Vergleich paralleler Datenverarbeitung in Haskell und C++ anhand eines MapReduce Szenarios

Hans Christian Rudolph

Hochschule für Telekommunikation Leipzig

hans-christian.rudolph@hft-leipzig.de

@hcrudolph

12. Oktober 2016

github.com/hcrudolph/oklab-parallel

Gliederung

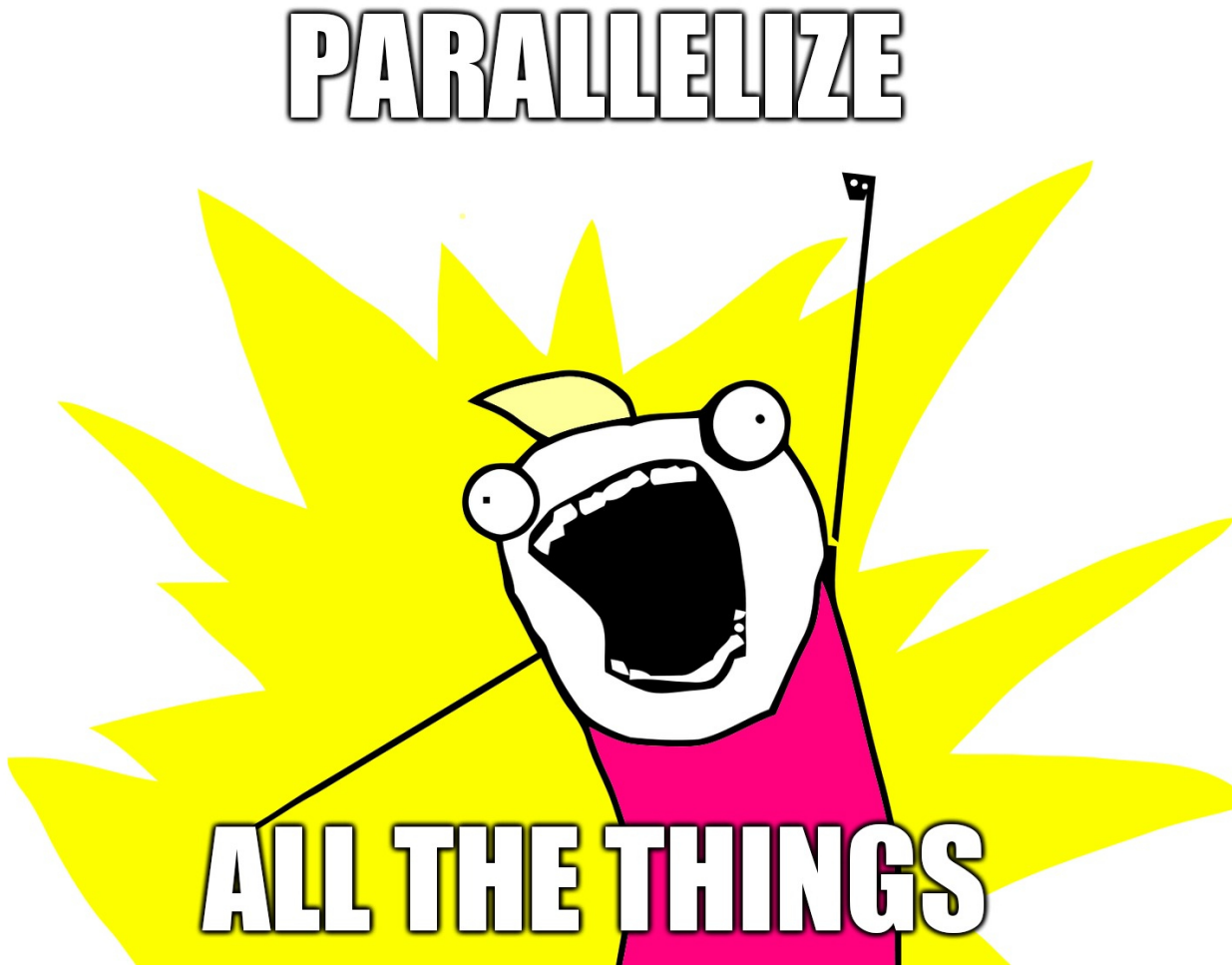
Funktional vs. Imperativ

Haskell

- Rein funktional
- Stark abstrahiert
- Referenzielle Transparenz

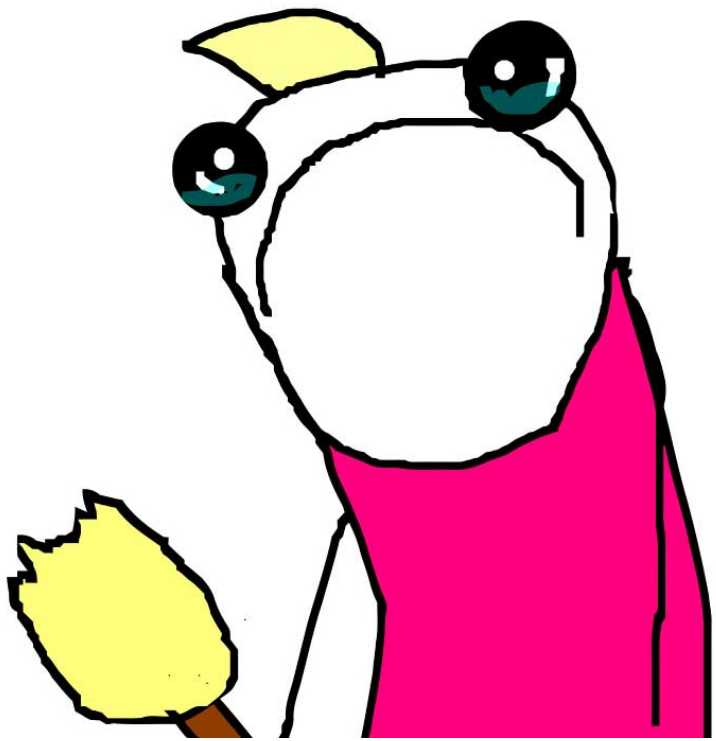
C++

- Multiple Paradigmen
- (Wenn nötig) hardwarenah
- Zero-Cost Abstractions


PARALLELIZE
ALL THE THINGS

Marlow, Peyton Jones und Singh (2009, S.1) stellen fest:

„Plenty of papers describe promising ideas,
but vastly fewer describe real implementations [. . .]".



Begriffserklärung

Nebenläufigkeit

beschreibt die unabhängige Ausführung von Berechnungen. Dies kann sowohl auf paralleler Hardware (mehrere CPUs) als auch auf sequenzieller geschehen. Es ist somit nicht ausgeschlossen, dass auch nebenläufige Aufgaben von paralleler Hardware profitieren. Das Resultat ist nicht-deterministisch. Bsp.: Webserver

Parallelität

beschreibt die *gleichzeitige*, unabhängige Ausführung von Berechnungen auf paralleler Hardware. Das Resultat ist stets deterministisch. Bsp.: Grafikkarte

Begriffserklärung

Nebenläufigkeit

beschreibt die unabhängige Ausführung von Berechnungen. Dies kann sowohl auf paralleler Hardware (mehrere CPUs) als auch auf sequenzieller geschehen. Es ist somit nicht ausgeschlossen, dass auch nebenläufige Aufgaben von paralleler Hardware profitieren. Das Resultat ist nicht-deterministisch. Bsp.: Webserver

Parallelität

beschreibt die *gleichzeitige*, unabhängige Ausführung von Berechnungen auf paralleler Hardware. Das Resultat ist stets deterministisch. Bsp.: Grafikkarte

Begriffserklärung

Nebenläufigkeit

beschreibt die unabhängige Ausführung von Berechnungen. Dies kann sowohl auf paralleler Hardware (mehrere CPUs) als auch auf sequenzieller geschehen. Es ist somit nicht ausgeschlossen, dass auch nebenläufige Aufgaben von paralleler Hardware profitieren. Das Resultat ist nicht-deterministisch. Bsp.: Webserver

Parallelität

beschreibt die *gleichzeitige*, unabhängige Ausführung von Berechnungen auf paralleler Hardware. Das Resultat ist stets deterministisch. Bsp.: Grafikkarte

Szenario

- Aggregationsfunktion
- Lokal gespeicherte Textdateien
- Schlüsselfelder
- Aggregationsfelder
- Summiere die Aggregationsfelder für jeden distinkten Schlüssel

MapReduce

- Vorgestellt von den Googlern Dean und Ghemawat (2004)
- Ziel: Verarbeitung großer Datenmengen auf Clustern handelsüblicher Hardware
- Google Framework realisiert Datenverteilung, -sicherung und Fehlerbehandlung während der Übertragung
- Auch auf einzelnen Mehrkernrechnern anwendbar (vgl. Ranger et al. 2007, Talbot, Yoo und Kozyrakis 2011)
- Grundlegendes Programmiermodell eignet sich jedoch für eine Vielzahl von Problemen

Map Funktional

Definition

Das Funktional Map wendet eine gegebene Funktion `f` auf sämtliche Elemente einer Sequenz an. Das Resultat ist die Sequenz der einzelnen Funktionswerte.

Beispiele

```
map (*2) [1,2,3]       = [2,4,6]
map toUpper "foobar" = "FOOBAR"
```

Reduce/Fold Funktional

Definition

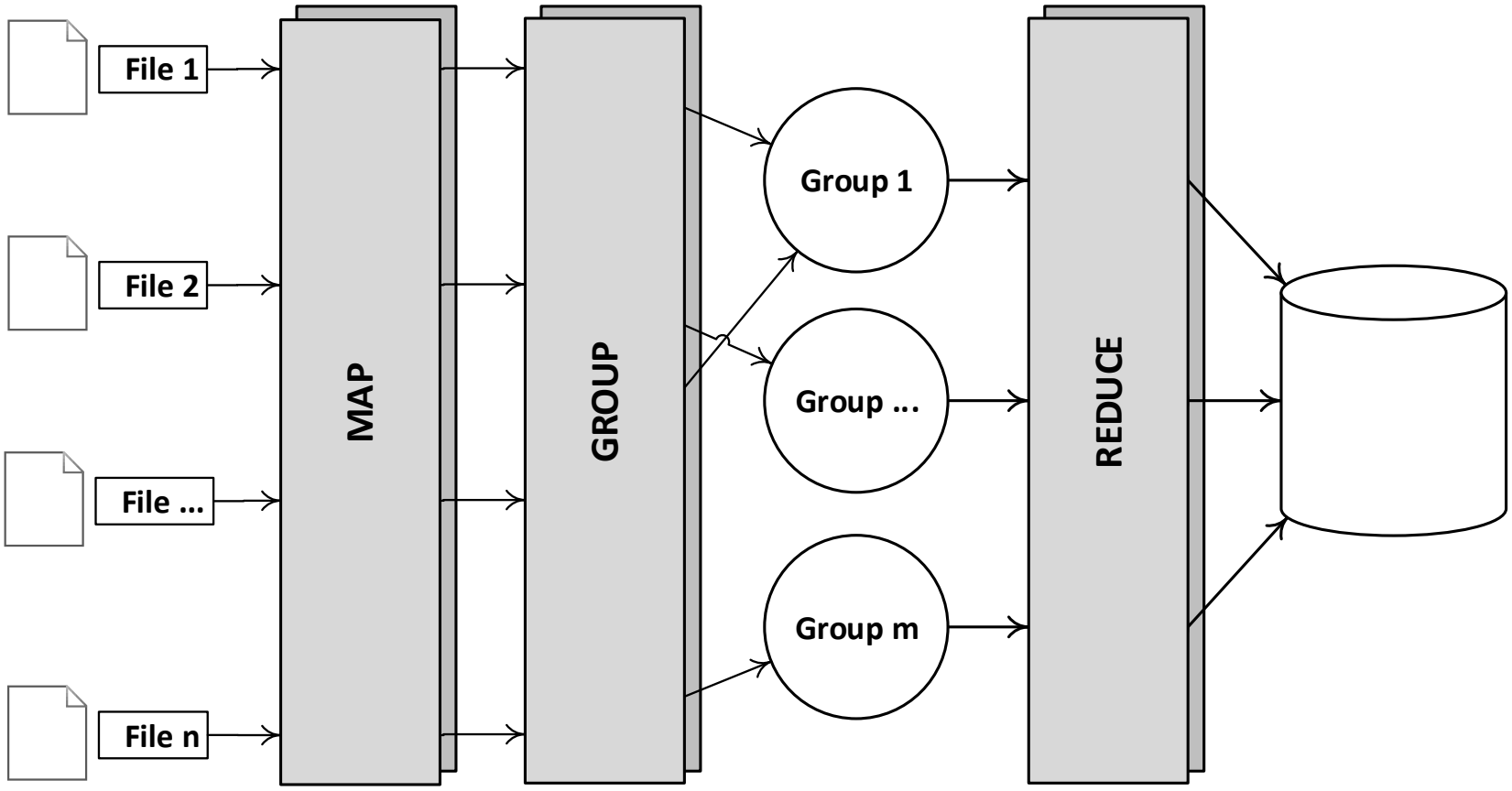
Das Funktional Reduce/Fold wendet eine binäre Funktion auf jedes Element einer Sequenz an. Eingangsparameter sind ein definierter Start-Wert, Akkumulator genannt, und das jeweilige Element.

Beispiel

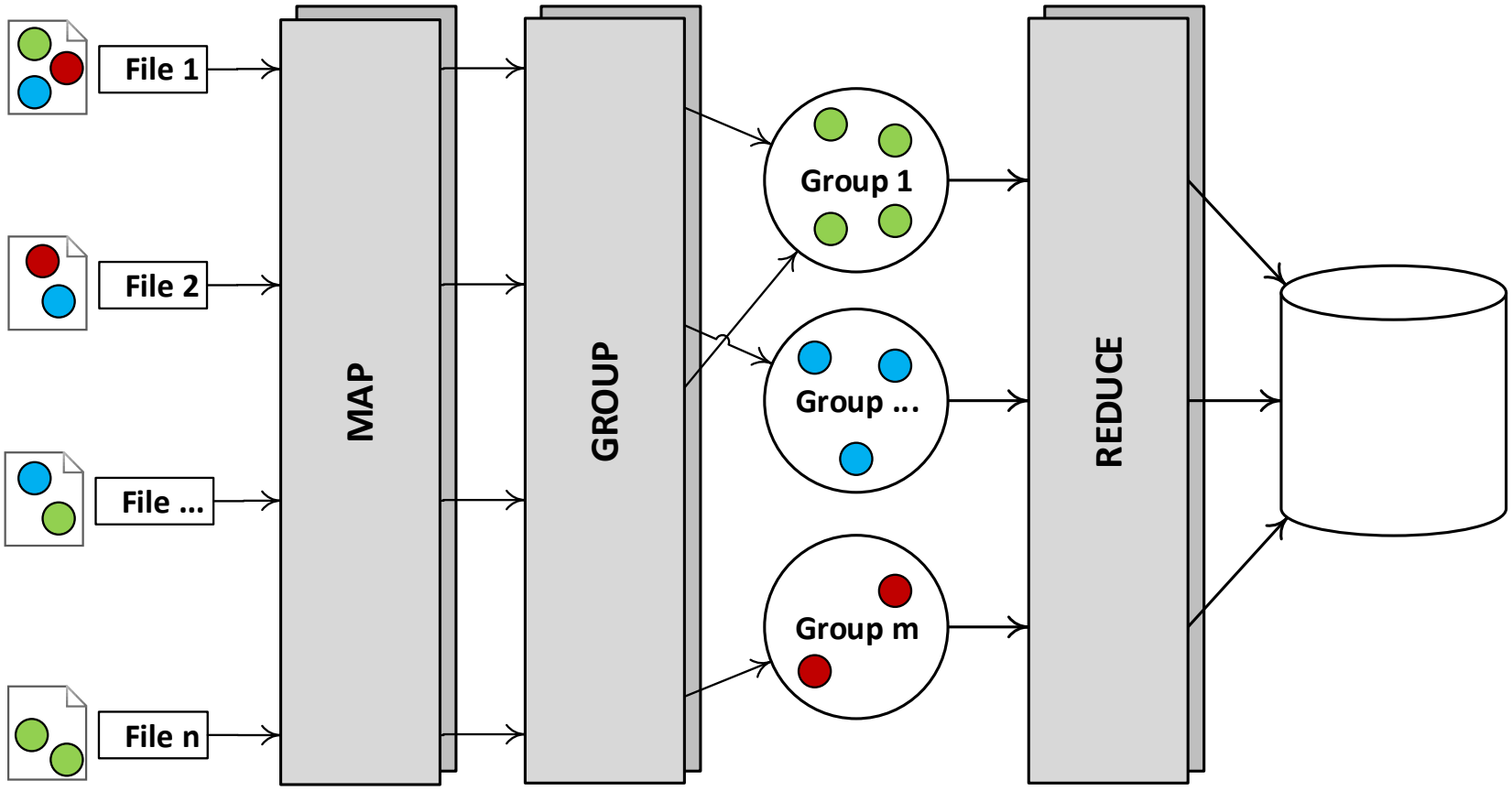
```
foldl (+) 0 [1,2,3] = 6

-- foldl (+) 0           [1,2,3] = ...
-- foldl (+) (0+1)       [2,3]   = ...
-- foldl (+) ((0+1)+2)   [3]     = ...
-- foldl (+) (((0+1)+2)+3) []    = 6
```

Logische Struktur I



Logische Struktur II



Funktionsblöcke

1. **Map** = Parsen der Textdateien
2. **Group** = Sortieren der Datenstrukturen nach Schlüssel
3. **Reduce** = Summieren der Aggregationsfelder

Flynnsche Klassifizierung

Flynn (1966) unterteilt Rechner nach ihrer Prozessoranzahl und den von ihnen gesteuerten Kontroll- und Datenflüssen in vier Kategorien:

- Single Instruction Stream–Single Data Stream (SISD)
- Single Instruction Stream–Multiple Data Stream (SIMD)
- Multiple Instruction Stream–Single Data Stream (MISD)
- Multiple Instruction Stream–Multiple Data Stream (MIMD)

Speicheraufbau

Zusammenhängender Speicher

Als Multiprozessoren, bzw. *Shared Memory Machines (SMMs)*, werden Rechner bezeichnet, deren CPUs einen physikalisch gemeinsamen Speicher mit zusammenhängendem Adressraum besitzen. Die darin enthaltenen Daten stehen allen Prozessoren direkt zur Verfügung und können zwischen ihnen geteilt werden. (Vgl. Rauber 2012, S.20ff)

Verteilter Speicher

Distributed Memory Machines (DMMs), auch Multicomputer genannt, arbeiten mit einem physikalisch separierten Speicher, welcher dem jeweiligen lokalen Prozessor privat zur Verfügung steht. (Vgl. Rauber 2012, S.26)

Speicheraufbau

Zusammenhängender Speicher

Als Multiprozessoren, bzw. *Shared Memory Machines (SMMs)*, werden Rechner bezeichnet, deren CPUs einen physikalisch gemeinsamen Speicher mit zusammenhängendem Adressraum besitzen. Die darin enthaltenen Daten stehen allen Prozessoren direkt zur Verfügung und können zwischen ihnen geteilt werden. (Vgl. Rauber 2012, S.20ff)

Verteilter Speicher

Distributed Memory Machines (DMMs), auch Multicomputer genannt, arbeiten mit einem physikalisch separierten Speicher, welcher dem jeweiligen lokalen Prozessor privat zur Verfügung steht. (Vgl. Rauber 2012, S.26)

Speicheraufbau

Zusammenhängender Speicher

Als Multiprozessoren, bzw. *Shared Memory Machines (SMMs)*, werden Rechner bezeichnet, deren CPUs einen physikalisch gemeinsamen Speicher mit zusammenhängendem Adressraum besitzen. Die darin enthaltenen Daten stehen allen Prozessoren direkt zur Verfügung und können zwischen ihnen geteilt werden. (Vgl. Rauber 2012, S.20ff)

Verteilter Speicher

Distributed Memory Machines (DMMs), auch Multicomputer genannt, arbeiten mit einem physikalisch separierten Speicher, welcher dem jeweiligen lokalen Prozessor privat zur Verfügung steht. (Vgl. Rauber 2012, S.26)

Programmiermodell

Shared Memory

Aus dem gemeinsamen Zugriff aller Prozessoren auf dieselben Daten erwächst ein Bedarf nach Synchronisation. Dies geschieht durch das Prinzip des gegenseitigen Ausschlusses (engl.: *Mutual Exclusion (Mutex)*). Mutex-Objekte signalisieren Threads/Prozessen ob auf eine Ressource im Hauptspeicher zugegriffen werden darf.

Message Passing

Beim Message Passing verfügt jede CPU über ihre eigene Kopie der Daten. Werden für eine Berechnung Daten eines fremden Speicherbereichs benötigt, so müssen diese explizit durch Nachrichten angefordert und anschließend übertragen werden.

Programmiermodell

Shared Memory

Aus dem gemeinsamen Zugriff aller Prozessoren auf dieselben Daten erwächst ein Bedarf nach Synchronisation. Dies geschieht durch das Prinzip des gegenseitigen Ausschlusses (engl.: *Mutual Exclusion (Mutex)*). Mutex-Objekte signalisieren Threads/Prozessen ob auf eine Ressource im Hauptspeicher zugegriffen werden darf.

Message Passing

Beim Message Passing verfügt jede CPU über ihre eigene Kopie der Daten. Werden für eine Berechnung Daten eines fremden Speicherbereichs benötigt, so müssen diese explizit durch Nachrichten angefordert und anschließend übertragen werden.

Programmiermodell

Shared Memory

Aus dem gemeinsamen Zugriff aller Prozessoren auf dieselben Daten erwächst ein Bedarf nach Synchronisation. Dies geschieht durch das Prinzip des gegenseitigen Ausschlusses (engl.: *Mutual Exclusion (Mutex)*). Mutex-Objekte signalisieren Threads/Prozessen ob auf eine Ressource im Hauptspeicher zugegriffen werden darf.

Message Passing

Beim Message Passing verfügt jede CPU über ihre eigene Kopie der Daten. Werden für eine Berechnung Daten eines fremden Speicherbereichs benötigt, so müssen diese explizit durch Nachrichten angefordert und anschließend übertragen werden.

Die Haskell Laufzeitumgebung

Bedarfsauswertung

Bedarfsauswertung (engl.: *lazy evaluation*) bezeichnet eine Evaluationsstrategie, bei der die Argumente einer Funktion erst dann ausgewertet werden, wenn sie tatsächlich benötigt werden.

Thunk

Ein Thunk ist eine unevaluierte Berechnungseinheit innerhalb eines Haskell Programms.

Die Haskell Laufzeitumgebung

Bedarfsauswertung

Bedarfsauswertung (engl.: *lazy evaluation*) bezeichnet eine Evaluationsstrategie, bei der die Argumente einer Funktion erst dann ausgewertet werden, wenn sie tatsächlich benötigt werden.

Thunk

Ein Thunk ist eine unevaluierte Berechnungseinheit innerhalb eines Haskell Programms.

Die Haskell Laufzeitumgebung

Bedarfsauswertung

Bedarfsauswertung (engl.: *lazy evaluation*) bezeichnet eine Evaluationsstrategie, bei der die Argumente einer Funktion erst dann ausgewertet werden, wenn sie tatsächlich benötigt werden.

Thunk

Ein Thunk ist eine unevaluierte Berechnungseinheit innerhalb eines Haskell Programms.

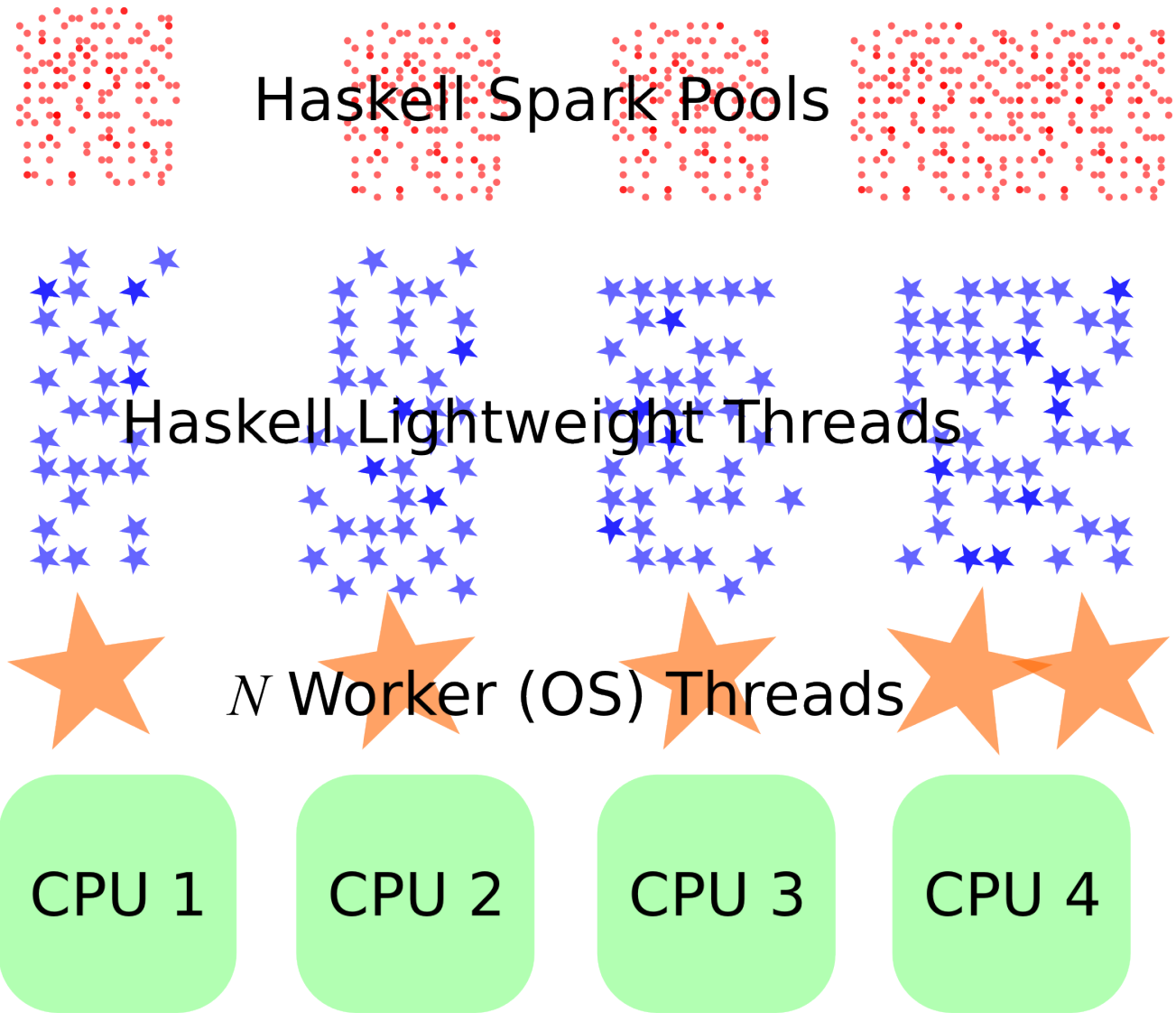


Abbildung: Don Stewart ($\rightarrow$ Stackoverflow-Link)

Parallele Kombinatoren I

Trinder et al. (1998) definiert folgende zwei Kombinatoren:

- `x ‘par‘ y` kreiert einen potentiell parallelen Thunk (=Spark) für das Argument `x` und gibt anschließend `y` zurück
- `x ‘pseq‘ y` sorgt für die Auswertung von `x` bevor `y` zurückgegeben wird

→ `x ‘par‘ y ‘pseq‘ f x y`

Parallele Kombinatoren I

Trinder et al. (1998) definiert folgende zwei Kombinatoren:

- `x ‘par‘ y` kreiert einen potentiell parallelen Thunk (=Spark) für das Argument `x` und gibt anschließend `y` zurück
- `x ‘pseq‘ y` sorgt für die Auswertung von `x` bevor `y` zurückgegeben wird

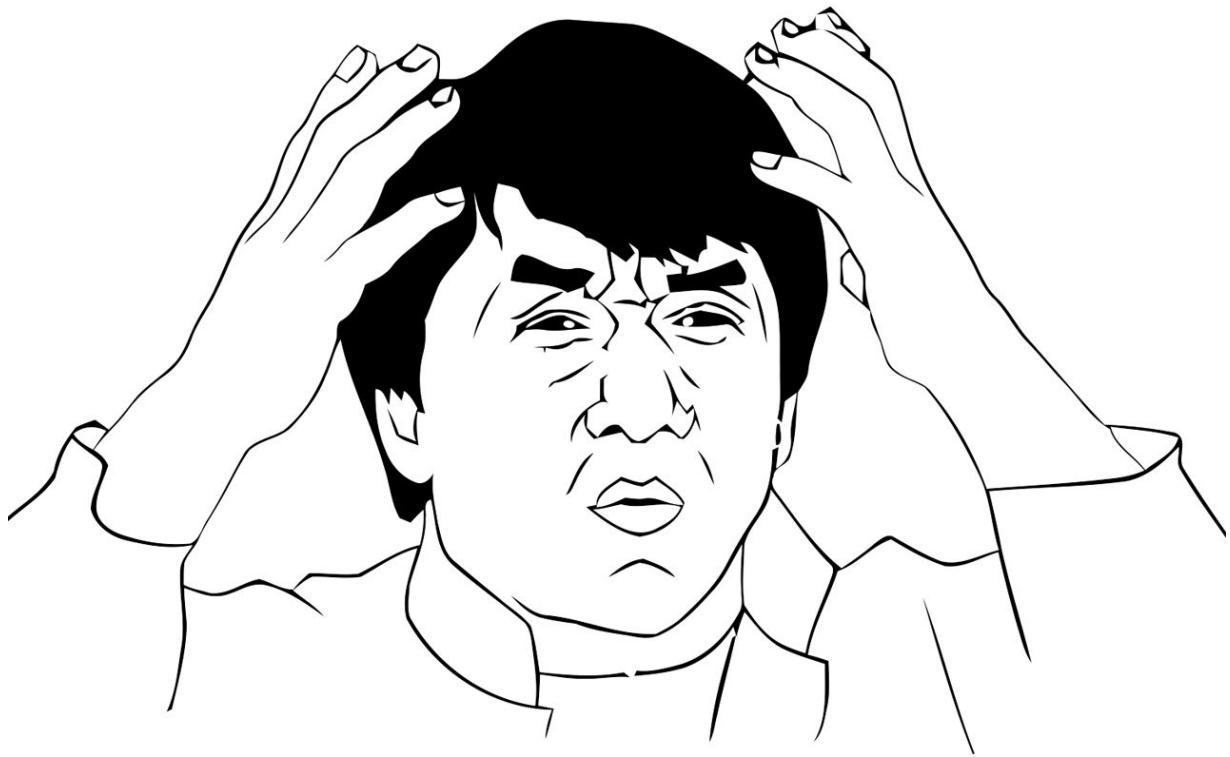
$\rightarrow$ `x ‘par‘ y ‘pseq‘ f x y`

Parallele Kombinatoren II

Marlow, Maier et al. (2010) definieren neue Formen dieser Kombinatoren:

- `r0 x` = führe keinerlei Evaluation von `x` aus
- `rpar x` = generiere einen Spark für `x`
- `rseq x` = evaluiere `x` bis zu seinem Konstruktor
- `rdeepseq x` = evaluiere `x` vollständig

Diese Funktionen geben eine Version ihres Argumentes zurück, in der die Evaluationsstrategie eingebettet ist.



Typspezifische Strategien

```
evalList :: Strategy a -> Strategy [a]
evalList s []     = return []
evalList s (x:xs) = do x' <- s x
                       xs' <- evalList s xs
                       return (x':xs')
```

Algorithmus + Strategie

```
map f [1,2,...] `using` (evalList rpar)
```

Evaluationsstrategien

+ Separation von Algorithmus und Strategie
+ Typspezifische Strategien
∼ Nur geeignet für „lazy" Datenstrukturen
– Hoher Komplexitätsgrad

Par Monade

- Explizite Erstellung paralleler Kontrollflüsse, ähnlich Kindprozessen
- Synchronisation und Datenaustausch über `IVars` ($\sim$ Futures)
- Grundlegende Funktionen:
  - `fork :: Par () -> Par ()`
  - `put :: NFData a => IVar a -> a -> Par ()`
  - `get :: IVar a -> Par a`

Par Monade

+ Explizite Kontrolle über Parallelität
∼ Vorrangig für Strukturen der Typklasse `NFData` gedacht

Manuelle Parallelisierung I

POSIX-Threads (PThreads)

- IEEE POSIX 1003.1c standard (1995)
- Implementierung des Threadmodells für POSIX-Systeme
- Prozeduren enthalten in Headerdatei `pthread.h`

std::threads

- Modernere Alternative (seit C++11) in Form einer Standardbibliothek
- Synchronisation z.B. mittels `std::mutex` Komponenten

Manuelle Parallelisierung I

POSIX-Threads (PThreads)

- IEEE POSIX 1003.1c standard (1995)
- Implementierung des Threadmodells für POSIX-Systeme
- Prozeduren enthalten in Headerdatei `pthread.h`

std::threads

- Modernere Alternative (seit C++11) in Form einer Standardbibliothek
- Synchronisation z.B. mittels `std::mutex` Komponenten

Manuelle Parallelisierung I

POSIX-Threads (PThreads)

- IEEE POSIX 1003.1c standard (1995)
- Implementierung des Threadmodells für POSIX-Systeme
- Prozeduren enthalten in Headerdatei `pthread.h`

std::threads

- Modernere Alternative (seit C++11) in Form einer Standardbibliothek
- Synchronisation z.B. mittels `std::mutex` Komponenten

Manuelle Parallelisierung II

\+ Größtmögliche Flexibilität

– **Aufwending**

Parallele APIs I

Open Multiprocessing (OpenMP)

- Seit 1997 in Zusammenarbeit mehrerer namhafter Hardware- und Compilerhersteller entwickelt
- „[M]it dem Ziel [. . .], einen einheitlichen Standard für die Programmierung von Parallelrechnern mit gemeinsamen Adressraum zur Verfügung zu stellen" (Rauber 2012, S.357)
- Pragma-Anweisungen, die für andere Compiler aussehen, wie Kommentare
- Vorrangig für Loop-Level Parallelisierung eingesetzt

Parallel for

```
#pragma omp parallel for
for (i = 0; i < 10; ++i) {
    /* ... */
}
```

Parallele APIs II

+ Einfach und intuitiv
– Weniger flexibel (Loop-Level, parallele Regionen)

- Auch interessant:
  - Intel Threading Building Blocks (TBB)
  - Open Accelerator (OpenACC)
  - Open Message Passing Interface (OpenMPI)

Parallele Standardbibliotheken

- Parallelität in vielen Funktionalen ist offensichtlich!
- Singler, Sanders und Putze (2007, S.1) bezeichnen sie als „Embarrassingly parallel", entwickelten parallele Versionen einiger Standardfunktionen
- Inklusive dynamischer Lastverteilung
- Basierend auf OpenMP
- Namespace `std::__parallel`

```
# include <parallel/algorithm>
# include <parallel/numeric>
namespace par = std::__parallel;

par::transform( ... ); /* Map    */
par::accumulate( ... ); /* Reduce */
```


THAT WAS EASY
SUSPICIOUSLY EASY

Antwortzeit

Definition

Die Antwortzeit T_A (engl.: *wallclock time*) eines Programms ist definiert als die Differenz zwischen Start seiner Ausführung und Beendigung seines letzten Prozesses, wie gemessen von einer herkömmlichen Wanduhr.

Durchschnittlich genutzter Arbeitsspeicher

Definition

Die *Resident Set Size (RSS)* entspricht dem belegten Bereich im Hauptspeicher der Maschine zu einem gegebenen Zeitunkt. Eventuell auf die Swap-Partition ausgelagerte Teile des Programms fließen nicht in den Wert ein.

Durchsatz

Definition

Der Durchsatz $D(N)$ ist in vorliegendem Szenario definiert als die Anzahl verarbeiteter Datensätze N pro Sekunde:

$$D(N) = \frac{N}{T_A}[\text{Records/s}]$$

Cache Hit-Rate

Definition

Der prozentuale Anteil der Read-Hits von der Gesamtzahl aller lesenden Cache-Zugriffe wird als Hit-Rate bezeichnet:

$$\text{Hit-Rate} = \frac{\text{Read-Hits}}{\text{Read-Hits} + \text{Read-Misses}}[\%]$$

Speedup

Definition

Der Speedup $S(P)$ beschreibt, wie sehr ein paralleles Programm von einer steigenden Zahl zur Verfügung stehender CPUs profitiert:

$$S_{rel}(P) = \frac{T_A(1)}{T_A(P)}$$

Effizienz

Definition

Die Effizienz eines parallelen Programms ist definiert als das Verhältnis aus Speedup $S(P)$ und der Zahl Prozessoren P (vgl. Rauber 2012, S.179):

$$E(P) = \frac{S(P)}{P}$$

Kosten

Definition

Die Kosten $C(P)$ eines parallelen Programms entsprechen der Arbeit, die von allen eingesetzten Prozessoren zur Ausführung des Algorithmus aufgebracht wurde. Sie ist das Produkt aus Antwortzeit T_A und der Zahl eingesetzter Prozessoren P (vgl. Rauber 2012, S.176):

$$C(P) = T_A \cdot P$$

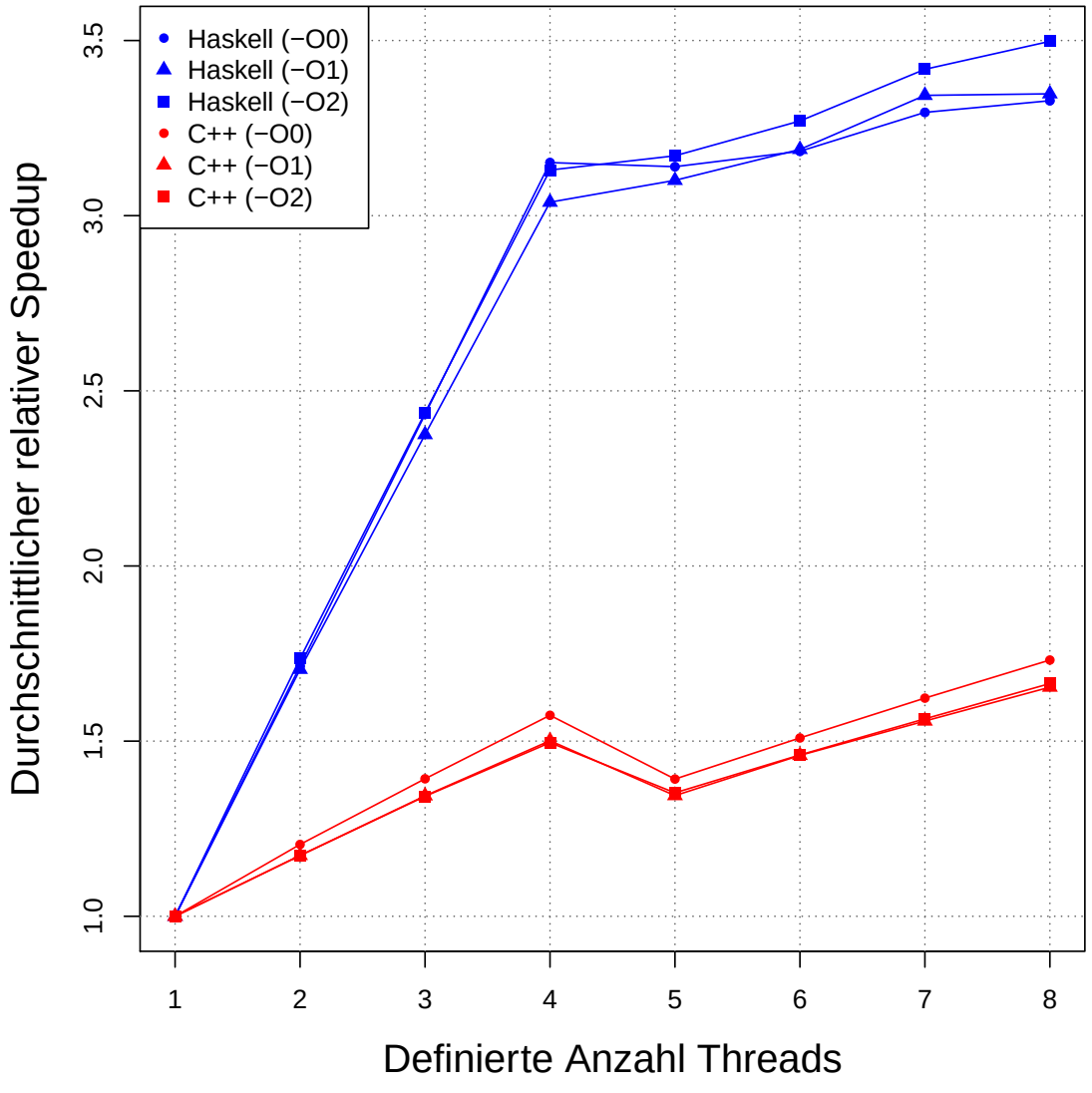
Paralleler Overhead

Definition

Der Parallele Overhead $O(P)$ eines Programms beschreibt den Mehraufwand, welcher aus der Erstellung, Koordination und Beendigung Thread- basierter Parallelität resultiert:

$$O(P) = \frac{C(P) - T_A(1)}{C(P)} \cdot 100[\%]$$

Speedup I


Haskell (−O0)
Haskell (−O1)
Haskell (−O2)
C++ (−O0)
C++ (−O1)
C++ (−O2)
Durchschnittlicher relativer Speedup
1.0
1.5
2.0
2.5
3.0
3.5
1
2
3
4
5
6
7
8
Definierte Anzahl Threads

Speedup II

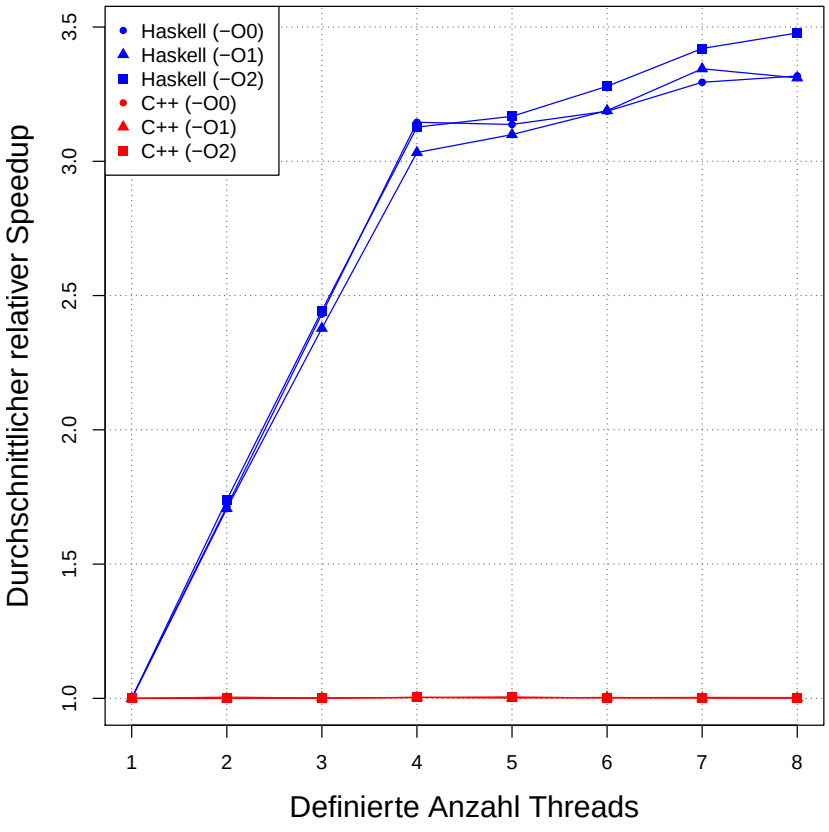


Abbildung: $<$ 1000 Dateien

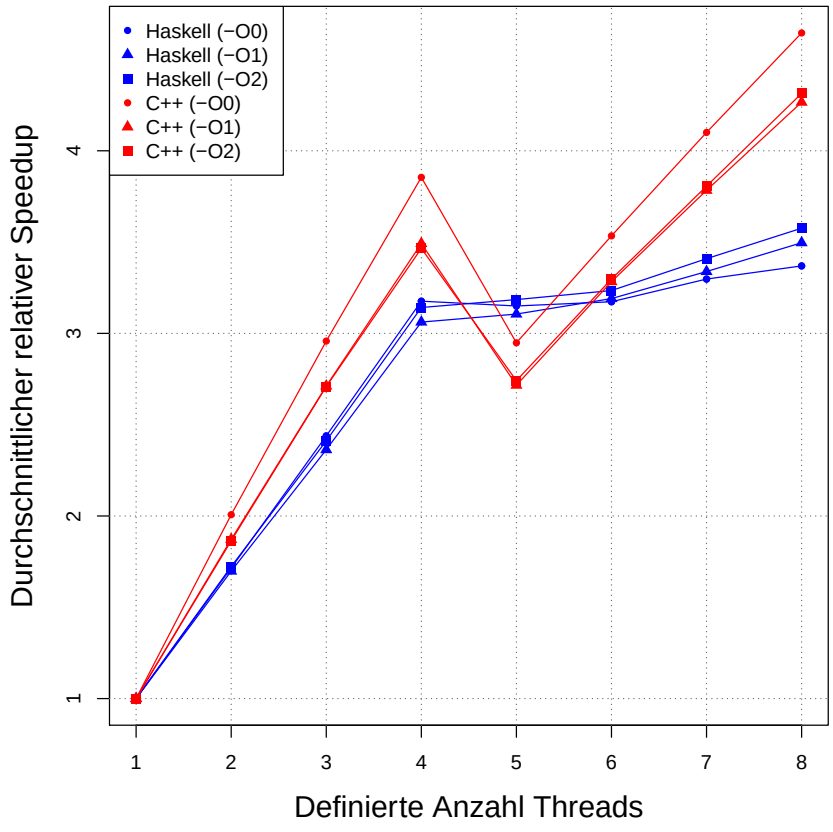


Abbildung: 1000 Dateien

CPU Auslastung

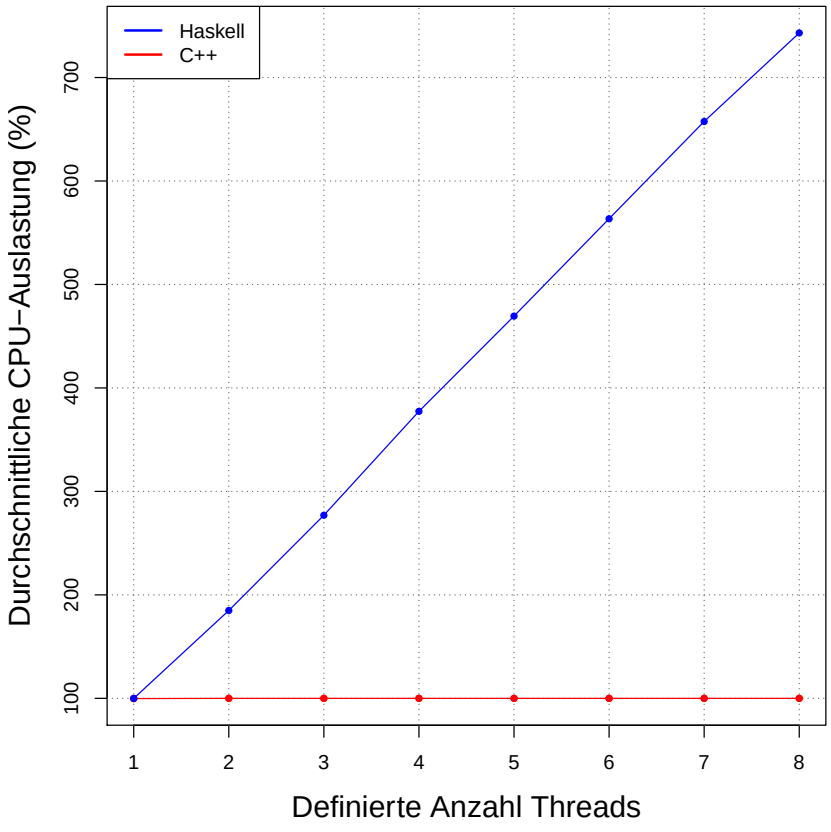


Abbildung: $<$ 1000 Dateien

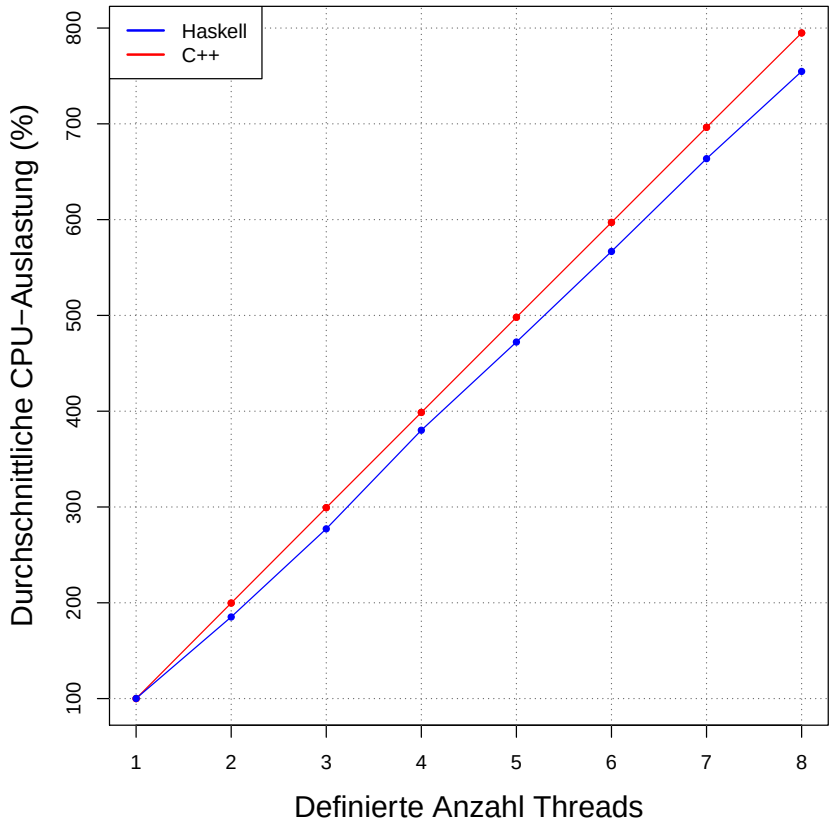


Abbildung: 1000 Dateien

Effizienz

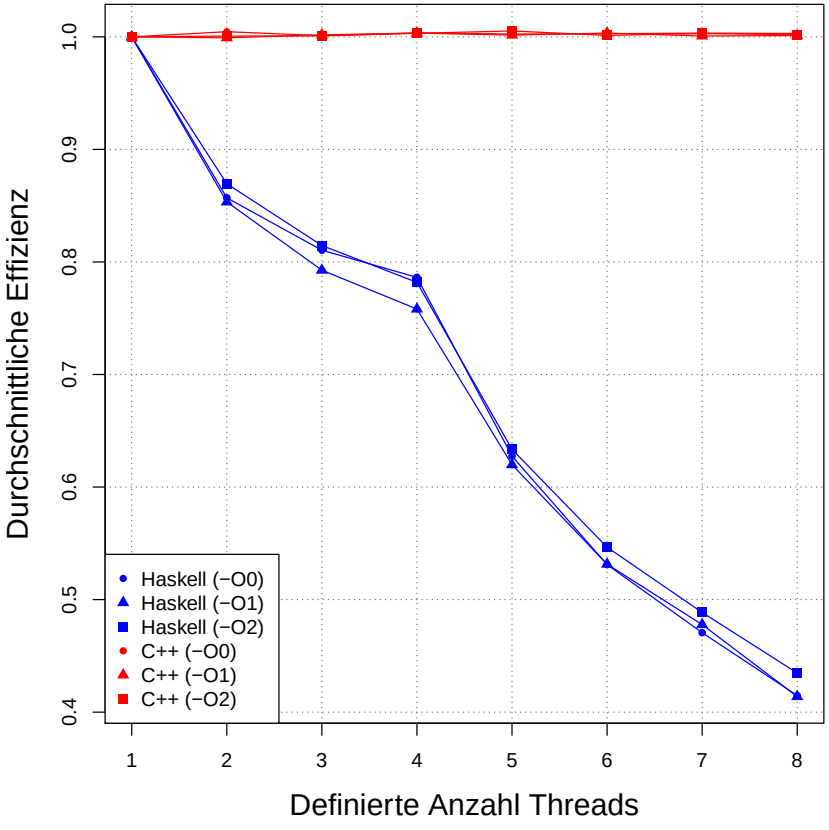


Abbildung: $<$ 1000 Dateien

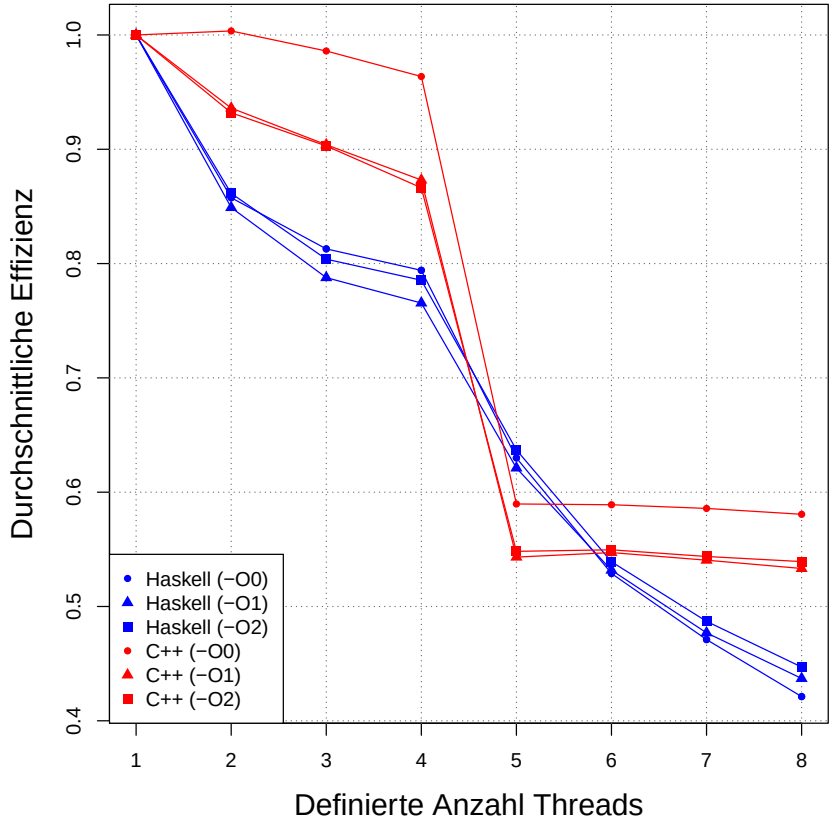


Abbildung: 1000 Dateien

Paralleler Overhead

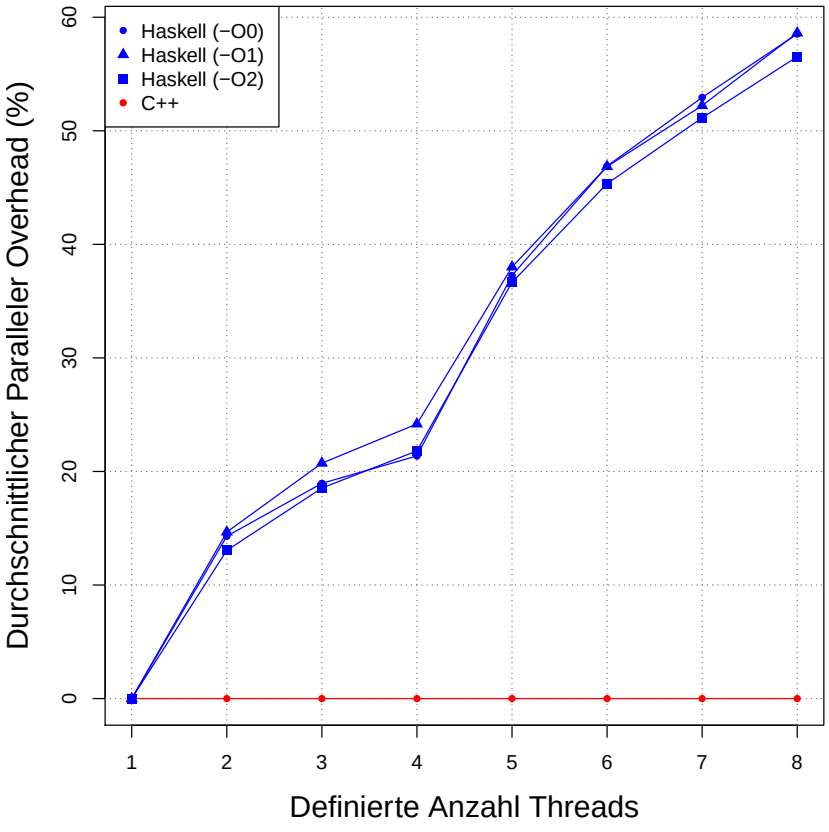


Abbildung: $<$ 1000 Dateien

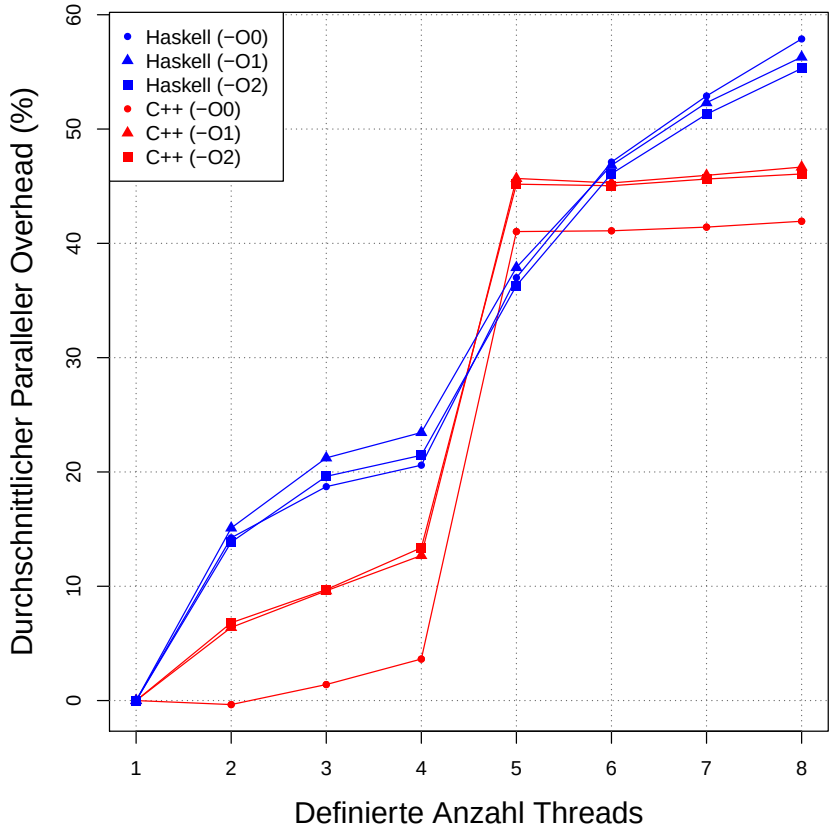


Abbildung: 1000 Dateien

Kosten

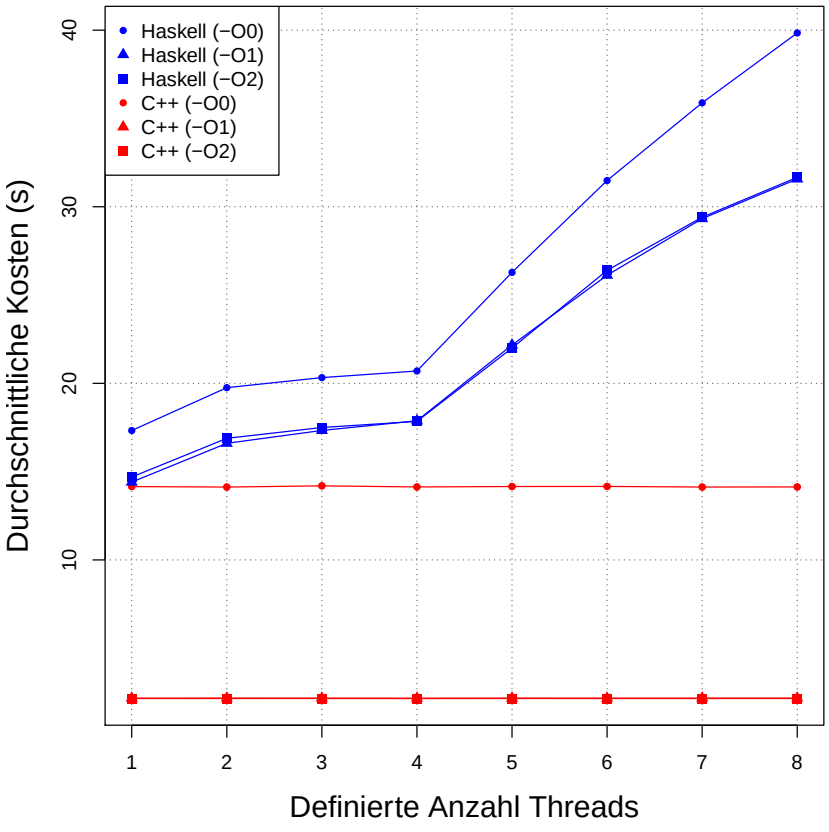


Abbildung: $<$ 1000 Dateien

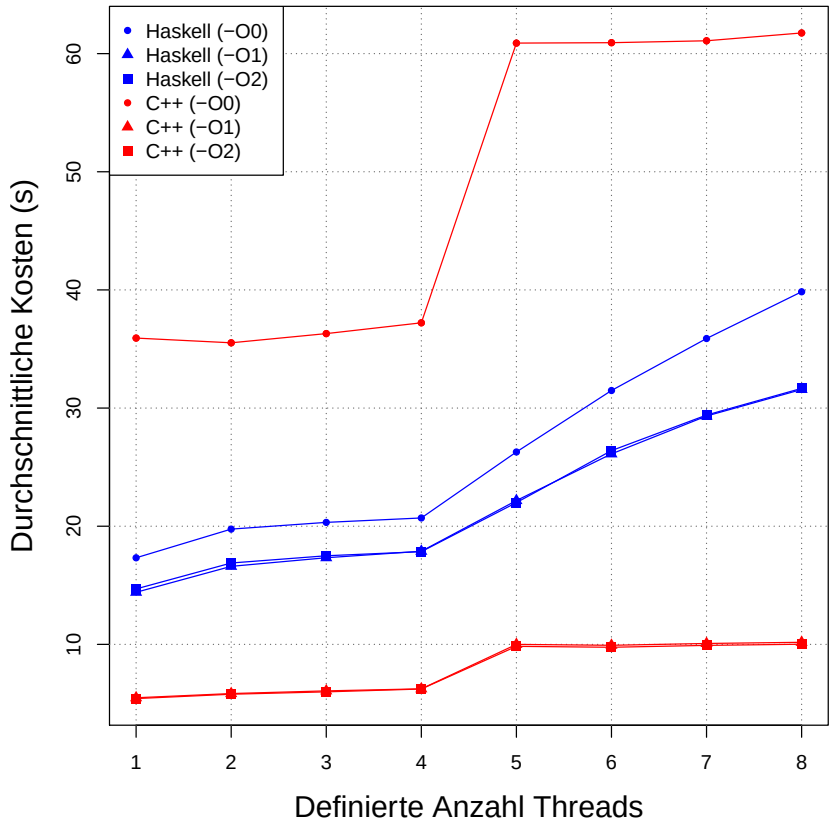
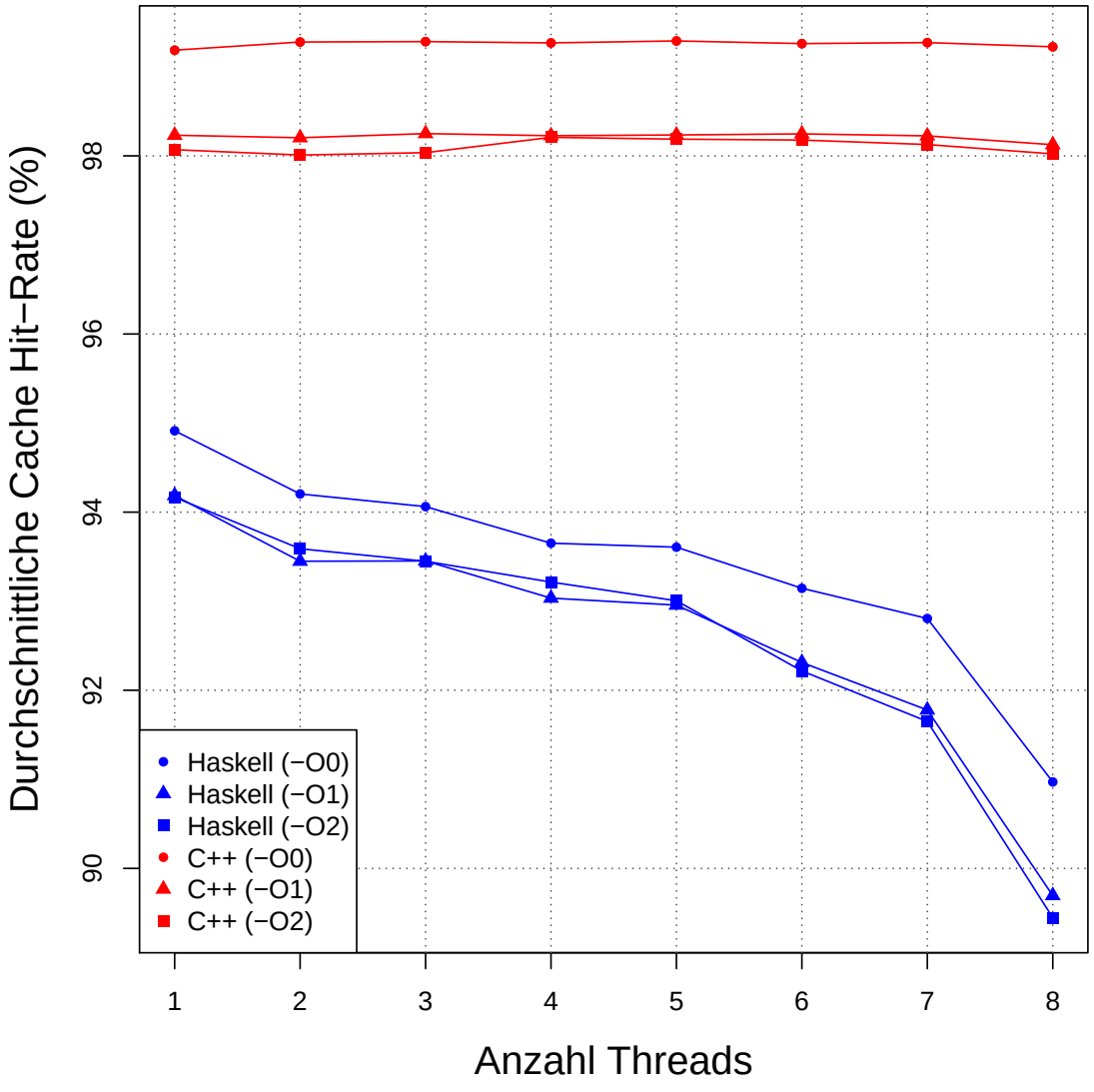


Abbildung: 1000 Dateien

Cache Hit-Rate



Durchsatz/Dateien

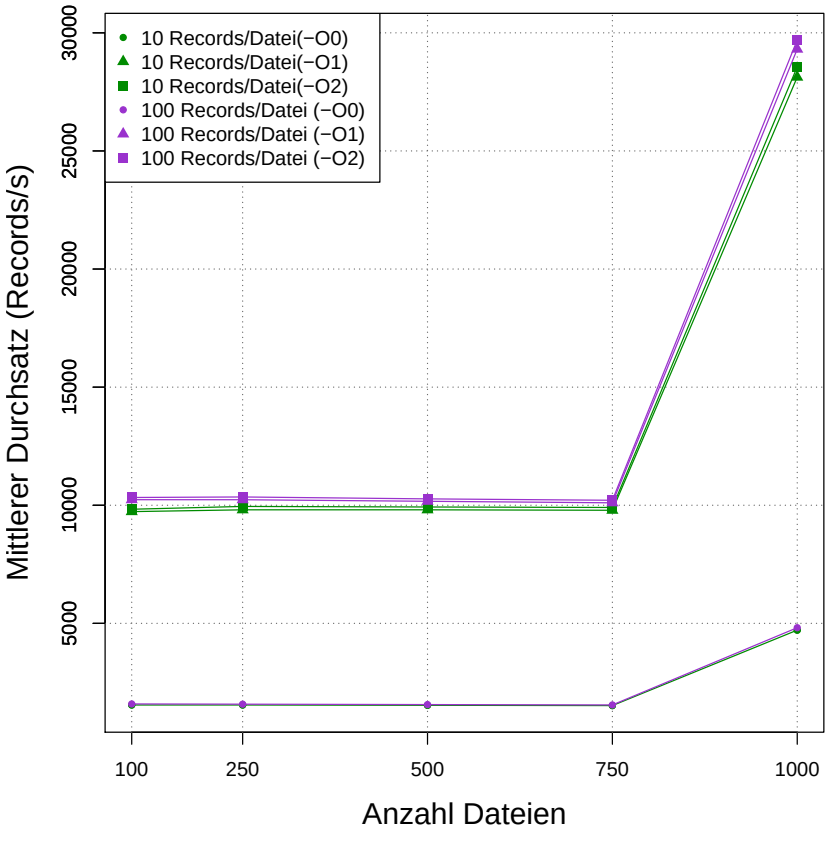


Abbildung: C++

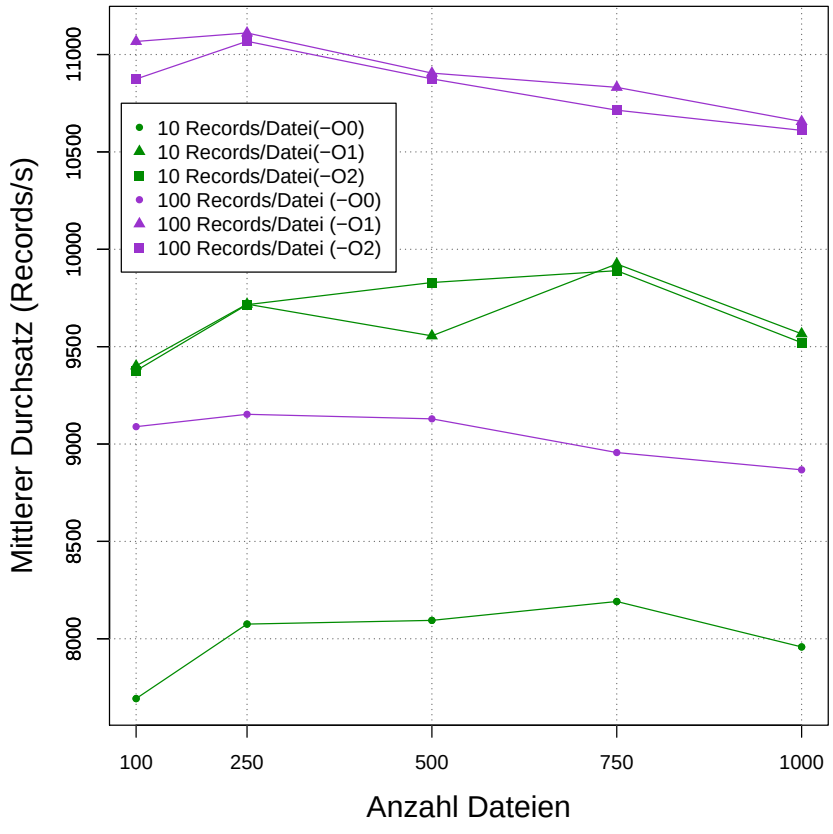


Abbildung: Haskell

Durchsatz/Threads

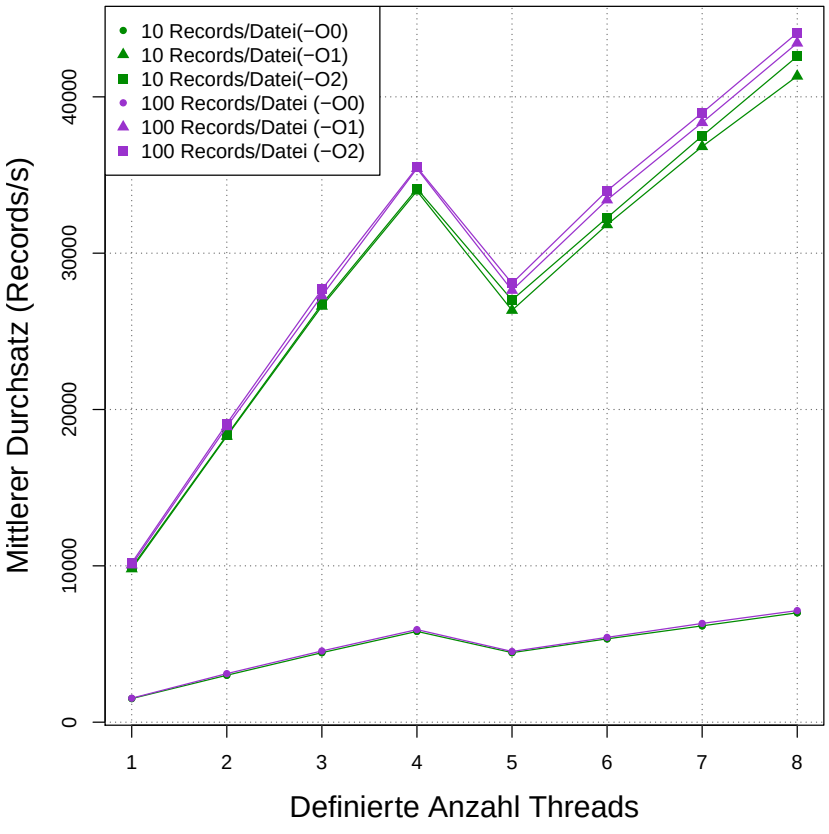


Abbildung: C++

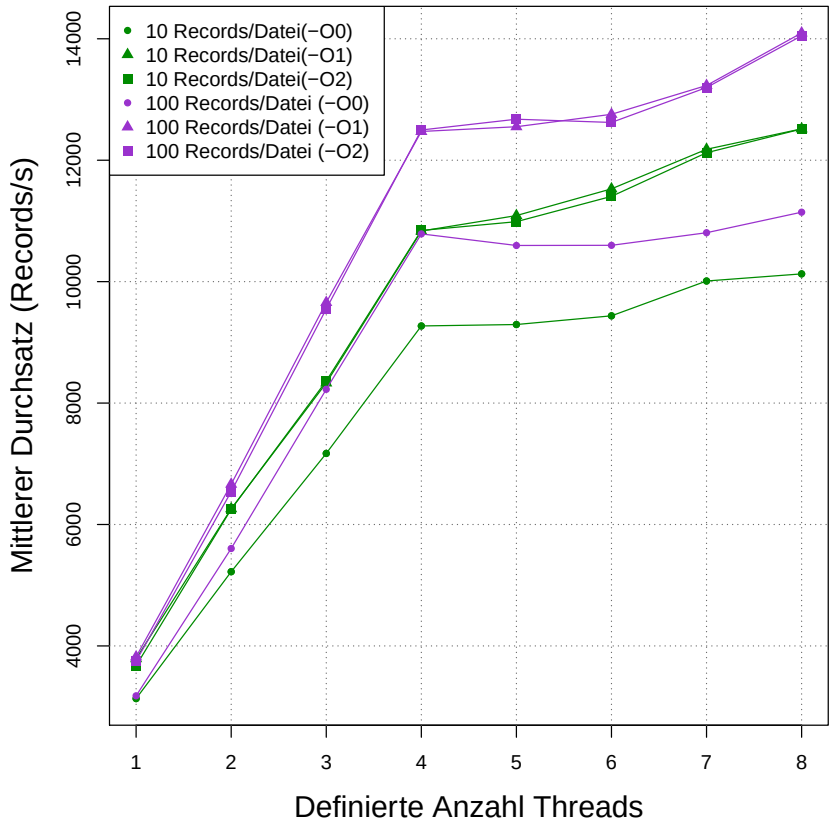
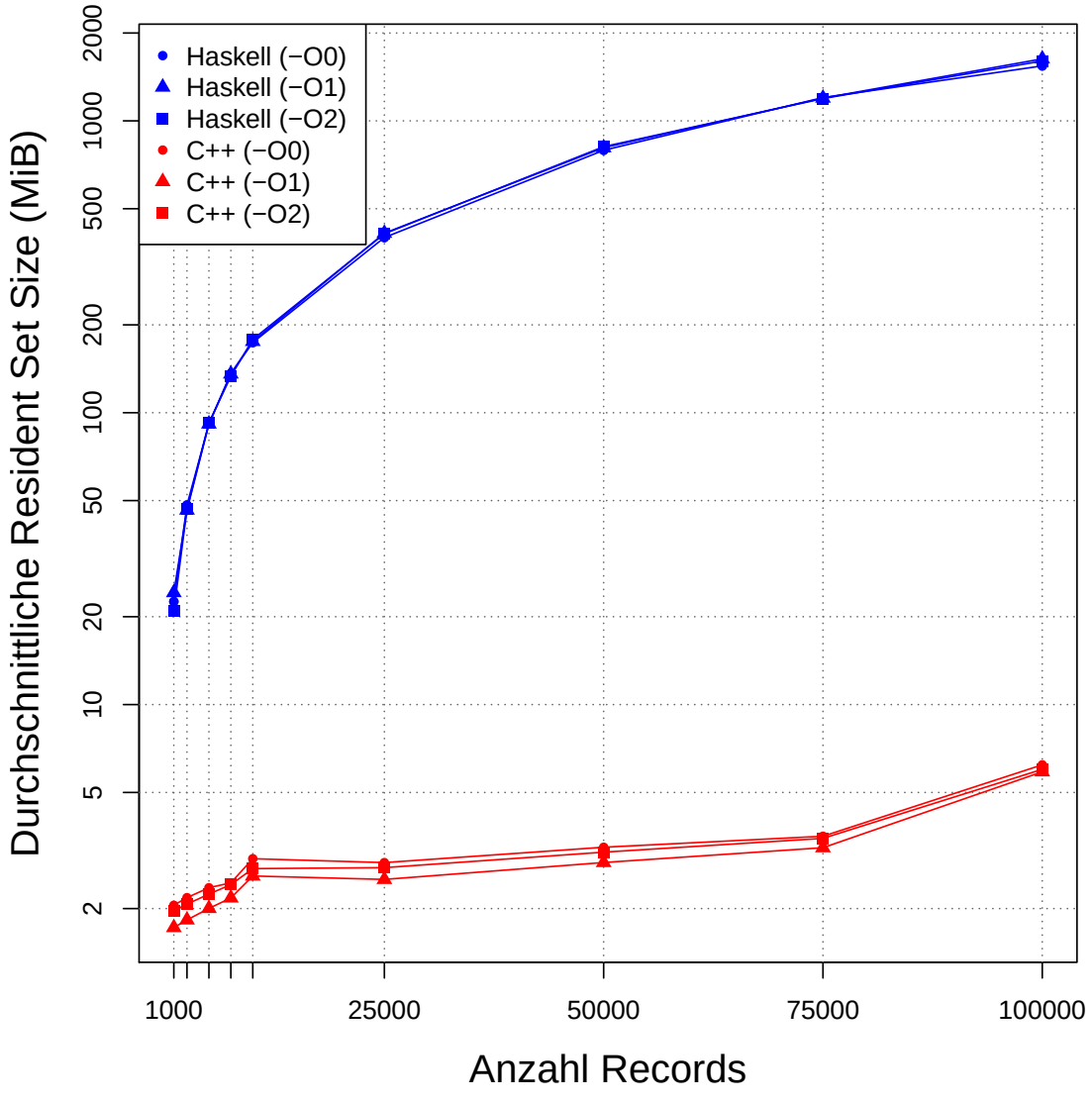


Abbildung: Haskell

Durchschnittliche RSS



Vielen Dank