HTWK

Hochschule für Technik,
Wirtschaft und Kultur Leipzig

Bachelorarbeit
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

im Studiengang Informatik
der Fakultät Informatik und Medien LFB Informatik

Untersuchung der Skalierbarkeit von parallelem Sortieren auf einem Multicore-Prozessor

vorgelegt von
Leon Andre Zoerner

Leipzig, den 2. Februar 2026

Erstprüfer: Prof. Dr. Rinke
Zweitprüfer: Prof. Dr. Weicker

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Motivation dieser Arbeit war es, herauszufinden, wie sehr man mit Threads an die theoretisch erwartete Laufzeitverbesserung herankommen kann und welche Strategie dafür am besten geeignet ist. Da sich für diese Untersuchungen ein geeigneter, leicht verständlicher und programmierbarer Anwendungsfall anbietet, werden Sortieralgorithmen betrachtet, die sich zudem sehr gut parallelisieren lassen. Dazu kommt, dass ich Recherche nach Möglichkeit vermeiden wollte. Daher war es naheliegend, selbst Code zu schreiben und diesen zu analysieren, da man hierfür weniger recherchieren muss.

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 Eingabegröße 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

Hinweis zur mathematischen Darstellung Die in dieser Arbeit genutzten mathematischen Beschreibungen und Formeln beziehen sich durchgehend auf die konkret implementierten Programmstrukturen und können daher von allgemeinen Standardformeln abweichen.

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, zwei bereits sortierte Teilarrays zu einem sortierten Array zusammenzuführen. In dieser Arbeit wird das unsortierte Eingabearray rekursiv in zwei möglichst gleich große Hälften geteilt, bis jedes Teilarray nur noch aus einem einzelnen Element besteht. Da ein Array mit einem Element per Definition sortiert ist, beginnt anschließend der sogenannte *Merge-Schritt*. In diesem Schritt werden jeweils zwei sortierte Teilarrays zu einem sortierten Gesamtergebnis zusammengeführt.
Hierfür werden beide Teilarrays mit einer Gesamtlänge von n Elementen sequenziell durchlaufen und die Elemente verglichen. Der Aufwand pro Merge-Schritt entspricht dabei n Vergleichen, da jedes Element genau einmal betrachtet wird, sowie $2n$ Lese- und Schreibzugriffen, da die Elemente temporär in ein neues Array der Größe n geschrieben und von dort wieder gelesen werden müssen.
Die Laufzeit von Mergesort lässt sich durch die Rekursionsgleichung

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$$

beschreiben, wobei der Term $+n$ den Aufwand des Merge-Schritts repräsentiert und $2 \cdot T\left(\frac{n}{2}\right)$ die beiden rekursiven Selbstaufrufe darstellt. Daraus ergibt sich eine Gesamtlaufzeit von

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

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. 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.
Der **heuristisch betrachtete Average-Case** ergibt sich aus den rekursiven Selbstaufrufen. Die Herleitung sowie die Endgleichung für die sequenzielle Laufzeit lauten:

$$
\begin{aligned}
T(n) &= q_1 + q_2 + n \\
q_1 &= T\left(\frac{1 + \cdots + (n-1)}{n-1}\right) = T\left(n \cdot \frac{n-1}{2} \cdot \frac{1}{n-1}\right) = T\left(\frac{n}{2}\right) = q_2 \\
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n \\
T(n) &= n \cdot \log_2(n) + n \\
O(T(n)) &= O(n \log n).
\end{aligned}
$$

Der **Worst-Case** tritt ein, wenn das gewählte Pivotelement in jedem Rekursionsschritt zu einer maximal asymmetrischen Partitionierung führt. Für die sequenzielle Laufzeit gilt hierbei:

$$
\begin{aligned}
T(n) &= T(n-1) + 1 + n, \\
T(n) &= \frac{1}{2} \cdot (n^2 + n) + n, \\
O(T(n)) &= O(n^2).
\end{aligned}
$$

Der **Best-Case** tritt ein, wenn das gewählte Pivotelement in jedem Rekursionsschritt zu einer exakt symmetrischen Partitionierung führt. Für die sequenzielle Laufzeit ergeben sich hierbei die Gleichungen:

$$
\begin{aligned}
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n \\
T(n) &= n \cdot \log_2(n) + n \\
O(T(n)) &= O(n \log n).
\end{aligned}
$$

Aufgrund der theoretischen Best-Case- und heuristischen Average-Case-Laufzeit ist Quicksort häufig 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

sequenzielle 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. Sequenzielle Programmanteile sowie zusätzlicher Verwaltungsaufwand, beispielsweise durch Thread-Erzeugung, Synchronisation und Kommunikation, begrenzen die erreichbare Mindestlaufzeit.
In der Praxis führt Parallelisierung daher nicht zwangsläufig zu der theoretisch erwarteten Laufzeitverbesserung, insbesondere bei steigender Anzahl von Threads.
Einfach ausgedrückt bedeutet dies, dass Code mit sequenziellen Abhängigkeiten auch bei mehreren Threads nicht schneller ausgeführt wird. Daher sollte nur der Teil des Codes parallelisiert werden, der keine solchen Abhängigkeiten enthält.

Die Formel für das **Amdahlsche Gesetz** lautet:

$$
\begin{aligned}
&p = \text{Thread-Anzahl},\\
&f = \text{serieller Code, wobei } 0 < f \leq 1,\\
t(p) = &\underbrace{f \cdot t(1)}_{\text{serielle Arbeit}} + \underbrace{(1-f) \cdot \frac{t(1)}{p}}_{\text{parallele Arbeit}}.
\end{aligned}
$$

f nahe 0 bedeutet, dass der Code fast die ganze Zeit alle Recheneinheiten beschäftigt hält. Dies impliziert, dass sein Nebenläufigkeitsgrad immer gleich oder höher als die Anzahl der parallelen Rechenknoten ist. f nahe 1 bedeutet, dass der Großteil unseres Codes nicht parallel läuft. [2]
Die Formel beschreibt jedoch nur ein lineares Skalieren mit Threads. Da dies aber bei den implementierten Varianten dieser Arbeit nicht der Fall ist, wird eine eigene $T(n,p)$ Formel genutzt, und nicht weiter auf das exakte Amdahlsche Gesetz eingegangen.

2.3 Thread-Modelle, Overheads und Skalierungsgrenzen

In der Praxis verursachen Threads verschiedene **Overheads** (Zusatzlaufzeiten), die verhindern, dass eine theoretisch ideale Zeitersparnis 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 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 und Speicherlatenz des Speichercontrollers (Memory Bound) als

durch die Rechenleistung der CPU-Kerne begrenzt wird. Zudem ist zwischen physischen und logischen Prozessoren zu unterscheiden, da letztere aufgrund geteilter Hardware-Ressourcen weniger effizient skalieren.
Eine weitere wesentliche Hardware-Grenze stellt die CPU dar. Moderne CPUs sind durch eine maximale Leistungsaufnahme (Thermal Design Power, TDP) begrenzt. Eine höhere Auslastung aller Kerne führt daher nicht zwangsläufig zu proportional höherer 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, wodurch einzelne Threads bei geringer Auslastung performanter laufen. Zudem wird ein Großteil der TDP als Abwärme freigegeben, die effizient abgeführt werden muss. Bei unzureichender Kühlung oder Überschreitung thermischer Grenzen kommt es zur automatischen thermischen Drosselung (Thermal Throttling) der CPU, wodurch der Takt des jeweiligen Kerns temporär reduziert wird. Der wenige Rest der TDP wird in elektromagnetische Felder umgewandelt, welche dann für Störströme (Kreisströme) sorgen. 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 Codeabschnitt ohne sequentielle Abhängigkeiten 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. Sind alle Worker-Threads belegt, kann der aufrufende Thread die Aufgabe selbst bearbeiten, um Wartezeiten zu minimieren. Die Vor- und Nachteile dieser Strategien werden im Abschnitt der Implementierungsvarianten (3.2) und der Messungen (4.5) detailliert analysiert.

2.4 Begriffserklärung: Worker-Thread und Work-Stealing

Ein *Worker-Thread* ist ein Thread, der einmalig erzeugt wird und über die gesamte Laufzeit des Programms aktiv bleibt. Seine Hauptaufgabe besteht darin, Aufgaben aus einer zugewiesenen Warteschlange kontinuierlich abzuarbeiten. Dies reduziert den Overhead durch ständiges Erzeugen und Zerstören von Threads und ermöglicht eine effiziente parallele Verarbeitung.

Work-Stealing bezeichnet ein Verfahren, bei dem Aufgaben in prozessorspezifischen Warteschlangen verteilt werden und jeder Prozessor auf seiner lokalen Warteschlange operiert.

Dabei können Prozessoren Aufgaben aus anderen Warteschlangen stehlen, um eine gleichmäßige Lastverteilung (*Load Balancing*) zu erhalten [1].

3 Methodik und Versuchsaufbau

3.1 Messumgebung und Hardware

Die relevanten Hardwarekomponenten sind im Anhang detailliert aufgeführt. Zusammenfassend wurden die Messungen auf einem System mit einer 8-Kern-CPU (8 physische Kerne, 16 logische Prozessoren) und 32 GB Arbeitsspeicher durchgeführt.
Als Betriebssystem kam Windows 10 in der Version 22H2 zum Einsatz. Der Code wurde in `C++20` implementiert und mittels CMake (Version 3.26.4) sowie dem MSVC-Compiler (Version 14.44.35207) unter Verwendung des `MultiThreaded`-Flags kompiliert. Die Ausführung der Messungen erfolgte im integrierten Terminal von Visual Studio Code in einer Release-Konfiguration (also mit Compiler-Optimierungen). Abweichungen hiervon werden an entsprechender Stelle gesondert angegeben.

3.2 Implementierungsvarianten

Im Rahmen dieser Arbeit wurden folgende Implementierungsvarianten für Mergesort und Quicksort umgesetzt:

- eine rekursiv sequenzielle Variante,
- eine rekursive Variante mit dynamischer Thread-Erzeugung bis zu einer Tiefe e, welche eine Thread-Anzahl von 2^e unterstützt (tiefenbasierte Thread-Erzeugung),
- eine Worker-Thread-Variante basierend auf einem Work-Stealing-ähnlichen Ansatz mit N Threads, umgesetzt als aufgabenbasierte Rekursionsverwaltung (Work-Stealing-Variante).

Für die tiefenbasierte Thread-Erzeugung ändert sich in der parallelen Phase die Rekursionsgleichung von $T(n) = 2 \cdot T(n/2) + n$ zu $T(n) = 1 \cdot T(n/2) + n$. Dies liegt daran, dass die zweite Hälfte der Arbeit parallel in einem anderen Thread verarbeitet wird und somit nicht zur Laufzeit der aktuellen Ebene beiträgt. Durch das Auflösen dieser Rekursion über die parallelen sowie anschließend sequenziellen Ebenen hinweg lässt sich die theoretische Laufzeit in eine geschlossene Form überführen. Daraus ergibt sich die theoretische Laufzeit von Mergesort sowie die Best-Case-Laufzeit von Quicksort bei der tiefenbasierten Thread-Erzeugung, die sich durch folgende Gleichungen beschreiben lassen, ohne die

durch die Threads entstehenden Overheads zu berücksichtigen:

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

Hierbei ist zu beachten, dass die Parallelisierung auf die vorhandene Rekursionstiefe begrenzt ist, woraus $p_{\max} = n$ folgt. Da die Rekursionstiefe eines balancierten Binärbaums auf $\log_2(n)$ Ebenen begrenzt ist, ergibt sich $e_{\max} = \log_2(n)$. Weil ebenenweise parallelisiert wird, gilt $e = \log_2(p)$, was schließlich zu $p_{\max} = n$ führt. Zudem besteht für Quicksort im Worst-Case bei $p > 1$ folgende theoretische Laufzeit:

$$
\begin{aligned}
T(n) &= T(n-1)+n, \\
T(n) &= \frac{1}{2}\cdot(n^2+n), \\
O(T(n)) &= O(n^2).
\end{aligned}
$$

Für die Work-Stealing-Variante ist auch keine exakte Vorhersage der Laufzeit möglich, weshalb eine Abschätzung erfolgt. Dies liegt daran, dass sich die Threads frei die Arbeit teilen und nicht mehr fest an Ebenen gebunden sind, wodurch ein zusätzlicher Overhead entsteht. Unter der Annahme, dass die zusätzlichen Overheads ignoriert werden, ist die theoretische Laufzeit der Work-Stealing-Variante identisch mit der tiefenbasierten Variante. Natürlich unterscheiden sich die Average-Case-Laufzeiten beider Varianten in der Praxis dramatisch. Für diese Arbeit ist es jedoch ausreichend, nur den auch durch $T(n,p)$ beschriebenen heuristischen Average-Case zu analysieren.

Bildlich gesprochen lässt sich Mergesort als balancierter und Quicksort als unbalancierter binärer Baum darstellen. Während die tiefenbasierte Thread-Erzeugung bis zu einer festen Ebene bei Mergesort zu einer optimalen Auslastung führt, resultiert dies bei Quicksort in einer schlechten Lastverteilung. Im Gegensatz dazu ermöglicht die Work-Stealing-Variante, dass freie Threads Aufgaben aus anderen Ästen (dem linken Teilbaum) übernehmen. Dadurch wird auch bei unbalancierten Baumstrukturen eine sehr gleichmäßige Lastverteilung erzielt, was wiederum in einer verkürzten Laufzeit resultiert.

3.3 Begriffserklärung Work-Stealing-ähnliche Ansatz

Der *Work-Stealing-ähnliche Ansatz* in dieser Arbeit basiert auf einem Worker-Thread-Pool mit einer gemeinsamen Aufgabenwarteschlange. Alle Worker-Threads ziehen Aufgaben aus dieser Warteschlange und bearbeiten sie. Dazu kann jeder Worker-Thread auch neue Aufgaben in die Warteschlange legen. Ist die Queue leer, warten die Threads auf neue Aufgaben. Dadurch wird sichergestellt, dass alle Threads möglichst konstant ausgelastet sind, ohne dass für jeden Thread eine eigene Warteschlange benötigt wird. Dies reduziert den Overhead durch Synchronisation und vermeidet Leerlaufzeiten, ähnlich wie beim klassischen Work-Stealing.

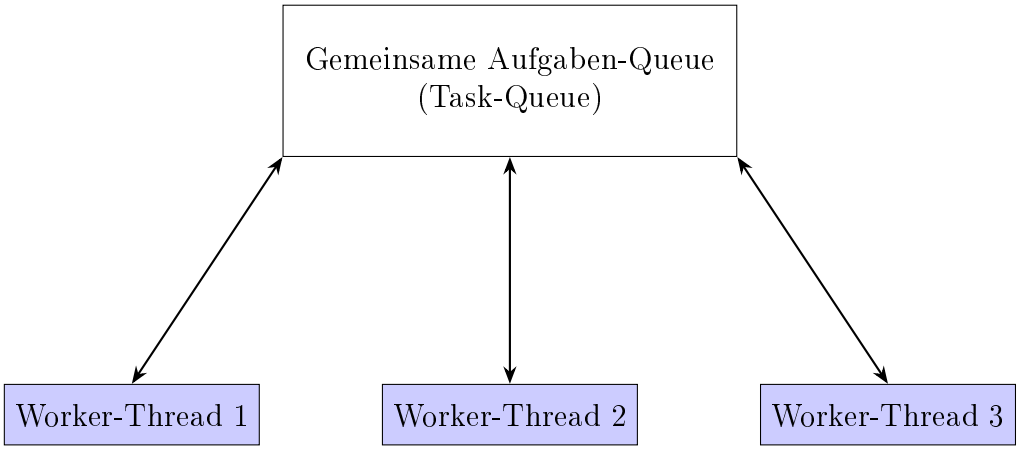

Abbildung 3.1: Work-Stealing-ähnlicher Ansatz mit gemeinsamer Aufgaben-Queue: Threads entnehmen Aufgaben aus der Queue und können neue Aufgaben auf diese legen.

3.4 Parallele Implementierungsvarianten Code

Listing 3.1: Tiefenbasierte Thread-Erzeugung Mergesort

```
1  void Mergesort::mergesortP(int *liste, const int links, const int
       rechts, const int aktuelleEbene, const int neueThreadsBisEbene)
       {
2      if (aktuelleEbene < neueThreadsBisEbene) {
3          int lange = rechts - links + 1;
4          if (lange > 1) {
5              int mitte = links + ((rechts - links) / 2);
6              // mergesort(liste, links, mitte);
7              std::thread thread(
8                  static_cast<void (*)(int *, const int, const int,
                       const int, const int)>(&Mergesort::mergesortP),
9                  liste, links, mitte, aktuelleEbene + 1,
                       neueThreadsBisEbene);
10             // mergesort(liste, mitte + 1, rechts);
11             mergesortP(liste, mitte + 1, rechts, aktuelleEbene + 1,
                   neueThreadsBisEbene);
12             thread.join();
13             mischen(liste, links, mitte, rechts, lange);
14         }
15     } else {
16         mergesort(liste, links, rechts);
17     }
18 };
```

Listing 3.2: Tiefenbasierte Thread-Erzeugung Quicksort

```
1  void Quicksort::quicksortP(int *liste, const int links, const int
       rechts, const int aktuelleEbene, const int neueThreadsBisEbene)
       {
```

```
    if (aktuelleEbene < neueThreadsBisEbene) {
        if (links < rechts) {
            int ml, mr;
            partitioniere(liste, links, rechts, ml, mr);
            // quicksort(liste, links, ml);
            std::thread thread(static_cast<void (*)(int *, const
                int, const int, const int, const int)>(&Quicksort::
                quicksortP), liste, links, ml, aktuelleEbene + 1,
                neueThreadsBisEbene);
            // quicksort(liste, mr, rechts);
            quicksortP(liste, mr, rechts, aktuelleEbene + 1,
                neueThreadsBisEbene);
            thread.join();
            }
    } else {
        quicksort(liste, links, rechts);
    }
};
```

Listing 3.3: Worker-Thread-Variante Mergesort

```
void Mergesort::mergesortW(int *liste, int links, int rechts, int
    workerThreads) {
    MergeWorkerPool pool(workerThreads - 1);
    pool.taskHandler = [&](int *liste, int links, int rechts,
        MergeWorkerPool &pool) {
        if (links < rechts) {
            int lange = rechts - links + 1;
            if (lange < Sortierverfaren::mindestLange) {
                mergesort(liste, links, rechts);
            } else {
                int mitte = links + ((rechts - links) / 2);
                auto leftHandle = pool.addTaskSmart({liste, links,
                    mitte});
                pool.taskHandler(liste, mitte + 1, rechts, pool);
                leftHandle.wait();
                mischen(liste, links, mitte, rechts, lange);
            }
        }
    };
    // Starttask
    pool.taskHandler(liste, links, rechts, pool);
}
```

Listing 3.4: Worker-Thread-Variante Quicksort

```
void Quicksort::quicksortW(int *liste, int links, int rechts, int
    workerThreads) {
    WorkerPool pool(workerThreads);
    pool.taskHandler = [](int *liste, int links, int rechts,
        WorkerPool &pool) {
```

```
4           if (links < rechts) {
5               if (rechts - links < Sortierverfaren::mindestLange) {
6                   quicksort(liste, links, rechts);
7               } else {
8                   int ml, mr;
9                   Quicksort::partitioniere(liste, links, rechts, ml,
                        mr);
10                  pool.addTask({liste, links, ml});
11                  pool.taskHandler(liste, mr, rechts, pool);
12              }
13          }
14      };
15      pool.addTaskWaitUntilDone({liste, links, rechts});
16  }
```

3.5 Optimierung

Auf eine Optimierung des Codes wurde bewusst verzichtet, da primär der Vergleich der Threading-Strategien im Fokus steht und eine perfekte Implementierung zu zeitaufwendig gewesen wäre.
Speziell die Eliminierung der Wartezeiten (Wait) in der Mergesort-Worker-Thread-Variante (Listing 3.3) durch einen Task-Drop-Mechanismus wäre zu aufwendig gewesen und würde den Rahmen eines einfachen Beispiels sprengen. Aus Gründen der Fairness wurde daher konsequent auf Optimierungen bei allen Varianten verzichtet, um keine künstlichen Vorteile zu schaffen und die Vergleichbarkeit der Ergebnisse zu gewährleisten.

3.6 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 nicht mitgemessen.
Nach den meisten Messungen wurde überprüft, ob die resultierende Liste korrekt sortiert ist, um die funktionale Korrektheit der Implementierung sicherzustellen.
In der vorliegenden Untersuchung wird jeder Datenpunkt als Einzelergebnis (Stichprobenmessung) aufgeführt. Dieser Ansatz wurde gewählt, um die reale Systemperformance unter wechselnden Lastbedingungen unverfälscht darzustellen. Da jede Messung ein in der Praxis aufgetretenes Laufzeitverhalten repräsentiert, wird auf eine Mittelwertbildung

verzichtet, um auch punktuelle Latenzen oder Ausreißer, die für die Stabilität des Algorithmus relevant sind, sichtbar zu machen.
Für die Darstellung in den Diagrammen wird der Übersicht halber von einer direkten 1:1-Zuordnung von theoretischer Laufzeit zu Dauer in ns ausgegangen. Diese Annahme ist jedoch stark vereinfacht, da sie nicht garantiert werden kann. Die erwarteten Werte sollten daher stets kritisch betrachtet werden.
Dafür folgt hier ein praktisches Beispiel: Damit der theoretischen Laufzeit eine Zeiteinheit zugeordnet werden kann, wird eine Variable benötigt, die wir x nennen. Dieses x repräsentiert die Zeit, die ein einzelner Vergleich benötigt. Es ergeben sich folgende Gleichungen:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + x \cdot n$$
$$T(n) = x \cdot (n \cdot \log_2(n) + n)\,.$$

Wenn die Vergleichszeit $x = 1\,\text{ns}$ beträgt, liegt eine direkte 1:1-Zuordnung vor. In der Praxis könnte x jedoch variieren und beispielsweise einen Best-Case von $0,25\,\text{ns}$ sowie einen Worst-Case von $1\,\text{ms}$ aufweisen. Durch diese Schwankungen, die unter anderem durch Speicherlatenzen entstehen, lässt sich die theoretische Laufzeit nie perfekt 1:1 zuordnen. Sie wird in der Realität immer schwanken.

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 Begriffsdefinition: Strong und Weak Scaling

Unter *Strong Scaling* versteht man die Untersuchung der Laufzeit eines festen Problemumfangs bei steigender Anzahl an Recheneinheiten (Threads). Ziel ist es zu analysieren, wie stark sich die Laufzeit durch zusätzliche Parallelisierung verkürzt.

Unter *Weak Scaling* versteht man die Untersuchung der Laufzeit, bei der der Problemumfang proportional zur Anzahl der Recheneinheiten wächst. Ziel ist es zu bewerten, ob die Laufzeit bei wachsender Parallelität konstant bleibt.

4.2 Threading anhand des einfachen Beispiels Inkrement-Array erklärt

Anhand dieses einfachen Beispiels soll gezeigt werden, was Parallelisierung in der Praxis bewirkt 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 (Abbildung 4.3, 4.4) ist deutlich zu erkennen, dass dieses Beispiel nicht linear mit der Thread-Anzahl skaliert, obwohl dies rein theoretisch zu erwarten wäre. Dies liegt wahrscheinlich an der Speicherlatenz als limitierendem Faktor. Dies würde erklären, warum ab einem gewissen Punkt mehr ausgelastete Kerne keinen weiteren Performancegewinn mehr bringen. Zudem ist zu erkennen (Abbildung 4.1), dass das Array mindestens 2^{20} groß sein muss, damit eine parallele Ausführung einen mindestens zweifachen Geschwindigkeitsvorteil gegenüber der sequenziellen Laufzeit erreicht. Zusätzlich ist anzumerken, dass die sequenzielle Laufzeit dieser Funktion normalerweise unter 100 ns liegt. Dies ist auf Compiler-Optimierungen zurückzuführen. Daher wurde 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 dieses Beispiel durch das Datenratenlimit und Speicherlatenz begrenzt ist. Zusätzlich ist die eigentliche Aufgabe trivial für die CPU und lastet diese daher nicht vollständig aus. Aufgrund dieses künstlich erzeugten Szenarios durch die bewusste Beeinflussung des Laufzeitverhaltens mit `volatile` wird auf eine detaillierte Einzelanalyse der Diagramme verzichtet.

2× Geschwindigkeit

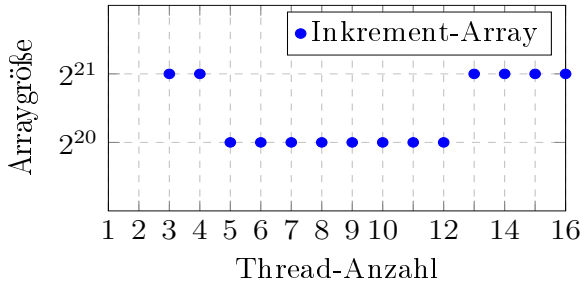

Abbildung 4.1: Zeigt, ab welcher Arraygröße zum ersten Mal die halbe Laufzeit gegenüber der sequentiellen Laufzeit erreicht wird.

Laufzeit bei Arraygröße 16

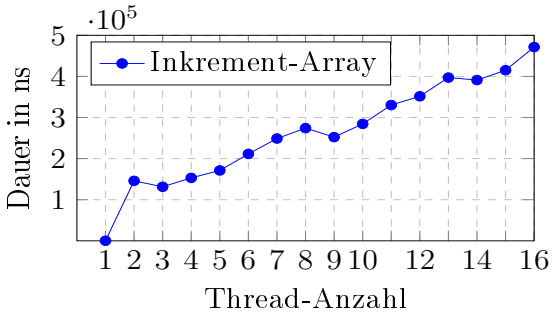

Abbildung 4.2: Zeigt die Overheads, die durch Threads entstehen.

Strong Scaling: Laufzeit bei Arraygröße 2^{21}

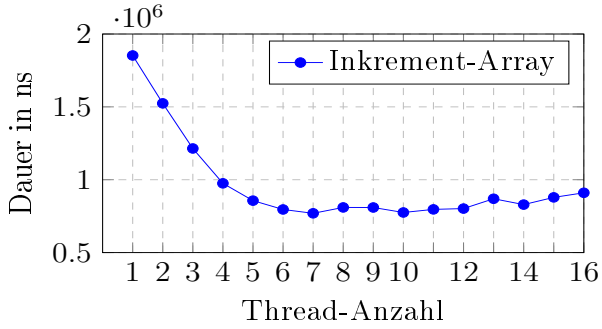

Abbildung 4.3

Strong Scaling: Laufzeit bei Arraygröße 2^{31}

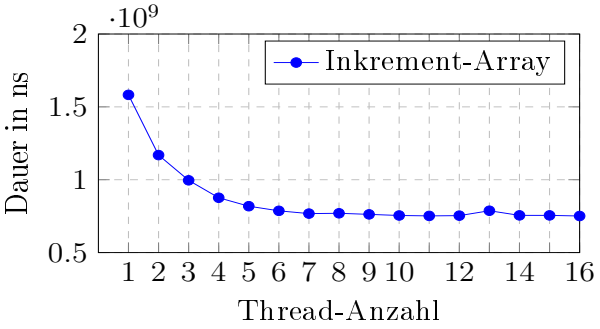

Abbildung 4.4

Weak Scaling (Start-Größe 2^{21})

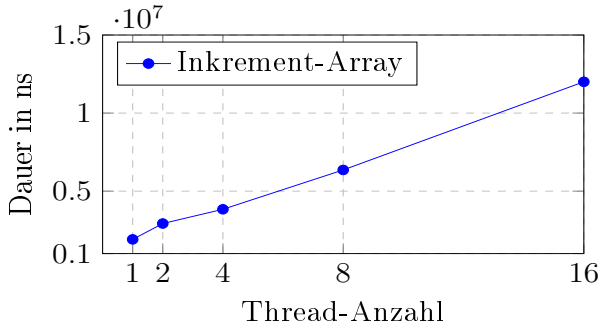

Abbildung 4.5

Arraygröße verdoppeln (16 Threads)

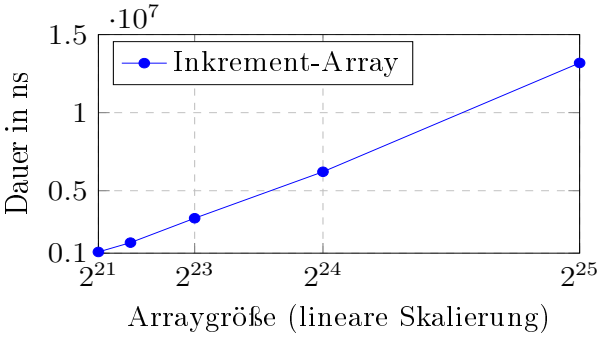

Abbildung 4.6

4.3 Grundlegende Laufzeiten abhängig von der Arraygröße (sequenziell)

4.3.1 Messziel

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

4.3.2 Erwartung

Da die durchschnittliche Laufzeit $O(n \log n)$ beträgt, wird auch diese bei wachsender Arraygröße erwartet.

4.3.3 Diagramme

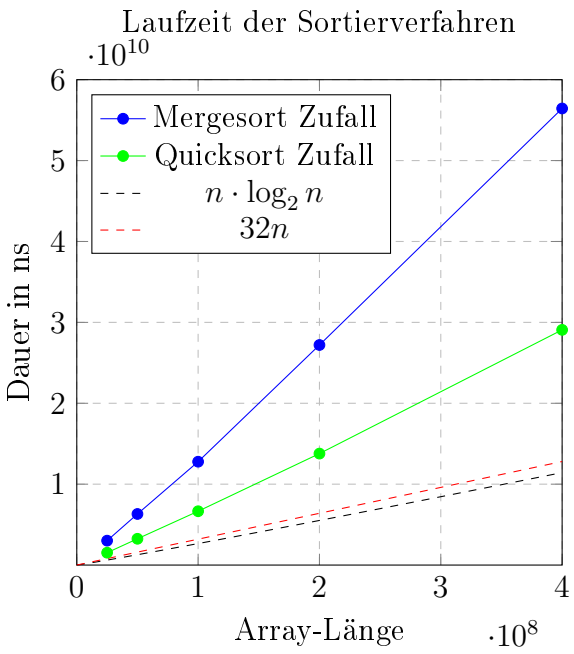

Abbildung 4.7

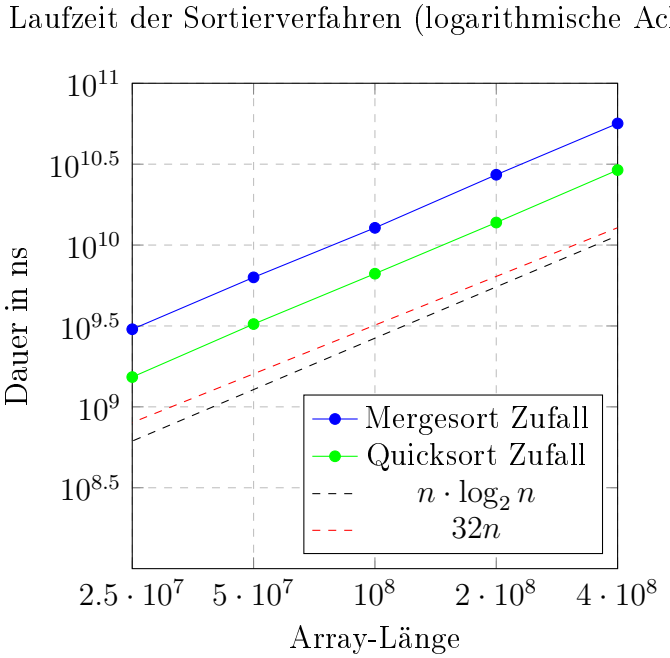

Abbildung 4.8

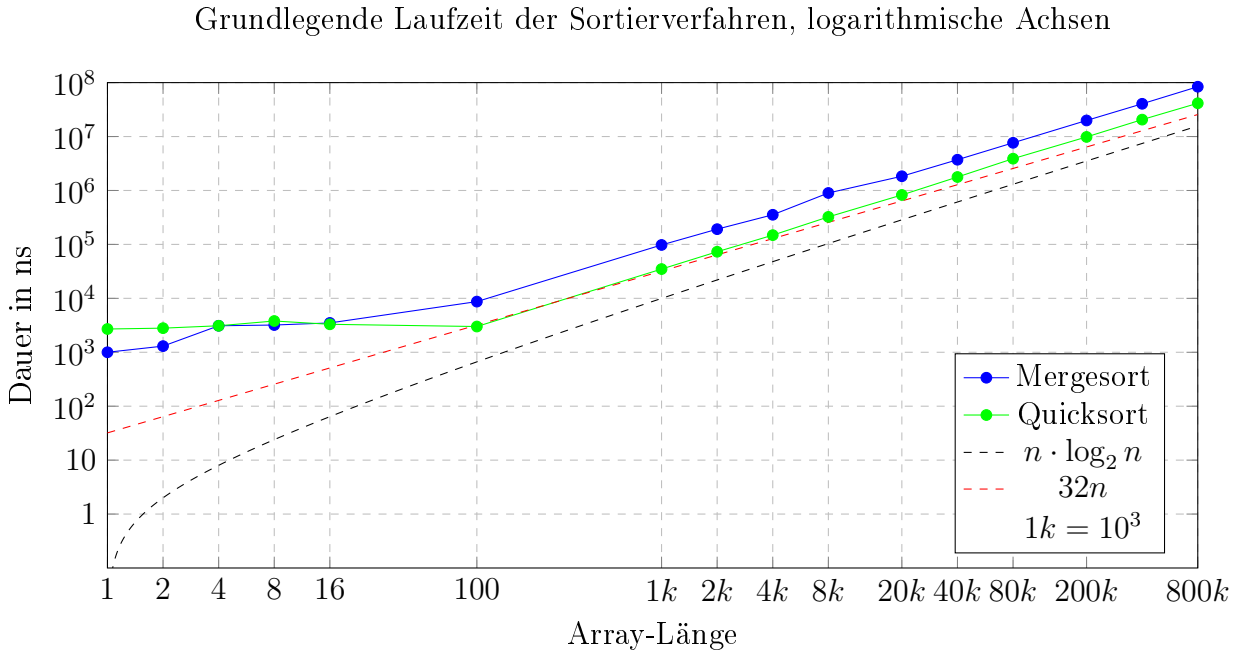

Abbildung 4.9

4.3.4 Analyse und Interpretation

In den ersten zwei Diagrammen (Abbildung 4.7 4.8) ist die Veränderung der Laufzeit zu sehen, wenn die Listengröße fünfmal verdoppelt wird und bei $2.5 \cdot 10^7$ startet. Beim zweiten Diagramm (Abbildung 4.8) sind die Achsen logarithmisch dargestellt, da dies die

Darstellung und den Vergleich der Laufzeiten erleichtert.
Unter diesen beiden Diagrammen befindet sich ein drittes Diagramm (Abbildung 4.9), in dem die gemessenen Laufzeiten ebenfalls logarithmisch dargestellt sind und der Größenbereich von 1 bis 800 000 betrachtet wird.
Anhand dieser Diagramme (Abbildung 4.8 4.9) ist deutlich erkennbar, dass sowohl Mergesort als auch Quicksort tatsächlich eine Laufzeit von $O(n \log n)$ besitzen. Da die Graphen in dem logarithmischen Diagrammen (Abbildung 4.8 4.9) parallel zum eingezeichneten $n \cdot \log_2(n)$-Graphen verlaufen.
Zudem ist erkennbar, dass Mergesort etwa die doppelte Laufzeit von Quicksort benötigt und dass Quicksort näherungsweise ein Zeitverhalten 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 $1 \cdot n \log_2(n)$. Daraus folgt, dass von einer gerundeten durchschnittlichen 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.

4.4 Einfluss des Listentyps (sequenziell)

4.4.1 Messziel

Der Begriff *Listentyp* beschreibt in dieser Arbeit die Anfangsanordnung der Elemente in der zu sortierenden Liste. Untersucht werden konkret die Listentypen *zufällig*, *sortiert*, *invertiert sortiert*, *fast sortiert* sowie *dupliziert*. Ein Beispiel für eine duplizierte Liste ist eine Liste der Größe 100, die aus 50 Einsen und 50 Zweien besteht.
Der Listentyp ist relevant, da Quicksort sowohl einen Best-Case als auch einen Worst-Case aufweist. Diese hängen vom Inhalt der zu sortierenden Liste und somit vom jeweiligen Listentyp ab. Gleichzeitig dient diese Messung der Vollständigkeit, sodass die Laufzeiten auch mit anderen, nicht in dieser Arbeit implementierten Sortieralgorithmen gut vergleichbar sind. Auch hierbei werden zunächst ausschließlich die sequenziellen Laufzeiten der Algorithmen gemessen. Zur Vollständigkeit werden zusätzlich weitere Listentypen außer zufällig und sortiert betrachtet.

4.4.2 Erwartung

Es wird erwartet, dass der Listentyp bei Mergesort nahezu keinen Einfluss auf die Laufzeit hat, sodass lediglich sehr geringe Laufzeitänderungen zu beobachten sind. Für Quicksort wird bei einer sortierten Liste der Best-Case erwartet, da als Pivotelement jeweils das

mittlere Element gewählt wird. Ebenso wird erwartet, dass bei Quicksort bei Wahl des jeweils rechten Elements als Pivotelement bei einer sortierten Liste der Worst-Case eintritt, daher wird der Graph direkt als Worst-Case beschriftet.

4.4.3 Diagramme

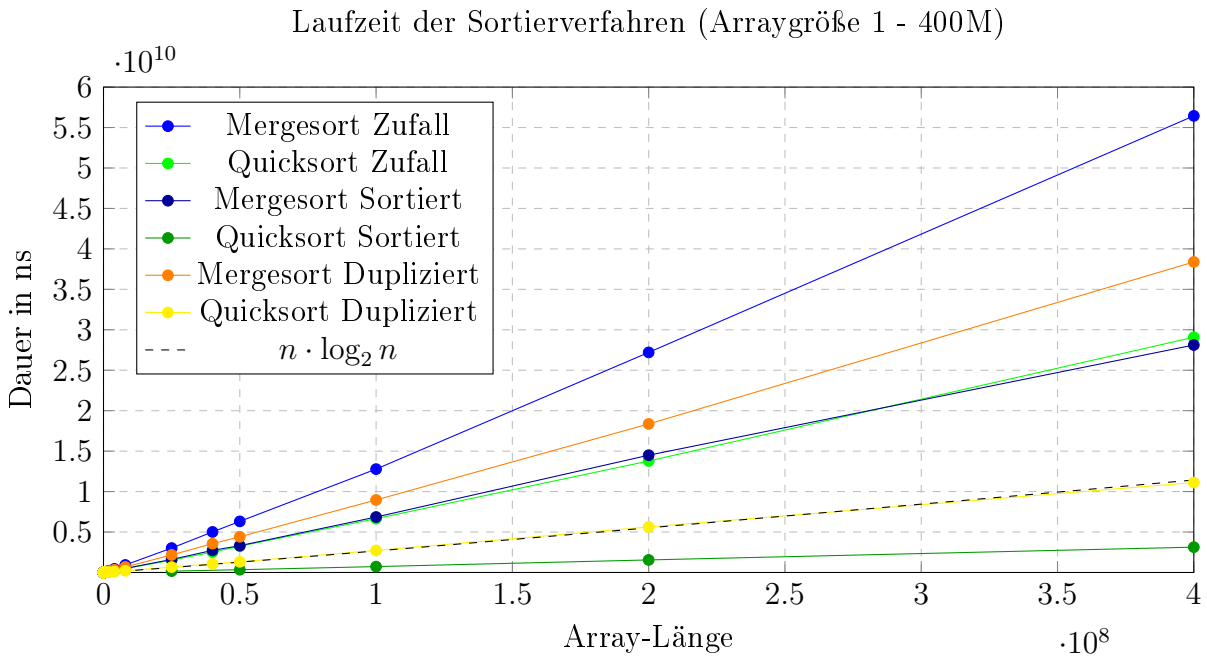

Abbildung 4.10

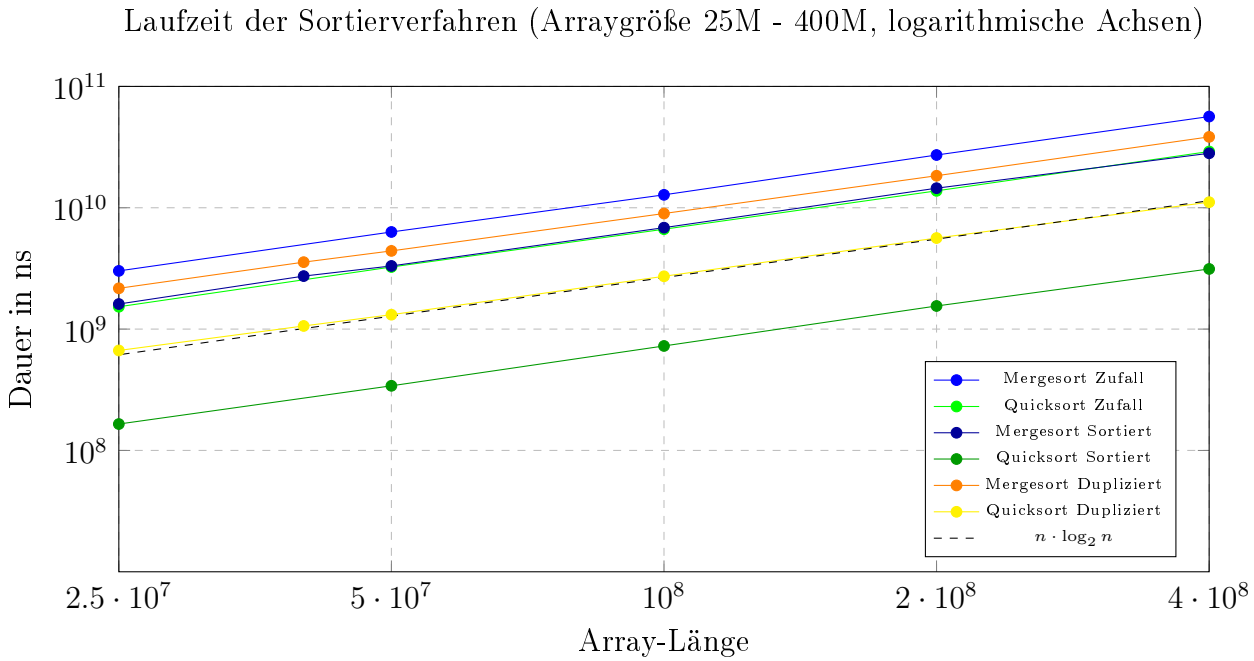

Abbildung 4.11

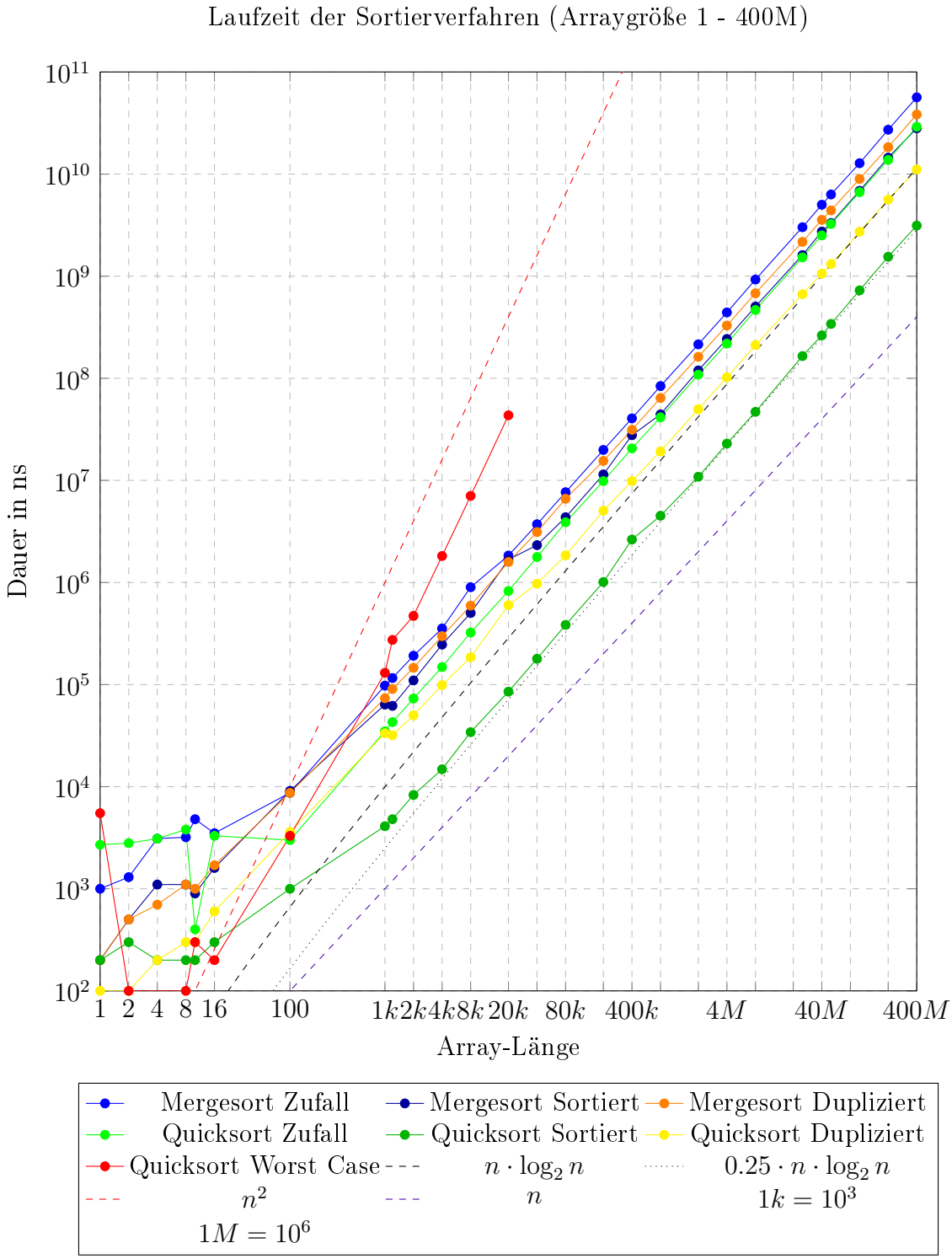

Abbildung 4.12

4.4.4 Analyse und Interpretation

Die gemessenen Daten (Abbildung 4.12) zeigen deutlich, dass die reale Laufzeit im Best-Case unterhalb der Referenzfunktion $n \log_2(n)$ liegt, was auf einen vorteilhaften konstanten Faktor von ca. 0,25 zurückzuführen ist. 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 gemessen, welche oberhalb der berechneten Werte von $1 \cdot n \log_2(n)$ liegt. Die Messungen zeigen außerdem, dass sortierte, invertiert sortierte sowie fast sortierte Arrays nahezu identische Laufzeiten aufweisen. Die entsprechenden Kurven würden nahezu übereinanderliegen, weshalb zugunsten der Übersichtlichkeit auf eine separate grafische Darstellung dieser Listentypen verzichtet wurde. Im Diagramm (Abbildung 4.12) ist ebenfalls zu erkennen, dass der Worst-Case von Quicksort eine Laufzeit von $O(n^2)$ besitzt, da dieser Graph parallel verläuft. Allerdings endet der Graph ab 20k, da bereits bei 40k ein Stackoverflow durch die Rekursionstiefe ausgelöst wird. Dies ließe sich durch eine iterative Version von Quicksort vermeiden.

4.5 Tiefenbasierte Thread-Erzeugung

4.5.1 Messziel

Hier wird die rekursive Variante gemessen, 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.

4.5.2 Erwartung

Es wird der theoretisch Performance-Zuwachs $T(n, p)$ erwartet, solange kein Hardware-Limit erreicht wird. Dafür skalieren wir die $T(n, p)$-Formel um einen Faktor, sodass sie mit der gemessenen Grundlaufzeit übereinstimmt ($x \cdot T(n, 1)$). Der Faktor wird auf eine Stelle nach dem Komma gerundet und ergibt für Quicksort $x = 2,5$ sowie für Mergesort $x = 5$. Zudem wird in dieser Variante erwartet, dass Mergesort besser skaliert als Quicksort, da Quicksort die Liste nicht exakt in der Mitte teilt, sondern dies nur theoretisch im heuristischen Durchschnitt tut. Der Best Case von Quicksort sollte jedoch weiterhin wesentlich besser sein als der von Mergesort, da Quicksort in diesem Fall die Liste immer perfekt in der Mitte teilt. Außerdem wird erwartet, 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.

4.5.3 Diagramme

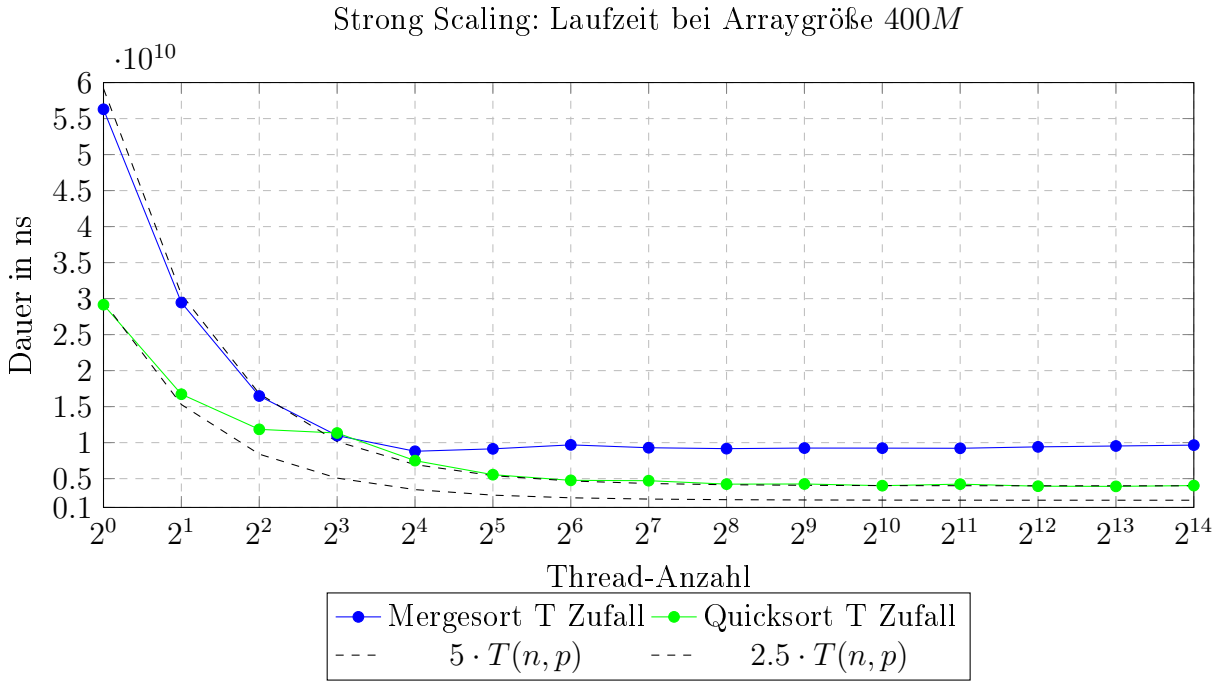

Abbildung 4.13: Strong Scaling (Tiefenbasierte Thread-Erzeugung): Die gestrichelten Linien zeigen die theoretisch zu erwartenden Laufzeitverbesserungen. Quicksort kann diese Verbesserung nur im Best-Case erreichen. T = Tiefenbasierte Thread-Erzeugung

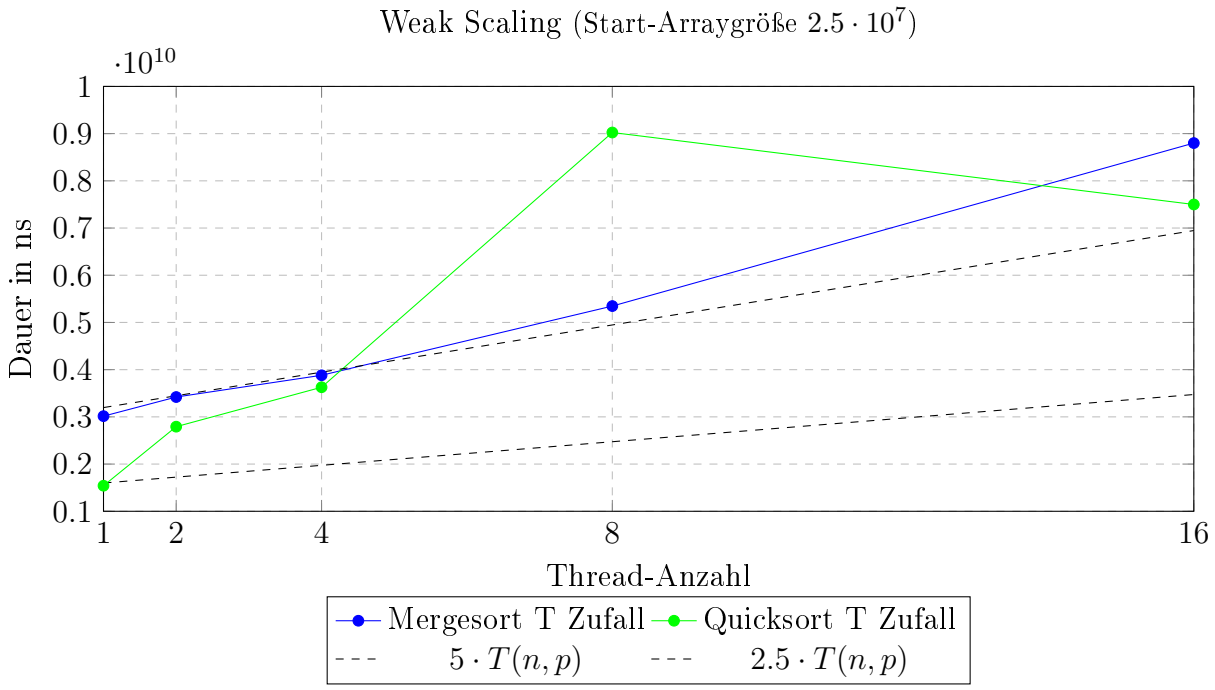

Abbildung 4.14: Weak Scaling (Tiefenbasierte Thread-Erzeugung): Die gestrichelten Linien zeigen die theoretisch zu erwartenden Laufzeitverbesserungen. Quicksort kann diese Verbesserung nur im Best-Case erreichen. T = Tiefenbasierte Thread-Erzeugung

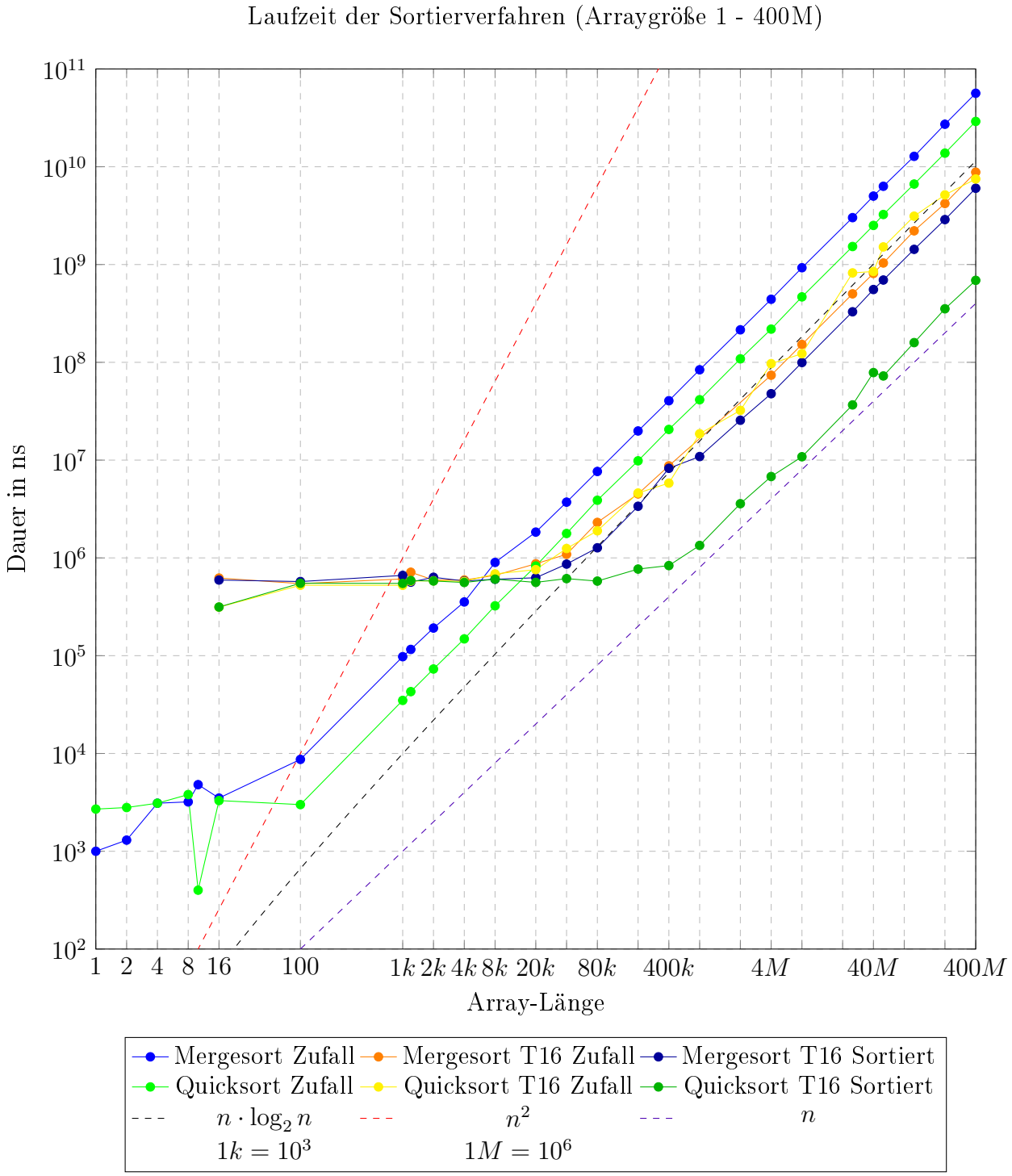

Abbildung 4.15: T16 = Tiefenbasierte Thread-Erzeugung mit 16 Threads

4.5.4 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 sequenziellen Variante benötigt, während Quicksort 26 % erreicht. Die Laufzeit von Mergesort wächst dabei sehr regelmäßig, während Quicksort im Durchschnitt ebenfalls eine regelmäßige Steigerung der Laufzeit zeigt, jedoch sehr 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 , die eine direkte Konsequenz der tiefenbasierten Parallelisierungs-Strategie bei unbalancierten Bäumen ist. 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. Während bei Mergesort zu beobachten ist, 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.

4.6 Workerthreads

4.6.1 Messziel

Hier wird die Worker-Thread-Variante nach einem Work-Stealing-Ansatz mit einer unterstützten Thread-Anzahl von N gemessen. Zusätzlich erfolgt eine Messung, welche die Grund-Overheads der Threads näherungsweise repräsentieren soll (Abbildung 4.16). 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, da Worker-Threads nur einmalig am Anfang erstellt werden würden und somit die Thread-Initialisierungs-Overheads bei jedem Aufruf des Sortierverfahrens wegfallen würden, aber ein gemeinsamer Thread-Pool für verschiedene Algorithmen würde den Rahmen dieser Arbeit sprengen.
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.
Grundlage der Messungen ist die in Abschnitt 3.2 erläuterte Implementierung der Worker-Thread-Variante.

4.6.2 Erwartung

Auch hier wird die $T(n, p)$-Formel zum Vorhersagen der Performanceverbesserung genutz, genau wie bei der tiefenbasierten Thread-Erzeugung.
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 nicht mehr perfekt auf alle Threads verteilt und das Wait nicht wegoptimiert wurde. Dazu wird für die Mergesort Worker-Thread-Variante erwartet das es eine zu eine laufzet verbeung komt bis 32 thads erzugt wreden. Dies liegt an dem Wait und wird nur durch addTaskSmart kompensiert, indem die Funktion dafür sorgt, dass immer mindestens die Hälfte aller Threads ausgelastet bleibt.
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.

4.6.3 Diagramme

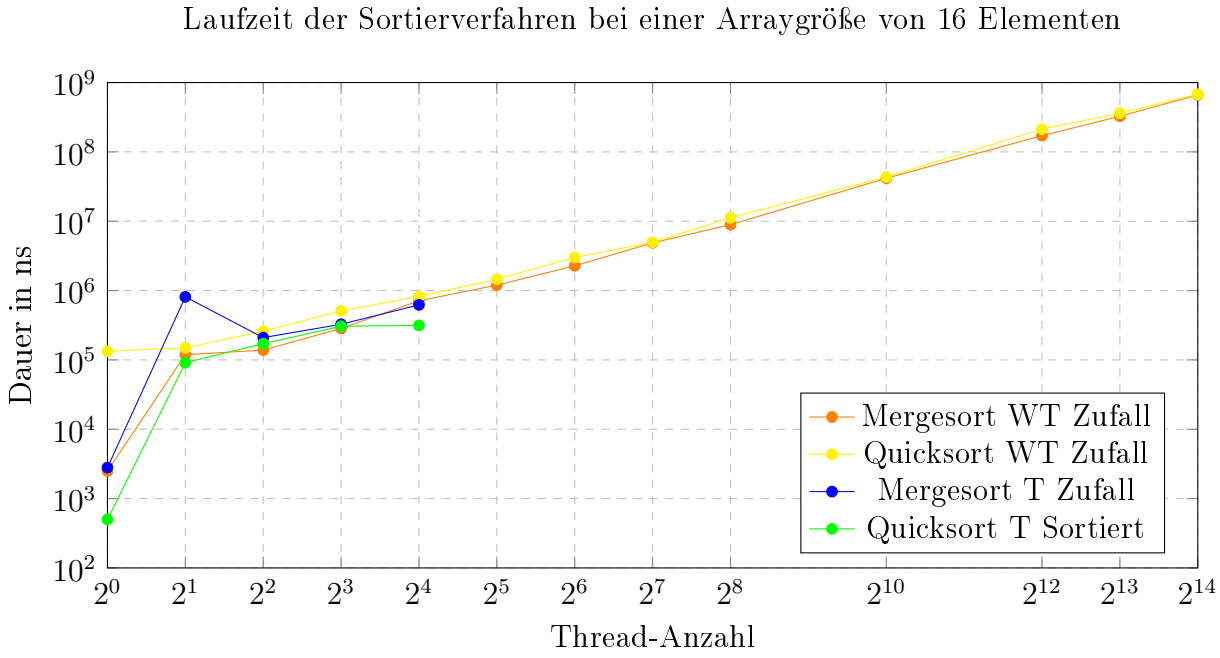

Abbildung 4.16: Dieses Diagramm dient der Abschätzung der durch Threads entstehenden Overheads. T = Tiefenbasierte Thread-Erzeugung, WT = Worker-Thread-Variante

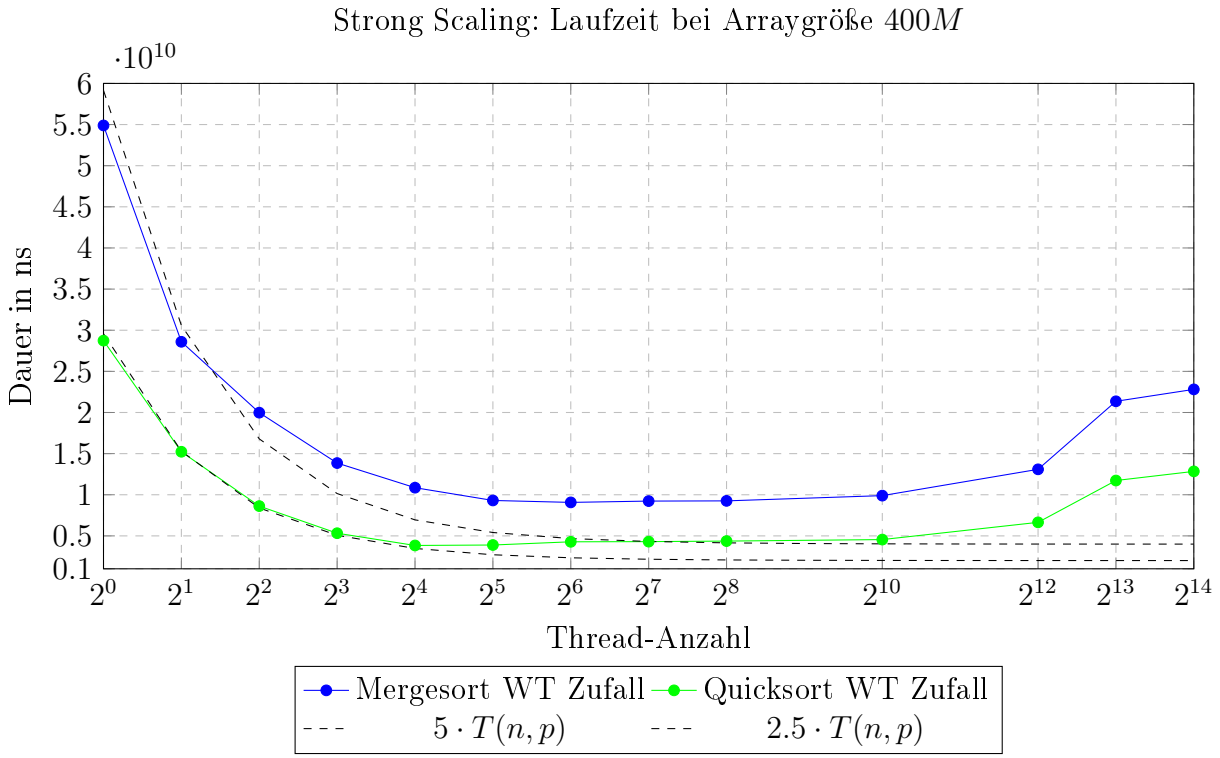

Abbildung 4.17: Strong Scaling (Worker-Thread): Die gestrichelten Linien zeigen die theoretisch zu erwartenden Laufzeitverbesserungen. WT = Worker-Thread-Variante

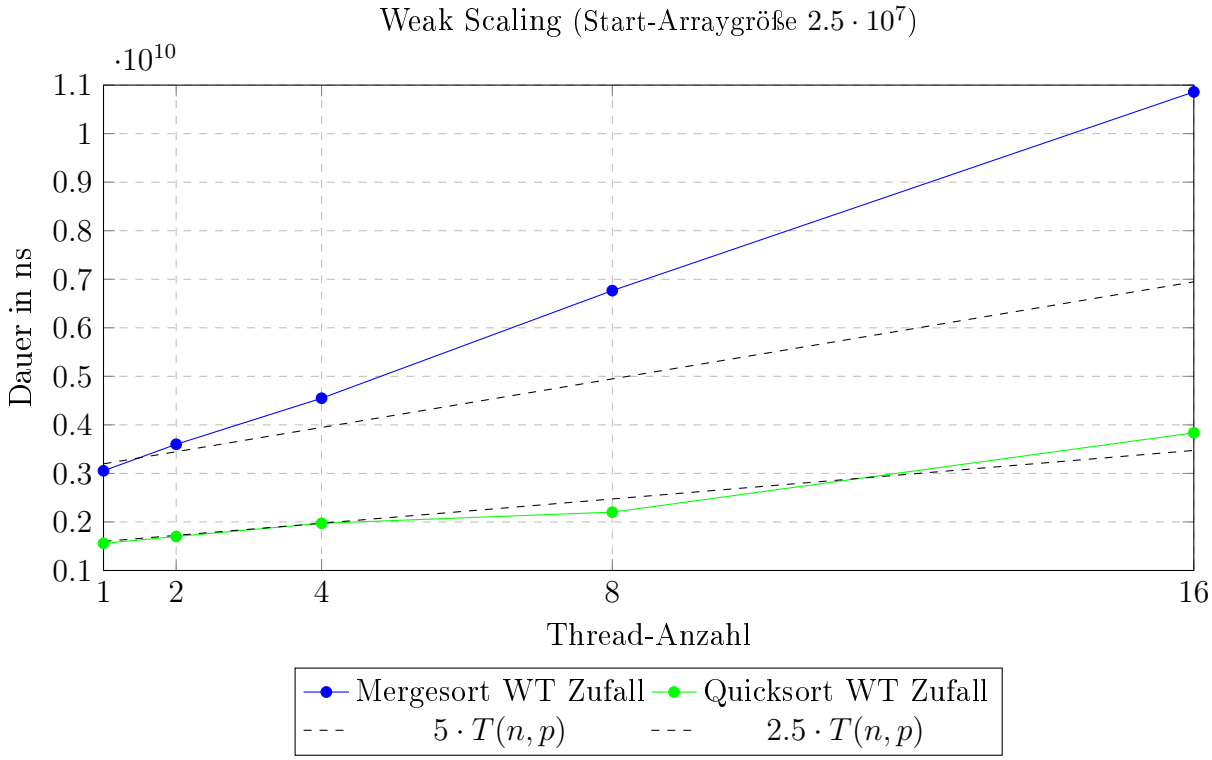

Abbildung 4.18: Weak Scaling (Worker-Thread): Die gestrichelten Linien zeigen die theoretisch zu erwartenden Laufzeitverbesserungen. WT = Worker-Thread-Variante

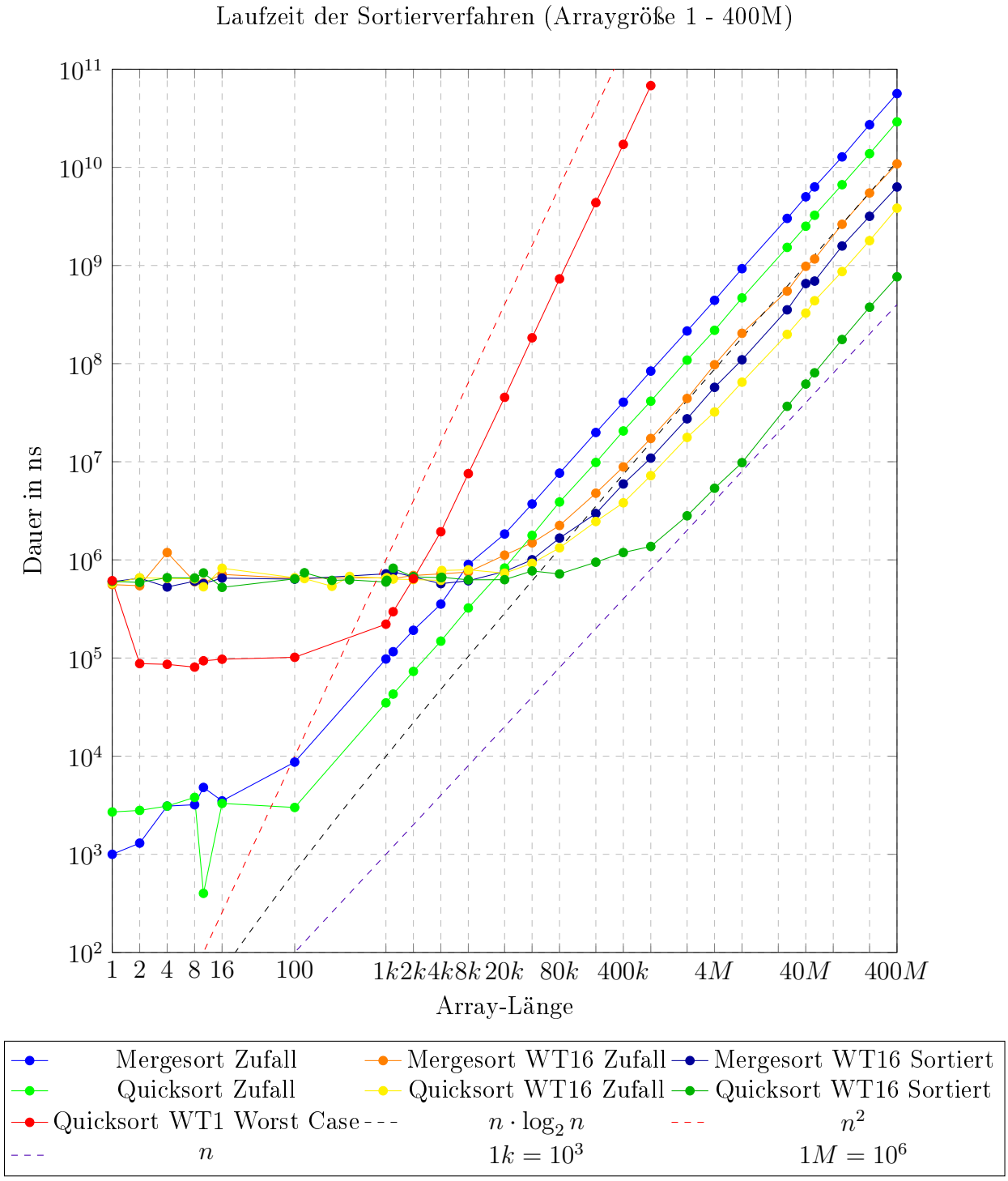

Abbildung 4.19: WT16 = Worker-Thread-Variante mit 16 Worker-Threads

4.6.4 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 (Abbildung 4.17) und Weak Scaling (Abbildung 4.18), 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. Beim Strong Scaling ist außerdem zu beobachten, dass die Performance bei sehr vielen Threads wieder abnimmt. Ursache hierfür sind die Sperren zur Thread-Arbeitsverteilung, die in dieser Variante erforderlich für die gemeinsame Aufgaben-Queue sind. Bei sehr vielen Threads führt dies zu einem Performance-Einbruch, da jeweils nur ein Thread die Sperre belegen kann. Hinzu kommen die Kosten der reinen Initialisierungszeit, da 2^{14} Threads rund 1 s für ihre Erstellung benötigen. Im Diagramm 4.16 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 als die anderen Varianten erstellen, was zu diesem minimalen Overhead führt.

4.7 Einfluss des Datentyps der Liste

4.7.1 Messziel

Hier wird die Zeit gemessen, die zum Sortieren von Strings benötigt wird, da dies auch ein realistischer Anwendungsfall ist. Dafür werden direkt die Strategien mit der besten Laufzeit miteinander verglichen.

4.7.2 Erwartung

Auch hier wird die $T(n, p)$-Formel zur Vorhersage der Performanceverbesserung genutzt, genau wie bei den Messungen zuvor. Nur ist der Faktor jetzt für Quicksort $x = 13$ und für Mergesort $x = 26$.
Es wird erwartet, dass das Sortieren länger dauert als bei `int`, da ein String wesentlich aufwendiger zu vergleichen ist. Dies liegt daran, dass er aus mehreren Zeichen (`char`) besteht und entsprechend mehr Speicher benötigt. Folglich sollte der durch die Threads entstehende Overhead einen geringeren Einfluss auf die gemessene Endzeit haben. Dies sollte dazu führen, dass die Graphen im Strong Scaling und Weak Scaling weniger von den theoretisch erwarteten Laufzeitverbesserungen abweichen.

4.7.3 Diagramme

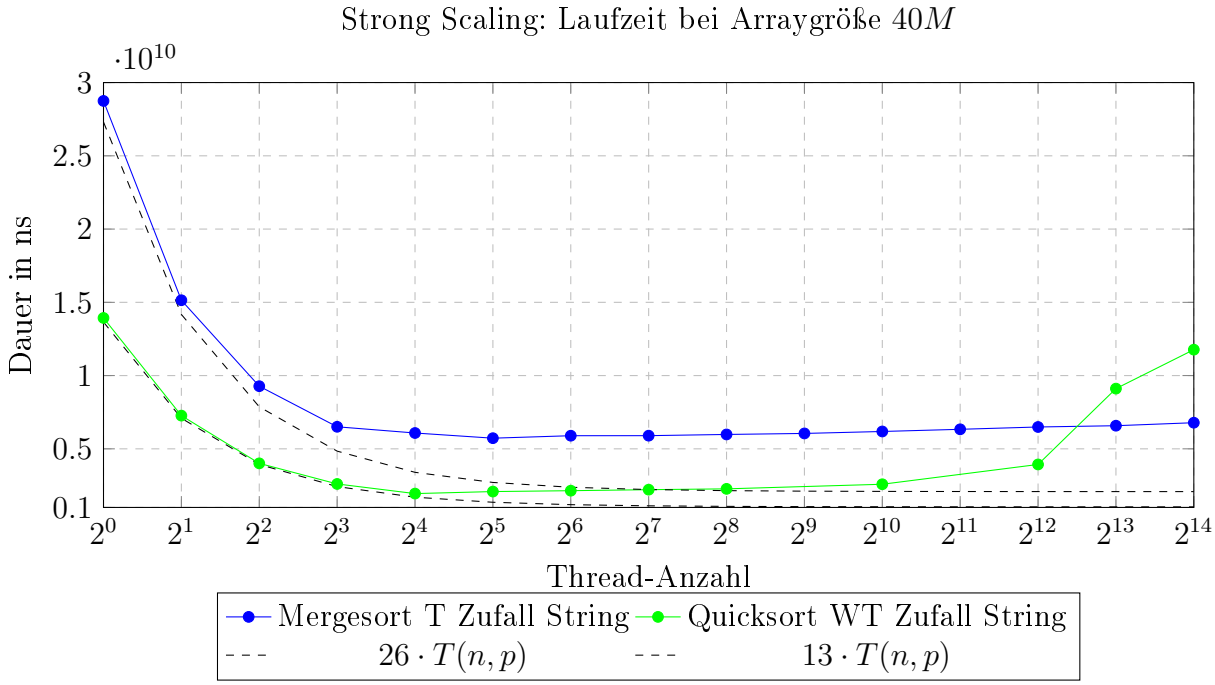

Abbildung 4.20: Strong Scaling (String): Die gestrichelten Linien zeigen die theoretisch zu erwartenden Laufzeitverbesserungen. T = Tiefenbasierte Thread-Erzeugung, WT = Worker-Thread-Variante

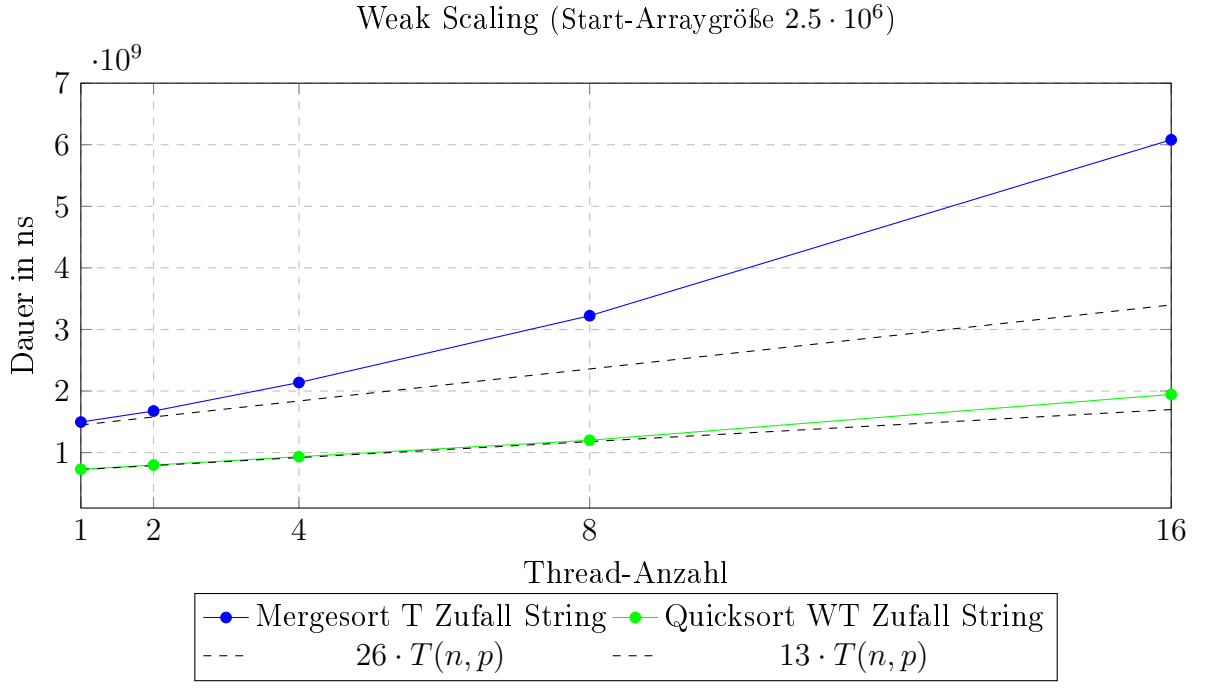

Abbildung 4.21: Weak Scaling (String): Die gestrichelten Linien zeigen die theoretisch zu erwartenden Laufzeitverbesserungen. T = Tiefenbasierte Thread-Erzeugung, WT = Worker-Thread-Variante

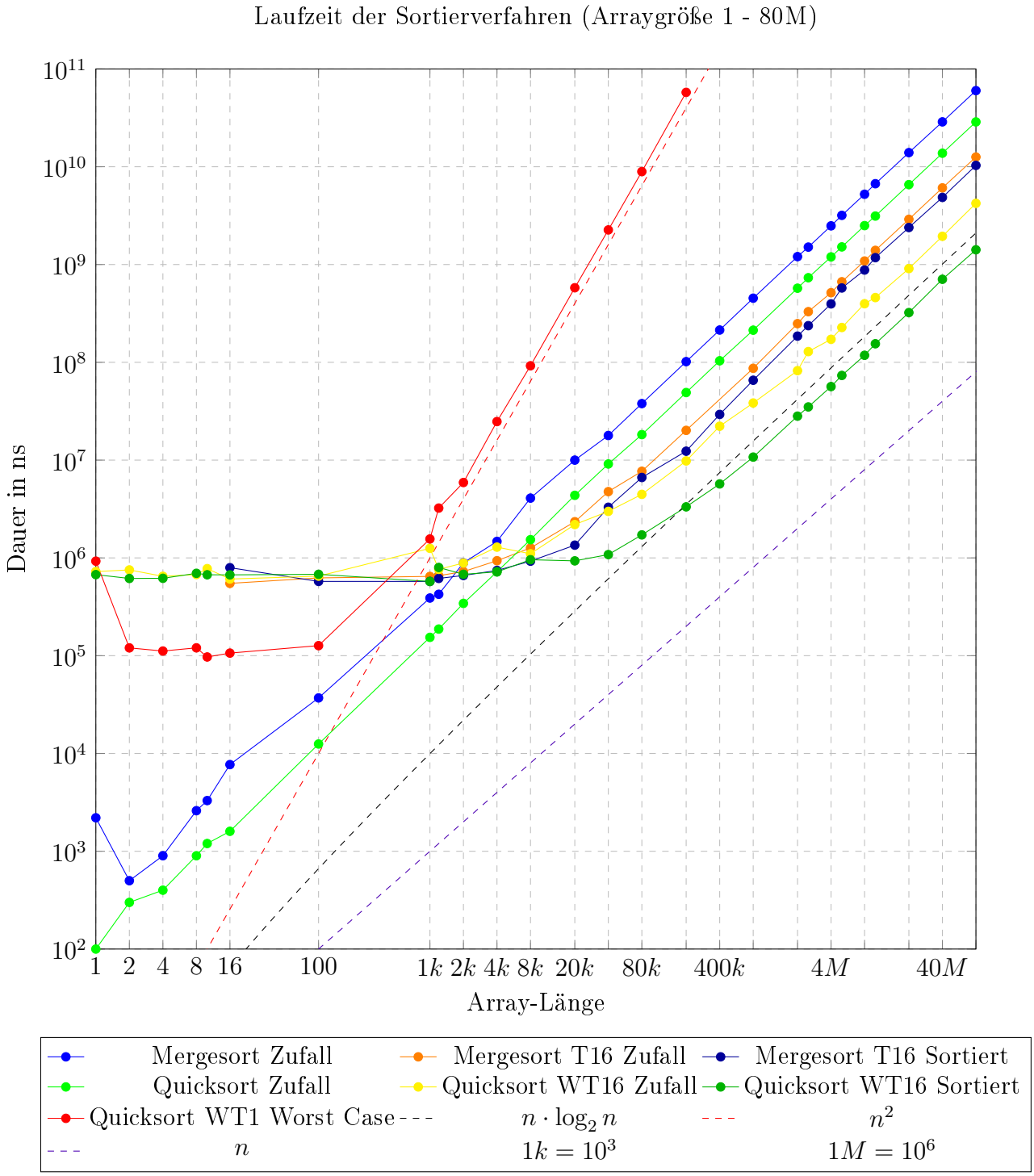

Abbildung 4.22: T16 = Tiefenbasierte Thread-Erzeugung mit 16 Threads, WT16 = Worker-Thread-Variante mit 16 Worker-Threads

4.7.4 Analyse und Interpretation

Bei den Graphen zu Strong Scaling (Abbildung 4.20) und Weak Scaling (Abbildung 4.21) 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. Vereinfacht ausgedrückt liegt dies am Code zur Listengenerierung, der für sortierte Listen im Durchschnitt längere Strings erzeugt als für zufällige Listen. Dies erhöht den Vergleichsaufwand und wirkt sich negativ auf die Laufzeit aus.

5 Diskussion und Fazit

5.1 Interpretation aller Ergebnisse

Es ist eindeutig zu erkennen, dass Mergesort in der tiefenbasierten Thread-Erzeugungsvariante die beste Performance erzielt und dass Quicksort in der Worker-Thread-Variante mit Work-Stealing-Ansatz die beste Laufzeit erreicht, wie zuvor erwartet. Ebenso ist deutlich zu erkennen, dass diese Varianten nur geringfügig schlechtere Laufzeiten erreichen, als theoretisch möglich wäre. Die Ergebnisse zeigen jedoch, dass die Abweichung von der theoretisch möglichen Laufzeit geringer ausfällt als aufgrund der anfangs genannten Overheads zu erwarten gewesen wäre. Vergleicht man die entstehenden Overheads durch die Thread-Erzeugung (Abbildung 4.16) mit den sequenziellen Laufzeiten (Abbildung 4.12), ergibt sich ein guter Eindruck davon, ab welcher Problemgröße Parallelisierung überhaupt einen Vorteil bietet. Um beispielsweise 16 Threads effizient nutzen zu können, sollte eine Mindestlaufzeit von etwa 1 ms erreicht werden. Dies ist bei Quicksort der Fall, wenn ein `int`-Array eine Mindestgröße von 20k aufweist. Zusammenfassend lässt sich festhalten, dass Mergesort insbesondere dann die bessere Wahl ist, wenn die *Worst-Case*-Laufzeit von Quicksort nicht tolerierbar ist. Quicksort zeigt jedoch für typische, nicht gezielt konstruierte Listen im Durchschnitt eine bessere Laufzeit, wobei die *Best-* und *Worst-Case*-Laufzeiten stets berücksichtigt werden sollten. Vergleicht man die jeweils beste Variante beider Algorithmen, so zeigt sich, dass Quicksort nach der Parallelisierung im Durchschnittsfall weiterhin etwa doppelt so schnell ist wie Mergesort.

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.
Ebenso lässt sich die Worst-Case-Laufzeit von Quicksort unwahrscheinlicher machen. Ein möglicher Ansatz besteht darin, das Pivot-Element zufällig zu bestimmen. Darüber hinaus können mehrere Elemente zufällig ausgewählt sowie die Listenränder und die Listenmitte in die Pivot-Auswahl einbezogen werden, aus denen anschließend der Median gebildet wird. Die Anzahl der zufällig hinzugewählten Elemente kann dabei abhängig von der Listengröße gewählt werden. Diese Methoden führen dazu, dass die Worst-Case-Laufzeit nur noch eine Frage der Wahrscheinlichkeit ist. Da diese Wahrscheinlichkeit mit zunehmender Anzahl zufällig bestimmter Pivot-Elemente weiter sinkt, ist es in der Praxis extrem unwahrscheinlich, dass der Worst-Case eintritt.

. Das perfekte Pivotelement ist der Median der zu partitionierenden Liste. Allerdings kann der Median nur bestimmt werden, indem die Liste vorher sortiert wird, weswegen es sinnlos ist, ihn perfekt zu bestimmen.
Zudem existieren Varianten, bei denen sowohl der Partitionierungs- als auch der Merge-Schritt parallelisiert sind. Diese Ansätze lassen sich mit rekursiv parallelisierten Varianten kombinieren, um eine möglichst gleichmäßige Auslastung aller Threads zu erreichen, was zu weiteren Laufzeitverbesserungen führen kann.
Weiterhin ist anzumerken, dass moderne CPUs häufig auf einem Multi-Chiplet-Design basieren. Bei Intel betrifft dies CPUs mit einer Kombination aus Performance- und Effizienz-Kernen. Diese Kerne unterscheiden sich in ihrer Taktrate und Leistungsfähigkeit, was eine perfekte Parallelisierung zusätzlich erschwert. Bei AMD besteht zum Beispiel der 7950X3D aus einem 7800X3D und einem 7800X. Diese beiden Chips unterscheiden sich in der Cache-Größe, in der Cache-Latenz und in der Taktrate. Der 7800X3D ist zudem aufgrund der Architektur des 3D-Caches thermisch stärker limitiert, da sich der Cache direkt auf den Kernen befindet. Dies führt auch bei AMD dazu, dass sich CPUs wie der 7950X3D nur sehr schwer perfekt parallelisieren lassen. Aus diesen Gründen sowie aufgrund der anfangs genannten Overheads erscheint eine perfekte Parallelisierung moderner CPUs nicht erreichbar. Stattdessen kann sich dem idealen Szenario lediglich angenähert werden.

5.3 Beantwortung der Forschungsfrage

Von den implementierten Varianten besitzt die Quicksort-Worker-Stealing-Variante die beste Laufzeit, unter der Bedingung, dass das zu sortierende `int`-Array mindestens 20k bzw. das `String`-Array mindestens 8k groß ist. Erst wenn die sehr unwahrscheinliche *Worst-Case*-Laufzeit untolerierbar ist, stellt die tiefenbasierte Thread-Erzeugung von Mergesort die bessere Wahl dar. Diese bietet jedoch erst eine bessere Laufzeit, wenn das zu sortierende `int`-Array mindestens 8k bzw. das `String`-Array mindestens 4k groß ist. Ist das zu sortierende Array kleiner, ist die sequenzielle Variante zu bevorzugen, da diese in diesem Bereich schneller ist. Dies ist auf die anfangs genannten Overheads zurückzuführen. Dabei ist zu beachten, dass sich diese Angaben zur verbesserten parallelen Laufzeit bei beiden Algorithmen auf die jeweilige 16-Thread-Variante beziehen.

5.4 Zusammenfassung

Zusammenfassend lässt sich festhalten, dass die Quicksort-Worker-Stealing-Variante der tiefenbasierten Thread-Erzeugungsvariante von Mergesort in der Regel überlegen ist. Allerdings ist darauf zu achten, erst dann von der sequenziellen Version auf die Parallelisierung umzuschalten, wenn die entsprechende Mindestlaufzeit bzw. Mindest-Array-Größe erreicht ist. Zudem zeigen die Ergebnisse, dass die theoretisch mögliche Laufzeitverbesserung nahezu erreicht wird. Weiterhin hat sich herausgestellt, dass Compiler-Optimierungen einen erheblichen Einfluss auf die Laufzeit haben, insbesondere im Zusammenhang mit Threading, auch wenn dieser Aspekt in dieser Arbeit nicht explizit unter-

sucht wurde. Darüber hinaus besteht weiterhin Verbesserungspotenzial, wie im Ausblick dargestellt.

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
Im Folgenden sind die relevanten Hardwarekomponenten aufgeführt.
CPU: AMD Ryzen 7 5800X, 8C/16T, 3.80–4.70 GHz
CPU-Kühler: be quiet! Dark Rock Pro 4
RAM: G.Skill Aegis UDIMM 16GB Kit, DDR4-3200, CL16-18-18-38 (2 Kits, insgesamt $4 \times 8\,\text{GB} = 32\,\text{GB}$)
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] Tobias Weinzierl. *Skalierungsmodelle und Kallibrieren von Messungen*, pages 191–201. Springer International Publishing, Cham, 2024. `doi:10.1007/978-3-031-49082-8_15`.

Abbildungsverzeichnis

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Graduierungsarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle den benutzten Quellen wörtlich oder sinngemäß entnommene Stellen sind als solche einzeln kenntlich gemacht.

Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Erklärung zur Verwendung generativer KI:
Die vorliegende Arbeit wurde von mir eigenständig konzipiert und im Entwurf schriftlich ausgearbeitet. Die KI-Modelle **ChatGPT (OpenAI)** und **Gemini (Google)** wurden ausschließlich als unterstützende Werkzeuge zur sprachlichen Überarbeitung (Korrektur von Rechtschreibung und Grammatik), zur Stiloptimierung sowie zur Unterstützung bei der Implementierung einzelner Code-Fragmente eingesetzt. Die fachliche Prüfung aller sprachlichen Anpassungen, die Durchführung der Messreihen sowie die gesamte inhaltliche Ausarbeitung erfolgten vollständig durch mich selbst.

Leipzig, 2. Februar 2026 Unterschrift