OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

ELEKTRO- UND
INFORMATIONSTECHNIK

Praktikum Algorithmen und Datenstrukturen

Merge-Sort und Parallelisierung

Autor(en): M. Farmbauer, M. Niemetz

Dezember 2024

Version: 1.1

letzte Aktualisierung: 17. Dezember 2024

Ostbayerische Technische Hochschule Regensburg
Fakultät für Elektro- und Informationstechnik

Vorbereitung

☐ **Merge-Sort:** Lesen Sie Idee des Merge-Sort-Algorithmus nach. 2

1 Einleitung

Sie haben in der Vorlesung den MergeSort-Algorithmus kennengelernt. In dieser Aufgabe werden sie diesen Algorithmus in verschiedenen Ausprägungen implementieren und das Verhalten der Implementierungen bei der Verwendung paralleler Verarbeitungseinheiten untersuchen.

> **Hinweis:** Die Ergebnisse der Arbeiten werden unter anderem davon abhängen, wie viele parallele Verarbeitungseinheiten die von Ihnen verwendete Hardware hat, aber auch von anderen Eigenschaften Ihrer Hardware wie dem zur Verfügung stehenden Speicher und dessen Anbindung!

Recherche
Merge-Sort

Um die Wirkung der Parallelisierung auf ein hierfür ideal geeignetes Problem kennenzulernen, implementieren Sie im Anschluss die Berechnung eines sogenannten Apfelmännchens, einer fraktalen Struktur.

2 Merge-Sort

2.1 Der vorbereitete Coderahmen

Um Ihnen die Möglichkeit zu geben, sich auf das Wesentliche der Implementierung zu konzentrieren, finden Sie im Upstream-Projekt zum Praktikum einen vorbereiteten Coderahmen.

Aktualisieren Sie ihr Projekt, damit dieser Coderahmen in Ihr Projekt aktualisiert wird.

Aktualisieren Sie ihren Fork des Betreuerprojekts indem Sie `git pull upstream master` ausführen.

Danach überprüfen Sie, ob sich die erwarteten Aktualisierungen in Ihrem Repository befinden.

Machen Sie sich mit dem Coderahmen zu Merge-Sort vertraut. Insbesondere sollten Sie die Funktion `main` analysieren und mit Kommentaren versehen.

Machen Sie sich mit dem Coderahmen vertraut.

Der Coderahmen enthält einige Funktionen zur Unterstützung Ihrer Untersuchungen sowie leere Funktionen, die Sie im weiteren

Verlauf mit Implementierungen des MergeSort-Algorithmus füllen.

2.2 Implementierung von Merge-Sort

2.2.1 Implementierung der Funktion `mergesort`

Implementieren Sie die Funktion `mergesort`.

Implementieren Sie nun die Funktion `mergesort`. Die Funktion erhält als Parameter einen Pointer auf die zu sortierenden Daten sowie die Anzahl der Daten.

Als Rückgabewert wird ein Pointer auf die sortierten Daten erwartet.

Dabei ist es das Ziel, die zurückgelieferten Daten in einem neuen, dynamisch angelegten Speicherbereich abzulegen, der natürlich die selbe Länge besitzt, wie die ursprünglichen unsortierten Daten. Dieser zurückgelieferte Zielbereich darf nicht der selbe Speicherbereich sein, wie der durch die ursprünglich unsortierten Daten genutzte Bereich.

Gehen Sie hierfür folgendermaßen vor:

- Wenn die Anzahl der sortierenden Daten gleich Null oder eins ist: Lösen Sie die Aufgabe explizit wie beschrieben. **Wichtig:** Auch in diesem sehr einfachen Fall ist es wichtig, dass Sie die Daten in einem neu angelegten Speicherbereich zurückliefern. Die aufrufende Funktion erwartet das, weil wir das so festgelegt haben.

- Teilen Sie die Liste der zu sortierenden Daten in zwei möglichst gleiche Teile **ohne** die Daten dabei umzukopieren, indem Sie zwei entsprechende Pointer in das Datenarray berechnen. Speichern Sie diese beiden Pointer in einem Array der Länge zwei (lokale Variable) ab.

  Berechnen Sie außerdem die zugehörigen Längen der beiden Teile des Arrays und speichern Sie diese in einem weiteren Array der Länge zwei ab.

- Rufen Sie nun die Funktion `mergesort` für beide Teillisten auf, indem Sie eine `for`-Schleife und die beiden soeben erzeugten Arrays verwenden. Speichern Sie die von den `mergesort`-Aufrufen zurückgelieferten Rückgabewerte (Pointer auf die dann jeweils für sich sortierten Teillisten) dabei in einem weiteren Array ab.

  Wichtig: Bitte verwenden Sie für die rekursiven Aufrufe von `mergesort` unbedingt wie beschrieben eine `for`-Schleife. Das erleichtert Ihnen später die Parallelisierung.

- Legen Sie nun dynamisch (mit `new`) ein weiteres Array an, welches die sortierten Daten aufnehmen kann, und rufen Sie dann die Funktion `simplemerge` auf, an die sie die beiden im vorangehenden Schritt sortierten Teillisten, die Elementanzahlen (Längen) dieser beiden Listen und das soeben erzeugte Ergebnisarray übergeben.
- Nun können (eher: müssen) Sie die beiden sortierten Teillisten wieder aus dem Speicher löschen und das Ergebnisarray als Rückgabewert zurückliefern.

Hinweis: Ihr Programm funktioniert noch nicht. Sie müssen erst noch `simplemerge` implementieren.

2.2.2 Implementierung der Funktion `simplemerge`

Implementieren Sie die Funktion `simplemerge`.

Die naheliegendste Lösung des Problems, zwei jeweils für sich sortierte Listen zu einer Sortierten Liste zusammenzuführen, besteht darin, das Reißverschlussverfahren anzuwenden:

Beginnend beim Anfang der beiden Listen wird jeweils das kleinere der beiden Elemente am Anfang der beiden Listen in die aktuelle Zielposition des Zielbereichs geschrieben. Dann wird das Element aus seiner bisherigen Liste entfernt und die Zielposition im Zielbereich erhöht und das Verfahren so lange fortgesetzt, bis beide der zusammenzuführenden Listen abgearbeitet sind.

Hinweis: Selbstverständlich verschieben/entfernen Sie dabei keine Elemente tatsächlich aus den beiden zusammenzuführenden Teillisten. Dies würde nur unnötig Bearbeitungszeit verschwenden.

Arbeiten Sie stattdessen mit geeigneten Indizes in die betreffenden Arrays, die sie beim Entnehmen eines Elements aus einer der beiden Teillisten bzw. beim Einfügen eines Elements in den Zielbereich erhöhen.

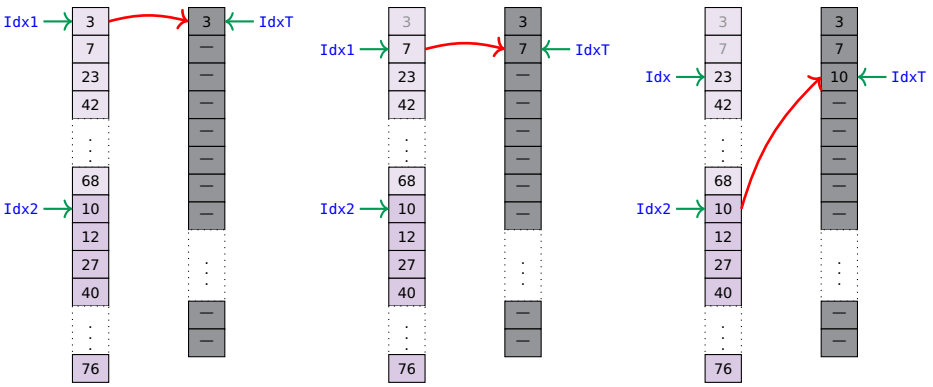

Abbildung 1: Skizze der ersten Schritte beim Zusammenführen der beiden sortierten Teillisten durch `simplemerge`.

> **Wichtig:** Beachten Sie, dass beide Listen nicht exakt die selbe Länge haben müssen, vergessen Sie insbesondere nicht, **alle** Elemente aus **beiden** Listen in den Zielbereich zu kopieren.

Testen Sie nun Ihren Sortieralgorithmus und beseitigen Sie ggf. vorhandene Fehler.

☕?
Testen Sie Ihre Sortierfunktion und beseitigen Sie ggf. vorhandene Fehler.

2.3 Vermessen Sie die Laufzeit

Die mitgelieferte `main`-Funktion berechnet die Laufzeit für unterschiedliche Anzahlen von Elementen und gibt eine Tabelle aus.

Lassen Sie die Tabelle berechnen und kopieren Sie sie in eine Textdatei, um ein Diagramm erzeugen zu können. Stellen Sie dabei bitte beide Achsen logarithmisch dar.

> **Hinweis:** Der Programmrahmen enthält eine leere Ergebnisdatei und ein gnuplot-Script, die sie für die Darstellung verwenden können. Sie können aber auch jedes andere Programm zur Darstellung der Diagramme auswählen.

☕
Vermessen Sie die Laufzeit Ihrer Implementierung.

2.4 Parallelisierung des Algorithmus mit OpenMP

In der Vorlesung haben Sie bereits von OpenMP gehört. Nun werden Sie OpenMP verwenden, um Ihre Implementierung in mehreren parallelen Threads ausführen zu lassen.

> **Hinweis:** Wenn Sie CodeBlocks 20.3 aus dem Paket mit mingw installiert haben, dann unterstützt der Compiler OpenMP ohne weitere Maßnahmen. Der Projektrahmen, den Sie erhalten haben, enthält auch bereits die nötigen Optionen für den Compiler (`-fopenmp`) bzw. den Linker (`-lgomp`). Bei anderen Compilern bzw. Entwicklungsumgebungen müssen Sie evtl. erst Dinge neu installieren bzw. ermitteln, ob bzw. wie die OpenMP-Unterstützung aktiviert werden kann.

☕☕
Erweitern Sie Ihre Implementierung auf die Verwendung paralleler Threads.

Um Ihr Programm parallel ausführen zu lassen, fügen Sie eine Definition für ein Präprozessorsymbol ein, mit dessen Hilfe Sie dann die Parallelisierung zuschalten können, sowie eine abschaltbare Include-Direktive für den OpenMP-Header:

```
#define PARALLEL
#ifdef PARALLEL
#include <omp.h>
#endif // PARALLEL
```

Dann fügen Sie **unmittelbar** vor Ihrer Schleife in der Funktion `mergesort`, welche die rekursiven Aufrufe erledigt, die Direktive

```
#pragma omp parallel for
```

ein, die Sie ebenfalls über die Compilerdirektive **abschaltbar** gestalten.

> **Frage:** Beobachten Sie den Erfolg in der Prozessorauslastanzeige Ihres Rechners. Wie erklären Sie sich das Ergebnis?

Fügen Sie nun eine (ausnahmsweise) globale Variable `unsigned depth` hinzu und stellen Sie sicher, dass sie in der `main`-Funktion vor jedem Aufruf von `mergesort` auf Null gesetzt wird. Inkrementieren Sie die Variable `depth` am Anfang der Funktion `mergesort` und dekrementieren Sie sie vor dem Beenden der Funktion.

Ergänzen Sie nun innerhalb der Schleife, direkt vor dem rekursiven Aufruf von `mergesort`, den Funktionsaufruf

```
omp_set_nested(NumItems>10000 && depth<155);
```

> **Frage:** Beobachten Sie den Erfolg in der Prozessorauslastanzeige Ihres Rechners. Wie erklären Sie sich das Ergebnis?

> **Frage:** Was bewirkt der Parameter in dem oben gezeigten Aufruf von `omp_set_nested`? Was passiert, wenn Sie hier einfach `true` angeben?

2.4.1 Vermessen Sie die Laufzeit

☕

Vermessen Sie die Laufzeit Ihrer parallelisierten Implementierung.

Lassen Sie erneut die Laufzeittabelle berechnen und ergänzen Sie Ihr Diagramm um die zweite Datenreihe.

> **Frage:** Wie viel schneller wurde Ihr Programm durch die Parallelisierung auf Ihrem Rechner tatsächlich? Was hätten Sie erwartet? Wie erklären Sie sich die Diskrepanz?

2.5 Verbesserung der Parallelisierbarkeit – V1

☕☕☕

Erzeugen Sie eine parallelisierbare Alternative zu `simplemerge`.

Ein offensichtliches Problem bei unserer Parallelisierung ist, dass wir das Zusammenführen der beiden Teillisten in die jeweilige Ergebnisliste nicht parallelisieren können, da das Reißverschlussverfahren ein sequentielles Vorgehen zwingend erforderlich macht.

Aus dem Skript zu parallelen Algorithmen kennen Sie allerdings bereits eine Möglichkeit, den Vorgang des Zusammenfügens zu parallelisieren.

Kurz zusammengefasst besteht die Idee darin, die Zielposition eines beliebigen Elements aus einer der beiden sortierten Teillisten

in der gemeinsamen Zielliste unabhängig von der Einsortierung der übrigen Elemente zu berechnen. Hierfür ist erforderlich zu wissen, wie viele Elemente vor dem einzusortierenden Element in die Zielliste gelegt werden müssen.

Da beide Listen sortiert sind, ist dies leicht möglich: Einerseits müssen für ein Element a der Teilliste A alle Elemente, die in der Teilliste A vor diesem Element liegen, auch im Zielbereich vor diesem Element liegen, wenn also Element a in der Teilliste A den Index i_a^A hat, liegen schon einmal so viele Elemente sicher auch im Zielbereich vor dem Element a.

Um die Frage zu klären, wie viele Elemente aus der Teilliste B vor dem Element a einsortiert werden müssen, gilt es zu ermitteln, an welcher Indexposition i_a^B das Element a in der Teilliste B einsortiert würde. Man nennt dies auch den Rang von a in B.

Hat man diese Information zur Verfügung ergibt sich die Position in der Zielliste als Summe

$$i_a^Z = i_a^A + i_a^B$$

Da beide Teillisten sortiert sind, ist die Rangermittlung über eine binäre Suche möglich und erfordert daher asymptotisch nur logarithmische Laufzeit $\mathcal{O}$(lg n).

Allerdings kann der Algorithmus in der im Skript beschriebenen Form nicht mit der Situation umgehen, dass mehrere Elemente mit gleicher Ordnung (also zum Beispiel in unserem Fall identische Zahlenwerte) auftreten, was aber bei unseren Zufallszahlen durchaus vorkommt. Für solche Zahlenwerte würde es zu Kollisionen im Zielbereich kommen, so dass noch eine Detailvariation nötig wird.

2.5.1 Implementierung der Rangermittlung

Entwickeln Sie **zwei** Varianten der Rangermittlung.

Die Lösung für das Problem ist, dass Sie zwei verschiedene Funktionen zur Ermittlung des Rangs eines Wertes in einer Liste entwickeln:

- **equalbelow:** Diese Variante liefert den größten Index, für den das entsprechende Element noch kleiner ist als der einzusortierende Wert.
- **equalabove:** Diese Variante liefert den kleinsten Index, für den das entsprechende Element noch größer ist als der einzusortierende Wert.

Das bedeutet, im einen Fall werden identische Werte *vor* ihren Zwillingen, die schon in der Liste sind, einsortiert, im anderen Fall werden diese Werte *nach* ihren Zwillingen, die schon in der Liste sind, in der der Rang ermittelt wird, einsortiert.

Verwendet man nun konsequent `equalbelow` zur Rangermittlung in der Liste A und `equalabove` zur Rangermittlung in der Liste B führt dies dazu, dass bei identischen (besser: ranggleichen) Werten, die ranggleichen Elemente aus A vor denen aus B in den Zielbereich einsortiert werden. Siehe Abbildung 2 für ein einfaches Beispiel.

> **Wichtig:** Beide Varianten der Rangermittlung müssen ihr Ergebnis über eine binäre Suche ($\mathcal{O}(\lg n)$) ermitteln – keinesfalls über eine lineare Suche ($\mathcal{O}(n)$). Sonst ist der Aufwand für die Rangermittlung zu hoch, um einen Vorteil erzielen zu können.

2.5.2 Implementierung der parallelisierbaren Mergefunktion

Entwickeln Sie eine parallelisierbare Mergefunktion.

Verwenden Sie nun die beiden Funktionen aus der letzten Teilaufgabe, um den Algorithmus aus dem Skript in Form einer neuen Funktion `pmerge` mit identischer Signatur wie `simplemerge` umzusetzen.

Achten Sie dabei darauf, die eine Funktion des oben entwickelten Paares `equalabove` bzw. `equalbelow` für die Ermittlung des Ranges in der einen Teilliste zu verwenden, während Sie die andere für die Ermittlung des Ranges in der anderen Teilliste verwenden.

Die im Algorithmus ebenfalls abgegebene Ermittlung des eigenen Ranges eines Elements in der eigenen Liste ersetzen Sie einfach durch die Indexposition des Elements.

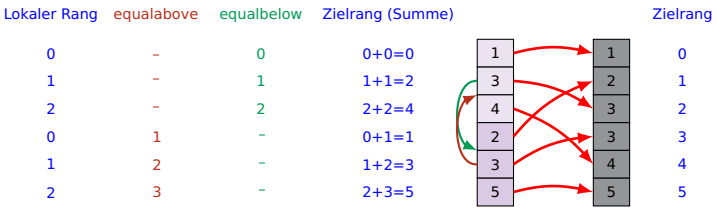

Abbildung 2: Verbessertes und parallelisierbares Merge durch Rangermittlung. Falls mehrere gleiche Elemente in der zu sortierenden Menge auftreten können, müssen diese in beiden Teillisten asymmetrisch einsortiert werden, wie hier für den Wert „3“ gezeigt.

Vor die beiden Schleifen, welche die Elemente in den Zielbereich kopieren, setzen Sie wieder jeweils eine Parallelisierungsdirektive:

```
#pragma omp parallel for
```

Die Sie wieder mit dem Präprozessorsymbol abschaltbar gestalten.

2.5.3 Vermessen Sie die Laufzeit

Vermessen Sie die Laufzeit Ihrer verbesserten parallelisierten Implementierung.

Lassen Sie erneut die Laufzeittabelle berechnen und ergänzen Sie Ihr Diagramm um die dritte (mit abgeschalteter Parallelisierung) und vierte (mit eingeschalteter Parallelisierung) Datenreihe.

> **Frage:** Wie viel schneller wurde Ihr Programm durch die Parallelisierung? Was hätten Sie erwartet? Wie erklären Sie sich die Diskrepanz?

2.6 Verbesserung der Parallelisierbarkeit – V2

Erzeugen Sie eine weiteren parallelisierbare Alternative zu `simplemerge`.

Erzeugen Sie eine weitere Konstante:

```
const unsigned P2mergeThreads = 16;
```

und definieren Sie sie auf die **Anzahl der Ausführungseinheiten** Ihres Rechners.

Entwickeln Sie dann eine weitere Funktion `p2merge`, welche die selbe Signatur aufweist wie `simplemerge` und `pmerge`.

Sie soll Mergevorgang so erledigen, dass die Liste A in `P2mergeThreads` gleich lange Blöcke aufgeteilt wird (der letzte Block kann evtl. etwas kürzer werden), diesen Blöcken über die Rang-Funktion die passenden Blöcke in der Liste B zugeordnet werden und dann die zusammengehörenden Blöcke in einer Schleife abgearbeitet werden, indem für jedes Blockpaar `simplemerge` aufgerufen wird.

Vor die Schleife setzen Sie wieder die bekannte, abschaltbare Parallelisierungdirektive.

> **Wichtig:** Verdeutlichen Sie sich den Ansatz über eine Skizze!

> **Hinweis:** Die Funktion soll aus Effizienzgründen für Listen mit einer Länge unter 100 Elementen direkt simplemerge aufrufen, ohne die Aufteilung vorzunehmen.

2.6.1 Vermessen Sie die Laufzeit

Vermessen Sie die Laufzeit Ihrer verbesserten parallelisierten Implementierung.

Lassen Sie erneut die Laufzeittabelle berechnen und ergänzen Sie Ihr Diagramm um die fünfte (mit abgeschalteter Parallelisierung) und sechste (mit eingeschalteter Parallelisierung) Datenreihe.

> **Frage:** Welche Lehren ziehen Sie aus Ihren bisherigen Experimenten?

2.7 Threads im Handbetrieb

C++ bietet auch eine schlanke Schnittstelle für die Erzeugung paralleler Threads ohne den Komfort von OpenMP an.

Als Beispiel verwenden wir das Programm zur Berechnung der Mandelbrotmenge, besser bekannt als „Apfelmännchen“.

Hierfür haben Sie eine komplette Implementierung erhalten, welche die Grafik berechnet und nach einigen zehn Sekunden auch anzeigt.

> **Hinweis:** Das Programm enthält ein Verzögerungskommando, um auch auf sehr schnellen Rechnern eine gut beobachtbare Wirkung erkennen zu lassen. Passen Sie den Wert für besonders schnelle Rechner an oder entfernen Sie die Verzögerung vollständig für besonders langsame Rechner.

Machen Sie sich mit dem Programmrahmen vertraut.

Analysieren Sie das Programm und die Klasse Mandelbrot um sich einen Überblick über das Programm zu verschaffen. Legen Sie dabei besonders großen Wert auf die Methode `calc`, mit der wir uns im folgenden näher auseinandersetzen werden.

> **Wichtig:** Praktischerweise führen Sie alle Änderungen in der Folge so durch, dass Sie sie über eine Präprozessordefinition umschalten können.

2.7.1 Vorbereitung der Parallelisierung

C++ kennt eine Klasse `thread`, deren Objekte unabhängig voneinander laufende parallele Ausführungspfade darstellen, die aber alle den **Speicher** des Programms **gemeinsam** nutzen.

Dieses Modell kommt uns hier sehr gelegen, da die Berechnung der einzelnen Bildpunkte der Mandelbrotmenge unabhängig von den Nachbarn vorgenommen werden kann.

Wichtig: Die gemeinsame Verwendung von Speicher durch mehrere parallele Threads ist im allgemeinen nicht so problemlos wie in unserem hier vorliegenden Fall. Dies liegt daran, dass es bei der Berechnung hier zu keiner Wechselwirkung der Arbeit der unterschiedlichen Threads kommt – alle arbeiten ausschließlich mit „eigenen“ Daten.

Das ist eher selten so. Man spricht dann von „peinlich parallelem“ Verhalten („embarrassingly parallel“) - weil alles so einfach ist.

Bei der Instanziierung der Objekte der Klasse `thread` sind als Parameter zu übergeben:

- Die Funktion (Funktionspointer), deren Code in dem Thread ausgeführt werden soll.
- Die Parameter, die der Funktion zu übergeben sind.

  Wichtig: Falls, wie in unserem Fall, hier eine Methode der Klasse aufgerufen werden soll, muss zusätzlich zu den „eigentlichen“ Parametern der Funktion als erster Parameter der `this`-Pointer auf das jeweilige Objekt übergeben werden.

Erzeugen Sie eine Funktion `calcinthread`, welche den parallel laufenden Code enthält.

Um also hier voranzukommen, muss zunächst der Teil der `calc`-Funktion, der parallel ausgeführt werden soll, in eine eigene Funktion ausgegliedert werden.

Dabei soll die Funktion entsprechend den übergebenen Werten jeweils einen anderen Teil der Zielpunkte berechnen.

Diese Funktion muss also als Parameter die „Nummer“ des Threads erhalten, damit anhand dieser Nummer die einzelnen Threads auch unterschiedliche Teile des Problems bearbeiten können.

Schreiben Sie eine Funktion `calcinthread`, welche abhängig von der übergebenen Threadnummer `ThreadPhase` die Pixelzeilen mit den Indizes `ThreadPhase`$+i*$`numthreads` berechnet, wobei i soweit läuft, bis das Bildende erreicht ist.

Damit sind alle Threads für nicht überlappende Bereiche von Zeilen zuständig.

Rufen Sie die `calcinthread`-Funktion in einer Schleife auf.

Ändern Sie nun die Funktion `calc` so ab, dass in einer Schleife die Funktion `calcinthread` für alle späteren Threads von 0 bis `numthreads` einmal aufgerufen wird.

Testen Sie Ihr Programm.

Testen Sie Ihr Programm und beheben Sie eventuelle Fehler.

2.7.2 Parallele Threads

Erzeugen Sie parallele Threads.

Anstatt die Funktion `calcinthread` in der Schleife aus der vorangehenden Teilaufgabe direkt aufzurufen, erzeugen Sie nun in der Schleife bei jedem Durchlauf dynamisch ein neues Thread-Objekt, dem Sie die entsprechenden Parameter (Funktionspointer, `this`-Pointer und Threadnummer) übergeben.

> **Frage:** Was beobachten Sie im Unterschied zu vorher? Stellt sich der von Ihnen erwartete Effekt ein?

2.7.3 Vermessen Sie die Laufzeit

Vermessen Sie die Laufzeit der beiden Implementierungen.

Integrieren Sie die Zeitmessbibliothek in Ihr Programm und vermessen Sie die Laufzeit beider Implementierungen (parallel und nicht-parallel) und ermitteln Sie den Einfluss der Anzahl an erzeugten Threads auf die Abarbeitungsgeschwindigkeit.

> **Frage:** Was lernen Sie aus dem Ergebnis?