Untersuchung der Skalierbarkeit von parallelem Sortieren auf einem Multicore-Prozessor

Verteidigung der Bachelorarbeit

Leon Zoerner

Informatik

18. Februar 2026

Agenda

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

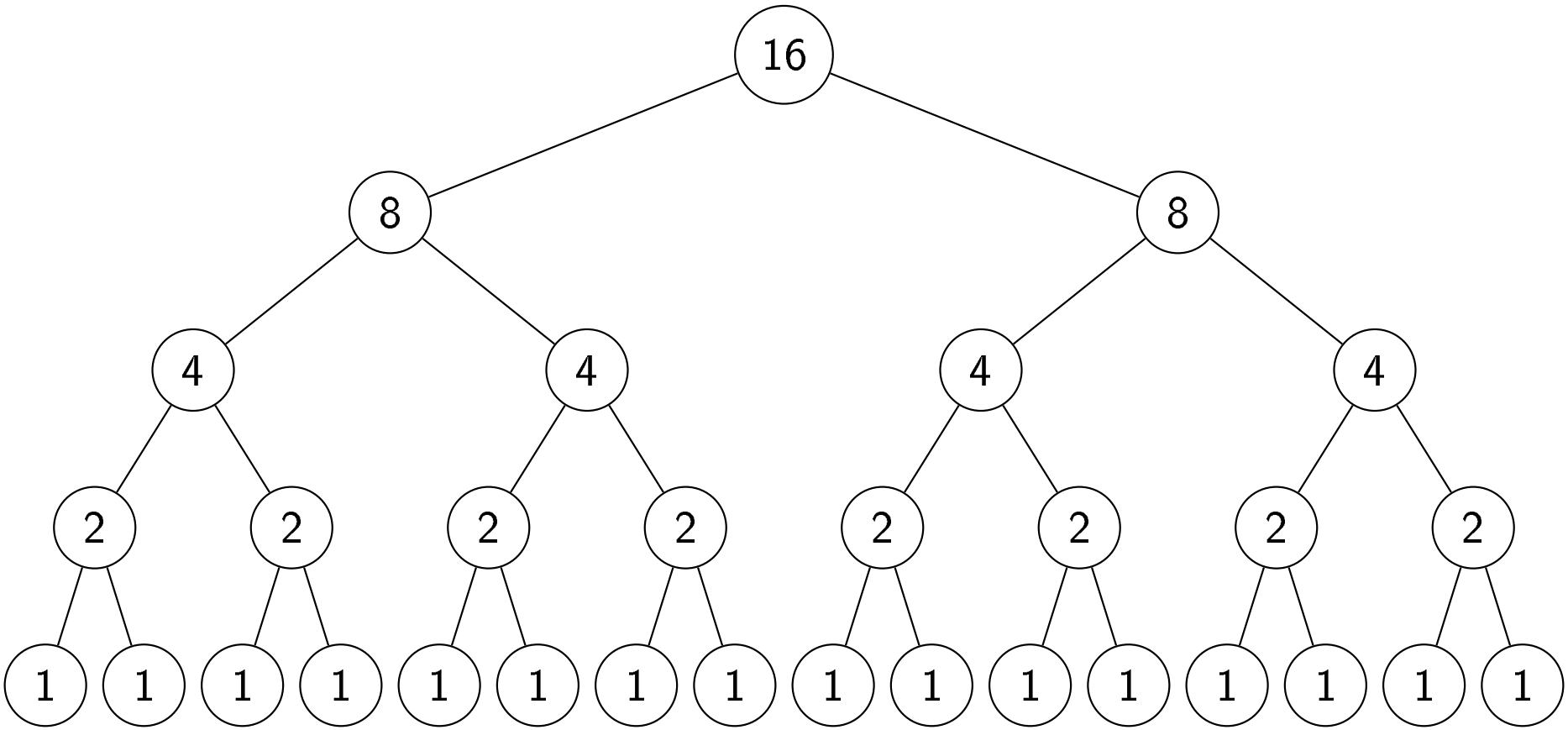
```
1         ...
2         // Sortiere liste
3         int i = 0;           // links
4         int j = lange - 1; // rechts
5         int k = links;       // links
6         while (i < j) {
7             if (listeB[i] < listeB[j]) {
8                 liste[k] = listeB[i];
9                 i++;
10            } 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  };
```

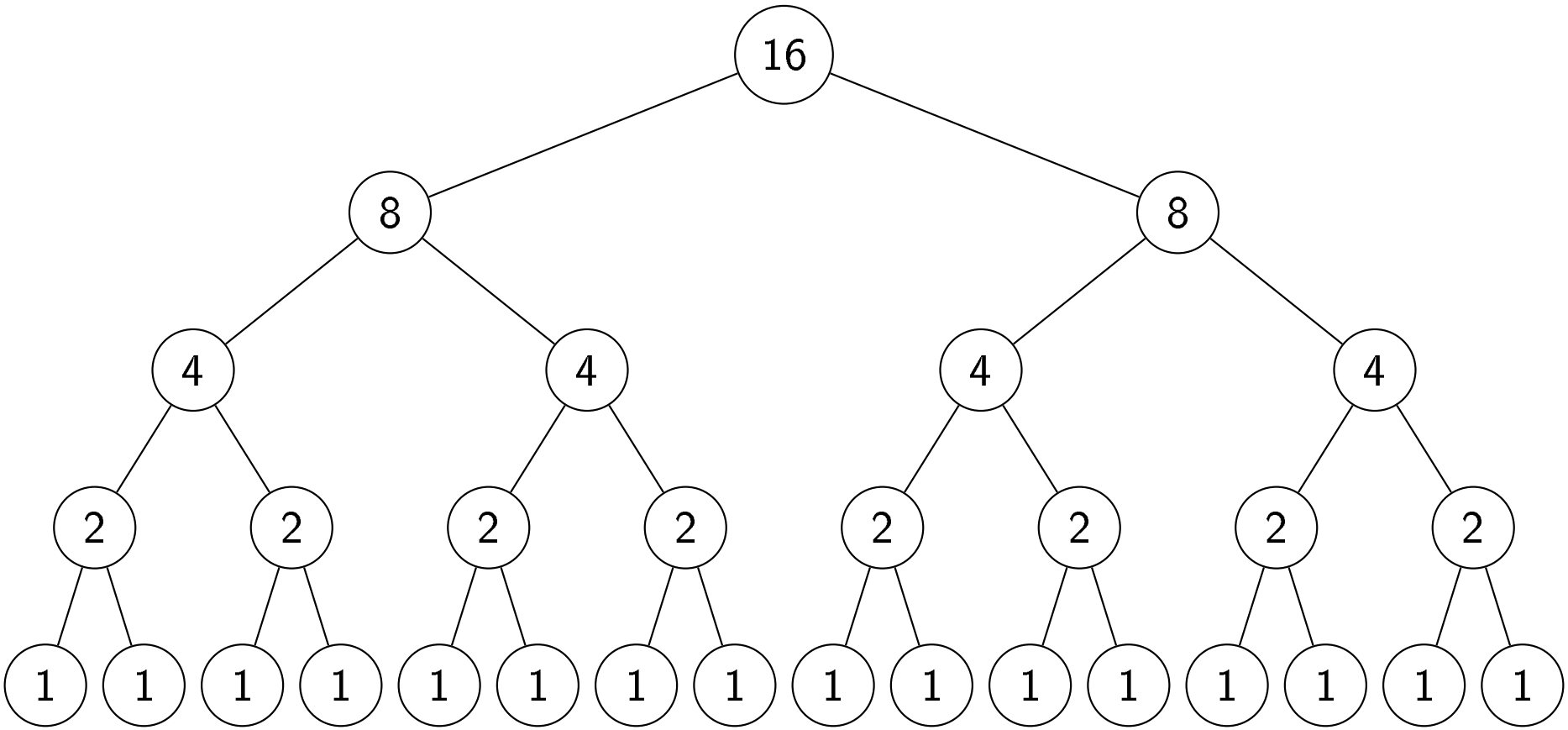
Sortieralgorithmen: Quicksort Best-Case

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:

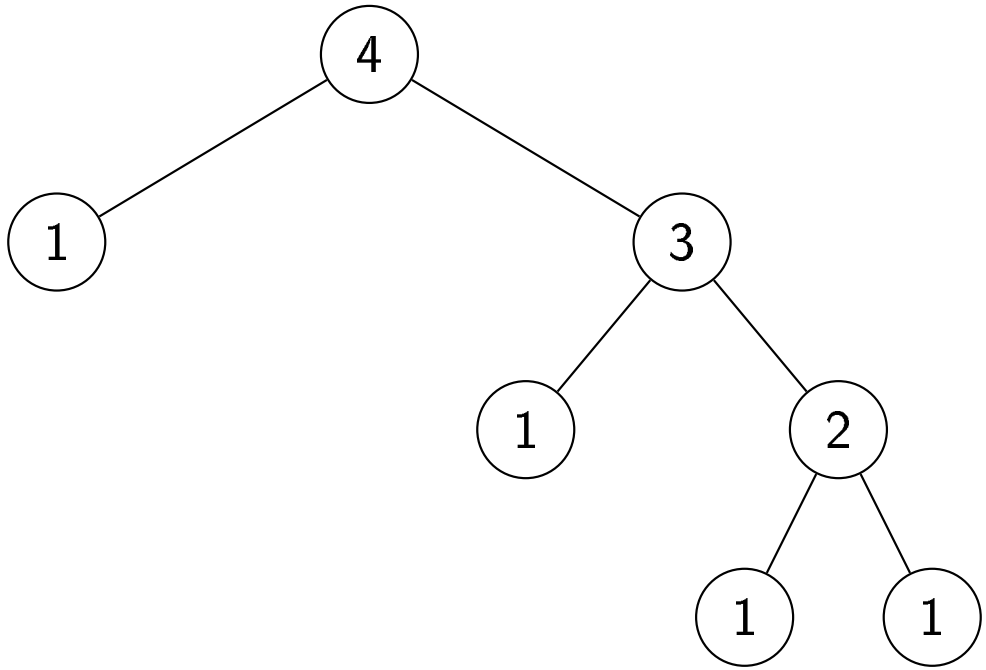
$$
\begin{aligned}
T(n) &= q_1 + q_2 + n \\
T(n) &= T(n-1) + 1 + n, \\
T(n) &= \frac{1}{2} \cdot (n^2 + n) + n, \\
O(T(n)) &= O(n^2).
\end{aligned}
$$

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

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

Sortieralgorithmen: Quicksort

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



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. [1]

- 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

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

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

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.

Messumgebung und Implementierung

Implementierungsvarianten Code

Laufzeitmessungen (sequenziell)

Analyse der Threading-Methoden

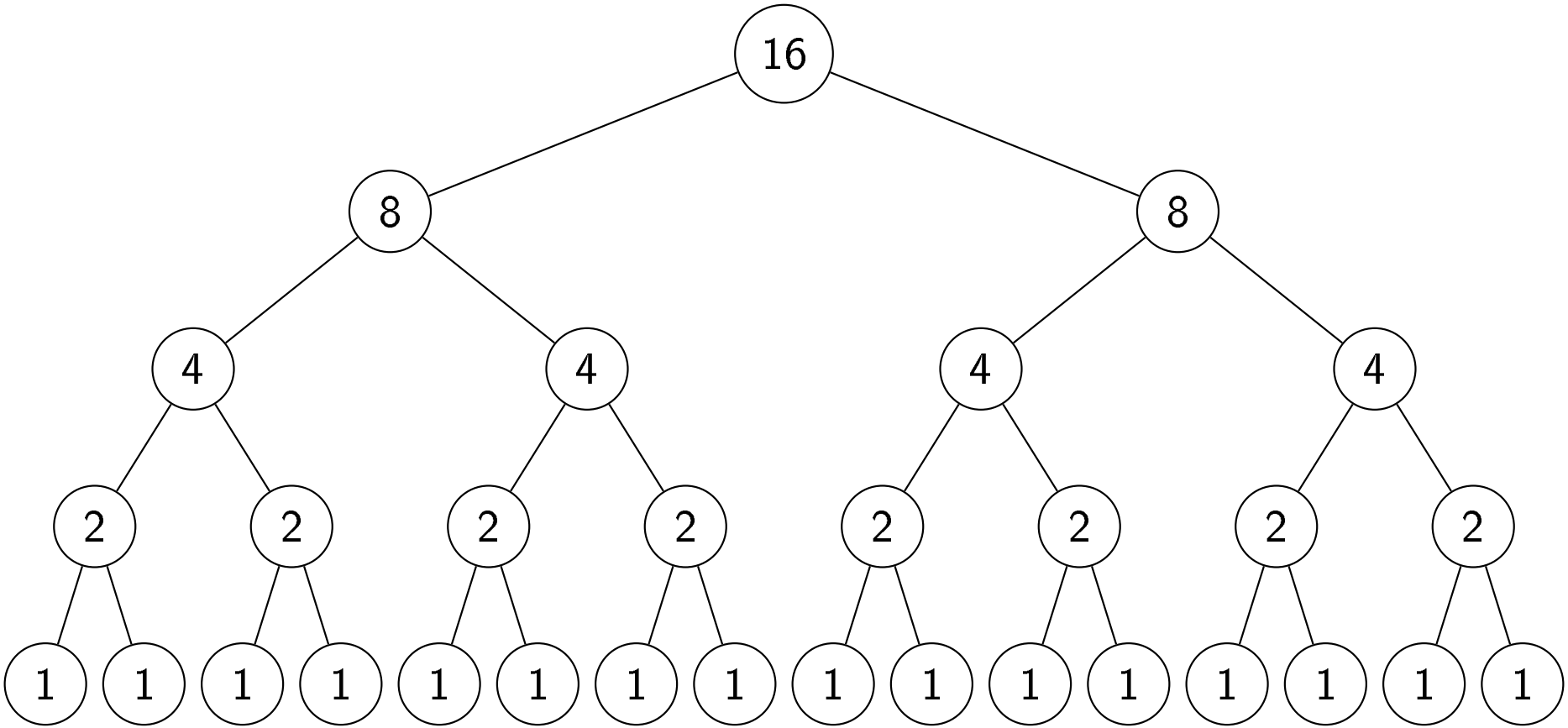
Beantwortung der Forschungsfrage

Zusammenfassung und Ausblick

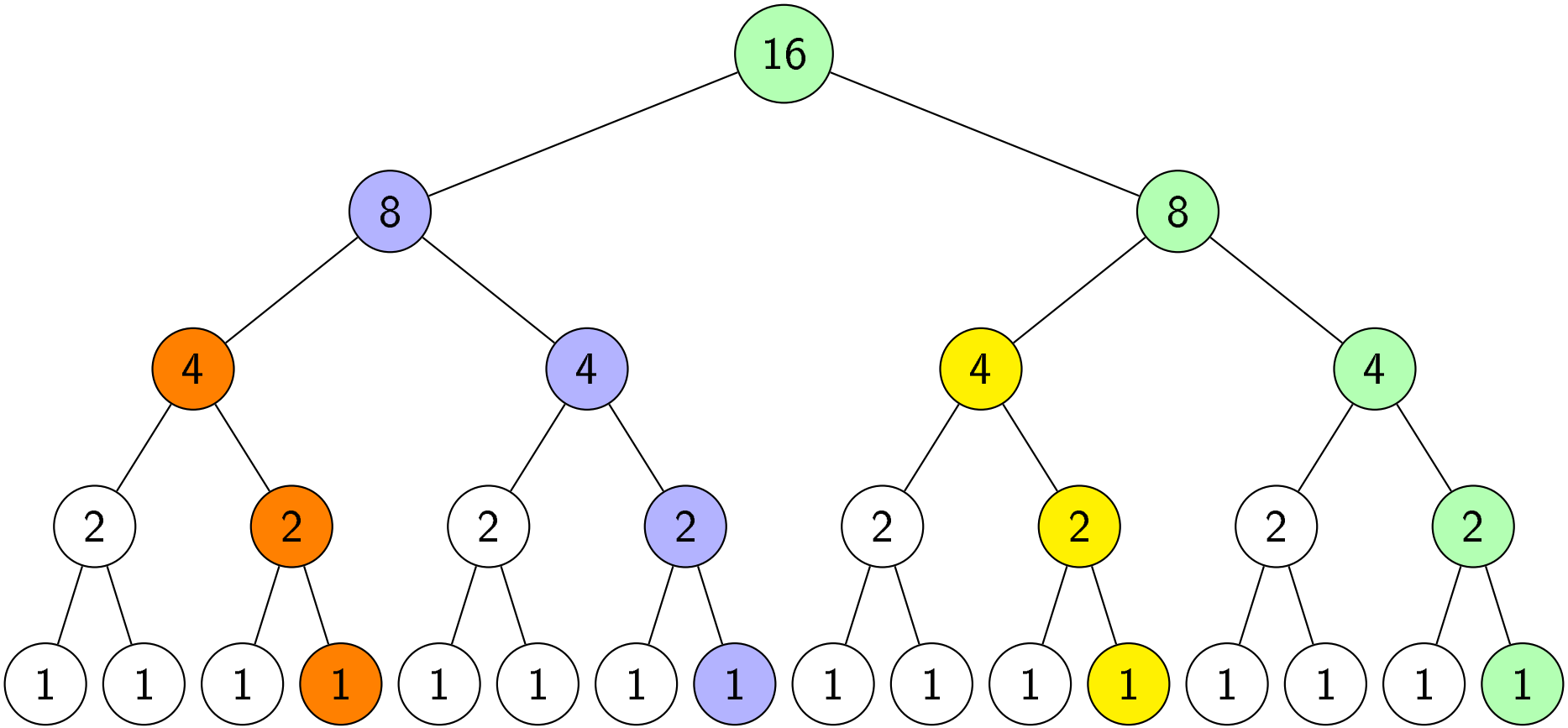
Vielen Dank für Ihre Aufmerksamkeit!

Fragen und Diskussion

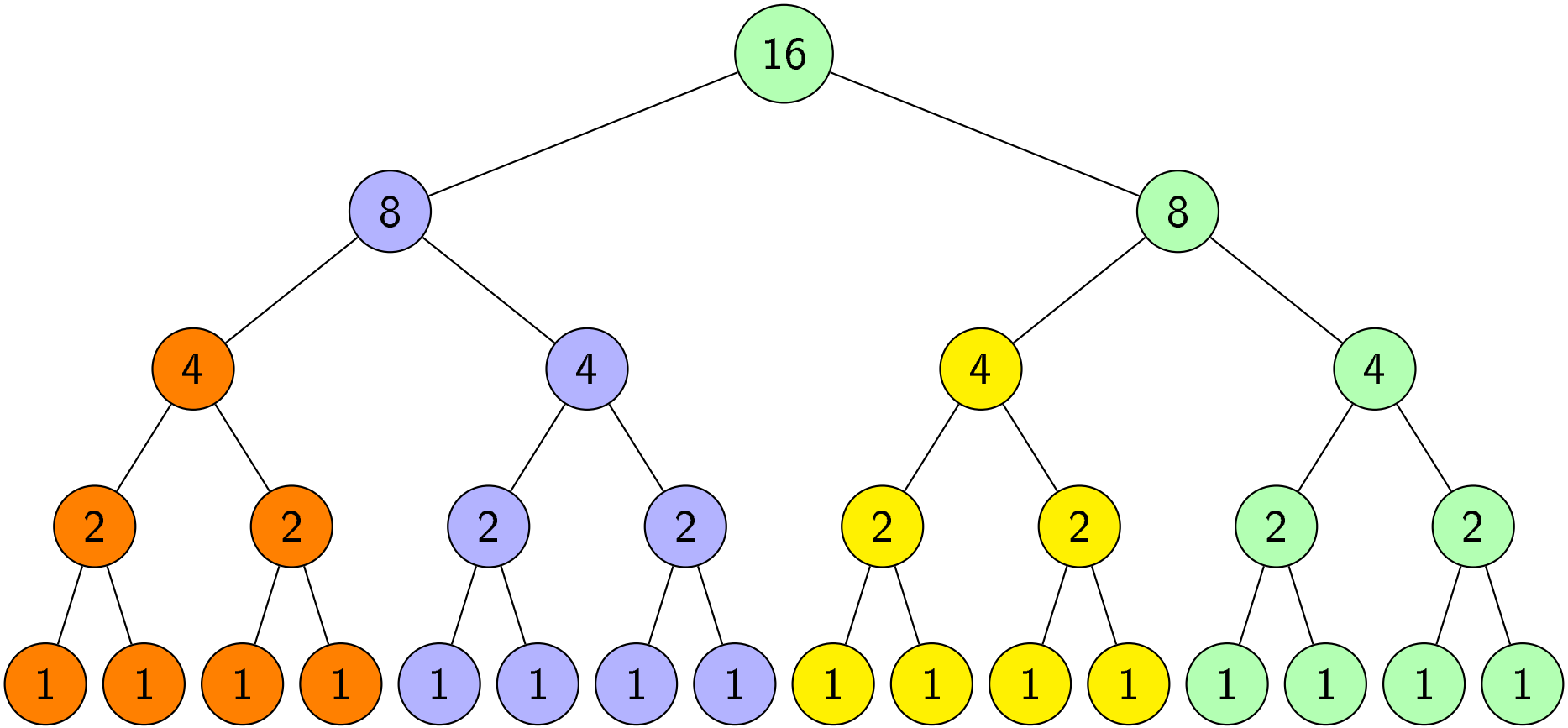
Balancierter Binärbaum für $n = 16$



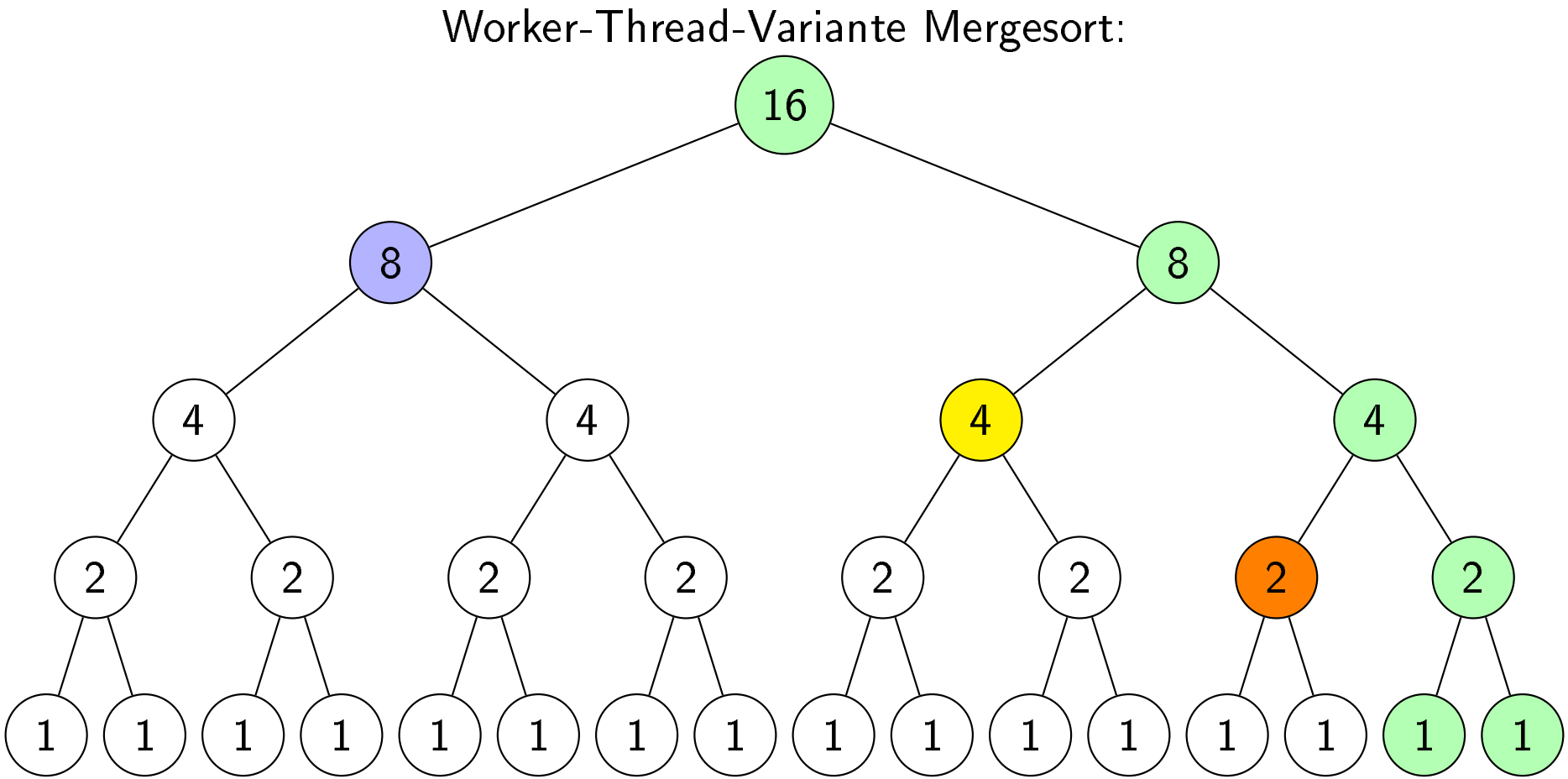
Balancierter Binärbaum für $n = 16$, $p = 4$, $e = 2$



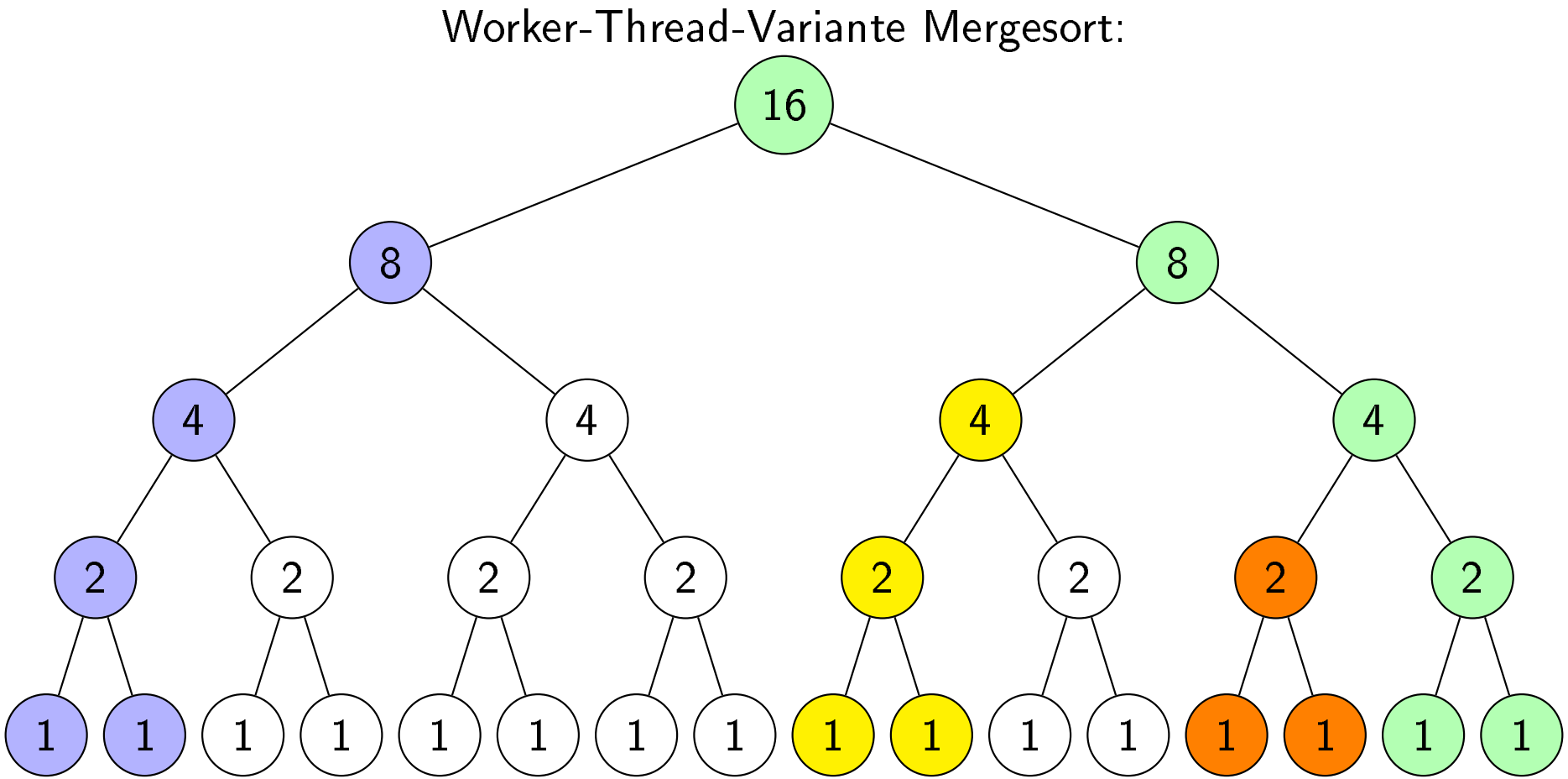
Balancierter Binärbaum für $n = 16$, $p = 4$, $e = 2$



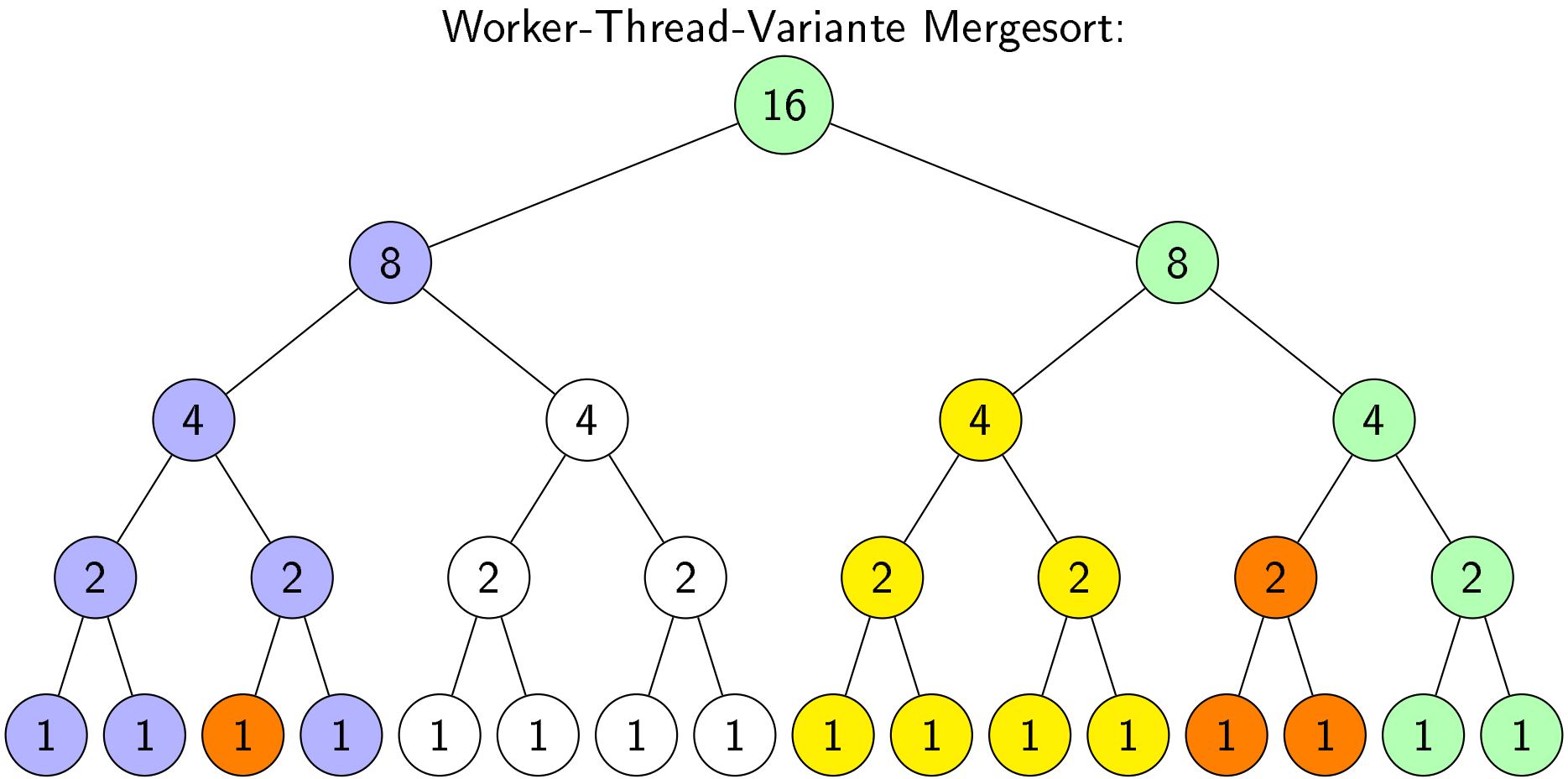
Balancierter Binärbaum für $n = 16$, $p = 4$



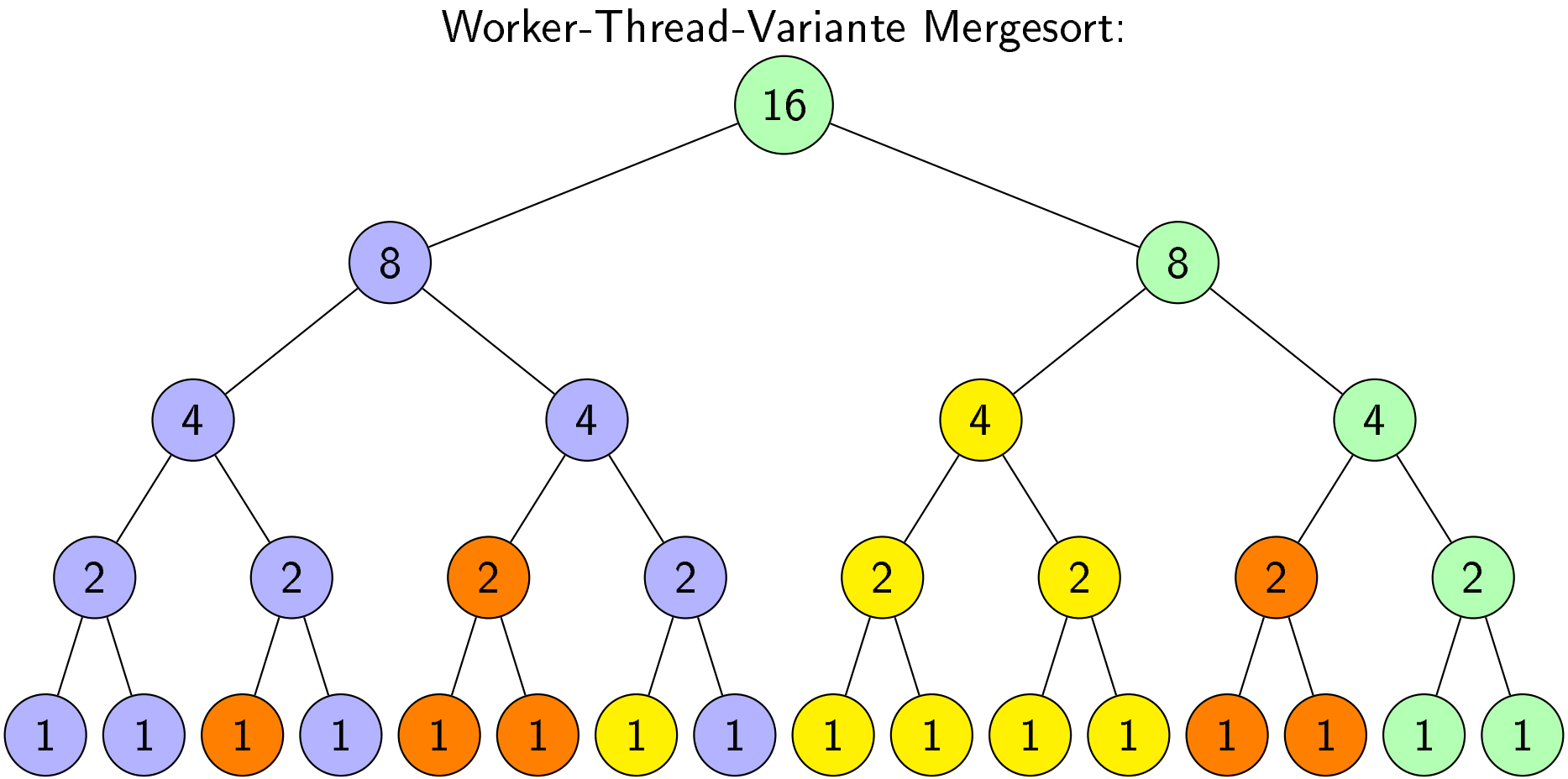
Balancierter Binärbaum für $n = 16$, $p = 4$



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

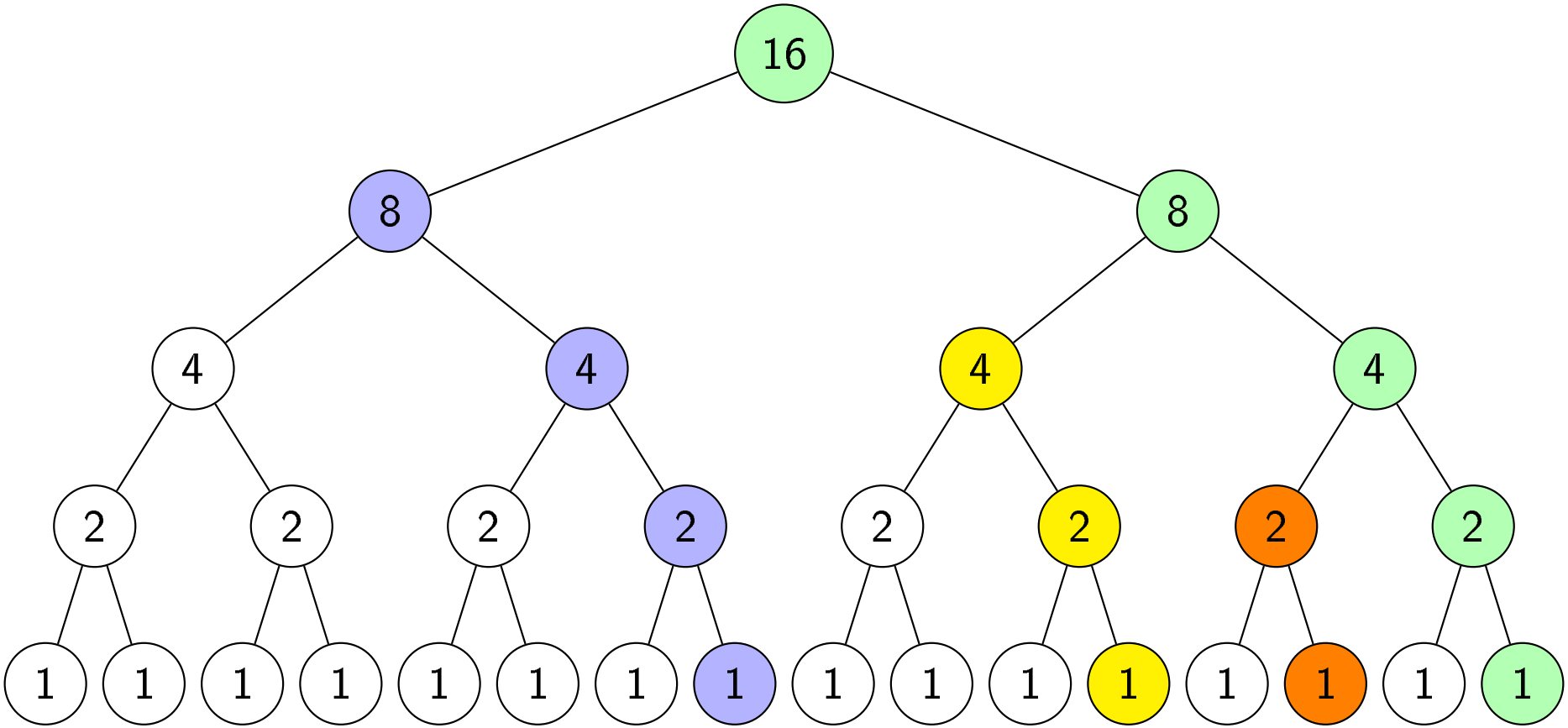


Balancierter Binärbaum für $n = 16$, $p = 4$



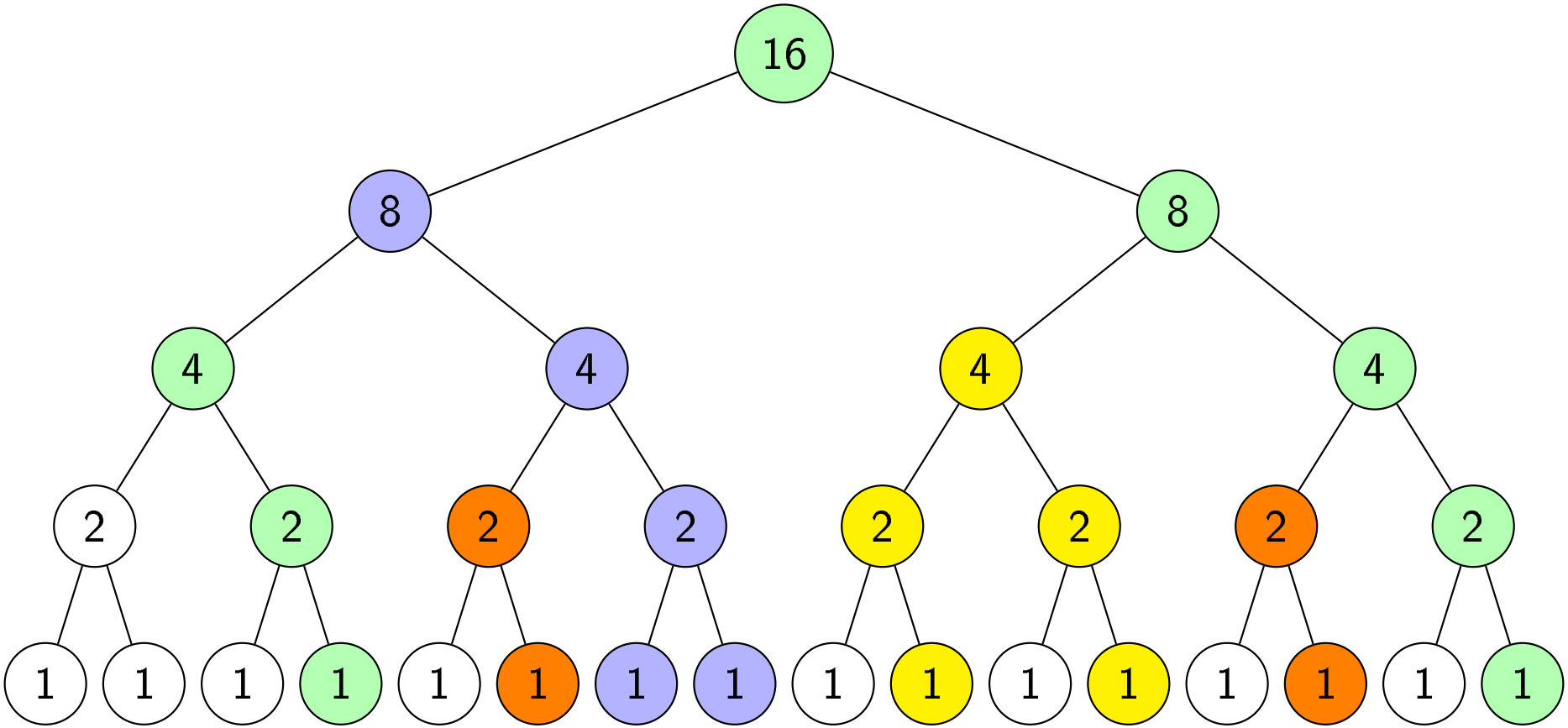
Balancierter Binärbaum für $n = 16$, $p = 4$

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



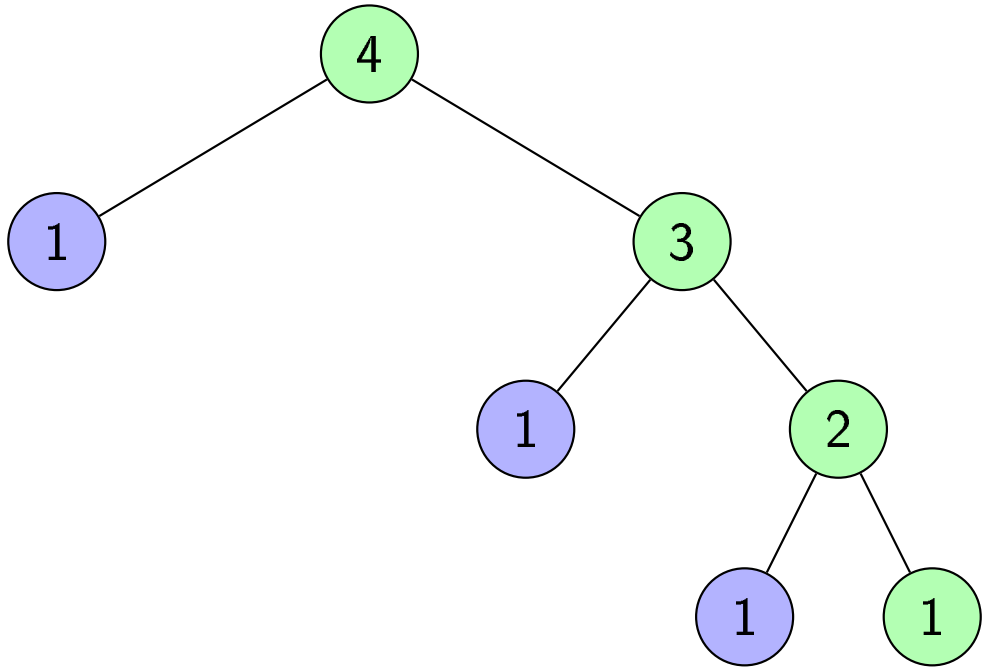
Balancierter Binärbaum für $n = 16$, $p = 4$

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



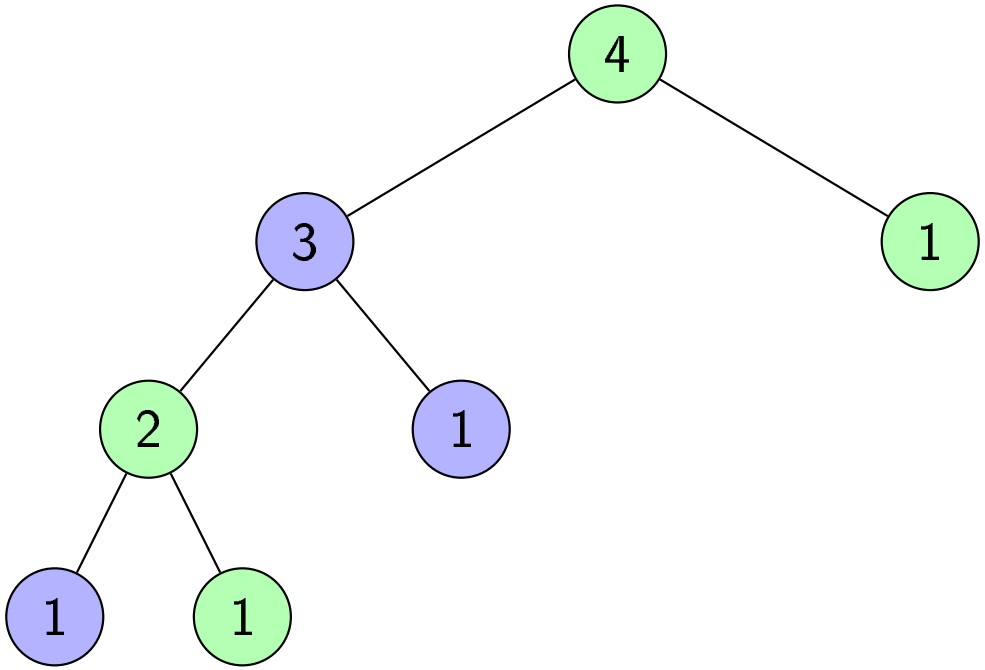
Maximal unbalancierter Binärbaum für $n = 4$, $p = 2$

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:

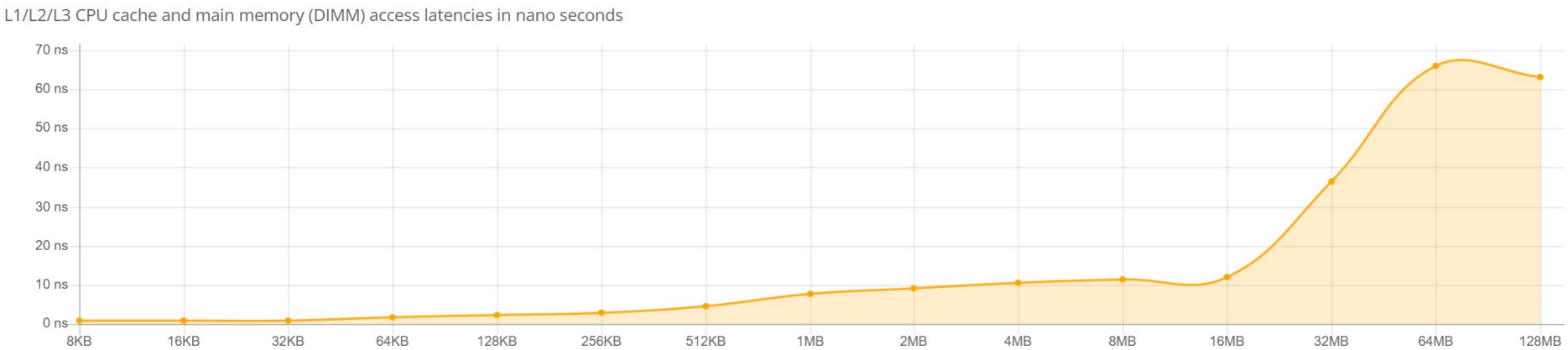
$$
\begin{aligned}
T(n) &= q_2 + n, \\
T(n) &= T(n-1) + n, \\
T(n) &= \frac{1}{2} \cdot (n^2 + n), \\
O(T(n)) &= O(n^2).
\end{aligned}
$$

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


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