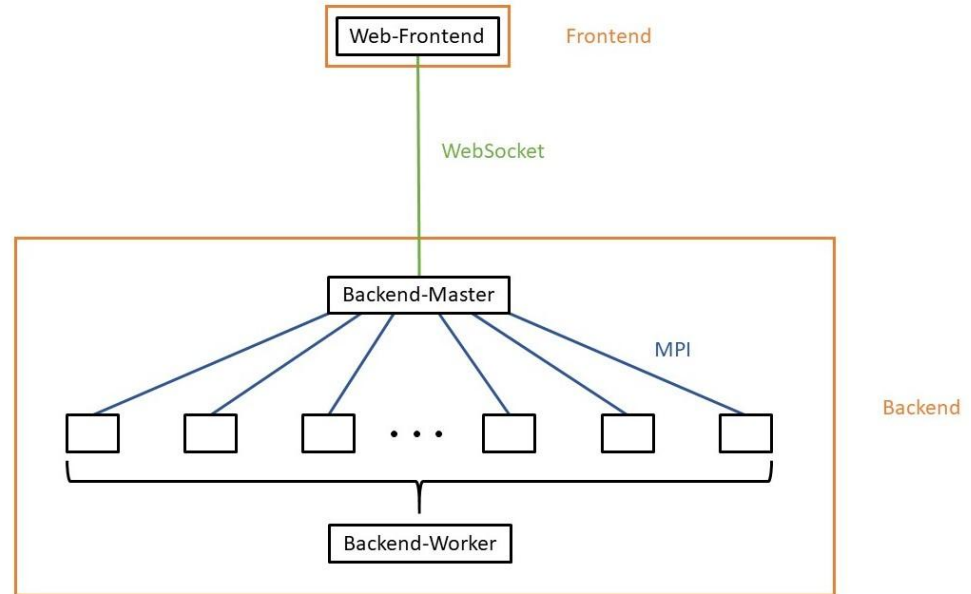


# ERAGP Mandelbrot

Maximilian Frühauf, Tobias Klausen,  
Florian Lercher, Niels Mündler



1. Einführung und Motivation
2. Ansatz
3. Beschreibung der Implementierung
4. Evaluation
5. Zusammenfassung
6. Demonstration

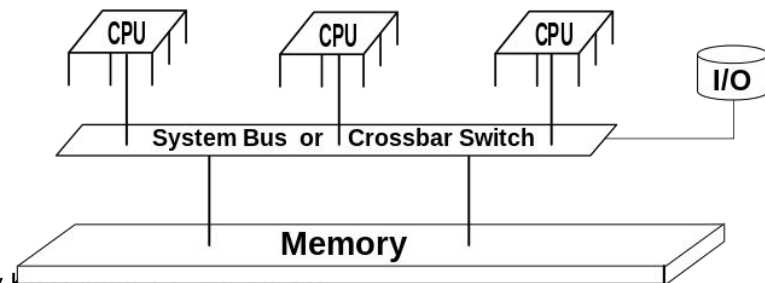


- MPI Kommunikation
  - Message Passing Interface
  - unabhängige Rechenknoten
  - versenden von Daten z.B. über ssh
- OpenMP
- SIMD



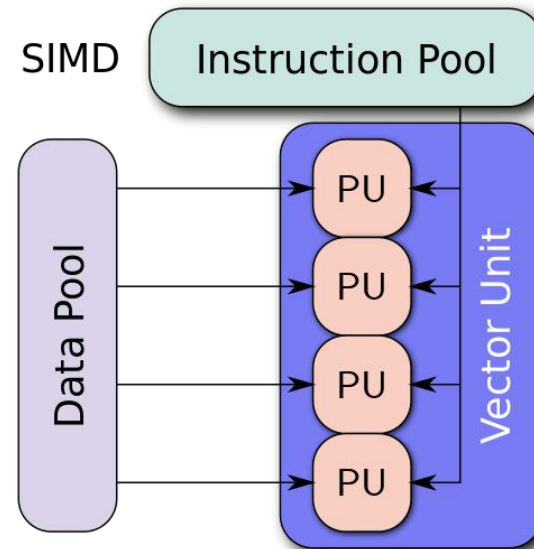
HimMUC Cluster des CAPS-Lehrstuhl TUM  
<https://www.caps.in.tum.de/himmuc/>

- MPI Kommunikation
- OpenMP
  - Automatische Parallelisierung von untereinander unabhängigen Programmblöcken
  - Realisierung durch Threads, geteilten Hauptspeicher
  - nur für Rechenkerne auf einer CPU
- SIMD

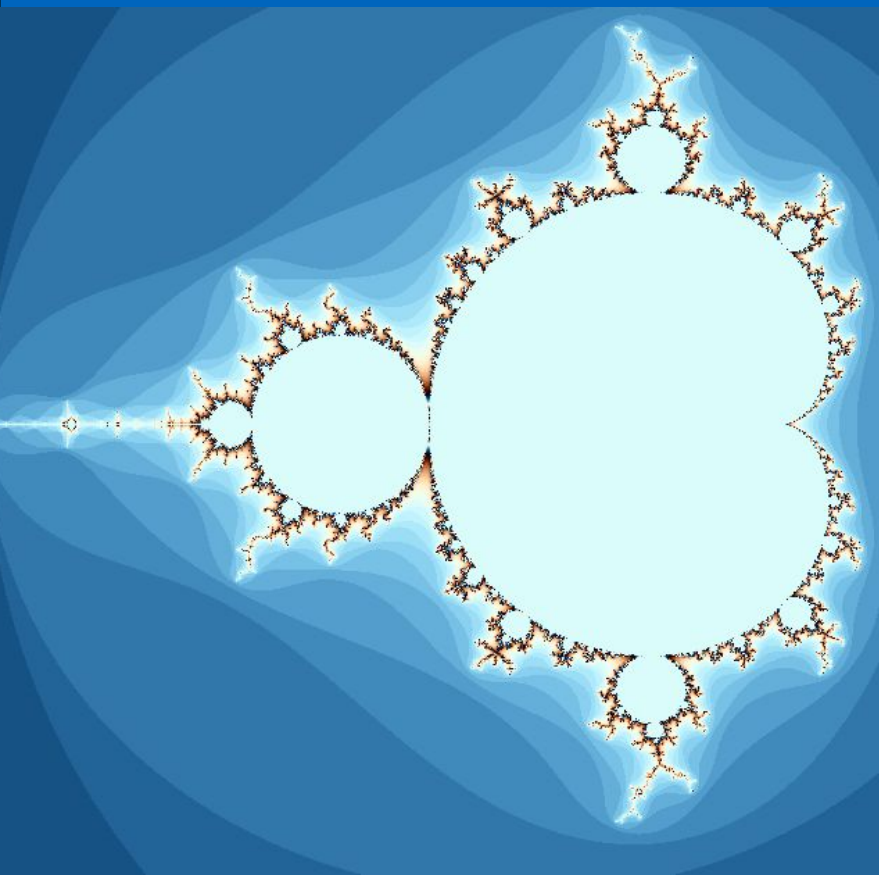


By Knaazuum, CC BY-SA 3.0,  
<https://commons.wikimedia.org/w/index.php?curid=3465954>

- MPI Kommunikation
- OpenMP
- SIMD
  - Single Instruction Multiple Data
  - Parallelisierung derselben Prozessorinstruktion auf unabhängigen Daten
  - Assembler-Ebene



# Ein einfaches Problem...

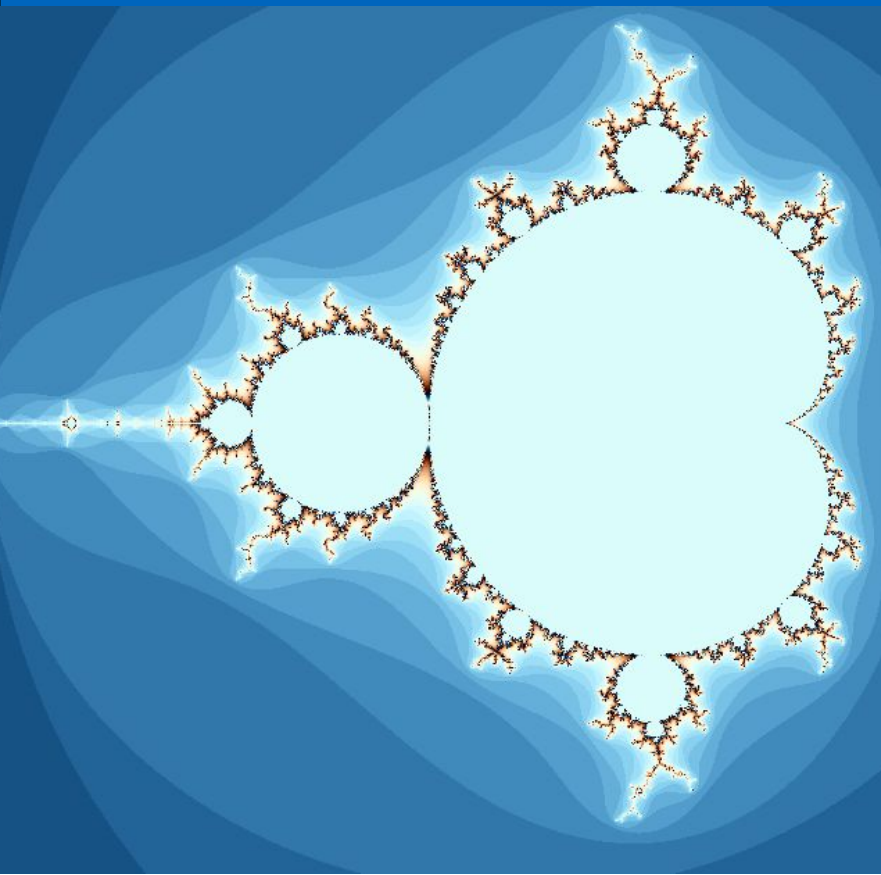


## Allgemein:

- Teilmenge der komplexen Zahlen
- Berechnung:  $z_{i+1} = z_i^2 + c$   
wiederholt anwenden für komplexe Zahl  $c$  ( $z_0 = 0$ )
- $|z_i|$  für beliebig große  $i$  endlich  
 $\Rightarrow c$  ist in der Mandelbrotmenge

## Programm:

- Maximale Iterationsanzahl  $n$  muss beschränkt werden. Hier gilt  $x \leq n$ .  
 $\rightarrow |z_n| \leq 2 \Rightarrow c$  ist in der Mandelbrotmenge  
 $\rightarrow |z_x| > 2 \Rightarrow c$  ist nicht in der Mandelbrotmenge
- Darstellung von  $c$  in der komplexen Ebene:  
Jede Iteration  $x$  bekommt einen Farbwert zugewiesen. Einfärbung der Koordinaten von  $c$  bzgl. dem kleinsten  $x$  mit  $|z_x| > 2$



- **Ziel:** Minimale Rechenzeit
- Bei der Parallelisierung stoßen wir auf ein Problem:
  - Aufteilung in gleich große Bereiche  
⇒ Ungleiche Rechenzeiten der Knoten

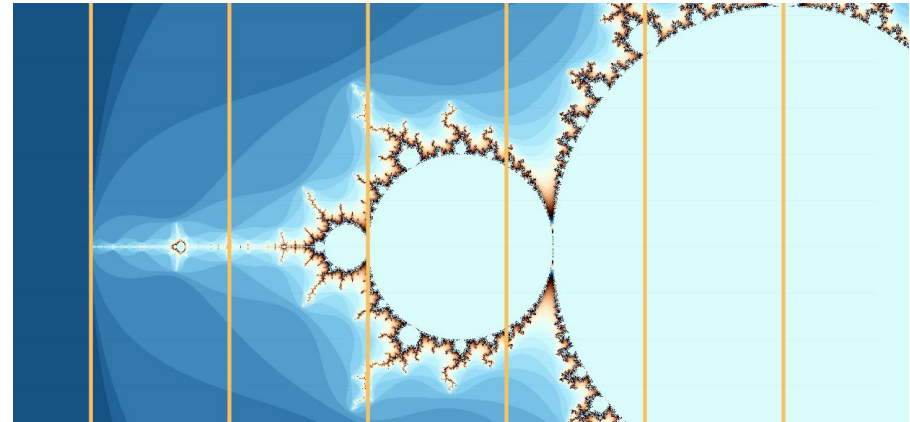
! Verschwendete Rechenzeit im Idle Mode

# Lösungsansatz



$$\begin{aligned} \text{minimiere } \max_{i \in N}(t_i) \text{ mit } \sum_{i \in N} t_i &= C \\ \Leftrightarrow t_1 = \dots = t_n &= \frac{1}{C} \end{aligned}$$

- **Ziel:** Minimierung der maximalen Rechenzeit, volle Auslastung aller Knoten
- Wird erreicht durch gute Lastverteilung  
→ Sonst: untätige Rechenknoten



**Ziel:** Minimierung der maximalen Rechenzeit, volle Auslastung aller Knoten

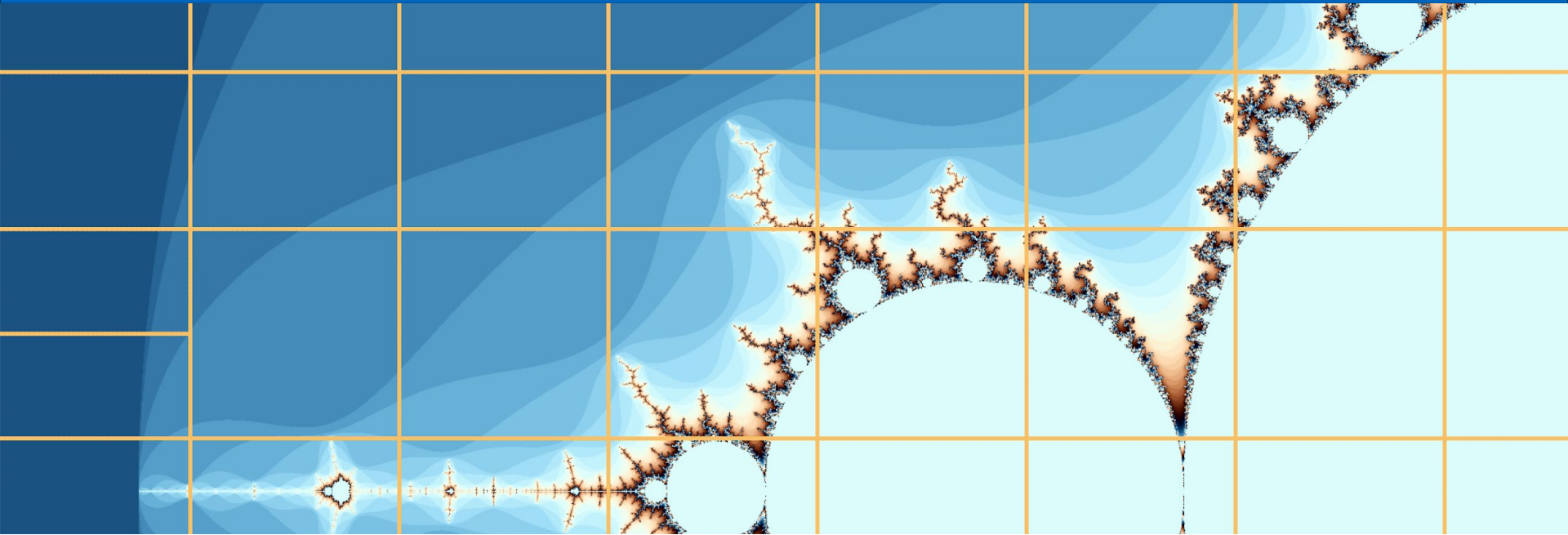
- globaler Ansatz:
  - Aufteilung in Zeilen und Spalten
  - “Gitter”
  - Probleme bei Primzahlen
- rekursiver Ansatz:
  - Halbieren der Region
  - Wiederholen bis genug Teile erzeugt

**Ziel:** Minimierung der maximalen Rechenzeit, volle Auslastung aller Knoten

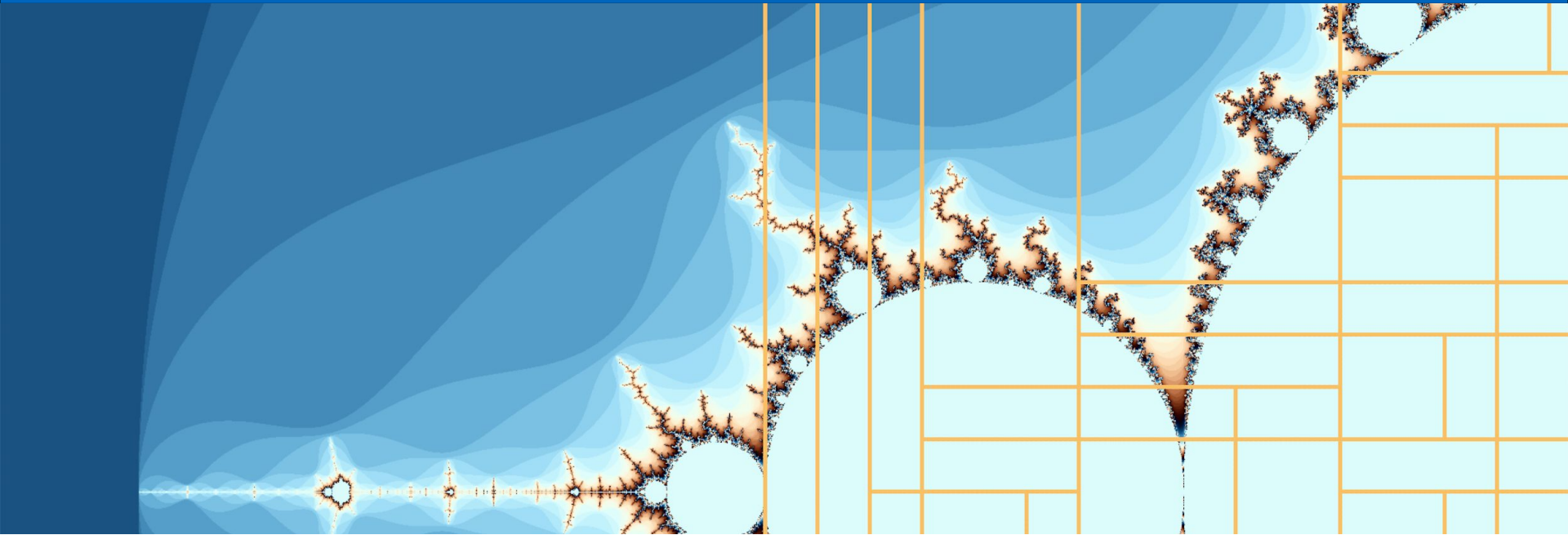
- globaler Ansatz:
  - Aufteilung in Zeilen und Spalten
  - “Gitter”
  - Probleme bei Primzahlen
- rekursiver Ansatz:
  - Halbieren der Region
  - Wiederholen bis genug Teile erzeugt



Wie entscheidet man, wo geteilt wird?



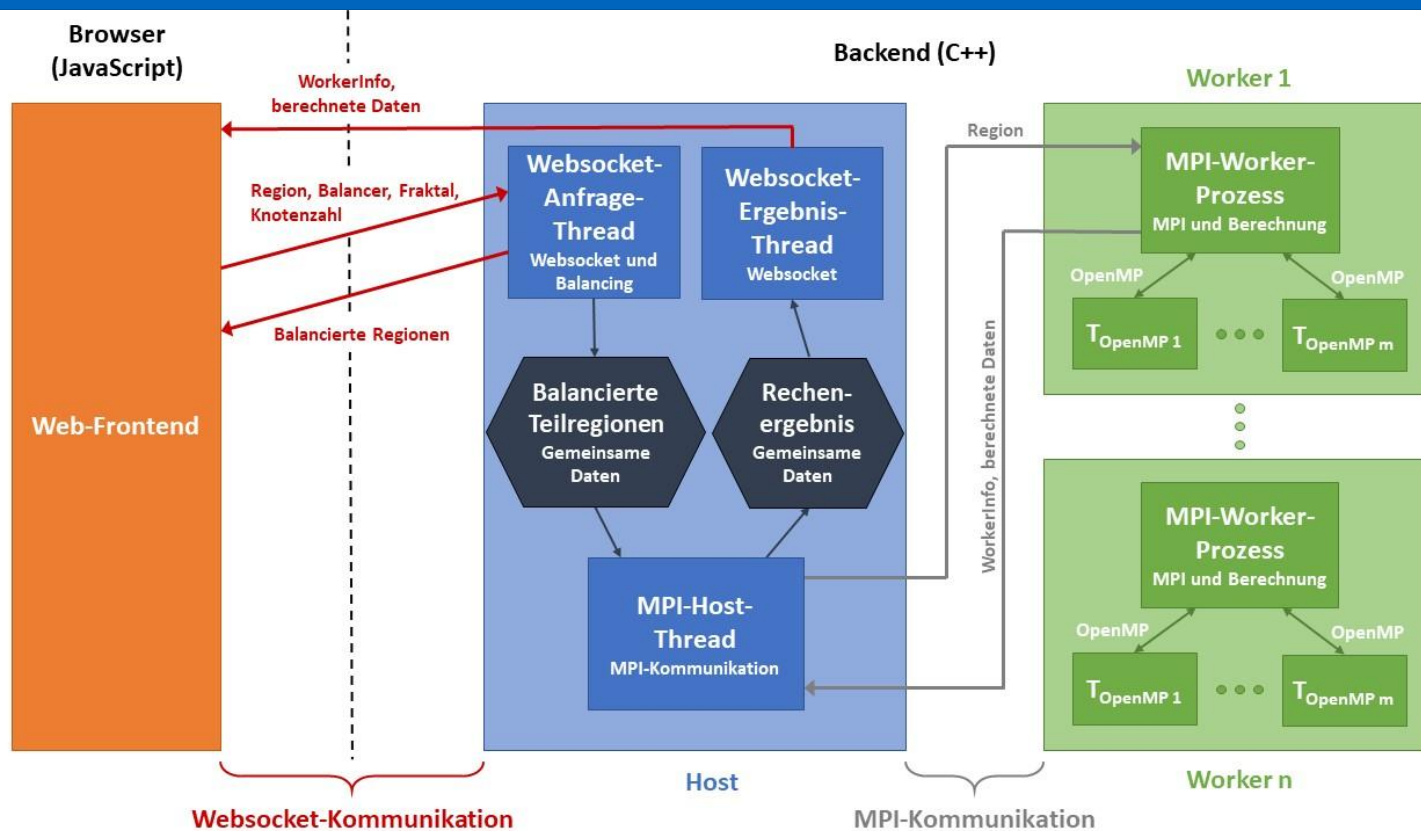
- Aufteilung in gleich große Bereiche
- Einfache Umsetzung, schnelle Aufteilung
- Meist schlechte Lastverteilung



- Berechnung des Fraktals in deutlich niedrigerer Auflösung → “Vorhersage”
- Aufteilung in gleich aufwendige Bereiche
- Bessere Verteilung der Last
- Vorhersage ungenau und aufwendig

# Architektur

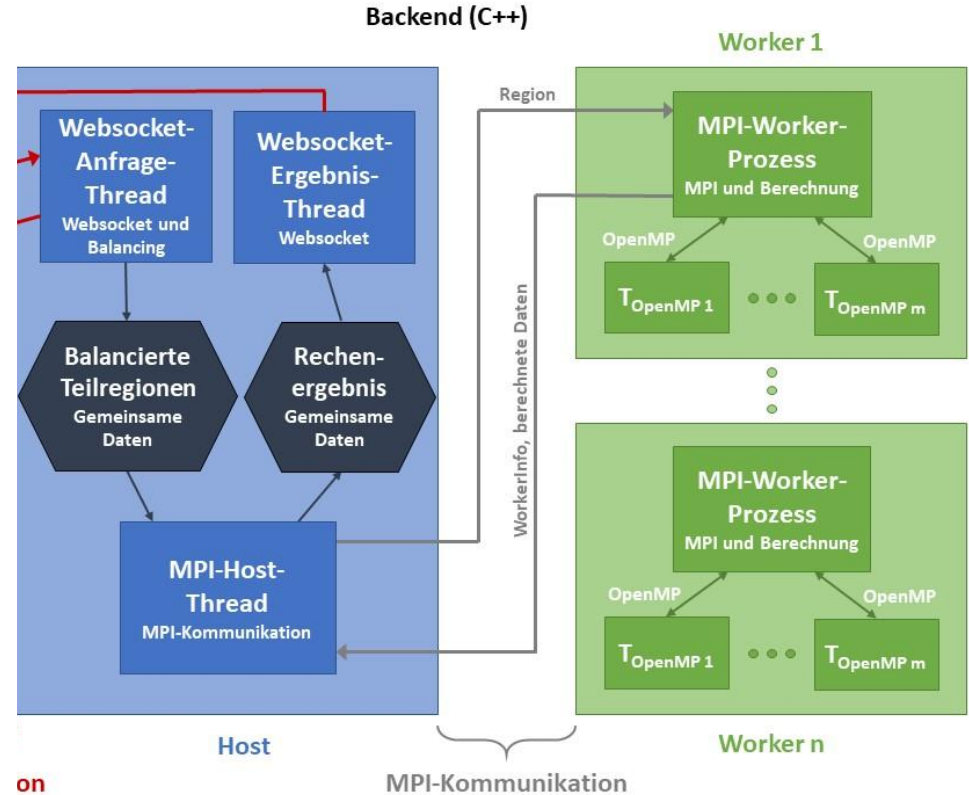
# Architekturübersicht



# Implementierung

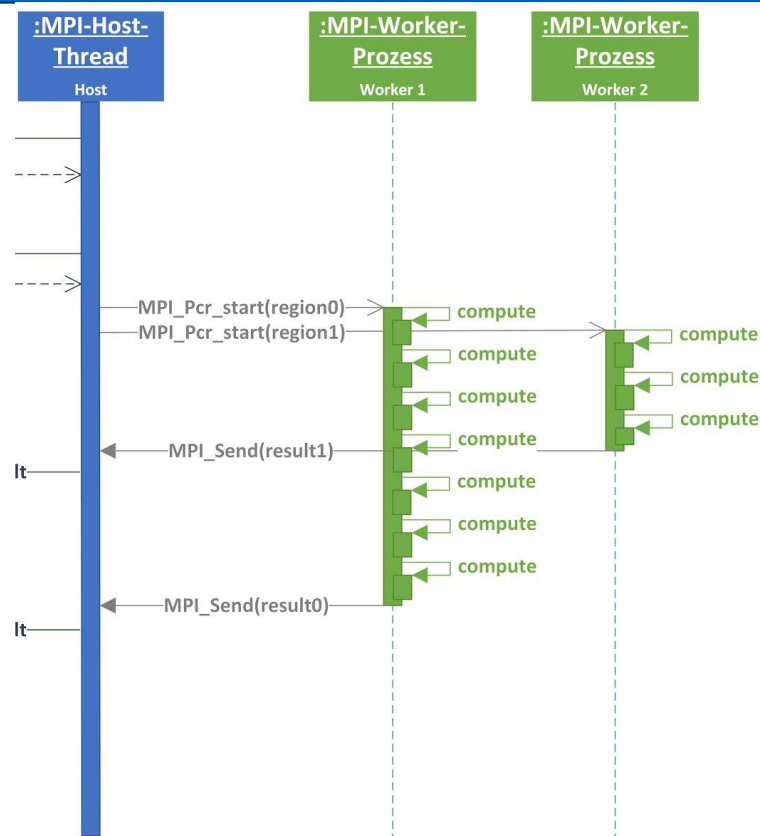


- Nur der Hauptthread in Host und Worker tätigt MPI-Aufrufe  
(→ MPI\_THREAD\_FUNNELED)



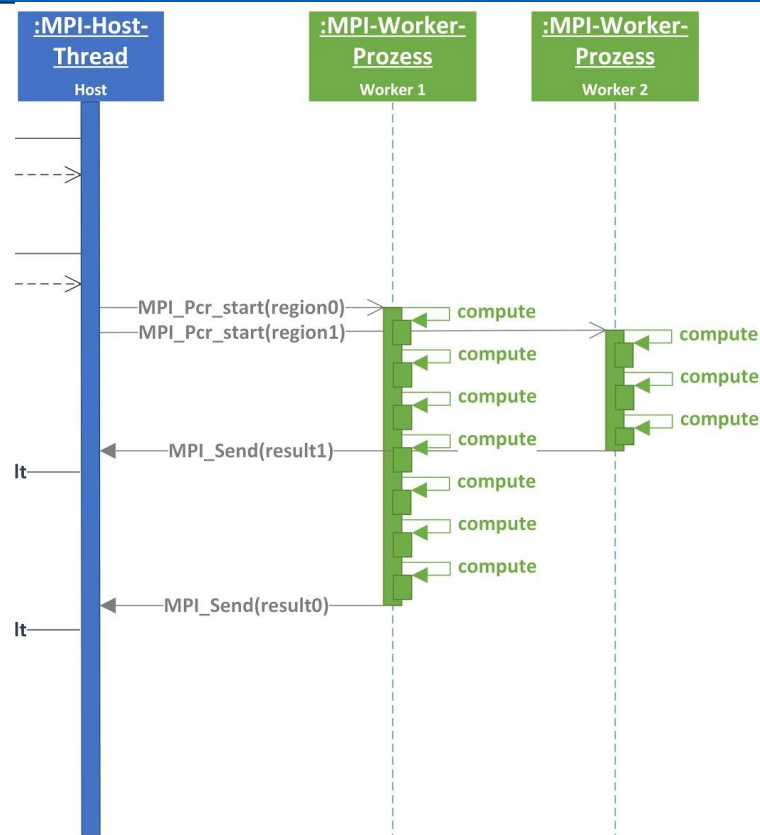
## Senden der Subregionen im Host

- Persistent Communication Request
- Modus: nicht-blockierend
  - Gleichzeitiges Starten aller Sendeoperationen
  - Gemeinsamer Abschluss mit `MPI_Waitall()`



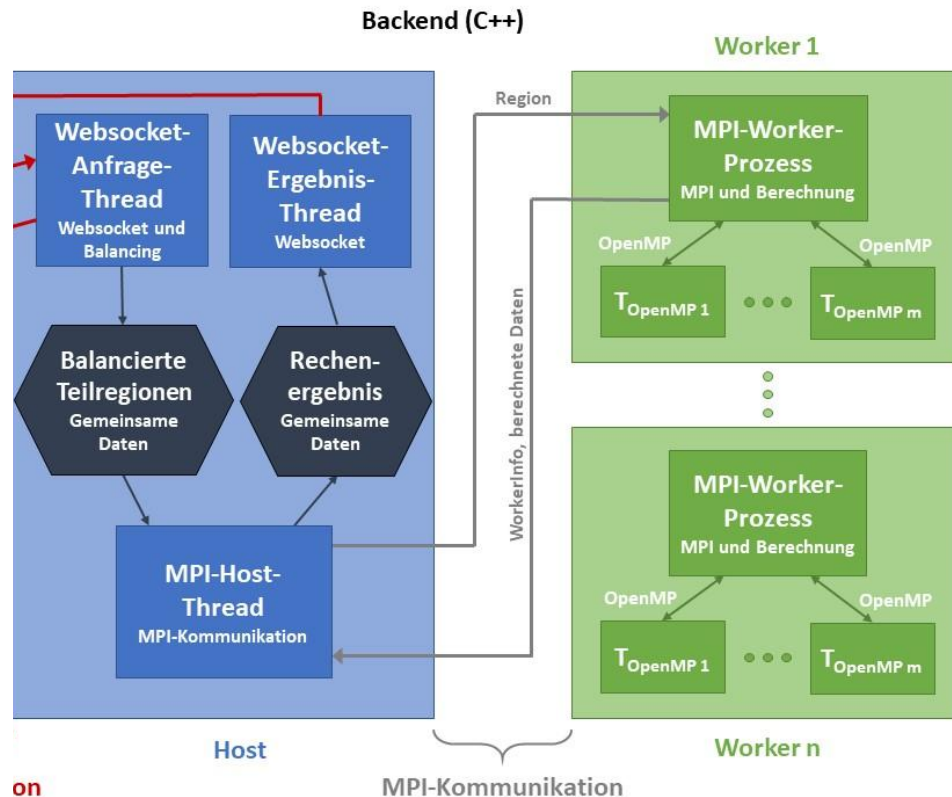
## Empfangen der Subregion im Worker

- Persistent Communication Request
- Modus: nicht-blockierend
  - Abbruch laufender Berechnungen

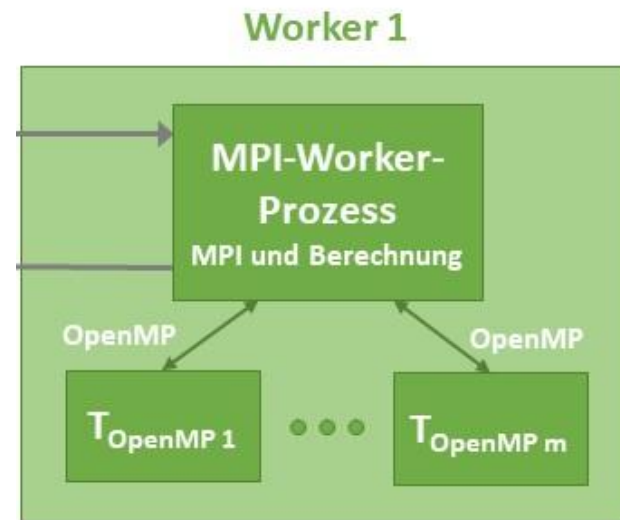


## Busy waiting im MPI-Host-Thread

- Reaktion auf neue Arbeitsaufträge und eingehende Rechenergebnisse



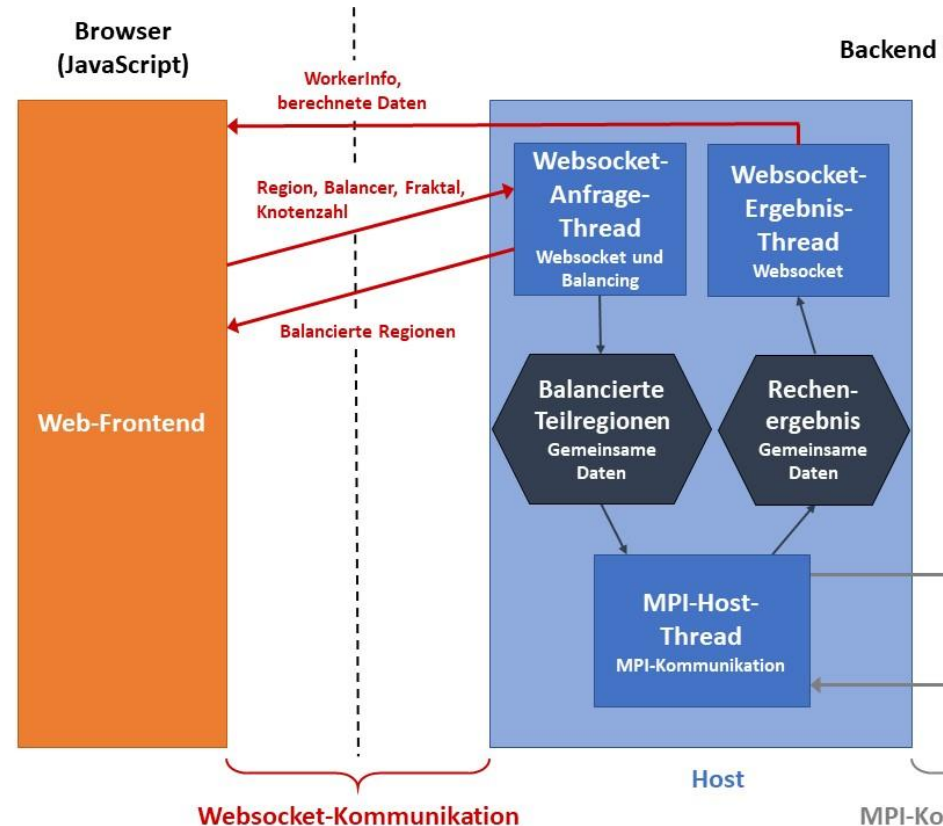
- Weitere Parallelisierung der Hauptberechnungsschleife über mehrere Rechenkerne auf Threadebene im Worker
- Hauptunterschied zu MPI: Worker bekommt nur eine Teilregion und muss deren Berechnung nochmals auf Threads aufteilen
- Scheduling
  - Nonmonotonic
  - Dynamic



- Standard 80 bit
  - 64 bit, 32 bit
- SIMD
  - Feingranulare Parallelisierung
  - Rechnet bis zu vier Punkte der Mandelbrotmenge parallel
    - 64 bit: 2 Lanes
    - 32 bit: 4 Lanes
  - Auch hier: maximale Rechenzeit aller vier Punkte

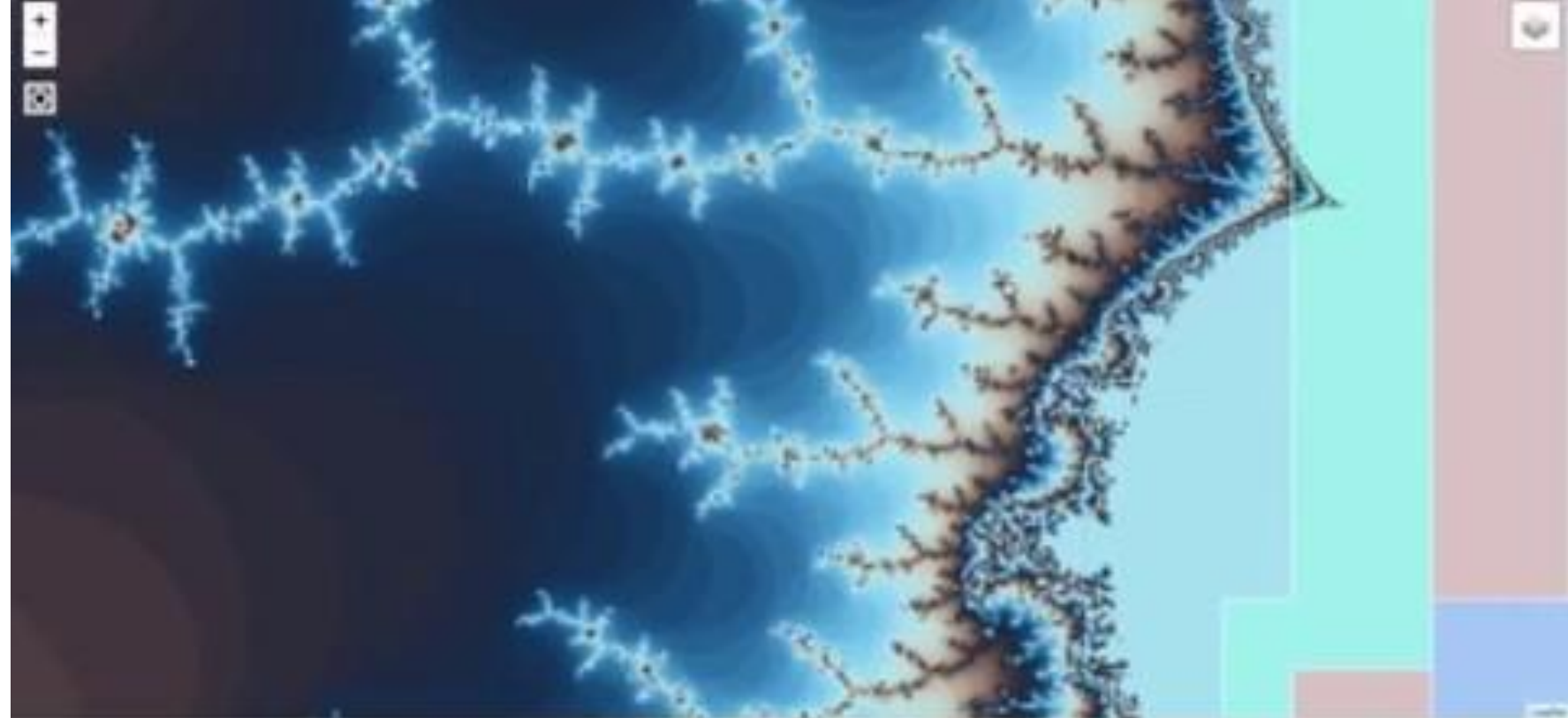
## Darstellung in Webbrowser

- “Live” Erfahrung der Rechenzeit  
→ Websockets
- JSON nativ, effizient in JS

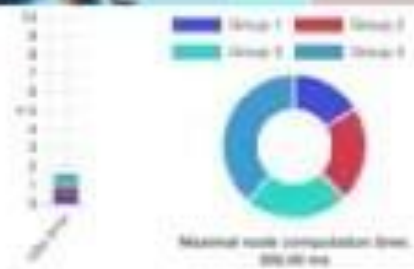


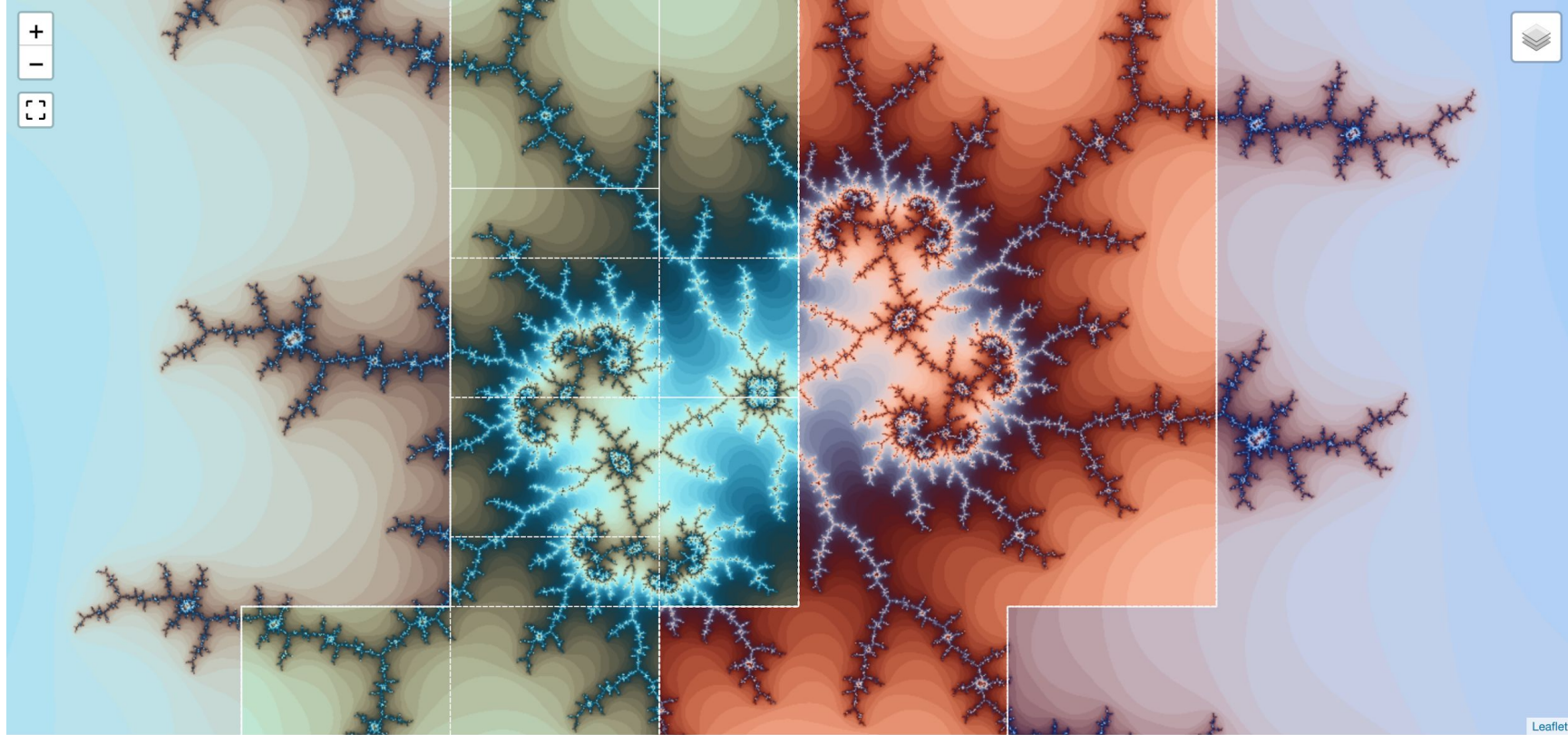
# Frontend





| Loadbalancer           | Implementation                |
|------------------------|-------------------------------|
| Round Robin            | 60 sec Round Robin            |
| Least Connections      | 60 sec Least Connections      |
| Weighted Round Robin   | 60 sec Weighted Round Robin   |
| IP Hash                | 60 sec IP Hash                |
| Random                 | 60 sec Random                 |
| Weighted Random        | 60 sec Weighted Random        |
| Round Robin (Weighted) | 60 sec Round Robin (Weighted) |



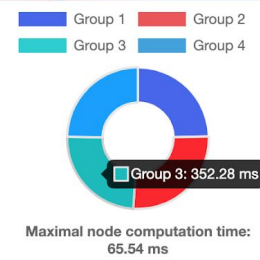
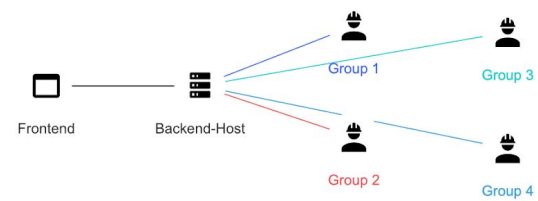


**Loadbalancer**

|                                      |  |
|--------------------------------------|--|
| Naive Balancer                       |  |
| Column Balancer                      |  |
| Prediction Balancer                  |  |
| Recursive Naive Balancer             |  |
| <b>Recursive Prediction Balancer</b> |  |

**Implementation**

|                     |  |
|---------------------|--|
| 80 bit float        |  |
| 32 bit float        |  |
| <b>64 bit float</b> |  |
| SIMD 32 bit         |  |
| SIMD 64 bit         |  |





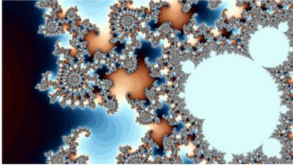

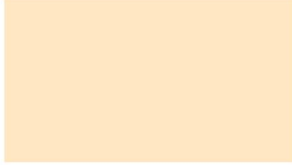
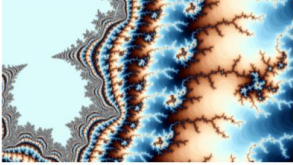
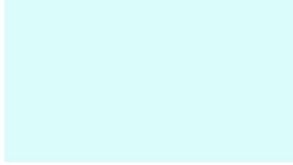


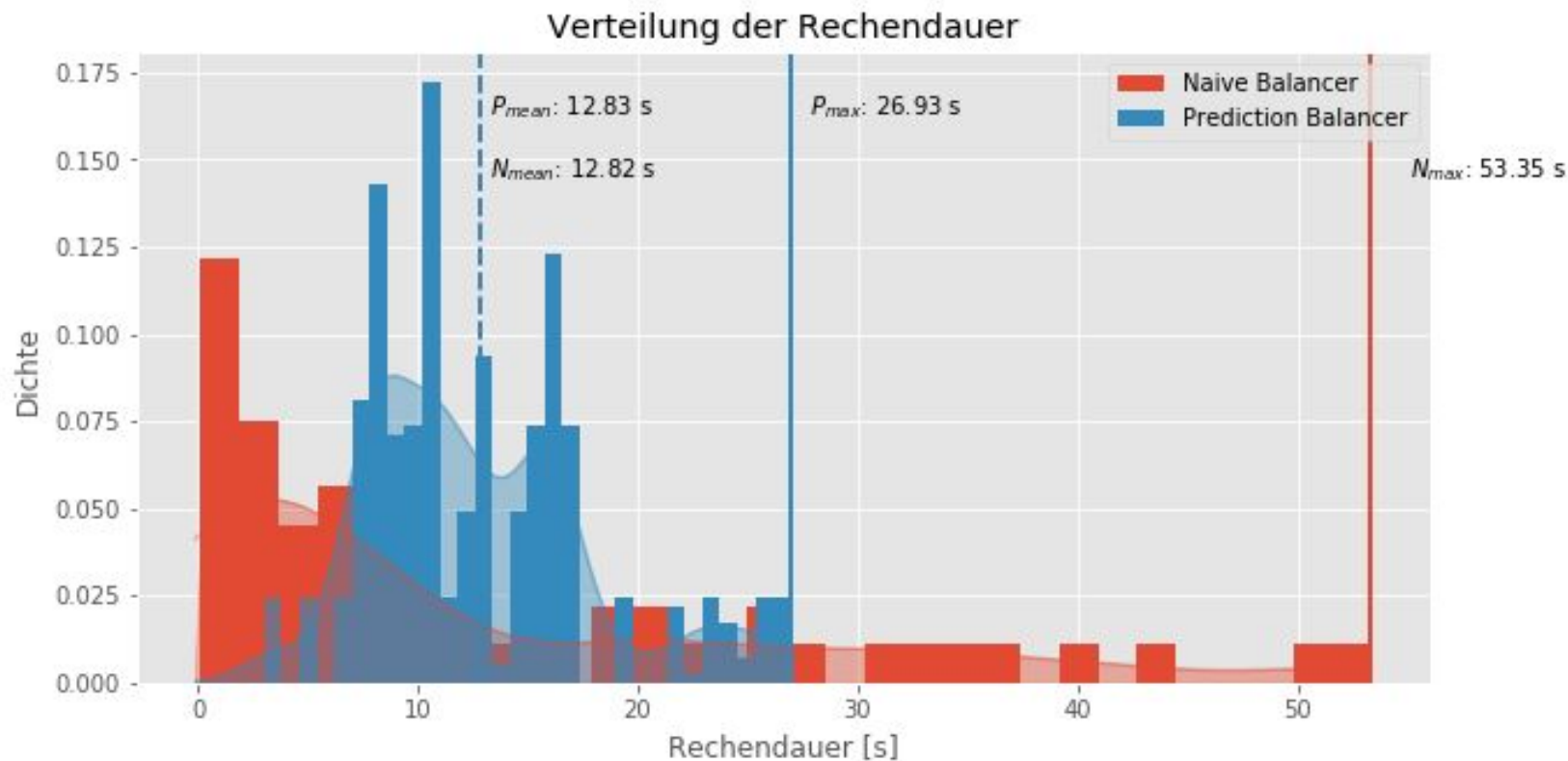
# Evaluation

- Datenerhebung auf den ODroids des TUM HimMUC Cluster (40 Knoten)
- MPI Implementierung: OpenMPI
- Unterteilung der Regionen in Klassen
  - Rand
  - Außen
  - Innen



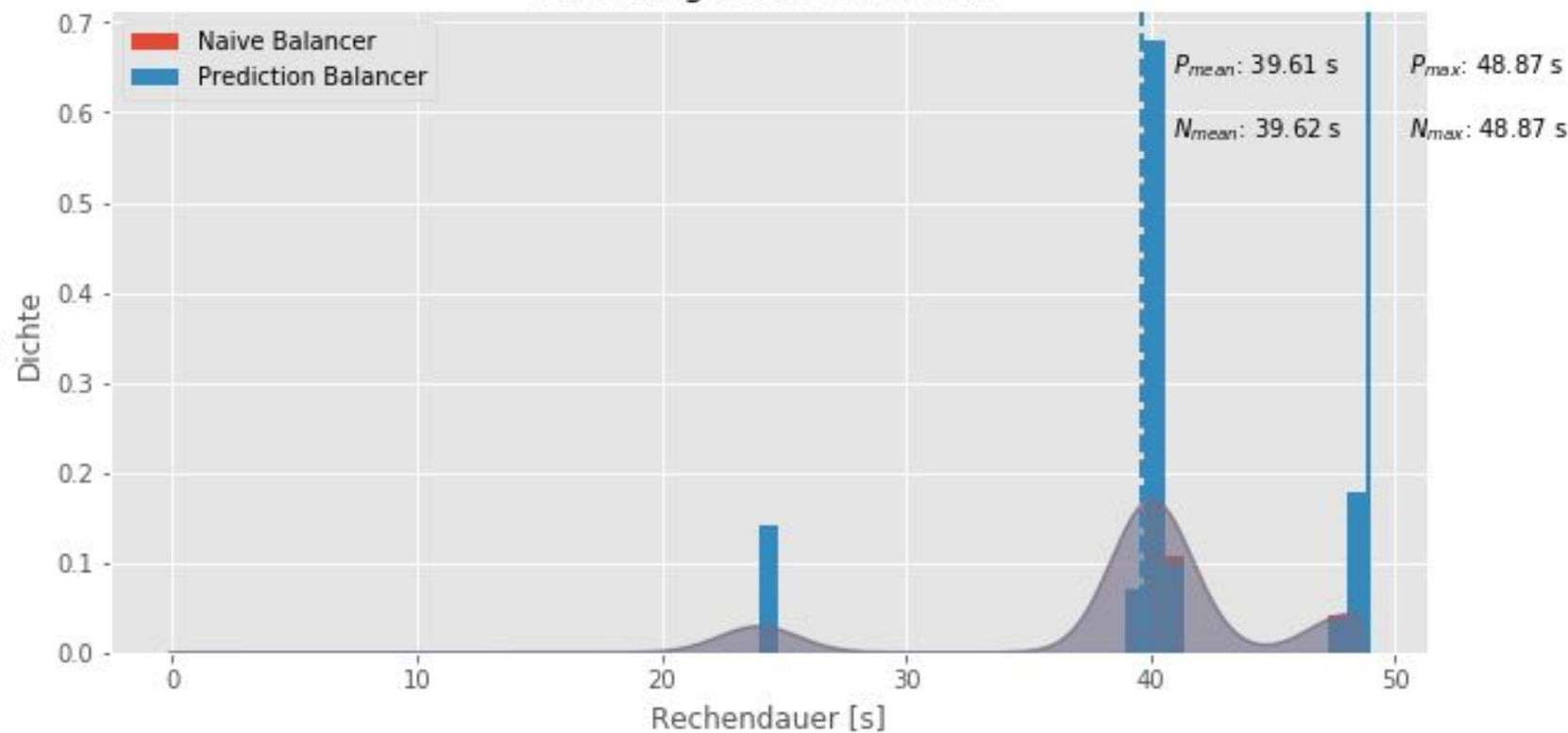
HimMUC Cluster des CAPS-Lehrstuhl TUM  
<https://www.caps.in.tum.de/himmuc/>

| Außerhalb   | Randregion   | Innerhalb   |
|---|--|---|
|  |  |  |
| $(-1.38866, 1.01930, 2)$  | $(-1.13528, -0.34512, 2)$  | $(-0.21118, 0.07617, 3)$  |
|  |  |  |
| $(-0.86151, 0.37891, 11)$   | $(-0.73341, 0.21651, 11)$  | $(-1.75620, 0.00049, 10)$   |
|  |  |  |
| $(-0.76671, -0.16323, 25)$  | $(-0.04666, 0.98511, 25)$  | $(-0.11136, -0.04529, 25)$  |



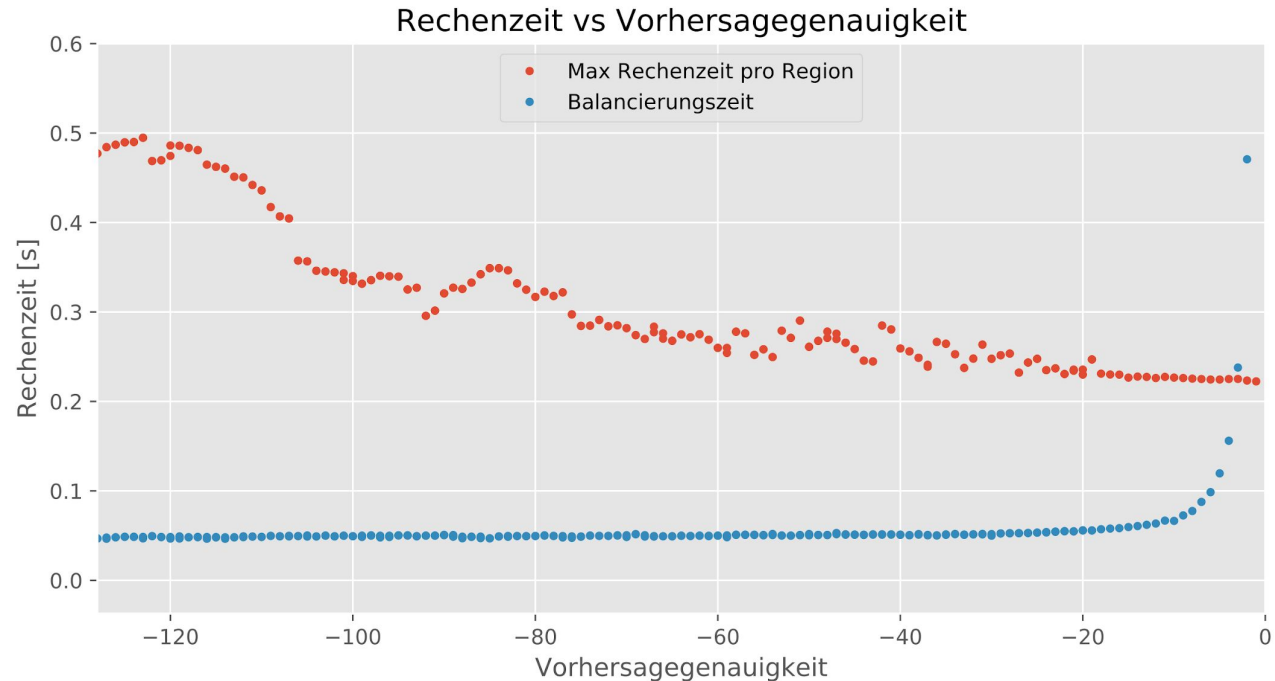


Verteilung der Rechendauer



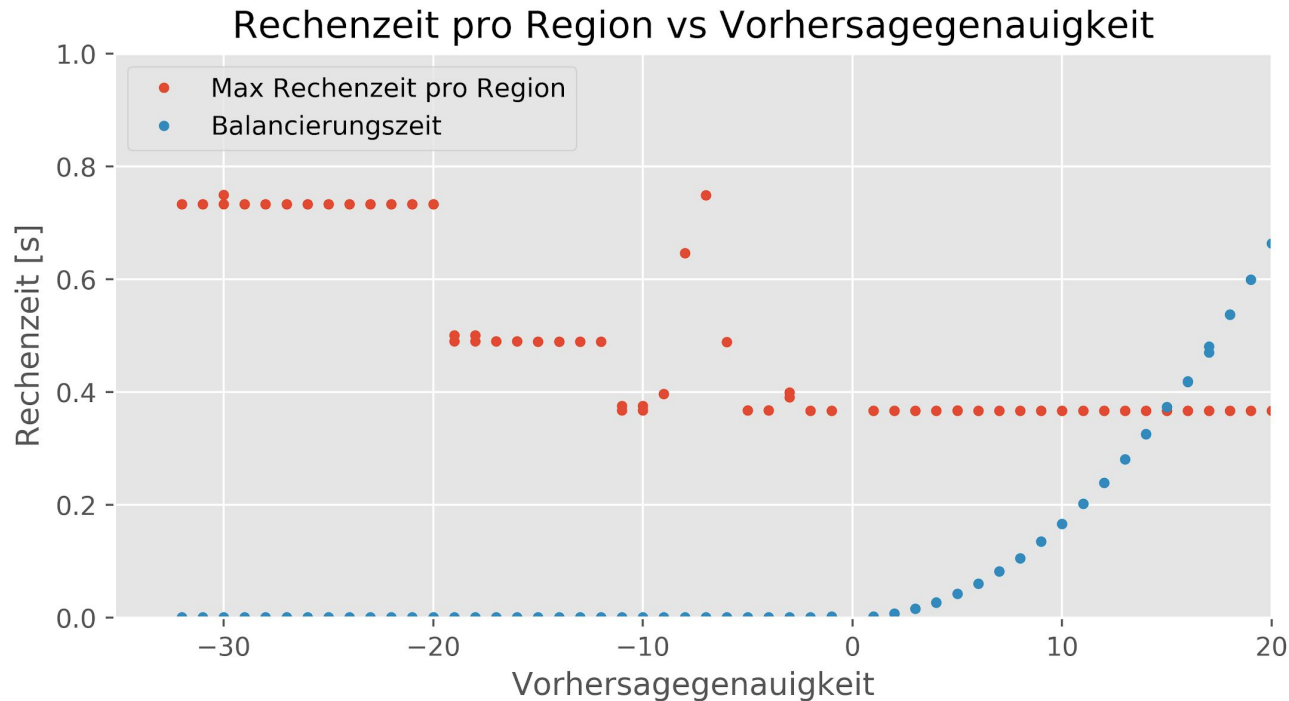
erreichbares Maximum

⇒ Teiler = 2

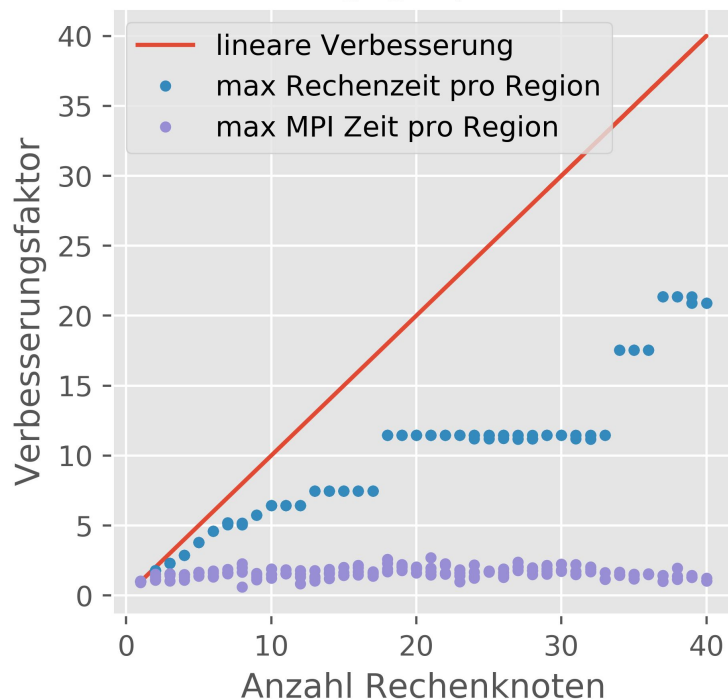




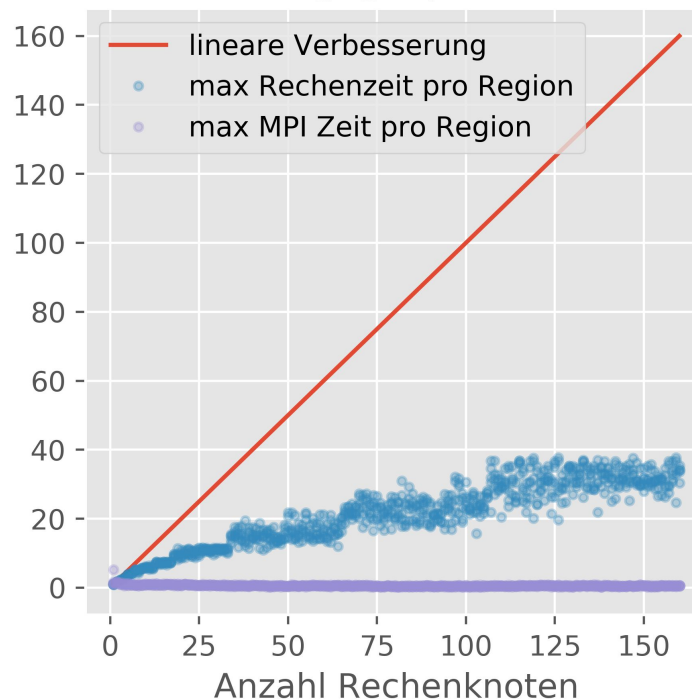
⇒ Teiler = 64

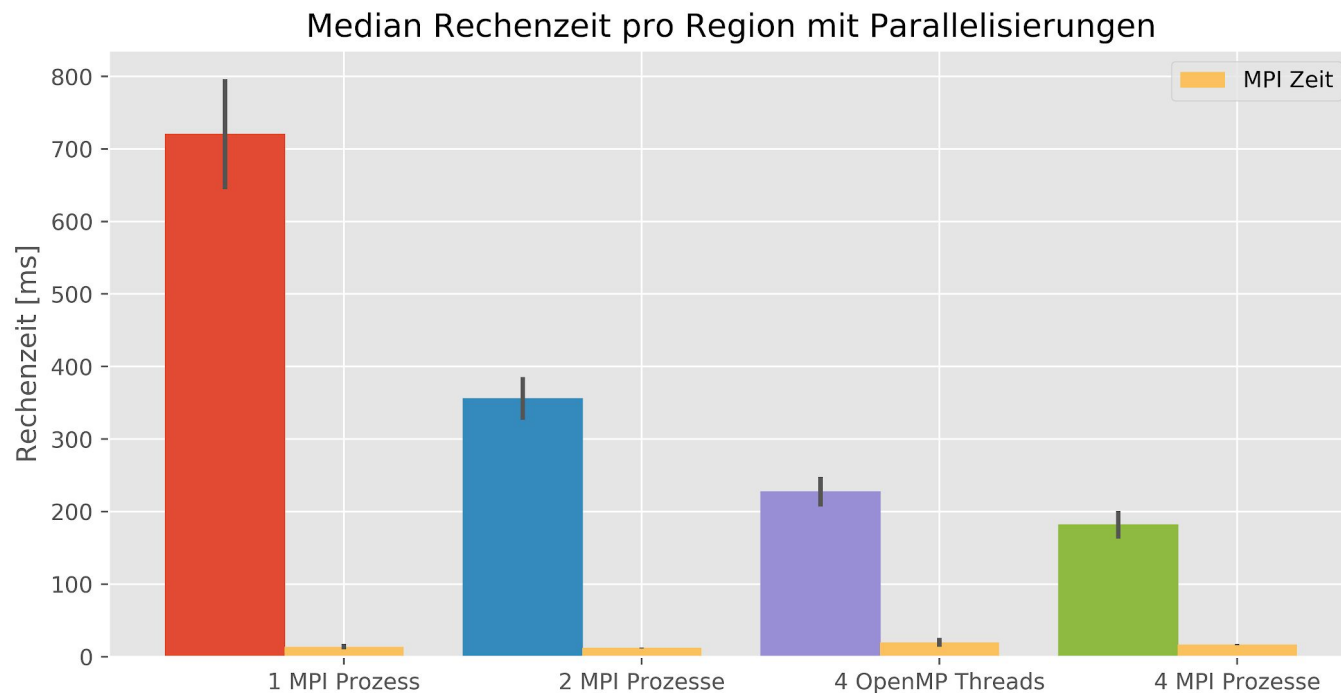


## Skalierungsgraph am Rand



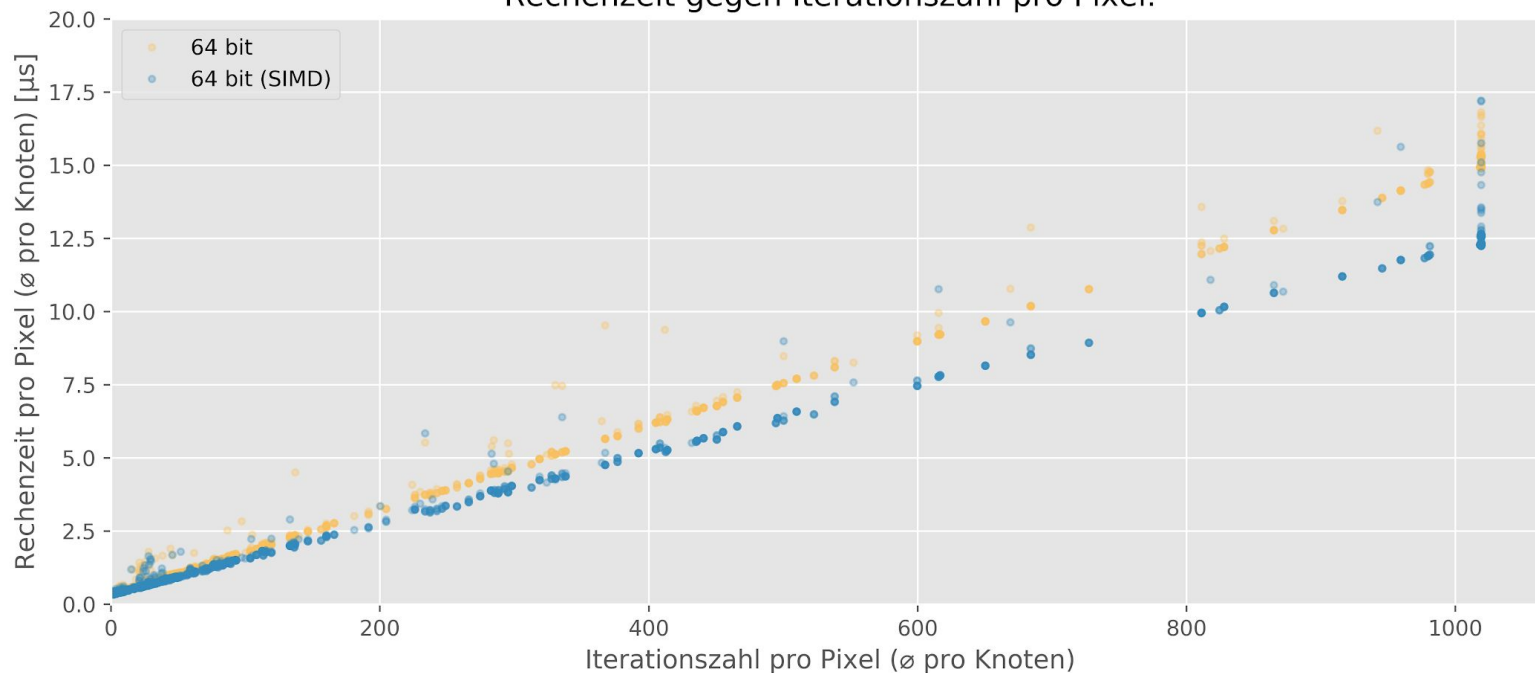
## Skalierungsgraph am Rand





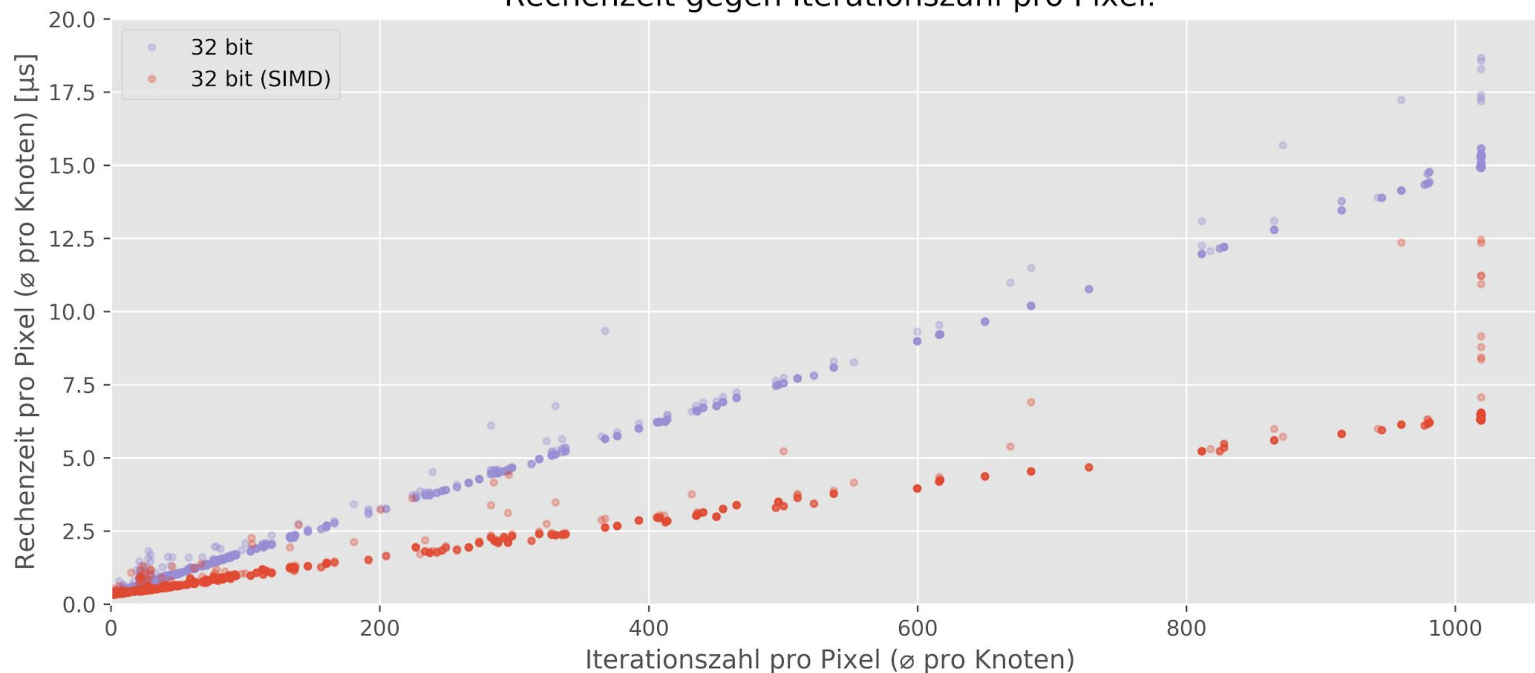
# Berechnung (SIMD 64 bit)

Rechenzeit gegen Iterationszahl pro Pixel.



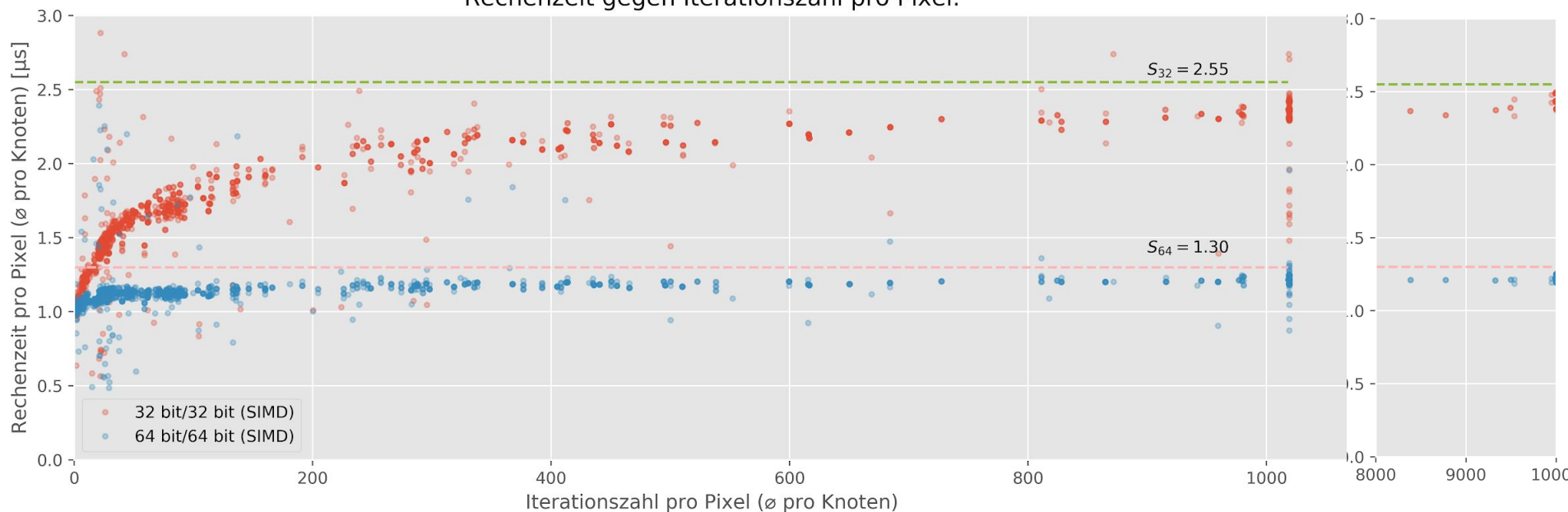
# Berechnung (SIMD 32 bit)

Rechenzeit gegen Iterationszahl pro Pixel.



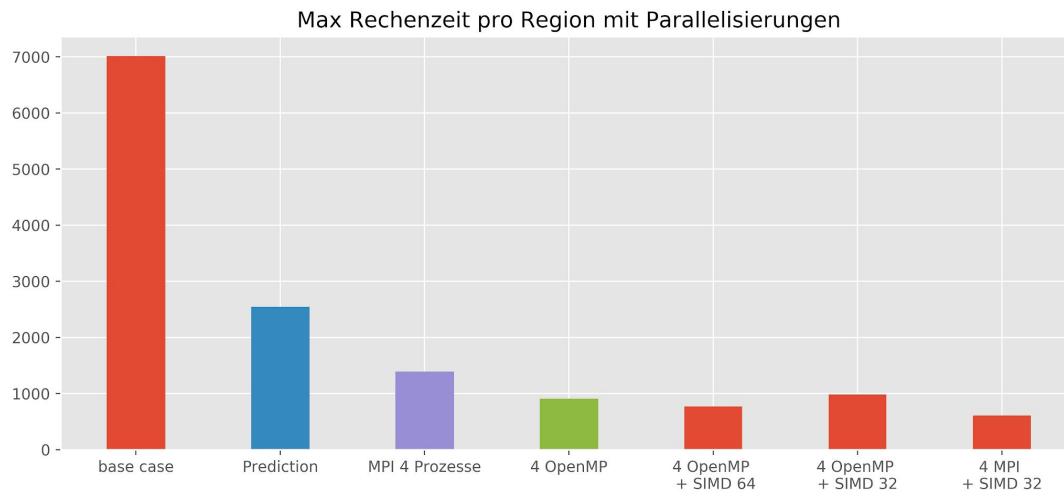
# Berechnung (SIMD)

Rechenzeit gegen Iterationszahl pro Pixel.



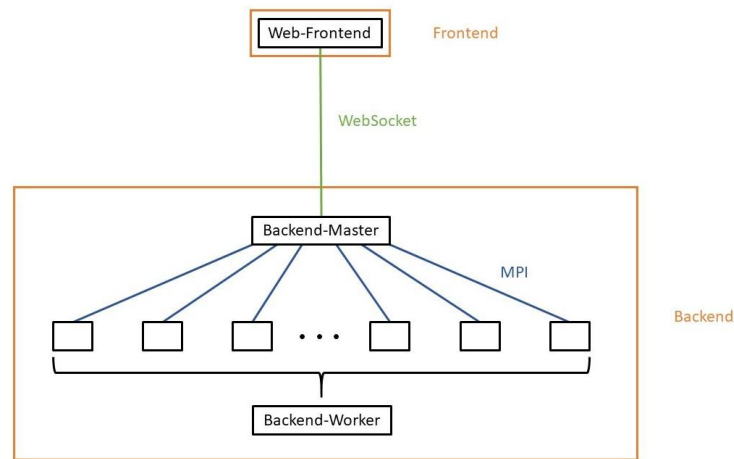
# Ausblick und Zusammenfassung

- Lastbalancierung reduziert  
Rechenzeit: **2.76 Speedup**
- MPI Kommunikationszeit fast  
konstant  
→ MPI Zeit vernachlässigbar
- 4 Core MPI:  
**5.06 Speedup**
- 4 Core OpenMP:  
**7.78 Speedup**
- 4 Core OpenMP + SIMD 64 bit :  
**9.1 Speedup**
- 4 Core OpenMP + SIMD 32 bit :  
**7.15 Speedup**
- 4 MPI + SIMD 32 bit:  
**11.59 Speedup**





- Verringerte max Rechenzeit durch Lastbalancierung
  - Naive Strategie
  - Strategie mit Vorhersage
- Parallelisierung interaktiv dargestellt
  - MPI
  - OpenMP
  - SIMD



- Standardmäßig kleinere Unterteilung
- Mehrbenutzerverwaltung
- komplexere Fraktale
  - Herausforderung für die Lastbalancierung
  - korrelierte Teilregionen
- Verbesserungen an der UI
- Messen von Verzögerungen bei Eingaben durch die Benutzerin
- dynamische Lastbalancierung (stealing)

# Live Demonstration