einsetze, die in der anderen Variante nicht erforderlich sind. Bei sehr vielen Threads führt dies zu einem Performance-Einbruch, da immer nur ein Thread die Sperre belegen kann. Dazu kommen die Kosten der reinen Initialisierungszeit der Threads, da $2^{14}$ Threads rund 1 s benötigen, um erstellt zu werden. Im Diagramm "Laufzeit bei Arraygröße 16" sind die Overheads durch das Erstellen von Threads zu erkennen. Dabei fällt auf, dass sich diese Overheads nur minimal unterscheiden. Außerdem zeigt sich, dass die Quicksort-Worker-Thread-Variante die höchsten Initialisierungs-Overheads aufweist. Dies liegt daran, dass alle anderen Varianten ihren Main-Thread weiter nutzen, während die Quicksort-Worker-Thread-Variante dies nicht tut. Dadurch muss sie stets einen Thread mehr erstellen als die anderen Varianten, was zu diesem minimalen Overhead führt.

## 4.5 Einfluss des Datentyps der Liste

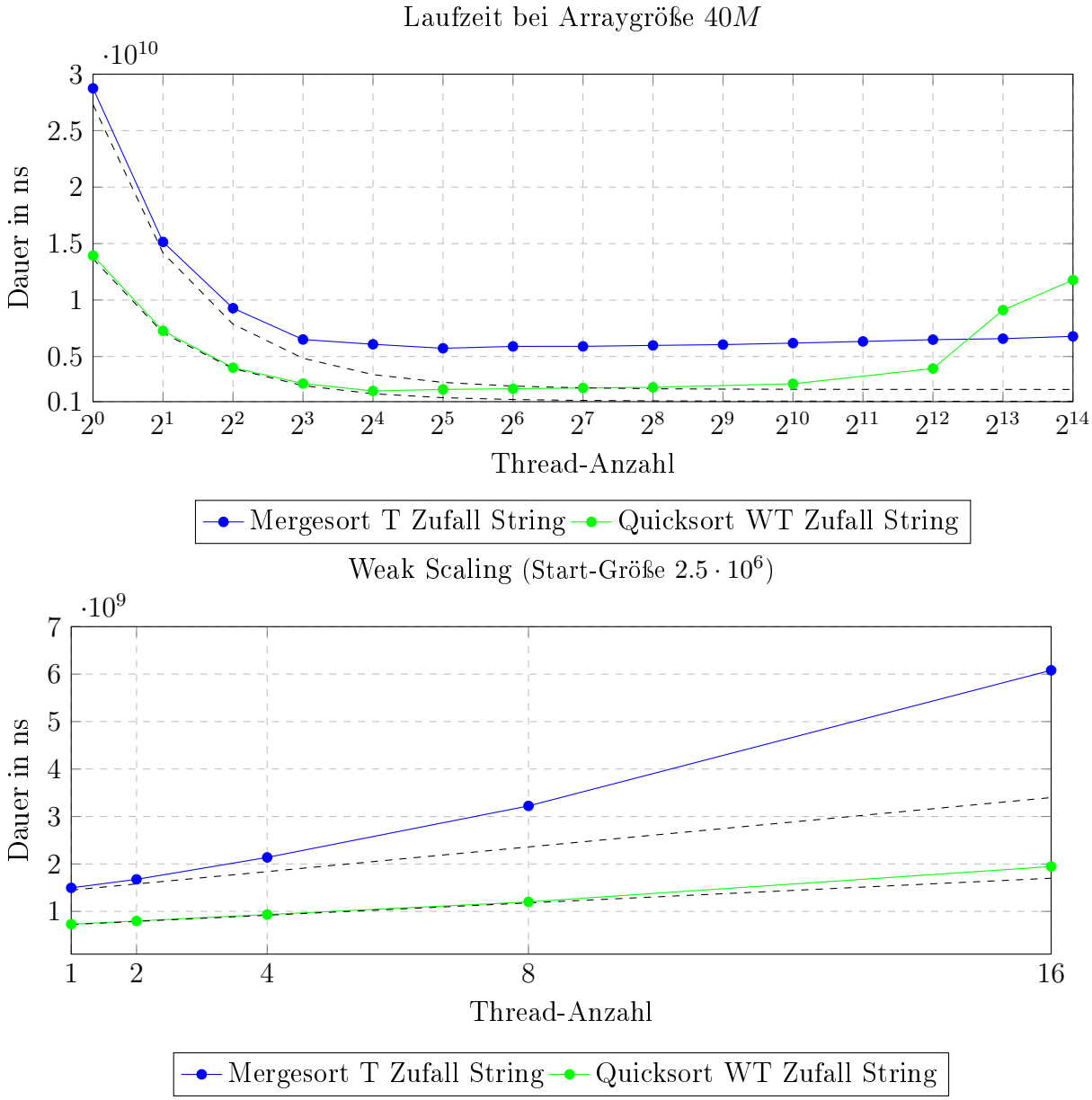
### 4.5.1 Messziel

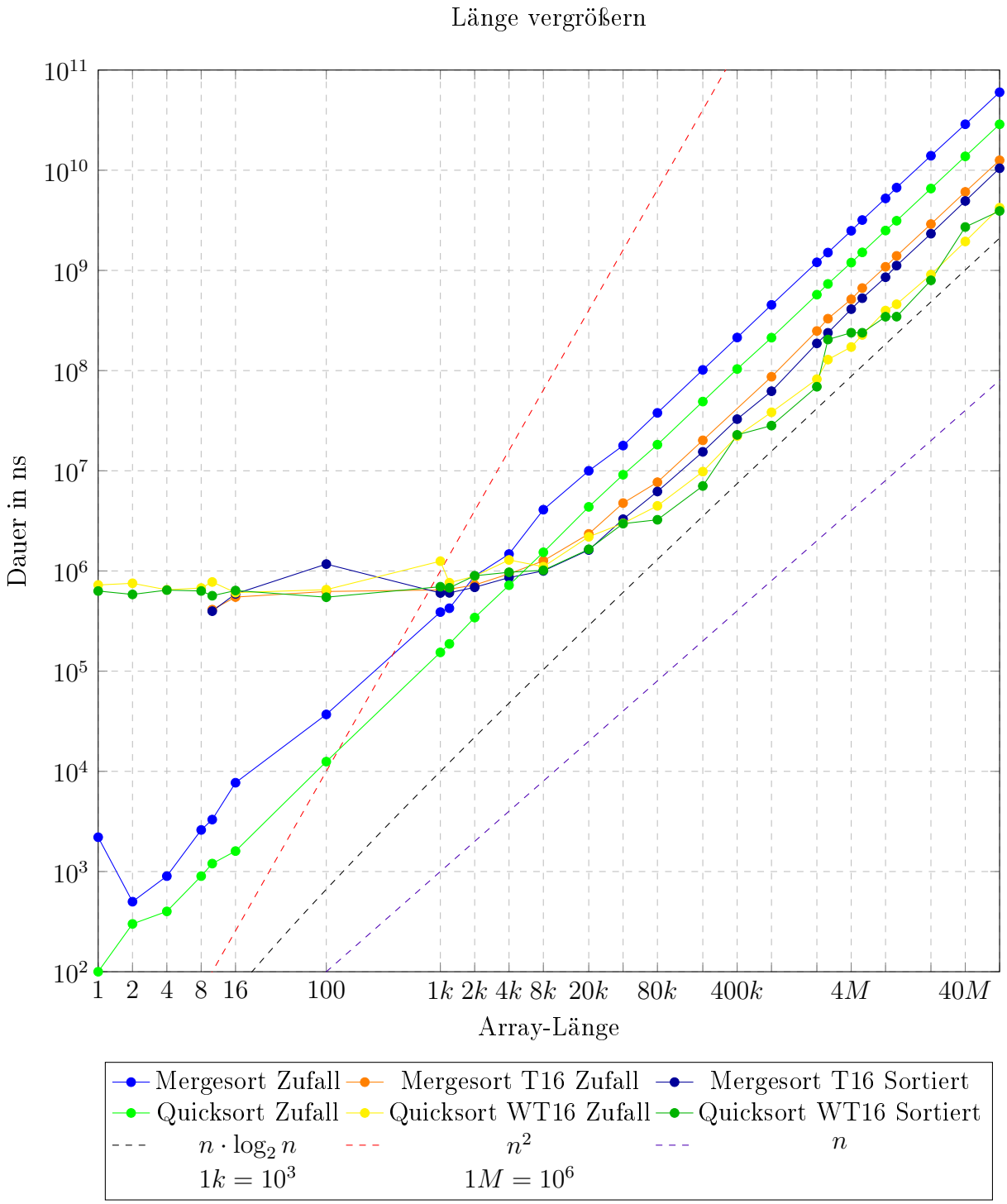
Hier wird die Zeit gemessen, die zum Sortieren von Strings benötigt wird, da dies auch ein realistischer Anwendungsfall ist.

### 4.5.2 Erwartung

Ich erwarte, dass das Sortieren länger dauert als bei `int`, da ein String wesentlich aufwendiger zu vergleichen ist, weil dieser aus mehreren Zeichen (`char`) besteht und somit auch mehr Speicher benötigt. Deswegen erwarte ich auch, dass der entstehende Overhead durch die Threads einen geringeren Einfluss auf die gemessene Endzeit hat. Dies sollte wiederum dazu führen, dass die Graphen im Strong Scaling und Weak Scaling besser abschneiden.

### 4.5.3 Diagramm






### 4.5.4 Analyse und Interpretation

Bei den Graphen zu Strong Scaling und Weak Scaling ist zu erkennen, dass Mergesort hier schlechter abschneidet als bei Integer-Daten, während Quicksort weiterhin nur minimal schlechter als die theoretische Laufzeit ist. Dass Mergesort schlechter abschneidet, liegt vermutlich daran, dass der Aufwand zum Kopieren der Teillisten bei Strings größer ist als bei Integern. Betrachtet man die Gesamtlaufzeit, fällt zudem auf, dass das Sortieren

einer bereits sortierten Liste deutlich langsamer ist, als man erwarten würde. Dies liegt an dem Code zur Listengenerierung, der für sortierte Listen deutlich längere Strings erzeugt als für zufällige Listen. Dies erhöht den Vergleichsaufwand und wirkt sich negativ auf die Laufzeit aus. Warum die Laufzeit beim Sortieren einer sortierten Liste mit Quicksort, den ich als Repräsentanten für den Best-Case nutze, so instabil ist, lässt sich nicht eindeutig erklären.

# 5 Diskussion und Fazit

## 5.1 Interpretation aller Ergebnisse

## 5.2 Ausblick und weiterführende Überlegungen

Eine theoretische Variante könnte Mergesort als Grundalgorithmus verwenden und bei Teilarrays kleiner gleich 40k auf Quicksort wechseln. Dadurch lässt sich eine absehbare Worst-Case-Laufzeit gewährleisten, während die durchschnittliche Laufzeit gegenüber reinem Mergesort verbessert wird.

Man kann ebenso die Worst-Case-Laufzeit von Quicksort unwahrscheinlicher machen. Ein Ansatz dafür ist, dass das Pivot-Element zufällig bestimmt wird. Man kann auch mehrere Elemente zufällig auswählen sowie die Listenränder und die Listenmitte in die Pivot-Auswahl einbeziehen und daraus den Median bilden. Die Anzahl der zufällig hinzugewählten Elemente könnte dabei abhängig von der Listengröße gewählt werden. Diese Methoden führen dazu, dass die Worst-Case-Laufzeit nur noch eine Frage der Wahrscheinlichkeit ist. Da die Wahrscheinlichkeit sinkt, je mehr Pivot-Elemente zufällig bestimmt werden, ist es in der Praxis dann extrem unwahrscheinlich, dass der Worst-Case jemals eintritt.

Es gibt auch Varianten, die den Partitionierungs- und Merge-Schritt parallelisiert haben. Man kann diese Variante auch noch mit der rekursiv parallelisierten Variante kombinieren, um so immer von allen Threads profitieren zu können. Dies sorgt dann wieder für eine noch bessere Laufzeit.

## 5.3 Beantwortung der Forschungsfrage

## 5.4 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

## 6.3 Quellen

Die verwendeten Quellen dienen ausschließlich der Einordnung etablierter Algorithmen und Konzepte. Alle Laufzeitanalysen und Herleitungen wurden eigenständig durchgeführt.

# Literaturverzeichnis


[1] Peter Thoman, Philipp Gschwandtner, Herbert Jordan, Thomas Fahringer, Kiril Dichev, Dimitrios S. Nikolopoulos, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Pierre Lemarinier, and Kostas Katrinis. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, 74(4):1422–1434, 2018. URL: https://link.springer.com/article/10.1007/s11227-018-2238-4, doi:10.1007/s11227-018-2238-4.

[2] Wikipedia. Amdahlsches gesetz. Zugriff am 01.01.2026. URL: https://de.wikipedia.org/wiki/Amdahlsches_Gesetz.

[3] Wikipedia. Mergesort. Zugriff am 01.01.2026. URL: https://de.wikipedia.org/wiki/Mergesort.

[4] Wikipedia. Parallel quicksort. Zugriff am 01.01.2026. URL: https://de.wikipedia.org/wiki/Parallel_Quicksort.

[5] Wikipedia. Quicksort. Zugriff am 01.01.2026. URL: https://de.wikipedia.org/wiki/Quicksort.

[6] Wikipedia. Work stealing. Zugriff am 02.01.2026. URL: https://en.wikipedia.org/wiki/Work_stealing.