

Untersuchung der Skalierbarkeit von parallelem Sortieren auf einem Multicore-Prozessor

Verteidigung der Bachelorarbeit

Leon Zoerner

Informatik

17. 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 Sorteralgorithmen die besten Laufzeiten erzielen.

Aus diesen Aspekten ergibt sich die zentrale Forschungsfrage dieser Arbeit:

Unter welchen Bedingungen liefern parallele Sorteralgorithmen anhand von Quicksort und Mergesort einen signifikanten Laufzeitvorteil gegenüber der sequentiellen Ausführung, und welche Threadingstrategien führen dabei zur besten Laufzeit?

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.

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

Sortieralgorithmen: Mergesort

```
1 void Mergesort::mischen(int *liste, int links, const int
2     mitte, const int rechts, const int lange) {
3     int *listeB = new int[lange];
4
5     // Kopiere nach listeB
6     for (int i = links; i < mitte + 1; i++) {
7         listeB[i - links] = liste[i];
8     }
9     for (int i = mitte + 1; i < rechts + 1; i++) {
10        listeB[lange - 1 + mitte + 1 - i] = liste[i];
11    }
12
13    // Sortiere liste
14    int i = 0;           // links
15    int j = lange - 1; // rechts
16    int k = links;      // links
17    while (i < j) {
18        if (listeB[i] < listeB[j]) {
19            liste[k] = listeB[i];
20            i++;
21        } else {
22            liste[k] = listeB[j];
23            j--;
24        }
25        k++;
26    }
27}
```

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

$$T(n) = m_1 + m_2 + n$$

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n$$

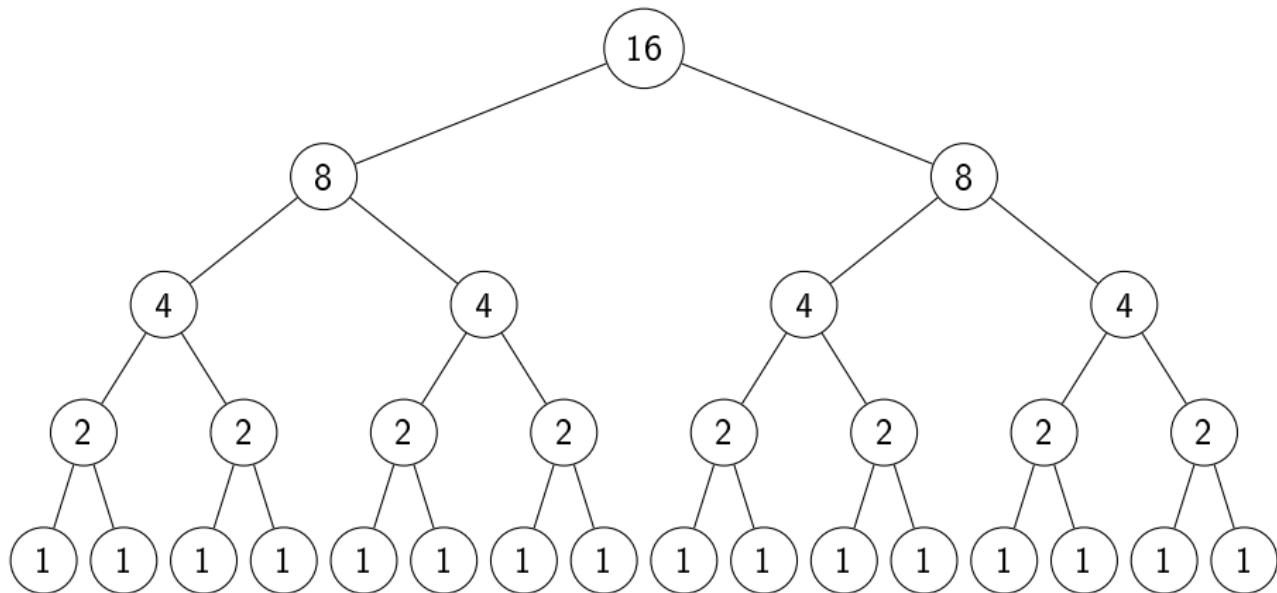
$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n,$$

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

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

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

Sortieralgorithmen: Quicksort

```
1 void Quicksort::partitioniere(int *liste, const int
2     links, const int rechts, int &ml, int &mr) {
3     int i = links;
4     int j = rechts;
5     int mitte = links + ((rechts - links) / 2);
6     int p = liste[mitte];
7     while (i <= j) {
8         while (liste[i] < p) {
9             i++;
10        }
11        while (liste[j] > p) {
12            j--;
13        }
14        if (i <= j) {
15            vertausche(liste, i, j);
16            i++;
17            j--;
18        }
19    };
20    ml = j; mr = i;
21 }.
```

Sortieralgorithmen: Quicksort

```
1 void Quicksort::vertausche(int *liste, const int a,
2     const int b) {
3     int temp = liste[a];
4     liste[a] = liste[b];
5     liste[b] = temp;
6 }
```

Sortieralgorithmen: Quicksort

Best-Case von Quicksort:

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

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n$$

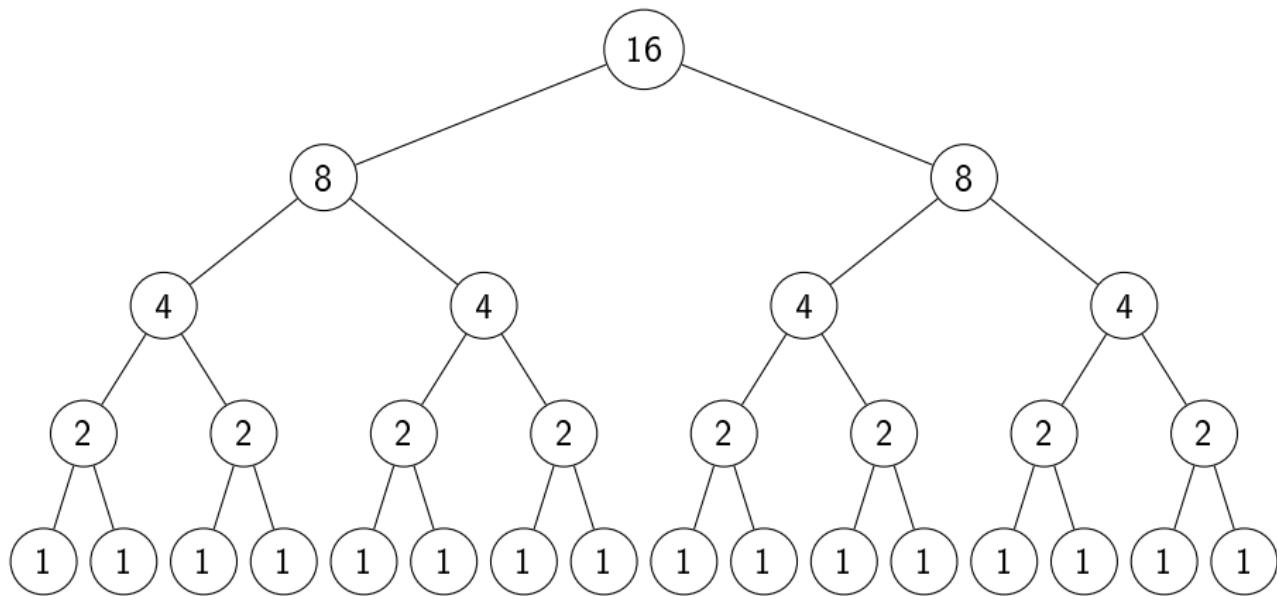
$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n,$$

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

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

Sortieralgorithmen: Quicksort

Balancierter Binärbaum für $n = 16$



Sortieralgorithmen: Quicksort

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

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

$$T(n) = T(n - 1) + 1 + n,$$

$$T(n) = \frac{1}{2} \cdot (n^2 + n) + n,$$

$$O(T(n)) = O(n^2).$$

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

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

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

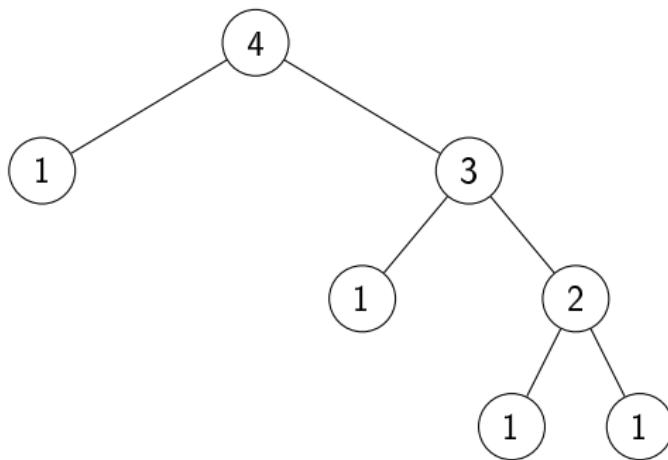
$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$$

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

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

Sortieralgorithmen: Quicksort

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



Messumgebung und Implementierung

Implementierungsvarianten Code

Laufzeitmessungen (sequenziell)

Analyse der Threading-Methoden

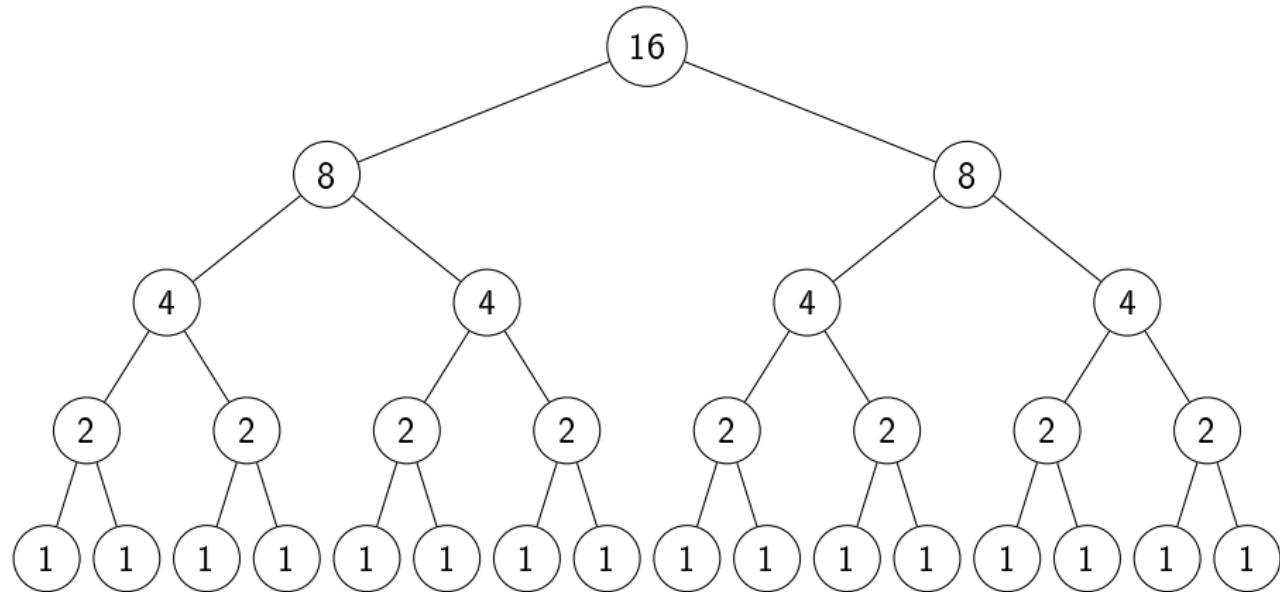
Beantwortung der Forschungsfrage

Zusammenfassung und Ausblick

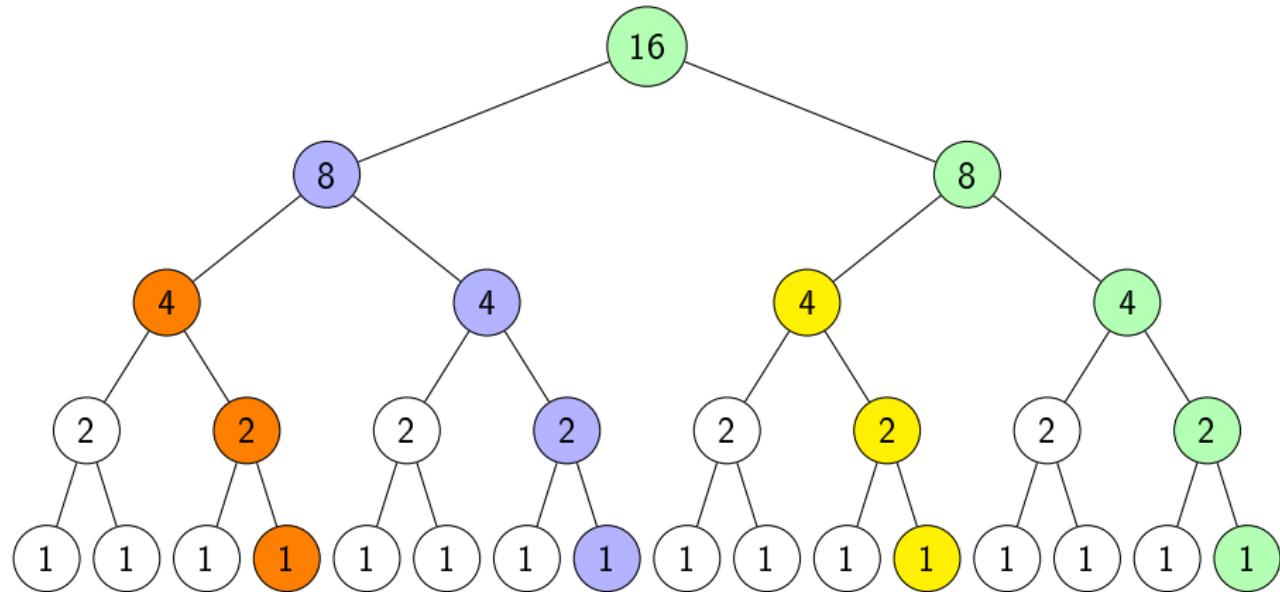
Vielen Dank für Ihre
Aufmerksamkeit!

Fragen und Diskussion

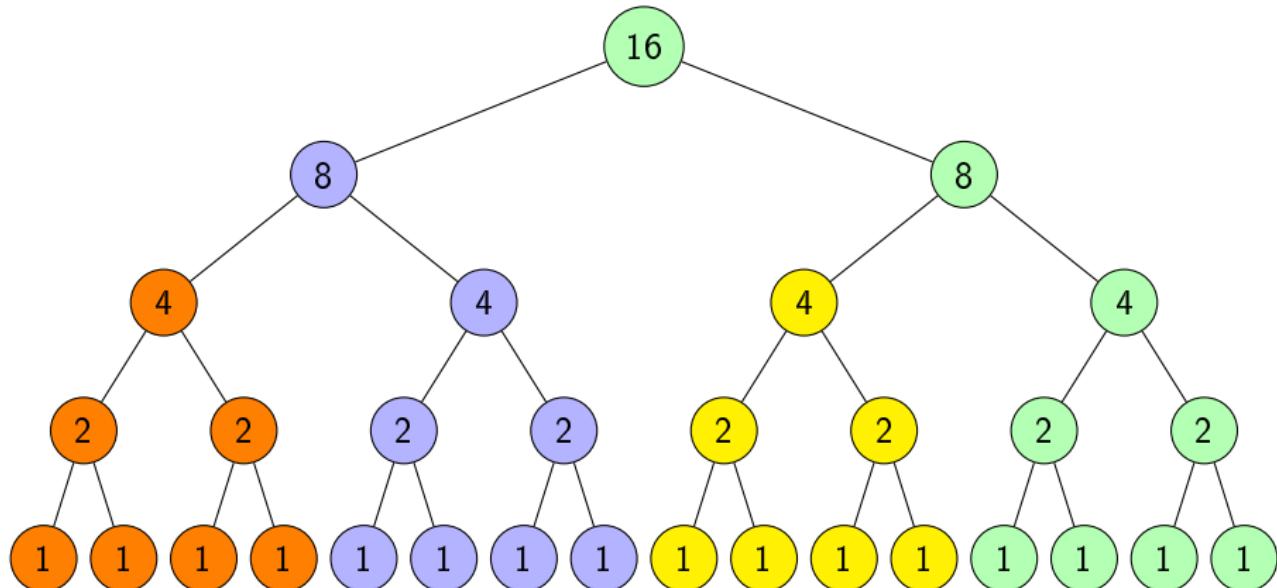
Balancierter Binärbaum für $n = 16$



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

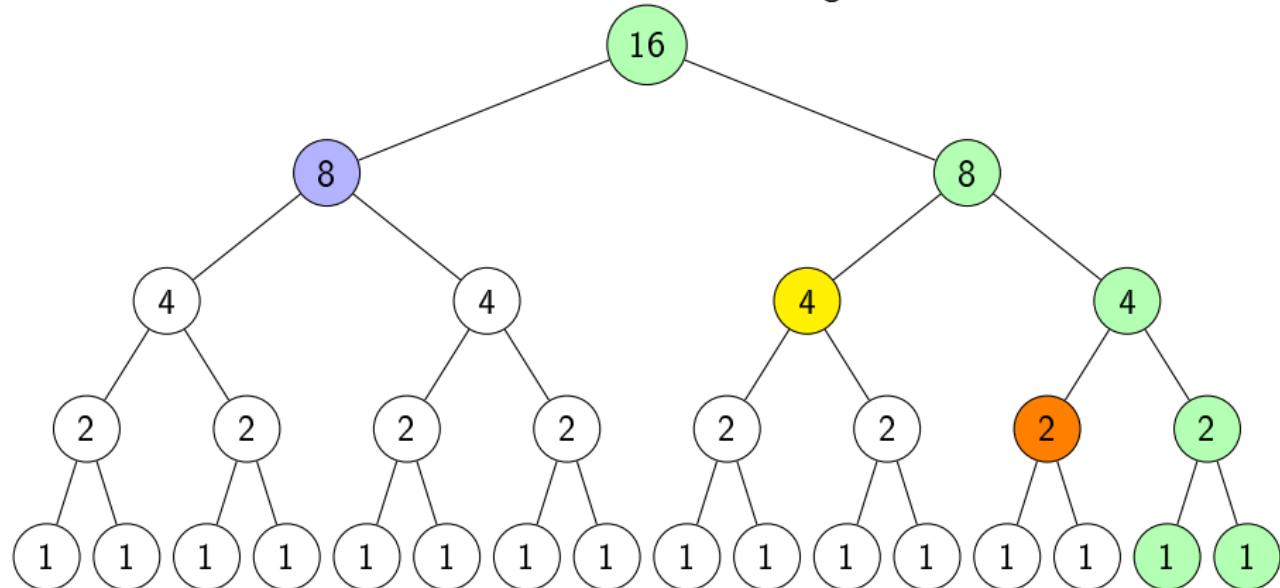


Balancierter Binärbaum für $n = 16$, $p = 4$, $e = 2$



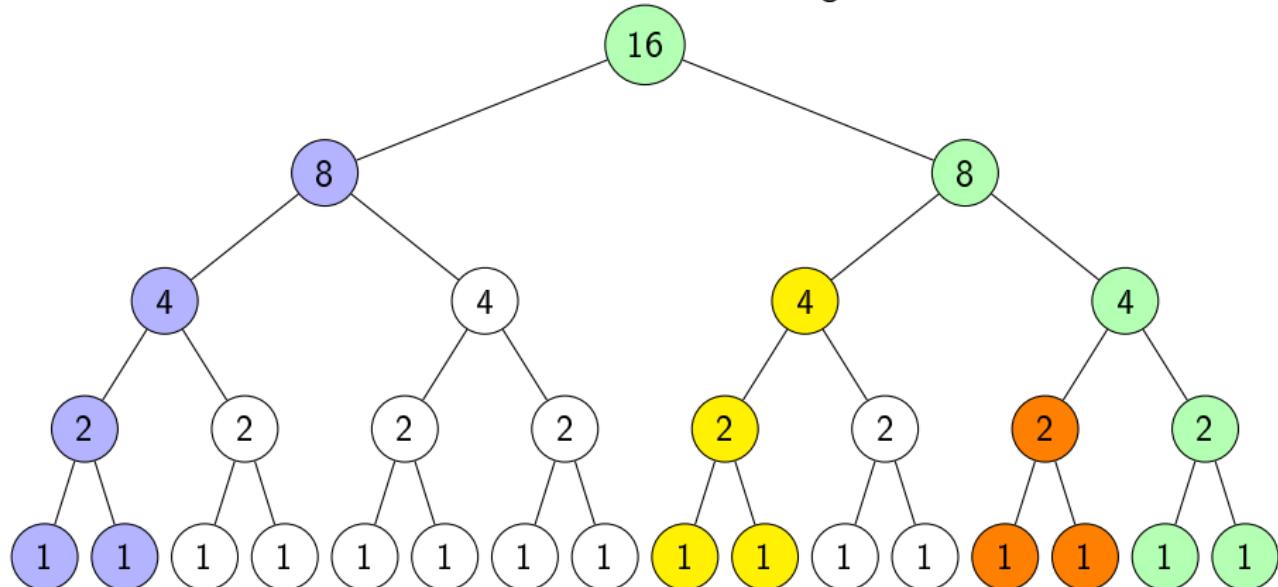
Balancierter Binärbaum für $n = 16$, $p = 4$

Worker-Thread-Variante Mergesort:



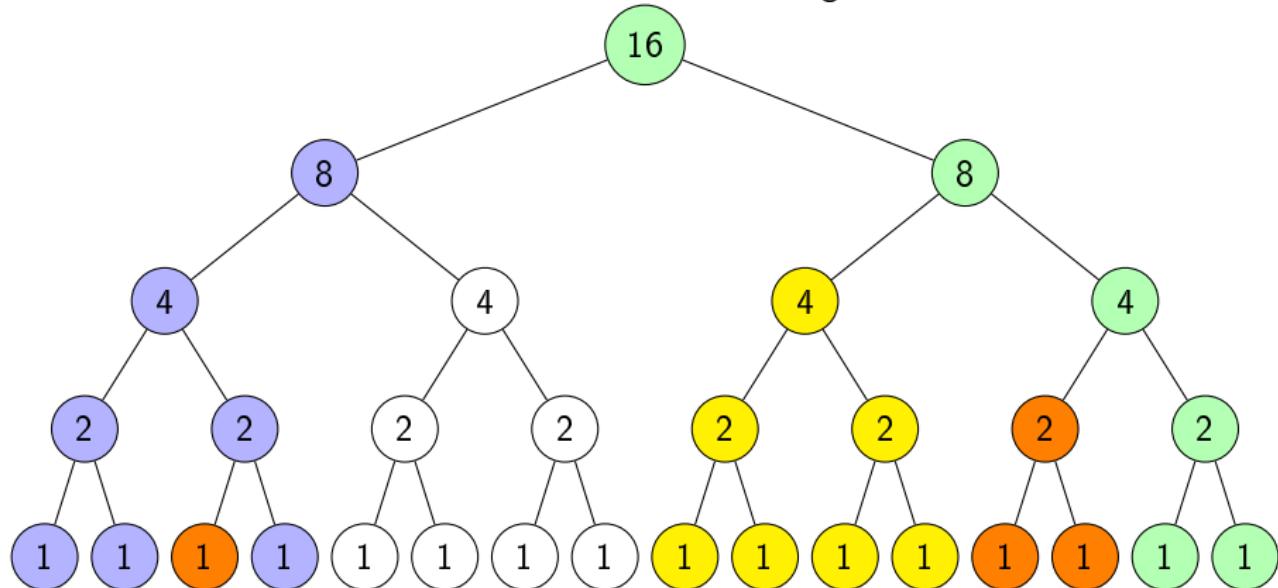
Balancierter Binärbaum für $n = 16$, $p = 4$

Worker-Thread-Variante Mergesort:



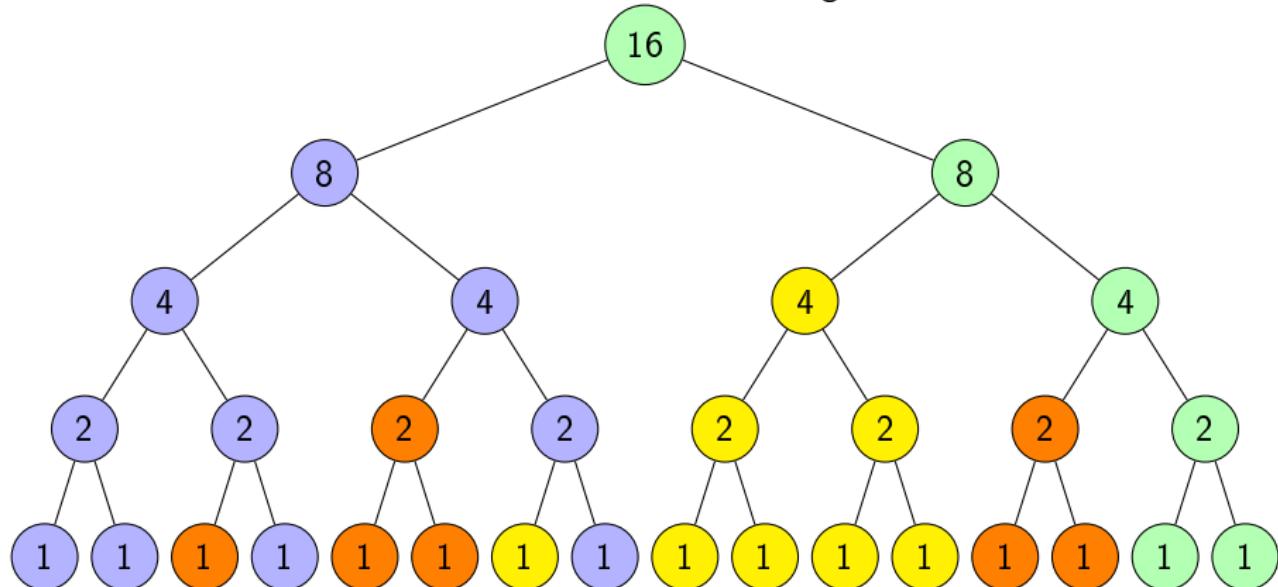
Balancierter Binärbaum für $n = 16$, $p = 4$

Worker-Thread-Variante Mergesort:



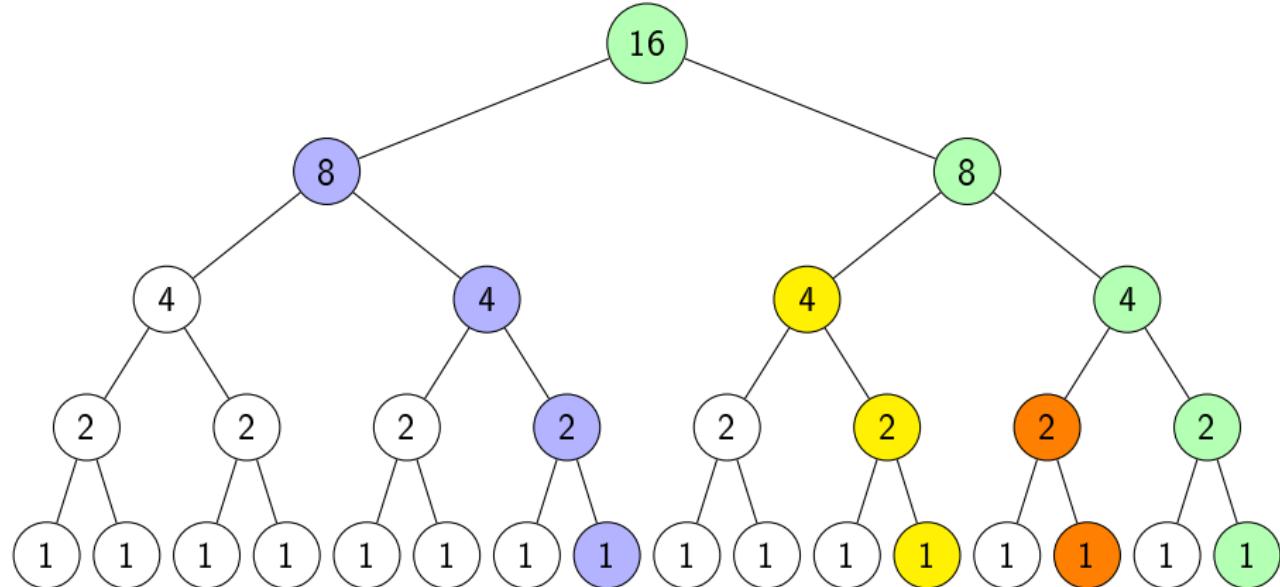
Balancierter Binärbaum für $n = 16$, $p = 4$

Worker-Thread-Variante Mergesort:



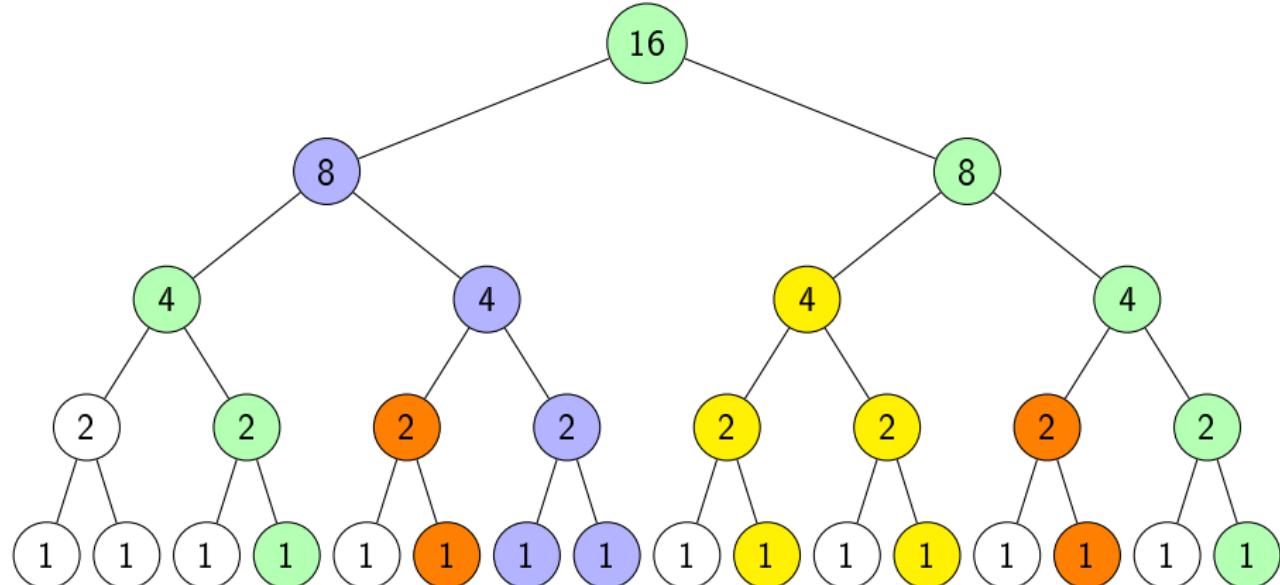
Balancierter Binärbaum für $n = 16$, $p = 4$

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

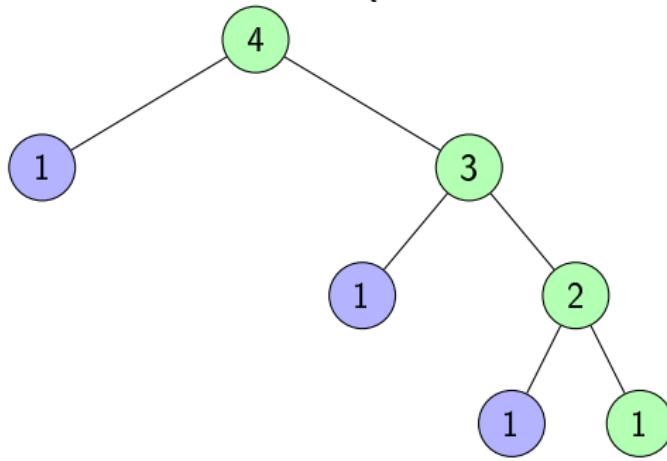


Balancierter Binärbaum für $n = 16$, $p = 4$

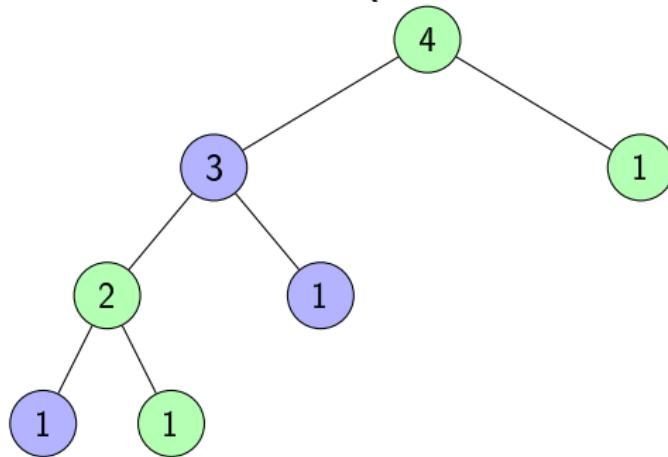
Worker-Thread-Variante Quicksort **Best-Case:**



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



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



Formeln: Mergesort

$$T(n) = m_1 + m_2 + n$$

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n$$

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

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

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

Formeln: Quicksort

Best-Case von Quicksort:

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

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n$$

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

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

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

Formeln: Quicksort

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

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

$$T(n) = T(n-1) + 1 + n,$$

$$T(n) = \frac{1}{2} \cdot (n^2 + n) + n,$$

$$O(T(n)) = O(n^2).$$

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

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

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

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

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

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

Formeln: parallel Mergesort

p = 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:

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

$$T(n) = T(n - 1) + n,$$

$$T(n) = \frac{1}{2} \cdot (n^2 + n),$$

$$O(T(n)) = O(n^2).$$

Formeln: Einheiten und Skalierung

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

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

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

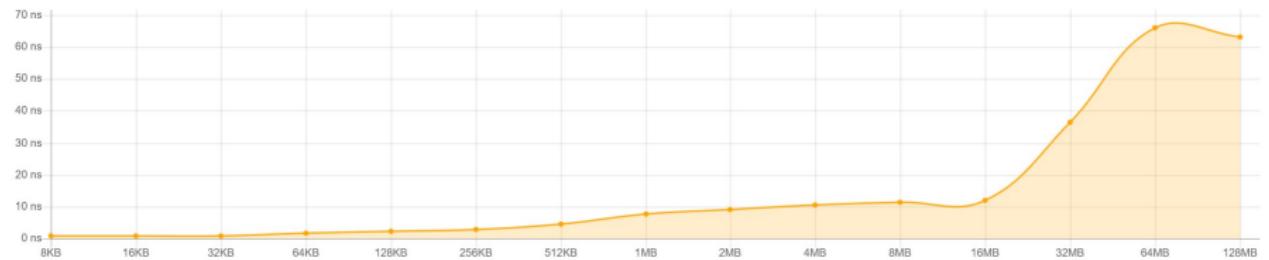
Aufgrund von Overheads wie z. B. erhöhte Speicherlatenzen durch Cache-Misses:

$$x = f(n, p)$$

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

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

L1/L2/L3 CPU cache and main memory (DIMM) access latencies in nano seconds



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:

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