Automatic Parallelization: Executing Sequential Programs on a Task-Based Parallel Runtime

SFI-2223

Paul Bartel

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

6. Dezember 2022

Agenda

Motivation

- Legacy Code profitiert nicht von modernen Architekturen
- Umschreiben und Instandhaltung ist aufwendig
- Kostet viel Zeit, Nerven und Geld

Ansatz

- Sequentiellen Code mit S2S Compiler in Task aufteilen
- Tasks in einer dedizierten Laufzeitumgebung parallel ausführen
- Managment aller Tasks während Laufzeit

Task-Based Parallelismus

- Programm wird mit Tasks umgesetzt, welche in der Laufzeitumgebung ausgeführt werden.
- Ein Tasks ist definiert durch seinen
  - Ausführbaren Code(beliebiger Größe)
  - Input/Output
  - Abhängigkeiten zu anderen Tasks
- Wenn keine Abhängigkeiten zueinander bestehen können
  - Tasks in beliebiger Reihenfolge ausgeführt werden
  - Tasks parallel ausgeführt werden

Überblick Architektur

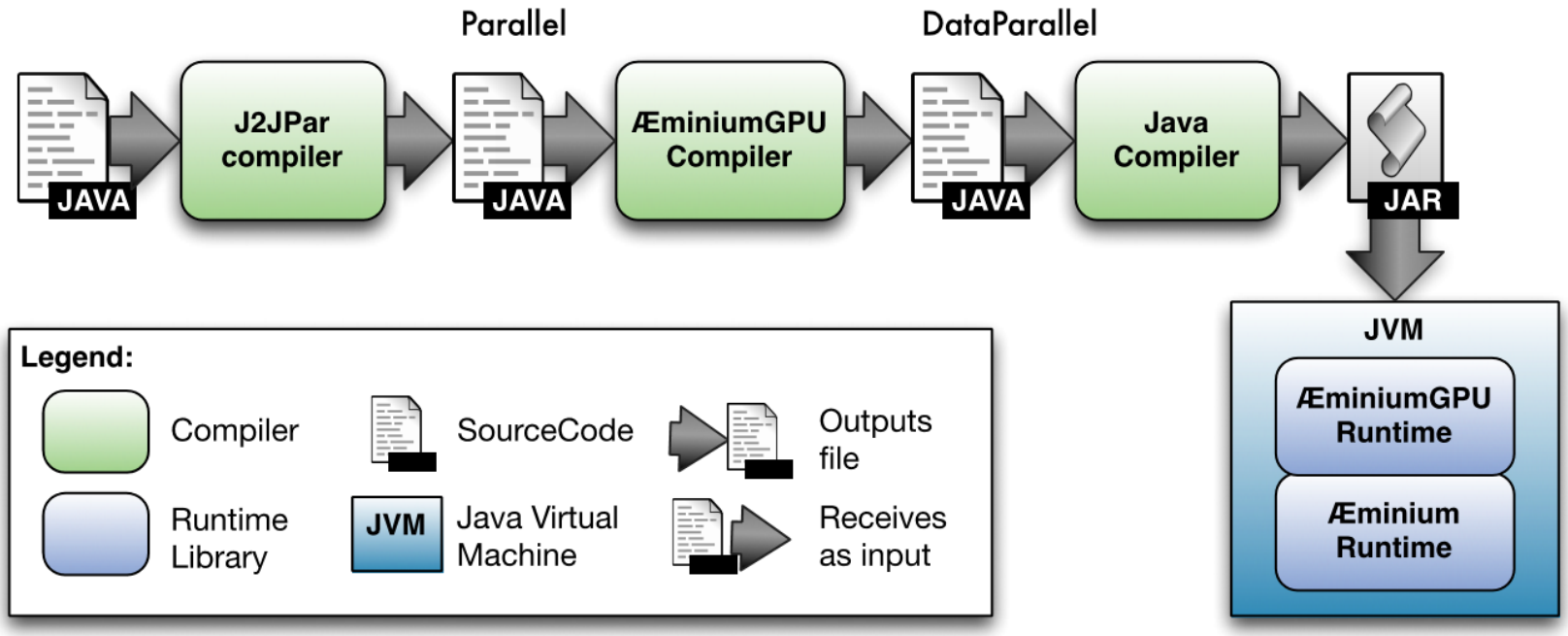

Abbildung: Grundlegender Aufbau der Architektur [Alc16]

Phasen der Kompilierung

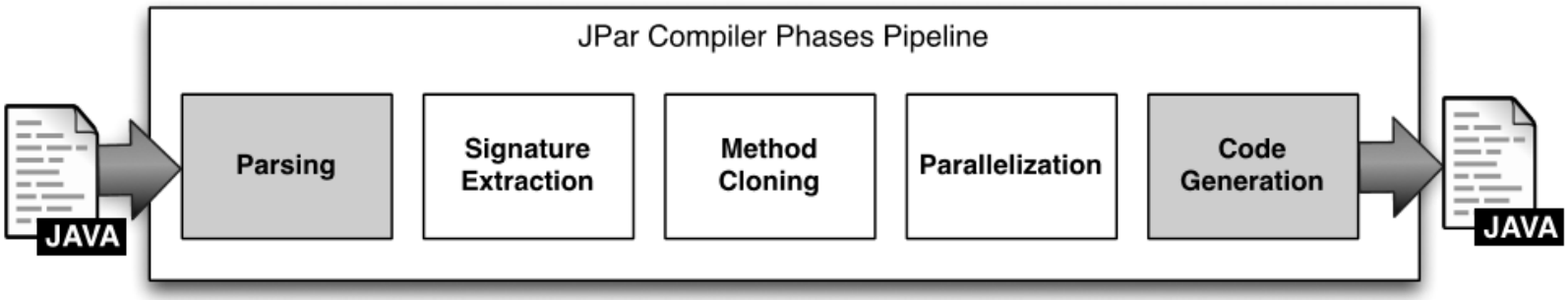

Abbildung: JPar Compiler Phasen [Alc16]

Abstract Syntax Tree

- Ordnet Source Code in eine Baum Struktur an
- Jeder Knoten stellt ein Sprach Konstrukt dar

```
1  int f(int a, int b){
2  while b != 0:
3  if a > b:
4      a := a - b
5  else:
6      b := b - a
7  return a
8  }
```

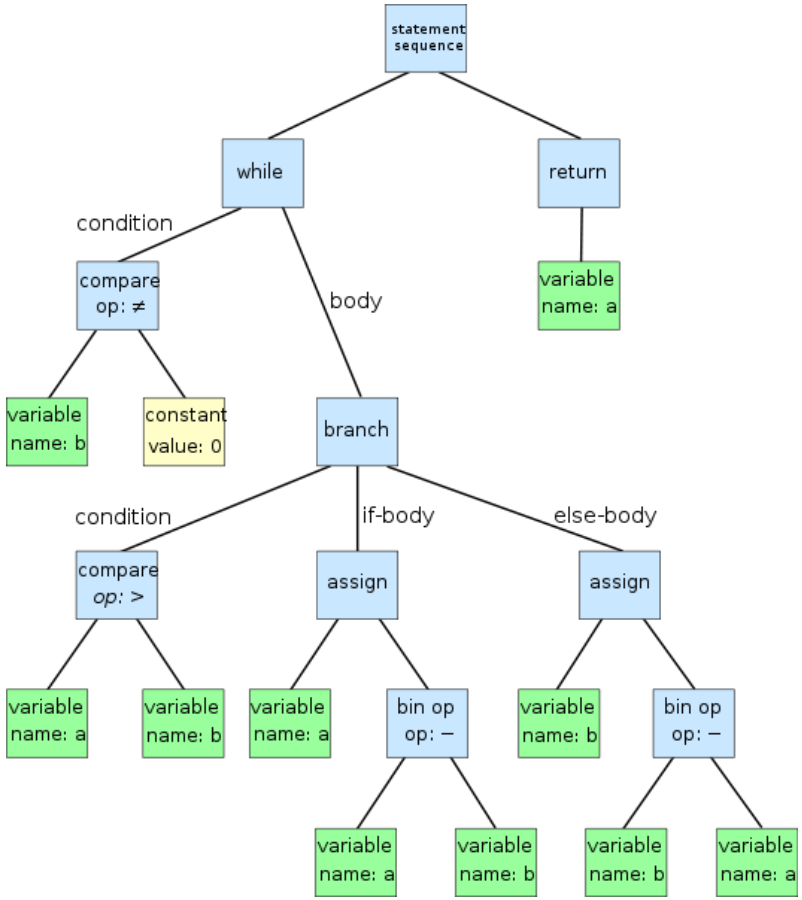

Abbildung: Abstract Syntax Tree[Wik22]

Signature Extraction

- Compiler muss Abhängigkeiten in AST finden und entsprechend gruppieren
- Wenn zwei Knoten zur gleichen Gruppe gehören, können sie nicht parallelisiert werden
- Mögliche Signaturen der Knoten sind
  - read()
  - write()
  - control()

```
1  int f(int n) {
2      if (n < 2) { // read(n), control(f)
3      return n; // write(return), control(f)
4      }
5      int a = f(n - 1); // call(f), read(n), write(a)
6      int b = f(n - 2); // call(f), read(n), write(b)
7      return a + b; // read(a), read(b), control(f), write(return)
```

Method Cloning

- Verfügbarkeit gleicher Methode in paralleler und sequentieller Form
- Overhead von Task Erstellung kann länger dauern als sequentielle Ausführung
- Laufzeitumgebung entscheidet wann welche Version ausgeführt wird

Parallelization

- Theoretisch möglich sehr fein zu parallelisieren, praktisch unsinnig
- Compiler kann Tasks für Methoden Aufrufe und Schleifen erzeugen
- Damit der Compiler eine Methode parallelisiert muss diese
  - mindestens 10 Instruktionen haben
  - Schleifen beinhalten
  - andere rechenintensiven Methoden aufrufen
  - oder rekursiv sein

Parallelization: Methoden Aufrufe I

```
1  int f(int n) {
2      if (RuntimeManager.shouldSeq())
3          return jpar_sequential_version_of_f(n);
4      if (n < 2) {
5          return n;
6      }
7      Future<Integer> b_tmp = new Future<Integer>(task -> f(n-2));
8      Future<Integer> a_tmp = new Future<Integer>(task -> f(n-1));
9      int a = a_tmp.get();
10     int b = b_tmp.get();
11     return a + b;
12 }
```

- Future erzeugt = Task zur Bearbeitung bereit
- Platzierung von Future wichtig um maximalen Parallelismus und richtige Semantik zu gewährleisten

Parallelization: Methoden Aufrufe II

- Zur Platzierung müssen bestimmte Anforderungen erfüllt sein
  - Muss nach allen benutzen Variablen initialisiert werden
  - Muss nach einem Ausdruck initialisiert werden, welcher eventuell in die Methode returned
  - Muss nach einem Ausdruck initialisiert werden, welcher den Kontrollfluss ändert
  - Muss nach einem Ausdruck ausgeführt werden, welcher eventuell zu einer Variablen im Task schreibt
  - Muss nach einem Ausdruck ausgeführt werden, welcher eventuell aus einer Variable im Task liest
  - Muss vor get() initialisiert werden
- Hard Dependency: Punkt ab dem der Future initialisiert werden darf.
- Soft Dependencies: Eine Menge an Tasks auf welche der Future warten muss bevor er starten kann

Parallelization: Schleifen DO-ALL

- DO-ALL: Jede Iteration ist unabhängig

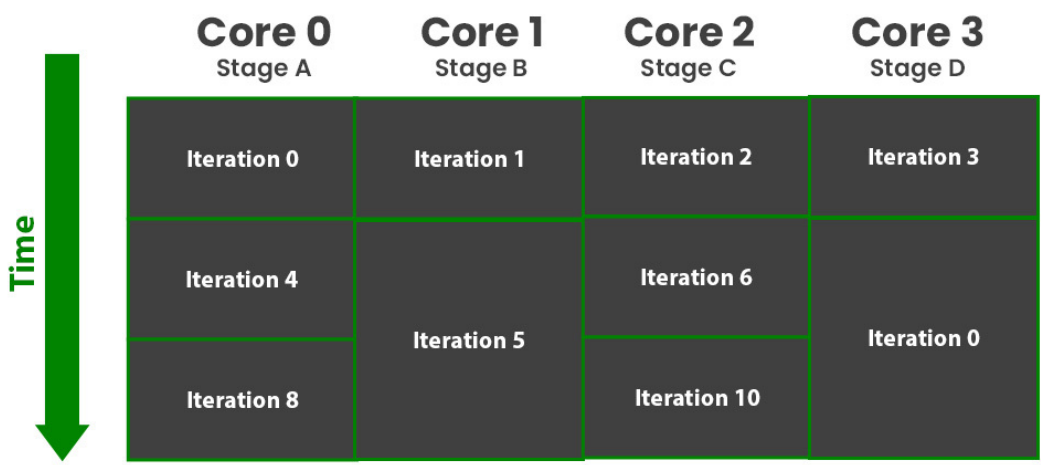

Abbildung: [Gee22]

- Aufruf einer statischen Methode: Erstellt dynamisch Tasks für Teile der Schleife, jede Iteration erhält write(Array[i])
- Return eines Futures (Task Erstellung)

Parallelization: Schleifen DO-ACROSS

- DO-ACROSS: Iteration+1 ist von der letzten Iteration abhängig

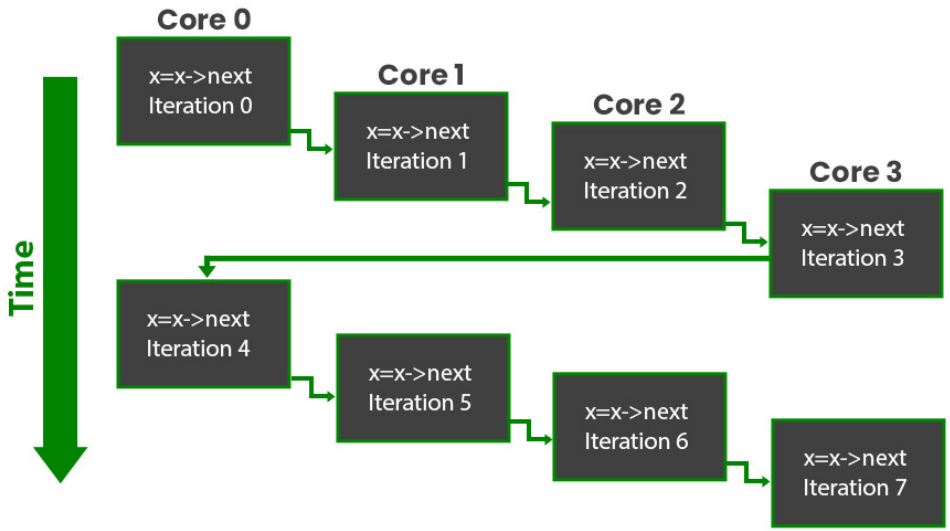

Abbildung: [Gee22]

- Genau wie DO-ALL, aber einige write permissions sind erlaubt(Kommutative und Assoziative)

Tasks und Abhängigkeiten I

- Laufzeitumgebung im Fall von Java Æminium
- API um asynchrone Ausführung von Code zu ermöglichen
- Asynchrone Ausführung in Form von Tasks implementiert
- Es laufen immer eine feste Anzahl an Threads
- Besteht aus:
  - Scheduler scheduled die Tasks
  - Decider entscheidet ob ein neuer Task erstellt wird
  - Profiler zeichnet informationen während Ausführung aus

Tasks und Abhängigkeiten II

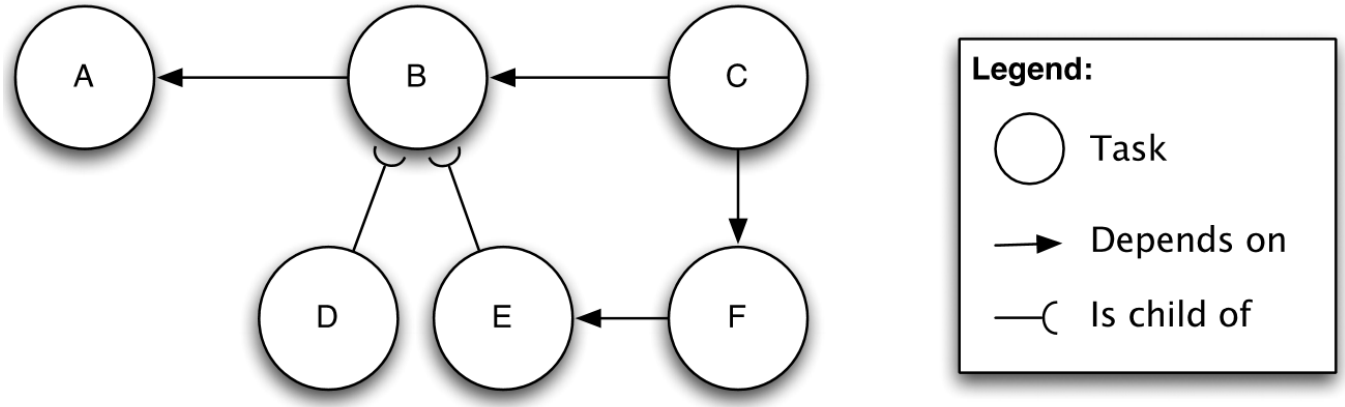

Abbildung: Beispiel für einen Taskgraph[Alc16]

- Typen von Tasks
  - Non-Blocking-Task: reine rechnerischer Task
  - Blocking Task: mindestens eine In/Out Operation
  - Atomic Task: können nicht gleichzeitig ausführt werden, wenn sie in der gleichen Gruppe sind

Laufzeit Umgebung

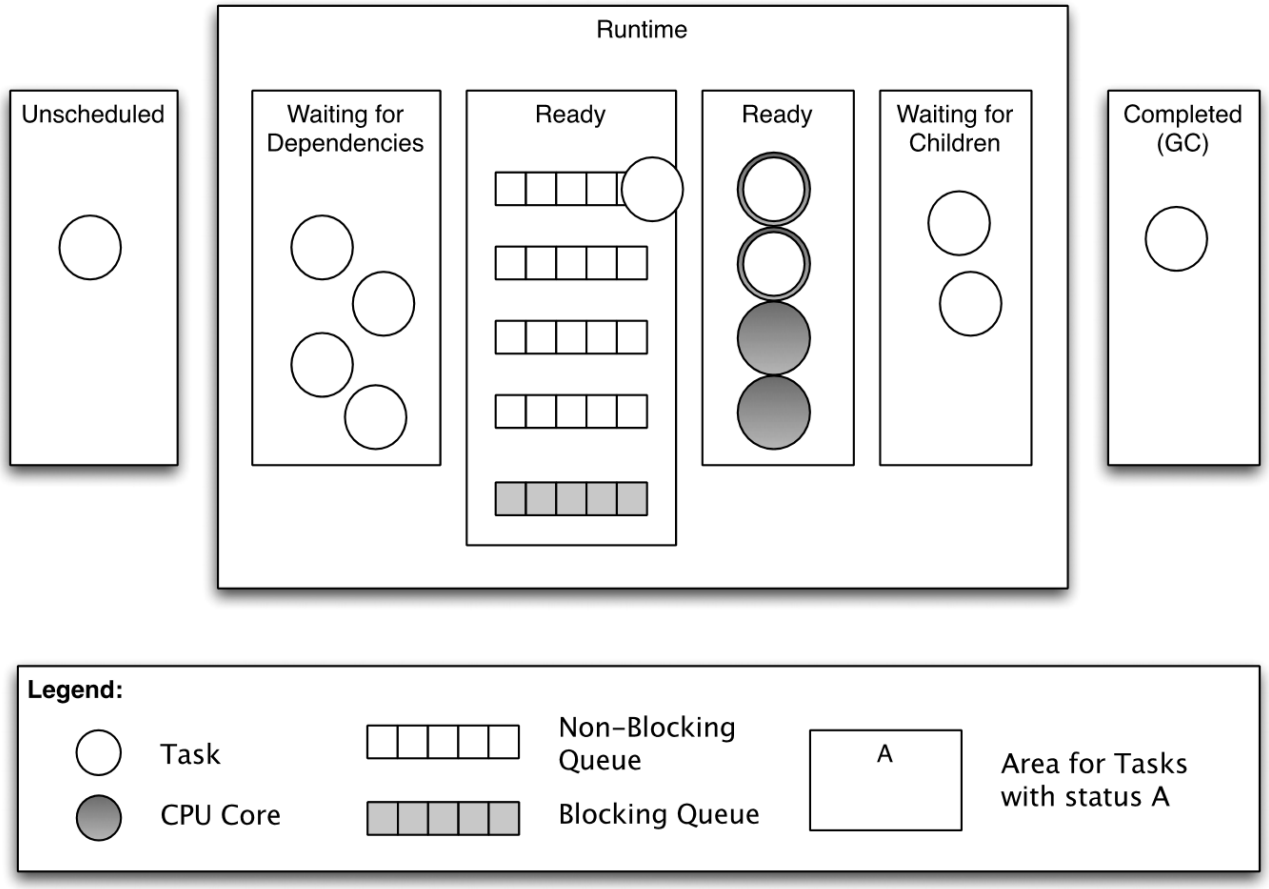

Abbildung: Laufzeit Umgebung für ein 4 Core System[Alc16]

Ausführung von Schleifen

- Schleifen brauchen teilweise je nach Iteration unterschiedliche Zeit
- Dynamischer Ansatz Schleifen aufzuteilen.
- Binary Splitting: Wenn Decider entscheidet neuer Task ist Sinnvoll, dann wird die Range durch 2 geteilt
- Lazy Binary Splitting: Prüft nach einer bestimmten Anzahl ob die Range geteilt werden soll (Parameter setzbar)

Cutoff-Mechanismen I

- Decider entscheidet ob neuer Task sinnvoll ist und kontrolliert so die Feinheit
- Unnötiger Overhead gilt es zu verhindern
- Gibt 8 Mechanismen, um diese Feinheit zu Kontrollieren
- lassen sich mit tradeoff kombinieren

Cutoff-Mechanismen II

- **Maximum Task Recursion Level**
  Erstellung neuer Tasks durch Tiefe der Rekursion definiert, mehr geeignet für balancierte Programme
- **Maximum Number of Tasks**
  Erstellung neuer Tasks durch eine Maximale Zahl an Tasks definiert
- **Load Based**
  Erstellung neuer Tasks durch Idle-Cores definiert, mindestens ein Core im Idle

Cutoff-Mechanismen III

- **Surplus Queued Task Count**
  Erstellung neuer Tasks durch Verteilung der Tasks auf Threads definiert
- **Adaptive Tasks Cut-O**
  Neue Tasks werden erstellt wenn
  - weniger Tasks als Threads auf der gegebenen Rekursions Tiefe
  - Rekursions Tiefe geringer als gegebener Wert
- **Maximum Queue Size**
  Erstellung neuer Tasks durch die lokale Anzahl an Tasks in einer Queue definiert

Cutoff-Mechanismen IV

- **Stack Size**
  Erstellung neuer Tasks durch die Stack Größe definiert
- **System Monitoring**
  Erstellung neuer Tasks durch Last des Systems definiert

Benchmark Programme

Program	Parallelism	Input size
Black-Scholes	DO-ACROSS	100000
FFT	Recursive	16777216
Fibonacci	Recursive	n=51
Health	Recursive, DO-ALL	n=6
Integrate	Recursive	s=-2101, e=1700, error=10^{-14}
MergeSort	Recursive	n=251658240
N-Body	DO-ALL	it=10, bodies=25000
Pi	DO-ACROSS	n=1500000000

Abbildung: Beschreibung der Programme welche zur Evaluation benutzt wurden[Alc16]

Compiler Task Aufteilung

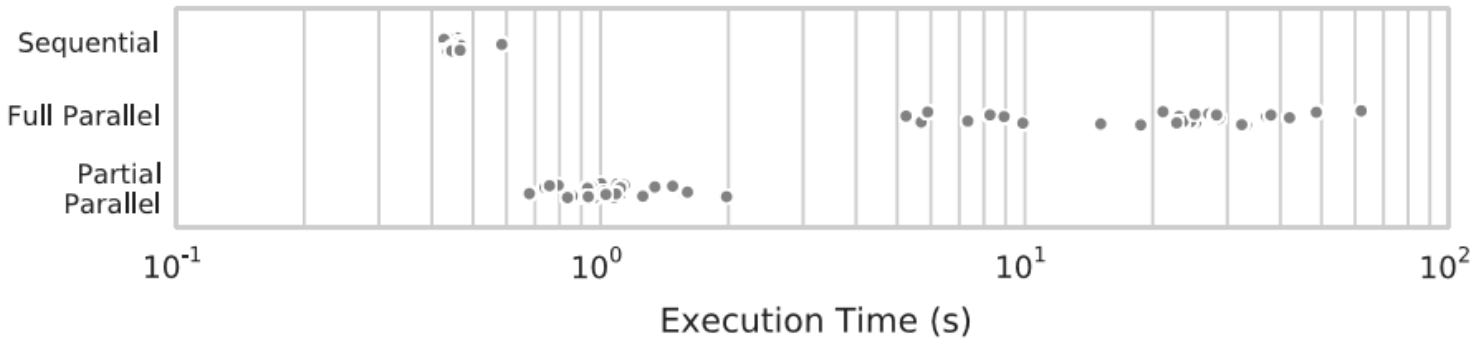

Abbildung: Verteilung der Ausführungszeit von FFT mit zufälligem Array der Größe 262144 [Alc16]

- Kein Speedup bei der Inputgröße
- Mit größeren Input
  - Fully Parallel: Extrem langsam
  - Partial Parallel: Gibt Speedups

Binary Splitting vs Lazy Binary Splitting

- Binary Splitting default die beste Option
- Lazy Binary Splitting performanter mit passenden Parametern

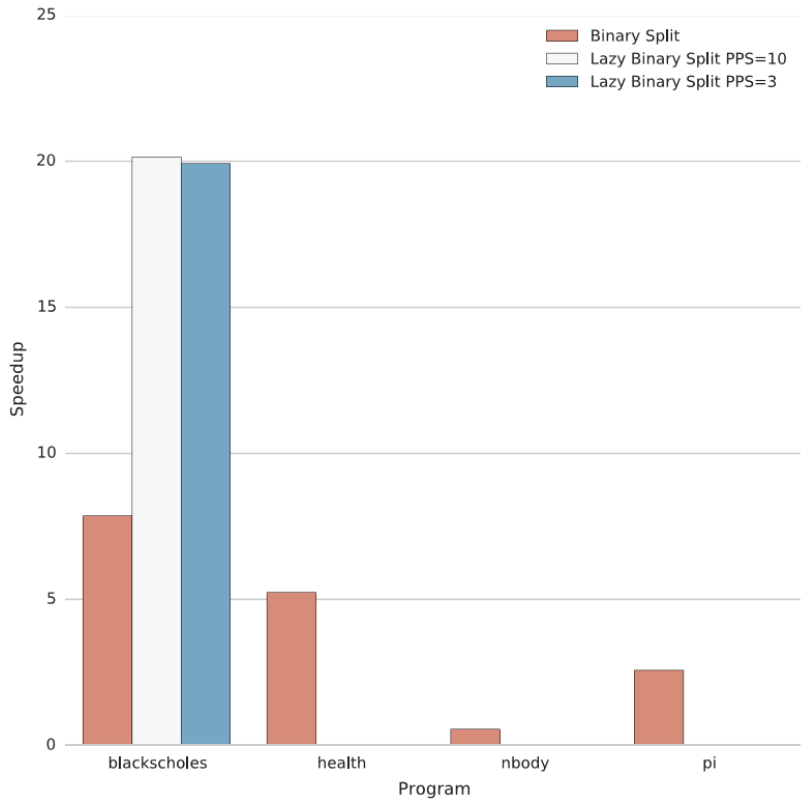

Abbildung: Vergleich LBS BS [Alc16]

Cutoff Mechanismen

- Speedup der Cuto Mechanimen stark abhängig von der Ressourcen Nutzung des jeweiligen Programm

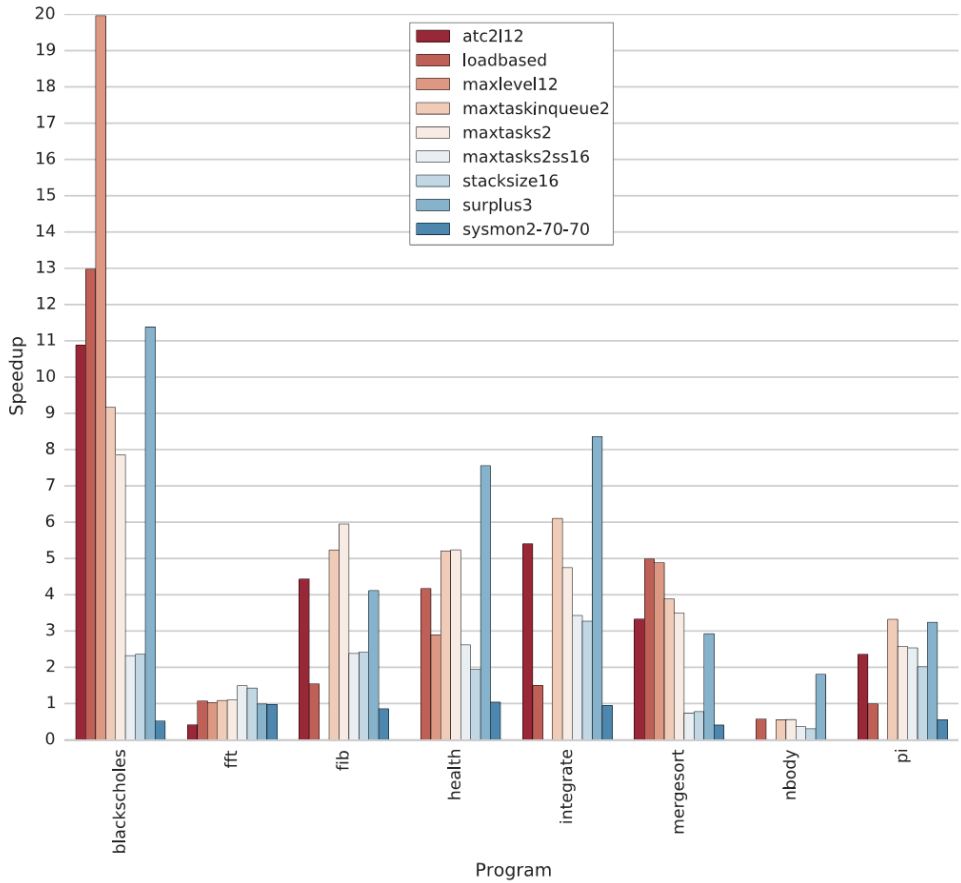

Abbildung: Vergleich der Cutoff Mechanismen[Alc16]

Vergleich

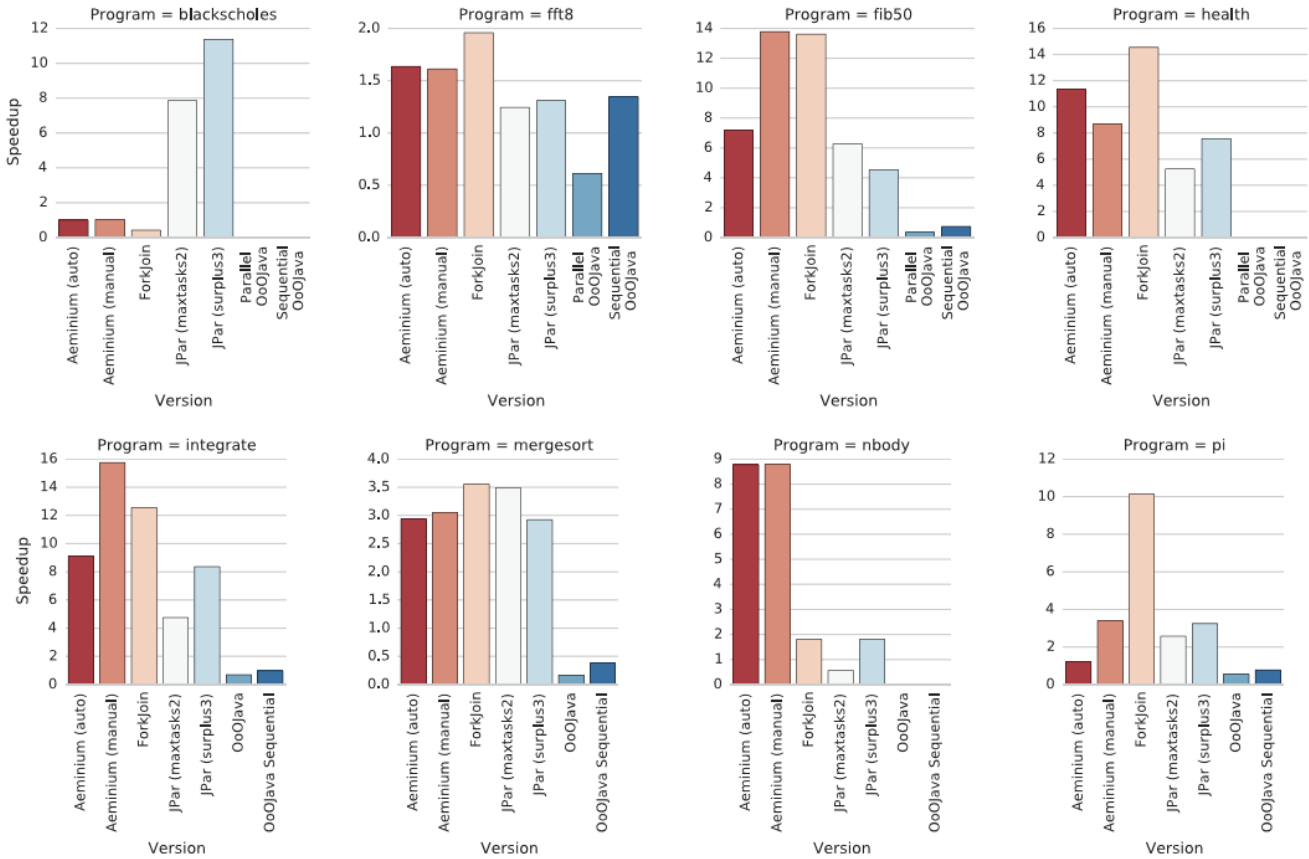

Abbildung: Leistungsvergleich [Alc16]

Zusammenfassung

- Sequentieller Code lässt sich automatisch in ein Task-Based Programm transformieren
- Besteht aus:
  - Kompilierung
  - Laufzeit Management
- Für optimale Ergebnisse müssen abhängig vom Programm verschiedene Laufzeit Parametern getestet werden
- Mit gut gewählten Einstellungen, besser als alternative Lösungen, aber schlechter als Hand geschrieben
- Zukünftig: Beste Parameter finden durch Programmstruktur

Literatur

[Alc16] Alcides Fonseca, Bruno Cabral, Joao Rafael, Ivo Correia. Automatic Parallelization: Executing Sequential Programs on a Task-Based Parallel Runtime. *International Journal of Parallel Programming No. 44*, April 2016.

[Gee22] GeeksForGeeks. Loop level parallelism in computer architecture. https://www.geeksforgeeks.org/loop-level-parallelism-in-computer-architecture, 2022.

[Lei02] Leino, K., Poetzsch-He□ter, A., Zhou, Y. Using data groups to specify and check side e□ects. *ACM SIGPLAN Notice*, 2002.

[Wik22] Wikipedia. Abstract syntax tree. https://en.wikipedia.org/wiki/Abstract_syntax_tree, 2022. Accessed: -.