

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

Bachelorarbeit

Studiengang: Informatik

Bearbeiter: Leon Zoerner

10. Dezember 2025

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>2</b>
1.1	Motivation . . . . .	2
1.2	Zielsetzung und Forschungsfrage . . . . .	2
1.3	Threading anhand des einfachen Beispiels Inkrement-Array erklärt . .	2
<b>2</b>	<b>Theoretische Grundlagen</b>	<b>4</b>
2.1	Sortieralgorithmen: Quicksort und Mergesort . . . . .	4
2.2	Grundlagen der Parallelisierung . . . . .	4
2.3	Thread-Modelle, Overheads und Skalierungsgrenzen . . . . .	4
<b>3</b>	<b>Methodik und Versuchsaufbau</b>	<b>5</b>
3.1	Messumgebung und Hardware . . . . .	5
3.2	Implementierungsvarianten . . . . .	5
3.3	Messmethodik . . . . .	5
<b>4</b>	<b>Ergebnisse und Analyse</b>	<b>6</b>
4.1	Grundlegende Laufzeiten abhängig von der Arraygröße . . . . .	6
4.1.1	Messziel . . . . .	6
4.1.2	Erwartung . . . . .	6
4.1.3	Diagramm . . . . .	6
4.1.4	Analyse und Interpretation . . . . .	6
4.2	Einfluss des Listentyps . . . . .	7
4.2.1	Messziel . . . . .	7
4.2.2	Erwartung . . . . .	7
4.2.3	Diagramm . . . . .	7
4.2.4	Analyse und Interpretation . . . . .	7
4.3	Einfluss der Arraygröße im Detail . . . . .	8
4.3.1	Messziel . . . . .	8
4.3.2	Erwartung . . . . .	8
4.3.3	Diagramm . . . . .	8
4.3.4	Analyse und Interpretation . . . . .	8
4.4	Tiefenbasierte Thread-Erzeugung . . . . .	9
4.4.1	Messziel . . . . .	9
4.4.2	Erwartung . . . . .	9
4.4.3	Diagramm . . . . .	9
4.4.4	Analyse und Interpretation . . . . .	9
4.5	Workerthreads . . . . .	10
4.5.1	Messziel . . . . .	10
4.5.2	Erwartung . . . . .	10
4.5.3	Diagramm . . . . .	10
4.5.4	Analyse und Interpretation . . . . .	10
4.6	Vergleich der Threading-Methoden . . . . .	11
4.6.1	Messziel . . . . .	11
4.6.2	Erwartung . . . . .	11
4.6.3	Diagramm . . . . .	11
4.6.4	Analyse und Interpretation . . . . .	11
4.7	Einfluss des Datentyps der Liste . . . . .	12

4.7.1	Messziel . . . . .	12
4.7.2	Erwartung . . . . .	12
4.7.3	Diagramm . . . . .	12
4.7.4	Analyse und Interpretation . . . . .	12
<b>5</b>	<b>Diskussion und Fazit</b>	<b>13</b>
5.1	Interpretation aller Ergebnisse . . . . .	13
5.2	Beantwortung der Forschungsfrage . . . . .	13
5.3	Zusammenfassung . . . . .	13
<b>6</b>	<b>Anhang</b>	<b>14</b>
6.1	Hardware-Spezifikationen . . . . .	14
6.2	Code . . . . .	14

# 1 Einleitung

## 1.1 Motivation

Ziel dieser Arbeit ist es, die Grenzen von Threads und Parallelisierung aufzuzeigen. Dabei soll insbesondere untersucht werden, wie groß der Overhead durch Threads ist und welchen Performanceunterschied es macht, bereits initialisierte Workerthreads zu verwenden, im Vergleich zur Erstellung neuer Threads. Da sich für diese Untersuchungen ein geeigneter, leicht verständlicher und programmierbarer Anwendungsfall anbietet, habe ich mich für Sortieralgorithmen entschieden, die sich zudem sehr gut parallelisieren lassen.

## 1.2 Zielsetzung und Forschungsfrage

Ziel dieser Bachelorarbeit ist die systematische Analyse der Laufzeitentwicklung paralleler Sortierverfahren. Dabei soll untersucht werden, wie sich parallele Implementierungen von Quicksort und Mergesort im Vergleich zu ihren sequentiellen Varianten verhalten. Im Fokus stehen insbesondere folgende Punkte:

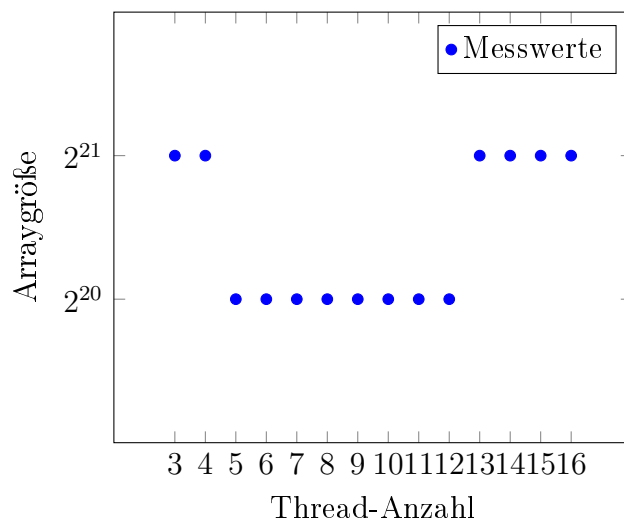
- der Einfluss verschiedener Threadingstrategien auf die Laufzeit,
- die Frage, ab welcher Eingangsgröße und bei welcher Anzahl von Threads ein messbarer Geschwindigkeitsvorteil entsteht,
- sowie die Identifikation von Thread-Management-Techniken, die für Sortieralgorithmen die besten Laufzeiten erzielen.

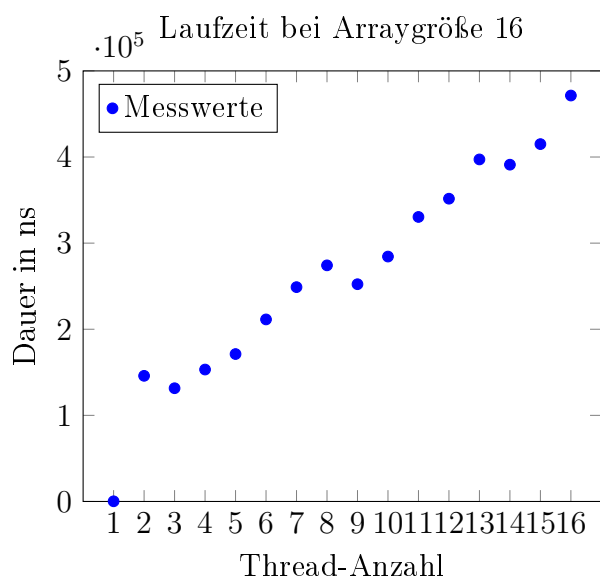
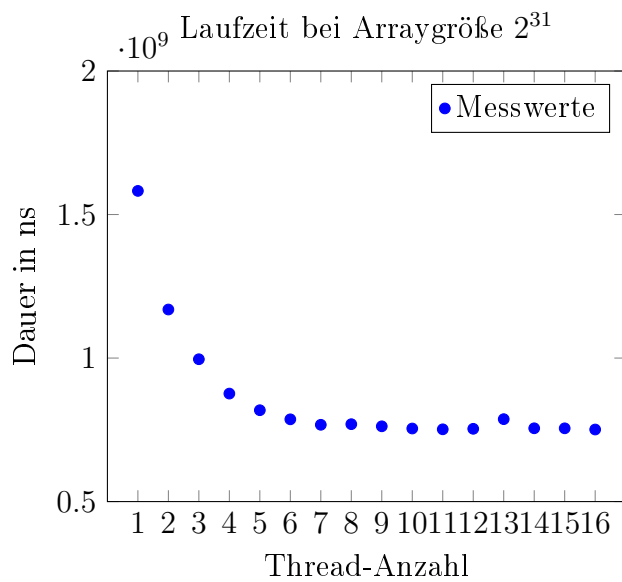
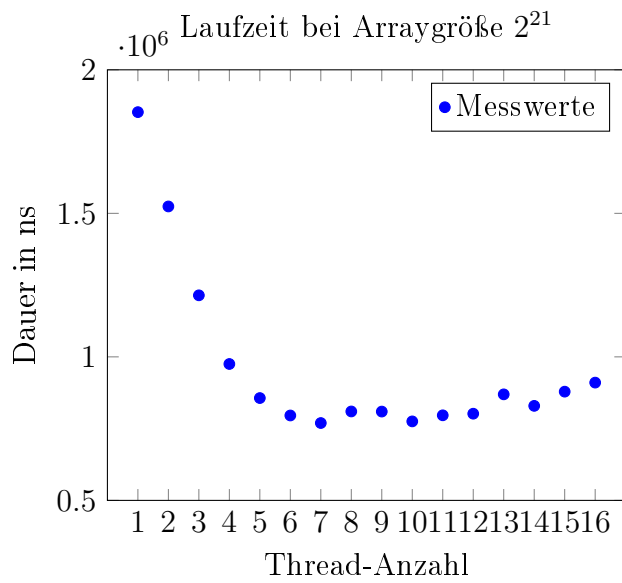
Aus diesen Aspekten ergibt sich die zentrale Forschungsfrage dieser Arbeit:

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

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

Zeitpunkt der 50% Geschwindigkeitssteigerung (incArray)





## 2 Theoretische Grundlagen

### 2.1 Sortieralgorithmen: Quicksort und Mergesort

### 2.2 Grundlagen der Parallelisierung

### 2.3 Thread-Modelle, Overheads und Skalierungsgrenzen

## **3 Methodik und Versuchsaufbau**

### **3.1 Messumgebung und Hardware**

### **3.2 Implementierungsvarianten**

### **3.3 Messmethodik**

## 4 Ergebnisse und Analyse

### 4.1 Grundlegende Laufzeiten abhängig von der Arraygröße

#### 4.1.1 Messziel

#### 4.1.2 Erwartung

#### 4.1.3 Diagramm

#### 4.1.4 Analyse und Interpretation



## **4.2 Einfluss des Listentyps**

### **4.2.1 Messziel**

### **4.2.2 Erwartung**

### **4.2.3 Diagramm**

### **4.2.4 Analyse und Interpretation**

## **4.3 Einfluss der Arraygröße im Detail**

### **4.3.1 Messziel**

### **4.3.2 Erwartung**

### **4.3.3 Diagramm**

### **4.3.4 Analyse und Interpretation**

## **4.4 Tiefenbasierte Thread-Erzeugung**

### **4.4.1 Messziel**

### **4.4.2 Erwartung**

### **4.4.3 Diagramm**

### **4.4.4 Analyse und Interpretation**

## **4.5 Workerthreads**

### **4.5.1 Messziel**

### **4.5.2 Erwartung**

### **4.5.3 Diagramm**

### **4.5.4 Analyse und Interpretation**

## **4.6 Vergleich der Threading-Methoden**

### **4.6.1 Messziel**

### **4.6.2 Erwartung**

### **4.6.3 Diagramm**

### **4.6.4 Analyse und Interpretation**

## **4.7 Einfluss des Datentyps der Liste**

### **4.7.1 Messziel**

### **4.7.2 Erwartung**

### **4.7.3 Diagramm**

### **4.7.4 Analyse und Interpretation**

## 5 Diskussion und Fazit

### 5.1 Interpretation aller Ergebnisse

### 5.2 Beantwortung der Forschungsfrage

### 5.3 Zusammenfassung

## 6 Anhang

### 6.1 Hardware-Spezifikationen

Zur besseren Einordnung der Leistungsfähigkeit der verwendeten Hardware befindet sich unter folgendem Link ein Benchmark:

<https://www.userbenchmark.com/UserRun/70984567>

Ich liste aber jetzt hier auch nochmal die relevanten Hardwarekomponenten auf.

CPU: AMD Ryzen 7 5800X, 8C/16T, 3.80-4.70GHz

CPU Kühler: be quiet! Dark Rock Pro 4)

RAM: G.Skill Aegis UDIMM 16GB Kit, DDR4-3200, CL16-18-18-38 (2 Kits, insgesamt 4x8 GB = 32 GB)

Betriebssystem: Windows 10 Version 22H2

### 6.2 Code

Der gesamte Quellcode dieser Arbeit ist öffentlich unter folgendem Link verfügbar:

<https://github.com/Leon333M/Sortiervverfahren>

Listing 1: Quicksort

```
1 // Quicksort.cpp
2 #include "Quicksort.h"
3 #include "WorkerPool.h"
4 #include <chrono>
5 #include <thread>
6 #include <vector>
7
8 Quicksort::Quicksort() {};
9
10 void Quicksort::sortG(int *liste, int lange) {
11     int links = 0;
12     int rechts = lange - 1;
13     quicksort(liste, links, rechts);
14 };
15
16 void Quicksort::sortM(int *liste, int lange, int messEbene) {
17     int links = 0;
18     int rechts = lange - 1;
19     quicksort(liste, links, rechts, 1, messEbene);
20 };
21
22 void Quicksort::sortP(int *liste, int lange, int
    neueThreadsBisEbene) {
23     int links = 0;
24     int rechts = lange - 1;
25     // int neueThreadsBisEbene = static_cast<int>(std::ceil(
26         std::log2(static_cast<double>(anzahlThreads) + 1.0)));
27     quicksortP(liste, links, rechts, 1, neueThreadsBisEbene);
28 };
29
```



```

29 void Quicksort::sortPM(int *liste, int lange, int
    neueThreadsBisEbene, int messEbene) {
30     int links = 0;
31     int rechts = lange - 1;
32     quicksortP(liste, links, rechts, 1, neueThreadsBisEbene,
        messEbene);
33 };
34
35 void Quicksort::sortW(int *liste, const int lange, const int
    workerThreads) {
36     int links = 0;
37     int rechts = lange - 1;
38     quicksortW(liste, links, rechts, workerThreads);
39 };
40
41 void Quicksort::quicksort(int *liste, const int links, const
    int rechts) {
42     if (links < rechts) {
43         int ml, mr;
44         partitioniere(liste, links, rechts, ml, mr);
45         quicksort(liste, links, ml);
46         quicksort(liste, mr, rechts);
47     }
48 };
49
50 void Quicksort::quicksort(int *liste, const int links, const
    int rechts, const int aktuelleEbene, const int messEbene) {
51     if (aktuelleEbene == messEbene) {
52         quicksortM(liste, links, rechts, aktuelleEbene);
53     } else {
54         if (links < rechts) {
55             int ml, mr;
56             partitioniere(liste, links, rechts, ml, mr);
57             quicksort(liste, links, ml, aktuelleEbene + 1,
                messEbene);
58             quicksort(liste, mr, rechts, aktuelleEbene + 1,
                messEbene);
59         }
60     }
61 };
62
63 void Quicksort::quicksortM(int *liste, const int links, const
    int rechts, const int aktuelleEbene) {
64     Messdaten *pos = new Messdaten();
65     pos->start1 = std::chrono::high_resolution_clock::now();
66     if (links < rechts) {
67         int ml, mr;
68         partitioniere(liste, links, rechts, ml, mr);
69         pos->start2 = std::chrono::high_resolution_clock::now
            ();
70         quicksort(liste, links, ml);

```

```

71         quicksort(liste, mr, rechts);
72         pos->ende2 = std::chrono::high_resolution_clock::now()
73         ;
74     }
75     pos->ende1 = std::chrono::high_resolution_clock::now();
76     Messdaten::addMessDaten(aktuelleEbene, pos);
77 };
78 void Quicksort::quicksortP(int *liste, const int links, const
79     int rechts, const int aktuelleEbene, const int
80     neueThreadsBisEbene) {
81     if (aktuelleEbene < neueThreadsBisEbene) {
82         if (links < rechts) {
83             int ml, mr;
84             partitioniere(liste, links, rechts, ml, mr);
85             // quicksort(liste, links, ml);
86             std::thread thread(static_cast<void (*)(&int *,
87                 const int, const int, const int, const int)>(&
88                 Quicksort::quicksortP),
89                 liste, links, ml, aktuelleEbene
90                 + 1, neueThreadsBisEbene);
91             // quicksort(liste, mr, rechts);
92             quicksortP(liste, mr, rechts, aktuelleEbene + 1,
93                 neueThreadsBisEbene);
94             thread.join();
95         }
96     } else {
97         quicksort(liste, links, rechts);
98     }
99 };
100 void Quicksort::quicksortP(int *liste, const int links, const
101     int rechts, const int aktuelleEbene, const int
102     neueThreadsBisEbene, const int messEbene) {
103     if (aktuelleEbene < neueThreadsBisEbene) {
104         if (aktuelleEbene == messEbene) {
105             quicksortPM(liste, links, rechts, aktuelleEbene,
106                 neueThreadsBisEbene);
107         } else {
108             if (links < rechts) {
109                 int ml, mr;
110                 partitioniere(liste, links, rechts, ml, mr);
111                 // quicksort(liste, links, ml);
112                 std::thread thread(
113                     static_cast<void (*)(&int *, const int,
114                         const int, const int, const int, const
115                         int)>(&Quicksort::quicksortP),
116                     liste, links, ml, aktuelleEbene + 1,
117                     neueThreadsBisEbene, messEbene);
118                 // quicksort(liste, mr, rechts);
119                 quicksortP(liste, mr, rechts, aktuelleEbene +

```

```

109         1, neueThreadsBisEbene, messEbene);
110         thread.join();
111     }
112 } else {
113     quicksort(liste, links, rechts, aktuelleEbene,
114               messEbene);
115 }
116 };
117 void Quicksort::quicksortPM(int *liste, const int links, const
118                             int rechts, const int aktuelleEbene, const int
119                             neueThreadsBisEbene) {
120     Messdaten *pos = new Messdaten();
121     pos->start1 = std::chrono::high_resolution_clock::now();
122     if (links < rechts) {
123         int ml, mr;
124         partitioniere(liste, links, rechts, ml, mr);
125         // quicksort(liste, links, ml);
126         pos->start2 = std::chrono::high_resolution_clock::now
127             ();
128         std::thread thread(
129             static_cast<void (*)(>(int *, const int, const int,
130                                   const int, const int)>(&Quicksort::quicksortP),
131             liste, links, ml, aktuelleEbene + 1,
132             neueThreadsBisEbene);
133         ;
134         // quicksort(liste, mr, rechts);
135         quicksortP(liste, mr, rechts, aktuelleEbene + 1,
136                   neueThreadsBisEbene);
137         thread.join();
138         pos->ende2 = std::chrono::high_resolution_clock::now()
139             ;
140     }
141     pos->ende1 = std::chrono::high_resolution_clock::now();
142     Messdaten::addMessDaten(aktuelleEbene, pos);
143 };
144
145 void Quicksort::quicksortW(int *liste, int links, int rechts,
146                             int workerThreads) {
147     WorkerPool pool(workerThreads);
148
149     pool.taskHandler = [](int *liste, int links, int rechts,
150                           WorkerPool &pool) {
151         if (links < rechts) {
152             if (rechts - links < Sortierverfahren::mindestLange
153                 ) {
154                 quicksort(liste, links, rechts);
155             } else {
156                 int ml, mr;
157                 Quicksort::partitioniere(liste, links, rechts,

```

```

148         ml, mr);
149         pool.addTask({liste, links, ml});
150         pool.taskHandler(liste, mr, rechts, pool);
151     }
152 };
153
154 pool.addTask({liste, links, rechts});
155 pool.waitUntilDone();
156 }
157
158 void Quicksort::partitioniere(int *liste, const int links,
159     const int rechts, int &ml, int &mr) {
160     int i = links;
161     int j = rechts;
162     int mitte = links + ((rechts - links) / 2);
163     int p = liste[mitte];
164     while (i <= j) {
165         while (liste[i] < p) {
166             i++;
167         }
168         while (liste[j] > p) {
169             j--;
170         }
171         if (i <= j) {
172             vertausche(liste, i, j);
173             i++;
174             j--;
175         }
176     };
177     ml = j;
178     mr = i;
179 };
180
181 void Quicksort::vertausche(int *liste, const int a, const int
182     b) {
183     int temp = liste[a];
184     liste[a] = liste[b];
185     liste[b] = temp;
186 };

```

Listing 2: Quicksort WorkerPool

```

1 // WorkerPool.h
2 #include <atomic>
3 #include <condition_variable>
4 #include <functional>
5 #include <mutex>
6 #include <queue>
7 #include <thread>
8 #include <vector>
9

```

```

10 struct Task {
11     int *liste;
12     int links;
13     int rechts;
14 };
15
16 class WorkerPool {
17 private:
18     std::vector<std::thread> threads;
19     std::queue<Task> taskQueue;
20     std::mutex mutex;
21     std::condition_variable cv;
22     std::atomic<int> activeTasks;
23     bool finished;
24
25 public:
26     std::function<void(int *, int, int, WorkerPool &)>
        taskHandler;
27
28 public:
29     WorkerPool(int numThreads) : finished(false), activeTasks
        (0) {
30         for (int i = 0; i < numThreads; ++i)
31             threads.emplace_back(&WorkerPool::worker, this);
32     }
33
34     ~WorkerPool() {
35         {
36             std::lock_guard<std::mutex> lock(mutex);
37             finished = true;
38         }
39         cv.notify_all();
40         for (auto &t : threads)
41             t.join();
42     }
43
44     void addTask(const Task &task) {
45         {
46             std::lock_guard<std::mutex> lock(mutex);
47             taskQueue.push(task);
48             activeTasks++;
49         }
50         cv.notify_one();
51     }
52
53     void waitUntilDone() {
54         std::unique_lock<std::mutex> lock(mutex);
55         cv.wait(lock, [this] { return activeTasks == 0; });
56     }
57
58 private:

```

```

59     void worker() {
60         while (true) {
61             Task task;
62             {
63                 std::unique_lock<std::mutex> lock(mutex);
64                 cv.wait(lock, [this] { return finished || !
                    taskQueue.empty(); });
65
66                 if (finished && taskQueue.empty())
67                     return;
68                 if (taskQueue.empty())
69                     continue;
70
71                 task = taskQueue.front();
72                 taskQueue.pop();
73             }
74
75             // Task bearbeiten
76             if (taskHandler) {
77                 taskHandler(task.liste, task.links, task.
                    rechts, *this);
78             }
79
80             activeTasks--;
81             cv.notify_all();
82         }
83     }
84 };

```

Listing 3: Mergesort

```

1 // Mergesort.cpp
2 #include "Mergesort.h"
3 #include "MergeWorkerPool.h"
4 #include <memory>
5 #include <thread>
6 #include <vector>
7
8 Mergesort::Mergesort() {};
9
10 void Mergesort::sortG(int *liste, int lange) {
11     int links = 0;
12     int rechts = lange - 1;
13     mergesort(liste, links, rechts);
14 };
15
16 void Mergesort::sortM(int *liste, int lange, int messEbene) {
17     int links = 0;
18     int rechts = lange - 1;
19     mergesort(liste, links, rechts, 1, messEbene);
20 };
21

```

```

22 void Mergesort::sortP(int *liste, int lange, int
    neueThreadsBisEbene) {
23     int links = 0;
24     int rechts = lange - 1;
25     mergesortP(liste, links, rechts, 1, neueThreadsBisEbene);
26 };
27
28 void Mergesort::sortPM(int *liste, int lange, int
    neueThreadsBisEbene, int messEbene) {
29     int links = 0;
30     int rechts = lange - 1;
31     mergesortP(liste, links, rechts, 1, neueThreadsBisEbene,
        messEbene);
32 };
33
34 void Mergesort::sortW(int *liste, int lange, int workerThreads
    ) {
35     int links = 0;
36     int rechts = lange - 1;
37     mergesortW(liste, links, rechts, workerThreads);
38 };
39
40 void Mergesort::mergesort(int *liste, const int links, const
    int rechts) {
41     int lange = rechts - links + 1;
42     if (lange > 1) {
43         int mitte = links + ((rechts - links) / 2);
44         mergesort(liste, links, mitte); // A
45         mergesort(liste, mitte + 1, rechts); // B
46         mischen(liste, links, mitte, rechts, lange);
47     }
48 };
49
50 void Mergesort::mergesort(int *liste, const int links, const
    int rechts, const int aktuelleEbene, const int messEbene) {
51     if (aktuelleEbene == messEbene) {
52         mergesortM(liste, links, rechts, aktuelleEbene);
53     } else {
54         int lange = rechts - links + 1;
55         if (lange > 1) {
56             int mitte = links + ((rechts - links) / 2);
57             mergesort(liste, links, mitte, aktuelleEbene + 1,
                messEbene);
58             mergesort(liste, mitte + 1, rechts, aktuelleEbene
                + 1, messEbene);
59             mischen(liste, links, mitte, rechts, lange);
60         }
61     }
62 };
63
64 void Mergesort::mergesortM(int *liste, const int links, const

```

```

int rechts, const int aktuelleEbene) {
65     Messdaten *pos = new Messdaten();
66     pos->start1 = std::chrono::high_resolution_clock::now();
67     int lange = rechts - links + 1;
68     if (lange > 1) {
69         int mitte = links + ((rechts - links) / 2);
70         pos->start2 = std::chrono::high_resolution_clock::now
            ();
71         mergesort(liste, links, mitte);
72         mergesort(liste, mitte + 1, rechts);
73         pos->ende2 = std::chrono::high_resolution_clock::now()
            ;
74         mischen(liste, links, mitte, rechts, lange);
75     }
76     pos->ende1 = std::chrono::high_resolution_clock::now();
77     Messdaten::addMessDaten(aktuelleEbene, pos);
78 };
79
80 void Mergesort::mergesortP(int *liste, const int links, const
    int rechts, const int aktuelleEbene, const int
    neueThreadsBisEbene) {
81     if (aktuelleEbene < neueThreadsBisEbene) {
82         int lange = rechts - links + 1;
83         if (lange > 1) {
84             int mitte = links + ((rechts - links) / 2);
85             // mergesort(liste, links, mitte);
86             std::thread thread(
87                 static_cast<void (*)(&Mergesort::
                    int, const int, const
                    int, const int, const int)>(&Mergesort::
                        mergesortP),
88                 liste, links, mitte, aktuelleEbene + 1,
                    neueThreadsBisEbene);
89             // mergesort(liste, mitte + 1, rechts);
90             mergesortP(liste, mitte + 1, rechts, aktuelleEbene
                + 1, neueThreadsBisEbene);
91             thread.join();
92             mischen(liste, links, mitte, rechts, lange);
93         }
94     } else {
95         mergesort(liste, links, rechts);
96     }
97 };
98
99 void Mergesort::mergesortP(int *liste, const int links, const
    int rechts, const int aktuelleEbene, const int
    neueThreadsBisEbene, const int messEbene) {
100     if (aktuelleEbene < neueThreadsBisEbene) {
101         if (aktuelleEbene == messEbene) {
102             mergesortPM(liste, links, rechts, aktuelleEbene,
                neueThreadsBisEbene);
103         } else {

```



```

104         int lange = rechts - links + 1;
105         if (lange > 1) {
106             int mitte = links + ((rechts - links) / 2);
107             // mergesort(liste, links, mitte);
108             std::thread thread(
109                 static_cast<void (*) (int *, const int,
110                     const int, const int, const int, const
111                     int)>(&Mergesort::mergesortP),
112                 liste, links, mitte, aktuelleEbene + 1,
113                 neueThreadsBisEbene, messEbene);
114             // mergesort(liste, mitte + 1, rechts);
115             mergesortP(liste, mitte + 1, rechts,
116                 aktuelleEbene + 1, neueThreadsBisEbene,
117                 messEbene);
118             thread.join();
119             mischen(liste, links, mitte, rechts, lange);
120         }
121     }
122 } else {
123     mergesort(liste, links, rechts, aktuelleEbene,
124         messEbene);
125 }
126 };
127
128 void Mergesort::mergesortPM(int *liste, const int links, const
129     int rechts, const int aktuelleEbene, const int
130     neueThreadsBisEbene) {
131     Messdaten *pos = new Messdaten();
132     pos->start1 = std::chrono::high_resolution_clock::now();
133     int lange = rechts - links + 1;
134     if (lange > 1) {
135         int mitte = links + ((rechts - links) / 2);
136         pos->start2 = std::chrono::high_resolution_clock::now
137             ();
138         // mergesort(liste, links, mitte);
139         std::thread thread(
140             static_cast<void (*) (int *, const int, const int,
141                 const int, const int)>(&Mergesort::mergesortP),
142             liste, links, mitte, aktuelleEbene + 1,
143             neueThreadsBisEbene);
144         // mergesort(liste, mitte + 1, rechts);
145         Mergesort::mergesortP(liste, mitte + 1, rechts,
146             aktuelleEbene + 1, neueThreadsBisEbene);
147         thread.join();
148         pos->ende2 = std::chrono::high_resolution_clock::now()
149             ;
150         mischen(liste, links, mitte, rechts, lange);
151     }
152     pos->ende1 = std::chrono::high_resolution_clock::now();
153     Messdaten::addMessDaten(aktuelleEbene, pos);
154 };

```

```

142
143 void Mergesort::mergesortW(int *liste, int links, int rechts,
144     int workerThreads) {
145     MergeWorkerPool pool(workerThreads);
146
147     pool.taskHandler = [&](int *liste, int links, int rechts,
148         MergeWorkerPool &pool) {
149         if (links < rechts) {
150             int lange = rechts - links + 1;
151             if (lange < Sortiervverfahren::mindestLange) {
152                 mergesort(liste, links, rechts);
153             } else {
154                 int mitte = links + ((rechts - links) / 2);
155                 auto leftHandle = pool.addTaskSmart({liste,
156                     links, mitte});
157                 pool.taskHandler(liste, mitte + 1, rechts,
158                     pool);
159                 leftHandle.wait();
160                 mischen(liste, links, mitte, rechts, lange);
161             }
162         }
163     };
164
165     // Starttask
166     pool.taskHandler(liste, links, rechts, pool);
167 }
168
169 void Mergesort::mischen(int *liste, int links, const int mitte
170     , const int rechts, const int lange) {
171     int *listeB = new int[lange];
172
173     // Kopiere nach listeB
174     for (int i = links; i < mitte + 1; i++) {
175         listeB[i - links] = liste[i];
176     }
177     for (int i = mitte + 1; i < rechts + 1; i++) {
178         listeB[lange - 1 + mitte + 1 - i] = liste[i];
179     }
180
181     // Sortiere liste
182     int i = 0;           // links
183     int j = lange - 1;   // rechts
184     int k = links;       // links
185     while (i < j) {
186         if (listeB[i] < listeB[j]) {
187             liste[k] = listeB[i];
188             i++;
189         } else {
190             liste[k] = listeB[j];
191             j--;
192         }
193     }

```

```

188         k++;
189     }
190     liste[rechts] = listeB[i];
191
192     delete[] listeB;
193 };

```

Listing 4: Mergesort WorkerPool

```

1 // MergeWorkerPool.h
2 #pragma once
3 #include <atomic>
4 #include <condition_variable>
5 #include <functional>
6 #include <mutex>
7 #include <queue>
8 #include <thread>
9 #include <vector>
10
11 struct MergeTask {
12     int *liste;
13     int links;
14     int rechts;
15 };
16
17 class MergeTaskHandle {
18 public:
19     std::shared_ptr<std::atomic<bool>> done;
20     void wait() const {
21         while (!done->load(std::memory_order_acquire)) {
22             std::this_thread::yield();
23         }
24     }
25 };
26
27 class MergeWorkerPool {
28 private:
29     std::vector<std::thread> threads;
30     std::queue<MergeTask> taskQueue;
31     std::queue<std::shared_ptr<std::atomic<bool>>> doneFlags;
32
33     std::mutex mutex;
34     std::condition_variable cv;
35
36     std::atomic<int> activeTasks;
37     bool finished;
38
39 public:
40     std::function<void(int *, int, int, MergeWorkerPool &)>
41         taskHandler;
42 public:

```

```

43 MergeWorkerPool(int numThreads) : activeTasks(0), finished
    (false) {
44     for (int i = 0; i < numThreads; ++i) {
45         threads.emplace_back(&MergeWorkerPool::worker,
                                this);
46     }
47 }
48
49 ~MergeWorkerPool() {
50     {
51         std::lock_guard<std::mutex> lock(mutex);
52         finished = true;
53     }
54     cv.notify_all();
55     for (auto &t : threads)
56         t.join();
57 }
58
59 // intelligente Task-Anmeldung
60 MergeTaskHandle addTaskSmart(const MergeTask &task) {
61     std::unique_lock<std::mutex> lock(mutex);
62
63     // Kein freier Thread selber ausführen
64     if (activeTasks >= threads.size()) {
65         lock.unlock();
66         taskHandler(task.liste, task.links, task.rechts, *
                     this);
67
68         MergeTaskHandle h;
69         h.done = std::make_shared<std::atomic<bool>>(true)
            ;
70         return h;
71     }
72
73     // Worker frei -> auf Queue legen
74     MergeTaskHandle h;
75     h.done = std::make_shared<std::atomic<bool>>(false);
76     taskQueue.push(task);
77     doneFlags.push(h.done);
78
79     activeTasks++;
80     lock.unlock();
81     cv.notify_one();
82     return h;
83 }
84
85 private:
86     void worker() {
87         while (true) {
88             MergeTask task;
89             std::shared_ptr<std::atomic<bool>> doneFlag;

```

```

90
91     {
92         std::unique_lock<std::mutex> lock(mutex);
93         cv.wait(lock, [this] { return finished || !
          taskQueue.empty(); });
94
95         if (finished && taskQueue.empty())
96             return;
97
98         task = taskQueue.front();
99         taskQueue.pop();
100         doneFlag = doneFlags.front();
101         doneFlags.pop();
102     }
103
104     if (taskHandler)
105         taskHandler(task.liste, task.links, task.
          rechts, *this);
106
107     doneFlag->store(true, std::memory_order_release);
108
109     activeTasks--;
110     cv.notify_all();
111 }
112 }
113 };

```

Listing 5: Listenersteler

```

1 // Listenersteler.cpp
2 #include "Listenersteler.h"
3 #include "Messdaten.h"
4 #include <algorithm>
5 #include <random>
6
7 Listenersteler::Listenersteler() {};
8
9 int *Listenersteler::erstelleListe(char listeVariante, int
  lange) {
10     Messdaten::arrayArt = "int";
11     int *liste;
12     switch (listeVariante) {
13         case 'z':
14             liste = erstelleZufallsListe(lange);
15             break;
16         case 's':
17             liste = erstelleSortierteListe(lange);
18             break;
19         case 'i':
20             liste = erstelleInvertierteListe(lange);
21             break;
22         case 'f':

```

```

23         liste = erstelleFastSortierteListe(lange);
24         break;
25     case 'd':
26         liste = erstelleDuplizierteListe(lange);
27         break;
28     default:
29         liste = erstelleZufallsListe(lange);
30     }
31     return liste;
32 };
33
34 int *Listenersteler::erstelleZufallsListe(int lange) {
35     listenLange = lange;
36     liste = std::make_unique<int[]>(listenLange);
37
38     // Zufallsgenerator vorbereiten
39     // std::random_device rd;
40     // std::mt19937 gen(rd());
41     std::mt19937 gen(seed);
42     std::uniform_int_distribution<> dis(std::numeric_limits<
43         int>::min(), std::numeric_limits<int>::max());
44     // std::uniform_int_distribution<> dis(0, 9);
45
46     std::generate_n(liste.get(), lange, [&]() { return dis(gen
47         ); });
48
49     Messdaten::arrayTyp = "Zufall";
50     Messdaten::arrayLange = lange;
51
52     return liste.get();
53 }
54
55 int *Listenersteler::erstelleSortierteListe(int lange) {
56     listenLange = lange;
57     liste = std::make_unique<int[]>(listenLange);
58
59     for (int i = 0; i < lange; i++) {
60         liste[i] = i;
61     }
62
63     Messdaten::arrayTyp = "Sortiert";
64     Messdaten::arrayLange = lange;
65
66     return liste.get();
67 }
68
69 int *Listenersteler::erstelleInvertierteListe(int lange) {
70     listenLange = lange;
71     liste = std::make_unique<int[]>(listenLange);
72
73     for (int i = 0; i < lange; i++) {

```

```

72         liste[i] = lange - i;
73     }
74
75     Messdaten::arrayTyp = "InvertSortiert";
76     Messdaten::arrayLange = lange;
77
78     return liste.get();
79 }
80
81 int *Listenersteler::erstelleFastSortierteListe(int lange, int
swaps) {
82     listenLange = lange;
83     liste = std::make_unique<int[]>(listenLange);
84
85     // Erst sortierte Liste erzeugen
86     for (int i = 0; i < lange; i++) {
87         liste[i] = i;
88     }
89
90     // Reproduzierbare Zufallsvertauschungen
91     std::mt19937 gen(seed);
92     std::uniform_int_distribution<> dis(0, lange - 1);
93
94     for (int i = 0; i < swaps; i++) {
95         int a = dis(gen);
96         int b = dis(gen);
97         std::swap(liste[a], liste[b]);
98     }
99
100    Messdaten::arrayTyp = "FastSortiert";
101    Messdaten::arrayLange = lange;
102
103    return liste.get();
104 }
105
106 int *Listenersteler::erstelleDuplizierteListe(int lange, int
uniqueValues) {
107     listenLange = lange;
108     liste = std::make_unique<int[]>(listenLange);
109
110     std::mt19937 gen(seed);
111     std::uniform_int_distribution<> dis(0, uniqueValues - 1);
112
113     std::generate_n(liste.get(), lange, [&]() { return dis(gen
); });
114
115     Messdaten::arrayTyp = "Dupliziert";
116     Messdaten::arrayLange = lange;
117
118     return liste.get();
119 }

```

```

120
121 bool Listenersteler::istSortiert() const {
122     return std::is_sorted(liste.get(), liste.get() +
123         listenLange);
124 }

```

Listing 6: ListenerstelerString

```

1 // ListenerstelerString.cpp
2 #include "ListenerstelerString.h"
3 #include "Messdaten.h"
4 #include <algorithm>
5 #include <random>
6
7 ListenerstelerString::ListenerstelerString() {}
8
9 std::string *ListenerstelerString::erstelleListe(char
10     listeVariante, int lange) {
11     Messdaten::arrayArt = "string";
12     std::string *liste;
13     switch (listeVariante) {
14         case 'z':
15             liste = erstelleZufallsListe(lange);
16             break;
17         case 's':
18             liste = erstelleSortierteListe(lange);
19             break;
20         case 'i':
21             liste = erstelleInvertierteListe(lange);
22             break;
23         case 'f':
24             liste = erstelleFastSortierteListe(lange);
25             break;
26         case 'd':
27             liste = erstelleDuplizierteListe(lange);
28             break;
29         default:
30             liste = erstelleZufallsListe(lange);
31     }
32     return liste;
33 }
34
35 std::string *ListenerstelerString::erstelleZufallsListe(int
36     lange) {
37     listenLange = lange;
38     liste = std::make_unique<std::string[]>(listenLange);
39
40     std::mt19937 gen(seed);
41     std::uniform_int_distribution<> dis(0, 25); // Buchstaben
42         A-Z
43
44     for (int i = 0; i < lange; i++) {

```



```

42         int len = 5 + (dis(gen) % 10); // 5-14 Zeichen
43         std::string s;
44         s.reserve(len);
45         for (int j = 0; j < len; j++)
46             s.push_back('A' + dis(gen));
47         liste[i] = s;
48     }
49
50     Messdaten::arrayTyp = "Zufall";
51     Messdaten::arrayLange = lange;
52
53     return liste.get();
54 }
55
56 std::string *ListenerstelerString::erstelleSortierteListe(int
lange) {
57     listenLange = lange;
58     liste = std::make_unique<std::string[]>(listenLange);
59
60     for (int i = 0; i < lange; i++) {
61         liste[i] = "STR_" + std::to_string(i);
62     }
63
64     Messdaten::arrayTyp = "Sortiert";
65     Messdaten::arrayLange = lange;
66
67     return liste.get();
68 }
69
70 std::string *ListenerstelerString::erstelleInvertierteListe(
int lange) {
71     listenLange = lange;
72     liste = std::make_unique<std::string[]>(listenLange);
73
74     for (int i = 0; i < lange; i++) {
75         liste[i] = "STR_" + std::to_string(lange - i);
76     }
77
78     Messdaten::arrayTyp = "InvertSortiert";
79     Messdaten::arrayLange = lange;
80
81     return liste.get();
82 }
83
84 std::string *ListenerstelerString::erstelleFastSortierteListe(
int lange, int swaps) {
85     listenLange = lange;
86     liste = std::make_unique<std::string[]>(listenLange);
87
88     for (int i = 0; i < lange; i++) {
89         liste[i] = "STR_" + std::to_string(i);

```

```

90     }
91
92     Messdaten::arrayTyp = "FastSortiert";
93     Messdaten::arrayLange = lange;
94
95     std::mt19937 gen(seed);
96     std::uniform_int_distribution<> dis(0, lange - 1);
97
98     for (int i = 0; i < swaps; i++) {
99         std::swap(liste[dis(gen)], liste[dis(gen)]);
100     }
101
102     return liste.get();
103 }
104
105 std::string *ListenerstelerString::erstelleDuplizierteListe(
106     int lange, int uniqueValues) {
107     listenLange = lange;
108     liste = std::make_unique<std::string[]>(listenLange);
109
110     std::mt19937 gen(seed);
111     std::uniform_int_distribution<> dis(0, uniqueValues - 1);
112
113     for (int i = 0; i < lange; i++) {
114         liste[i] = "STR_" + std::to_string(dis(gen));
115     }
116
117     Messdaten::arrayTyp = "Dupliziert";
118     Messdaten::arrayLange = lange;
119
120     return liste.get();
121 }
122
123 bool ListenerstelerString::istSortiert() const {
124     return std::is_sorted(liste.get(), liste.get() +
125         listenLange);
126 }

```