Untersuchung der Skalierbarkeit von parallelem Sortieren auf einem Multicore-Prozessor

Verteidigung der Bachelorarbeit

Leon Zoerner

Informatik

20. Februar 2026

Agenda

1. Einleitung
2. Theoretische Grundlagen
3. Methodik und Versuchsaufbau
4. Ergebnisse und Analyse
5. Diskussion und Fazit
6. Binärbaum
7. Formeln
8. Herleitung T(n,p)

Einleitung: Motivation

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.

Einleitung: Zielsetzung und Forschungsfrage

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?**

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.

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.

Sortieralgorithmen: 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 (Mischen)*. 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.

Sortieralgorithmen: Mergesort

```
1  void Mergesort::mergesort(int *liste, const int links,
       const int rechts) {
2      int laenge = rechts - links + 1;
3      if (laenge > 1) {
4          int mitte = links + ((rechts - links) / 2);
5          mergesort(liste, links, mitte);        // A
6          mergesort(liste, mitte + 1, rechts); // B
7          mischen(liste, links, mitte, rechts, laenge);
8      }
9  }
```

Sortieralgorithmen: Mergesort

```
1   void Mergesort::mischen(int *liste, int links, const int
        mitte, const int rechts, const int lange) {
2       int *listeB = new int[lange];
3
4       // Kopiere nach listeB
5       for (int i = links; i < mitte + 1; i++) {
6           listeB[i - links] = liste[i];
7       }
8       for (int i = mitte + 1; i < rechts + 1; i++) {
9           listeB[lange - 1 + mitte + 1 - i] = liste[i];
10      }
11
12      // Sortiere liste
13      int i = 0;              // links
14      int j = lange - 1; // rechts
15      int k = links;          // links
16      while (i < j) {
17          if (listeB[i] < listeB[j]) {
18              liste[k] = listeB[i];
19              i++;
20          } else {
```

Sortieralgorithmen: Mergesort

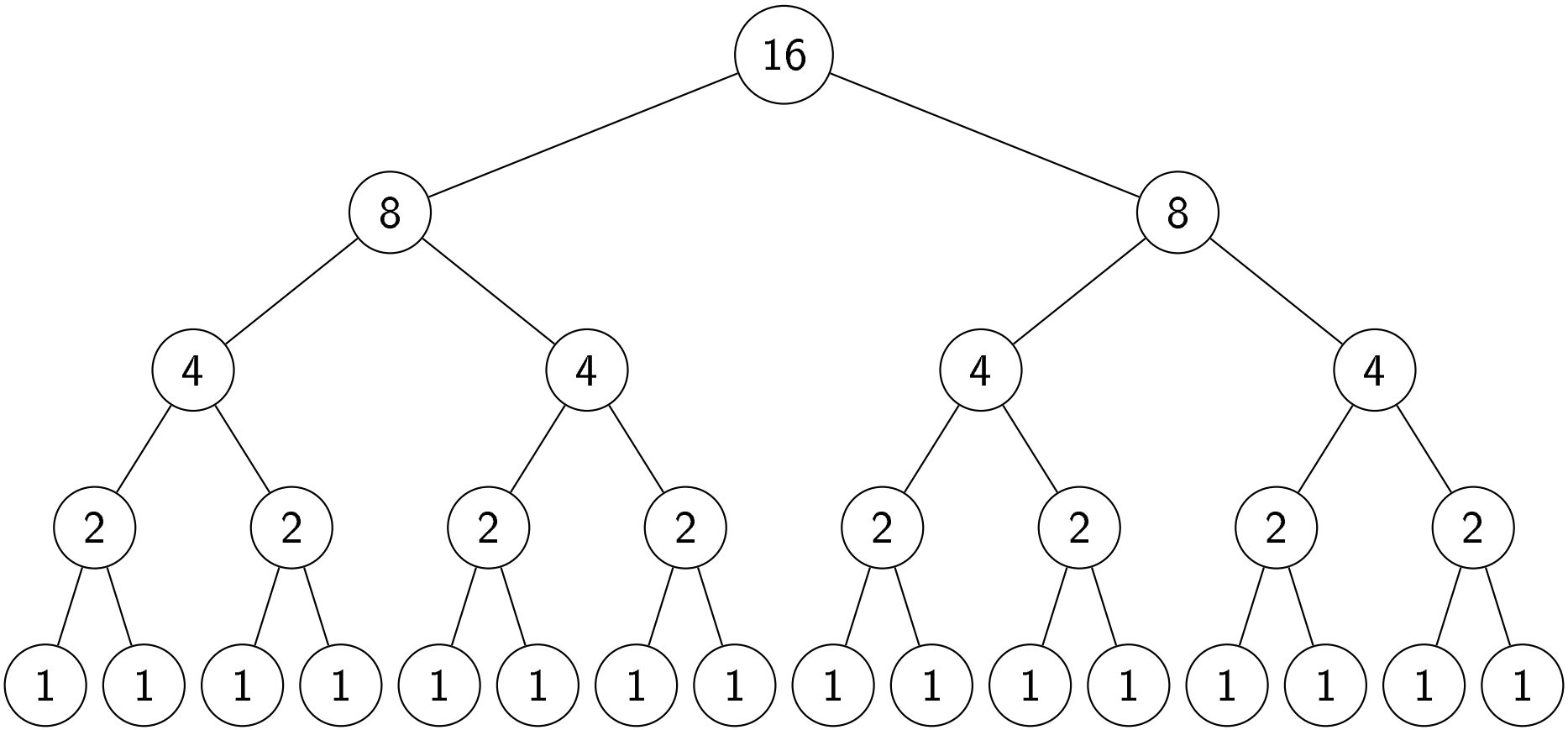
```
1         ...
2         // Sortiere liste
3         int i = 0;           // links
4         int j = lange - 1; // rechts
5         int k = links;       // links
6         while (i < j) {
7             if (listeB[i] < listeB[j]) {
8                 liste[k] = listeB[i];
9                 i++;
10            } else {
11                liste[k] = listeB[j];
12                j--;
13            }
14            k++;
15        }
16        liste[rechts] = listeB[i];
17        delete[] listeB;
18    };
```

Sortieralgorithmen: Mergesort

$$
\begin{aligned}
T(n) &= m_1 + m_2 + n \\
T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \\
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n, \\
T(n) &= n \cdot \log_2(n) + n, \\
O(T(n)) &= O(n \log n).
\end{aligned}
$$

Sortieralgorithmen: Mergesort

Balancierter Binärbaum für $n = 16$



Sortieralgorithmen: 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.

Sortieralgorithmen: Quicksort

```
1  void Quicksort::quicksort(int *liste, const int links,
       const int rechts) {
2      if (links < rechts) {
3          int ml, mr;
4          partitioniere(liste, links, rechts, ml, mr);
5          quicksort(liste, links, ml);
6          quicksort(liste, mr, rechts);
7      }
8  };
```

Sortieralgorithmen: Quicksort

```
1   void Quicksort::partitioniere(int *liste, const int
        links, const int rechts, int &ml, int &mr) {
2       int i = links;
3       int j = rechts;
4       int mitte = links + ((rechts - links) / 2);
5       int p = liste[mitte];
6       while (i <= j) {
7           while (liste[i] < p) {
8               i++;
9           }
10          while (liste[j] > p) {
11              j--;
12          }
13          if (i <= j) {
14              vertausche(liste, i, j);
15              i++;
16              j--;
17          }
18      };
19      ml = j; mr = i;
20  };
```

Sortieralgorithmen: Quicksort

```
1  void Quicksort::vertausche(int *liste, const int a,
       const int b) {
2      int temp = liste[a];
3      liste[a] = liste[b];
4      liste[b] = temp;
5  };
```

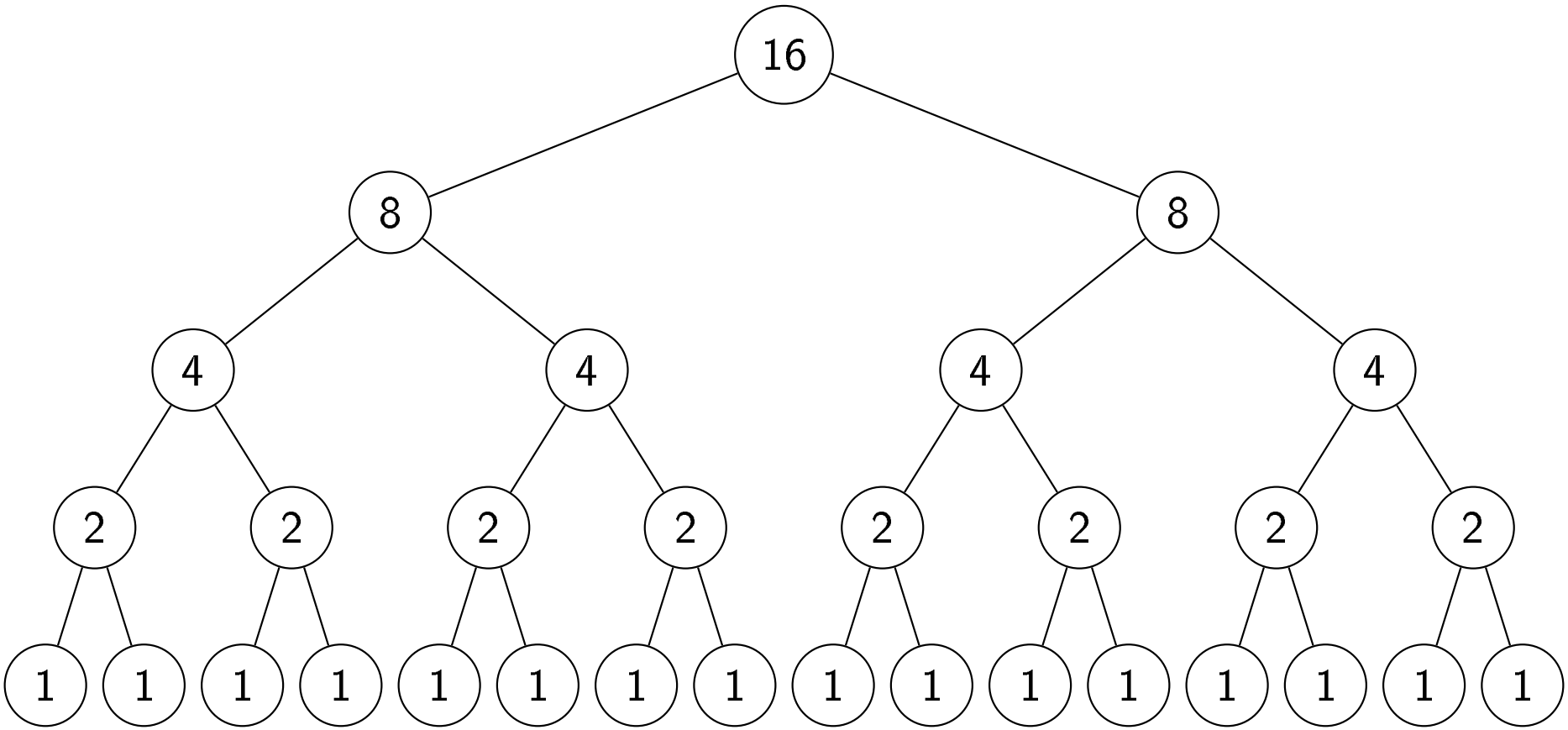
Sortieralgorithmen: Quicksort Best-Case

Best-Case von Quicksort:

$$
\begin{aligned}
T(n) &= q_1 + q_2 + n \\
T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \\
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n, \\
T(n) &= n \cdot \log_2(n) + n, \\
O(T(n)) &= O(n \log n).
\end{aligned}
$$

Sortieralgorithmen: Quicksort Best-Case

Balancierter Binärbaum für $n = 16$



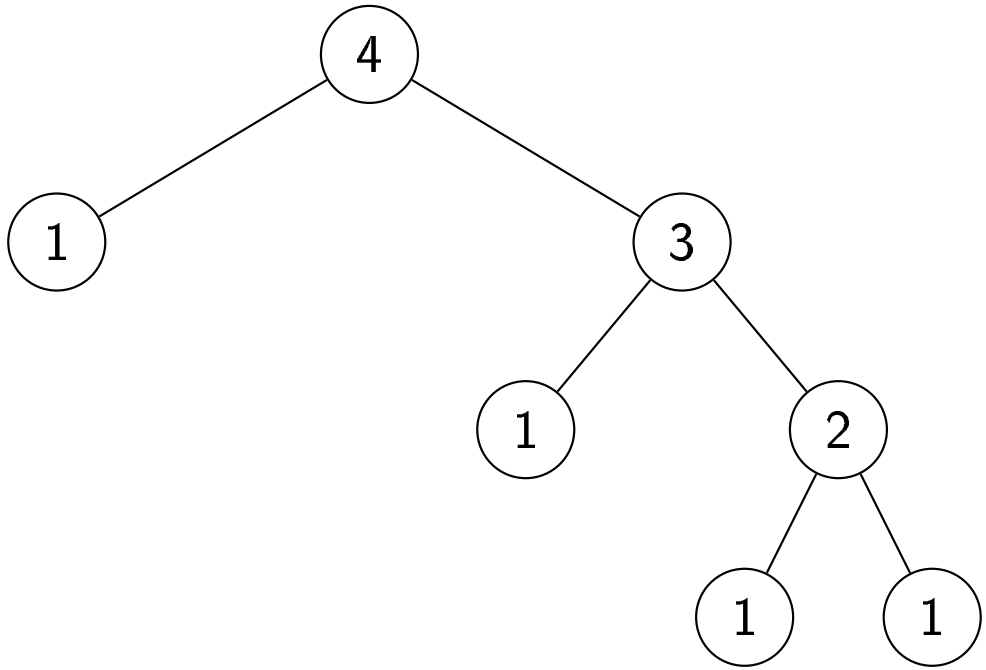
Sortieralgorithmen: Quicksort

Der **Worst-Case** von Quicksort ist:

$$
\begin{aligned}
T(n) &= q_1 + q_2 + n \\
T(n) &= T(n-1) + 1 + n, \\
T(n) &= \frac{1}{2} \cdot (n^2 + n) + n, \\
O(T(n)) &= O(n^2).
\end{aligned}
$$

Sortieralgorithmen: Quicksort

Maximal unbalancierter Binärbaum für $n = 4$ **Worst-Case**:



Sortieralgorithmen: Quicksort Average-Case

- Nicht zielführend, da eine einzige Laufzeit nie die Laufzeit des echten Zufalls abbilden kann.
- Die echte Laufzeit wird fast nie Durchschnitt oder Median sein.
- Man sollte generell eher vom gesamten Bereich, der vom Best-Case bis hin zum Worst-Case reicht, ausgehen.
- Aus Sicht des Programmablaufs (unbalancierte Baumstruktur) liegt man im Zufall eher am Best Case als am Worst Case.

Sortieralgorithmen: Quicksort Average-Case

Der **heuristisch betrachtete Average-Case** von Quicksort ist:

$$T(n) = q_1 + q_2 + n$$

$$q_1 = T\left(\frac{1 + \cdots + (n-1)}{n-1}\right) = T\left(n \cdot \frac{n-1}{2} \cdot \frac{1}{n-1}\right) = T\left(\frac{n}{2}\right) = q_2$$

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

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

$$O(T(n)) = O(n \log n).$$

- Mathematisch falsche Herangehensweise. Dadurch wird selbst der Worst-Case zu $T(n/2)$.
- Soll mathematisch nur zeigen, dass man eher am Best-Case als am Worst-Case liegt.
- Jedem sollte klar sein, dass man nur im Best-Case auch die Laufzeit des Best-Case erreichen kann und somit der echte Average-Case nie die Laufzeit des Best-Case haben kann.
- Der echte Average Case wird über den Durchschnitt von $T(*)$ berechnet und nicht über die möglichen Werte von n, die $T(*)$ annehmen kann.

Sortieralgorithmen: Quicksort Average-Case

$$M_{\text{mit}}(n) = \frac{1}{n}\sum_{k=0}^{n-1}\frac{k(n-k)}{n} = \frac{1}{n^2}\sum_{k=1}^{n}(kn-k^2) = \frac{1}{n}\sum_{k=1}^{n}k - \frac{1}{n^2}\sum_{k=1}^{n}k^2$$
$$= \frac{n(n+1)}{2n} - \frac{n(n+1)(2n+1)}{6n^2} = \frac{1}{6}\left(n-\frac{1}{n}\right) = \mathcal{O}(n) \quad . \tag{13.53}$$

Man erhält demnach eine lineare Komplexität für den Partitions-Algorithmus.

Zur Berechnung der Komplexität des Quicksort geht man zunächst davon aus, dass die Anzahl der erforderlichen Partitionen im Mittel $\operatorname{ld} n$ beträgt. Daraus folgt dann, dass für den Quicksort die Komplexität sowohl für die Anzahl der Vergleiche als auch für die Anzahl der Zuweisungen im Mittel von der Ordnung $\mathcal{O}(n \operatorname{ld} n)$ ist. Allerdings ist zu bedenken, dass dieses günstige Verhalten nur im Mittel gilt, da im ungünstigsten Fall eine Entartung möglich ist. Es sind dann n Partitionen durchzuführen, was zu einer Komplexität von $\mathcal{O}(n^2)$ führt. Durch die Auswahl des Vergleichselements als Median von mehreren Kandidaten lässt sich dieser Entartung jedoch effizient vorbeugen.

Quicksort ist ein typischer Vertreter der in Kap. 12.2.2, S. 543 besprochenen Divide and Conquer Algorithmen. Daher lässt sich die Komplexität auch mit (12.6) und (12.7) bestimmen: Im Mittel wird das Problem der Größe n in zwei Teilprobleme halber Größe geteilt. Der Zusatzaufwand in jedem Schritt, nämlich die Partitionierung, hat wie oben erläutert lineare Komplexität. Damit ergibt sich:

$$T(n) = 2T(n/2) + \Theta(n) \tag{13.54}$$

und aus (12.7) für die Gesamtkomplexität $\Theta(n \log n)$.

Quelle: Datenstrukturen, Bäume und Graphen [1, S. 611]

Sortieralgorithmen: Quicksort Average-Case

Der echte **Average-Case** ist:

$$T_{\text{avg}}(n) \approx 1.39 \cdot T_{\text{best}}(n)$$
$$T_{\text{avg}}(n) \approx 1.39 \cdot (n \cdot log_2(n) + n)$$

Quelle: 4. Auflage, *Algorithmen*, Sedgewick [2]

Grundlagen der Parallelisierung

- **Amdahlsches Gesetz**, Speedup $1/p$
- theoretischer Speedup aufgrund von Overheads nie erreichbar

$p =$ Thread-Anzahl,

$f =$ serieller Code, wobei $0 < f \leq 1$,

$$t(p) = \underbrace{f \cdot t(1)}_{\text{serielle Arbeit}} + \underbrace{(1-f) \cdot \frac{t(1)}{p}}_{\text{parallele Arbeit}}.$$

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. [4]

- Es ist kein linearer Speedup $(1/p)$, daher nur eine eigene $T(n, p)$-Formel statt des exakten Amdahlschen Gesetzes.

Thread-Modelle, Overheads und Skalierungsgrenzen

- Overheads
- Skalierungsgrenzen
- Thread-Modelle, Implementierungsstrategien

Overheads

- Betrachten **Overheads** aus siecht eines Softwarearchitekten, für den die gesamte Hardware eine Blackbox ist.
- **Overheads** (Zusatzlaufzeiten) verhindern theoretisch ideale Zeitersparnis.
- **Overheads**:
  - Initialisierungs- und Join-Zeiten
  - Destruktoren
  - Synchronisation
  - Swapping (Kontextwechsel)
  - Speicherlatenzen (begrenzte Speichergröße, Cache-Misses, Datenübertragungsgeschwindigkeit (Latenzen), Datenübertragungsrate)
  - ...

Overheads (Text)

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.

Skalierungsgrenzen: Hardware-Grenze CPU

- Begrenzte Anzahl physischer Kerne (im Testsystem 8 physische bzw. 16 logische Prozessoren).
- Je größer die Leistungsaufnahme der CPU ist, desto ineffizienter arbeitet sie.
- Begrenzte maximale Leistungsaufnahme (Thermal Design Power, TDP), welche dafür sorgt, dass mehr Threads nicht unbedingt mehr Leistung bedeuten.
- Leistungsaufnahme wird größtenteils zu Abwärme, die im schlimmsten Fall zur Runtertaktung (Thermal Throttling) einzelner CPU-Kerne führt.
- Der wenige Rest der Leistungsaufnahme baut ein sich ständig wechselndes Magnetfeld auf, welches dabei Störströme (Kreisströme) induziert.

Overheads: Hardware-Grenze CPU (Text)

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.

Thread-Modelle, Implementierungsstrategien

- Jeden Codeabschnitt ohne sequentielle Abhängigkeiten in einen neuen Thread auslagern
- Thread-Anzahl begrenzen, da sonst oft Overheads dominieren.
- Nutzen von Worker-Threads (Threadpool)
- Work-Stealing-Ansatz
- Nur neue Aufgaben auf Aufgabenliste legen, wenn freie Worker-Threads vorhanden sind.

Thread-Modelle, Implementierungsstrategien (Text)

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 und der Messungen detailliert analysiert.

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 [3].

Begriffserklärung Work-Stealing-ähnliche Ansatz

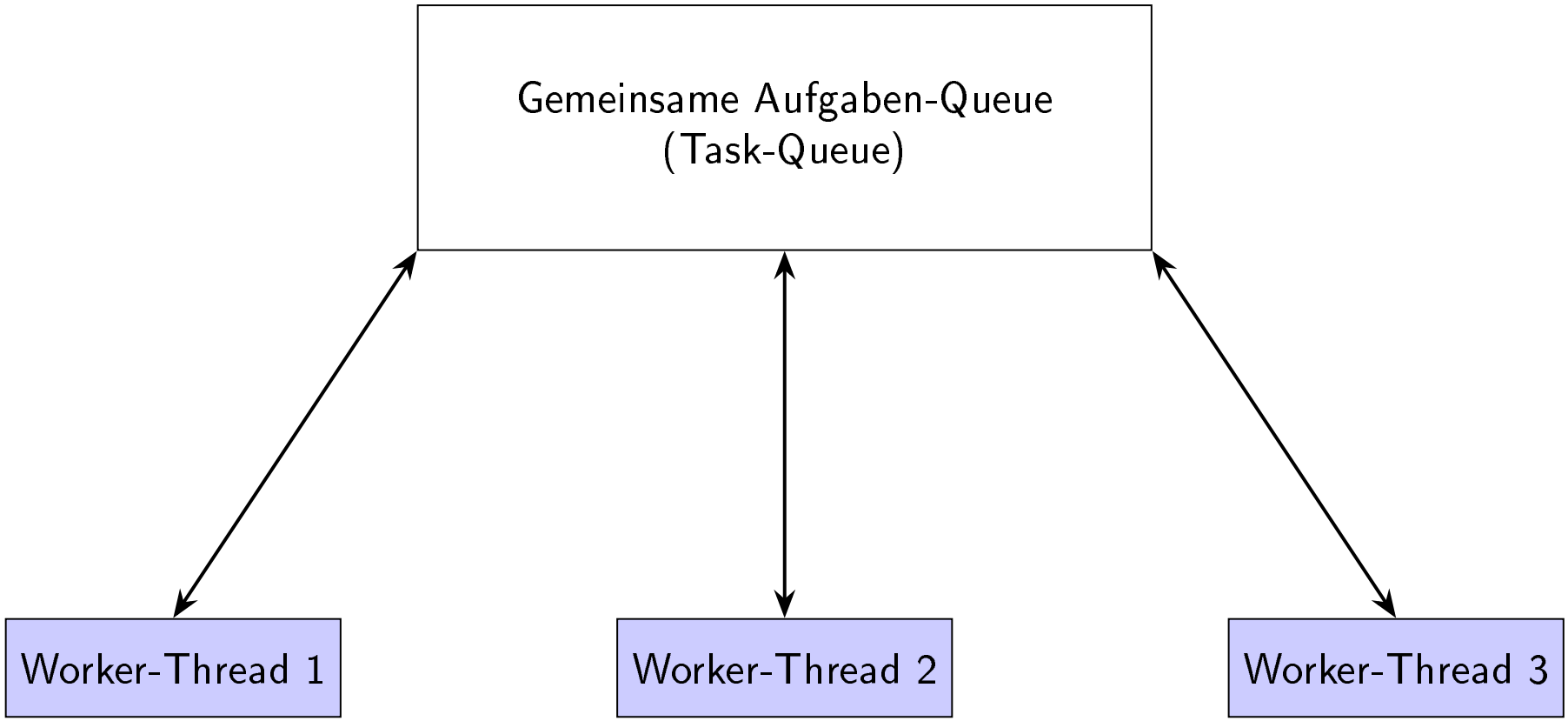


Abbildung: Work-Stealing-ähnlicher Ansatz mit gemeinsamer Aufgaben-Queue: Threads entnehmen Aufgaben aus der Queue und können neue Aufgaben auf diese legen.

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.

Methodik und Versuchsaufbau

- Messumgebung und Hardware
- Implementierungsvarianten
- Programmablauf als Baumstruktur
- Parallele Implementierungsvarianten Code
- Optimierung
- Messmethodik

Messumgebung und Hardware

- CPU: AMD Ryzen 7 5800x, 8-Kern-CPU (8 physische Kerne, 16 logische Prozessoren)
- 32 GB Ram: DDR4-3200, CL16-18-18-38 (2 Kits, insgesamt 4×8 GB)
- Betriebssystem: Windows 10 Version 22H2
- 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.
- Im integrierten Terminal von Visual Studio Code.
- In einer Release-Konfiguration (also mit Compiler-Optimierungen).

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.

Implementierungsvarianten

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

- eine rekursiv sequenzielle Variante,
- eine rekursive Variante mit 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).

Implementierungsvarianten: Formeln: parallel Mergesort

$$p = \text{Thread-Anzahl}$$

$$e = \log_2(p)$$

$$T(n, e) = \begin{cases} 2 \cdot T\left(\frac{n}{2}, 0\right) + n & , \text{wenn } e = 0 \\ 1 \cdot T\left(\frac{n}{2}, e-1\right) + n & , \text{wenn } e > 0 \end{cases}$$

$$T(n, p) = 2n\left(1 - \frac{1}{p}\right) + \frac{n}{p} \cdot \log_2\left(\frac{n}{p}\right) + \frac{n}{p}$$

$$O(T(n, p)) = O\left(\frac{n}{p} \cdot \log_2(n) + n\right)$$

$$e_{\max} = \log_2(n)$$

$$p_{\max} = n$$

Implementierungsvarianten: Formeln: parallel Quicksort

Der **Best-Case** von Quicksort ist:

$$p = \text{Thread-Anzahl}$$

$$e = \log_2(p)$$

$$T(n, e) = \begin{cases} 2 \cdot T\left(\frac{n}{2}, 0\right) + n & , \text{wenn } e = 0 \\ 1 \cdot T\left(\frac{n}{2}, e-1\right) + n & , \text{wenn } e > 0 \end{cases}$$

$$T(n, p) = 2n\left(1 - \frac{1}{p}\right) + \frac{n}{p} \cdot \log_2\left(\frac{n}{p}\right) + \frac{n}{p}$$

$$O(T(n, p)) = O\left(\frac{n}{p} \cdot \log_2(n) + n\right)$$

$$e_{\max} = \log_2(n)$$

$$p_{\max} = n$$

Implementierungsvarianten: Formeln: parallel Quicksort

Der **Worst-Case** von Quicksort (Worker-Thread-Variante) bei $p > 1$ ist:

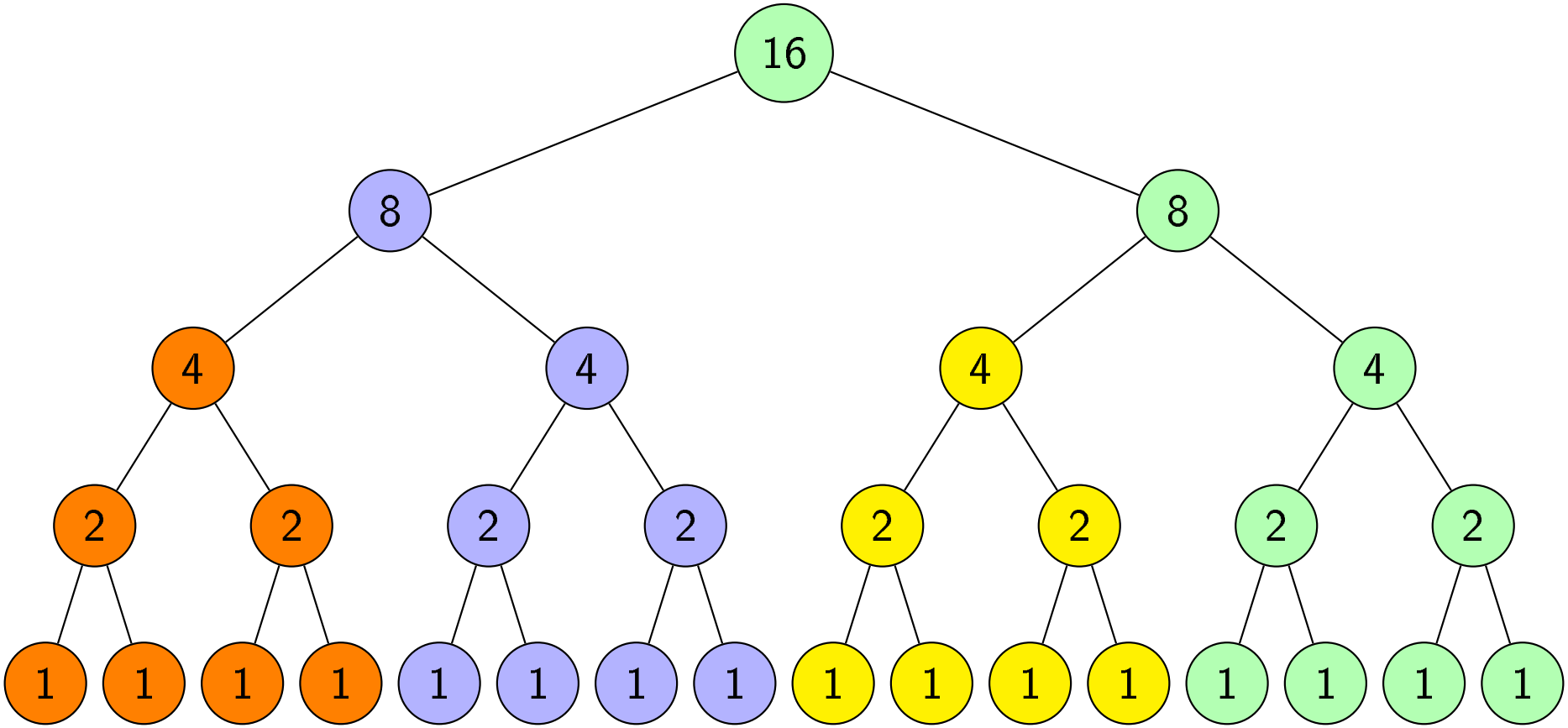
$$
\begin{aligned}
T(n) &= q_2 + n, \\
T(n) &= T(n-1) + n, \\
T(n) &= \frac{1}{2} \cdot (n^2 + n), \\
O(T(n)) &= O(n^2).
\end{aligned}
$$

Implementierungsvarianten

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.

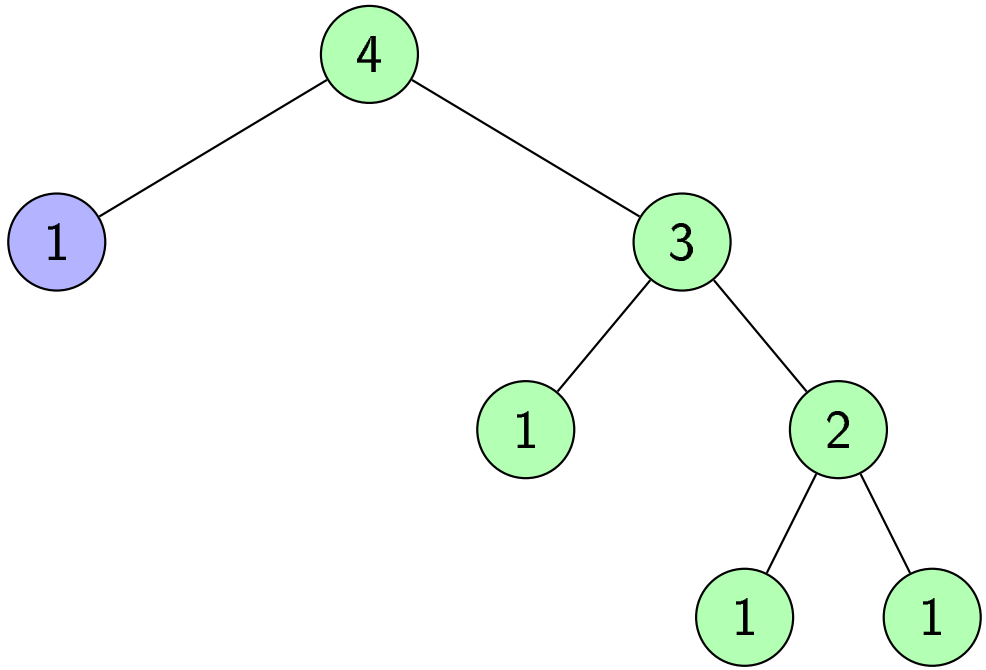
Balancierter Binärbaum für $n = 16$, $p = 4$, $e = 2$

Tiefenbasierte Thread-Erzeugung bei Mergesort und Best-Case von Quicksort:



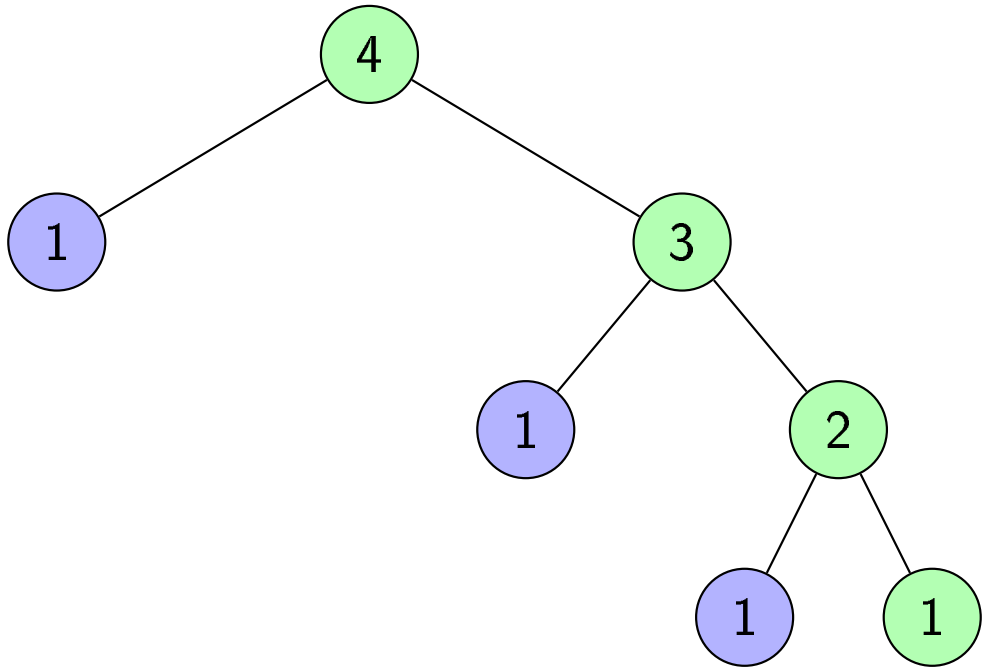
Maximal unbalancierter Binärbaum für $n = 4$, $p = 2$, $e = 1$

Tiefenbasierte Thread-Erzeugung Quicksort **Worst-Case**:



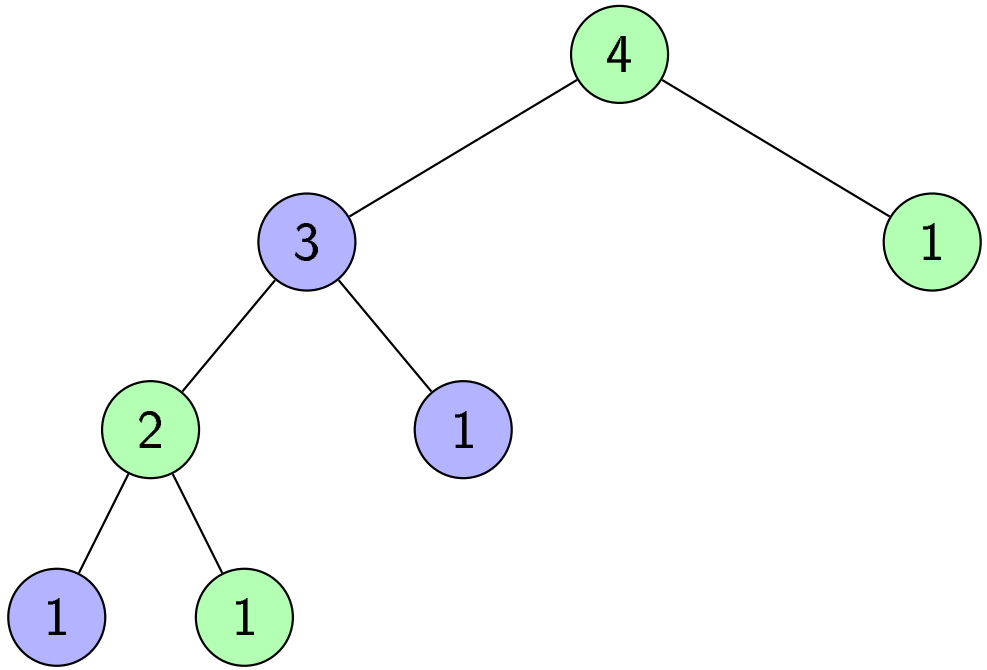
Maximal unbalancierter Binärbaum für $n = 4$, $p = 2$

Worker-Thread-Variante Quicksort **Worst-Case**:



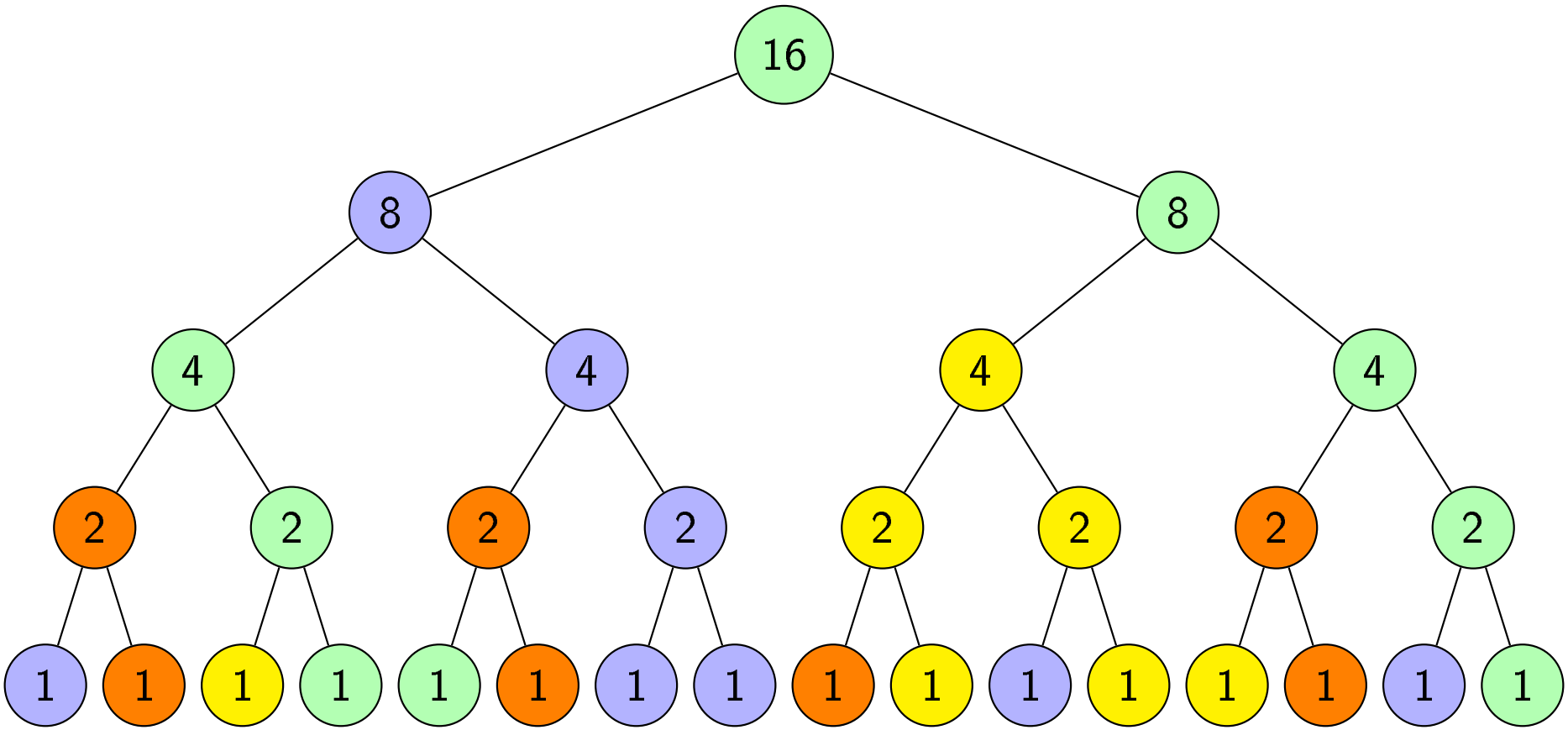
Maximal unbalancierter Binärbaum für $n = 4$, $p = 2$

Worker-Thread-Variante Quicksort **Worst-Case**:



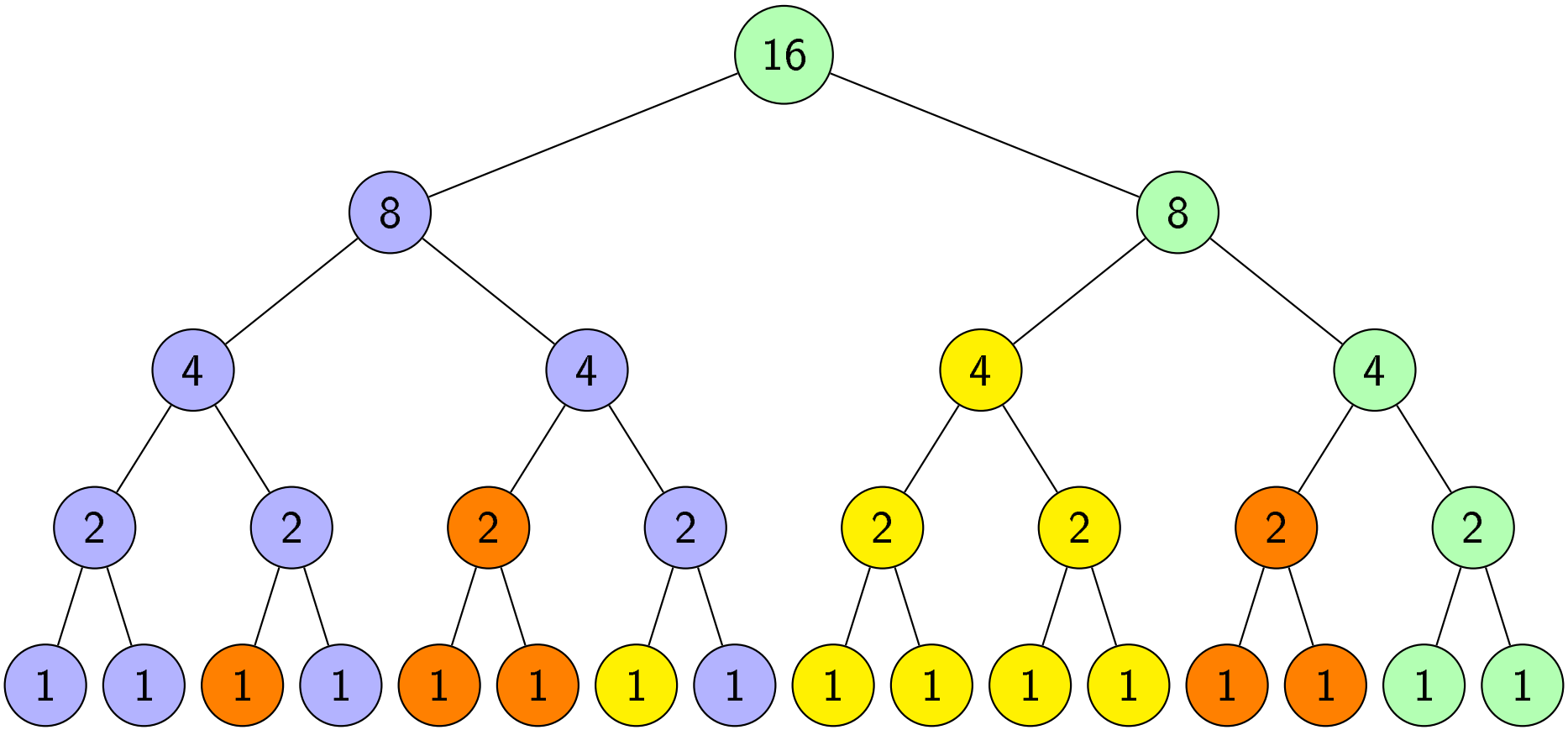
Balancierter Binärbaum für $n = 16$, $p = 4$

Worker-Thread-Variante Quicksort **Best-Case**:



Balancierter Binärbaum für $n = 16$, $p = 4$

Worker-Thread-Variante Mergesort:



Implementierungsvarianten Code

Listing: 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 };
```

Implementierungsvarianten Code

Listing: Tiefenbasierte Thread-Erzeugung Quicksort

```
1  void Quicksort::quicksortP(int *liste, const int links, const int
       rechts, const int aktuelleEbene, const int neueThreadsBisEbene)
       {
2      if (aktuelleEbene < neueThreadsBisEbene) {
3          if (links < rechts) {
4              int ml, mr;
5              partitioniere(liste, links, rechts, ml, mr);
6              // quicksort(liste, links, ml);
7              std::thread thread(static_cast<void (*)(int *, const
                   int, const int, const int, const int)>(&Quicksort::
                   quicksortP), liste, links, ml, aktuelleEbene + 1,
                   neueThreadsBisEbene);
8              // quicksort(liste, mr, rechts);
9              quicksortP(liste, mr, rechts, aktuelleEbene + 1,
                   neueThreadsBisEbene);
10             thread.join();
11         }
12     } else {
13         quicksort(liste, links, rechts);
14     }
15 };
```

Implementierungsvarianten Code

Listing: Worker-Thread-Variante Mergesort

```
1  void Mergesort::mergesortW(int *liste, int links, int rechts, int
       workerThreads) {
2      MergeWorkerPool pool(workerThreads - 1);
3      pool.taskHandler = [&](int *liste, int links, int rechts,
           MergeWorkerPool &pool) {
4          if (links < rechts) {
5              int lange = rechts - links + 1;
6              if (lange < Sortierverfaren::mindestLange) {
7                  mergesort(liste, links, rechts);
8              } else {
9                  int mitte = links + ((rechts - links) / 2);
10                 auto leftHandle = pool.addTaskSmart({liste, links,
                       mitte});
11                 pool.taskHandler(liste, mitte + 1, rechts, pool);
12                 leftHandle.wait();
13                 mischen(liste, links, mitte, rechts, lange);
14             }
15         }
16     };
17     // Starttask
18     pool.taskHandler(liste, links, rechts, pool);
19 }
```

Implementierungsvarianten Code

Listing: Worker-Thread-Variante Quicksort

```
1  void Quicksort::quicksortW(int *liste, int links, int rechts, int
       workerThreads) {
2      WorkerPool pool(workerThreads);
3      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 }
```

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) 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.

Messmethodik

- Zufällige Listen (Arrays), gleicher Seed $\rightarrow$ reproduzierbar, vergleichbar
- Zeitmessung mit `chrono`, Auflösung von 100 ns
- Zur Einordnung gilt: $1\,\text{s} = 10^3\,\text{ms} = 10^9\,\text{ns}$.
- Zeitpunkte der Messung: unmittelbar vor und nach dem Aufruf zum Sortieren.
- Fast immer überprüft, ob das Ergebnis sortiert ist, um Korrektheit sicherzustellen.
- Einzelergebnis (Stichprobenmessung), da praxisnäher und realitätssicher.
- Insgesamt rund 20 000 Einzelmessungen (Messdateien).
- Es wird nur auf die wichtigsten Messungen eingegangen (rund 300 Einzelmessungen).

Messmethodik (Text)

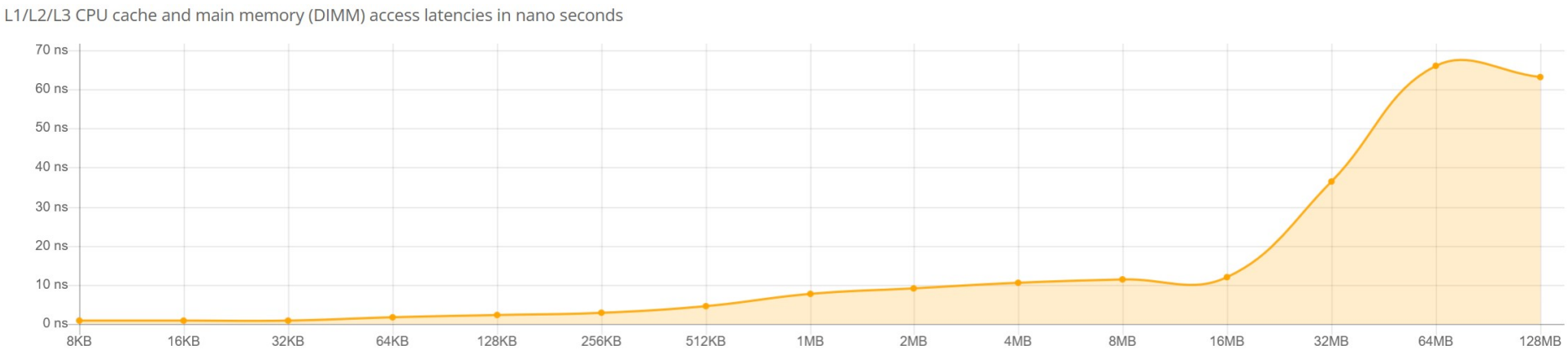
Zur Laufzeitmessung wurden zufällig erzeugte Listen verwendet, die stets mit demselben Seed initialisiert wurden. Dadurch sind alle Messungen reproduzierbar und miteinander vergleichbar.
Die Zeitmessung erfolgte mithilfe der Bibliothek `chrono`. Die gemessenen Zeiten besitzen eine Auflösung von 100 ns 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.

Formeln: Einheiten und Skalierung

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

Aufgrund von Overheads wie z. B. erhöhte Speicherlatenzen durch Cache-Misses:

$$x = f(n, p)$$
$$f(n, p) \leq f(n + 1, p)$$
$$f(n, p) \leq f(n, p + 1)$$



Nebeneffekt des Skalierens mit dem Faktor x

Sollte die Standard-Formel

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

falsch sein und stattdessen diese Formel gelten

$$T(n) = 2T\left(\frac{n}{2}\right) + (n-1)$$

kann man diese auch schreiben als

$$T(n) = 2T\left(\frac{n}{2}\right) + \underbrace{\left(1 - \frac{1}{n}\right)}_{x} \cdot n$$

Dabei ist $\left(1 - \frac{1}{n}\right)$ implizit auch durch x repräsentiert. Durch diesen Nebeneffekt korrigiert der Faktor x auch gleich implizit die Formel. Dies macht die Formel nach dem Hochskalieren auf die sequentielle Laufzeit so präzise und sorgt damit dafür, dass die T(n,p)-Formel so exakt die untere Grenze der Laufzeitverbesserung vorhersagen kann.

Ergebnisse und Analyse

- Inkrement-Array
- Grundlegende Laufzeiten abhängig von der Arraygröße (sequenziell)
- Einfluss des Listentyps (sequenziell)
- Tiefenbasierte Thread-Erzeugung
- Workerthreads

Ergebnisse und Analyse: Begriffsdefinition

- 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.

Inkrement-Array

```
1     std::unique_ptr<int[]> liste(new int[lange]);
2     volatile int *vptr = liste.get();
3
4     auto start = std::chrono::high_resolution_clock::now();
5     for (long long i = 0; i < lange; i++) {
6         vptr[i] = vptr[i]++;
7     }
8     auto stop = std::chrono::high_resolution_clock::now();
9
10    long long dauer = std::chrono::duration_cast<std::chrono
          ::nanoseconds>(stop - start).count();
```

Inkrement-Array

```
void Manager::incArrayMT(volatile int *liste, long long lange, int
    threadCount) {
    std::vector<std::thread> threads;
    threads.reserve(threadCount - 1);
    long long chunk = lange / threadCount;

    for (int t = 0; t < threadCount; t++) {
        long long start = t * chunk;
        long long lambdaStart = start;
        if (t == threadCount - 1) {
            // letzter Thread
            long long lambdaEnd = lange;
            for (long long i = lambdaStart; i < lambdaEnd; i++) {
                liste[i] = liste[i] + 1;
            }
        } else {
            long long lambdaEnd = start + chunk;
            threads.emplace_back([=]() {
                for (long long i = lambdaStart; i < lambdaEnd; i++)
                    {
                    liste[i] = liste[i] + 1;
                }
            });
        }
    }
    for (auto &t : threads) {
        t.join();
    }
};
```

Inkrement-Array

2× Geschwindigkeit

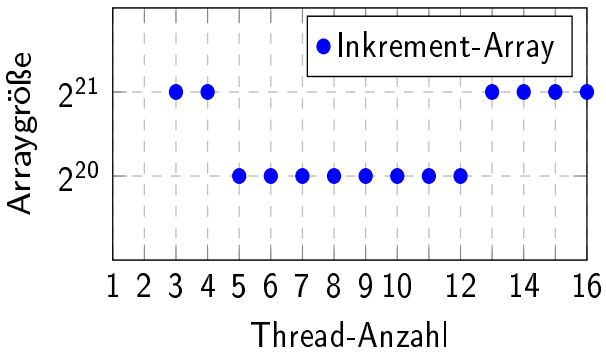
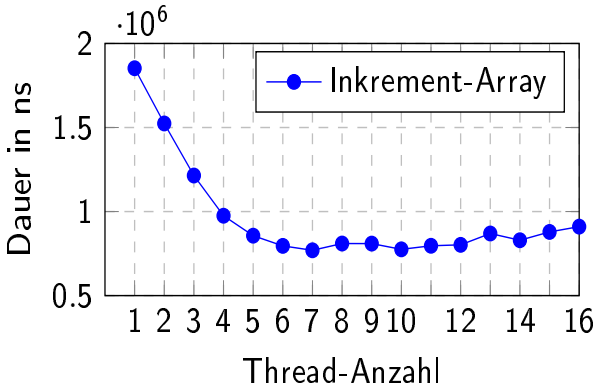


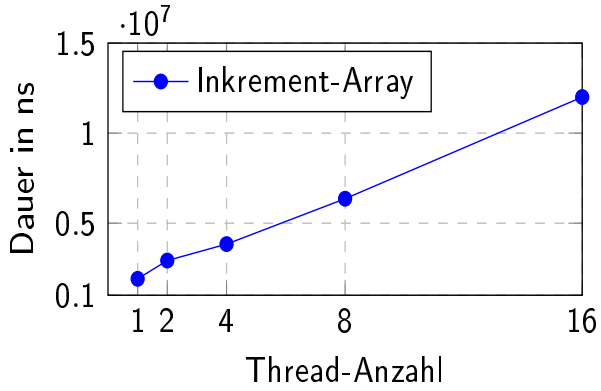
Abbildung: Zeigt, ab welcher Arraygröße zum ersten Mal die halbe Laufzeit gegenüber der sequentiellen Laufzeit erreicht wird.

Strong Scaling: Laufzeit bei Arraygröße 2^{21}



Abbildung

Weak Scaling (Start-Größe 2^{21})



Abbildung

Laufzeit bei Arraygröße 16

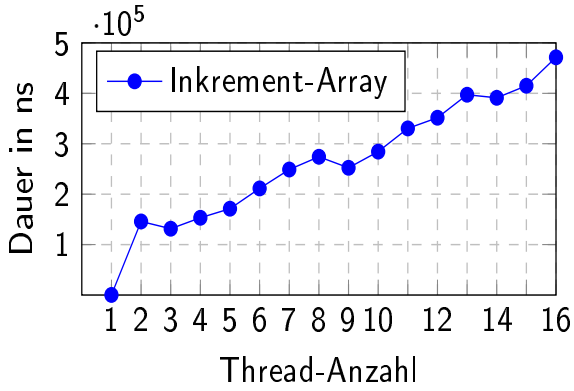
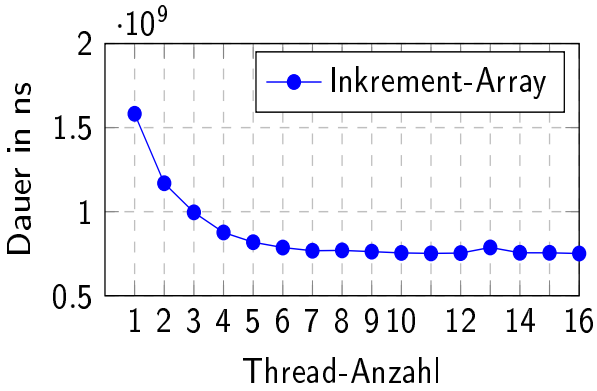


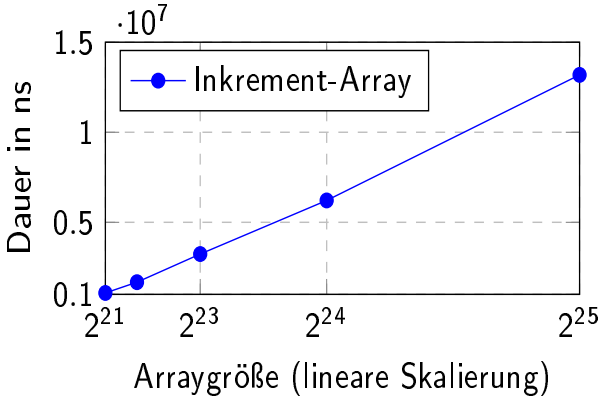
Abbildung: Zeigt die Overheads, die durch Threads entstehen.

Strong Scaling: Laufzeit bei Arraygröße 2^{31}



Abbildung

Arraygröße verdoppeln (16 Threads)



Abbildung

Grundlegende Laufzeiten abhängig von der Arraygröße (sequenziell)

Einfluss des Listentyps (sequenziell)

Tiefenbasierte Thread-Erzeugung

Workerthreads

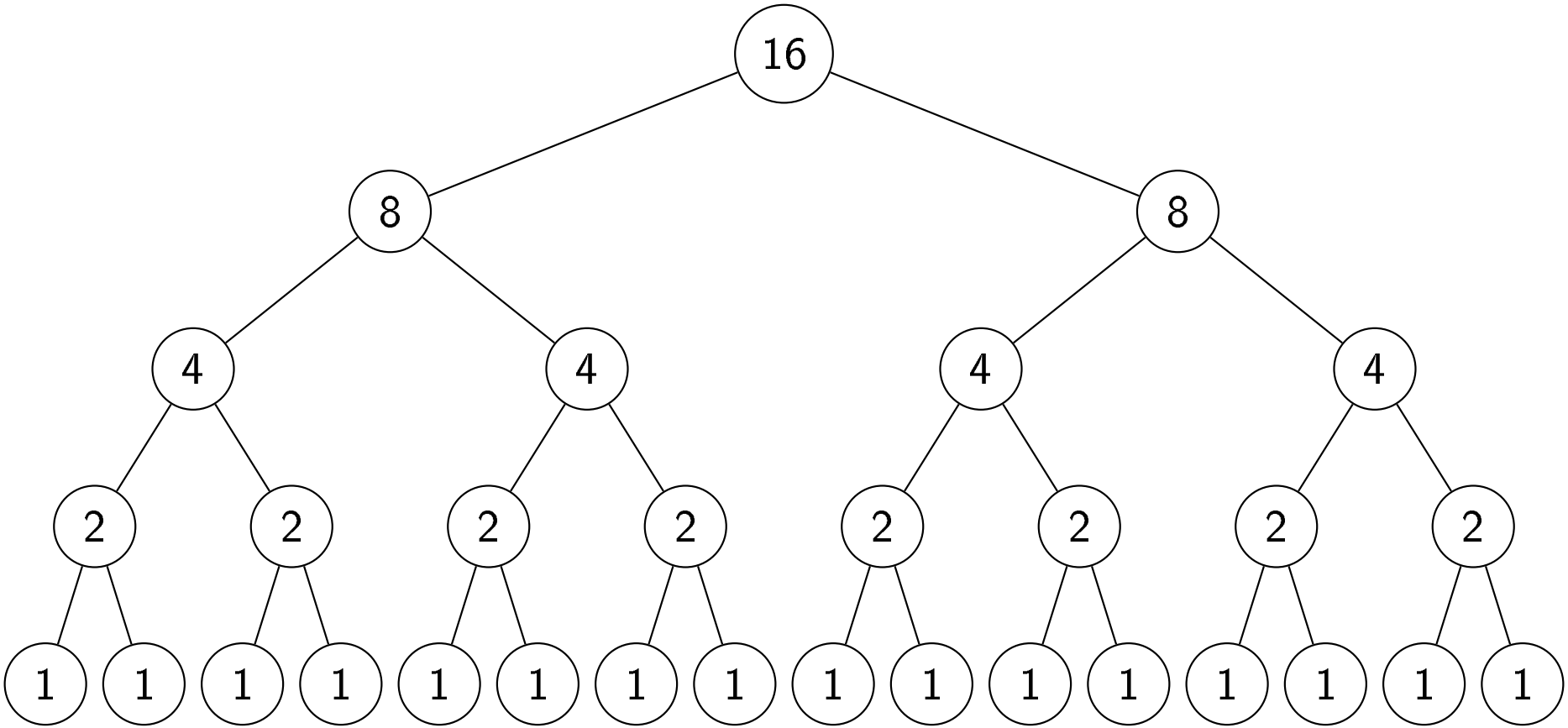
Beantwortung der Forschungsfrage

Zusammenfassung und Ausblick

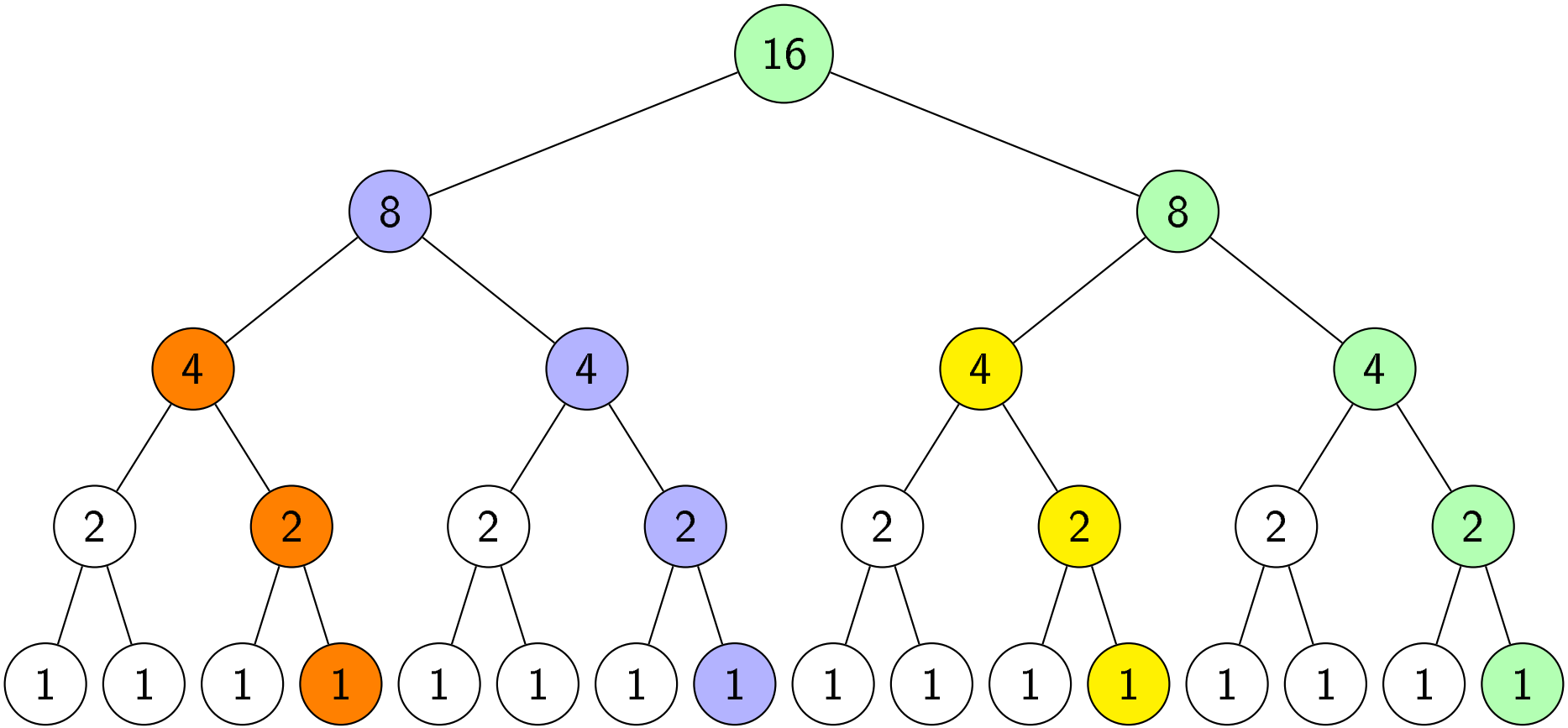
Vielen Dank für Ihre Aufmerksamkeit!

Fragen und Diskussion

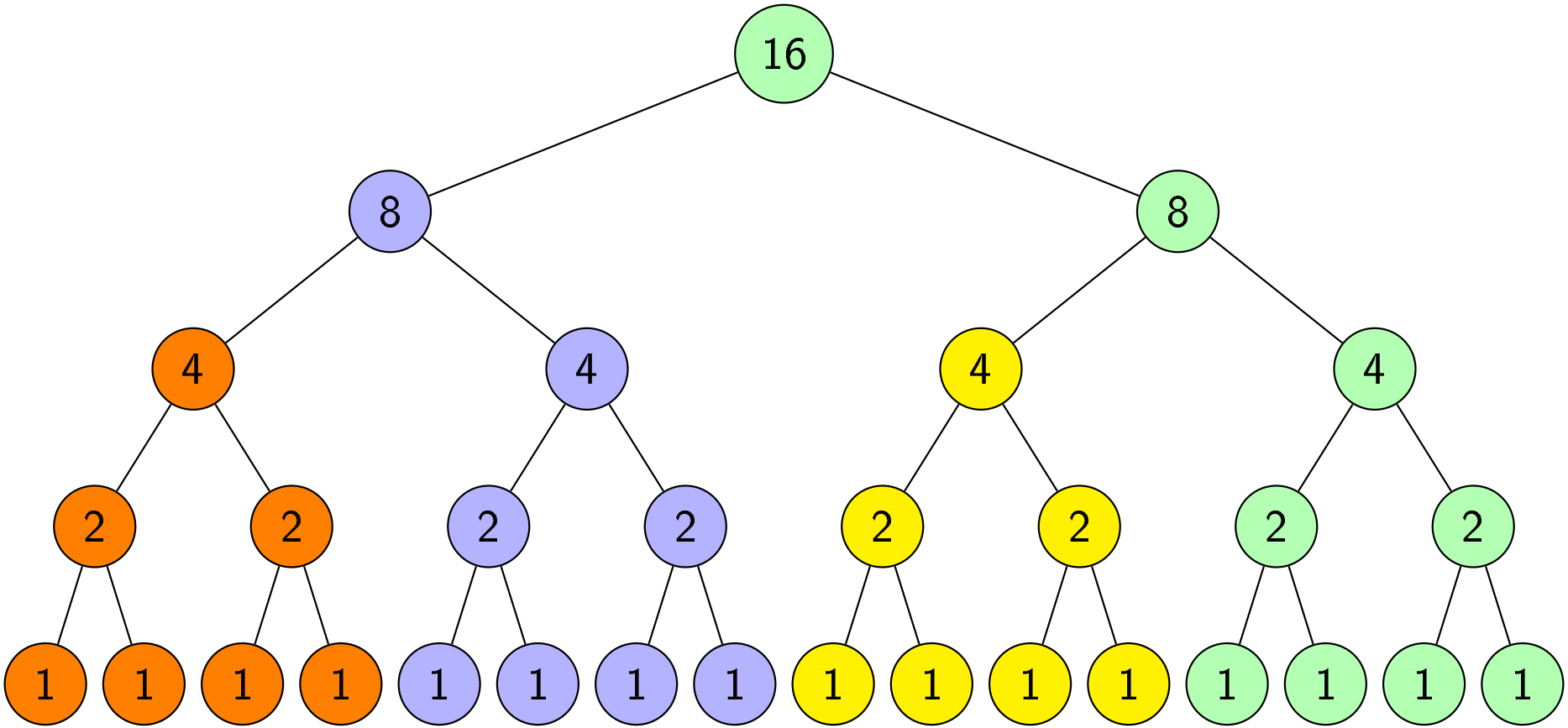
Balancierter Binärbaum für $n = 16$



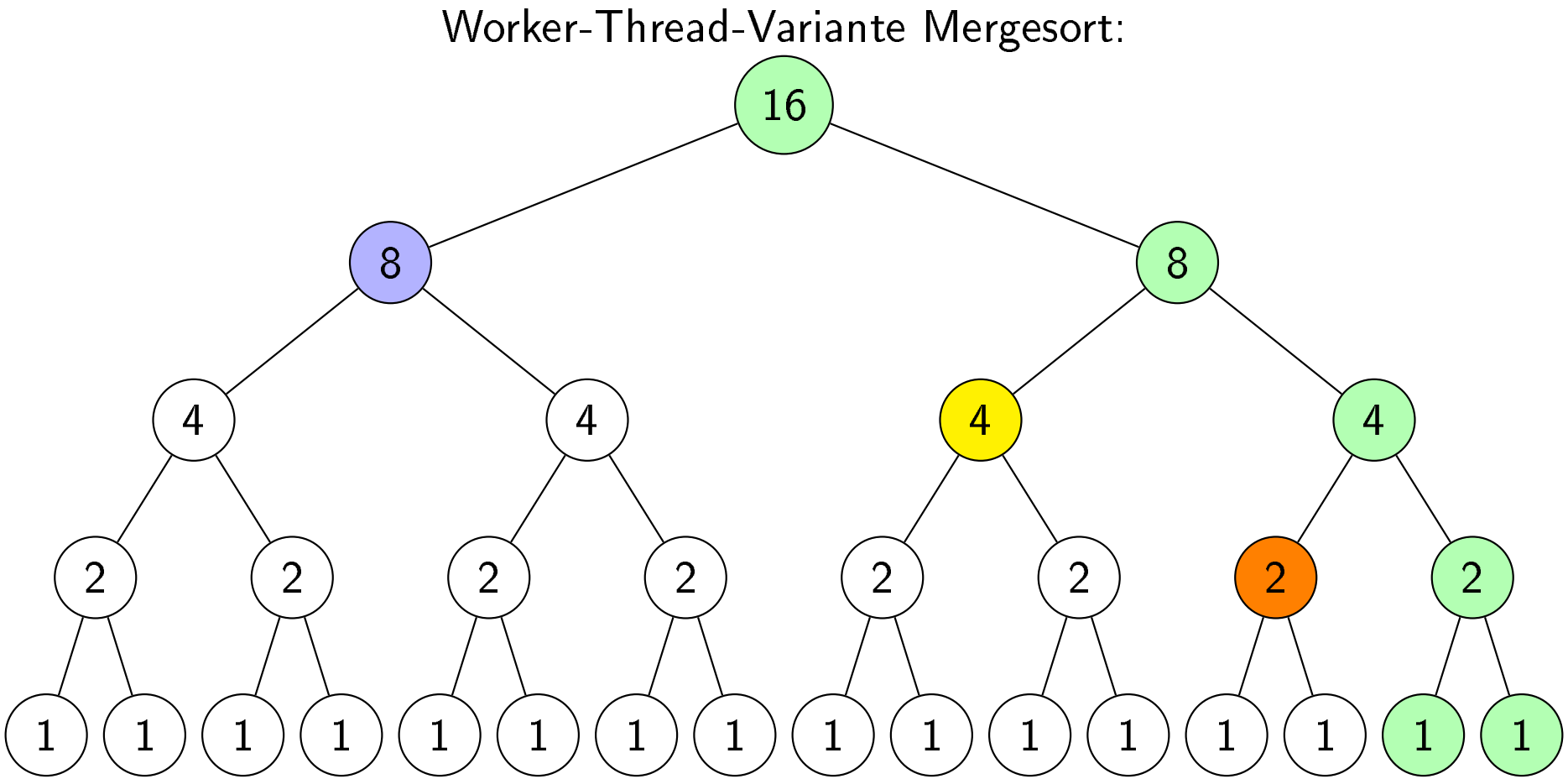
Balancierter Binärbaum für $n = 16$, $p = 4$, $e = 2$



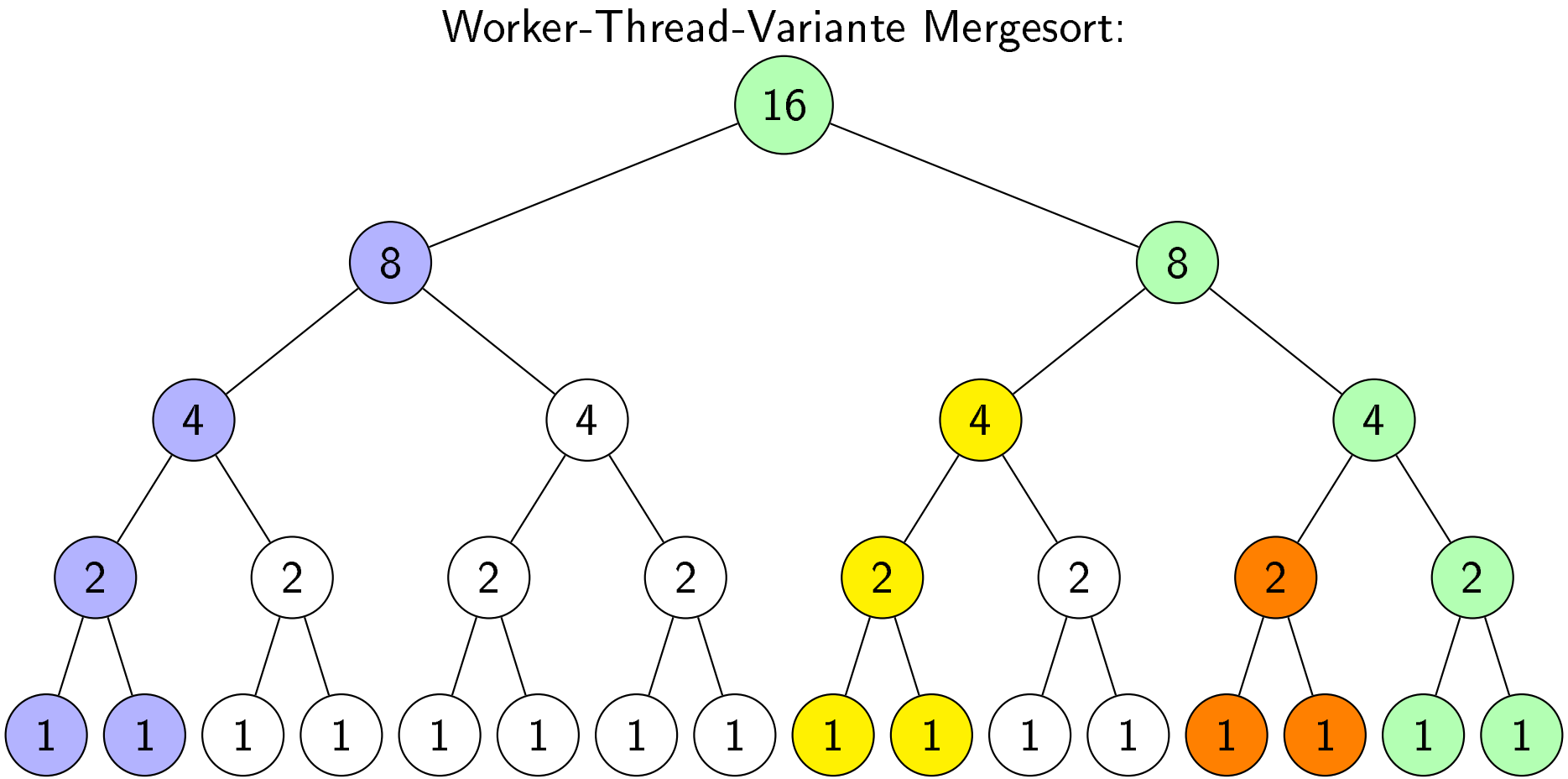
Balancierter Binärbaum für $n = 16$, $p = 4$, $e = 2$



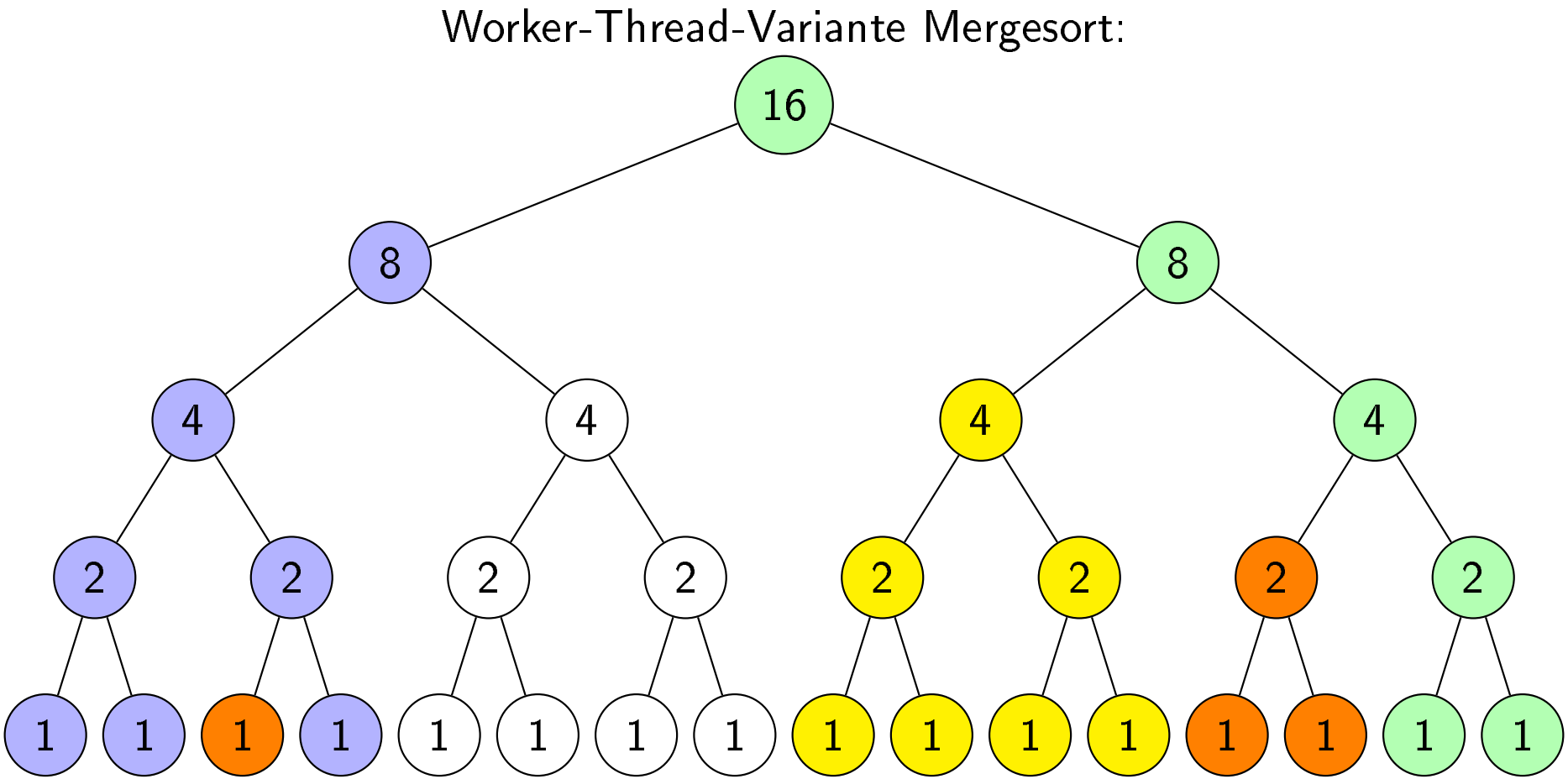
Balancierter Binärbaum für $n = 16$, $p = 4$



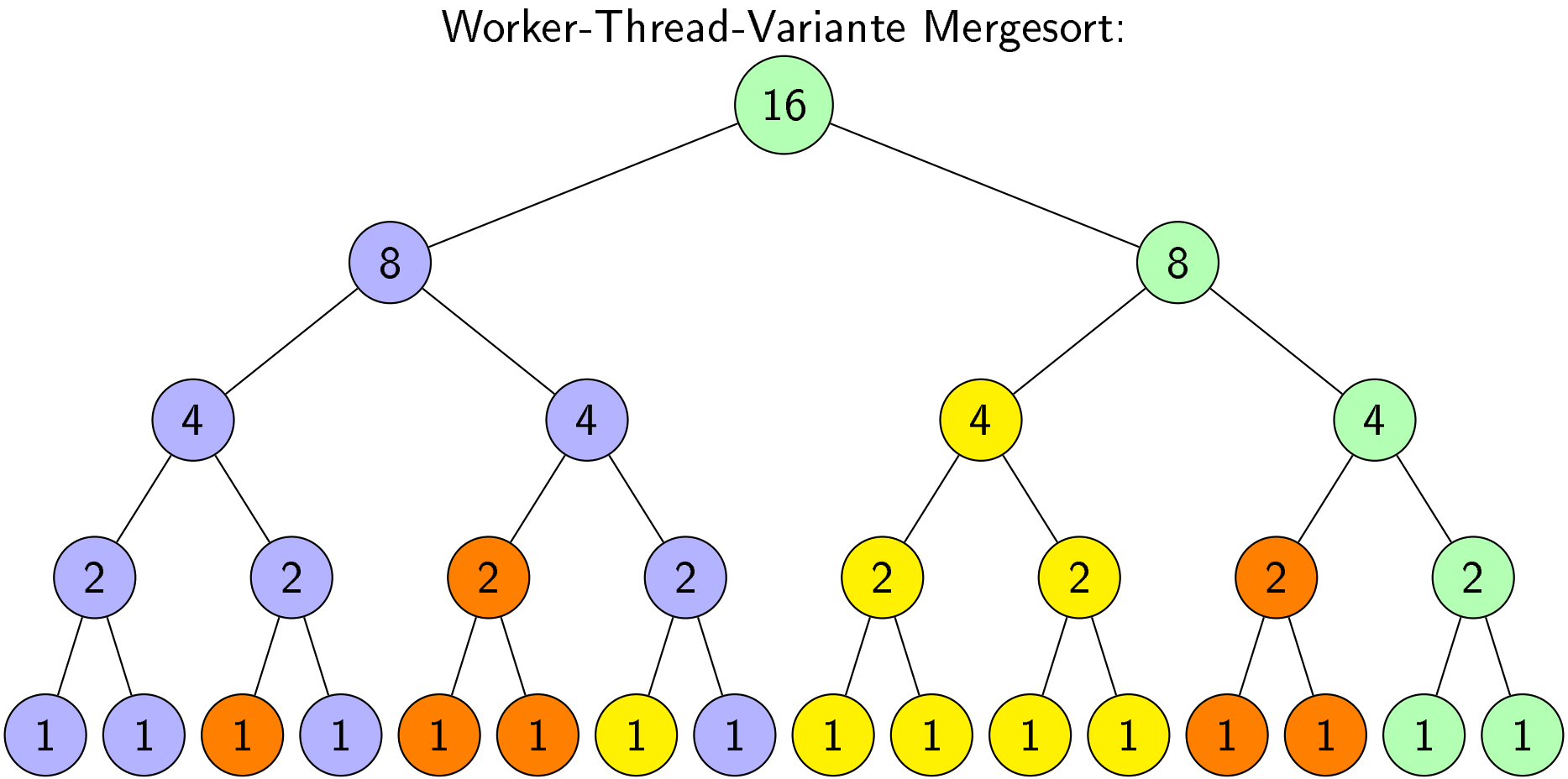
Balancierter Binärbaum für $n = 16$, $p = 4$



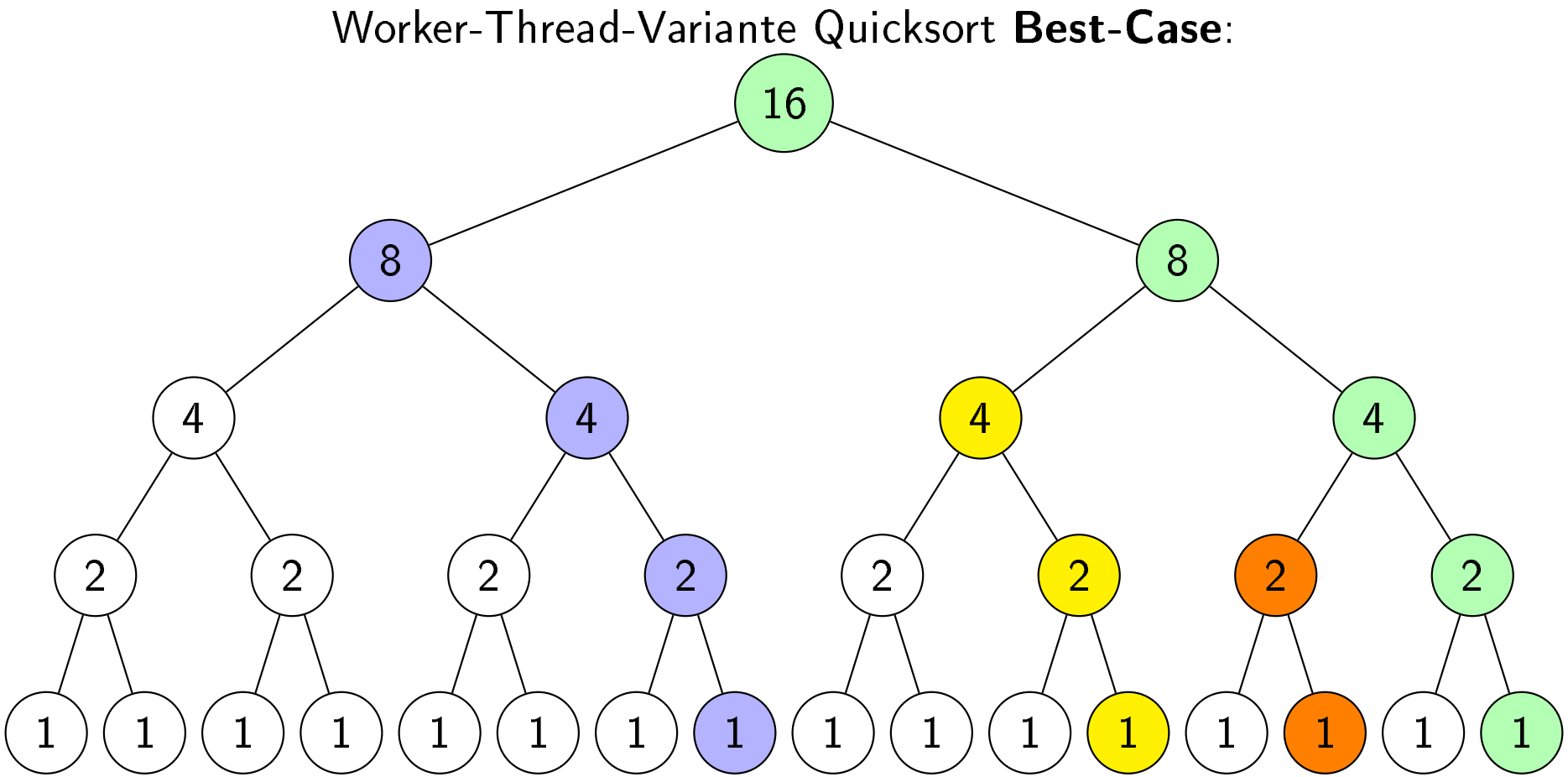
Balancierter Binärbaum für $n = 16$, $p = 4$



Balancierter Binärbaum für $n = 16$, $p = 4$

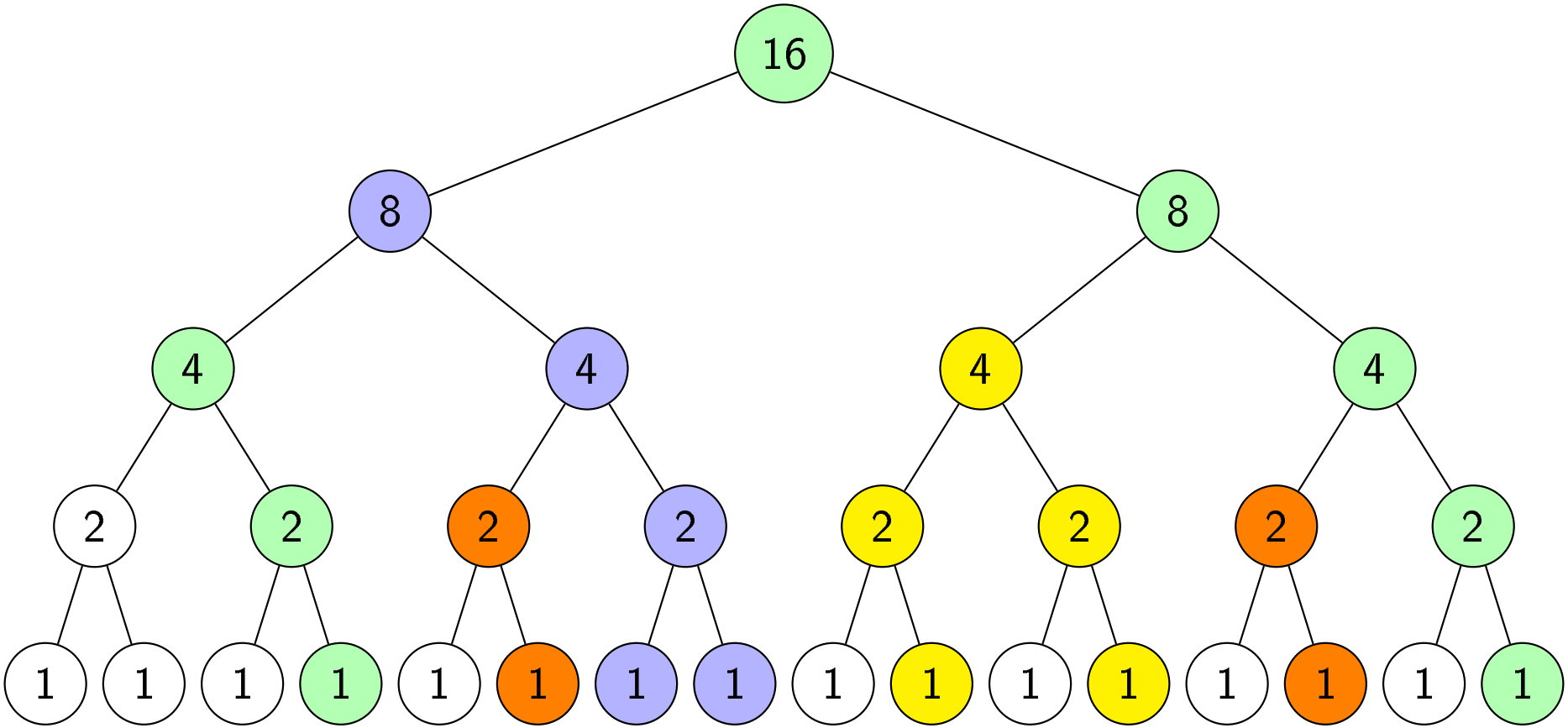


Balancierter Binärbaum für $n = 16$, $p = 4$



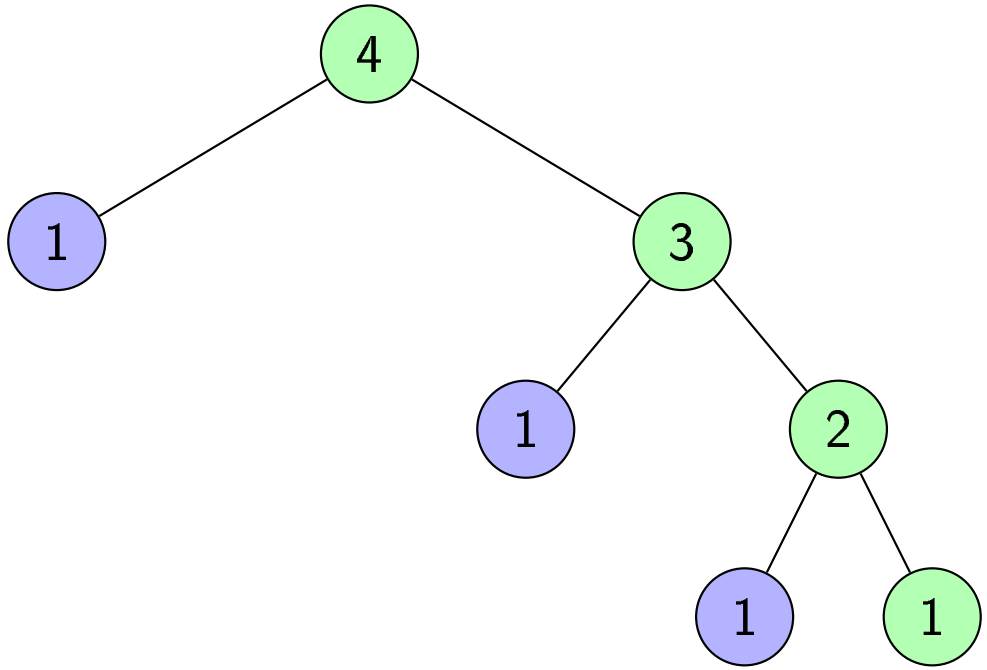
Balancierter Binärbaum für $n = 16$, $p = 4$

Worker-Thread-Variante Quicksort **Best-Case**:



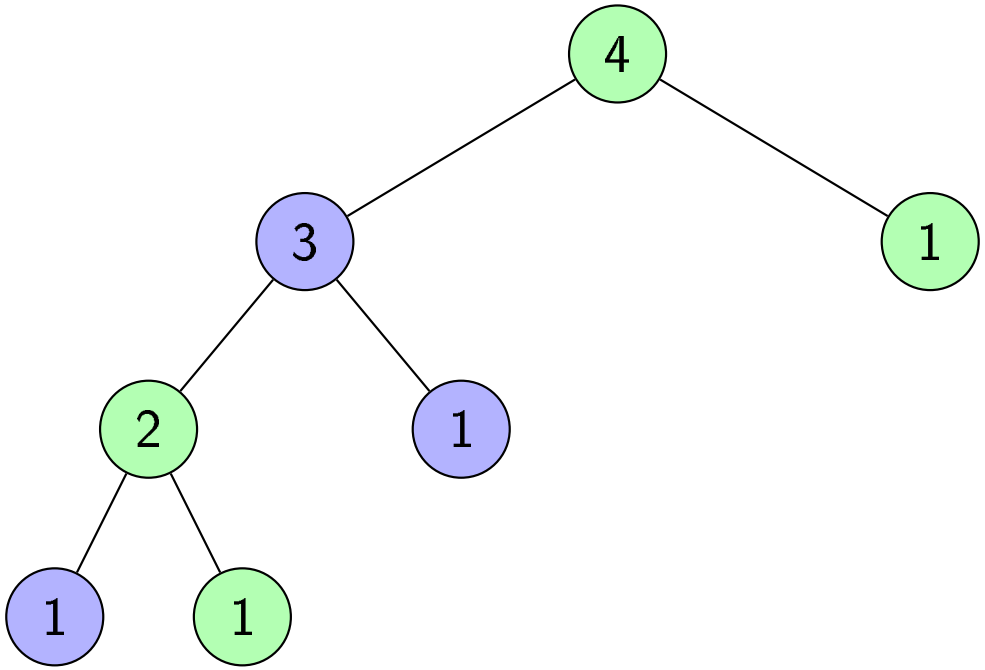
Maximal unbalancierter Binärbaum für $n = 4$, $p = 2$

Worker-Thread-Variante Quicksort **Worst-Case**:



Maximal unbalancierter Binärbaum für $n = 4$, $p = 2$

Worker-Thread-Variante Quicksort **Worst-Case**:



Formeln: Mergesort

$$
\begin{aligned}
T(n) &= m_1 + m_2 + n \\
T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \\
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n, \\
T(n) &= n \cdot \log_2(n) + n, \\
O(T(n)) &= O(n \log n).
\end{aligned}
$$

Formeln: Quicksort

Best-Case von Quicksort:

$$
\begin{aligned}
T(n) &= q_1 + q_2 + n \\
T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \\
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n, \\
T(n) &= n \cdot \log_2(n) + n, \\
O(T(n)) &= O(n \log n).
\end{aligned}
$$

Formeln: Quicksort

Der **Worst-Case** von Quicksort ist:

$$
\begin{aligned}
T(n) &= q_1 + q_2 + n \\
T(n) &= T(n-1) + 1 + n, \\
T(n) &= \frac{1}{2} \cdot (n^2 + n) + n, \\
O(T(n)) &= O(n^2).
\end{aligned}
$$

Der **heuristisch betrachtete Average-Case** von Quicksort ist:

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

Formeln: parallel Mergesort

$$p = \text{Thread-Anzahl}$$

$$e = \log_2(p)$$

$$T(n, e) = \begin{cases} 2 \cdot T\left(\frac{n}{2}, 0\right) + n & , \text{wenn } e = 0 \\ 1 \cdot T\left(\frac{n}{2}, e - 1\right) + n & , \text{wenn } e > 0 \end{cases}$$

$$T(n, p) = 2n\left(1 - \frac{1}{p}\right) + \frac{n}{p} \cdot \log_2\left(\frac{n}{p}\right) + \frac{n}{p}$$

$$O(T(n, p)) = O\left(\frac{n}{p} \cdot \log_2(n) + n\right)$$

$$e_{\max} = \log_2(n)$$

$$p_{\max} = n$$

Formeln: parallel Quicksort

Der **Best-Case** und **heuristisch betrachtete Average-Case** von Quicksort ist:

$$p = \text{Thread-Anzahl}$$

$$e = \log_2(p)$$

$$T(n, e) = \begin{cases} 2 \cdot T\left(\frac{n}{2}, 0\right) + n & , \text{wenn } e = 0 \\ 1 \cdot T\left(\frac{n}{2}, e-1\right) + n & , \text{wenn } e > 0 \end{cases}$$

$$T(n, p) = 2n\left(1 - \frac{1}{p}\right) + \frac{n}{p} \cdot \log_2\left(\frac{n}{p}\right) + \frac{n}{p}$$

$$O(T(n, p)) = O\left(\frac{n}{p} \cdot \log_2(n) + n\right)$$

$$e_{\max} = \log_2(n)$$

$$p_{\max} = n$$

Formeln: parallel Quicksort

Der **Worst-Case** von Quicksort (Worker-Thread-Variante) bei $p > 1$ ist:

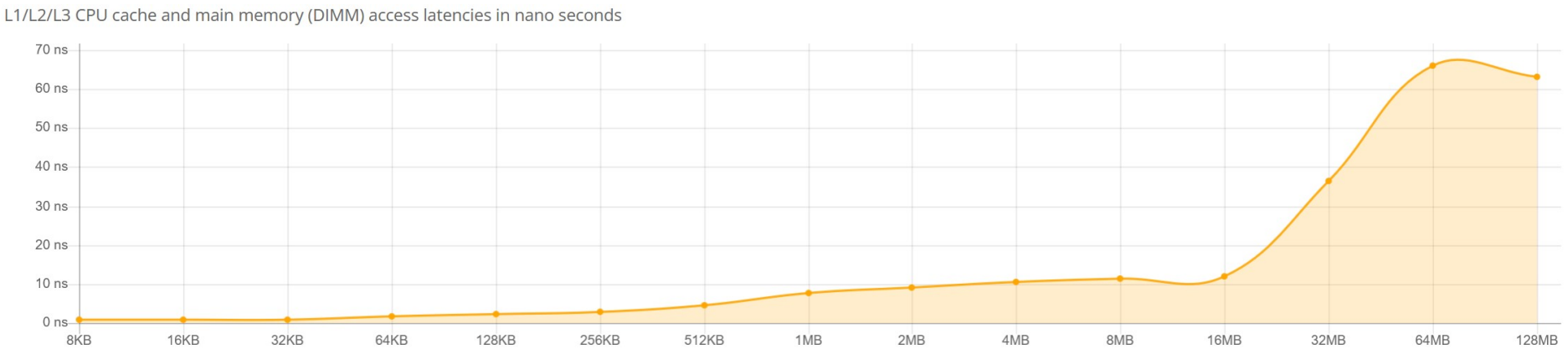
$$
\begin{aligned}
T(n) &= q_2 + n, \\
T(n) &= T(n-1) + n, \\
T(n) &= \frac{1}{2} \cdot (n^2 + n), \\
O(T(n)) &= O(n^2).
\end{aligned}
$$

Formeln: Einheiten und Skalierung

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

Aufgrund von Overheads wie z. B. erhöhte Speicherlatenzen durch Cache-Misses:

$$x = f(n, p)$$
$$f(n, p) \leq f(n+1, p)$$
$$f(n, p) \leq f(n, p+1)$$



Nebeneffekt des Skalierens mit dem Faktor x

Sollte die Standard-Formel

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

falsch sein und stattdessen diese Formel gelten

$$T(n) = 2T\left(\frac{n}{2}\right) + (n-1)$$

kann man diese auch schreiben als

$$T(n) = 2T\left(\frac{n}{2}\right) + \underbrace{\left(1 - \frac{1}{n}\right)}_{x} \cdot n$$

Dabei ist $\left(1 - \frac{1}{n}\right)$ implizit auch durch x repräsentiert. Durch diesen Nebeneffekt korrigiert der Faktor x auch gleich implizit die Formel. Dies macht die Formel nach dem Hochskalieren auf die sequentielle Laufzeit so präzise und sorgt damit dafür, dass die T(n,p)-Formel so exakt die untere Grenze der Laufzeitverbesserung vorhersagen kann.

Herleitung T(n,p): Ausgangsdefinition

Gegeben ist die Rekursion:

$$T(n,e) = \begin{cases} 2\,T\left(\frac{n}{2},0\right) + n & \text{falls } e = 0 \\ T\left(\frac{n}{2},e-1\right) + n & \text{falls } e > 0 \end{cases}$$

Zusätzlich gilt:

$$p = 2^e$$

Herleitung T(n,p): Entfaltung der Rekursion (1. Schritt)

Für $e > 0$ gilt:

$$T(n, e) = T\left(\frac{n}{2}, e - 1\right) + n$$

Ein Einsetzen ergibt:

$$T(n, e) = T\left(\frac{n}{4}, e - 2\right) + \frac{n}{2} + n$$

Noch ein weiteres Einsetzen:

$$T(n, e) = T\left(\frac{n}{8}, e - 3\right) + \frac{n}{4} + \frac{n}{2} + n$$

Nach e Schritten ergibt sich:

$$T(n, e) = T\left(\frac{n}{2^e}, 0\right) + n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{e-1}}\right)$$

Herleitung T(n,p): Auswertung der geometrischen Reihe

Die Summe

$$\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{e-1}}\right) = \sum_{i=0}^{e-1} \frac{1}{2^i}$$

ist eine geometrische Reihe mit Quotient $\frac{1}{2}$.
Sie ergibt:

$$\sum_{i=0}^{e-1} \frac{1}{2^i} = 2\left(1 - \frac{1}{2^e}\right)$$

Damit folgt:

$$T(n, e) = T\left(\frac{n}{2^e}, 0\right) + 2n\left(1 - \frac{1}{2^e}\right)$$

Herleitung T(n,p): Einsetzen des Basisfalls

Für $e = 0$ gilt:

$$T(n, 0) = 2T\left(\frac{n}{2}, 0\right) + n$$

Dies entspricht der Rekursion von sequenziellem Mergesort.
Bekannt ist:

$$T(n, 0) = n \log_2 n + n$$

Ersetzen von n durch $\frac{n}{2^e}$ ergibt:

$$T\left(\frac{n}{2^e}, 0\right) = \frac{n}{2^e} \log_2\left(\frac{n}{2^e}\right) + \frac{n}{2^e}$$

Einsetzen in $T(n, e)$ liefert:

$$T(n, e) = \frac{n}{2^e} \log_2\left(\frac{n}{2^e}\right) + \frac{n}{2^e} + 2n\left(1 - \frac{1}{2^e}\right)$$

Herleitung T(n,p): Substitution $p = 2^e$

Da gilt:

$$p = 2^e$$

folgt:

$$\frac{n}{2^e} = \frac{n}{p}$$

Damit ergibt sich:

$$T(n,p) = \frac{n}{p}\log_2\left(\frac{n}{p}\right) + \frac{n}{p} + 2n\left(1 - \frac{1}{p}\right)$$

Umsortiert erhält man:

$$T(n,p) = 2n\left(1 - \frac{1}{p}\right) + \frac{n}{p}\log_2\left(\frac{n}{p}\right) + \frac{n}{p}$$

Literatur I


Hartmut Ernst, Jochen Schmidt, and Gerd Beneken.
Datenstrukturen, Bäume und Graphen, pages 631–712.
Springer Fachmedien Wiesbaden, Wiesbaden, 2023.


Robert Sedgewick and Kevin Wayne.
Algorithmen.
Pearson Studium, München, 4 edition, 2014.
Deutsche Übersetzung der 4. englischen Auflage von 2011.


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.


Tobias Weinzierl.
Skalierungsmodelle und Kallibrieren von Messungen, pages 191–201.
Springer International Publishing, Cham, 2024.