Hochschule für Telekommunikation Leipzig

Abschlussarbeit zum Erlangen des akademischen Grades
Bachelor of Engineering

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

vorgelegt von:	Hans Christian Rudolph
geboren am:	29.04.1994
in:	Weißenfels
Studiengang:	Kommunikations- und Medieninformatik 2013 (dual)
Matrikelnummer:	133227

Themensteller:	Hochschule für Telekommunikation Leipzig Gustav-Freytag-Str. 43-45 04277 Leipzig

Erstprüfer:	Prof. Dr. rer. nat. Matthias Krause
Zweitprüfer:	Dipl.-Pol. Thomas Günther

Datum:	10.10.2016

Vorwort

Warum sollten sich Programmierende mit den Konzepten funktionaler Sprachen auseinandersetzen? Schließlich existiert eine Vielzahl von Alternativen, die einfacher zu erlernen sind, eine deutlich größere Verbreitung genießen und mehr Community-Unterstützung erfahren.
Die Antwort ist denkbar einfach: weil die grundverschiedenen Herangehensweisen dieses Paradigmas den Horizont erweitern und neue, effiziente Lösungswege zu bestehenden Problemen offenbaren können. Wer sich mit einer dieser Programmiersprachen befasst hat, wird das Gelernte auch auf andere Bereiche transferieren und so zu expressivem, sicherem und leicht wartbarem Quellcode beitragen.
Diese Erkenntnis ist auch an etablierten Sprachen, wie C++, C# und Java, nicht spurlos vorbeigegangen. Obgleich sich ein rein funktionales Vorgehen nicht für jeden Anwendungsfall eignet, ist hier der deutliche Trend zu beobachten, stetig mehr funktionale Komponenten zu integrieren – ein erster Schritt in Richtung maximal modularer, zukunftsfähiger Softwareprojekte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

Automatische Speicherbereinigung (engl.: *Garbage Collection*) ist eine Funktionalität mancher Programmiersprachen, die zur Laufzeit eigenständig nicht länger benötigte Objekte aus dem Hauptspeicher der Maschine entfernt. Ist diese Speicherbereinigung nicht vorhanden, muss der Speicher manuell freigegeben werden.

Bedarfsauswertung (engl.: *lazy evaluation*) bezeichnet eine Evaluationsstrategie, bei der die Argumente einer Funktion erst dann ausgewertet werden, wenn sie benötigt werden. Mit dem Glasgow Haskell Compiler (GHC) erstellte Haskell Programme verwenden standardmäßig diese Form der nicht-strikten Auswertung.

Cache Hit-Rate beschreibt den prozentualen Anteil erfolgreicher Cache-Anfragen.

Effizienz ist das Verhältnis von Speedup und der Zahl eingesetzter Prozessoren (vgl. Rauber 2012, S.179)

Funktionen höherer Ordnung oder *Funktionale* sind „Funktionen, die als Parameter oder Resultate wieder Funktionen haben [...]" (Pepper und Hofstedt 2006, S.24).

Kosten sind definiert als die gesamte Zeit, die von allen Prozessoren für die Ausführung eines parallelen Programms benötigt wurde, also das Produkt der parallelen Ausführungszeit und Anzahl der Prozessoren (vgl. Rauber 2012, S.176).

Paralleler Overhead bezeichnet den Mehraufwand für den Prozessor, welcher durch die Aufteilung der Berechnung in parallelen Systemen entsteht. Quellen parallelen Overheads sind daher vor allem Erstellung, Synchronisation und Destruktion von Prozessen oder Threads.

Relativer Speedup ist das Verhältnis der Zeit, welche für die Ausführung eines Algorithmus auf einem CPU-Kern benötigt wird und der Zeit, die auf mehreren Prozessoren benötigt wird (vgl. Sahni und Thanvantri 1995, S.13).

Skalierbarkeit beschriebt „[d]as Verhalten der Leistung eines parallelen Programms bei steigender Prozessoranzahl" (Rauber 2012, S.179). Halbiert sich die Antwortzeit bei einer Verdopplung der eingesetzten CPUs, weißt das Programm einen linearen Speedup auf – es „skaliert" linear.

Thunk ist eine unevaluierte Berechnungseinheit innerhalb eines Haskell Programms.

Wettlaufsituation „Eine Wettlaufsituation (engl.: *race condition*) besteht, wenn sich zwei oder mehr Prozesse oder Threads um eine Ressource bemühen und der Gewinner nicht vorhersagbar ist. Solche Wettlaufsituationen haben häufig den Effekt, dass ein Programm mal korrekt arbeiten, mal abbrechen kann, also das Ergebnis vom Ausgang des 'Wettlaufs' abhängig sein kann" (Gropp et al. 2007, S.273).

Abkürzungen

API Application Programming Interface (dt. *Programmierschnittstelle*)

CPU Central Processing Unit (dt. *Zentrale Recheneinheit / Prozessor*)

DMM Distributed Memory Machine (dt. *Rechner(-netz) mit verteiltem Speicher)*

DSL Domain Specific Language (dt. *Domänenspezifische Sprache*)

EC2 Elastic Compute Cloud

GC Garbage Collector (dt. *Automatische Speicherbereinigung*)

GCC GNU Compiler Collection

GHC Glasgow Haskell Compiler

GpH Glasgow parallel Haskell

HdpH Haskell distributed parallel Haskell

HEC Haskell Evaluation Context

HNF Head Normal Form

KVP Key-Value Pair (dt. *Schlüssel-Wert Paar*)

MPI Message Passing Interface

NF Normal Form

NUMA Non Uniform Memory Access (dt. *Nicht-einheitlicher Speicherzugriff*)

OpenMP Open Multi-Processing

PMU Performance Measurement Unit

Pthread POSIX-Thread

RSS Resident Set Size

SMM Shared Memory Machine (dt. *Rechner mit gemeinsam genutztem Speicher*)

SMP Symmetric Multi Processor (dt. *Symmetrischer Mehrkernprozessor*)

UMA Uniform Memory Access (dt. *Einheitlicher Speicherzugriff*)

WHNF Weak Head Normal Form

Kapitel 1

Einleitung

Das folgende Kapitel gewährt einen Überblick über die zu untersuchende Thematik und die zugrundeliegende wissenschaftliche Motivation. Es wird beschrieben mit welcher konkreten Fragestellung sich diese Arbeit beschäftigt und welche Methodik zur Beantwortung ebenjener angewandt wird.

1.1 Problemstellung und Erkenntnisinteresse

Seit ihrer breiten Markteinführung im Jahr 2005[1]haben sich Mehrkernprozessoren als de facto Standard unter den zentralen Recheneinheiten (engl.: *Central Processing Units (CPUs)*) etabliert. So enthält ein Großteil aktuell veröffentlichter Endgeräte, gleich ob Desktop-Rechner oder Smartphone, mehrere Kerne pro Chip. In moderne Intel-Prozessoren der Serverklasse werden gar bis zu 24 dedizierte Recheneinheiten integriert (vgl. Intel Corporation 2016). Der Trend steigender CPU-Zahlen sowie die von Märtin (2014) beschriebene Diversifikation der Architekturen verlangen nach effizienten Mitteln diese Rechensysteme zu programmieren. Es erwächst der Bedarf für generalisierte, möglichst einfache Methoden und Werkzeuge um Programme für Mehrkernprozessoren zu entwickeln. Denn eine Steigerung der Performanz kann nur erreicht werden, wenn sich Software die zugrundeliegende Hardware auch zu Nutze macht.

C++ und Haskell sind Programmiersprachen, welche die Entwicklung paralleler und nebenläufiger Programme ermöglichen. Die grundlegenden Ansätze, wie dies erreicht und umgesetzt wird, unterscheiden sich jedoch stark voneinander. Dies ist vorrangig bedingt durch die verschiedenen Paradigmen, welche beiden Sprachen zugrunde liegen: zum einen, im Fall von C++, die imperative Programmierung, bei der eine inhärent sequenzielle Folge von Anweisungen beschrieben wird, welche durch den Prozessor auszuführen ist. Zum anderen die funktionale Programmierung, zu der Haskell gehört, bei der Programme einzig und allein aus Definition, Komposition und Applikation von Abbildungen, d.h. Funktionen im engeren mathematischen Sinne bestehen, die durch den Prozessor ausgewertet werden. Das generelle Verständnis von Programmen in diesen Sprachen beinhaltet also per se keinerlei Sequenzialität.

[1]Der allererste nicht-integrierte CPU mit mehreren Kernen war im Jahre 2001 der IBM POWER4. Erst im Jahre 2005 jedoch folgten vergleichbare Modelle der Hersteller Intel und AMD.

Diese Eigenschaft lässt den Schluss zu, funktionale Sprachen würden sich besser für die Programmierung von Mehrkern- und Multiprozessorsystemen eignen als andere. Jedoch hat es sich in der Vergangenheit als schwierig erwiesen, auf Basis dieser günstigen Voraussetzung, hinreichend überzeugende Programme zu entwickeln. Marlow, S. Peyton Jones und Singh (2009, S.1) stellen fest: „Viele Arbeiten beschreiben vielversprechende Ideen, aber sehr viel weniger beschreiben reale Implementierungen"[2].

Es stellt sich daher die Frage: Inwieweit unterscheidet sich die Performanz paralleler Programme beider Sprachen, generiert mit aktuellen Compilern, tatsächlich? Im Rahmen der Bachelorarbeit soll dies exemplarisch am Beispiel der nachfolgend beschriebenen Aggregationsfunktion untersucht werden.

Es sind Datenstrukturen aus einer Menge lokal gespeicherter Dateien einzulesen. Diese Strukturen enthalten jeweils definierte Schlüsselattribute sowie die Attribute „Dauer" und „Volumen", deren ganzzahlige Werte es zu aggregieren gilt. Eine Datei kann dabei mehrere Datensätze mit unterschiedlichen Schlüsseln enthalten. Anhand dieser Schlüssel ist zunächst eine Gruppierung der eingelesenen Daten durchzuführen. Die abschließende Summation der beiden relevanten Werte erfolgt danach pro Gruppe. Resultierende Aggregate sind erneut lokal auf der Festplatte zu speichern. Bei der Realisierung dieser Funktionalität soll das Programmiermodel MapReduce, welches im Abschnitt 2.3 beschrieben wird, dazu dienen, die Verarbeitungslogik in verschiedene Phasen paralleler Berechnungen zu gliedern.

1.2 Stand der Technik

Parallelität findet in Computersystemen auf mehreren Ebenen statt. Rauber (2012, S.10ff) bietet diesbezüglich eine sehr gute Übersicht. Für die Softwareentwicklung sind allen voran gleichzeitige Berechnungen auf Prozess- bzw. Threadebene von Interesse, da diese mit den Mitteln der parallelen Programmierung aktiv zu kontrollieren sind. Daher soll fortlaufend beim Begriff der Parallelisierung bzw. Parallelität nur noch von ebenjener Form der Programmoptimierung die Rede sein. Grundidee ist, dass jede der verfügbaren CPUs gleichzeitig einen separaten Kontrollfluss in Form von Prozessen oder Threads abarbeitet. Die hierfür nötige Koordination kann dabei über verschiedene Arten der Inter- und Intraprozesskommunikation erfolgen. Besonders verbreitete Methoden sind, je nach physikalischem Aufbau des Rechners, gemeinsam genutzter Speicher (engl.: *Shared Memory*) und nachrichtenbasierte Kommunikation (engl.: *Message-Passing*).

Parallelisierung von Programmen in imperativen Sprachen erweist sich häufig als eine relativ aufwendige Aufgabe. Da die Implementierung oftmals einen sehr hardwarenahen Charakter hat und detaillierte Abläufe des Algorithmus, wie Datenverteilung und Prozesskommunikation, durch die Programmierenden zu beschreiben sind, bringt eine solche Veränderung des Kontrollflusses großen Mehraufwand bei der Entwicklung mit sich.

Funktionale Sprachen hingegen realisieren Parallelität auf einer höheren Abstraktionsebene. Die tatsächliche Umsetzung auf maschinennahe Instruktionen wird durch den Compiler übernommen und so ein Großteil der Komplexität verschleiert. Das Resultat sind in der Regel einfachere Implementierungen, deren Leistungen jedoch selten an die imperativer Programme heranreichen. (Vgl. Aljabri 2015, S.37f)

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

Imperative Programmierung: Die Bibliothek POSIX-Threads (Pthreads) ist eine Implementierung des Threadmodells für UNIX-ähnliche Systeme in den Sprachen C und C++. Dabei obliegt es den Programmierenden, diese leichtgewichtigen Prozesse zu erstellen, ihnen Aufgaben zuzuweisen, ihre Ausführung zu synchronisieren sowie die einzelnen Datenflüsse zu koordinieren. Anschließend werden sie vom Scheduler des Betriebssystem auf die vorhandenen Prozessorkerne aufgeteilt.

Eine modernere Alternative stellt seit C++11 die Standardbibliothek `<std::thread>` dar, welche jedoch prinzipiell die selbe Funktionalität bietet. Diese Ansätze können in umfangreichen Softwareprojekten schnell sehr komplex und unübersichtlich werden. Daher ist ein alternativer Ansatz, die Beschreibung von Parallelität auf einer höheren Abstraktionsebene durch definierte Programmierschnittstellen (engl.: *Application Programming Interfaces (APIs)*) zu ermöglichen, welche Mittel zur Annotation paralleler und nebenläufiger Bereiche des Quellcodes bieten.

Prominentes Beispiel dafür ist das Projekt *Open Multi-Processing (OpenMP)*, bestehend aus Compiler-Anweisungen, Bibliotheksfunktionen und Umgebungsvariablen mit APIs für die Programmiersprachen C, C++ und FORTRAN (vgl. van Waveren 2013, What is OpenMP?). Es wird seit 1997 in Zusammenarbeit mehrerer namhafter Hardware- und Compilerhersteller entwickelt, „mit dem Ziel [...], einen einheitlichen Standard für die Programmierung von Parallelrechnern mit gemeinsamen Adressraum zur Verfügung zu stellen" (Rauber 2012, S.357). Parallel auszuführende Regionen werden duch das Hinzufügen einfacher Direktiven in bestehenden Quellcode eingeführt. Alle weiteren Aufgaben werden von der API, bzw. vom jeweiligen Compiler übernommen. Bei Umsetzung der parallelen Konstrukte existiert kein standardisiertes Vorgehen, so dass die tatsächliche Leistungsfähigkeit auch stets von der Implementierung des jeweiligen Compilers abhängt. Laut Chapman, Jost und Van Der Pas (2008, S.277) findet meist eine Übersetzung der OpenMP Befehle in compilerspezifische Laufzeitroutinen statt, seltener in modifizierten Quellcode, welcher Aufrufe von Pthread-Funktionen enthält. OpenMP ist jedoch in vielen Fällen eine sehr unkomplizierte und gleichzeitig effiziente Methode, existierende Programme ohne tiefgreifende Eingriffe von paralleler Hardware profitieren zu lassen (vgl. Abschnitt 3.3.3).

Shared Memory Bibliotheken wie *OpenMP* oder auch Intels *Threading Building Blocks* eignen sich für monolithische Rechner, deren Prozessoren Zugriff auf einen physikalisch zusammenhängenden Speicher haben. Wird jedoch ein Verbund mehrerer getrennter Recheneinheiten eingesetzt, sind diese Technologien nicht mehr anwendbar, da jede von Ihnen einen privaten Speicher verwaltet und ausschließlich auf diesen zugreifen kann. In solchen Szenarien kommt für die Synchronisation und den Austausch zu verarbeitender Daten nachrichtenbasierte Kommunikation zum Einsatz. Werden von einem Prozess eines verteilten Programms Daten benötigt, die sich nicht im lokalen Speicher der jeweiligen Maschine befinden, so müssen diese explizit von einem der verbundenen Rechenknoten angefordert werden. Für diese Funktionalität existieren mehrere vorgefertigte Bibliotheken, die den sogenannten *Message Passing Interface (MPI)* Standard implementieren. Das Projekt OpenMPI beispielsweise bietet APIs in den Sprachen C, C++, FORTRAN und Java.

Funktionale Programmierung: Funktionalen Programmiersprachen ist Parallelität im Sinne der gleichzeitigen Ausführung von Programmteilen konzeptionell inhärent. Bei diesem Paradigma werden Funktionen nach mathematischem Vorbild als Abbildung von Werten aus einem Definitions- in einen Wertebereich verstanden. In rein funktionalen Sprachen – angelehnt an „rein mathematische" Funktionen – ist daher jegliche Berechnung frei von Nebeneffekten. Sie verändert keine Programmzustände und ist auch nicht

von solchen abhängig. Daraus folgt, dass eine Funktion bei gleichen Eingangsparametern stets das gleiche Ergebnis zurückliefern muss, unabhängig davon, zu welchem Zeitpunkt oder in welchem Kontext sie ausgewertet wird. Diese Eigenschaft wird als referenzielle Transparenz (engl.: *„Referential transparency"* (Søndergaard und Sestoft 1990)) bezeichnet. Durch sie ist es dem Compiler möglich den Algorithmus genau zu analysieren und, wo sinnvoll, Optimierungen in Form von Bedarfsauswertung oder paralleler Berechnung vorzunehmen. Theoretisch können in reinen Funktionen sämtliche Teilberechnungen gleichzeitig ausgewertet werden, was jedoch aufgrund zu hoher Granularität der Parallelisierung einen eher negativen Effekt auf die Laufzeit hätte. Da auch die Zuweisung von Werten zu Variablen die Einführung eines Programmzustandes bedeutet und somit die referenzielle Transparenz zerstören würde, beinhalten rein funktionale Sprachen keine veränderlichen Parameter sondern nur Konstanten. Ein einmal zugewiesener Wert ändert sich im Laufe des Programms nicht.

Als eine der richtungsweisendsten Arbeiten auf dem Gebiet der funktionalen Programmierung, sei an dieser Stelle auf Hughes (1989) verwiesen, der zusätzlich zu den genannten Merkmalen, die Modularität funktionaler Programme als einer ihrer wesentlichen Vorteile aufzeigt. Darauf aufbauend ziehen Hu, Hughes und Wang (2015) ein Resumé, welchen tiefgreifenden Einfluss dieses Paradigma im Laufe der letzten 25 Jahre auf die Softwareentwicklung genommen hat. So wurden seitdem nicht nur eine Vielzahl von Ideen aus der funktionalen Welt in Sprachen wie C++, C# und Java übernommen, auch funktionale Sprachen selbst finden vermehrt Anwendung außerhalb des akademischen Sektors: Der Nachrichtendienst WhatsApp etwa nutzt das ursprünglich von Ericsson entwickelte *Erlang* für seine serverseitige Programmierung. Die Infrastrukturen großer Webservices wie Twitter und LinkedIn hingegen basieren zum Teil auf *Scala*, einer funktionalen Sprache für die Java Virtual Machine (vgl. Hu, Hughes und Wang 2015, S.1).

Haskell ist eine rein funktionale Sprache, welche Ende der 1980er Jahre von einem Expertenkomitee entworfen wurde, um eine gemeinsame Terminologie für die Forschung an Programmiersprachen zu etablieren sowie eine Basis für die Entwicklung praktikabler, funktionaler Applikationen zu schaffen (vgl. Hudak et al. 2007, S.4). Seitdem ist es zu einer der populärsten funktionalen Sprache avanciert und erfreut sich noch immer stetig wachsender Verbreitung. Verschiedene Methoden für die Parallelisierung von Programmen sind in Haskell standardmäßig integriert. Dazu zählen Möglichkeiten zur expliziten Steuerung von Nebenläufigkeiten, wie das Erstellen von leichtgewichtigen Haskell-Threads (S. L. Peyton Jones und Wadler 1993), sowie Beschreibungen semi-expliziter Parallelität, bei denen lediglich Bereiche potentiell paralleler Ausführung annotiert werden und die eigentliche Koordination durch Haskells Laufzeitumgebung geschieht. Letztere werden im Abschnitt 2.2 näher erläutert. Bestimmte Funktionalitäten werden durch spezialisierte Spracherweiterungen für die Entwicklung paralleler Programme in Haskell zur Verfügung gestellt. Die beiden bekanntesten Vertreter sind *Glasgow parallel Haskell (GpH)* (Philip W Trinder et al. 1996) und *Eden* (Breitinger, Loogen, Mallen et al. 1997).

Die Implementierung von *GpH* variiert abhängig von der zugrundeliegenden Architektur. Auf *Shared Memory Machines (SMMs)*, also Mehrkernrechnern mit physikalisch zusammenhängendem Speicher, wird die Koordination zwischen den verschiedenen Kontrollflüssen mittels geteilter Datenstrukturen realisiert. Auf verteilten Systemen hingegen basiert die Synchronisation der Prozesse auf dem MPI Standard und der dedizierten Laufzeitumgebung *GUM*. Aljabri (2015, S.57ff) spricht in diesem Kontext von „GpH on GHC-SMP" und „GpH on GHC-GUM".

Eden wiederum erweitert Haskell um Sprachkonstrukte für die explizite Instanziierung von parallelen Eden-Prozessen. Diese repräsentieren reine Berechnungen, allein abhängig von Parametern, Eingabegrößen und lokal definierten Funktionen. Die Synchronisation

und Steuerung der Datenflüsse wird mittels sogenannter Channels von *Edens* Laufzeitumgebung realisiert, welche ihrerseits ebenfalls auf dem Versenden von Nachrichten basieren (vgl. Breitinger, Loogen, Ortega-Mallén et al. 1996, S.2ff). *Eden* ist damit primär für die Programmierung auf verteilten Systemen gedacht.

Aljabri (2015) bietet eine Beschreibung dieser und weiterer paralleler Haskell Erweiterungen, wie z.B *Cloud Haskell* und *Haskell distributed parallel Haskell (HdpH)*.

MapReduce Modell: Es wurden in der Vergangenheit bereits mehrere Beschreibungen des MapReduce-Algorithmus in Haskell präsentiert. Lämmel (2008) bietet eine sehr ausführliche Analyse des Modells, eine entsprechende Beschreibung in Standard-Haskell und zeigt dabei Möglichkeiten der Parallelisierung auf. Die Arbeit erläutert allerdings keine vollständige Implementierung. Loogen (2012, S.5f) beschreibt anschaulich die Erstellung funktionaler MapReduce-Programme unter Verwendung der in *Eden* integrierten *Skeletons* – vordefinierte Funktionen höherer Ordnung zur Steuerung paralleler Berechnungsabläufe.

1.3 Forschungsfrage

Der fundamentale Unterschied bei der Erstellung von Programmen für verteilte Systeme durch nachrichtenbasierte Kommunikation und für monolithische Hardware mit geteiltem Speicher erfordert eine klare Abgrenzung bei Betrachtung der beschriebenen Problemstellung. Es kann im Zuge dieser Bachelorarbeit keine generalisierte Aussage getroffen werden, die bei der Leistungsbeurteilung paralleler Programme beide technische Szenarien berücksichtigt. Daher wird die Einschränkung getroffen, Implementierungen für einzelne Rechner mit Shared Memory zu untersuchen. Dies ist zum einen dadurch begründet, dass für die durchzuführenden Messungen kein verteiltes System zur Verfügung steht. Zum anderen würde sich bei der zu untersuchenden Problemgröße ein verteiltes System und die damit verbundenen Mehraufwände – sowohl bei der Programmierung als auch während der Laufzeit in Form eines erhöhten Kommunikations-Overheads für das System – nicht lohnen.
Die folgenden Ausführungen fokussieren sich deshalb auf die Beantwortung der Frage:

> *Welche der beiden Programmiersprachen, Haskell und C++, erlaubt für das gegebene Problem die Erstellung des performanteren Programms für ein Mehrprozessorsystem mit Shared Memory?*

Im Zuge der Implementierung soll außerdem gezeigt werden, warum das MapReduce Modell einen adäquaten Lösungsansatz für die Parallelisierung dieses speziellen Problems darstellt. Während zu Beginn einige grundlegende Konzepte der funktionalen Programmierung im Allgemeinen und Haskell im Speziellen vorgestellt werden, ist diese Arbeit keineswegs als Einstieg in die Programmiersprache Haskell gedacht. Daher sollen im weiteren Verlauf lediglich einzelne zentrale Funktionen näher erläutert werden. Für eine Einführung in die Syntax und die Strukturierung von Haskell Programmen wird auf die entsprechende Literatur verwiesen (z.B. Thompson 2015, O'Sullivan, Goerzen und Stewart 2008).

Kenntnisse in der Programmiersprache C++ sowie das Verständnis grundlegender Elemente eines Betriebssystems im Zusammenhang mit der gleichzeitigen Ausführung von Programmen, wie Prozessen, Threads, Schedulern, et cetera, werden vorausgesetzt. Auf sie wird in den theoretischen Vorbetrachtungen nicht näher eingegangen.

Des Weiteren ist es nicht Ziel der Arbeit, detailliert zu analysieren *warum* eine der Sprachen performanter ist als die andere. Dies ist primär von dem verwendeten Compiler

abhängig, da er die Aufgabe hat, aus dem Quellcode möglichst effiziente Programme für den Prozessor zu erstellen. Für die Kompilierung der Software wird eine aktuelle Version des populärsten Compilers der jeweiligen Programmiersprache verwendet und die von ihm generierten Programme als Referenz festgelegt. Für Haskell ist dies der Glasgow Haskell Compiler (GHC), für C++ die GNU Compiler Collection (GCC).

1.4 Methodik

Für die Beantwortung der Forschungsfrage wird ein Vergleich zwischen zwei zu entwickelnden Prototypen erstellt. Brunswig (1910, S.62) definiert die Methodik des Vergleichs als Tätigkeit „[z]wei Objekte [...] aufmerksam nacheinander mit spezieller Hinsicht auf ihr gegenseitiges Verhältnis [zu] betrachten". Als solche erlaubt eine komparatistische Vorgehensweise, in der begrenzt zur Verfügung stehenden Zeit, auf Basis empirischer Untersuchungen, begründete Aussagen über die Leistung beider Programmiersprachen im Kontext der beschriebenen Problematik zu treffen.

Zunächst wird die im Abschnitt 1.1 beschriebene Aggregationsfunktion in das Programmiermodell MapReduce überführt. Dieser Arbeitsschritt beinhaltet eine Analyse, welche Programmteile sich für eine parallele Bearbeitung eignen und welche Abschnitte zwangsläufig sequenzieller Natur sind. Es schließt sich die Implementierung der Software an, welche nach einem inkrementellen Entwicklungsmodell erfolgt. Das bedeutet, die logisch abgeschlossenen Funktionsblöcke der Software werden losgelöst voneinander entwickelt und Schritt für Schritt zur vollumfänglichen Softwarelösung zusammengesetzt. Da die Problemstellung klar definiert ist sowie der grundlegende Aufbau des Algorithmus durch MapReduce gegliedert wird, ist zu erwarten, dass sich die Spezifikation im Laufe der Entwicklung nicht ändert und eine schrittweise Umsetzung somit einen durchaus validen Lösungsansatz darstellt. Darüber hinaus begünstigt dieses Vorgehen die Durchführung separater Leistungstests und Optimierungen der einzelnen Softwarekomponenten. Zur einfacheren Überprüfung der Korrektheit der parallelen Programme, werden zunächst funktional identische, sequenzielle Lösungen entwickelt, welche die Basis der parallelen Versionen bilden.

Daraufhin werden verschiedene Möglichkeiten der Parallelisierung in beiden Sprachen verglichen und die jeweils am besten geeignete Methode für die Realisierung ausgewählt. Diese Entscheidung ist stark vom verwendeten Zielsystem abhängig, dessen technische Merkmale im Abschnitt 4.1 dargelegt werden.

Die für eine Verhältniserkenntnis mindestens zu betrachtenden zwei Objekte sind die beiden Software-Implementierungen. Des Weiteren gilt es Eigenschaften zu definieren, bezüglich derer wertende Relationen zwischen beiden aufgestellt werden können. Um eine möglichst nachvollziehbare Gegenüberstellung zu erreichen, wird die Leistung daher anhand der folgenden Kriterien bewertet, welche im Abschnitt 2.4 detailliert erläutert werden:

- Relativer Speedup
- Effizienz
- Paralleler Overhead [%]
- Kosten [s]
- Durchsatz [Records/s]
- Durchschnittlich allozierter Arbeitsspeicher [MiB]
- Cache Hit-Rate [%]

Die Erfassung der Leistungsdaten erfolgt mithilfe geeigneter Programme für das Profiling von Linux-Applikationen. Nähere Informationen zu der verwendeten Test-Software werden ebenfalls im Abschnitt 2.4 präsentiert.

Ein Ziel der Bachelorarbeit ist es, auf Basis der oben genannten Kriterien eine gesonderte Analyse anhand einzelner Gesichtspunkte vorzunehmen. So lässt sich eine detaillierte Bewertung bezüglich der Stärken und Schwächen der jeweiligen Implementierungen erstellen. Aus den Ergebnissen dieser Einzelbetrachtungen werden im Anschluss Aussagen über die Eigenschaften der verwendete Sprache abgeleitet.

Für die Auswertung der Messungen, das Berechnen relevanter Werte und die Visualisierung in Diagrammform wird die Programmiersprache R verwendet. Als eine Sprache, die primär für statistische Berechnungen und die Auswertung von Daten gedacht ist, stellt sie ein geeignetes Werkzeug für diese Aufgaben dar.

Ein weiteres Ergebnis ist eine Gesamtbewertung, welche sämtliche dargelegten Leistungskriterien einbezieht. Hierfür werden die einzelnen Messwerte beider Programme zueinander ins Verhältnis gesetzt und normalisiert. Durch eine Summation ebenjener Werte pro Programmiersprache ergibt sich eine Gleitkommazahl, durch die sich der Leistungsunterschied beider Implementierungen quantitativ beschreiben lässt. Für eine genaue Erläuterung dieser Rechnung sowie die Identifizierung relevanter Testfälle, sei auf Abschnitt 2.4 verwiesen.

Kapitel 2

Theoretische Grundlagen

Dieses Kapitel erläutert Begrifflichkeiten und Sachverhalte, welche für das Verständnis und die Einschätzung der darauffolgenden Untersuchungen notwendig sind. Es wird außerdem beschrieben auf welche Weise die zu implementierenden Programme verglichen werden sollen.

2.1 Nebenläufigkeit, Parallelität und Parallelrechner

Begriffsklärung

Die Bezeichnungen „Parallelität" und „Nebenläufigkeit" lassen vermuten, Berechnungen würden gleichzeitig, bzw. nebeneinander ablaufen und dass sie somit synonym verwendet werden könnten. Jedoch besteht zwischen beiden ein entscheidender Unterschied, den es hervorzuheben gilt. Im Laufe dieser Arbeit sollen diese Begriffe verstanden werden, wie nachfolgend definiert:

Parallelität (engl.: *parallelism*) beschreibt die *gleichzeitige* Ausführung von Aufgaben. Hierfür ist die Verwendung eines Parallelrechners, d.h. eines Rechners mit mehreren Recheneinheiten, zwingend erforderlich. Ziel ist es, das Ergebnis einer Berechnung schneller zu erhalten oder Berechnungen durchzuführen, die mit den Mitteln serieller Berechnung in ihrem Umfang oder ihrer Genauigkeit nicht möglich wären. Die gleichzeitige Berechnung darzustellender Bildelemente durch eine Grafikkarte ist ein Beispiel für parallele Berechnung.

Nebenläufigkeit (engl.: *concurrency*) beschreibt die *unabhängige* Ausführung von Aufgaben. Dies kann auf einer oder mehreren Recheneinheiten geschehen. Es ist also nicht ausgeschlossen, dass auch nebenläufige Algorithmen hinsichtlich ihrer Laufzeit von mehreren Prozessoren profitieren. Ziel ist es, mehrere Berechnungen „quasi-gleichzeitig" bearbeiten zu können, um so die Benutzerfreundlichkeit eines Programms zu verbessern. Ein Webserver, der mehreren Nutzern unabhängig voneinander Daten zu Verfügung stellt, ist ein Beispiel für nebenläufige Verarbeitung.

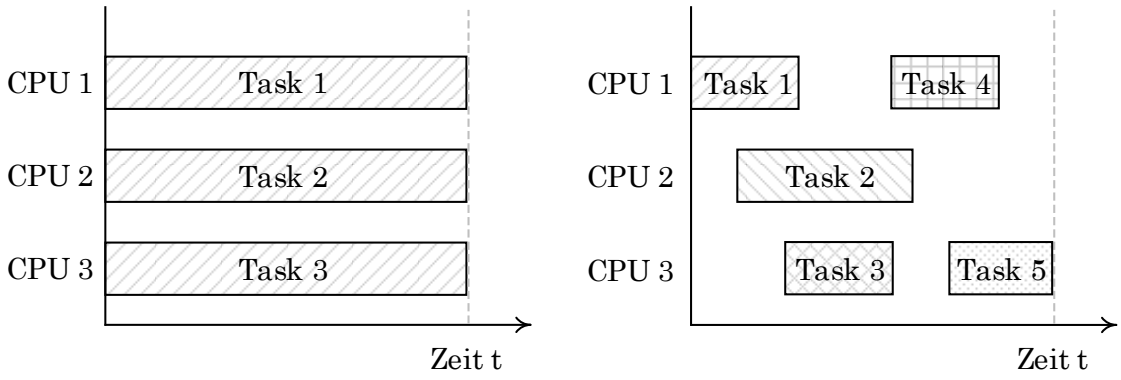

Abbildung 2.1: Parallelität (links) und Nebenläufigkeit (rechts) auf paralleler Hardware

Beide Modelle sind in Abbildung 2.1 noch einmal bildlich dargestellt. Aus diesen Definitionen lässt sich folgender Schluss ziehen: Während Nebenläufigkeit einen zentralen Bestandteil der Definition gewisser Anwendungen darstellt, ist Parallelität eine reine Möglichkeit der Optimierung von Algorithmen ohne semantischen Einfluss. Dies hat zur Folge, dass nebenläufige Berechnungen notwendigerweise nicht deterministisch sind, d.h. die Ergebnisse verschiedener Ausführungen eines solchen Algorithmus unterschiedlich sein können. Ein möglicher Grund dafür sind z.B. Scheduling-Entscheidungen des Prozessors, der die Ausführung eines Prozesses startet und dafür einen anderen zurückstellt. Parallele Berechnungen hingegen garantieren ein deterministisches Resultat – bei gleichen Eingangsparametern haben sie stets das gleiche Ergebnis. (Vgl. Hoareetal 2001, S.22)

Parallelrechner

An dieser Stelle ist eine klare Definition des zu betrachtenden Rechnertyps nötig. Hierfür existieren mehrere Modelle, welche Computer anhand verschiedener Charakteristika unterteilen. Nachfolgend werden drei gängige Kategorisierungen vorgestellt, basierend auf logischer Architektur, Speicherorganisation und der Speicherzugriffszeit.

Häufig wird zunächst die Flynnsche Klassifizierung (Flynn 1966) bemüht, welche die Einteilung anhand der Prozessoranzahl und den von ihnen gesteuerten Kontroll- und Datenflüssen vornimmt. So werden folgende Klassen unterschieden:

1. Single Instruction Stream–Single Data Stream (SISD)
2. Multiple Instruction Stream–Single Data Stream (MISD)
3. Single Instruction Stream–Multiple Data Stream (SIMD)
4. Multiple Instruction Stream–Multiple Data Stream (MIMD)

Aktuell verwendete Mehrkernprozessoren arbeiten üblicherweise nach dem MIMD-Prinzip. Sie enthalten eine gewisse Anzahl von dedizierten Recheneinheiten, „von denen jede einen separaten Zugriff auf einen [...] Datenspeicher und auf einen lokalen Programmspeicher hat" (Rauber 2012, S.19). Diese Form der Parallelität kommt zum Einsatz, wenn mehrere Prozessoren gleichzeitig unterschiedliche Aufgaben bearbeiten. Spielt einer von ihnen lokal Musik ab, während der andere eine Website im Browser herunterlädt, so ist das MIMD-Parallelismus.

Im Kontext des Programmiermodells MapReduce steht allerdings das SIMD-Modell im Vordergrund, bei dem identische Instruktionen von mehreren CPUs gleichzeitig auf eine Menge von Daten angewendet werden. Für diese Form wird daher auch der Begriff

Daten-Parallelismus (engl.: *„data parallelism"* (S. Peyton Jones, Leshchinskiy et al. 2008)) verwendet. Um solche Optimierungen zu nutzen, muss ein Programm mit entsprechenden Technologien entwickelt worden sein, wie im Abschnitt 1.2 beschrieben.

Weiterhin wird nach dem Aufbau des Hauptspeichers der Maschine unterschieden. 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 daher zwischen diesen geteilt werden. Daraus erwächst der Bedarf nach Synchronisation zwischen den Prozessen einer Maschine. Denn jede Veränderung, die einer der Prozessoren an gemeinsamen Variablen vornimmt, beeinflusst auch alle anderen. Insbesondere gilt es bei parallelen und nebenläufigen Zugriffen auf die Daten des globalen Speichers Wettlaufsituationen und daraus erwachsende Inkonsistenzen zu verhindern. Dies wird meist nach dem Verfahren des gegenseitigen Ausschlusses (engl.: *Mutual Exclusion (Mutex)*) durch spezielle Kontrollobjekte wie Semaphoren und Locks realisiert. Sie zeigen an, dass ein bestimmter Prozess mit der Bearbeitung einer geteilten Ressource beschäftigt ist, dieses Datum also momentan weder gelesen noch überschrieben werden sollte. Vor jedem Zugriff auf ein Element im Hauptspeicher muss daher geprüft werden, ob dessen Mutex-Objekt „frei" ist. Ist dies der Fall, wird es belegt und die Bearbeitung kann beginnen. Sämtliche weiteren Zugriffe durch andere Prozesse werden so lange blockiert, bis das Mutex-Objekt wieder freigegeben wird.

Gegenstück zu den SMMs sind *Distributed Memory Machines (DMMs)*, auch Multicomputer genannt. Wie in Abschnitt 1.2 beschrieben, arbeiten sie mit einem physikalisch separierten Speicher, welcher dem jeweiligen lokalen Prozessor privat zur Verfügung steht. Werden für eine Berechnung Daten eines entfernten Speicherbereichs benötigt, so werden diese mittels Nachrichtenübertragung angefordert und anschließend übertragen. Auch Mischformen aus Multiprozessor und Multicomputer sind möglich. (Vgl. Rauber 2012, S.20ff).

Symmetrische Multiprozessoren, bzw. *Symmetric Multi Processors (SMPs)*, stellen eine besondere Form von Rechnern mit gemeinsam genutzten Speicher dar. Ihre CPUs kommunizieren über einen zentralen Bus mit dem globalen Speicher. Sie verfügen daher nur über eine verhältnismäßig kleine Zahl von Prozessoren, da ansonsten die Auftrittswahrscheinlichkeit von Kollisionen beim Zugriff zu hoch wäre. Die Symmetrie bezieht sich dabei auf die Funktionalität der einzelnen Prozessoren sowie deren Sicht auf das System, „d.h. insbesondere, dass die Dauer eines Zugriffs auf den gemeinsamen Speicher für jeden Prozessor unabhängig von der zugegriffenen Speicheradresse gleich lange dauert" (Rauber 2012, S.26).

Da SMP-Systeme auch in einem Verbund betrieben werden können, der sich einen gemeinsamen virtuellen Speicher teilt, jedoch keine einheitlichen Speicherzugriffszeiten mehr gewährleisten kann, werden diesbezüglich Maschinen mit Uniform Memory Access und Non Uniform Memory Access unterschieden. Nicht-einheitliche Zugriffszeiten entstehen, wenn ein Prozessor Daten des logisch globalen Speichers anfordert, welche sich jedoch an einer physikalisch entfernten Lokation befinden. Die Adresse kann in diesem Fall zwar direkt angesprochen werden aber die Übertragung über den jeweiligen Transportbus ist langsamer als ein lokaler Zugriff.

Aufgrund der im Abschnitt 1.3 begründeten Abgrenzung, soll im Kontext dieser Bachelorarbeit lediglich die Programmierung von SMMs, betrachtet werden. Für die Realisierung der im Abschnitt 1.1 beschrieben Funktionalität werden daher ausschließlich Programme für Rechner entwickelt, welche das MIMD-/SIMD-Prinzip unterstützen und einheitliche Speicherzugriffszeiten besitzen.

2.2 Besonderheiten der Programmiersprache Haskell

Die im Folgenden beschriebenen Eigenschaften sind in dieser Form allein Haskell zu eigen. Übergreifende Elemente, wie die in Abschnitt 2.3 beschriebenen Funktionen höherer Ordnung, sind hingegen in einer Vielzahl funktionaler und mittlerweile auch imperativer Sprachen zu finden. Besonders die Thematik der Monaden ist hochkomplex und würde bei detaillierter Betrachtung den Rahmen dieser Arbeit übersteigen. Es soll daher nur ein kurzer Überblick geboten werden, für nähere Informationen wird auf die einschlägige Literatur verwiesen (Moggi 1988, Wadler 1992).

Haskell ist eine rein funktionale Programmiersprache. Als solche muss sie die Bedingung erfüllen, alle Funktionen seien frei von Nebeneffekten, um referenzielle Transparenz zu wahren (vgl. Abschnitt 1.1). Ein Nebeneffekt jedoch, unerlässlich für jede auch nur halbwegs nützliche Programmiersprache, ist die Möglichkeit der Ein- und Ausgabe, kurz IO. Um diese Funktionalität zu gewährleisten, ohne mit den rein funktionalen Eigenschaften der Sprache zu brechen, macht sich Haskell das Konzept der Monaden zu Nutze.

Monaden

Hauptaufgabe dieser, aus dem Gebiet der Kategorientheorie stammenden, mathematischen Strukturen, ist es eine generalisierte Art der Verkettung bestimmter Berechnungen zu definieren. Auch abgesehen von der Darstellung nebeneffektbehafteter Berechnungen in Form der `IO` Monade, finden sie breite Verwendung in Haskell – sei es für die Behandlung von Nicht-Determinismen, multipler Funktionsergebnisse oder für die Repräsentation interner Programmzustände.
Die Definition einer Monade umfasst in ihrer minimalen Ausprägung die folgenden drei Bestandteile:

- Einen Typkonstruktor `m`.
- Die Funktion `return`, welche einen beliebigen Wert entgegen nimmt und den entsprechenden monadischen Wert zurückgibt (`return :: a -> m a`)[3].
- Die Funktion `bind` (`>>=`), welche definiert, wie monadische Werte mit Funktionen zu kombinieren sind, die normale Werte entgegennehmen und erneut einen monadischen Wert erzeugen (`a >>= b :: m a -> (a -> m b) -> m b`).

Oftmals wird zum einfachen Sequenzieren zweier monadischer Funktionen noch der Operator `then` (`>>`) definiert, welcher einem `bind` entspricht, der das Ergebnis der ersten Funktion verwirft (`a >> b :: a >>= \_ -> b`). Ein monadischer Wert `m a` kann als Berechnung verstanden werden, die bei ihrer Ausführung den Wert `a` zurückgibt und dabei unter Umständen auch Nebeneffekte verursacht (vgl. Hudak et al. 2007, S.23).

Monadischer Input, repräsentiert durch die Typsignatur `IO a`, basiert auf der Prämisse, dass die entsprechende Aktion eine Funktion darstellt, deren Argument quasi der aktuelle Zustand ihrer „Umwelt" ist. Ihr Rückgabewert ist ein Paar aus einem veränderten Zustand und dem zu verarbeitenden Wert `a` (vgl. S. L. Peyton Jones und Wadler 1993). Eine Output-Operation des Typs `a -> IO ()` hingegen nimmt neben dem auszugebenden Wert `a` noch den Zustand der Umwelt entgegen und hat die ausgeführte IO-Aktion `IO ()` sowie einen veränderten Zustand als Ergebnis.

Die Bedeutung des abstrakten Arguments „Umwelt" für die `IO` Monade ist es, die Datenabhängigkeiten zwischen mehreren Operationen adäquat darzustellen. Als solches

[3]Die Doppelpunk-Notation ist eine Repräsentation von Funktionstypen in Haskell. Hinter dem `::` stehen getrennt durch `->` sowohl die Typen der Parameter als auch an letzter Stelle der des Ergebnisses.

macht es die sequenzielle Verkettung von Ein- und Ausgabeaktionen mit rein funktionalen Eigenschaften erst möglich. Dies lässt sich am Beispiel einer Funktion verdeutlichen, welche zwei Werte vom Typ `Char` auf die Standardausgabe druckt:

```
-- print :: a -> IO ()
printChars a b = print a >>
                 print b
```

Listing 2.1: Sequenzieren von IO Operationen

Die `print` Funktionen werden mit dem zuvor vorgestellten Kombinator `then` konkateniert. Auch wenn zwischen beiden keine offensichtliche Abhängigkeit besteht, sollen sie dennoch in der korrekten Reihenfolge ausgeführt werden. Dafür ist das von S. L. Peyton Jones und Wadler (1993, S.6f) beschriebene Übergeben des Zustands der Umwelt unerlässlich. Ohne diese, für die Programmierenden verborgene, Abhängigkeit zwischen aufeinanderfolgenden Funktionsaufrufen, würde der Compiler davon ausgehen, dass es keine Rolle spielt in welcher Reihenfolge die Aktionen ausgeführt werden. Das Ergebnis wäre nicht länger deterministisch.

Während der Erläuterungen zur Haskell Implementierung im Abschnitt 3.2 werden noch weitere Typen von Monaden eine Rolle spielen. Ein grundlegendes Verständnis der `IO` Monade ist nicht zuletzt deshalb wichtig, als das die Main-Funktion eines jeden Haskell-Programms in ihr operiert und stetig den Zustand der umliegenden Welt verändert.

Bedarfsauswertung, Thunks und Normalformen

Wie bereits in Abschnitt 1.2 erwähnt, werden Haskell Programme nach dem Prinzip der nicht-strikten Evaluierung, bzw. Bedarfsauswertung ausgeführt. Elemente, die für den aktuellen Berechnungsschritt nicht benötigt werden, bleiben so lange unausgewertet, bis ihr Ergebnis zu einem Punkt des Programmablaufs tatsächlich gebraucht wird. Trifft das Programm während der Laufzeit auf eine Berechnung, deren Auswertung es momentan nicht bedarf, so wird lediglich ein sogenannter *Thunk* alloziert. Thunks stellen eine Art abstrakte Berechnungseinheit dar. Sie können verschiedenste Elemente enthalten, z.B. primitive und komplexe Datentypen oder auch reine Funktionen. Sobald es zur Auswertung eines Thunks kommt, wird er im Hauptspeicher durch sein Ergebnis ersetzt, um mehrfacher Evaluierung durch verschiedene CPUs vorzubeugen.

Um während des Programmablaufs differenzieren zu können, zu welchem Grad ein bestimmtes Element bereits ausgewertet wurde, wird in der funktionalen Programmierung von verschiedenen Normalformen gesprochen: Weak Head Normal Form (WHNF), Head Normal Form (HNF) und Normal Form (NF). Eine Berechnung in NF ist vollständig ausgewertet und enthält keinerlei Teilberechnungen, die nicht auch in NF sind. Voraussetzung für die WHNF hingegen ist, dass eine Berechnung zumindest bis zu ihrem äußersten Konstruktor ausgewertet wurde, d.h. es ist bekannt, welchen Typ das Ergebnis dieser – eventuell noch unausgewerteten – Berechnung letztendlich haben wird. Laut dieser Definition sind Berechnungen in NF auch gleichzeitig in WHNF. Umgekehrt gilt dies jedoch nicht. Die Unterscheidung zwischen WHNF und HNF spielt für die folgenden Betrachtungen keine Rolle, daher soll letztere nicht weiter beleuchtet werden.

Haskells nicht-strikte Auswertung sowie die beiden relevanten Normalformen sollen nun an einem Beispiel in Listing 2.2 veranschaulicht werden. Die folgenden Befehle wurden im interaktiven Interpreter des GHC, GHCi, ausgeführt, der es erlaubt mittels des Befehls `:sprint` bereits evaluierte Elemente von anderen zu unterscheiden. Das Symbol `_` steht in diesem Kontext für einen Thunk.

```
>let x = (head xs, [head xs] ++ [last xs]) where xs = [1,2]
>:sprint x
x = (_,_)            -- Tupel in WHNF
>fst x
1
>:sprint x
x = (1,_)            -- Tupel in WHNF, 1. Element in NF, 2. unausgewertet
>length (snd x)
2
>:sprint x
(1,[_,_])            -- Tupel in WHNF, 1. Element in NF, 2. in WHNF
>print x
(1,[1,2])
>:sprint x
(1,[1,2])            -- Tupel in NF
```

Im ersten Schritt wird mittels der Standardfunktionen `head` und `last` ein Tupel konstruiert. Es besteht zum einen aus dem ersten Element einer Liste von Ganzzahlen, zum anderen aus einer neuen, zusammengesetzten Liste aus dem ersten und letzten Element. Das Ergebnis ist ein Tupel in WHNF. Es ist lediglich bekannt, dass es sich bei dem Ergebnis um ein Element des Haskell Tupelkonstruktors `(,)` handelt. Anschließend wird durch die Funktion `fst` das erste Elemente des Tupels zurückgegeben. Es wird dabei in NF überführt. Die Funktion `length (snd x)` gibt die Länge der Liste im zweiten Element des Tupels zurück. Um dies zu bewerkstelligen, ist es nicht nötig zu wissen, welche Werte die Listenelemente haben. Daher liegt nach diesem Schritt eine Liste mit zwei Thunks vor. Die Funktion `print` schreibt letztendlich das komplette Tupel auf die Standardausgabe. Erst an dieser Stelle werden alle enthaltenen Elemente vollständig ausgewertet.

Parallelität in der Haskell Laufzeitumgebung

Während in vielen Programmiersprachen üblicherweise nur zwischen Prozessen und Threads unterschieden wird, führt Haskell weitere Ebenen der Abstraktion ein – in Form von Haskell-Threads (S. Peyton Jones, Gordon und Finne 1996) und Sparks (Marlow, S. Peyton Jones und Singh 2009). Erstere ermöglichen es explizit nebenläufige Berechnungen zu erstellen, sind dabei aber wesentlich kostengünstiger in ihrer Kon- und Destruktion als herkömmliche Betriebssystemthreads. Sparks hingegen repräsentieren Einheiten potenziell paralleler Verarbeitung. Da das Ziel der folgenden Untersuchungen ist, die gegebenen Problemstellung durch Parallelisierung zu optimieren, sind an dieser Stelle lediglich Sparks von Interesse. Es sei jedoch angemerkt, dass ihre tatsächliche gleichzeitige Ausführung erreicht wird, indem die Laufzeitumgebung sie auf mehrere Haskell-Threads abbildet.

Die Berechnungseinheit eines Sparks enthält jeweils einen Thunk und wird durch die Funktion `par` initiiert. Der Ausdruck `x ‘par‘ y` führt dazu, dass für `x` ein Spark erstellt wird, bevor er `y` als Ergebnis zurückgibt. Welchen Wert die eventuelle Berechnung von `x` hat, spielt dabei keine Rolle. Die Definition sieht allein `y` in WHNF als Resultat vor. Das Erstellen eines Sparks ist eine verhältnismäßig einfache Operation, welche lediglich einen Zeiger auf den betreffenden Thunk dem lokalen *Spark Pool* (s.u.) hinzufügt (vgl. Philip W Trinder et al. 1996, S.3). Damit die so generierten Möglichkeiten paralleler Berechnung auch gleichzeitig mit anderen ausgeführt werden können, ist eine zusätzliche Notation für die Kontrolle der Berechnungsreihenfolge nötig. Diese wird durch die Funktion `pseq` bereitgestellt. Der Term `x ‘pseq‘ y` überführt `x` in WHNF und gibt anschließend `y` zurück. Im Unterschied zu `par` wird also von `pseq` impliziert, dass die Auswertung des ersten Arguments definitiv vor der des zweiten stattfindet. (Vgl. Marlow, Maier et al. 2010, S.2).

Beispiel: Der Ausdruck `x ‘par‘ y ‘pseq‘ (x*y)` kreiert einen Spark für den Term `x`, welcher daraufhin von einem anderen Prozessor evaluiert werden kann. Anschließend stellt `pseq` sicher, dass `y` ausgewertet wird, bevor die finale Berechnung von `(x * y)` erfolgt.

Haskells Laufzeitumgebung verwaltet pro Prozessorkern einen sogenannten Haskell Evaluation Context (HEC), welcher sich unter anderem aus den folgenden Komponenten zusammensetzt (vgl. Marlow, S. Peyton Jones und Singh 2009, S.3f):

- Eine Warteschlange für Nachrichten anderer HECs (*Message Queue*)
- Eine Warteschlange für Haskell-Threads, die bereit zur Ausführung sind (*Run Queue*)
- Eine Menge lokaler Sparks (*Spark Pool*)
- Eine Menge zur Verfügung stehender Betriebssystem-Threads (*Worker Pool*)
- Eine globale Menge von Haskell-Threads, die durch *Black Holes*[4] blockiert sind (*Black Hole Pool*)

Der Verarbeitungszyklus eines HEC gestaltet sich dabei wie folgt, geordnet nach der Priorität einer jeweiligen Aufgabe (vgl. Marlow, S. Peyton Jones und Singh 2009, S.4):

1. Bearbeite ein Element der Message Queue.
2. Führe einen Thread der Run Queue aus (im Round Robin Verfahren).
3. Falls ein beliebiger Spark Pool nicht leer ist, erstelle einen *Spark Thread* und führe diesen aus.
4. Überprüfe, ob einer der Threads des Black Hole Pools nicht mehr blockiert ist. Falls erfolgreich, führe diesen aus.

Sparks werden zwischen den einzelnen HECs geteilt, indem diese die Möglichkeit haben, sich freie Arbeit von anderen zu „stehlen”. Sobald ein HEC weder über Nachrichten, noch über ausführbare Threads verfügt, überprüft er sowohl seinen eigenen als auch die Spark Pools aller anderer HECs nach zu berechnenden Thunks und startet gegebenenfalls einen Spark Thread. Auch dessen Ausführung läuft nach einem periodischen Schema ab (vgl. Marlow, S. Peyton Jones und Singh 2009, S.5):

1. Falls die lokale Run Queue nicht leer ist, beende den Spark Thread.
2. Entnimm dem lokalen Pool einen Spark. Alternativ, hole ein Element eines anderen Spark Pools.
3. Falls kein Spark abgerufen werden konnte, beende den Spark Thread.
4. Evaluiere den Spark zu WHNF.

Es kann durchaus vorkommen, dass Thunks eines lokalen Spark Pools bereits von einem anderen Prozessor ausgewertet wurden. In diesem Fall wurde die betreffende Adresse im Hauptspeicher mit seinem Wert überschrieben. Solche Sparks, die bei einem erneuten Versuch der Evaluierung sofort das Ergebnis zurückgeben würden, werden als *fizzled* bezeichnet. Es sollten daher durch die Laufzeitumgebung alle fizzled Sparks entfernt werden (vgl. Marlow, Maier et al. 2010, S.4).

Wie eingangs erwähnt, übernimmt die Laufzeitumgebung die Aufgabe, während des Programmablaufs kreierte Sparks auf Haskell-Threads abzubilden. Diese werden in Betriebssystemthreads ausgeführt, um tatsächlich gleichzeitige Ausführung durch den Prozessor zu ermöglichen. Dabei kann im Normalfall stets von einem $N : 1$ Verhältnis ausgegangen werden: eine Vielzahl von Sparks wird in einem Haskell-Thread evaluiert, von

[4]Thunks, die momentan von einem HEC evaliert werden, sind für andere durch sogenannte *Black Holes* blockiert

denen wiederum mehrere einen Betriebssystemthread ausfüllen. Schon parallele Haskell-Programme überschaubarer Größe können schnell zehntausende von Sparks erstellen, sodass ein direktes Mapping auf Betriebssystemthreads keineswegs sinnvoll wäre.

Evaluationsstrategien

Um mit den Funktionalitäten von `par` und `pseq` einfacher umfangreiche, parallele Algorithmen konstruieren zu können, wurden von Philip W. Trinder et al. (1998) Evaluationsstrategien als zusätzliche Abstraktion vorgestellt. Sie entsprechen Funktionen höherer Ordnung, welche ausschließlich für ihren Effekt auf die Berechnungsweise ausgeführt werden. Dabei müssen die gleichzeitig auszuführenden Funktionen frei von Nebeneffekten sein, um eine deterministische Parallelisierung zu ermöglichen.

In einer überarbeiteten Form und enkapsuliert in der `Eval` Monade wird dieser Ansatz verwendet um sowohl die Ausführungsreihenfolge einer Berechnung als auch den Grad ihrer Evaluierung zu kontrollieren. Solche Strategien erlauben es, Parallelität in Algorithmen vollkommen losgelöst von der bestehenden Logik zu beschreiben und basierend auf einfachen Strategien, komplexere zusammenzusetzen. Die Implementierung von Marlow, Maier et al. (2010) definiert hierfür die vier folgenden Funktionen als Grundbausteine: `rpar`, welche für ihr Argument einen Spark kreiert, `rseq`, welche ihr Argument in WHNF überführt, `rdeepseq`, welche ihr Argument komplett auswertet und `r0`, welche überhaupt keine Evaluierung vornimmt. Der Grundgedanke, welcher allen von ihnen zu eigen ist, ist dass sie eine neue Version ihres Argumentes zurückgeben, welche die Semantik des entsprechenden Berechnungsgrades in sich trägt. So sind die Ergebnisse der beiden Terme `x` und `(rpar x)` zwar bezüglich ihrer Werte identisch, aber letzterer enthält zusätzlich die Information parallel berechenbar zu sein (vgl. Marlow, Maier et al. 2010, S.4).

Die Struktur einer Monade wurde gewählt, um ähnlich wie beim zuvor beschriebenen `IO` Beispiel eine Reihenfolge der Berechnungen beschreiben zu können. Die sogenannte `do` Notation, welche im Listing 2.2 zu sehen ist, entspricht einer einfachen Sequenzierung mehrerer Funktionen durch den `then` Operator (>>).

Dieser sehr abstrakte Ansatz hat den entscheidenden Vorteil, dass die Beschreibung paralleler Berechnungen weitestgehend unabhängig von der zugrundeliegenden Datenstruktur ist. So sind die Bausteine zur parallelen Auswertung eines Baumes die selben, wie die für eine Liste. Es muss lediglich initial eine Funktion definiert werden, die beschreibt, wie die Applikation von Strategien auf den jeweiligen Datentyp zu erfolgen hat. Nachfolgend ist dies am Beispiel der Funktion `evalList` dargestellt (entnommen aus Marlow, Maier et al. 2010), welche eine gegebene Strategie `s` auf alle Elemente einer Liste anwendet.

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

Listing 2.2: evalList – Applikation von Evaluationstrategien auf Listen

Wird zum Beispiel eine Kombination der Strategien `rpar` und `rdeepseq`[5] für den Parameter `s` eingesetzt, ist das Resultat eine Strategie, welche sämtliche Listenelemente parallel und vollständig, d.h. bis zur Normal Form (NF) evaluiert.

[5]Die Kombination zweier Strategien erfolgt mittels des `dot` Operators, definiert im Modul `Control.Parallel.Strategies`

2.3 Das MapReduce Programmiermodell

Ein wesentlicher Aspekt bei der Entwicklung paralleler Programme – neben einer Sprache, die Programmierung für Multiprozessor-Architekturen unterstützt – ist die Wahl des richtigen Algorithmus.

Das von Google entwickelte MapReduce-Framework (Dean und Ghemawat 2004) ist ein von funktionaler Programmierung inspiriertes Programmiermodell sowie eine Implementierung ebenjenes. Es war ursprünglich dafür konzipiert, große Datenmengen auf Clustern handelsüblicher Hardware (engl.: *commodity hardware*) zu verarbeiten. Um dies zu ermöglichen, übernimmt es die zuverlässige Verteilung der Daten über die Netzwerkinfrastruktur, steuert die Kommunikation zwischen den Clusterknoten und beinhaltet dedizierte Mechanismen zur Fehlerbehandlung, so dass der Ausfall einzelner Rechnern nicht die komplette Berechnung behindert. Allen voran bietet es aber eine leicht verständliche Möglichkeit, Algorithmen zu parallelisieren, indem es den Ablauf in drei definierte Phasen gliedert. Das grundlegende Berechnungsschema ist dabei so aufgebaut, dass es sich auf eine Vielzahl von Problemen anwenden lässt. Seine Autoren führen unter anderem parallele Implementierungen eines `sort`-Programms und des UNIX-Werkzeugs `grep` als Beispiele an (vgl. Dean und Ghemawat 2004).

Diese günstigen Eigenschaften führten dazu, dass MapReduce-Implementierungen inzwischen auch auf einzelnen Multiprozessorsystemen mit Shared Memory erfolgreich eingesetzt wurden (vgl. Ranger et al. 2007, Talbot, Yoo und Kozyrakis 2011). Der grundlegende Ablauf von MapReduce lässt sich in drei logische Schritte gliedern: Map, Group und Reduce. Wie eingangs erwähnt, entstammen die grundlegenden Konzepte der funktionalen Programmierung. Sowohl `map` als auch `reduce` – in Haskell `fold` genannt – sind Funktionen höherer Ordnung, d.h. „Funktionen, die als Parameter oder Resultate wieder Funktionen haben [...]" (Pepper und Hofstedt 2006, S.24). Sie stellen einen zentralen Bestandteil funktionaler Sprachen dar und sind oftmals Surrogat für Kontrollstrukturen, wie `for` oder `while` Schleifen.

Im Folgenden sollen beide in Haskell-Syntax erläutert werden. Die Ausführungen sind dabei stark an die Arbeit von Lämmel (2008) angelehnt.

Die in Haskells Standardbibliothek enthaltene `map` Funktion kann wie folgt definiert werden:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

Bsp.: map (*3) [3,4,5] = [9,12,15]
```

Listing 2.3: Mögliche Definition der Funktion `map`

Dabei wird die Funktion `f` auf jedes Element der zu bearbeitenden Liste `xs` angewendet. Die Resultate der Einzelberechnungen bilden anschließend wieder eine Liste. Für den trivialen Fall der leeren Liste `[]`, ist das Ergebnis ebenfalls leer. Die Symbole `a` und `b` in der Funktionsdefinition repräsentieren polymorphe Typen. Eine Ergebnisliste ist also nicht notwendigerweise vom gleichen Typ, wie die als Funktionsparameter übergebene.

Die Funktion `fold` wird für das Traversieren und gleichzeitige Reduzieren einer Liste benutzt, d.h. sie bildet die Elemente einer eindimensionalen Liste auf einen atomaren Ergebniswert ab:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f y []     = y
foldl f y (x:xs) = y foldl f (f y x) xs

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

Listing 2.4: Mögliche Definition der Funktion `foldl`

Im abgebildeten Beispiel wird die Liste beginnend von links durchlaufen, daher der Funktionsname `foldl`. Eine konträre Operation, `foldr`, ist ebenfalls in Haskell definiert. Der erste Parameter des Fold ist eine binäre Funktion, oder Kombinator. Sie nimmt zwei Werte der Typen `a` und `b` entgegen und gibt einen Wert des Typ `a` zurück. Das zweite Argument des Fold, ebenfalls vom Typ `a`, stellt den sogenannten Akkumulator dar. Er enthält die Zwischenergebnisse der Berechnung und ist gleichzeitig der erste Parameter für die binäre Funktion bei Anwendung auf die einzelnen Listenelemente. Ist die übergebene Liste leer, so ist der Akkumulator das Standard-Ergebnis. Das dritte und letzte Argument ist die Liste, über die der Fold ausgeführt wird. Die Methodik wird verständlicher, wenn obiges Beispiel in einzelne Verarbeitungsschritte zerlegt wird:

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

Listing 2.5: Logischer Ablauf der `foldl` Funktion

In jeder Phase eines MapReduce müssen die zu behandelnden Daten die Struktur von Schlüssel-Wert-Paaren (engl.: *Key-Value Pairs (KVPs)*) haben. So beschreiben Dean und Ghemawat (2004, S.2) die Funktionstypen im Laufe der Berechnung wie folgt:

$$MAP\ (\text{k1,v1}) \rightarrow \text{list(k2,v2)}$$

$$REDUCE\ (\text{k2,list(v2)}) \rightarrow \text{list(v2)}$$

Wie Lämmel (2008, S.5f) anmerkt, spiegelt die Verwendung dieser beiden Operationen in Googles MapReduce Framework aber nicht exakt die Funktionsweise der zuvor präsentierten Definitionen für `map` und `fold` wider, wie sie in funktionalen Sprachen zu finden sind. Tatsächlich stellt Googles *MAP* lediglich ein Argument für eine `map` Funktion dar, welche aus einem gegebenen KVP, ein Zwischenergebnis in Form einer Liste produziert, die wiederum mehrere KVPs enthält.

Die *REDUCE* Operation erfüllt die gleiche Funktion, wie die eines `fold`, indem sie die im ersten Schritt von *MAP* generierten Listen auf einzelne Werte reduziert. Um dies allerdings ebenfalls parallel geschehen zu lassen, ist sie selbst wiederum Eingangsparameter für eine `map` Funktion.

Der Bedarf nach einer *GROUP* Operation erwächst aus der Tatsache, dass die Elemente der durch *MAP* generierten Listen nicht notwendigerweise den selben Schlüssel haben. Damit die abschließende *REDUCE*-Operation aber parallel erfolgen kann, ist es essenziell, dass ein Endergebnis jeweils nur von den Werten einer Gruppe abhängt. In Abbildung 2.2 ist der logische Aufbau von MapReduce noch einmal bildlich dargestellt.

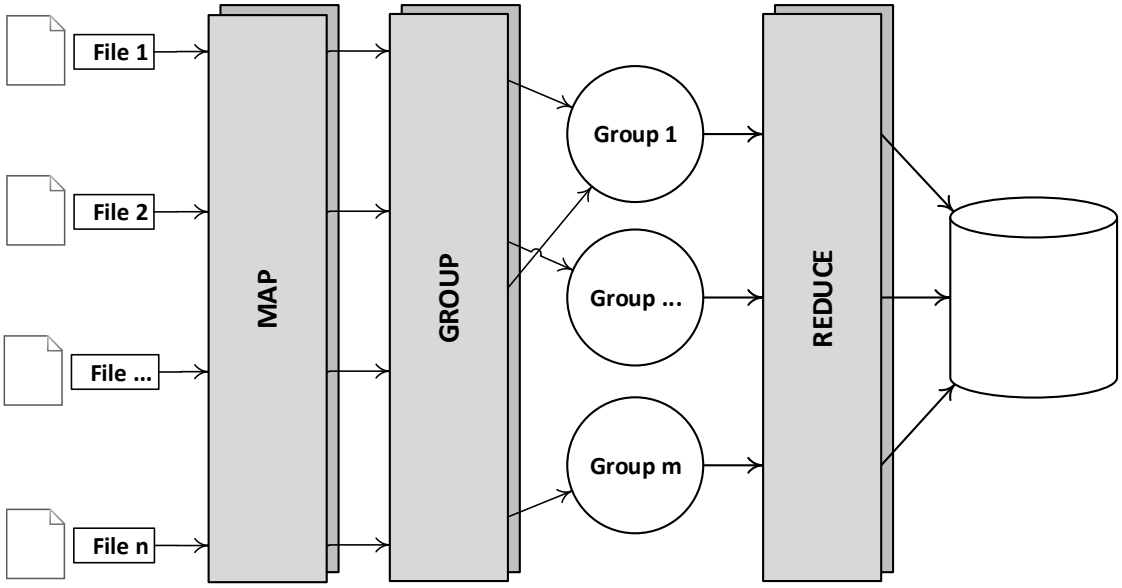

Abbildung 2.2: Logischer Aufbau des MapReduce Programmiermodells

Mit diesen Kenntnissen kann der komplette Ablauf eines parallelen MapReduce wie folgt beschrieben werden: Jedes Zwischenergebnis der initialen Liste von KVPs kann von einem beliebigen Prozessor, separat von den anderen, berechnet werden. Eine solche Operation, bei der identische Funktionen parallel auf eine Menge von Eingangsdaten angewendet werden, ist eine Ausprägung von Daten-Parallelismus (vgl. Abschnitt 2.1). Dabei werden bei einer Prozessoranzahl p und einer Anzahl zu bearbeitender Dateien n, idealerweise n/p Elemente pro Prozessor verarbeitet. Resultat dieser *MAP* Phase ist eine zweidimensionale Liste der Länge n.

Anschließend werden die Elemente aller Teillisten in der *GROUP* Phase anhand ihrer Schlüssel gruppiert. Es ist nicht möglich diesen Arbeitsschritt vollkommen zu parallelisieren, denn das gleichzeitige Zusammenführen von Elementen in einer zwischen den Prozessoren geteilten Datenstrukturen entspricht einer Wettlaufsituation. Da es sich jedoch beim Gruppieren effektiv um ein Sortierungs-Problem handelt (vgl. Lämmel 2008, S.15), können vor einem Zusammenfügen zumindest die Eingangslisten unabhängig voneinander geordnet werden. Die *GROUP* Funktion bildet effektiv n Listen von KVPs auf m Listen mit KVPs des jeweils gleichen Schlüssel ab.

Letztendlich wird für jede der m gebildeten Gruppen eine *REDUCE*-Operation ausgeführt, welche unabhängig voneinander – und somit potentiell parallel – ein Aggregat berechnet.

2.4 Testkonzept und Testwerkzeuge

Auf Basis welcher Faktoren kann die Leistung paralleler Programme verglichen werden? Auch wenn diese Frage trivial erscheinen mag, so ist es besonders bei der hier zu untersuchenden Problemstellung essenziell sich eingehend damit zu beschäftigen.

Zum einen soll ein Vergleich zwischen zwei grundverschiedenen Programmiersprachen vorgenommen werden. C++ ist eine etablierte Sprache, die sowohl in der Forschung als auch im industriellen Bereich einen immensen Verbreitungsgrad genießt und in deren meistgenutzter Compiler, GCC, Unmengen an Optimierungen stecken. Haskell wiederum erhält erst in den vergangenen Jahren die Aufmerksamkeit einer breiten Öffentlichkeit

(vgl. S. Peyton Jones 2011, S.12). Selbst wenn der GHC als ein hochentwickelter Compiler gilt, ist dennoch nicht abzusehen, wieviel leistungsfähiger die erstellten Programme sein könnten, wäre Haskell ähnlich populär wie C++. Als eine Sprache, die auf einer sehr hohen Abstraktionsebene operiert, erlaubt Haskell außerdem keine direkte Kontrolle über die Speicherverwaltung. Dies wiederum erfordert das Vorhandensein eines *Garbage Collectors (GC)*, der periodisch nicht länger benötigten Speicher freigibt. Der Haskell GC arbeitet seinerseits auch parallel und verteilt über alle Prozessoren, daher entsteht während der Laufzeit des Programms für jede der CPUs ein nicht vorhersagbarer[6] Mehraufwand.

Zum anderen ist das in diesem Rahmen untersuchte Szenario ein sehr spezielles. Die erzielten Ergebnisse spiegeln nicht notwendigerweise die Leistung beider Sprachen bei der Bearbeitung anderen Problemstellungen oder gar unter abweichenden technischen Gegebenheiten wider. Dennoch soll eine möglichst reproduzierbare Gegenüberstellung erzielt werden, welche sich auf die im Folgenden beschriebenen Leistungsmerkmale stützt.

Testkriterien

Bei der Bewertung paralleler Software können zwei grundlegende Arten von Leistungskriterien unterschieden werden. Einerseits weisen diese Programme Merkmale auf, die vollkommen losgelöst von der Zahl verwendeter Prozessoren oder erstellter Threads erfasst werden können, die also auch für nicht-parallele Programme zutreffen. Ebenjene Kriterien werden im Folgenden „allgemeine" Leistungsmerkmale genannt.

Zusätzlich kann ihre Performance aber mittels spezieller Kriterien bewertet werden, bei denen die Zahl der Prozessoren und somit der Grad der Parallelisierung sehr wohl eine Rolle spielt.

Allgemeine Leistungsmerkmale

Antwortzeit: Ein intuitives Maß für die Performance eines jeden Programms ist seine Laufzeit. Dabei gilt es jedoch zwischen verschiedenen Arten der Zeitmessung zu differenzieren. Die Antwortzeit (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 – daher der englische Begriff. Diese Wert setzt sich aus den folgenden drei Komponenten zusammen (vgl. Rauber 2012, S.166):

- Die *Benutzer-CPU-Zeit* (engl.: *User Time*) verbringt der Prozessor mit der tatsächlichen Ausführung des Programms.
- Die *System-CPU-Zeit* (engl.: *System Time*) wird für durch den Programmcode verursachte Betriebssystemaufrufe benötigt (z.B. `read`, `write`, `fork`).
- Ein Prozess befindet sich in der *Wartezeit*, während er blockiert ist und somit andere Prozesse die CPU belegen oder auf Ein- und Ausgabeoperationen gewartet wird.

Je nach der aktuellen Auslastung eines jeweiligen Systems kann die Wartezeit stark variieren und auch die System-CPU-Zeit ist stets abhängig von der Implementierung des Betriebssystems. Daher sollten diese beiden Größen keinen Einfluss auf eine Bewertung der sequenziellen Programmleistung nehmen, argumentiert Rauber (2012, S.166).

Werden allerdings parallele Programme betrachtet, so würde ein Ausschluss der Wartezeit auch die Dauer außer Acht lassen, in dem Threads aufgrund von Synchronisation oder

[6]Das De-Allozieren von nicht mehr benötigten Objekten im Hauptspeicher durch den Garbage Collector erfolgt durch verschiedene Strategien. Existieren z.B. keine Referenzen zu einem gewissen Objekt, so kann dieses sicher entfernt werden. Eine detaillierte Betrachtung, nach welchem Muster Haskells Laufzeitumgebung Hauptspeicherbereiche freigibt würde den Rahmen dieser Arbeit übersteigen.

blockierenden Zugriffen auf Variablen des gemeinsamen Speichers keine Arbeit verrichten. Umgekehrt ist die CPU-Zeit in einem solchen Szenario weit weniger aussagekräftig, da sie meist nur auf Prozess-Ebene bestimmt werden kann und lediglich eine Bildung des Mittelwerts über alle Threads möglich ist – dabei ist eine ideale Verteilung der Arbeitslast zwischen den Threads in einem realen Szenario die Ausnahme. Da also diese Wartezeiten ein zentraler Bestandteil paralleler und nebenläufiger Programmen sind und einen nicht zu vernachlässigenden Teil zur kompletten Laufzeit beitragen können, wird für die nachfolgenden Untersuchungen die komplette Antwortzeit T_A herangezogen.

Damit bei einer Erfassung des Wertes möglichst nur das tatsächlich zu messende Programm Rechenressourcen belegt, ist die Anzahl der im Hintergrund laufenden Prozesse auf ein Minimum zu begrenzen. Im Abschnitt 4.1 wird die Messung aller Leistungsdaten und der verwendete Versuchsaufbau detailliert beschrieben.

Cache Hit-Rate: Speicherzugriffe auf den privaten Cache-Speicher eines Prozessors sind bedeutend schneller als solche auf den globalen Arbeitsspeicher. Ihre Funktionsweise basiert auf dem Lokalitätsprinzip. Dieses besagt, dass ein aktuell berechneter Wert oder eine kürzlich verwendete Instruktion mit hoher Wahrscheinlichkeit in naher Zukunft erneut genutzt wird. Daher wird jedes Berechnungsergebnis im Cache des Prozessors abgelegt, bis es von einem anderen daraus verdrängt wird. Auch wenn moderne Prozessoren zwischen verschiedenen Ebenen des Cache-Speichers unterscheiden, von denen manche auch zwischen allen Prozessoren geteilt werden[7], soll der Cache in den folgenden Untersuchungen als eine Einheit betrachtet werden. Eine sogenannter *Read Hit* tritt auf, wenn der Prozessor seinen Cache erfolgreich nach einem Datum durchsucht. Ein *Read Miss* hingegen entspricht einer fehlgeschlagenen Lese-Operation - die benötigten Daten befinden sich aktuell nicht im Cache-Speicher und müssen aus dem Hauptspeicher nachgeladen werden. Der prozentuale Anteil der Read Hits von der Gesamtzahl aller lesenden Cache-Zugriffe wird als *Hit-Rate* bezeichnet:

$$\text{Hit-Rate} = \frac{\text{Read Hits}}{\text{Read Hits} + \text{Read Misses}}[\%] \tag{2.1}$$

Eine möglichst hohe Hit-Rate ist deshalb erstrebenswert, weil sie widerspiegelt, in welchem Maße der Prozessor bei der Ausführung eines Programms, Mehrwert aus seinem Cache generieren kann.

Durchschnittlich genutzter Arbeitsspeicher: Im Zuge dieser Arbeit wird eine relativ simple Implementierung des MapReduce Modells auf einem Shared Memory System untersucht. Dabei werden die zu verarbeitenden Daten während der Laufzeit nicht persistent gespeichert sondern befinden sich ausschließlich im Hauptspeicher des Systems. Der belegte Arbeitsspeicher ist daher eine kritische Größe, weil er die Menge der gleichzeitig zu verarbeitenden Daten nach oben begrenzt. Da die Speicherbelegung während der Laufzeit eines Programms variiert, wird ein Durchschnittswert für den Vergleich herangezogen.

Um die Speicherbelegung während der Verarbeitung möglichst gering zu halten, sollte es vermieden werden unnötige Kopien der Daten zu erstellen. Nicht mehr benötigte Daten sollten außerdem möglichst schnell gelöscht werden. Wie zuvor bereits angesprochen, kann dies jedoch nur in C++ auf manuelle Weise geschehen.

[7] Häufig sind nur die ersten zwei Ebenen der Cache-Hierarchie einem Prozessorkern zu eigen, von denen der Level-1-Cache nochmals in zwei Sektoren für Instruktionen und Daten geteilt ist. Eine dritte Ebene des Cache-Speichers wird, falls vorhanden, meist zwischen mehreren CPUs geteilt.

Durchsatz: Um einen Zusammenhang mit der Problemgröße – also der Anzahl verarbeiteter Records N_{rec} – herzustellen, soll außerdem der Durchsatz $D(N)$ bestimmt werden, den es zu maximieren gilt. Generell lässt sich dieser als die Anzahl der Records pro Zeiteinheit verstehen und kann nach folgender Formel berechnet werden:

$$D(N) = \frac{N_{rec}}{T_A}[Records/s] \tag{2.2}$$

Leistungsmerkmale paralleler Programme

Die bisher betrachteten Performancefaktoren sind generell auf sämtliche Programme anwendbar, egal ob die Ausführung auf paralleler oder rein sequenzieller Hardware erfolgt. Nachfolgende Eigenschaften sind hingegen stets auf die Anzahl der eingesetzten Prozessoren P bezogen.

Speedup: Der Speedup $S(P)$ beschreibt, wie sehr ein paralleles Programm von einer steigenden Zahl zur Verfügung stehender CPUs profitiert. Absoluter Speedup $S_{abs}(P)$ (engl.: *absolute speedup*) ist dabei das Verhältnis zwischen der Antwortzeit des optimalen sequenziellen Programms für ein gegebenes Problem und einer parallelen Version des selbigen, die auf einer Zahl von Prozessoren P ausgeführt wird (vgl. Sahni und Thanvantri 1995, S.9):

$$S_{abs}(P) = \frac{T_{A_{opt}}(1)}{T_A(P)} \tag{2.3}$$

Da der Nachweis, das Programm sei optimal nur schwer zu erbringen, bzw. die optimale Lösung nicht immer bekannt ist, wird alternativ der relative Speedup $S_{rel}(P)$ angegeben. Bei diesem steht im Zähler die Antwortzeit des parallelen Programms, ausgeführt auf einem Prozessor (vgl. Sahni und Thanvantri 1995, S.13). Diese Definition wird auch für sämtliche hier angestellten Betrachtungen genutzt:

$$S_{rel}(P) = \frac{T_A(1)}{T_A(P)} \tag{2.4}$$

Skalierbarkeit: Das Verhalten eines parallelen Programms bei Ausführung auf einer unterschiedlichen Zahl von Prozessoren wird durch die sogenannte Skalierbarkeit beschrieben. Diese soll nachfolgend in Anlehnung an die Speedup-Definition von Rauber (2012, S.178) verstanden werden. Resultiert eine Verdoppelung der Zahl eingesetzter Recheneinheiten in einer Halbierung der Laufzeit, so wird von linearer Skalierung gesprochen. Ist die Entwicklung der Laufzeit gar noch besser, so skaliert der Algorithmus superlinear. Jedoch werden diese beiden Fälle in der Praxis aber nur selten erreicht. Viel häufiger lässt sich eine sublineare Skalierung erkennen, welche mit zunehmender Anzahl von Prozessoren stagniert und aufgrund des erhöhten Verwaltungsaufwandes für erstellte Threads unter Umständen sogar wieder abnimmt. (Vgl. Rauber 2012, S.178f)

Die Ursache dafür liegt unter anderem im Aufbau des verwendeten Algorithmus. So stellte Gene M. Amdahl (1967) bereits fest, dass der Speedup eines parallelen Algorithmus durch den Anteil sequenzieller Programmfragmente nach oben hin begrenzt ist. Demnach setzt sich die auf den Wert 1 normierte Laufzeit T aus einem sequenziellen Teil σ und dem restlichen, parallelen Teil zusammen. Während die parallelen Abschnitte des Algorithmus von mehreren Prozessoren P profitieren, bleibt die für σ benötigte Zeit unverändert:

$$T = 1 = \sigma + (1 - \sigma) \qquad\qquad T(P) = \sigma + \frac{(1 - \sigma)}{P} \tag{2.5}$$

Diese Formel wird in den Nenner der Gleichung 2.4 eingesetzt und $T_A(1) = 1$ angenommen. Eine anschließende Grenzwertbetrachtung mit $P \to \infty$ lässt erkennen, dass der maximal erreichbare Speedup eines Programm mit einem sequenziellen Anteil σ durch $1/\sigma$ begrenzt ist:

$$\lim_{P\to\infty} S(P) = \frac{1}{\sigma + \frac{(1-\sigma)}{P}} \tag{2.6}$$

$$S(P) \leq S(P)_{max} = \frac{1}{\sigma} \tag{2.7}$$

Amdahls Gesetz ermöglicht somit eine theoretische Abschätzung des maximalen Leistungsgewinns auf einer bestimmten Anzahl von Prozessoren P. Ob dieser Wert tatsächlich erreicht wird, kann meist nur die Implementierung selbst zeigen. Der theoretisch mögliche Speedup nach Amdahl soll aber für beide zu erstellende Implementierungen trotzdem berechnet werden, um eine Wertung vorzunehmen, inwieweit die potentielle Leistungssteigerung tatsächlich erreicht werden kann. Basis für diese Berechnungen, zu finden im Abschnitt 4.2, sind die jeweiligen sequenziellen Ausführungszeiten der *MAP*, *GROUP* und *REDUCE* Funktionen.

Effizienz: Eine etwas andere Aussage als der Speedup liefert die Effizienz $E(P)$ eines Programms. Diese ist definiert als das Verhältnis aus Speedup $S_{rel}(P)$ und der Zahl genutzter Prozessoren P:

$$E(P) = \frac{S_{rel}(P)}{P} \tag{2.8}$$

Im Fall eines linearen Speedups, gilt für die Effizienz $E(P) = 1$. Liegt ein sublinearer Speedup vor, ist das Resultat kleiner als dieser Wert.

Kosten: Die Kosten $C(P)$ eines parallen Programms ist die gesamte Arbeit, die von allen eingesetzten Prozessoren zur Ausführung des Algorithmus aufgebracht wurde. Angelehnt an die Formel von Rauber (2012, S.176) soll diese für Anzahl der genutzten Prozessoren P unter Zuhilfenahme der Antwortzeit T_A wie folgt berechnet werden:

$$C(P) = T_A \cdot P \tag{2.9}$$

Paralleler Overhead: Der Parallele Overhead $O(P)$ eines Programms beschreibt den Mehraufwand, welcher aus der Erstellung, Koordination und Beendigung Thread-basierter Parallelität resultiert. Des Weiteren umfasst diese Größe die unproduktive Zeit, welche einzelne Prozessoren auf andere warten, weil die Arbeitslast nicht ideal auf alle Recheneinheiten verteilt wurde. Dieser, möglichst geringe, Overhead soll als prozentualer Wert mithilfe der Kosten aus Gleichung 2.9 bestimmt werden:

$$O(P) = \frac{C(P) - T_A(1)}{C(P)} \cdot 100[\%] \tag{2.10}$$

Der Grundgedanke dieser Berechnung ist folgender: Ein sequenzielles Programm benötigt für die vollumfängliche Bearbeitung einer gegebenen Problemgröße eine gewisse Zeit $T_A(1)$. Es ist der schnellste vorliegende Algorithmus (keineswegs der schnellstmögliche) und dient daher als Referenzgröße – alles was darüber hinausgeht, entspricht unnötiger Arbeit bzw. Leerlauf des Prozessors durch blockierte Threads. Das Ergebnis einer Differenzbildung der von allen Prozessoren ausgeführten Arbeit mit $T_A(1)$ ist also der zeitliche Mehraufwand, den das Programm aufgrund der Parallelisierung erbringt. Wird diese Differenz ins Verhältnis zu den Kosten gesetzt, ergibt sich der prozentuale Anteil

des Mehraufwandes an der gesamt verrichteten Arbeit. Chi et al. (1997, S.18) definiert den Overhead ähnlich, verzichtet allerdings auf die Division durch die parallelen Kosten.

Testwerkzeuge

test_generator.pl: Für die automatische Generierung von Testdaten wird ein Perl-Programm (`test/test_generator.pl`) bereitgestellt, welches im elektronischen Anhang zu finden ist. Anhand verschiedener, zur Laufzeit bereitgestellter Parameter erstellt es Textdateien, deren Inhalt Records der im Anhang A beschriebenen Form sind. Die möglichen Feldnamen sowie ihre Belegungsvorschrift anhand regulärer Ausdrücke werden im Modul `test/Record.pm` definiert.

Bei Ausführung des Programms ist es möglich die Anzahl verschiedener Belegungen der Schlüsselfelder zu bestimmen, auf Basis derer eine Aggregation stattfinden soll. Diese Parameter haben daher direkten Einfluss auf die resultierenden Anzahl der Gruppen und möglichen parallelen Operationen in der Reduce-Phase des Algorithmus. Die vordefinierten Schlüsselfelder sind: *AggregationsDatum*, *Eventklasse*, *Eventsource*, *Plattform* und *Tarifzone*. Die Menge distinkter Aggregationsschlüssel enthält alle Permutationen dieser fünf Felder. Darüber hinaus kann die Anzahl der zu erstellenden Records und Dateien spezifiziert werden, was Auswirkungen auf die Geschwindigkeit beim Einlesen der Dateien hat.

Nachfolgend ist eine kurze Übersicht über alle Kommadozeilenparamter des Testgenerators dargestellt:

- `r`: Anzahl Records
- `f`: Anzahl Dateien
- `a`: Anzahl AggregationsDatum
- `c`: Anzahl Eventklasse
- `s`: Anzahl Evensources
- `z`: Anzahl Tarifzonen
- `p`: Anzahl Plattformen
- `d`: Maximale Dauer
- `v`: Maximales Volumen
- `o`: Ausgabepfad

Beispiel: Der folgende Programmaufruf erstellt 100 Records in 10 Dateien im Ordner `./files`. Die möglichen Schlüssel, 120 an der Zahl, resultieren aus einem Aggregations-Datum, zwei Eventklassen, drei Eventsources, vier Tarifzonen und fünf Plattformen. Die Maximalwerte für die Aggregationsfelder Dauer und Volumen sind 80 und 20:

```
perl test_generator.pl -r100 -f10 -o files -a1 -c2 -s3 -z4 -p5 -d80 -v20
```

perf: Ein Großteil der zuvor beschriebenen Bewertungskriterien können unter Linux durch die Analyse- und Profiling-Software `perf` erfasst werden. Diese ist in aktuellen Kernel-Versionen ab 2.6.31 integriert. Das Programm erlaubt die Zählung sogenannter Hard- und Software-Events während der Laufzeit. Im ersten Fall findet die Messung durch den Kernel statt, im zweiten durch die in Hardware realisierte Performance Measurement Unit (PMU) des Prozessors (vgl. Stephane Eranian 2015, 1.2 Events). So wird eine Vielzahl der im vorangegangenen Abschnitt genannten Testkriterien vom Befehl `perf stat` abgedeckt, dazu zählen:

- Antwortzeit (elapsed time / wallclock time)
- Anzahl Cache-Lookups (cache-references)
- Anzahl Cache-Misses (cache-misses)

Diese Werte erlauben die Berechnung des relativen Speedups, der Effizienz, der Cache Hit-Rate, des Durchsatzes, der Kosten und des parallelen Overheads. Zusätzlich erlaubt das Programm die wiederholte Messung eines Programms und die automatische Berechnung der Standardabweichung gemessener Werte.

memc.sh: Um einen Wert für die Arbeitsspeicherbelegung während der Laufzeit eines Programms zu erfassen, wird ein Shell-Skript zur Verfügung gestellt, welches im elektronischen Anhang zu finden ist (`test/memc.sh`). Es macht sich das Werkzeug `smem` zu Nutze um periodisch die sogenannte Resident Set Size (RSS) der Programme zu messen. Diese entspricht dem belegten Bereich im Hauptspeicher der Maschine zu einem gegebenen Zeitpunkt. Eventuell auf die Swap-Partition ausgelagerte Teile des Programms fließen nicht in den Wert ein[8].

Der Aufruf des Shell-Skripts erfolgt mit drei Parametern: dem Pfad zum zu messenden MapReduce-Programm, dessen benötigten Kommandozeilenparametern und dem Intervall zwischen den Messungen in Sekunden. Aus sämtlichen erfassten Werten für die RSS wird im Folgenden ein Durchschnittswert ermittelt – je kleiner dieser ist, desto besser.

Testdurchführung

Die durchzuführenden Testfälle für die Bewertung beider Programme wurden mithilfe einer Entscheidungstabelle identifiziert, welche in Tabelle 2.1 schematisch dargestellt ist. Dabei sind im oberen Teil relevante Parameter für die Bedingungen eines Testfalls dargestellt, welche direkt Einfluss auf die erzielte Leistung haben.

Testbedingungen	**Wert**
Programmiersprache	Haskell/C++
Verfügbare Threads	1/2/3/4/5/6/7/8
Compiler-Optimierungen	`-O0` / `-O1`/ `-O2`
Anzahl Dateien	100/250/500/750 1.000/2.500/5.000/7.500 10.000
Records pro Datei	10/100
Testkriterium	
Relativer Speedup	-
Effizienz	-
Paralleler Overhead [%]	-
Kosten [s]	-
Durchsatz [Records/s]	-
Durchschnittliche RSS [MiB]	-
Cache Hit-Rate [%]	-

Tabelle 2.1: Schematischer Aufbau der relevanten Testfälle

Der Wertebereich der genutzten Threads wurde auf einen maximalen Wert von acht festgelegt, da die verwendete CPU bis zu acht Kontrollflüsse „quasi-gleichzeitig" bearbeiten

[8]Muss ein Rechner mehr Arbeitsspeicher allozieren, als momentan im RAM verfügbar ist, so lagert er inaktive Teile auf eine sogenannte Swap-Partition aus. Diese befindet sich üblicherweise auf der Festplatte des Systems, daher ist der Zugriff darauf deutlich langsamer.

kann (vgl. Abschnitt 4.1). Eine Messung der parallelen Programme bei Ausführung mit nur einem Thread ist für die Berechnung des relativen Speedups und des parallelen Overheads nötig, wie im vorangegangenen Abschnitt beschrieben.

Die drei verschiedenen Kompilierungsoptionen, welche den Optimierungsgrad des Compilers steuern, haben sowohl beim GCC als auch beim GHC etwa die gleichen Bedeutungen: keine Optimierung (`-O0`), einige Optimierungen (`-O1`) und möglichst viele Optimierungen[9](`-O2`). Auch wenn für beide Programme wesentlich mehr Möglichkeiten zur Steuerung der Programmgenerierung durch den Compiler existieren, wurden diese grundlegenden Optionen ausgewählt um im begrenzten Rahmen dieser Arbeit einen Überblick über den Einfluss unterschiedlicher Optimierungsgrade zu erhalten. Für eine detaillierte Beschreibung der dabei durchgeführten Umwandlungen des Quellcodes wird auf die Benutzerhandbücher der GCC (R. M. Stallman und GCC Developer Community 2016) und des Haskell Compilers (GHC Team 2016) verwiesen.

Verschiedene Werte für die Anzahl der Records pro Datei werden betrachtet um die Auswirkungen der Dateigröße auf die Leistung der Programme zu untersuchen. Beide Werte wurden dabei in der Größenordnung gewählt, wie sie auch im realen Betrieb der zu optimierenden Software auftreten können. Gemeinsam mit der Anzahl der Dateien bestimmt dieser Wert die Problemgröße des MapReduce-Algorithmus.

Im unteren Teil der Tabelle sind sämtliche relevanten Bewertungskriterien aus dem vorherigen Abschnitt aufgeführt. Ausgenommen davon sind die Antwortzeit, welche ausschließlich zur Berechnung anderer Leistungsmerkmale genutzt wird, und der theoretisch mögliche Speedup. Die explizite Bewertung der Antwortzeit würde im Hinblick auf sehr ähnliche Kriterien, wie den Kosten $C(P)$ und dem Durchsatz $D(N)$, keine neuen Aussagen über die Performance der Programme zulassen. Der theoretische Speedup nach Amdahls Gesetz wiederum soll lediglich auf Basis der sequenziellen Programmversionen berechnet werden um zu zeigen, dass beide Programme ein hinreichendes Potenzial zur Parallelisierung aufweisen.

Da jegliche Kombination der vorgestellten Parameter getestet werden soll, ergeben sich aus der kompletten Tabelle insgesamt 864 Testfälle. Dank automatisierter Erstellung der Testdaten und einer Möglichkeit zur Steuerung der genutzten Threadanzahl zur Laufzeit, sind diese ohne Probleme abzudecken. Lediglich für die unterschiedlichen Optimierungsgrade müssen beide Programme gesondert kompiliert werden. Da eine tabellarische Auswertung der einzelnen Ergebnisse den Umfang des Dokuments übersteigen würde, sind alle erfassten Testergebnisse im elektronischen Anhang hinterlegt.

Wie im Abschnitt 1.4 beschrieben, sollen auf Basis dieser Kriterien in einem ersten Schritt gesonderte Aussagen über die erstellten parallelen Programme getroffen werden. Für eine anschließende Gesamtbewertung der Haskell und C++ Implementierungen bedarf es hingegen einer Normalisierung der Messwerte. Diese setzt sich für jedes Kriterium aus den folgenden Rechenschritten zusammen:

1. Dividiere den jeweils schlechteren Wert mit dem besseren. Lege den besseren Wert anschließend auf 1,0 fest.

2. Falls für das Kriterium gilt „geringer = besser", bilde den Kehrwert des schlechteren Wertes aus Schritt 1.

[9]Im Handbuch des GHC ist die Rede von allen Optimierungen, die sich nicht negativ auf Laufzeit oder Speichernutzung auswirken könnten (vgl. GHC Team 2016, S.103).

Ergebnis dieser Normalisierung ist ein Wert von 1,0 für die bessere der beiden Implementierungen hinsichtlich eines spezifischen Kriteriums. Das zweite Programm erreicht hingegen einen Wert $\leq$ 1,0, welcher seine Leistung im Verhältnis widerspiegelt.

Diese so gebildeten Werte werden anschließend für beide Programmiersprachen aufsummiert. Aus den sieben betrachteten Performancekriterien der Tabelle 2.1 ergibt sich ein erreichbarer Maximalwert von $7 \cdot 1,0 = 7,0$. Auch diese Summen werden noch einmal ins Verhältnis zueinander gesetzt, so dass die Leistungen beider Implementierungen in Form einzelner prozentualer Werte gegenübergestellt werden können.

Kapitel 3

Praktische Umsetzung

Im folgenden Kapitel werden die Implementierungen der Programme in den beiden Programmiersprachen, Haskell und C++, beschrieben. Dabei werden pro Funktionsblock jeweils verschiedene Möglichkeiten der Realisierung gegenübergestellt um eine fundierte Entscheidung treffen zu können.

3.1 Beschreibung des Szenarios als MapReduce

Im Folgenden wird die gegebene Problemstellung im Kontext des MapReduce Programmiermodells beschrieben, ohne dabei bereits auf spezifische Merkmale beider Sprachen einzugehen.

Zunächst gilt es die zu aggregierenden Daten einzulesen und in interne Datenstrukturen des Programms umzuwandeln. Dieser Vorgang wird auch als „Parsing" bezeichnet. Beim Parsen mehrerer Dateien bestehen in diesem Szenario keinerlei Datenabhängigkeiten. Jede Datei enthält eine bestimmte Anzahl abgeschlossener Records, mit allen Informationen, die für ihre Aggregation benötigt werden. Dieser Schritt eignet sich daher ideal für die *MAP* Phase und kann ohne Komplikationen durch eventuelle Wettlaufsituationen parallelisiert werden. Dabei ist zum einen das Vorhandensein aller definierten Felder in den jeweiligen Datensätzen zu prüfen. Zum anderen soll sichergestellt werden, dass die Belegung der einzelnen Attribute laut Recorddefinition wohlgeformt ist, d.h. vorgegebene Wertebereiche eingehalten werden und die Pflichtfelder mit Werten gefüllt sind. Eine Beschreibung dieser Implementierung, inklusive eines Vergleichs dafür in Frage kommender Parser für Haskell und C++ ist in den Abschnitten 3.2.1 respektive 3.3.1 zu finden.

Ergebnis der *MAP* Phase ist laut der Definition des Algorithmus eine Menge von Zwischenergebnissen in Form von Schlüssel-Wert Paaren. Eine solche Abbildung kann anhand der im Abschnitt 2.4 beschriebenen Schlüssel- und Aggregationsfelder erfolgen. Die Gesamtheit aller enthaltenen Schlüsselfelder ergibt den eindeutigen Schlüssel einer zu aggregierenden Gruppe von Records. Die referenzierten Werte wiederum entsprechen den Inhalten der Aggregationsfelder *Dauer* und *Volumen*. Da jede Datei mehrere Records des gleichen Schlüssels enthalten kann, muss ein Schlüssel auf potenziell mehrere Werte verweisen können, wie Abbildung 3.1 zeigt.

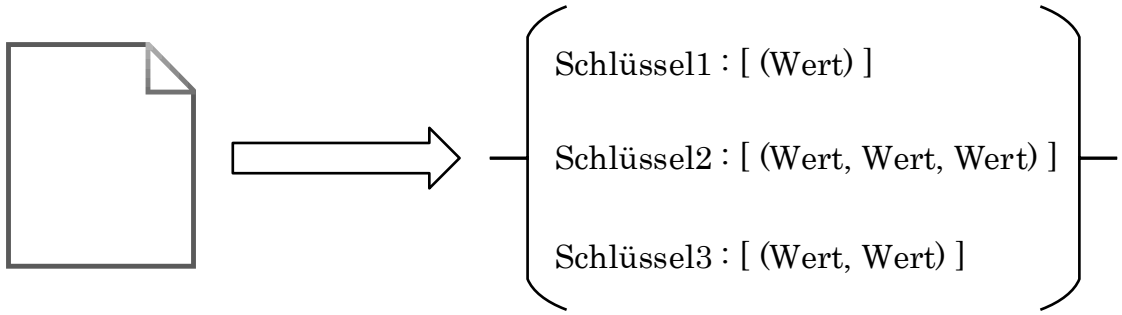

Abbildung 3.1: Logische Struktur der MAP Funktion

Solche Datenstrukturen, welche mehrere eindeutige Schlüssel-Wert Paare enthalten, sind in vielen Programmiersprachen als Maps oder Dictionaries bekannt. Auch in Haskell und C++ gibt es vordefinierte Datentypen, die für diesen Zweck genutzt werden können: in Form des `Data.Map.Map` Typs (Haskell) und des `std::map` Containers (C++). Zur besseren Unterscheidung von der gleichnamigen Operation des MapReduce-Algorithmus werden diese im Folgenden allgemein als *Key-Value Pair (KVP)* Strukturen bezeichnet. Mit diesen Kenntnissen kann der Typ *MAP* Funktion, wie folgt beschrieben werden:

```
Dateiname →
KVP( Schlüssel : LIST((Dauer-Wert, Volumen-Wert)) )
```

Für die Untersuchung paralleler Programme muss dabei von mehreren Eingangsparametern ausgegangen werden. Die eigentliche Parallelisierung erfolgt durch das im Abschnitt 2.3 beschriebene Funktional `map`, welches die gleichzeitige Anwendung obiger Funktion auf eine Liste von Dateien koordiniert. Das Resultat ist dann wiederum eine Liste von Schlüssel-Wert Strukturen.

Im Anschluss an die erfolgreiche Umwandlung der Records in interne Datenstrukturen, erfolgt deren Gruppierung. Konkret bedeutet dies, das Zusammenfassen aller Schlüssel-Wert Paare der einzelnen Dateien in einer globalen Datenstruktur, in der eine Gruppe durch einen Schlüssel repräsentiert wird. Wie bereits im Abschnitt 2.3 erwähnt, ist es nicht möglich diesen Arbeitsschritt vollständig zu parallelisieren. Dennoch soll der sequenzielle Anteil so gering wie möglich gehalten werden.

Da die verwendeten Schlüssel-Wert Datenstrukturen in Haskell und C++ intern durch Binärbäume realisiert werden, ist es möglich sie mit linearer Komplexität in eine sortierte Liste umzuwandeln. Dies wiederum erlaubt das effiziente Zusammenfügen, bei dem die grundlegende Idee eines parallelen Mergesort-Algorithmus genutzt wird: Werden zwei Listen der Längen n_1 und n_2 getrennt voneinander sortiert, so kann das anschließende Zusammenfügen mit einer Komplexität von $O(n_1 + n_2)$ erfolgen. Sowohl das Transformieren der Schlüssel-Wert Strukturen in sortierte Teillisten als auch große Teile des Zusammenfügens in eine Gesamtliste sind dabei unabhängig voneinander und können deterministisch parallelisiert werden. Wie Abbildung 3.2 anschaulich darstellt, ist lediglich das Zusammenführen der zwei zuletzt verbleibenden Teillisten von nur einem Prozessor gleichzeitig durchzuführen.

Aber nicht nur die effektiv genutzten CPUs, auch die maximal benötigten Vergleichsoperationen bei gleichgroßen, sortierten Teillisten unterscheidet beide Vorgehensweisen.

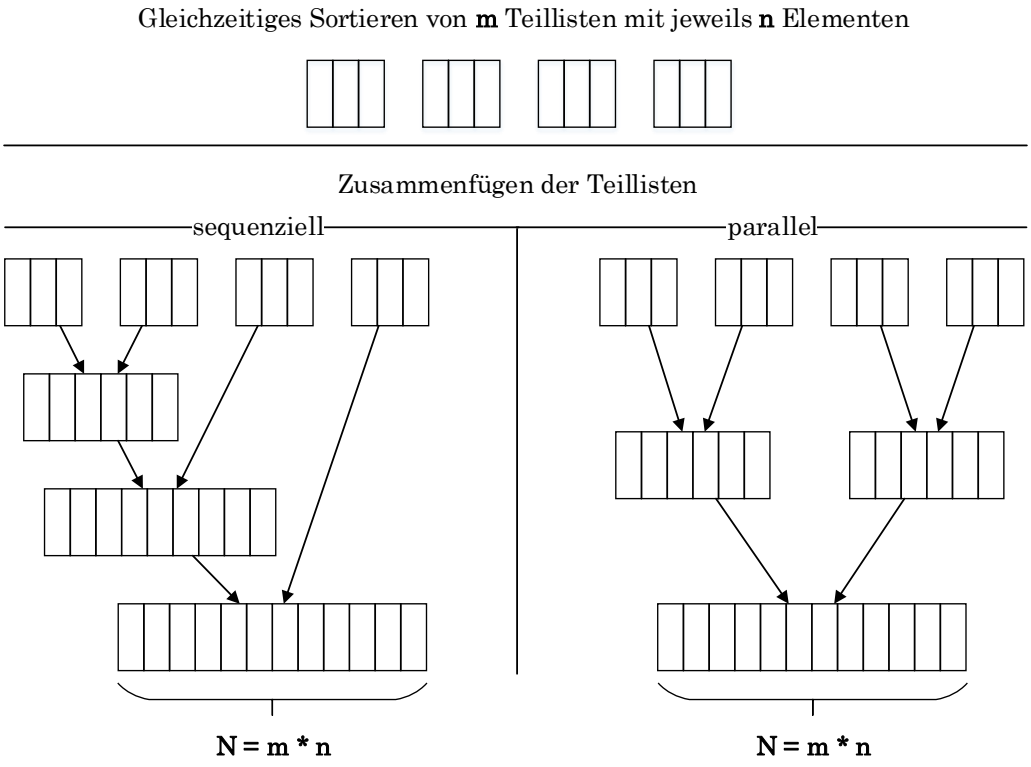

Abbildung 3.2: Sequenzielles und paralleles Zusammenfügen mehrerer Teillisten

Diese ist für den sequenziellen Fall (OPS_{seq}) und den parallelen Fall (OPS_{par}) in Abhängigkeit von der Zahl der Teillisten m und deren einheitlicher Länge n mittels der folgenden Formeln zu berechnen[10]:

$$OPS_{seq} = (m-1) \cdot \frac{(2n-1) \cdot (m \cdot n - 1)}{2} \tag{3.1}$$

$$OPS_{par} = \sum_{i=1}^{\log_2(m)} m \cdot n - \frac{m}{2i} \tag{3.2}$$

Paralleles Zusammenfügen der Teillisten hat somit zum einen den Vorteil, dass ab einer gewissen Anzahl von Listen weniger Vergleichsoperationen erforderlich sind als mit der sequenziellen Variante (vgl. Abbildung 3.3). Zum anderen ist es im vorliegenden Szenario generell zu präferieren, da es weitestgehend von mehreren Prozessoren gleichzeitig ausgeführt werden kann. Es erhöht somit den parallelen Anteil des Quellcodes und resultiert in einer besseren Skalierbarkeit des Programms (vgl. Abschnitt 2.4).

Abschließend wird aus der Gesamtliste erneut eine Schlüssel-Wert Struktur erstellt. Auch dies kann aufgrund der Tatsache, dass die N Elemente der Liste sortiert sind, theoretisch mit einer Komplexität von $O(N)$ geschehen. Für die *GROUP* Funktion resultiert somit die folgende Typbeschreibung:

```
LIST(KVP(Schlüssel : LIST((Dauer-Wert, Volumen-Wert)) ) ) →
KVP(Schlüssel : LIST(Dauer-Wert, Volumen-Wert) )
```

[10]Die Herleitung der Funktionen ist im Anhang A (Effizienz beim Zusammenfügen von Listen) zu finden.

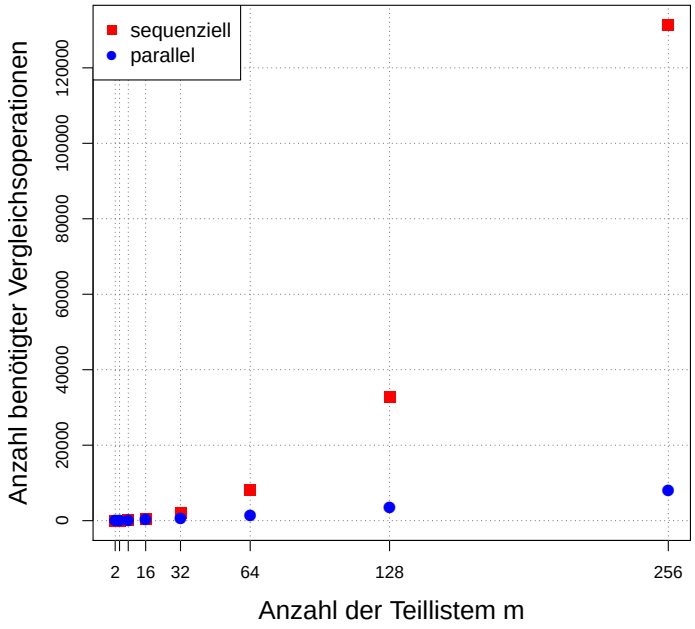

Abbildung 3.3: Anzahl maximal benötigter Vergleichsoperationen beim optimal-parallelen Zusammenfügen mehrerer Teillisten mit jeweils 4 Elementen

Die *REDUCE* Phase entspricht im vorliegenden Szenario einer simplen Summation der Werte aus den Aggregationsfeldern *Dauer* und *Volumen*. Diese sind in der globalen Schlüssel-Wert Struktur in den Listen enthalten, auf die jeweils ein distinkter Schlüssel referenziert. Die Bildung eines Aggregats pro Gruppe ist unabhängig von allen anderen und daher wieder vollständig parallelisierbar. Ebenso wie beim *MAP* geschieht dies durch die Funktion höher Ordnung `map`, welche die eigentliche Funktion auf sämtliche Listen anwendet. Somit ergibt sich für ihren Typ folgende Beschreibung:

```
(Schlüssel : LIST((Dauer-Wert, Volumen-Wert)) ) →
(Schlüssel, (Dauer-Aggregat, Volumen-Aggregat))
```

3.2 Haskell Implementierung

3.2.1 Map Funktion

Der Begriff des Parsen beschreibt die syntaktische Analyse von String-basierten oder binären Daten auf Konformität mit einer formalen Grammatik. Eine solche Grammatik kann bspw. als regulärer Ausdruck oder in Backus–Naur Form[11] vorliegen. Es wird generell zwischen *Top-Down* und *Bottom-Up* Parsern unterschieden. Erstere versuchen die kompletten Eingangsdaten ausgehend von einem Startsymbol gegen die Regeln der Grammatik zu prüfen. Eine hierfür weitverbreitete Implementierung sind sogenannte rekursiv-absteigende Parser. Die *Bottom-Up* Vorgehensweise hingegen versucht, ausgehend von den einfachsten Symbolen, die Eingangsdateien schrittweise in das Startsymbol umzuschreiben. Da die Implementierung einer solchen Transformation oftmals sehr komplex ist, werden für diese Aufgabe dedizierte Programme eingesetzt. Bei der Auswahl einer Methode

[11] Benannt nach John Backus und Peter Naur aus dem Bericht über die Programmiersprache Algol 60 (Backus et al. 1962)

für die Erstellung von Parsern offenbaren sich daher in vielen Programmiersprachen zwei grundlegende Optionen: Parser-Generatoren, welche mithilfe domänenspezifischer Sprachen (engl.: *Domain Specific Languages (DSLs)*) meist Bottom-Up Parser konstruieren[12] oder spezialisierte Bibliotheken für deren Beschreibung direkt in der jeweiligen Programmiersprache.

Sogenannte Parser-Generatoren erstellen auf Basis einer DSL den kompilierbaren Quellcode eines Parsers für eine bestimmte Zielsprache. Dieser wiederum kann daraufhin von entsprechenden Compilern in Maschinenbefehle übersetzt werden. In Haskell existiert für diese Aufgabe das Programm *Happy*, welches auch intern im GHC für das Parsen des Quellcodes eingesetzt wird (vgl. Marlow und Gill 2009, Introduction). *Happy* wird oftmals im Verbund mit einem lexikalischen Analysator namens *Alex* eingesetzt, welcher die Eingangsdaten scannt und relevante Fragmente in sogenannte Token umwandelt. Diese werden anschließend an den eigentlichen Parser übergeben, der daraufhin nicht den gesamten Input Byte für Byte durchsuchen muss, sondern lediglich auf dem von *Alex* bereitgestellten Token-Strom operiert. Diese Arbeitsweise ist stark an den beiden UNIX-Programmen *Yacc* und *Lex* für die Erstellung von C/C++ Parsern angelehnt (vgl. Abschnitt 3.3.1).

Bei der zweiten Möglichkeit, den Parser-Bibliotheken, wird im Kontext von Haskell auch von Parser-Kombinatoren gesprochen. Sie erlauben das Beschreiben der Parser-Grammatik direkt im Quellcode des Programms und sind somit nicht abhängig von weiteren externen Programmen. Die Grammatik ist dabei meist aus mehreren atomaren Parserfunktionen aufgebaut, welche mithilfe von Funktionen höherer Ordnung (den eigentlichen Kombinatoren) verbunden werden um komplexe Strukturen zu erfassen. Dank seiner sehr expressiven Syntax ist das Erstellen von Parsern dieser Art einer der Bereiche, in denen Haskell besonders erfolgreich eingesetzt wird[13]. Weitverbreite Parser-Bibliotheken sind z.B. *Parsec*, *attoparsec* oder *uu-parsinglib*.

Sie alle erstellen rekursiv-absteigende Parser, welche gegenüber den *Bottom-Up* Parsern, die von Programmen wie Happy oder Yacc generiert werden, einen entscheidenden Nachteil aufweisen: Es ist ihnen nicht ohne weiteres möglich links-rekursive Grammatiken, d.h. Grammatiken in denen ein Ausdruck sich selbst erneut enthalten kann, zu analysieren[14]. Die Grammatik der einzulesenden Dateien besitzt diese Eigenschaft jedoch nicht.

Theoretisch wäre es unter den gegebenen Bedingungen sogar möglich, den kompletten Parser basierend auf regulären Ausdrücken zu bauen. Dieser Ansatz wird allerdings sehr schnell unübersichtlich und es gibt keine generalisierte Möglichkeit Fehlermeldungen bei einem fehlgeschlagenen Versuch des Parsens zu erhalten, d.h. besonders das Debugging solcher Lösungen kann äußert mühsam sein. Daher werden in Tabelle 3.1 ausschließlich die beiden relevanten Optionen für die Erstellung eines Parsers der vorliegenden Grammatik gegenübergestellt: Der Parser-Generator Happy und die Bibliothek Parsec.

Aufgrund der wesentlich unkomplizierteren Verwendung wird für die Haskell Implementierung die Bibliothek Parsec verwendet. Die zusätzliche Funktionalität sowie das Performance-Plus eines automatisch generierten *Bottom-Up* Parsers ist insofern irrelevant, als das die zu verarbeitende Grammatik sehr einfach ist und solange eine ähnliche Lösung für die C++ Implementierung genutzt wird, trotzdem ein repräsentativer Vergleich der Programme gezogen werden kann.

[12] Es existieren auch Ausnahmen, wie z.B. der Generator ANTLR, welcher Quellcode für *Top-Down* Parser erstellt.

[13] So z.B. in Facebooks Anti-Spam Software, in Form des JSON-Parsers *aeson*, welcher wiederum auf der Parser-Bibliothek *attoparsec* basiert (vgl. Marlow 2015).

[14] Es existieren jedoch auch Algorithmen zur automatischen Transformation links-rekursiver in unkritische, rechts-rekursive Grammatiken (vgl. Frost, Hafiz und Callaghan 2008).

Kriterium	Happy	Parsec
Benutzerfreundlichkeit	– Beschreibung der Grammatik in eigener DSL nötig – Overhead durch externes Programm	+ Beschreibung der Grammatik direkt im Haskell-Code + Mit Vorkenntnissen über Funktionen höherer Ordnung sehr intuitiv
Dokumentationsgrad	∼ Lediglich in Form eines einzigen User-Guides – Teilweise veraltet	+ Mehrere Veröffentlichungen, Online-Anleitungen (z.B. Leijen 2001, Leijen und Meijer 2002)
Performance	+ Maschinell generierte Parser meist schneller als Parser-Kombinatoren	∼ Abhängig vom Grad der Ambiguität der Grammatik (aufgrund von Backtracking)
Warnungen und Fehlermeldungen	+ Warnungen zur Grammatik während des Kompilierens	– Fehlermeldungen erst zur Laufzeit

Tabelle 3.1: Vergleich des Parser-Generators Happy mit der Bibliothek Parsec

Die Beschreibung der Grammatik für das Haskell Programm befindet sich in dem dedizierten Haskell-Modul `Grammar.hs` (Listing B.4). Die logische Struktur des daraus resultierenden Parsers ist in Abbildung A.2 dargestellt, welche ebenfalls im Anhang zu finden ist. Im Folgenden soll die Funktionalität dessen zentraler Bestandteile, unter Verwendung einiger Quellcodefragmente, erläutert werden.

Wie zuvor bereits angemerkt, versucht ein rekursiv-absteigender Parser ausgehend von einer Startregel den gesamten übergebenen Zeichenstrom zu konsumieren. Dieser Ausgangspunkt ist in diesem Fall die in Listing 3.1 abgebildete Funktion `recordFile`. Ihr Funktionstyp verrät, dass der Rückgabewert des Parsers eine Liste von `Records` ist. Der Parameter `st` kann optional für die Repräsentation eines Zustands während der Laufzeit verwendet werden, wird in der vorliegenden Implementierung jedoch nicht genutzt.

```
13 recordFile :: GenParser st [Record]
14 recordFile = do
15     contents <- record `sepEndBy1` eol
16     eof
17     return contents
```

Listing 3.1: haskell/Grammar.hs

Dabei wird die Nutzung der angesprochenen Funktionen höherer Ordnung als Kombinatoren ersichtlich: Die Funktion `(a `sepEndBy1` b)` analysiert eine Sequenz mindestens einer Vorkommnis von `a`, getrennt und optional beendet durch `b`. Ergebnis einer erfolgreichen Applikation ist eine Liste der konsumierten Vorkommnisse von `a`, welche im Symbol `contents` gespeichert wird. Sämtliche Instanzen von `b` werden hingegen verworfen. Die in diesem Fall verwendeten Symbole `record` und `eol` sind wiederum eigens definierte Parser-Funktionen. Konkret besagt diese Grammatikregel also, dass eine Datei aus mehreren Records bestehen kann, die jeweils durch einen Zeilenumbruch getrennt sind. Wird das Ende der Datei (`eof`) erreicht, so werden sämtliche enthaltenen Records in Form einer Liste zurückgegeben.

Die nächste aufgerufene Funktion ist der `record` Parser in Listing 3.2.

```
19 record :: GenParser st Record
20 record = do
21     recordBegin <?> "start of record"
22     metas       <- count 7 (char '#' *> spaces *> metaKvp <* eol)
23     attribs     <- count 32 (char 'F' *> spaces *> attribKvp <* eol)
24     recordEnd   <?> "end of record"
25     return $! Record (M.fromList metas) (M.fromList attribs)
```

Listing 3.2: haskell/Grammar.hs

Es ist zu erkennen, dass ein Record aus vier logischen Abschnitten aufgebaut ist: einem Beginnzeichen (`recordBegin`), Metafeldern (`metas`), Attributfeldern (`attribs`) und einem Endezeichen (`recordEnd`). Zunächst wird versucht das Beginnzeichen zu erkennen, welche einem simplen Parser für die String-Sequenz „RECORD" entspricht. Schlägt dieser Versuch fehl, so sorgt der Operator `<?>` dafür, dass eine aussagekräftige Nachricht über den erwarteten Input ausgegeben wird.

Da eine feste Anzahl von Meta- und Attributfeldern erwartet wird, kann deren Analyse mithilfe der Funktion `count n p` gesteuert werden, welche einen Parser `p` exakt `n`-mal auf den Datenstrom anwendet. Die Funktionen `(a *> b)` und `(a <* b)` entstammen dem Haskell Modul `Control.Applicative` und veranschaulichen, wie Parser-Kombinatoren, aufgrund ihrer direkten Integration in Haskell, von existierenden Funktionen höherer Ordnung profitieren können. Sie führen die übergebenen Funktionen `a` und `b` in dieser Reihenfolge aus, verwerfen allerdings den Rückgabewert der linken, bzw. rechten Seite. Da für die interne Repräsentation der Recordfelder die jeweiligen Startzeichen `'#'` und `'F'` nicht relevant sind, können diese z.B. nach erfolgreicher Analyse ignoriert werden. Wurden sämtliche Record-Inhalte sowie das Endezeichen erfolgreich erfasst, so erstellt die Funktion einen Rückgabewert des Typs `Record`, in dem die Meta- und Attributfelder jeweils in Form einer Schlüssel-Wert Struktur (`Data.Map.Map`) enthalten sind. Die Definitionen dieses und sämtlicher weiterer benutzerdefinierter Typen des Haskell Programms, befindet sich im Modul `Types.hs` (Listing B.3).

Die beiden Funktionen `metaKvp` und `attribKvp` für die Analyse von Schlüssel-Wert Paaren beider Mengen sind nahezu identisch aufgebaut, daher wird im Folgenden nur `metaKvp` detailliert erläutert.

```
27 metaKvp :: GenParser st KeyValuePair
28 metaKvp = do
29     key   <- validMetaName <* space
30     value <- option "" valueChars
31     return $! (key, value)
```

Listing 3.3: haskell/Grammar.hs

Der Parser in Listing 3.3 ist aus den zwei Komponenten Feldname und Wert aufgebaut. Beide sind getrennt durch ein Leerzeichen, welches hier erneut durch Applikation von `(<*)` verworfen wird. Generell werden sämtliche Werte als optional angenommen, daher gibt die Funktion `option` entweder das erfolgreich analysierte Ergebnis des Parsers `valueChars` oder den leeren String zurück. Der Rückgabewert `KeyValuePair` ist lediglich ein Alias für ein Tupel der Form `(String, String)`.

```
53 validMetaName :: GenParser st String
54 validMetaName = choice (map (try . string) all_metas)
55                 <?> "valid meta field name"
```

Listing 3.4: haskell/Grammar.hs

Die Funktion `validMetaName` in Listing 3.4 erstellt auf Basis einer Liste aller möglichen Meta-Feldnamen (`all_metas`) einen String-Parser, welcher genau eines dieser Felder konsumiert. Diese Auswahl-Semantik wird dabei durch die Funktion `choice` realisiert. Das Schlüsselwort `try` hingegen ist notwendig, um sogenanntes Backtracking zu unterstützen: Schlägt das Parsen eines Feldnamens mitten im Wort fehl, so muss der Parser den eingelesenen Input erneut von Beginn an analysieren, denn andere Feldnamen könnten durchaus mit dem gleichen Präfix beginnen. Ohne Backtracking wäre es nicht möglich, die bereits konsumierten Teile einer Regel erneut zu parsen.
`validAttribName` stellt die entsprechende Funktion für Attributfelder dar.

Die Anwendung des Parsers auf Eingangsdaten des Typs `Text` wird in der Funktion `parseRecords` zusammengefasst, dargestellt in Listing 3.5. Die Parsec Bibliotheksfunktion `parse p e d` wendet den Parser `p` auf die Eingangsdaten `d` an. Der Parameter `e` wird lediglich für Fehlermeldungen genutzt und kann einem leeren String entsprechen. Der monadische Rückgabewert innerhalb eines `Either` impliziert die Möglichkeit einer fehlgeschlagenen Funktionsapplikation. Konsumiert der Parser die gesamten zur Verfügung gestellten Daten, so hat das Ergebnis die Form `Right [Record]`, im Fehlerfall hingegen `Left ParseError`.

```
10 parseRecords :: T.Text -> Either ParseError [Record]
11 parseRecords input = parse recordFile "" input
```

Listing 3.5: haskell/Grammar.hs

Im Anschluss an die erfolgreiche Umwandlung der Dateiinhalte in die interne Datenstruktur `Record` verbleiben zwei Funktionalitäten, welche ebenfalls in der *MAP* Phase zu verorten sind – die Validierung der eingelesenen Daten und das Transformieren der Record-Listen in den Typ von Schlüssel-Wert Strukturen. Die erste Aufgabe wird von der Funktion `validate` übernommen (vgl. Listing B.2). Dabei besteht die Überprüfung aus drei Teilfunktionen, welche sicherstellen, dass jeder Record folgende Eigenschaften erfüllt:

1. Es ist kein Meta- oder Attributfeld doppelt enthalten.
2. Schlüssel- und Aggregationswerte sind nicht leer, d.h. ungleich `Nothing`.
3. Die Werte der beiden Aggregationsfelder sind ungleich Null.

Die Konformität zu Punkt 1 stellt sicher, dass es sich um einen wohlgeformten Record handelt, der ohne Probleme in eine Schlüssel-Wert Struktur mit distinkten Schlüsseln umgewandelt werden kann. Punkt 2 garantiert, dass sämtliche für die Aggregation relevante Felder mit einem Wert hinterlegt sind. Wäre dies nicht der Fall, so könnte kein vollständiger Schlüssel, bzw. kein korrektes Aggregat gebildet werden. Zuletzt werden durch Punkt 3 alle Records ausgefiltert, welche aufgrund ihrer Wertbelegung für die Felder *Dauer* und *Volumen* keinen Einfluss auf die Berechnung der Aggregate haben. Auf diese Weise können im Verlauf der weiteren Verarbeitung Speicherressourcen eingespart werden, die sonst unnötig belegt wären.

An diese Idee anknüpfend, findet beim Transformieren der einzelnen Record-Listen in Schlüssel-Wert Strukturen durch die Funktion `makeMap` (vgl. Listing B.2) eine Verkleinerung der Datenstruktur statt. So werden an dieser Stelle alle Meta- und Attributfelder entfernt, welche nicht für die Berechnung der Aggregate benötigt werden. Ausschließlich Schlüssel- und Aggregationsfelder sind Teil der resultierenden Datenstruktur. Für die Erstellung der Schlüssel werden dafür durch die Teilfunktion `makeKey` die Werte der Schlüsselfelder, getrennt durch jeweils ein Leerzeichen, in dieser Reihenfolge konkateniert: *Aggregationsdatum*, *Eventklasse*, *Eventsource*, *Plattform*, *Tarifzone*. Einer dieser Schlüssel referenziert dabei, wie im Abschnitt 3.1 beschrieben, auf eine Liste zugehöriger Tupel

aus den Werten der Aggregationsfelder. Diese Tupel werden mit der Funktion `makeTuple` (Listing B.3) aus der ursprünglichen Record-Struktur extrahiert.

Das Einlesen und Parsen der Dateiinhalte wird in die separate Funktion `readAndParse` ausgelagert (vgl. Listing B.2), welche vor dem Aufruf des eigentlichen MapReduce Algorithmus ausgeführt wird. Auf diese Weise kann als Zwischenschritt eine Behandlung möglicherweise aufgetretener Fehler erfolgen, so dass der weiteren Verarbeitung nur die Records übergeben werden, die vom Parser erfolgreich analysiert wurden. Dateien, welche hingegen fehlerhafte Datenstrukturen enthalten, werden ausgefiltert und ihr Name zur Information auf der Standardausgabe ausgedruckt.

Die Funktion `readAndParse` enthält den Operator `($!)`, welcher essenziell für die Performance des Programms ist. Er sorgt dafür, dass die ihm übergebenen Argumente zu WHNF evaluiert werden. Würde dies nicht geschehen, so gäbe es zu diesem Punkt keinen Bedarf für das Einlesen und Parsen der Dateien. Haskells Laufzeitumgebung würde lediglich Thunks für die eventuelle Auswertung dieser Berechnungen erstellen, diese aber weiter verzögern. Die so erzwungene strikte Evaluierung stellt sicher, dass die Funktion diese Aufgaben sofort ausführt. Mittels der Funktion `mapM`, welches einem parallelen `map` für monadische Berechnungen entspricht, wird sie auf die Liste der zu bearbeitenden Dateinamen angewendet. Die Ergebnisse werden in der Variable `parsed` gespeichert. (Vgl. Listing B.1, Zeile 22).

3.2.2 Group und Reduce Funktion

Es folgt das Zusammenfassen der einzelnen Schlüssel-Wert Datenstrukturen in einer globalen Variable. Auch wenn das Modul `Data.Map` bereits die Funktionen `union` und `unionWith` für die Kombination zweier Instanzen dieses Typs bereitstellt, soll dies im Sinne der besseren Vergleichbarkeit des Haskell und C++ Programms mittels der selbst definierten Funktion `mergeMaps` erfolgen.

Die Typdefinition in Listing 3.6 lässt erkennen, dass deren Eingangsparameter die Liste aus Schlüssel-Wert Strukturen der einzelnen Dateien ist, welche in der zuvor beschriebenen *MAP* Phase erstellt wurde. Ihr Rückgabewert ist ebenfalls eine solche Struktur, die aber sämtliche Schlüssel-Wert Paare der Eingangsdaten in sich vereint. In den Zeilen 37 und 38 werden zunächst die Ergebnisse für die trivialen Fälle der leeren und einelementigen Liste definiert. Im Fall der leeren Liste, ist dies die leere Schlüssel-Wert Struktur `Data.Map.empty`. Liegt hingegen genau eine Schlüssel-Wert Struktur vor, entspricht sie selbst dem Ergebnis der Funktion, da kein weiteres Element existiert, das es mit ihr zu kombinieren gilt.

```
36 mergeMaps :: (Ord k, Ord v) => [M.Map k [v]] -> M.Map k [v]
37 mergeMaps []       = M.empty
38 mergeMaps [x]      = x
39 mergeMaps (x:y:xs) =
40   let x'  = M.fromAscListWith (++) $ mergeLists (M.toAscList x)
41                                                 (M.toAscList y)
42       xs' = mergeMaps xs
43   in M.fromAscListWith (++) $ mergeLists (M.toAscList x')
44                                          (M.toAscList xs')
```

Listing 3.6: haskell/Helpers.hs

In der dritten Definition von `mergeMaps` wird Pattern Matching genutzt, um die ersten beiden Elemente der übergebenen Liste zu extrahieren. Mithilfe der vordefinierten Funktion `Data.Map.toAscList` werden diese zunächst in eine aufsteigend geordnete Liste

ihrer Schlüssel-Wert Paare umgewandelt[15]. Die Funktion `mergeLists`, ebenfalls definiert im Modul `Helpers.hs`, kombiniert anschließend beide Listen, bevor ihr Resultat durch `Data.Map.fromAscListWith` erneut in eine Schlüssel-Wert Struktur umgewandelt wird. Der erster Parameter von `fromAscListWith` ist eine Kombinatorfunktion, welche benötigt wird, sollte die Liste mehrere Elemente des gleichen Schlüssels enthalten. In diesem Fall werden die Werte identischer Schlüssel mit der Funktion `(++)` kombiniert, welche zwei Listen konkateniert, so dass der Rückgabewert der *GROUP* Phase der gleiche ist, wie in Abschnitt 3.1 beschrieben.

Darüber hinaus erfolgt neben den Zusammenfügen der beiden ersten Elemente ein rekursiver Aufruf der Funktion `mergeMaps` für den Rest der Liste. In einem letzten Schritt werden die Ergebnisse beider Berechnungen zur finalen Schlüssel-Wert Struktur zusammengefasst.

Bevor die `reduce` Funktion angewendet werden kann, müssen die Zwischenergebnisse in Form einer Liste von Schlüssel-Wert Paaren vorliegen. Diese Restriktion besteht, damit die *REDUCE* Phase im nächsten Schritt durch `map` parallelisiert werden kann. Die Funktion `splitMap` in Listing 3.7 realisiert diese Typumwandlung durch einen simplen Aufruf der Funktion `toList` aus Haskells `Data.Map` Modul. Es wird `toList` statt `toAscList` genutzt, da eine Sortierung der Schlüssel an diesem Punkt nicht mehr relevant ist.

```
29 splitMap :: M.Map k [v] -> [(k,[v])]
30 splitMap = M.toList
31
32 reduce :: (String, [(Int,Int)]) -> (String, (Int,Int))
33 reduce (key,values) = (key,aggregates) where
34     aggregates = foldl' (\(d',v') (d,v) -> ((d'+d),(v'+v))) (0,0) values
```

Listing 3.7: haskell/Helpers.hs

Die eigentliche `reduce` Funktion wird mittels der Funktion höherer Ordnung `foldl` realisiert, welche im Abschnitt 2.3 vorgestellt wurde. Deren Variante `Data.List.foldl'` erfüllt die gleiche Funktionalität, evaluiert jedoch ihre Elemente strikt, d.h. sie werden generell zu WHNF ausgewertet. Auf diese Weise wird der Akkumulator bei jedem Aufruf der Kombinationsfunktion soweit ausgewertet, dass sich keine Sequenz unevaluierter Funktionsaufrufe bildet, so wie dies in Listing 2.5 dargestellt wurde. Dies wirkt sich oftmals positiv auf die Laufzeit der Funktion aus und ein möglicher Überlauf des Stackspeichers bei zu großer Rekursionstiefe wird vermieden. Als Kombinator dient eine anonyme Funktion, welche zwei Tupel entgegennimmt und ihre ersten und zweiten Elemente miteinander addiert. Ausgangswert dieser Summation ist das Tupel neutraler Integer-Elemente `(0,0)`. Das Ergebnis der `reduce` Funktion ist wiederum ein Tupel aus Schlüssel und dazugehörigen Aggregaten für *Dauer* und *Volumen*.

Somit wurden nun alle notwendigen Teilfunktionen definiert. Die eigentliche Komposition zu einer voll umfänglichen `mapReduce` Funktion ist im Hauptmodul des Programms zu finden (Listing B.1). Dabei wird die Kombination aus den Funktionen `validate` und `makeMap`, welche gemeinsam mit dem vorangegangenen Einlesen der Dateien die logische *MAP* Phase bilden, mittels eines `map` auf die analysierten Eingangsdaten angewandt. Anschließend werden in der *GROUP* Phase die Ergebnisse durch die Funktion `mergeMaps` zusammengefasst und durch `splitMap` in eine Liste von Schlüssel-Wert Paaren transformiert. Die finale Applikation der `reduce` Funktion geschieht durch ein `map` über alle Elemente in der Liste der Zwischenergebnisse.

[15] Die Sortierung basiert dabei auf dem Wert des Schlüssel vom Typ String.

3.2.3 Parallelisierung

Wie bereits im Abschnitt 1.2 dargelegt, gibt es verschiedene Möglichkeiten, Haskell-Programme von paralleler Hardware profitieren zu lassen. Es bleibt jedoch festzustellen, welche dieser Optionen für die Parallelisierung des erstellten Programms am besten geeignet ist.

Eine dieser Möglichkeiten ist die Nutzung von Funktionen der Spracherweiterung *Eden*. Diese können über das Modul `Control.Parallel.Eden` eingebunden werden. *Edens* Modell der Parallelisierung zielt vorrangig auf verteilte Systeme ab, bei Kompilierung des Quellcodes mit dem normalen GHC werden die sonst verteilten Konstrukte jedoch auf lokale Threads abgebildet (vgl. Eden Group 2016). Es eignet sich daher per se auch für die Programmierung von Software für *Shared Memory Machines (SMMs)*.

Eines der zentralen Elemente *Edens* ist die `process` Funktion, welche aus einer übergebenen Funktion eine Prozessabstraktion erstellt. Diese Abstraktionen sind effektiv Funktionen, die parallel auf entfernten Rechenknoten ausgeführt werden können (engl.: *remote functions*). *Eden* bietet weiterhin den Operator `(#)` zur Prozessinstanziierung, d.h. das Bereitstellen von Eingangsdaten für eine Prozessabstraktion und damit verbunden das tatsächliche Starten eines Prozesses.

Es ist ersichtlich, dass diese Art der Parallelisierung, wenn auch vielseitig einsetzbar, für das gegebene Problem zu detailliert ist. Im Idealfall sollte sich die Auszeichnung von parallelen Bestandteilen des Quellcodes auf so wenig Änderungen wie möglich beschränken, schließlich bleibt die eigentliche Funktionalität des Programms gänzlich unverändert. *Eden* enthält zwar zusätzlich vordefinierte Funktionen für die Beschreibung eines Map-Reduce Problems (`Control.Parallel.Eden.MapReduce`), diese erlauben jedoch nicht die Spezifizierung einer eigenen Gruppierungsfunktion und sind somit für den vorliegenden Algorithmus zu restriktiv.

Ein weiteres Argument, welches gegen den Einsatz von *Eden* spricht, ist der zusätzliche Aufwand für das Verständnis der Spracherweiterung. Es wird sich zeigen, dass für den gegebenen Anwendungsfall geeignete Lösungen in Standard-Haskell existieren, die eine wesentlich einfachere Lösung ermöglichen.

Eine weitere Möglichkeit ist die `Par` Monade, vorgestellt von Marlow, Newton und S. Peyton Jones (2011), welche deterministische Parallelität, enkapsuliert in einer eigens dafür geschaffene Monade, ermöglicht. Hauptbestandteil der auf diese Weise eingeführten Parallelität ist die Funktion `fork`, welche eine parallele Teilberechnung erstellt, ähnlich einem Kind-Prozess. Für die Kommunikation zwischen mehreren dieser Berechnungen werden sogenannte `IVars` genutzt, welche mithilfe der Funktionen `get` und `put` gelesen und geschrieben werden können. Ihre Typdefinitionen lauten wie folgt:

```
fork :: Par () -> Par ()
get  :: IVar a -> Par a
put  :: NFData a => IVar a -> a -> Par ()
```

Listing 3.8: Funktionen der `Par` Monade nach Marlow, Newton und S. Peyton Jones 2011

Die Argumente der Funktion `put` müssen dabei der Typklasse `NFData` angehören, d.h. es muss möglich sein, sie vollständig nach NF auswerten zu können. Jedes Mal wenn die Funktion gerufen wird, evaluiert sie ihre Argumente vollständig. Diese Restriktion wurde der `Par` Monade auferlegt um sicherzugehen, dass Aufgaben, welche in parallelen Bereichen des Codes spezifiziert wurden auch tatsächlich an diesen Stellen berechnet werden und keine unevaluierten Thunks zurückgegeben werden.

Diese Eigenschaft ist im vorliegenden Beispiel jedoch eher kontraproduktiv. Tatsächlich besteht vor der eigentlichen Aggregation kein Bedarf die Werte der intern verwendeten

`Data.Map` Strukturen auszuwerten. Nicht einmal während der Validierung der Werte in der *MAP* Phase sollte es zum Forcieren der NF kommen. Für den Test auf Vorhandensein aller Schlüssel sind die Werte gänzlich irrelevant, für die Überprüfung auf einen Wert ungleich `Nothing` müssen sie ebenfalls nur bis zum `Just` Konstruktor der monadischen `Maybe` Typen ausgewertet werden und bei der Bedingung, `Dauer` und `Volumen` seien ungleich Null, ist lediglich die Evaluation des ersten Zeichens nötig, da beide Werte zu diesem Zeitpunkt noch als Liste von `Chars` (`= String`) vorliegen. Auch die `Par` Monade ist daher nicht die ideale Wahl für die Optimierung des Programms

Eine dritte Variante sind die im Abschnitt 2.2 vorgestellten Evaluationsstrategien, definiert im Modul `Control.Parallel.Strategies`. Ihr Einsatz beschränkt sich auf „lazy" Datenstrukturen, d.h. solche, die durch Bedarfsauswertung nicht immer vollständig evaluiert werden müssen. Die zu bearbeitenden Schlüssel-Wert Datenstrukturen weisen genau diese Eigenschaft auf[16]. Beim Einfügen von Werten in den Typ `Map` werden zunächst ausschließlich die Schlüssel ausgewertet.

Das Modul definiert bereits einige Sprachkonstrukte zum Traversieren häufig verwendeter Datenstrukturen, darunter die Funktion `parMap`, welche die parallele Applikation einer Funktion auf eine Liste unter Verwendung einer bestimmten Strategie realisiert. Die eigentliche Parallelität ist der Funktion dabei schon inhärent. Die zu spezifizierende Strategie gilt lediglich dem Evaluationsgrad der Listenelemente. In Listing 3.9 ist dargestellt, wie sie anstelle der herkömmlichen `map` Aufrufe in der parallelen Version der Funktion `mapReduce` verwendet werden kann.

```
39 mapReducePar input = parMap rdeepseq reduce $
40                      (splitMap . mergeMapsPar) $
41                      parMap rseq (makeMap . validate) input
```

Listing 3.9: haskell/Main.hs

In der *MAP* Phase genügt eine Auswertung in WHNF durch die Strategie `rseq`, da die referenzierten Werte der Schlüssel zu diesem Zeitpunkt noch nicht benötigt werden. Erst beim abschließenden *REDUCE*, also bei der Summation der Werte-Paare, kommt es zu einer vollständigen Auswertung durch `rdeepseq`.

Es verbleibt die Parallelisierung der *GROUP* Phase. Diese unterscheidet sich insofern von *MAP* und *REDUCE*, als dass sie nicht mittels eines `map` auf eine Menge von Daten angewendet wird. Dennoch sind Evaluationsstrategien auch hier ein geeigneter Ansatz, um die originale Funktion aus Listing 3.6 mit nur wenigen Veränderungen zu parallelisieren:

```
46 mergeMapsPar :: (Ord k, Ord v) => [M.Map k [v]] -> M.Map k [v]
47 mergeMapsPar []       = M.empty
48 mergeMapsPar [x]      = x
49 mergeMapsPar (x:y:xs) = runEval $ do
50     x'  <- rpar $ M.fromAscListWith (++) $ mergeLists (M.toAscList x)
51                                                       (M.toAscList y)
52     xs' <- rseq $ mergeMapsPar xs
53     rseq x'
54     return $ M.fromAscListWith (++) $ mergeLists (M.toAscList x')
55                                                  (M.toAscList xs')
```

Listing 3.10: haskell/Helpers.hs

[16]Das `Data.Map` Modul kann in der strikten Variante `Data.Map.Strict` oder als `Data.Map.Lazy` eingebunden werden. In dem beschriebenen Programm wird die „lazy" Variante verwendet.

Mittels `rpar` wird ein Spark für das Zusammenfügen der Schlüssel-Wert Strukturen `x` und `y` erstellt, welcher im Folgenden von einem beliebigen Prozessor berechnet werden kann. Währenddessen sorgt der rekursive Aufruf `mergeMapsPar xs` in Verbindung mit `rseq` dafür, dass gleichzeitig auch der Rest der übergebenen `Data.Maps` zusammengefügt wird. Dabei werden bei jeder Rekursionsstufe weitere Sparks erstellt. Der nachgelagerte Aufruf von `rseq x'` garantiert, dass die Berechnung von `x'` vor der abschließenden Operation tatsächlich erfolgt ist. Schlussendlich wird die Vereinigungsmenge beiden Schlüssel-Wert Strukturen `x'` und `xs'` zurückgegeben. Da all diese parallelen Berechnungen in der `Eval` Monade stattfinden (vgl. Abschnitt 2.2, Evaluationsstrategien), ist ein Aufruf der Funktion `runPar` notwendig, um das Resultat wieder in einen nicht-monadischen Wert umzuwandeln.

Diese zwei Änderungen an den Funktionen `mapReduce` und `mergeMapsPar` sind alles was nötig ist um den bestehenden Algorithmus auf parallele Ausführung vorzubereiten. Damit diese Optimierungen auch tatsächlich genutzt werden, sind bestimmte Flags beim Kompilierungsprozess zu setzen, so dass Haskells Laufzeitumgebung die Funktionalität auf mehrere Betriebssystemthreads abbildet. Genaue Informationen dazu sind im Abschnitt 4.1 zu finden.

3.3 C++ Implementierung

3.3.1 Map Funktion

In der Programmiersprache C++ existieren die selben grundlegenden Optionen zur Erstellung von Parsern, die bereits im Abschnitt 3.2 dargelegt wurden: Parser-Generatoren und -Bibliotheken sowie eigens erstellte Parserfunktionen auf Basis regulärer Ausdrücke. Yacc und Lex, bzw. ihre GNU-Alternativen Bison und Flex gehören zu den verbreitetsten Parser-Generatoren überhaupt. Auf Basis der Beschreibung einer kontextfreien Grammatik in Backus-Naur Form sind sie in der Lage Quellcode für *Bottom-Up* Parser in den Sprachen C, C++ und Java zu erstellen. Aber die automatische Generierung von Parsern durch externe Software bringt auch in diesem Fall einen nicht-trivialen Mehraufwand für die Entwicklung der Programme mit sich. Die aktuelle Version des Bison Benutzerhandbuchs (Donnelly und R. Stallman 2015) umfasst 224 Seiten und viele der darin beschriebenen Funktionalitäten gehen über die für das Parsen der vorliegenden Grammatik benötigten Features hinaus.

Auch die komplett selbstständige Erstellung eines Parsers mittels regulärer Ausdrücke scheitert aus dem selben Grund, wie bei der Haskell Version: ein solcher Lösungsansatz bietet per se keine Möglichkeit Informationen über aufgetretene Fehler während der Syntaxanalyse zu erhalten. Die Ausgabe der Fehlerposition im Datenstrom beispielsweise, die wertvolle Details beim Debuggen eines Parsers liefern kann, müsste aufwendig neu implementiert werden. Abhängig von der Granularität der einzelnen Parserfunktionen, wird das Parsing mit regulären Ausdrücken außerdem schnell unübersichtlich. Je nachdem wie die Eingangsdaten gelesen werden – eine komfortable Möglichkeit wäre z.B. ein `std::stringstream`, welcher es erlaubt String-Input mittels der Funktionen des Stream-Interface zu behandeln – kann diese Methodik auch in deutlich langsameren Parsern resultieren als solche, die durch ausgereifte Parser-Bibliotheken erstellt werden. Aus diesen Gründen, aber auch um die Untersuchung beider Programme möglichst vergleichbar zu gestalten, sollte daher für die C++ Implementierung ebenfalls eine Parser-Bibliothek genutzt werden.

Die Auswahl an entsprechenden Bibliotheken in der C++ Welt ist im Vergleich zu Has-

kell eher beschränkt. Ein von Parsecs Kombinatoren inspiriertes Projekt namens *Parsnip* (Rubinsteyn 2016) versucht ähnliche Funktionalitäten bereitzustellen, wurde allerdings auf der Software-Plattform Sourceforge schon seit dem Jahr 2007 nicht mehr aktualisiert. Es kann daher nicht als ein repräsentatives Beispiel moderner C++ Parser angesehen werden.

Das *Boost* Projekt hingegen ist eine aktiv entwickelte Sammlung verschiedenster C++ Bibliotheken, die mitunter zu einem späteren Zeitpunkt auch in den offiziellen Sprachstandard übergehen. Durch einen geregelten Peer-Review Prozess, den sämtliche Teilprojekte zu durchlaufen haben, ist eine gewisse Qualität der enthaltenen Software sichergestellt (vgl. Dawes 2000). Teil dieser Sammlung ist unter anderem die Bibliothek *Spirit*, welche rekursiv-absteigende Parser mithilfe einer eingebetteten *Domain Specific Language (DSL)*, basierend auf C++ Expression Templates und Metaprogrammierung[17], erstellt. Auf diese Weise kann die Grammatik eines Parsers in einer Header-Datei definiert, in den bestehenden Programmcode eingebunden und durch den GCC mitkompiliert werden.

Ähnlich wie bei der Parsec-Grammatik sollen im Folgenden die wichtigsten Regeln des *Spirit* Parsers von oben nach unten erläutert werden. In Abbildung A.3 ist die logische Reihenfolge der aufgerufenen Regeln bildlich dargestellt. Die `start` Regel in Listing 3.11 stellt den Eintrittspunkt des Parser dar. Auch hier wird eine Datei beschrieben als Kombination mehrerer Ergebnisse des Parsers `record`, getrennt durch jeweils einen Newline-Charakter.

```
28  start = +(record >> eol)    /* std::vector<Record>()    */
29        >> eoi;
    /* ... */
116 qi::rule<Iterator, std::vector<Record>(), qi::blank_type> start;
```

Listing 3.11: cpp/grammar.hpp

Der Operator `(+)` steht in der Grammatik – ähnlich wie bei regulären Ausdrücken – für ein oder mehr Vorkommnisse des nachfolgenden Parsers. Eine simple Sequenzierung zweier Regeln wird durch `( >> )` beschrieben. Die *Spirit* Bibliothek übernimmt indes automatisch die Erstellung der internen Datenstrukturen. Dafür muss jede selbst definierte Parserregel, `rule` genannt, innerhalb der Grammatik mit einem Typ deklariert werden (vgl. Listing C.4 Zeile 116ff). Oben abgebildete Deklaration in Zeile 116 bedeutet, dass die Regel auf einem übergebenen `Iterator` operiert und einen Wert vom Typ `std::vector<Record>` produziert. Iteratoren sind ein fundamentales Konzept von Datenstrukturen in der C++ Standardbibliothek. Sie stellen im Grunde genommen Zeiger dar, die auf die Position eines Elements innerhalb eines Containers zeigen und ermöglichen so einen einheitlichen Zugriff auf eine Vielzahl verschiedener Typen.

Der dritte Template-Parameter der Regel erlaubt das Spezifizieren eines *Skippers*, also eines Parsers, mit dem irrelevate Teile der Eingangsdaten übersprungen werden sollen. In diesem Fall sind das Whitespaces vom Typ `spirit::qi::blank_type`, was Tabulatoren und Leerzeichen umfasst, nicht jedoch Zeilenumbrüche. *Spirit* versucht daraufhin diesen Parser automatisch zwischen allen enthaltenen Regeln anzuwenden. Um sicherzustellen, dass die gesamte Datei konsumiert wird, erwartet die `start` Regel letztendlich den vordefinierten End-of-Input Parser `eoi`.

Die Parser-Regel `record` (Listing C.4, Zeile 31) setzt sich, ähnlich wie bei Parsec, aus Beginn- und Endezeichen sowie zwei Schlüssel-Wert Strukturen für Meta- und Attributfelder zusammen. *Spirit* kann auch hier den Typ der Regel automatisch ableiten, da bestimmte Parser ohne Rückgabewert definiert sind – darunter auch `RecordBegin` und `RecordEnd`.

[17] Metaprogrammierung bezeichnet die Erstellung von Quellcode durch anderen Quellcode.Vorteil dieser Methodik ist ein hoher Abstraktionsgrad und oftmals eine bessere Performance, da der Compiler generell besser optimierten Programmcode erstellt als dies händisch möglich wäre. Expression Templates (Veldhuizen 1995) sind dabei nur eine Form der Metaprogrammierung.

Denn für das Parsen ist allein ihr Vorhandensein wichtig, sie tragen jedoch keine relevanten Informationen in sich. Die weiterhin enthaltenen `metaSection` und `attrSection` sind hingegen Schlüssel-Wert Strukturen vom Typ `std::map<string,string>` und entsprechen daher zusammen einer Instanz des Typs `Record`. Dieser selbst definierte Typ und sämtliche benutzten Typaliasse sind in der separaten Header-Datei `types.hpp` (Listing C.3) zu finden.

Der restliche Teil des Parsers wird nachfolgend am Beispiel der Regeln für Metafelder erläutert. Die entsprechenden Regeln für Attributfelder sind nahezu identisch aufgebaut und dem Quellcode in Listing C.4 zu entnehmen.

```
44  metaLine = '#'              /* literal has no attribute */
45           >> metaKvp         /* stringPair               */
46           >> eol;            /* eol has no attribute     */
    /* ... */
52  metaKvp = validMetaKey
53          >> -value;    /* values are optional by default */
    /* ... */
100 value = +qi::char_("a-zA-Z0-9_.:/-"); /* value charset  */
    /* ... */
120 qi::rule<Iterator, stringMap(), qi::blank_type> metaSection;
121 qi::rule<Iterator, stringMap(), qi::blank_type> attrSection;
122 qi::rule<Iterator, stringPair(), qi::blank_type> metaLine;
123 qi::rule<Iterator, stringPair(), qi::blank_type> attrLine;
124 qi::rule<Iterator, stringPair(), qi::blank_type> metaKvp;
125 qi::rule<Iterator, stringPair(), qi::blank_type> attrKvp;
```

Listing 3.12: cpp/grammar.hpp

Eine `metaLine` beginnt mit dem Charakter `'#'`, gefolgt von einem Schlüssel-Wert Paar `metaKvp` und dem Zeilenende. Da weder das Charakter-Literal noch der vordefinierte Parser `eol` einen Rückgabewert besitzen, entspricht das Ergebnis dem der `metaKvP` Regel. Diese konsumiert bestimmte String-Sequenzen möglicher Feldnamen, definiert in der Regel `validMetaKey`, sowie einen darauffolgenden Wert. Der Spirit-Operator `(-)` zeigt an, dass letzterer optional ist. Eventuell vorhandene Whitespaces zwischen Feldname und Wert werden dabei vom zuvor beschriebenen Skipper übersprungen.

Das eigentliche Einlesen der Dateien geschieht innerhalb der Funktion `map_` mittels eines Input-Filestreams (`ifstream`). Die Spirit-Bibliothek erlaubt mit Objekten des Typs `istream_iterator`, wie es in Listing 3.13 (Zeile 80) initialisiert wird, das direkte Parsing von Dateien auf Basis eines solchen Streams. Die Funktion `spirit::qi::parse_phrase` nimmt dafür einen Start- und Ende-Iterator, eine Instanz der anzuwendenden Grammatik, eine Instanz des Skipper-Parsers sowie eine Variable entgegen, in der das gelesene Resultat gespeichert wird. Tritt während der syntaktischen Analyse ein Fehler auf, ist der Rückgabewert der Funktion `false`. So können durch das nachträgliche Prüfen der Variable `ok` Dateien identifiziert werden, die nicht wohlgeformte Records enthalten. Diese werden aus der weiteren Verarbeitung ausgesteuert.

```
80  spirit::istream_iterator begin(in);
81  spirit::istream_iterator end;
    /* ... */
85  bool ok = qi::phrase_parse(begin, end, parser, qi::blank, file_result);
86  in.close();
```

Listing 3.13: cpp/main.cpp

Nach erfolgreichem Parsen nimmt die Funktion `validate` (Listing C.2, Zeile 104) die gleiche Validierung der eingelesenen Records vor, wie bereits für die Haskell Implementierung in Abschnitt 3.2.1 beschrieben. Ein zentraler Bestandteil ist dabei die Funktion `std::remove_if`, welche in etwa Haskells `filter` Funktion entspricht. Sie durchläuft eine Datenstruktur mithilfe eines Iterators und entfernt dabei alle Elemente, die ein bestimmtes Prädikat erfüllen, welches in Form einer Funktion des Typs `bool` übergeben wird. Ihr Rückgabewert wiederum ist ein Iterator, der auf das letzte Element der validen Elemente in der Sequenz zeigt. Die drei Prädikate zu entfernender Records entprechen den Funktionen `containsDuplicateFields`, `containsEmptyFields` und `containsNullValues`, ebenfalls definiert in der Datei `helpers.hpp`.

Die noch verbleibende Funktionalität in der *MAP* Phase ist das Entfernen nicht länger benötigter Attribute durch die `for` Schleife in Listing C.1 (Zeile 97-109). Dabei werden mittels der Funktionen `make_key` und `make_value` die relevanten Schlüssel- und Aggregationswerte der Records extrahiert und anschließend in eine Datenstruktur des Typs `std::map` eingefügt. Die Funktion `std::map::insert` fügt ein Schlüssel-Wert Paar in das entsprechende Map-Objekt ein und gibt erneut ein Paar, bestehend aus einem Iterator der Map und einem Wahrheitswert zurück. Ist dieser Wert `true`, wurde das Element erfolgreich eingefügt und der Iterator zeigt auf die entsprechende Position in der Map. Ist der Wahrheitswert jedoch falsch, so existiert bereits ein Element mit dem selben Schlüssel und das aktuelle Element wurde nicht eingefügt. Der Iterator zeigt in diesem Fall auf das schon vorhandene Element an dieser Position. Daraufhin muss der aktuelle Wert an die Werteliste des bereits vorhandenen Schlüssels angefügt werden (vgl. allgemeine Typendefinition in Abschnitt 3.1).

Somit wurden alle Teilfunktionen der logischen *MAP* Phase definiert: Syntaktische Analyse, Validierung und Typtransformation in einen „kleineren" Datensatz des Typs `std::map`. Die Applikation dieser Operationen, zusammengefasst in der Funktion `map_` (vgl. Listing C.1, Zeile 68-113), kann auch in C++ mithilfe einer Funktion höherer Ordnung erfolgen. Das Äquivalent zu Haskells `map`, inklusive einer Typumwandlung, ist die Funktion `std::transform` aus der Standard-Headerdatei `<algorithm>`. Sie wendet eine Operation auf sämtliche Elemente einer Datenstruktur an, speichert die Ergebnisse aber in einer neuen Variable. Dadurch kann die übergebene Funktion den Typ der Elemente verändern.

```
38 /* parse and validate records from files */
39 vector<aggregateMap>* dir_result =
40     new vector<aggregateMap>(file_list->size());
41 transform(file_list->begin(), file_list->end(),
42           dir_result->begin(), map_);
```

Listing 3.14: cpp/main.cpp

Die Applikation der Funktion innerhalb der `main` Methode des Programms ist in Listing 3.14 abgebildet. Es werden Beginn- und Ende-Iterator der zu durchlaufenden Datenstruktur, der Beginn-Iterator der Ergebnis-Struktur und die anzuwendenden Operation übergeben. Da `map_` einen Wert des Typs `aggregateMap`[18] zurückgibt, resultiert die mehrfache Anwendung auf einen Vektor von Eingangswerten (`file_list`) im Rückgabewert `vector<aggregateMap>`.

[18] `std::map<std::string, std::vector<std::pair<int,int> > >`

3.3.2 Group und Reduce Funktion

Auch das anschließende Zusammenfassen der Schlüssel-Wert Strukturen einzelner Dateien wird mithilfe einer Funktionen höherer Ordnung realisiert: `std::accumulate` aus dem Header `<numeric>` entspicht der bereits bekannten Fold-Operation `foldl`, beschrieben in Abschnitt 2.3. Sie durchläuft eine Datenstruktur anhand zweier Iteratoren von Beginn bis Ende und wendet standardmäßig in jedem Zwischenschritt den binären Operator `(+)` auf den Akkumulator und das aktuelle Element an.

```
117 void group_(vector<aggregateMap>* input, aggregateMap* output)
118 {
119     /* neutral element for the left fold    */
120     aggregateMap empty = aggregateMap();
121     /* merge seperate file vectors together */
122     *output = accumulate(input->begin(), input->end(), empty);
123 }
```

Listing 3.15: cpp/main.cpp

In diesem Fall jedoch soll der binäre Operator `(+)` zwei Instanzen von `std::map` zusammenfügen. Dies lässt sich realisieren, indem der Operator für den Typ `aggregateMap` neu definiert, bzw. „überladen" wird, wie dies im Modul `types.hpp` geschehen ist (Listing C.3, Zeile 80-93). Ähnlich wie bei der Haskell Version, werden die Schlüssel-Wert Strukturen hierfür zunächst in sortierte Vektoren ihrer Elemente umgewandelt. Dies funktioniert nur, weil auch `std::map` intern als (sortierter) Binärbaum realisiert ist (vgl. Stroustrup und Langenau 2015, S.109). Die Hilfsfunktion `mergeVectors` führt anschließend das eigentliche Zusammenfügen der Elemente mit linearer Komplexität aus, indem es die jeweils ersten Elemente der Vektoren miteinander vergleicht und das kleinste von ihnen in die neue Datenstruktur einfügt. Abschließend wird aus dem Vektor aller Elemente erneut eine Schlüssel-Wert Struktur geschaffen.

Der dritte Parameter von `accumulate`, der neutrale Akkumulator für das Zusammenfügen zweier Maps und Ausgangswert des Fold ist die leere Map `empty` (vgl. Listing 3.16, Zeile 118). Das Ergebnis der `group_` Funktion ist erneut eine einzelne Schlüssel-Wert Struktur, welche die Schlüssel-Wert Paare sämtlicher Dateiinhalte enthält.

Die `reduce_` Funktion in Listing 3.16 nimmt eines dieser Schlüssel-Wert Paare entgegen und summiert die Liste der enthaltenen Integer-Paare miteinander. Für die Summation wird auch hier die Funktion `accumulate` in Verbindung mit einem überladenen `(+)` Operator für den Typ `std::pair<int,int>` benutzt (vgl. Listing C.3, Zeile 75-78). Dadurch werden wieder die jeweils ersten und zweiten Elemente der übergebenen Paare miteinander addiert.

```
125 pair<string,intPair> reduce_(aggregatePair input)
126 {
127     /* neutral element for the left fold  */
128     intPair init = pair<int, int>(0,0);
129     /* sum up integer pairs in the vector */
130     return make_pair(input.first,
131         accumulate(input.second.begin(), input.second.end(), init));
132 }
```

Listing 3.16: cpp/main.cpp

Das neutrale Element und dritter Parameter von `accumulate` ist in diesem Fall ein Tupel aus Null-Werten. Die Anwendung der Funktion auf alle Elemente der globalen Schlüssel-Wert Struktur erfolgt, wie auch bei dem im Abschnitt 3.3.1 beschriebenen `map_`,

mittels `std::transform`, indem die Map vom Beginn- bis zum Ende-Iterator durchlaufen wird. Ergebnis der Aggregationsphase ist ein Vektor mit Paaren bestehend aus Schlüssel und einem weiteren Paar des Dauer- und Volumenaggregats[19].

3.3.3 Parallelisierung

Im Abschnitt 1.2 wurden bereits mehrere Möglichkeiten der Parallelisierung von C++ Programmen benannt. Nachfolgend soll nun evaluiert werden, welche dieser Methoden für den MapReduce-Algorithmus am besten geeignet ist.

Aufgrund der technischen Gegebenheiten wurde festgelegt, dass die Implementierung auf Rechner mit physikalisch zusammenhängendem Speicher abzielen soll. Auch wenn die Prozesskommunikation mittels Message Passing in den allermeisten Fällen auf verteilten Systemen verwendet wird, so ist dies allein noch kein Ausschlusskriterium für den Einsatz des MPI Standards. Einer der wichtigsten Vorteile nachrichtenbasierter Programme ist tatsächlich, dass sie ebenso auf *Shared Memory Machines (SMMs)* laufen können und damit weit weniger abhängig von der zugrundeliegenden Architektur sind als Programme, die auf OpenMP setzen.

Dennoch ist diese Form des Realisierung alles andere als ideal für das bestehende Programm: denkbar wäre bspw. in den *MAP* und *REDUCE* Phasen die Daten explizit an verschiedene Prozessorkerne der Maschine zu senden und dort die Verarbeitung parallel durchzuführen. Für die teils sequenzielle Gruppierung müssten die Daten zwischenzeitlich jedoch wieder an eine einzelne Recheneinheit geschickt werden. Es ist offensichtlich, dass die Verwendung einer solchen Semantik, basierend auf dem Versenden und Empfangen und Nachrichten, auf einem monolithischen System zu einer unnötig erhöhten Komplexität führen würde. Da in diesem Szenario ein geteilter Speicherbereich vorhanden ist, soll dieser auch als solcher genutzt werden. Zudem würde die Umstellung auf MPI einen tiefgreifenden Eingriff in die bestehende Programmlogik bedeuten.

Bei Verwendung der Pthread-Bibliothek müsste ein solch explizites Versenden der Daten nicht erfolgen. Allerdings ist der Aufwand, das Programm manuell auf diese Weise zu parallelisieren immens, da die Steuerung der einzelnen Datenflüsse sowie die Koordination der Threads im Programmcode beschrieben werden muss. Es geht auch einfacher.

OpenMP erleichtert die Optimierung bestehenden Programmcodes erheblich, indem parallele Regionen des Quellcodes durch spezielle Compiler-Direktiven gekennzeichnet werden können. Die Erstellung und Koordination der Threads sowie die Verteilung der Daten wird daraufhin komplett von den Bibliotheksfunktionen, bzw. dem Compiler übernommen. Die grundlegende Methodik, mit der OpenMP dabei Software parallelisiert, wird als „Fork/Join Modell" bezeichnet. Sobald der Kontrollfluss eines Programms auf eine parallele Region im Quellcode trifft, teilt er sich auf mehrere Threads auf (fork). Der speziell annotierte Bereich wird daraufhin von mehreren dieser leichtgewichtigen Prozesse gleichzeitig ausgeführt. Sie alle bearbeiten dieselbe Programmlogik, es handelt sich also um SIMD-Parallelismus (vgl. Abschitt 2.1). Nach Ausführung dieser Sektion werden die zusätzlich erstellten Threads beendet und der Kontrollfluss läuft wieder in einem Thread zusammen (join). Diese Vorgehensweise ist in Abbildung 3.4 noch einmal bildlich dargestellt.

Eine der mit OpenMP am einfachsten zu nutzenden Optimierungsmöglichkeiten ist die sogenannte Loop-Level Parallelisierung, bei der die einzelnen Iterationen einer Schleife gleichzeitig von mehreren Prozessoren behandelt werden. Um diese in der vorliegenden

[19] `std::vector<std::pair<string, std::pair<int, int> > >`

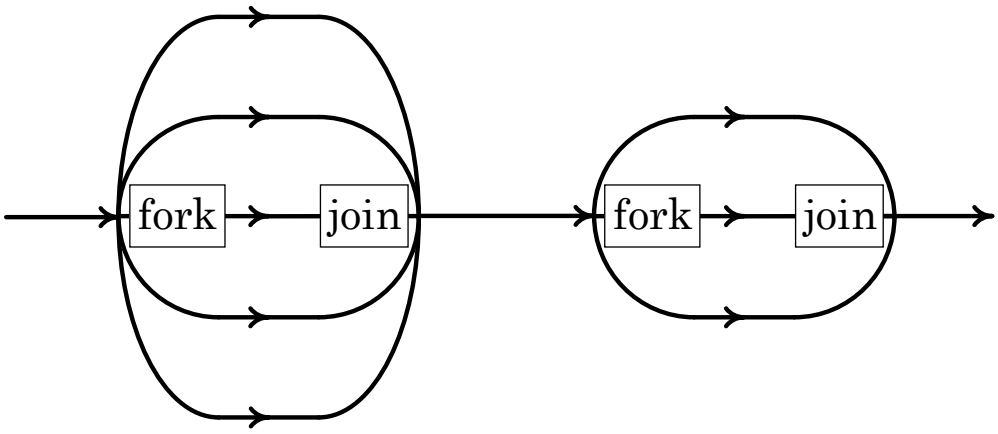

Abbildung 3.4: Parallelisierung nach dem Fork/Join Modell

Implementierung nutzen zu können, wäre es möglich die verwendeten Funktionen höherer Ordnung, wie `transform` und `accumulate`, in Schleifen umzuschreiben. Anschließend kann durch die beiden Konstrukte `parallel` und `for` die parallele Codeausführung beschrieben werden. Eine solche parallele `for` Schleife ist in Listing 3.17 beispielhaft dargestellt.

```
1 #pragma omp parallel for
2 for ( ... ; ... ; ... ) {
3     /* ... */
4 }
```

Listing 3.17: Parallelisierung einer `for` Schleife mit OpenMP

Jede OpenMP Anweisung im Quellcode eines C/C++ Programms beginnt standardmäßig mit dem Ausdruck `#pragma omp`, was fähigen Compilern signalisiert, dass der Rest der aktuellen Zeile aus OpenMP Direktiven besteht. Das Schlüsselwort `parallel` sorgt dafür, dass für den darauffolgenden Codeblock eine gewisse Menge von Threads erstellt wird. Es ist möglich eine gewünschte Anzahl mit der Funktion `num_threads`, bzw. der Umgebungsvariable `OMP_NUM_THREADS` zu definieren. Ob diese Zahl anschließend auch tatsächlich benutzt wird, hängt noch von zusätzlichen Faktoren ab, wie den – eventuell begrenzten – gesamt zur Verfügung stehenden Threads. Der komplette Algorithmus zur Bestimmung der Threadanzahl ist in der API-Spezifikation (OpenMP Architecture Review Board 2015, S.50f) beschrieben. Die Menge der erstellten leichtgewichtigen Prozesse wird innerhalb der parallelen Programmregion als „Team" bezeichnet. Die `for` Anweisung in Zeile 1 teilt anschließend die Iterationen einer Schleife auf das aktuelle Team auf. Hierfür wird vor Eintritt in die Schleife die Anzahl der Iterationen berechnet.

Das Konzept der Loop-Level Parallelisierung würde zumindest für die *MAP* und *REDUCE* Phase vollkommen ausreichen um einen adäquaten Grad der Parallelisierung zu erreichen. Allerdings besteht hierfür, wie bereits angemerkt, der Bedarf die bisher benutzten Funktionen höherer Ordnung in `for` Schleifen umzuschreiben. Damit würden sich auch die Typen der `map_` und `reduce_` Funktion, wie in Abschnitt 3.1 beschrieben, ändern, da sie nicht länger auf einzelne Elementen der Liste bzw. Schlüssel-Wert Struktur angewendet würden sondern diese als Ganzes entgegennähmen.

Die parallele `group_` Funktion hat eine etwas andere, mehrstufige Struktur (vgl. Abbildung 3.2). Eine auf Schleifen basierende Funktion müsste die Liste der zu vereinigenden Schlüssel-Wert Strukturen zunächst statisch partitionieren und anschließend zusammenfügen. Dieser Arbeitsschritt wird für die daraus resultierenden Listen solange wiederholt,

bis nur noch zwei Map-Elemente übrig sind. Das Zusammenfügen der beiden letzten Listen wird dann erneut von einer einzelnen CPU ausgeführt.

Auch wenn die Parallelisierung mit OpenMP selbst also recht einfach umzusetzen ist, muss der Programmcode dafür bestimmte Formfaktoren erfüllen. Idealerweise sollte sich die parallele Natur der `transform` und `accumulate` Funktionen allerdings einfacher nutzen lassen.

Tatsächlich gibt es noch eine weitere, simplere Methode zur Optimierung des vorliegenden MapReduce Programms. Die offensichtliche Möglichkeit zur Parallelisierung bei Anwendung der gleichen Funktion auf eine Menge von Daten, wie dies bei vielen Funktionen höherer Ordnung der Fall ist, hat zur Entstehung einer parallelen Version der betreffenden Standard-Bibliothek geführt. Ihre Entwickler, Singler, Sanders und Putze (2007), sprechen gar von „beschämend parallelen"[20] Funktionen.

Technisch basiert diese Umsetzung ebenfalls auf dem zuvor beschriebenen OpenMP Standard, verbirgt dessen `pragma` Anweisungen allerdings komplett vor den Nutzenden. Dieser verwendet lediglich die gleichnamigen Funktionen aus dem `std::__parallel` Namensbereich, so wie in Listing 3.18 dargestellt.

Die einzig notwendigen Änderungen am bestehenden Programmcode, um *REDUCE* und *MAP* Phase zu parallelisieren, sind das Einbinden der parallelen `algorithm` und `numeric` Header sowie das Ändern des Namensbereichs der `transform` Aufrufe:

```
5  #include <parallel/algorithm>   /* parallel transform   */
6  #include <parallel/numeric>     /* parallel accumulate  */
   /* ... */
12 namespace par = std::__parallel;
   /* ... */
39 vector<aggregateMap>* dir_result =
40     new vector<aggregateMap>(file_list->size());
41 par::transform(file_list->begin(), file_list->end(),
42                dir_result->begin(), map_);
   /* ... */
51 vector<pair<string,intPair> >* result =
52     new vector<pair<string,intPair> >(groups->size());
53 par::transform(groups->begin(), groups->end(),
54                result->begin(), reduce_);
```

Listing 3.18: cpp/main.cpp

Auf dieselbe Weise kann auch die Parallelisierung von `accumulate`, innerhalb der Funktion `group_`, erfolgen. Da die komplette Koordination des Kontrollflusses den Bibliotheksfunktionen überlassen wird, lässt sich keine genaue Aussage darüber treffen, wie das Zusammenfügen der Map-Strukturen tatsächlich parallelisiert wird. Im besten Fall geschieht es nach der in Abbildung 3.2 aufgezeigten Methode. Die wesentlich einfachere Programmierung wird also auf Kosten einer weniger detaillierten Kontrolle über die Ausführung des Algorithmus erreicht.

Des Weiteren nehmen bei Verwendung der parallelen Standard-Bibliothek auch gewisse quantitative Faktoren, wie Problemgröße und verwendete Datentypen, Einfluss auf die Ausführung des Algorithmus. Dadurch soll die Parallelisierung des Programms erst dann einsetzen, wenn sich der entstehende Mehraufwand auch tatsächlich positiv auf die Laufzeit auswirkt. (Vgl. Singler und Konsik 2008, S.8).

[20] „Embarrassingly parallel" (Singler, Sanders und Putze 2007, S.1)

Es ist anzumerken, dass dieser sogenannte *parallel mode* laut GNU Benutzerhandbuch noch immer den Status „experimental" hat (vgl. Free Software Foundation 2016, Kapitel 18 Parallel Mode). Da aber bei durchgeführten Softwaretests während der Implementierung keinerlei Abweichungen der Ergebnisse zu den sequenziellen Programmen festgestellt wurden, wird davon ausgegangen, dass die Funktionen ein korrektes Verhalten aufweisen.

Kapitel 4

Leistungsmessung und Evaluierung

> *Dieses Kapitel erläutert die technischen Gegebenheiten sowie das Vorgehen beim Test beider Implementierungen. Im Anschluss werden die erhobenen Messdaten der zuvor definierten Leistungskriterien vorgestellt.*

4.1 Erhebung der Messdaten

Beim Test der Haskell-Implementierung auf dem initial vorgesehenen Testsystem kann ein generell sehr hoher Speicherverbrauch festgestellt werden – so hoch, dass es die zur Verfügung stehenden Ressourcen bei weitem übersteigt. Daher wird die Entscheidung getroffen, die Untersuchungen in zwei Testreihen aufzuteilen: Eine Testreihe mit geringer Problemgröße (100 – 1.000 Dateien) wird auf dem ursprünglichen Testsystem, einem herkömmlichen Desktop-Rechner, durchgeführt. Die Verarbeitung einer größeren Zahl von Eingangsdaten (1.000 – 10.000 Dateien) wird hingegen separat, auf einem Server in Amazons *Elastic Compute Cloud (EC2)* getestet. Die Unterschiede der zugrundeliegenden Hardware schließen zwar allgemeine Aussagen über beide Testreihen aus, jedoch erlaubt nur dieses Vorgehen die Bewertung der Programme bei geplanten Problemgrößen von bis zu einer Million Records (vgl. Abschnitt 2.4).

Es wird stets darauf geachtet, möglichst viele unnötge Hintergrundprozesse vor dem Test der Programme zu beenden. Auflistungen aller noch laufenden Prozesse sind im elektronischen Anhang zu finden (`test/running_procs_desktop.txt` und `test/running_procs_ec2.txt`). Auf dem Desktop-System werden dabei unvermeidlich mehr Prozesse ausgeführt, da anders als beim EC2-Server eine grafische Benutzeroberfläche geboten wird.

Technische Voraussetzungen

Als zentrale Recheneinheit kommen in beiden Systemen Intel-Prozessoren mit *Hyper-Threading (HT)* (Marr et al. 2002) zum Einsatz. Diese Technologie erlaubt es mehreren logischen Prozessoren sich physikalische Ressourcen, wie das Rechenwerk, Caches, und Systembusse, zu teilen, während jeder von ihnen über einen eigenen Zustand und eigene Register verfügt (vgl. Intel Corporation 2015, S.44). Dabei bilden jeweils zwei logische CPUs einen „Core" mit dedizierter Hardware.

In den verwendeten Prozessor des Desktop-Rechners sind vier dieser Kerne integriert. Aus der Sicht des Betriebssystems sieht es daher so aus als stünden insgesamt acht Re-

cheneinheiten zur Verfügung und acht Threads könnten gleichzeitig ausgeführt werden. Tatsächlich ist es so, dass die wenigen duplizierten Teile eines Kerns zwar für etwas mehr Parallelität, z.B. beim Holen und Dekodieren von Instruktionen aus dem Cache sorgen, die eigentliche Ausführung durch das Rechenwerk jedoch trotz Allem nicht gleichzeitig geschehen kann. Das durch Hyper-Threading erreichte Leistungsplus ist daher vorrangig darauf zurückzuführen, dass es oftmals zu einer besseren Auslastung des Prozessors führt (vgl. Ding, Bolker und Kumar 2003, S.1f).

Als Referenz wird diese Zahl der logischen Prozessoren angenommen und beide Programme deshalb mit maximal acht Threads zur Ausführungszeit getestet. Sämtliche Berechnungen aus Abschnitt 2.4, deren Formel die Anzahl der Prozessoren P beinhalten, werden mit der Zahl logischer CPUs angestellt.

Weitere technische Merkmale des Desktop-Testsystem sind in der nachfolgenden Tabelle 4.1 aufgeführt.

Merkmal	**Ausstattung des Testsystems**
Betriebssystem	CentOS Linux 7.1
Prozessor	Intel Core i7-4770 @3,40GHz, 4 Kerne mit je 2 logischen CPUs
Hauptspeicher	16GB DDR3 @1600MHz
Festplatte	HDD (Leserate 157,43MB/s)[21]

Tabelle 4.1: Relevante Kennzahlen des Desktop-Testsystems

Die Voraussetzungen auf Amazons EC2-Infrastruktur weichen insofern von obigem Desktop-Testsystem ab, als das die gemieteten Rechner standardmäßig virtuelle Maschinen sind, die auf geteilter Hardware laufen. Für sie werden lediglich logische CPUs, also eine festgelegte Zahl von Hyper-Threads, reserviert. Daher lässt sich nicht mit Sicherheit sagen, wie viele tatsächliche Prozessoren für die Ausführung eines Programms genutzt werden.

Ein weiterer Nachteil eines solchen virtualisierten Systems ist, dass bestimmte Leistungsdaten nicht erfasst werden können. So erlaubt das Programm `perf` in dieser Testkonfiguration nicht die Messung der beschriebenen Cache-Informationen (Gesamtzahl der Anfragen und Cache-Misses). Eine Bewertung der Cache Hit-Rate entfällt daher an dieser Stelle.

Die technischen Kennzahlen des EC2-Servers sind in Tabelle 4.2 aufgelistet.

Merkmal	**Ausstattung des Testsystems**
Betriebssystem	RedHat Enterprise Linux 7.2 (Hardware Virtual Machine)
Prozessor	Intel Xeon E5-2676 v3 @2,40GHz, 12 Kerne mit je 2 logischen CPUs Davon stehen 8 logische CPUs zur Verfügung.
Hauptspeicher	61GB[22]
Festplatte	SSD (Leserate 167,772MB/s)[23]

Tabelle 4.2: Relevante Kennzahlen des EC2-Testsystems

[21] Leserate („buffered disk reads") gemessen mit dem Linux Programm `hdparm`.

Kompilierung und Ausführung der Software

Haskell

Die Kompilierung der Haskell Software erfolgt mit dem Glasgow Haskell Compiler (GHC) in der Version 8.0.1. Um die Ausführung des Programms mit mehreren Betriebssystemthreads zu aktivieren, muss dabei der Kommandozeilenparameter `-threaded` verwendet werden.

Zur Steuerung der Laufzeitumgebung wird außerdem die Option `-rtsopts` genutzt. Diese erlaubt unter anderem die Zahl der zu verwendenden Betriebssystemthreads beim Aufruf des Haskell Programms zu spezifizieren.

Der Optimierungsgrad, welchen der Compiler beim Erstellen des Programms anwenden soll, kann mit der Option `-Ox` gesteuert werden. Wie bereits in Abschnitt 2.4 dargelegt, entspricht `x` in den folgenden Untersuchungen einer Ganzzahl von 0 bis 2.

Ein möglicher Aufruf des Compilers zur Erstellung eines Programms namens „MapReduce" sieht demnach wie folgt aus:

```
ghc -O0 -rtsopts -threaded -o MapReduce Main.hs
```

Listing 4.1: Kompilieren des parallelen Haskell Programms ohne Optimierungen

Für Messungen bezüglich des Cache-Speichers sowie der Antwortzeit wird das MapReduce Programm im Kontext von `perf stat` aufgerufen. Mit der Option `-e` werden dabei die zu erfassenden Events angegeben. Sowohl die Gesamtzahl der Cacheanfragen (`cache-references`) als auch die aufgetretenen Fehlschläge (`cache-misses`) werden dazu benötigt, um die Cache Hit-Rate bei Ausführung eines Programms zu erfassen.

Die Option `-rx` erlaubt unter Angabe einer Zahl `x` (≤ 100) wiederholte Messungen und die automatische Berechnung der Standardabweichung aller erfassten Werte. Sämtliche hier durchgeführten Testläufe werden fünf Mal wiederholt um einen repräsentativen Mittelwert zu erhalten.

Auch wenn der Wert nicht direkt in die Bewertung einfließt, soll dennoch die CPU-Zeit des Prozesses (`task-clock`) erfasst werden. Denn damit einher geht die Ausgabe der Prozessornutzung in Form einer Gleitkommazahl durch `perf stat`. Dieser Wert gibt Aufschluss darüber, wie viele der zur Verfügung stehenden Rechenressourcen das Programm tatsächlich belegt hat. Im Hinblick auf eine maximale Auslastung des Systems sollte die Zahl der tatsächlich genutzten Prozessoren nah an der Zahl erstellter Threads sein.

Ein möglicher Aufruf des Programms durch `perf stat` ist in Listing 4.2 dargestellt. Die Messung der Antwortzeit muss dabei nicht explizit aktiviert werden.

```
perf stat -r 5 -e cache-references,cache-misses,task-clock ./MapReduce
    ../test/files +RTS -N8
```

Listing 4.2: Fünffache Messung von Cache-Anfragen, Cache-Misses, CPU-Zeit und Antwortzeit des Haskell Programms unter Verwendung von 8 Threads

Ähnlich funktioniert auch die Messung der durchschnittlichen Resident Set Size (RSS). Hierfür wird das MapReduce-Programm mit dem Shell-Skript `memc.sh` aufgerufen, welches im Abschnitt 2.4 vorgestellt wurde:

```
./memc.sh ../haskell/MapReduce "../test/files +RTS -N8" 0.5
```

Listing 4.3: Erfassen der RSS mit 0,5 Sekunden Verzögerung zwischen den Messungen

[22] Keine genauen Angaben zur Speichertechnologie oder Taktrate verfügbar.

[23] Amazon gibt für seine „General Purpose SSD" (gp2) einen Durchsatz von 160 MiB/s an. (docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html)

C++

Das C++ Programm wird mit dem GCC erstellt. Auf dem Desktop-Rechner ist dieser in der Version 6.1 verfügbar, auf dem EC2-Server wird die Version 4.8.5 genutzt.

Einige der genutzten Sprachkonstrukte sind erst ab dem C++11 Standard verfügbar. Dieser wird mit der Option `-std=c++11` aktiviert. Da die parallele Version des Algorithmus, wie in Abschnitt 3.3.3 beschrieben, den *GNU parallel Mode* benutzt, muss die Kompilierung zwingend mit dem Kommandozeilenparameter `-fopenmp` erfolgen. Dieser signalisiert dem Compiler, dass in der Linker-Phase die OpenMP Bibliotheken einzubinden sind:

```
g++ -std=c++11 -fopenmp -O0 -o MapReduce main.cpp
```

Listing 4.4: Kompilieren des parallelen C++ Programms ohne Optimierungen

Die Messung der Leistungskennzahlen erfolgt ähnlich wie bei dem zuvor beschriebenen Haskell Programm. Lediglich die Anzahl der zu erstellenden Threads werden in diesem Fall nicht über einen Kommandozeilenparameter sondern über die Umgebungsvariable `OMP_NUM_THREADS` gesteuert. Listing 4.5 zeigt ihre Definition unter Benutzung des Bourne-Again Shells (Bash) durch den `export` Befehl.

```
export OMP_NUM_THREADS=8

perf stat -e cache-references,cache-misses,task-clock ./MapReduce ../
    test/files

./memc.sh ../cpp/MapReduce ../test/files .5
```

Listing 4.5: Beispielhafte Messung des C++ Programms mit 8 Threads

Die beschriebenen Aufrufe der Programme sowie das Einfügen der Messergebnisse in Datenbanken zu späteren Auswertung erfolgt mithilfe von Perl-Skripten, welche im elektronischen Anhang zu finden sind (`measure_desktop.pl` bzw. `measure_ec2.pl`).

Testdaten

Es werden Mengen von zufälligen Testdaten, generiert durch das im Abschnitt 2.4 beschriebene Perl-Skript, verarbeitet. Für beide Testreihen wird dabei die gleiche Konfiguration der Aggregationsschlüssel genutzt: ein Aggregationsdatum (`-a`), eine Plattform (`-p`), zwei Eventklassen (`-c`), zwei Eventsources (`-s`), zwei Tarifzonen (`-z`). Daraus ergeben sich $1 \cdot 1 \cdot 2 \cdot 2 \cdot 2 = 8$ zu aggregierende Gruppen, so dass bei maximaler Threadanzahl sämtliche logische CPUs beider Testmaschinen während der *REDUCE* Phase optimal ausgelastet sein sollten.

4.2 Präsentation der Ergebnisse

Sämtliche gemessenen Werte sind in SQLite Datenbanken gespeichert, welche im elektronischen Anhang zu finden (`test/test_results_desktop.db` und `test/test_results_ec2.db`) und deren Schemata in den Tabellen A.1 und A.2 beschrieben sind. Die folgenden Darstellungen wurden mittels der ebenfalls beiliegenden R-Skripte erstellt und die präsentierten Werte auf jeweils zwei Nachkommastellen gerundet. Soweit nicht anders angegeben, handelt es sich um Durchschnittswerte bei denen nur die explizit dargestellten Testbedingungen festgesetzt wurden.

Desktop-Testsystem

Einzelbewertungen

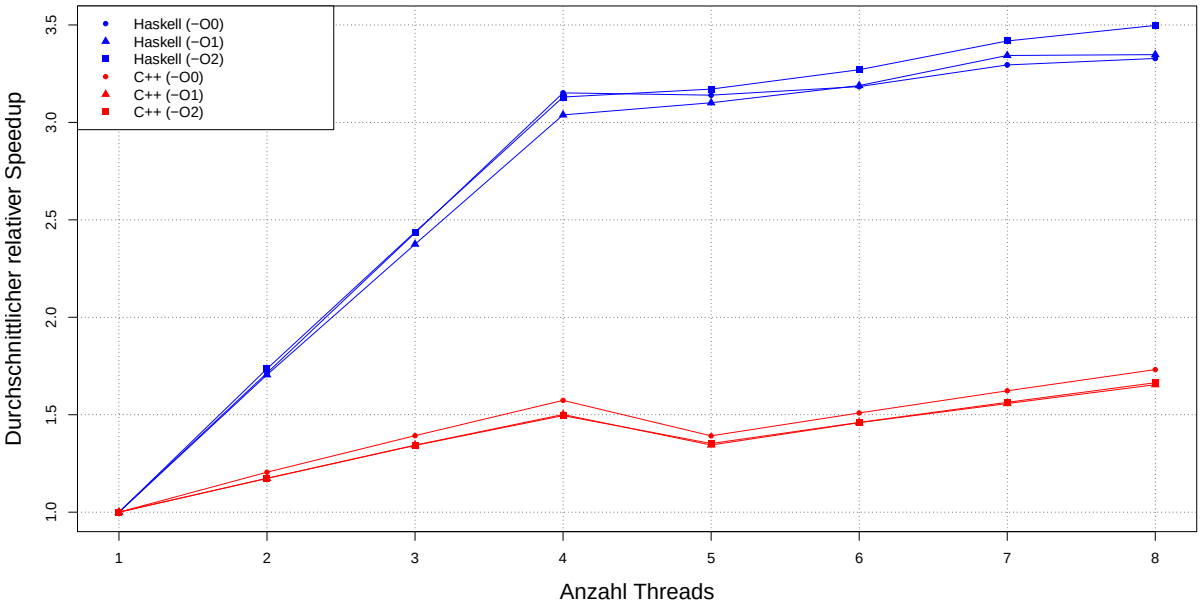

Abbildung 4.1: Durchschnittlicher relativer Speedup

In der obigen Abbildung 4.1 ist der durchschnittliche Speedup beider Programme mit verschiedenen Optimierungsgraden dargestellt. Es ist zu erkennen, dass die Haskell-Implementierung beim Einsatz mehrerer Threads mit einem maximalen Wert von 3,50 (`-O2`) deutlich besser skaliert als die C++ Variante. Diese erreicht ihr Maximum von 1,73 beim Einsatz von acht Threads ohne Optimierungen (`-O0`).

Beide Programme bleiben somit weit hinter ihrem theoretisch möglichen Speedup nach Amdahls Gesetz zurück. Dieser kann durch Messung der Laufzeiten potenziell paralleler und zwangsläufig sequenzieller Programmteile bestimmt werden. Im Anhang A (Theoretischer Speedup) wird dieser Wert beispielhaft für eine Problemgröße von 1.000 Dateien mit je 100 Records berechnet. Die Ergebnisse suggerieren, dass beide Programme zumindest nach dieser idealisierten Betrachtungsweise durchaus Potenzial für höhere Speedup-Werte aufweisen. Unter realen Bedingungen kommt jedoch ein gewisser paralleler Overhead hinzu, sodass die berechneten Werte für keine Prozessoranzahl P erreicht werden.

Warum die Werte beider Programme so immens voneinander abweichen erklärt eine erneute Betrachtung des relativen Speedups unter Berücksichtigung der Dateianzahl. In den Abbildungen 4.2 und 4.3 ist dieser für Problemgrößen unterhalb von 1.000 Dateien, bzw. exakt 1.000 Dateien dargestellt. Das C++ Programm hat bis einschließlich 750 Dateien

einen konstanten Speedup von 1,0, während sein Haskell Äquivalent bei jeder Problemgröße positive Skalierungseffekte zeigt. Für 1.000 Dateien hingegen übersteigt der relative Speedup der C++ Variante den von Haskell in einem Großteil der Messungen sogar deutlich und erreicht einen Maximalwert von 4,65 (`-O2`). Dieses Verhalten ist zurückzuführen auf die im Abschnitt 3.3.3 erwähnte Heuristik des *GNU parallel mode*, bei der auch die Problemgröße Einfluss auf die tatsächliche Ausführung hat. In der verwendeten Testreihe wird die Arbeit während der *MAP* Phase des Algorithmus erst ab 1.000 Dateien auf mehrere Threads aufgeteilt. Die bei den Messungen erreichte CPU-Auslastung, zu sehen in den Abbildungen D.1 und D.2, bestätigt diese Annahme.

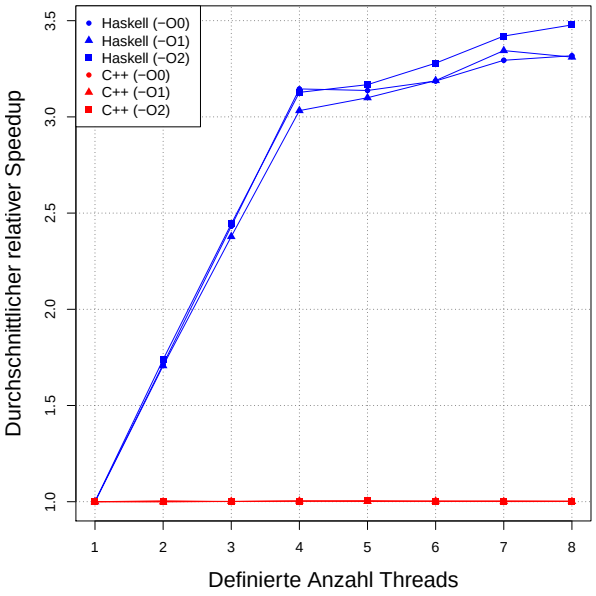

Abbildung 4.2: Durchschnittlicher relativer Speedup bei Verarbeitung von 100, 250, 500 und 750 Dateien (Desktop)

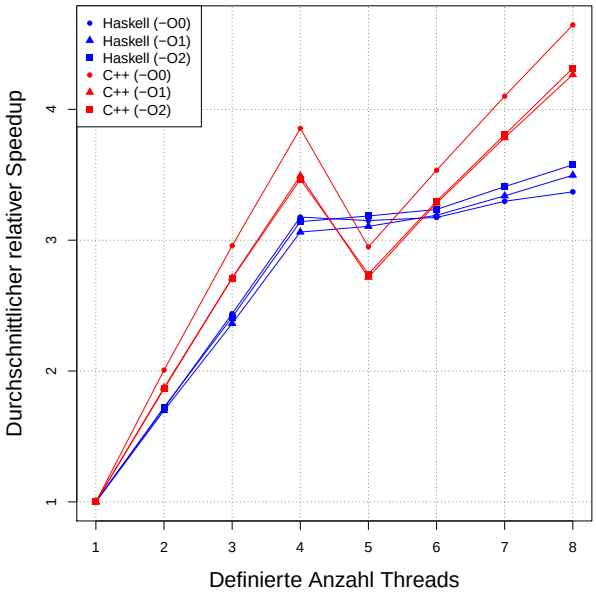

Abbildung 4.3: Durchschnittlicher relativer Speedup bei Verarbeitung von 1.000 Dateien (Desktop)

Der Speedup beider Programme bei Verarbeitung von 1.000 Dateien zeigt eine stark veränderte Charakteristik ab fünf verwendeten Threads auf. Wie im Abschnitt 4.1 dargelegt, verfügt das Testsystem über lediglich vier vollständig in Hardware ausgeprägte CPU-Kerne. Daher ist es nicht verwunderlich, dass der Speedup darüber hinaus abflacht, bzw. im Fall von C++ zunächst sogar etwas zurückgeht (Abbildung 4.3). Es ist jedoch erkennbar, dass das parallele C++ Programm deutlich besser von der Hyper-Threading Technologie profitiert. Wird die mittlere Differenz zwischen den Speedups bei acht und bei vier verwendeten Threads gebildet, so ergibt sich für C++ ein Zuwachs von 0,80. Die Haskell Version dagegen kann ihren Speedup durch die zusätzlichen logischen Prozessoren nur um durchschnittlich 0,35 steigern.

Noch deutlicher wird dieser Effekt bei Betrachtung der Effizienz für die Verarbeitung von 1.000 Dateien, dargestellt in Abbildung D.4. Während der Wert für beide Programme bei fünf Threads stark abfällt, bleibt er darüber hinaus für C++ etwa konstant bei 0,59 (`-O0`) bzw. 0,54/0,55 (`-O1`/`-O2`). Die Effizienz der Haskell Implementierung fällt in diesem Bereich wesentlich steiler bis auf einen Minimalwert von 0,42 (`-O0`) ab.

Dementsprechend verläuft auch die Kurve des parallelen Overheads in Abbildung D.8, welcher sich reziprok zur Effizienz der Programme verhält: je größer der Mehraufwand durch parallele Codeausführung, desto geringer die Effizienz des Programms.

Für Problemgrößen unterhalb von 1.000 Dateien ergibt sich für C++ hinsichtlich der

Effizienz und des parallelen Overheads ein anderes Bild. Da hier keinerlei Parallelisierung stattfindet, ist die Effizienz des Programms konstant 1,00 und paralleler Mehraufwand nicht vorhanden (vgl. Abbildungen D.3 und D.7).

Dass die unterschiedlichen Optimierungsgrade des Compilers einen erheblichen Einfluss auf die Laufzeit der Programme haben, verdeutlicht eine Untersuchung der Kosten, also das Produkt aus Antwortzeit und genutzten Prozessoren. Da diese Berechnung für das C++ Programm unterhalb vom 1.000 Dateien aufgrund sequenzieller Ausführung wenig aussagekräftig ist, wurden in Abbildung D.6 lediglich Problemgrößen mit 1.000 Dateien einbezogen.

Zwischen der nicht optimierten C++ Version (`-O0`) und der mit einfachen Optimierungen (`-O1`) zeichnet sich im Schnitt eine Kostendifferenz von 40,73 Sekunden ab. Beim Haskell Programm ist dieser Wert mit 4,52 Sekunden zwar weniger gravierend, aber dennoch deutlich. Der Leistungsgewinn durch den Optimierungsgrad `-O2` ist im Vergleich dazu marginal – die Kosten des C++ Programms sinken nochmals um 0,11 Sekunden. Haskell ist hierbei sogar um 0,12 Sekunden schlechter, als mit der Compiler-Option `-O1`.

Die Cache Hit-Raten der Implementierungen liegen beim überwiegenden Teil der Messungen oberhalb von 90%. Das C++ Programm ohne Optimierungen erreicht im Schnitt sogar eine Hit-Rate von 99,25%.

Bei beiden Programmiersprachen ist eine leichte Verschlechterung des Wertes unter Einsatz von Compiler-Optimierungen zu beobachten. So beträgt die mittlere Differenz der Hit-Rate zwischen `-O0` und `-O1` Version bei C++ 1,04% sowie 0,81% bei Haskell.

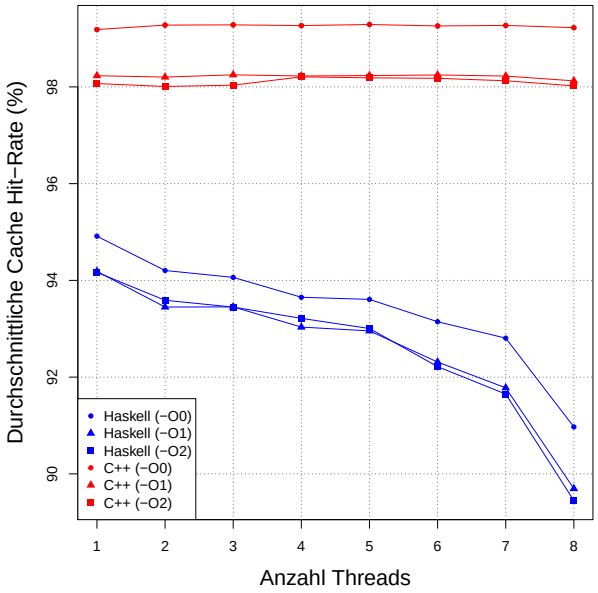

Abbildung 4.4: Durchschnittliche Cache Hit-Rate

Auffällig ist die negative Korrelation der Cache Hit-Rate mit der Zahl genutzter Prozessoren beim Haskell Programm. Unter Verwendung von acht Threads sinkt die Hit-Rate um durchschnittlich 4,38% im Vergleich zur sequenziellen Ausführung, während bei C++ keine auffällige Veränderung zu erkennen ist. Unter den technischen Gegebenheiten ist jedoch die Entwicklung der Werte für C++ oberhalb von vier Threads besonders bemerkenswert. Wie im vorangegangenen Abschnitt 4.1 erläutert, ist der Cache-Speicher einer der Prozessor-Bestandteile, die lediglich einmal pro CPU-Kern ausgeprägt sind. Unter Einsatz von Hyper-Threading teilen sich diese Komponente also zwei CPUs. Umso erstaunlicher ist deshalb, dass eine solche „Doppelbelegung" keinen gravierenden Einfluss auf die Hit-Rate des C++ Programms hat[24].

Die Arbeitsspeicherbelegung beider Programme unterscheidet sich besonders stark voneinander. In Darstellung 4.5 wurde eine logarithmische Einteilung für die Ordinatenachse gewählt, da die durchschnittliche RSS der Haskell Implementierung, die von C++ bei weitem übersteigt.

[24]Dieser Effekt tritt unabhängig davon auf, ob tatsächlich eine Parallelverarbeitung stattfindet oder nicht. In Abbildung D.13 ist die Hit-Rate bei Verarbeitung von Problemgrößen mit 1.000 dargestellt.

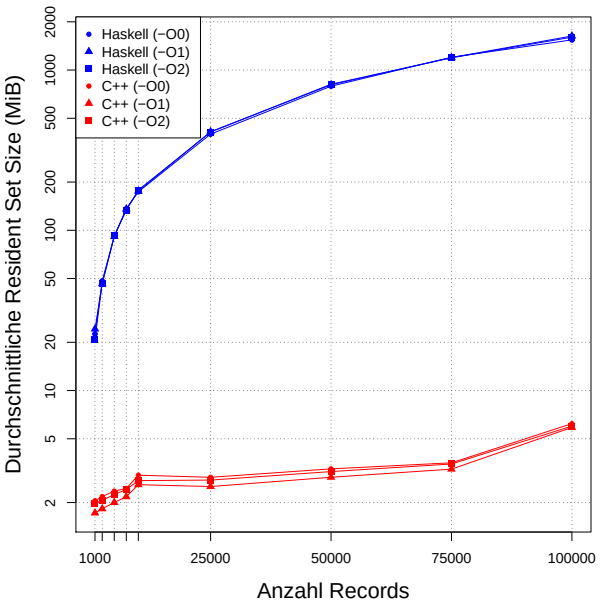

Abbildung 4.5: Durchschnittliche RSS

So belegt das Programm schon bei der geringsten Problemgröße von 1.000 Records mindestens 20,93MiB im Hauptspeicher der Maschine (`-O2`). C++ erreicht einen Minimalwert von gerade einmal 1,72MiB (`-O1`). Während die Verarbeitung von 100.000 Records bis zu 1.630,65MiB bei Haskell erfordert (`-O1`), kommt das C++ Programm mit maximal 6,22MiB aus (`-O0`).

Unterschiede solcher Größenordnungen haben einen gravierenden Effekt auf die Praktikabilität des Programms. Der verfügbare Hauptspeicher ist für die Haskell Implementierung ganz klar ein begrenzender Faktor. Über den Verlauf der Messreihe nimmt die Speicherbelegung des Prozesses um durchschnittlich 16,09KiB pro Record zu. Beim C++ Programm entspricht dieser Wert lediglich 0,04KiB. Es ist somit im Hinblick auf die Speichernutzung wesentlich effizienter.

Der mittlere Durchsatz bei Verarbeitung von 1.000 Dateien, dargestellt in den Abbildungen D.9 und D.10, ist für die beiden optimierten Varianten des C++ Programms (`-O1`/`-O2`) deutlich höher als der von Haskell. Die Implementierung verarbeitet unabhängig von der Dateigröße unter Verwendung von acht Threads mehr als 40.000 Records pro Sekunde. Das rein funktionale Programm wiederum erreicht einen Maximalwert von etwas mehr als 14.000 Records/s – dies jedoch nur bei Dateien mit 100 Records. Damit ist seine Leistung beinahe doppelt so groß, wie die des C++ Programms ohne Compiler-Optimierungen, welches mit acht genutzten Threads im Schnitt 7.138,85 Records/s verarbeitet.

Des Weiteren liefern die Darstellungen des Durchsatzes eine ähnliche Aussage, wie der zuvor untersuchte relative Speedup: C++ ist in der Lage die Zahl verarbeiteter Records pro Sekunde oberhalb von vier Threads etwas mehr zu steigern als Haskell.

Die Diagramme D.11 und D.12 lassen erkennen, das die Verarbeitung von größeren Dateien mit jeweils 100 Records in beiden Programmiersprachen zu einem höheren Durchsatz führen. Dieser Effekt ist in Haskell jedoch stärker ausgeprägt als in C++. Im Schnitt verarbeitet das rein funktionale Programm bei größeren Eingangsdateien pro Sekunde 1159,83 Records mehr. Der Durchsatz der C++ Variante ist unter den gleichen Bedingungen um 381,28 Records/s höher als für Dateien mit jeweils 10 Records.

Gesamtbewertung

Die im Folgenden aufgestellte Gesamtbewertung soll einen Überblick schaffen, wie stark ausgeprägt die Unterschiede beider Implementierungen sind, werden alle sieben Leistungsmerkmale gleich gewichtet. Zunächst wird der durchschnittliche Wert des jeweiligen Performancekriteriums über alle Messungen der Testreihe gebildet. Die hierfür verwendeten SQL-Abfragen für das Desktop-System sind in den Listings A.1 und A.2 zu finden. Anschließend erfolgt die Normalisierung und Summation der Werte, wie in Abschnitt 2.4 beschrieben.

Testkriterium	**Haskell**	**C++**
Relativer Speedup	2,67	1,40
Effizienz	0,69	0,95
Paralleler Overhead [%]	0,31	0,05
Kosten [s]	12,05	9,23
Durchsatz [Records/s]	9680,59	9942,14
Durchschnittliche RSS [MiB]	465,05	2,93
Cache Hit-Rate [%]	92,87	98,53
Normalisiert		
Relativer Speedup	1,00	0,52
Effizienz	0,73	1,00
Paralleler Overhead	0,16	1,00
Kosten	0,77	1,00
Durchsatz	0,97	1,00
Durchschnittliche RSS	0,01	1,00
Cache Hit-Rate	0,94	1,00
Summe	4,58	6,52
Normalisiert	**65,43%**	**93,14%**

Tabelle 4.3: Gesamtbewertung der Messreihe des Desktop-Testsystems

Die Haskell Implementierung erzielt einen wesentlich größeren durchschnittlichen Speedup, aufgrund der sequenziellen Ausführung des C++ Codes bei weniger als 1.000 Eingangsdateien. Aus demselben Grund ist die Effizienz des imperativen Programms mit einem Mittelwert von $0,95$ nahezu optimal. Da keine Parallelisierung stattfindet, ist diese Größe über weite Teile der Messreihe konstant 1,0 (vgl. Abbildung D.3). Der parallele Overhead ist mit $0,05$ dementsprechend gering.

Trotz Ausführung des Algorithmus durch nur einen Threads sind die Kosten des C++ Programms um durchschnittlich $2,82s$ geringer als die der rein funktionalen Implementierung.

Eine Betrachtung des Durchsatzes zeigt, dass C++ mit 9.942,14 Records/s fast 300 Datensätze pro Sekunde mehr verarbeitet als Haskell. Allerdings ist auch dieser Wert stark geprägt durch die ausbleibende Parallelverarbeitung bei kleinen Problemgrößen. In Tabelle A.3 ist eine weitere Gesamtbewertung des Desktop-Testsystems dargestellt, welche ausschließlich Problemgrößen mit 1.000 Dateien berücksichtigt. Dabei wird die tatsächliche Leistungsfähigkeit des C++ Programms erkennbar. Unter diesen Bedingungen erreicht es einen mittleren Durchsatz von 20.869,96 Records/s.

Relativ unabhängig von der Dateinanzahl ist hingegen die durchschnittliche Cache Hit-Rate. Auch hier schneidet C++ mit 98,53% etwas besser ab als Haskell mit 92,87%.

Insgesamt ergibt sich ein Wert von 93,14% für C++ und 65,43% für Haskell. C++ erreicht in der Bewertung aus Tabelle 4.3 nicht den größtmöglichen Prozentsatz, da Haskells Speedup in diesem Fall überwiegt. Die Differenz von 27,71% kann so gedeutet werden, dass die Haskell Implementierung unter diesen Gesichtspunkten nur etwa $^3/_4$ der Leistung des C++ Programms erbringt.

Für Problemgrößen mit 1.000 Dateien erzielt das C++ Programm durchgehend bessere Werte als sein Haskell Äquivalent. Daher kommt es in Tabelle A.3 auf einen prozentualen Wert von 100%. Haskell erreicht hier 29,29% weniger.

EC2 Server-Testsystem

Einzelbewertungen

Da sämtliche Problemgrößen der Messungen auf dem EC2-Server 1.000 Dateien und mehr beinhalten, tritt der unerwünschte Effekt sequenzieller Verarbeitung durch das C++ Programm hier nicht auf. Dies ermöglicht eine konsistent Betrachtung über die gesamte Testreihe.

Abbildung 4.6 zeigt den relativen Speedup beider Programme bei unterschiedlichen Optimierungsgraden. Die Haskell Implementierungen und die optimierten C++ Varianten (`-O1`/`-O2`) weisen bis zu vier verwendeten Threads einen nahezu identischen Speedup auf. Dabei skalieren alle von ihnen sublinear, erreichen also keinen Wert $S(P)$, für den gilt $S(P) \geq P$. Darüber hinaus flacht die Kurve beider Programme deutlich ab, ähnlich der Messung auf dem Desktop-Testsystem. Es ist daher zu vermuten, dass an diesem Punkt die Hyper-Threading Technologie einsetzt und manche Threads sich einen dedizierten CPU-Kern teilen müssen. Erneut nimmt der Speedup von C++ in diesem Bereich deutlich mehr zu als der von Haskell. So beträgt der durchschnittliche Zuwachs oberhalb von vier Threads 1,07 für die imperative und 0,35 für die rein funktionale Programmiersprache.

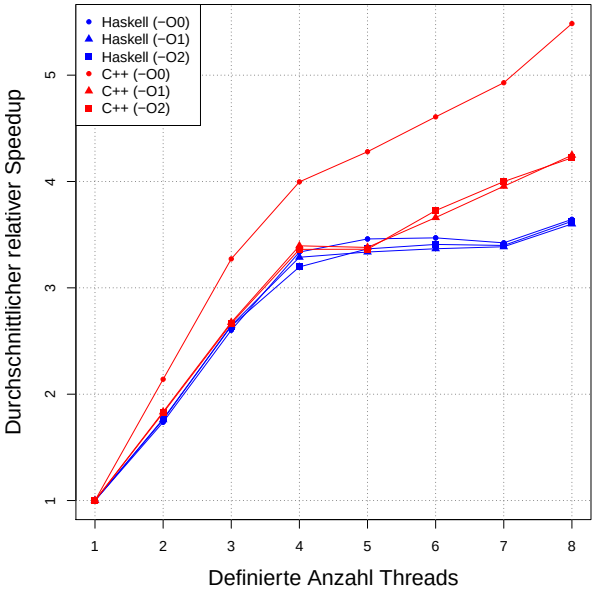

Abbildung 4.6: Durchschnittlicher Relativer Speedup (EC2-Server)

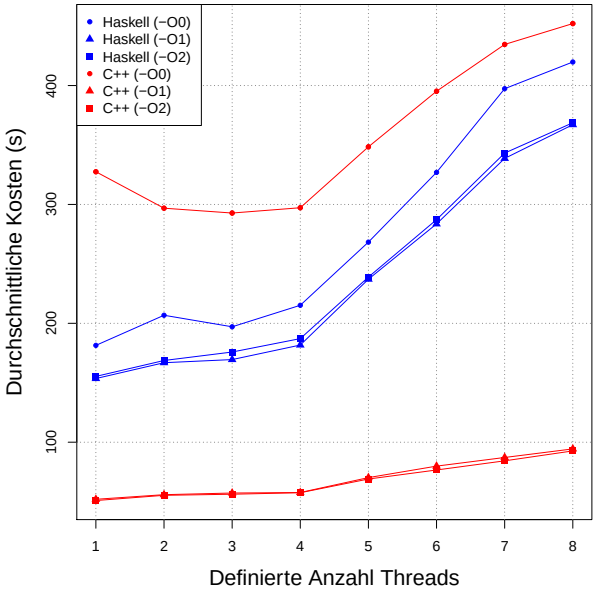

Abbildung 4.7: Durchschnittliche Kosten (EC2-Server)

Dass der Speedup des C++ Programms ohne Optimierungen alle anderen deutlich übersteigt, ist durch die sehr lange Antwortzeit unter Verwendung nur eines Threads zu erklären. Abbildung 4.7 lässt erkennen, dass die Programmversion zwar stets die größten Kosten – und somit auch die längste Antwortzeit – aufweist, diese aber im Verhältnis zur sequenziellen Ausführung am geringsten ansteigen. Daraus resultiert ein erhöhter Wert für den relativen Speedup, welcher bei zwei und drei Threads sogar einen Wert von 2,14 bzw. 3,27 aufweist. Das Programm skaliert unter diesen Bedingungen also superlinear.

Das mit `-O0` kompilierte C++ Programm ist somit auch das einzige, welches eine Effizienz größer als 1,00 erzielt. Der Abbildung E.3 ist zu entnehmen, dass dieser Wert unter Verwendung von drei Threads mit 1,09 maximal ist und anschließend wieder abfällt. Wie auch schon beim Desktop-System, nimmt die Effizienz beider Programme oberhalb von

vier Threads stark ab. Sie sinkt im Fall von C++ auf einen minimalen Wert von 0,53 (`-O1`/`-O1`) und bei Haskell auf 0,45 (`-O1`/`-O2`).

Folglich hält sich der parallele Overhead, dargestellt in Abbildung E.4, für C++ stets unterhalb von 50%. Das Haskell Programm überschreitet diesen Wert schon unter Verwendung von sieben Prozessoren mit etwa einem Prozent. Bei maximaler Threadanzahl verrichten die CPUs für die Ausführung des rein funktionalen Programms im Schnitt 54,71% mehr Arbeit als bei sequenzieller Ausführung.

Die gemessenen Kosten bestätigen die Beobachtungen der vorangegangenen Testreihe, dass das Aktivieren von Compiler-Optimierungen besonders bei C++ einen immens positiven Einfluss auf die Antwortzeit des Programms hat. Für C++ ergibt sich im Mittel eine Kostendifferenz von 286,30s zwischen der `-O0` und `-O1` Version des Programms sowie 1,46s zwischen `-O1` und `-O2`. Beim Haskell Programm sind es für den ersten Fall 39,35s. Von `-O1` nach `-O2` erhöhen sich die Kosten wieder minimal um 3,37s.

Auch der Verlauf der Resident Set Size bei verschiedenen Problemgrößen in Abbildung E.2 ähnelt dem aus der Messreihe des Desktop-Testsystem (vgl. Abbildung 4.5). Das Haskell Programm alloziert bei der kleinsten Verarbeitungsmenge von 10.000 Records durchschnittlich 190,44MiB Arbeitsspeicher, während C++ mit 3,69MiB deutlich darunter liegt. Die RSS der Implementierungen steigen bei einer Million Records bis auf maximal 19.308,56MiB (Haskell `-O2`), bzw. 23,88MiB (C++ `-O0`) an. Pro Record nimmt die Speicherbelegung des Haskell-Programms um durchschnittlich 19,18KiB zu. Beim C++ Programm entspricht dieser Wert lediglich 0,02KiB.

Beim Vergleich des Durchsatzes sind die optimierten Varianten des C++ Programms der Haskell Implementierung klar überlegen, wie die Abbildungen E.5 und E.6 verdeutlichen. Bei acht verwendeten Threads erreichen sie eine Verarbeitungsgeschwindigkeit von nahezu 25.000 Records/s. Haskell bleibt dagegen unabhängig vom Optimierungsgrad stets unterhalb von 8.000 Records/s. Die nicht optimierte Variante des rein funktionalen Programms verarbeitet aber dennoch mehr Datensätze pro Sekunde als C++ mit der `-O0` Compiler-Option. Mit etwa 5.000 Records/s bei maximaler Threadanzahl ist letzteres in diesem Vergleich das am wenigsten performante Programm.

Das Verhältnis von Durchsatz und Anzahl der verarbeiteten Dateien ist im Fall vom C++ relativ konstant (vgl. Abbildung E.7). Für das Haskell Programm allerdings zeichnet sich ein bemerkenswerter Negativtrend bei steigender Dateimenge ab. So verringert sich die Zahl der Datensätze pro Sekunde in jeder der Programmvarianten (`-O0`/`-O1`/`-O2`) um mehr als 1.800 (vgl. Abbildung E.8). Im Schnitt beträgt dieDifferenz des Durchsatzes zwischen der Verarbeitung von 1.000 und 10.000 Dateien 2.516,00 Records/s.

Gesamtbewertung

Auch für die Messergebnisse des EC2-Servers wird nachfolgend in der Tabelle 4.4 eine Gesamtbewertung durchgeführt.

Obwohl sich manche Werte, wie Effizienz und paralleler Overhead, nur geringfügig unterscheiden, ist der kumulierte Leistungsunterschied im Vergleich zum Desktop-System stärker ausgeprägt. So erreicht Haskell einen Prozentsatz von 59,17%. C++ kommt sogar auf den maximal möglichen Wert von 100%.

Testkriterium	**Haskell**	**C++**
Relativer Speedup	2,81	3,25
Effizienz	0,72	0,80
Paralleler Overhead [%]	0,28	0,20
Kosten [s]	251,50	164,24
Durchsatz [Records/s]	5492,39	12792,08
Durchschnittliche RSS [MiB]	5278,02	10,38
Normalisiert		
Relativer Speedup	0,86	1,00
Effizienz	0,90	1,00
Paralleler Overhead	0,71	1,00
Kosten	0,65	1,00
Durchsatz	0,43	1,00
Durchschnittliche RSS	0,00	1,00
Summe	3,55	6,00
Normalisiert	**59,17%**	**100%**

Tabelle 4.4: Gesamtbewertung der Messreihe des Server-Testsystems

Besonders bei der Resident Set Size (RSS) performt die C++ Implementierung besser als das Haskell Programm. Letzteres belegt mit durchschnittlich 5.278,02MiB über 500% mehr Ressourcen im Arbeitsspeicher der Maschine. Zeitweilig sind bei der Verarbeitung der umfassendsten Problemgröße von einer Million Records Werte von mehr als 50GiB zu beobachten. Da die normalisierten Werte auf lediglich zwei Nachkommastellen gerundet werden, erzielt Haskell diesbezüglich überhaupt keine Punkte.

Aber auch in puncto Kosten werden starke Unterschiede zwischen beiden Implementierung deutlich: die Prozessoren der Maschine sind im Schnitt $87,26$ Sekunden weniger mit der Ausführung des C++ MapReduce beschäftigt als mit der Haskell Variante.

Kapitel 5

Fazit

Ein Ergebnis der Untersuchungen ist die Feststellung, dass MapReduce eine durchaus effektive Methode für die logische Gliederung der Problemstellung darstellt. Zum einen lassen sich die benötigten Arbeitsschritte Parsen, Gruppieren und Aggregieren intuitiv den einzelnen Funktionsblöcken des Programmiermodells zuordnen. Zum anderen sind die betreffenden Datensätze während eines Großteils der Verarbeitung vollkommen unabhängig voneinander und die Funktionen somit parallelisierbar. Auf beiden Testsystemen werden deshalb Speedup-Werte von mehr als 3,0 unter Verwendung vier dedizierter Prozessoren erzielt. Obgleich nicht optimal, ist dies ein positives Resultat und es ist davon auszugehen, dass die Implementierungen unter Einsatz von mehr CPUs durchaus weitere Leistungssteigerungen aufweisen würden.

Der zu beobachtende sublineare Speedup hat verschiedene Ursachen. Obwohl der parallele Codeanteil sehr hoch ist, gibt es in jedem Programm stets auch Abschnitte sequenzieller Ausführung. Diese können zwangsläufig nur von nur einem der Prozessoren bearbeitet werden und lassen sich durch Parallelverarbeitung nicht beschleunigen.

Des Weiteren kann nicht davon ausgegangen werden, dass die zu verarbeitende Problemmenge perfekt zwischen allen Threads aufgeteilt wird. Dies hat zur Folge, dass Threads, die weniger Datensätze verarbeiten, für eine gewisse Zeit auf die Beendigung anderer Threads warten müssen. Um diese Form des parallelen Overheads zu verhindern, teilen beide Programmiersprachen die Daten zwischen den Recheneinheiten auf: Sparks werden zwischen Haskell Evaluation Contexts (HECs) geteilt, sobald einer von ihnen den Leerlauf erreicht und auch der *GNU parallel mode* enthält eine Art dynamischer Lastverteilung (vgl. Singler, Sanders und Putze 2007, S.1). Die sehr ähnliche CPU-Auslastung beider Implementierungen spricht dafür, dass diese Lastverteilung generell funktioniert. Allerdings stellen die Versuchsbedingungen insofern einen Sonderfall dar, als das sämtliche Eingangsdateien die gleiche Anzahl Datensätze enthalten.

Aber nicht nur die Verteilung der Arbeitslast, auch die Struktur des Algorithmus selbst hat Einfluss auf die Effizienz des Programms. Wie in Anhang A (Theoretischer Speedup) dargestellt, beansprucht das Einlesen und Parsen der Dateien den überwiegenden Teil der gesamten Antwortzeit. In einem solchen Fall wird auch von I/O gebundenen[25] Algorithmen gesprochen, da der limitierende Faktor für die Leistung vor allem Ein- und Ausgabeoperationen sind. Weil parallele Programmierung nur die CPU-Aktionen optimiert, nicht aber das Lesen von der Festplatte, nimmt die Effizienz der Programme mit mehr Threads stetig ab. Fordern außerdem sämtliche Threads gleichzeitig Ressourcen von einer einzigen lokalen Festplatte an, wie dies in der *MAP* Phase geschieht, so werden manche von ihnen stärker ausgebremst als andere, da nicht alle Dateien gleichzeitig gelesen werden können.

[25] „I/O bound" (Das 2013, S.238)

In beiden Testreihen bestätigt sich, dass die durch Hyper-Threading zusätzlich verfügbaren Prozessoren keine vollständigen CPUs ersetzen. Die Veringerung der Antwortzeit durch Hinzunahme eines solchen logischen Prozessors ist wesentlich geringer – schließlich teilen sich zwei Threads ein dediziertes Rechenwerk. Dieser Fakt ist bei Betrachtung der parallelen Leistungsmerkmale aus Abschnitt 2.4 in jedem Fall zu berücksichtigen. Der immense Anstieg des Overheads, bzw. die rapide abfallende Effizienz liefert somit nicht unbedingt klare Aussagen über die Programme selbst. Die parallele Ausführungen der Software wird oberhalb von vier Threads also nicht plötzlich schlechter, vielmehr sind die eingesetzten Prozessoren von einer anderen Qualität.

Es lässt sich allerdings feststellen, dass die C++ Implementierung deutlich mehr von Intels Hyper-Threading Technologie profitiert als die Haskell Version. Um daraus jedoch allgemeine Aussagen über die Programmiersprachen abzuleiten, wären weitere Untersuchungen mit verschieden strukturierten Programmen nötig. In jedem Fall tragen die zusätzlichen logischen Prozessoren zur einer besseren Auslastung der CPUs und einer messbaren Leistungssteigerung bei.

Der Test beider Programme auf unterschiedlicher Hardware erweist sich insofern als lohnenswert, als das manche Beobachtungen des einen Systems auf dem anderen bestätigt werden. So ist neben dem Speedup auch die CPU-Auslastung sowie der Verlauf der durchschnittlichen Resident Set Size (RSS) auf beiden Maschinen sehr ähnlich.

Eine umfangreichere Problemgröße, welche nur auf dem EC2-Server verarbeitet werden kann, erlaubt jedoch auch neue Einsichten, die auf dem Desktop-Rechner nicht zu beobachten sind. Der deutliche Performanceverlust der Haskell Implementierung bei einer steigenden Anzahl von Dateien etwa, tritt bis einschließlich 1.000 Dateien in dieser Form nicht auf (vgl. Abbildung D.12).

In beiden Programmiersprachen steht eine Vielzahl von Möglichkeiten für die Entwicklung paralleler und nebenläufiger Algorithmen zur Verfügung. Diese Arbeit erhebt daher keinen Anspruch auf eine vollständige Repräsentation. Entscheidendes Auswahlkriterium ist in diesem Fall die Simplizität der Parallelisierung, basierend auf sequentiellen Versionen der Programme. Sowohl in Haskell als auch in C++ ist diese Optimierung mit wenigen Änderungen im bestehenden Quellcode zu realisieren, wie in den Abschnitten 3.2.3 und 3.3.3 dargelegt. Funktionen höherer Ordnung erleichtern dabei nicht nur die Strukturierung der Programme aufgrund ihrer expressiven Semantik, sie sind auch Basis für eine anschließend einfach vorzunehmende Parallelisierung.

Allerdings gestaltet sich die Erstellung *leistungsstarker* paralleler Haskell Programme in mancher Hinsicht schwieriger als in C++. Der Grund dafür ist ein erhöhter Komplexitätsgrad. Die Bedarfsauswertung ist nur eine zusätzliche Komponente, welche bei der Implementierung des rein funktionalen Programms bedacht werden muss. Sie kann bei einer naiven Realisierung verheerenden Einfluss auf die Leistung des Programms haben und erschwert somit die Entwicklung. Zudem ist es nicht immer trivial, dadurch verursachte Schwachpunkte im Quellcode zu verorten. Treffendes Beispiel ist das Parsen der Dateien während der *MAP* Phase, wie in Abschnitt 3.2.1 beschrieben: Ohne den Operator `($!)` geschieht das Einlesen nicht strikt, wird also zurückgestellt und die Funktion `readAndParse` gibt einen unevaluierten Thunk zurück. Die Berechnung wird daraufhin erst zu einem späteren Zeitpunkt und nicht parallel ausgeführt. Eine Beschleunigung des Programms bleibt aus.

Des Weiteren ist die Wahl spezialisierter Datentypen essenziell für die Leistung der Haskell Implementierung. In Abbildung 5.1 ist beispielhaft eine Heap-Analyse des Programms dargestellt, in der der belegte Arbeitsspeicher nach Typ aufgeteilt wurde. Es ist

ersichtlich, dass Listen, erkennbar am Konstruktor `[]`, mit Abstand die meisten Ressourcen belegen[26]. Da der `String` Typ nichts anderes ist als eine Liste von `Chars`, ist der hohe Speicherverbrauch also auf die Verarbeitung solcher Zeichenketten zurückzuführen. Um diesem Problem vorzubeugen sollte daher ein speichereffizienter Datentyp bevorzugt werden, wie `Data.ByteString` oder `Data.Text`.

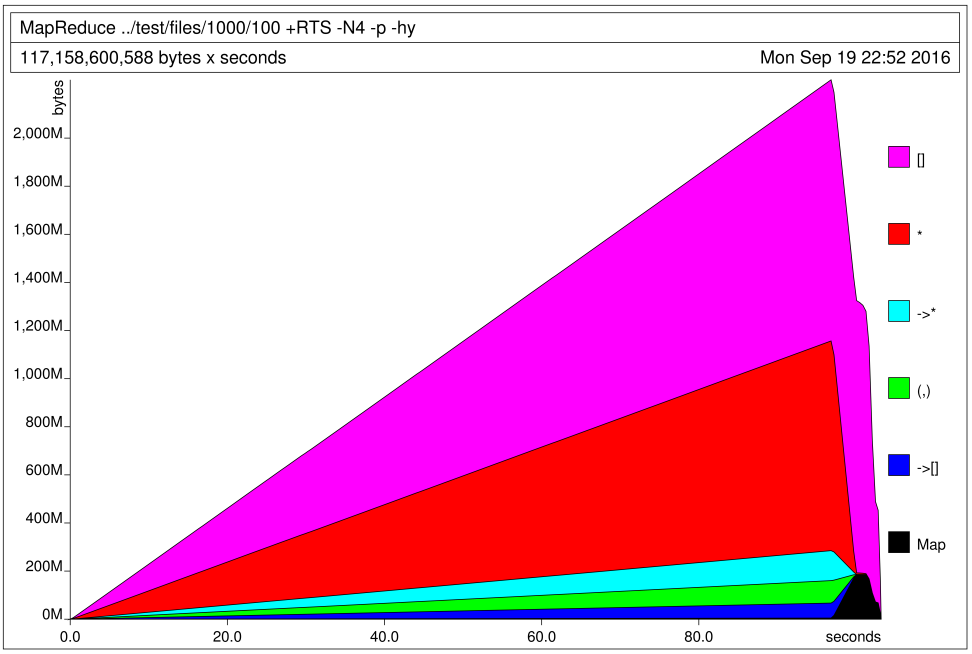

Abbildung 5.1: Heap-Profil des Haskell Programms nach Datentyp

Auch das Konzept der Evaluationsstrategien bedeutet mehr Komplexität für das Verständnis der Parallelisierung. Wenngleich keine größeren Eingriffe in den Quellcode eines bestehenden sequenziellen Programms nötig sind, so sind doch profunde Kenntnisse über die Funktionen `rpar`, `rseq` und `rdeepseq` unverzichtbar um effektive Strategien für die Parallelisierung des Algorithmus zu erstellen.

Das Programmverhalten in C++ hingegen ist aufgrund der niedrigeren Abstraktionsebene wesentlich expliziter. Allein der Umstand, dass Operationen genau da ausgeführt werden, wo sie in der sequenziellen Abfolge des Quellcodes definiert wurden, erleichtert Überlegungen bezüglich der Ausführungsweise erheblich. Außerdem ermöglicht die Hardwarenähe der Sprache etwa die manuelle Freigabe von überflüssigen Speicherobjekten, welche sich positiv auf die Ressourcennutzung auswirkt.

Im selben Zug bringt diese Art der Programmierung allerdings auch eine Fehleranfälligkeit mit sich, der die rein funktionale Sprache Haskell entbehrt. So muss etwa darauf geachtet werden, verwendete Variablen stets korrekt zu initialisieren und auch das manuelle Übergeben von Zeigern und Iteratoren kann eine Quelle für Fehler darstellen.

In beiden Sprachen müssen Programmierende einen Kompromiss zwischen vereinfachter Entwicklung und detaillierter Kontrolle über die Ausführung des Quellcodes eingehen. Je mehr Funktionalität parallelen Bibliotheken bzw. Evaluationsstrategien überlassen wird, umso weniger Aussagen lassen sich über den genauen Ablauf des Programms machen. So ist in der finalen Form der Programme keinesfalls sichergestellt, dass z.B.

[26]Das Symbol `*` in Abbildung 5.1 steht stellvertretend für einen unbekannten Typ (vgl. O'Sullivan, Goerzen und Stewart 2008, S.568).

die Gruppierung der Datensätze tatsächlich nach dem optimal-parallelen Schema aus Abschnitt 3.1 erfolgt.

Dass diese Vorgehensweise nicht immer zu der besten Leistung führt, beweist die ausbleibende Parallelverarbeitung in der *MAP* Phase durch C++ bei kleinen Problemgrößen. Daher ist auch nicht sichergestellt, dass die gruppierten Datensätze während der *REDUCE* Phase gleichzeitig aggregiert werden - schließlich sind es unter den beschriebenen Testbedingungen nur acht zu parallelisierende Operationen (vgl. Abschnitt 4.1). Tatsächlich lassen mehrere Testläufe der Programmversion, die im Anhang A für die Zeitmessung der MapReduce Funktionen genutzt wurde, keine deutliche Beschleunigung der Aggregation bei Verwendung mehrerer CPUs erkennen. Diese Beobachtung spricht dafür, dass der *GNU parallel mode* sich auch hier gegen eine Parallelverarbeitung entscheidet. Dies hat allerdings kaum Auswirkungen auf die Antwortzeit, da die Summation einen so geringen Bruchteil der Gesamtlaufzeit ausmacht (vgl. Anhang A, Theoretischer Speedup).

Findet jedoch eine Parallelverarbeitung in der *MAP* Phase statt, so ist das Resultat der Untersuchungen eindeutig. Die vorgestellte C++ Implementierung ist in diesem Fall wesentlich leistungsfähiger als ihr Haskell Äquivalent. So erzielen die beiden optimierten Programmversionen (`-O1`/`-O2`) in jeder Hinsicht vergleichbare oder bessere Performancewerte als die des rein funktionalen Programms. Dies bedeutet keineswegs, dass der Haskell Code nicht durchaus leistungsfähiger sein könnte, wenn eine tiefgreifendere Analyse und weitere Optimierungen vorgenommen würden. Es spiegelt einen Entwicklungsstand beider Prototypen wider, in dem etwa die gleiche Menge Arbeit steckt. Unter diesen Bedingungen ist das parallele C++ Programm klar zu präferieren.

Eine Tatsache bezüglich der Entwicklung sowohl paralleler als auch sequentieller Programme bestätigt sich jedoch in beiden Implementierungen: Funktionen höherer Ordnung tragen zu expressivem Quellcode und somit zu einer vereinfachten Verständlichkeit bei. Werden diese Sprachkonstrukte mit einer performanten Möglichkeit zu Auszeichnung gleichzeitiger Verarbeitung ergänzt, so ist eine darauf aufbauende Form der Programmoptimierung nahezu trivial.

Literatur

Aljabri, Malak Saleh (2015). „GUMSMP: a scalable parallel Haskell implementation“. Diss. University of Glasgow.

Amdahl, Gene M (1967). „Validity of the single processor approach to achieving large scale computing capabilities“. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, S. 483–485.

Backus, JW et al. (1962). „Revised report on the algorithmic language Algol 60“. In: *Numerische Mathematik* 4.1, S. 420–453.

Breitinger, Silvia, Rita Loogen, Yolanda Ortega Mallen et al. (1997). „The Eden coordination model for distributed memory systems“. In: *hips*. IEEE, S. 120.

Breitinger, Silvia, Rita Loogen, Yolanda Ortega-Mallén et al. (1996). „Eden—the paradise of functional concurrent programming“. In: *European Conference on Parallel Processing*. Springer, S. 710–713.

Brunswig, A. (1910). *Das Vergleichen und die Relationserkenntnis*. B. G. Teubner.

Chapman, Barbara, Gabriele Jost und Ruud Van Der Pas (2008). *Using OpenMP: portable shared memory parallel programming*. Bd. 10. MIT press.

Chi, Ed Huai-Hsin et al. (1997). *Efficiency of Shared-Memory Multiprocessors for a Genetic Sequence Similarity Search Algorithm*.

Das, Lyla B. (2013). *Embedded Systems: An Integrated Approach*. Pearson Education India. ISBN: 9789332511675.

Dean, Jeffrey und Sanjay Ghemawat (2004). „MapReduce: Simplified Data Processing on Large Clusters“. In: *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, S. 137–150.

Ding, Yiping, Ethan Bolker und Arjun Kumar (2003). „Performance implications of hyperthreading“. In: *In Proc. CMG*, S. 21–29.

Flynn, Michael J (1966). „Very high-speed computing systems“. In: *Proceedings of the IEEE* 54.12, S. 1901–1909.

Frost, Richard A, Rahmatullah Hafiz und Paul Callaghan (2008). „Parser combinators for ambiguous left-recursive grammars“. In: *International Symposium on Practical Aspects of Declarative Languages*. Springer, S. 167–181.

Gropp, W. et al. (2007). *MPI - Eine Einführung: Portable parallele Programmierung mit dem Message-Passing Interface*. De Gruyter. ISBN: 9783486841008.

Hoareetal, C.A.R. (2001). „Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell“. In: *Engineering theories of software construction* 180, S. 47.

Hu, Zhenjiang, John Hughes und Meng Wang (2015). „How functional programming mattered“. In: *National Science Review* 2.3, S. 349–370.

Hudak, Paul et al. (2007). „A history of Haskell: being lazy with class“. In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, S. 12–1.

Hughes, John (1989). „Why functional programming matters“. In: *The computer journal* 32.2, S. 98–107.

Lämmel, Ralf (2008). „Google's MapReduce programming model—Revisited“. In: *Science of computer programming* 70.1, S. 1–30.

Leijen, Daan (2001). *Parsec, a fast combinator parser*.

Leijen, Daan und Erik Meijer (2002). *Parsec: Direct style monadic parser combinators for the real world*.

Loogen, Rita (2012). „Eden–parallel functional programming with Haskell“. In: *Central European Functional Programming School*. Springer, S. 142–206.

Marlow, Simon, Patrick Maier et al. (2010). „Seq no more: better strategies for parallel Haskell“. In: *ACM Sigplan Notices*. Bd. 45. 11. ACM, S. 91–102.

Marlow, Simon, Ryan Newton und Simon Peyton Jones (2011). „A monad for deterministic parallelism“. In: *ACM SIGPLAN Notices*. Bd. 46. 12. ACM, S. 71–82.

Marlow, Simon, Simon Peyton Jones und Satnam Singh (2009). „Runtime support for multicore Haskell“. In: *ACM Sigplan Notices*. Bd. 44. 9. ACM, S. 65–78.

Marr, Deborah T et al. (2002). „Hyper-Threading Technology Architecture and Microarchitecture.“ In: *Intel Technology Journal* 6.1.

Märtin, Christian (2014). „Multicore processors: challenges, opportunities, emerging trends“. In: *Proc. Embedded World Conference*, S. 1–9.

Moggi, Eugenio (1988). *Computational lambda-calculus and monads*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science.

O'Sullivan, Bryan, John Goerzen und Donald Bruce Stewart (2008). *Real world haskell: Code you can believe in*. O'Reilly Media, Inc.

Pepper, Peter und Petra Hofstedt (2006). *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer-Verlag.

Peyton Jones, Simon L und Philip Wadler (1993). „Imperative functional programming“. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, S. 71–84.

Peyton Jones, Simon, Andrew Gordon und Sigbjorn Finne (1996). „Concurrent Haskell“. In: *POPL*. Bd. 96, S. 295–308.

Peyton Jones, Simon, Roman Leshchinskiy et al. (2008). „Harnessing the multicores: Nested data parallelism in Haskell“. In: *LIPIcs-Leibniz International Proceedings in Informatics*. Bd. 2. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Ranger, Colby et al. (2007). „Evaluating mapreduce for multi-core and multiprocessor systems“. In: *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. IEEE, S. 13–24.

Rauber Thomas und Rünger, Gudula (2012). *Parallele Programmierung*. eXamen.press. Springer Berlin Heidelberg. ISBN: 9783642136047.

Sahni, Sartaj und Venkat Thanvantri (1995). *Parallel Computing: Performance Metrics and Models*. Techn. Ber. University of Florida.

Singler, Johannes und Benjamin Konsik (2008). „The GNU libstdc++ parallel mode: software engineering considerations“. In: *Proceedings of the 1st international workshop on Multicore software engineering*. ACM, S. 15–22.

Singler, Johannes, Peter Sanders und Felix Putze (2007). „MCSTL: The multi-core standard template library“. In: *European Conference on Parallel Processing*. Springer, S. 682–694.

Søndergaard, Harald und Peter Sestoft (1990). „Referential transparency, definiteness and unfoldability“. In: *Acta Informatica* 27.6, S. 505–517.

Stroustrup, B. und F. Langenau (2015). *Die C++-Programmiersprache: aktuell zum C++11-Standard*. Carl Hanser Verlag GmbH & Company KG. ISBN: 9783446439818.

Talbot, Justin, Richard M Yoo und Christos Kozyrakis (2011). „Phoenix++: modular MapReduce for shared-memory systems“. In: *Proceedings of the second international workshop on MapReduce and its applications*. ACM, S. 9–16.

Thompson, Simon (2015). *Haskell: The Craft of Functional Programming*. International Computer Science Series. Pearson Education Limited. ISBN: 9781292127576.

Trinder, Philip W et al. (1996). „GUM: a portable parallel implementation of Haskell“. In: *ACM SIGPLAN Notices*. Bd. 31. 5. ACM, S. 79–88.
Trinder, Philip W. et al. (1998). „Algorithm + strategy = parallelism“. In: *Journal of functional programming* 8.01, S. 23–60.
Veldhuizen, Todd (1995). „Expression templates“. In: *C++ Report* 7.5, S. 26–31.
Wadler, Philip (1992). „Comprehending monads“. In: *Mathematical structures in computer science* 2.04, S. 461–493.

Online Ressourcen

Dawes, Beman (2000). *Boost Formal Review Process*. URL: `http://www.boost.org/community/reviews.html` (besucht am 12. 08. 2016).
Donnelly, Charles und Richard Stallman (2015). *Bison - The Yacc-compatible Parser Generator*. URL: `https://www.gnu.org/software/bison/manual/bison.pdf` (besucht am 12. 08. 2016).
Eden Group (2016). *Hackage: The edenmodules package*. URL: `https://hackage.haskell.org/package/edenmodules-1.2.0.0/docs/Control-Parallel-Eden.html` (besucht am 11. 08. 2016).
Free Software Foundation (2016). *The GNU C++ Library*. URL: `https://gcc.gnu.org/onlinedocs/libstdc++` (besucht am 18. 08. 2016).
GHC Team (2016). *GHC Users Guide Documentation*. URL: `https://downloads.haskell.org/~ghc/8.0.1/docs/users_guide.pdf` (besucht am 24. 08. 2016).
Intel Corporation (2015). *Desktop 4th Gen Intel Core Processors Datasheet, Vol. 1*. URL: `https://www-ssl.intel.com/content/www/us/en/processors/core/4th-gen-core-family-desktop-vol-1-datasheet.html` (besucht am 22. 08. 2016).
Intel Corporation (2016). *Intel Xeon Processor E7 v4 Family*. URL: `http://ark.intel.com/products/family/93797/Intel-Xeon-Processor-E7-v4-Family#@Server` (besucht am 11. 06. 2016).
Marlow, Simon (2015). *Fighting spam with Haskell*. URL: `https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell` (besucht am 08. 08. 2016).
Marlow, Simon und Andy Gill (2009). *Happy User Guide*. URL: `https://www.haskell.org/happy/doc/html` (besucht am 06. 08. 2016).
OpenMP Architecture Review Board (2015). *OpenMP 4.5 Complete Specifications*. URL: `http://www.openmp.org/mp-documents/openmp-4.5.pdf` (besucht am 16. 08. 2016).
Peyton Jones, Simon (2011). *Escape from the ivory tower. The Haskell journey*. URL: `https://cucats.soc.srcf.net/files/Escape%20from%20the%20ivory%20tower%20Feb12.pdf` (besucht am 26. 07. 2016).
Rubinsteyn, Alex (2016). *The Parsnip Parser Library*. URL: `https://sourceforge.net/projects/parsnip-parser/files/parsnip` (besucht am 12. 08. 2016).
Stallman, Richard M. und GCC Developer Community (2016). *Using the GNU Compiler Collection*. URL: `https://gcc.gnu.org/onlinedocs/gcc-6.1.0/gcc.pdf` (besucht am 24. 08. 2016).
Stephane Eranian Eric Gouriou, Tipp Moseley und Willem de Bruijn (2015). *Linux kernel profiling with perf*. URL: `https://perf.wiki.kernel.org/index.php/Tutorial` (besucht am 26. 07. 2016).
van Waveren, Matthijs et al. (2013). *Frequently Asked Questions on OpenMP*. URL: `http://openmp.org/openmp-faq.html` (besucht am 29. 05. 2016).

Anhang

Abbildungsverzeichnis (Anhang)

Anhang A

Allgemeiner Anhang

Effizienz beim Zusammenfügen von Listen

Allgemein gilt für die sequenzielle sowie die parallele Verfahrensweisen das Folgende:

1. 1.1. Zwei zu vereinende Listen enthalten jeweils n Elemente und sind sortiert nach einer gewissen Priorität.
   1.2. Dann kann das Zusammenfügen der Listen mit linearer Komplexität erfolgen, indem stets die ersten Elemente der Listen verglichen und das kleinere der beiden an das Ende der Zielliste angehängt wird.
   1.3. Nach jeder Vergleichsoperation wird der Zielstruktur somit genau ein Element hinzugefügt.
   1.4. Das letzte verbleibende Element in einer der Listen muss mit keinem anderen verglichen werden. Es ist nach dem Sortierschema definitiv das letzte Element.
   1.5. Somit müssen maximal $2n - 1$ Vergleichsoperationen durchgeführt werden um zwei Listen zusammenzuführen.
2. Für das Zusammenführen von m Teillisten zu einer gemeinsamen Liste, sind genau $m - 1$ solcher Zusammenfüge-Operationen erforderlich.

Die Formel der sequenziellen Verfahrensweise ergibt sich wie folgt:

1. Es sind m Teillisten mit jeweils n Elementen sequenziell zusammenzuführen.
2. Das Zusammenfügen der beiden ersten Teillisten erfordert $2n - 1$ Vergleichsoperationen (siehe oben).
3. Das Zusammenfügen der beiden letzten verbleibenden Teillisten muss $m \cdot n - 1$ Vergleichsoperationen erfordern, denn die Zielliste enthält $m \cdot n$ Elemente.
4. Die zusammengefügte Liste wächst mit jedem Schritt um n Elemente an (vgl. Abbildung 3.2).
5. Das Zusammenfügen zweier Teillisten erfordert daher im Schnitt $\frac{(2n-1)+(m \cdot n-1)}{2}$ Vergleiche.
6. Zusammen mit dem allgemeinen Punkt 2 (siehe oben), ergibt sich folgende Formel für die Anzahl Vergleichsoperationen:

$$OPS_{seq} = (m-1) \cdot \frac{(2n-1) \cdot (m \cdot n - 1)}{2} \tag{A.1}$$

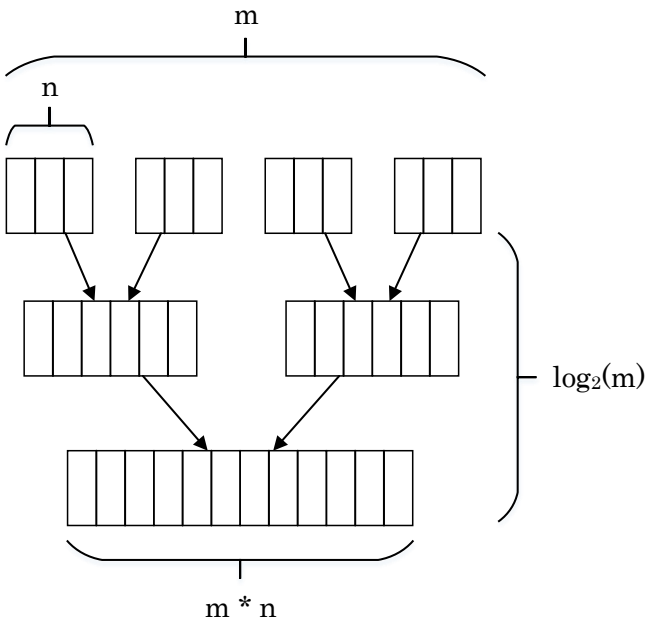

Abbildung A.1: Optimal-paralleles Zusammenfügen von Listen

Die Formel der parallelen Verfahrensweise ergibt sich aus folgenden Überlegungen:

1. Es sind m Teillisten mit jeweils n Elementen parallel zusammenzuführen.
2. Bezeichnen wir die Menge der Zusammenfüge-Operationen, die auf Listen gleicher Länge ausgeführt werden als „Stufen". Dann sind $\log_2(m)$ Stufen notwendig, um die Zielliste zu erhalten (vgl. Abbildung A.1).
3. In der ersten Stufe werden Teillisten der Größe n mittels $^m/_2$ Operationen zusammengeführt.
4. Jede Stufe erfordert halb so viele Zusammenfüge-Operationen, wie die vorhergehende.
5. Mit jeder Stufe verdoppelt sich die Größe der Teillisten.
6. Daraus lässt sich auf folgende Formel schließen*:

$$OPS_{par} = \sum_{i=1}^{\log_2(m)} \frac{m}{2i} \cdot (2in - 1) = \sum_{i=1}^{\log_2(m)} m \cdot n - \frac{m}{2i} \tag{A.2}$$

*Ein solches optimal-paralleles Zusammenfügen von Listen ist nur möglich, wenn die Anzahl von Teillisten m einer Zweierpotenz entspricht. Andernfalls werden auch bei dieser Vorgehensweise manche Teillisten sequenziell zusammengeführt. Wie Abbildung 3.3 jedoch zeigt, steigt die Anzahl der benötigten Operationen weit weniger stark an als bei dem rein sequenziellen Algorithmus.

Testdatenformat

Nachfolgend ist das Format der zu aggregierenden Records dargestellt. Die Belegung der Felder entspricht einem Wert, welcher dem jeweiligen regulären Ausdruck genügt.

```
RECORD
#addkey                 \w{10}
#filename               \w{10}
#input_id               \d{10}x\d{3}_\d{6}
#input_type             \w{4}
#output_id              \d{10}x\d{3}_\d{6}
#output_type            \w{5}
#source_id              \w{3}
F AccountingStatusType  \d
F AggregationsDatum     \d{4}(0[1-9]1[0-2])(0[1-9]1[0-9]2[0-9]3[0-1])    # Schluesselwert
F Anschlussvariante     \w{10}
F CFSID                 \w{8}-\w{4}-\w{4}-\w{4}-\w{12}
F CFSSID                \w{8}-\w{4}-\w{4}-\w{4}-\w{12}
F CycleDate             \d{4}(0[1-9]1[0-2])(0[1-9]1[0-9]2[0-9]3[0-1])
F Dauer                 \d+                                              # Aggregationswert
F Dauerstatus           [01]
F EventTypeID           \d
F Eventklasse           \w{3}-\w{7}                                      # Schluesselwert
F Eventsource           S\d{15}                                          # Schluesselwert
F EventsourcePraefix    \d
F FlatStatus            [01]
F Flatklasse            0[1-2]
F InInterfaceID         \d{2}
F InternalID            \w{35}
F LogicalInterface      \w{2}-\d\/\d\/\d\.\d{4}
F MRSEventType          \d{5}
F NASID                 \w{4}\d{2}
F Plattform             \d                                               # Schluesselwert
F ProcessingDate        \d{4}(0[1-9]1[0-2])(0[1-9]1[0-9]2[0-9]3[0-1])
F QoS_Class             [012]
F RFSID                 \w{8}-\w{4}-\w{4}-\w{4}-\w{12}
F Selektionsparameter   \d
F SubscriptionID        \d{15}
F Tarifzone             \w{3}-\w{3}-\w{2}                                # Schluesselwert
F Timestamp             \d{4}-(0[1-9]1[0-2])-(0[1-9]1[0-9]2[0-9]3[0-1])T
                        (0[0-9]1[0-9]2[0-4]):([0-5][0-9]):([0-5][0-9])
F UnixTimestamp         \d{10}
F VBSLKennzeichen       \w{4}
F Verkehrsrichtung      [IE]
F Volumen               \d+                                              # Aggregationswert
F ZirkCounter           \d+
.
```

Logische Struktur der Parserregeln

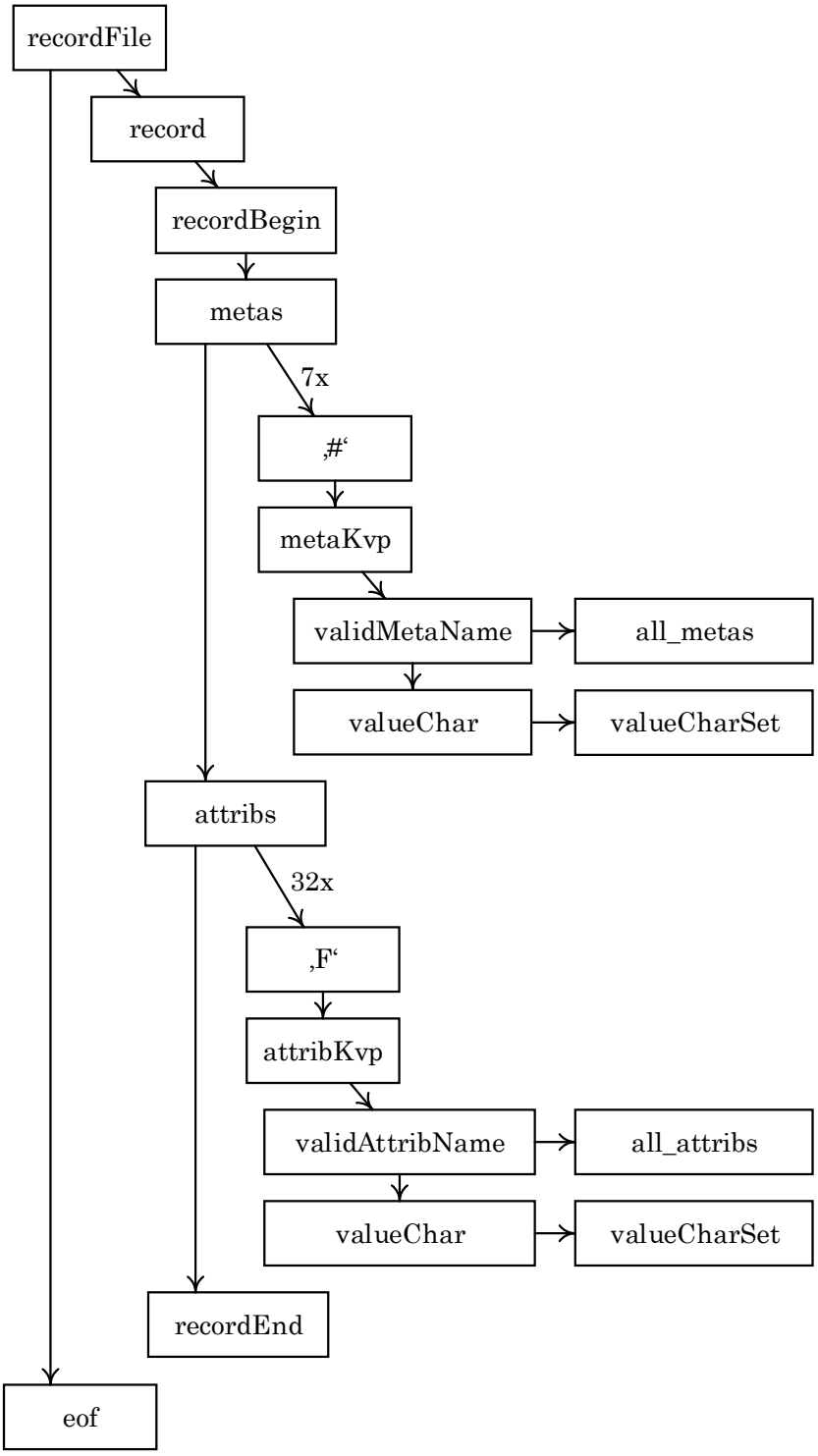

Abbildung A.2: Logische Struktur des Parsec-Parsers

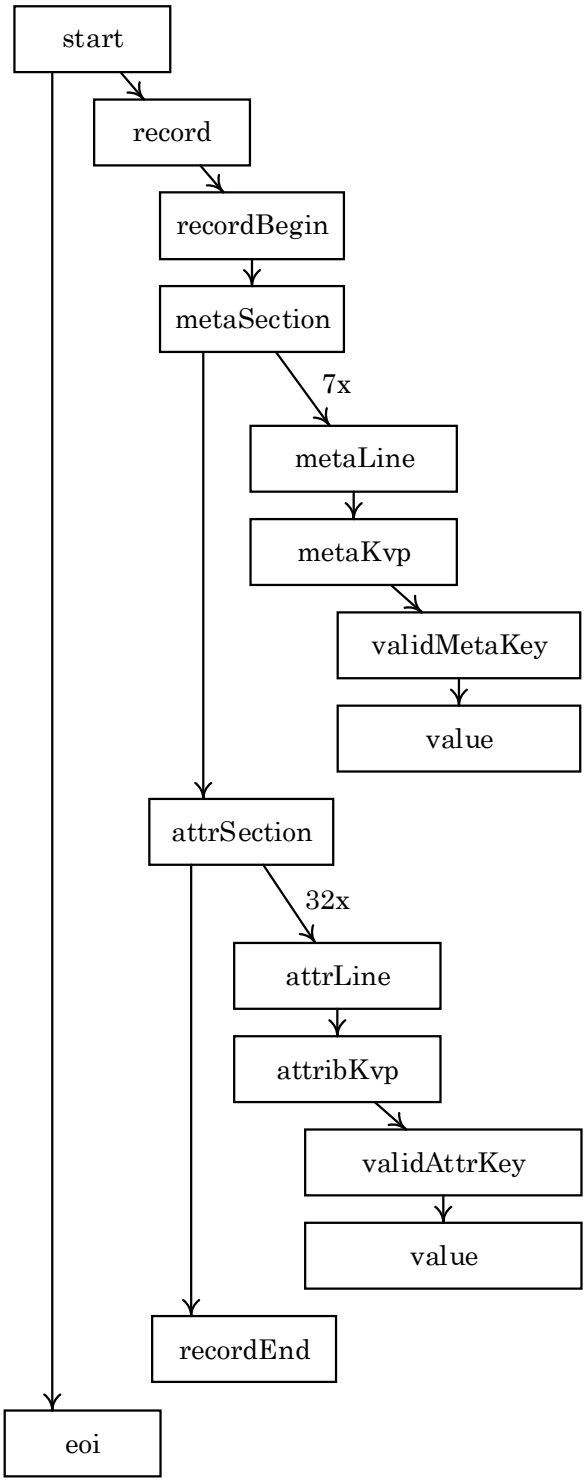

Abbildung A.3: Logische Struktur des Spirit-Parsers

Struktur der SQLite Datenbanken

Tabellenname	`test_results`
Primärschlüssel	`(language,threads,optimization,files,records_per_file)`
Attributname	**Attributtyp**
language	`TEXT NOT NULL`
optimization	`INTEGER NOT NULL`
threads	`INTEGER NOT NULL`
files	`INTEGER NOT NULL`
records_per_file	`INTEGER NOT NULL`
average_rss	`REAL`
cache_references	`REAL`
cache_misses	`REAL`
cache_hit_rate	`REAL`
cpu_time	`REAL`
cpu_util	`REAL`
efficiency	`REAL`
parallel_overhead	`REAL`
speedup	`REAL`
throughput	`REAL`
wallclock_time	`REAL`

Tabelle A.1: Schema der SQLite Datenbank `test_results_desktop.db`

Tabellenname	`test_results`
Primärschlüssel	`(language,threads,optimization,files,records_per_file)`
Attributname	**Attributtyp**
language	`TEXT NOT NULL`
optimization	`INTEGER NOT NULL`
threads	`INTEGER NOT NULL`
files	`INTEGER NOT NULL`
records_per_file	`INTEGER NOT NULL`
average_rss	`REAL`
cpu_time	`REAL`
cpu_util	`REAL`
efficiency	`REAL`
parallel_overhead	`REAL`
speedup	`REAL`
throughput	`REAL`
wallclock_time	`REAL`

Tabelle A.2: Schema der SQLite Datenbank `test_results_ec2.db`

SQL Abfragen für Gesamtbewertung

```
# Speedup
SELECT language, AVG(speedup)
FROM test_results
GROUP BY language;
# Effizienz C++
SELECT language, AVG(efficiency)
FROM (
    SELECT language, efficiency
    FROM test_results WHERE language='cpp' AND files='1000'
    UNION
    SELECT language, corrected_efficiency as efficiency
    FROM (
        SELECT language, speedup as corrected_efficiency
        FROM test_results WHERE language='cpp' AND NOT files='1000'
    )
)
# Effizienz Haskell
SELECT language, AVG(efficiency)
FROM test_results WHERE language='haskell';
# Paralleler Overhead
SELECT language, AVG(parallel_overhead)
FROM test_results
GROUP BY language;
# Kosten C++
SELECT language, AVG(cost)
FROM (
    SELECT language, wallclock_time * threads as cost
    FROM test_results WHERE language='cpp' AND files = '1000'
    UNION
    SELECT language, wallclock_time as cost
    FROM test_results WHERE language='cpp' AND NOT files = '1000'
)
# Kosten Haskell
SELECT language, AVG(cost)
FROM (
    SELECT language, wallclock_time * threads as cost
    FROM test_results WHERE language='haskell'
)
# Durchsatz
SELECT language, AVG(throughput)
FROM test_results
GROUP BY language;
# Durchschnittliche RSS
SELECT language, AVG(average_rss)
FROM test_results
GROUP BY language;
# Cache Hit-Rate
SELECT language, AVG(cache_hit_rate) * 100
FROM test_results
GROUP BY language;
```

Listing A.1: SQL-Abfragen für Gesamtbewertung des Desktop-Systems

```
# Speedup
SELECT language, AVG(speedup)
FROM test_results
WHERE files='1000' GROUP BY language;
# Effizienz
SELECT language, AVG(efficiency)
FROM test_results
WHERE files='1000' GROUP BY language;
# Paralleler Overhead
SELECT language, AVG(parallel_overhead)
FROM test_results
WHERE files='1000' GROUP BY language;
# Kosten
SELECT language, AVG(cost)
FROM (
    SELECT language, wallclock_time * threads as cost
    FROM test_results WHERE files='1000'
) GROUP BY language;
# Durchsatz
SELECT language, AVG(throughput)
FROM test_results
WHERE files='1000' GROUP BY language;
# Durchschnittliche RSS
SELECT language, AVG(average_rss)
FROM test_results
WHERE files='1000' GROUP BY language;
# Cache Hit-Rate
SELECT language, AVG(cache_hit_rate) * 100
FROM test_results
WHERE files='1000' GROUP BY language;
```

Listing A.2: SQL-Abfragen für Gesamtbewertung des Desktop-Systems bei 1.000 Dateien)

```
# Speedup
SELECT language, AVG(speedup)
FROM test_results
GROUP BY language;
# Effizienz
SELECT language, AVG(efficiency)
FROM test_results
GROUP BY language;
# Paralleler Overhead
SELECT language, AVG(parallel_overhead)
FROM test_results
GROUP BY language;
# Kosten
SELECT language, AVG(cost)
FROM (
    SELECT language, wallclock_time * threads as cost
    FROM test_results
) GROUP BY language;
# Durchsatz
SELECT language, AVG(throughput)
FROM test_results
GROUP BY language;
# Durchschnittliche RSS
SELECT language, AVG(average_rss)
FROM test_results
GROUP BY language;
```

Listing A.3: SQL-Abfragen für Gesamtbewertung des EC2-Servers

Gesamtbewertung Desktop-Testsystem

Testkriterium	**Haskell**	**C++**
Relativer Speedup	2,68	2,97
Effizienz	0,69	0,75
Paralleler Overhead [%]	0,31	0,25
Kosten [s]	23,48	21,51
Durchsatz [Records/s]	9529,82	20869,96
Durchschnittliche RSS [MiB]	887,67	4,52
Cache Hit-Rate [%]	92,14	98,65
Normalisiert		
Relativer Speedup	0,90	1,00
Effizienz	0,92	1,00
Paralleler Overhead	0,81	1,00
Kosten	0,92	1,00
Durchsatz	0,46	1,00
Durchschnittliche RSS	0,01	1,00
Cache Hit-Rate	0,93	1,00
Summe	4,95	7,00
Normalisiert	**70,71%**	**100%**

Tabelle A.3: Gesamtbewertung der Messreihe des Desktop-Testsystems bei Verarbeitung von 1.000 Dateien

Theoretischer Speedup

Um den maximalen theoretischen Speedup nach Amdahl (vgl. Abschnitt 2.4) zu berechnen werden die sequenziellen Ausführungszeiten der *MAP*, *GROUP*, und *REDUCE* Funktionen sowie die Antwortzeit beider Implementierungen gemessen. Der hierfür leicht veränderte Quellcode ist den Dateien `cpp/Main_scale0*.cpp` und `haskell/Main_scale0*.hs` (je ein Modul pro Optimierungsgrad) im elektronischen Anhang zu entnehmen. Die Haskell Funktionen zur Zeitmessung werden aus dem Modul `Criterion.Measurement` importiert. Für C++ wird der `<chrono>` Header verwendet, welcher seit C++11 zum Sprachstandard gehört.

Alle drei genannten Funktionen sind als potentiell parallele Programmbestandteile anzunehmen. Dann lässt sich der sequenzielle Anteil definieren als die Differenz aus der Antwortzeit eines Programms und der Laufzeit dieser drei Funktionen. Dies ist eine idealiserte Betrachtungsweise, da bereits im Abschnitt 3.1 festgestellt wurde, dass die *GROUP* Funktion nicht komplett parallelisiert werden kann. Das Vorgehen erlaubt jedoch eine ausreichend genaue Abschätzung des theoretischen Speedups $S(P)$, der sich aus der Formel 2.6 ergibt.

Die dargestellten Werte werden auf dem Desktop-Testsystem mit einer Problemgröße von 1.000 Dateien mit jeweils 100 Records erhoben und auf vier Nachkommastellen gerundet. Auf eine Berechnung mit alternativen Problemgrößen wird verzichtet, da allein das Verhältnis zwischen parallelen und sequenziellen Laufzeiten eine Rolle spielt, nicht die Laufzeiten selbst.

Sprache	**C++**			**Haskell**		
Optimierungsgrad	`-O0`	`-O1`	`-O2`	`-O0`	`-O1`	`-O2`
MAP [s]	61,0484	9,9304	9,7652	22,1500	22,1600	21,9600
GROUP [s]	2,3474	0,1209	0,1203	3,2200	3,1700	3,1830
REDUCE [s]	0,0038	0,0003	0,0001	1,0190	1,0180	1,0210
Antwortzeit [s]	63,4000	10,0600	9,8900	25,5600	26,5300	26,3300
Paralleler Programmanteil [%]	0,9999	0,9992	0,9996	0,9936	0,9931	0,9937
Theoretischer Speedup ($P = 2$)	1,9999	1,9983	1,9991	1,9872	1,9864	1,9875
Theoretischer Speedup ($P = 4$)	3,9999	3,9900	3,9947	3,9242	3,9193	3,9257
Theoretischer Speedup ($P = 8$)	7,9996	7,9536	7,9754	7,6551	7,6334	7,6619

Tabelle A.4: Berechnung des theoretischen Speedups

Die Messungen lassen erkennen, dass die *MAP* Funktion und somit das Parsen der Dateien den überwiegenden Teil der gesamten Antwortzeit beansprucht. Besonders bei dem C++ Programm ist die benötigte Zeit für die *REDUCE* Operation im Verhältnis verschwindend gering.

Den dargestellten Werten lässt sich außerdem entnehmen, dass unter der Annahme alle drei Funktionen ließen sich komplett parallelisieren, beide Implementierungen einen parallelen Programmanteil von mehr als 99% aufweisen. Dies resultiert in einem hohen

theoretischen Speedup, der für sämtliche Programmvariationen nah an die Zahl verwendeter Prozessoren P kommt. Beide Implementierungen bieten somit ausreichend Potenzial für die parallele Ausführung mit bis zu acht Prozessoren.

Die verwendete Formel 2.6 lässt allerdings den unvermeidlichen parallelen Overhead sowie den Fakt, dass nur vier komplett ausgeprägte CPUs vorhanden sind vollkommen außer Acht. Daher sind die berechneten Werte für den theoretischen Speedup bestenfalls als Orientierungswert zu verstehen.

Anhang B

Haskell Quellcode

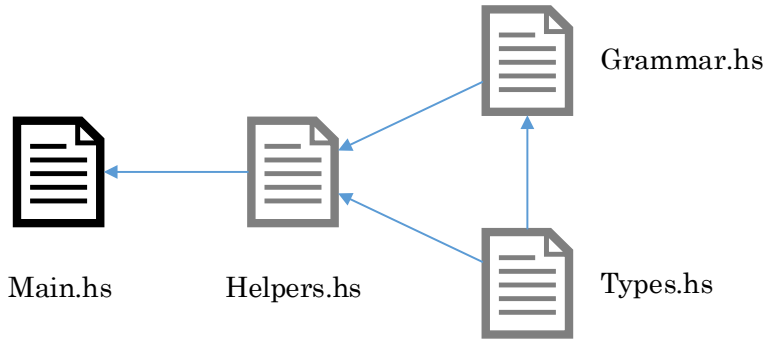

Abbildung B.1: Haskell Programmstruktur

```haskell
import qualified Control.Monad.Parallel as MP
import Control.Parallel.Strategies
import System.Directory (getDirectoryContents)
import System.Environment (getArgs)
import Data.Either (rights, lefts)
import Helpers

main :: IO ()
main = do
    -- get file location as command line parameter
    args <- getArgs
    let file_path = args!!0

    -- read directory contents
    dir_contents <- getDirectoryContents file_path

    -- filter directories and complete path
    let files = map (\f -> file_path ++ "/" ++ f) $
                filter (`notElem` [".", ".."]) dir_contents

    -- read and parse all files found
    parsed <- MP.mapM readAndParse files

    -- check if parsing failed for some files
    if (length $ lefts parsed) > 0
        then do
            putStrLn "Failed to parse the following files:"
            mapM_ putStrLn $ getMalformedFiles files parsed
        else do
            putStrLn "All files parsed successfully"

    let result = mapReducePar (rights parsed)
    mapM_ (putStrLn . prettyPrint) result

mapReduce input = map reduce $
                  (splitMap . mergeMaps) $
                  map (makeMap . validate) input

mapReducePar input = parMap rdeepseq reduce $
                     (splitMap . mergeMapsPar) $
                     parMap rseq (makeMap . validate) input
```

Listing B.1: haskell/Main.hs

```haskell
module Helpers
    ( readAndParse
    , makeMap
    , mergeMaps
    , mergeMapsPar
    , splitMap
    , reduce
    , getMalformedFiles
    , validate
    , prettyPrint
    ) where

import Control.Parallel.Strategies
import Types (Record(..), makeKey, makeTuple)
import Grammar
import Text.ParserCombinators.Parsec (ParseError)
import Data.List (foldl')
import qualified Data.Text.IO as T
import qualified Data.Map.Lazy as M

readAndParse :: String -> IO (Either ParseError [Record])
readAndParse file = do
    contents <- T.readFile file
    return $! parseRecords contents

makeMap :: [Record] -> M.Map String [(Int,Int)]
makeMap xs = M.fromListWith (++) $ zip (map makeKey xs) (map makeTuple xs)

splitMap :: M.Map k [v] -> [(k,[v])]
splitMap = M.toList

reduce :: (String, [(Int,Int)]) -> (String, (Int,Int))
reduce (key,values) = (key,aggregates) where
    aggregates = foldl' (\(d',v') (d,v) -> ((d'+d),(v'+v))) (0,0) values

mergeMaps :: (Ord k, Ord v) => [M.Map k [v]] -> M.Map k [v]
mergeMaps []       = M.empty
mergeMaps [x]      = x
mergeMaps (x:y:xs) =
    let x'  = M.fromAscListWith (++) $ mergeLists (M.toAscList x)
                                                  (M.toAscList y)
        xs' = mergeMaps xs
    in M.fromAscListWith (++) $ mergeLists (M.toAscList x')
                                           (M.toAscList xs')

mergeMapsPar :: (Ord k, Ord v) => [M.Map k [v]] -> M.Map k [v]
mergeMapsPar []       = M.empty
mergeMapsPar [x]      = x
mergeMapsPar (x:y:xs) = runEval $ do
    x'  <- rpar $ M.fromAscListWith (++) $ mergeLists (M.toAscList x)
                                                      (M.toAscList y)
    xs' <- rseq $ mergeMapsPar xs
    rseq x'
    return $ M.fromAscListWith (++) $ mergeLists (M.toAscList x')
                                                 (M.toAscList xs')

mergeLists :: Ord a => [a] -> [a] -> [a]
mergeLists xs [] = xs
mergeLists [] ys = ys
mergeLists (x:xs) (y:ys) = if x <= y
                           then (x : mergeLists xs (y:ys))
                           else (y : mergeLists (x:xs) ys)

validate :: [Record] -> [Record]
validate xs = filter valuesNonZero $
              filter mandatoryFilled $
              filter allKeysPresent xs
    where allKeysPresent xs = case ((M.size $ metas xs) == 7) of
              True -> case ((M.size $ attributes xs) == 32) of
                  True  -> True
                  False -> False
              False -> False

          mandatoryFilled xs =
              (M.lookup "Dauer" (attributes xs) /= Nothing) &&
              (M.lookup "Volumen" (attributes xs) /= Nothing) &&
```

```
77              (M.lookup "AggregationsDatum" (attributes xs) /= Nothing) &&
78              (M.lookup "Eventklasse" (attributes xs) /= Nothing) &&
79              (M.lookup "Eventsource" (attributes xs) /= Nothing) &&
80              (M.lookup "Plattform" (attributes xs) /= Nothing) &&
81              (M.lookup "Tarifzone" (attributes xs) /= Nothing)
82
83          valuesNonZero xs =
84              ((M.lookup "Dauer"   (attributes xs)) /= Just "0") ||
85              ((M.lookup "Volumen" (attributes xs)) /= Just "0")
86
87  getMalformedFiles :: [a] -> [Either b c] -> [a]
88  getMalformedFiles xs ys = map fst $ filter malformed (zip xs ys)
89      where malformed (a,b) = case b of
90                              Left b'   -> True
91                              otherwise -> False
92
93  prettyPrint :: (String,(Int,Int)) -> String
94  prettyPrint (k,(d,v)) = k ++ " => (" ++ show d ++ ", " ++ show v ++ ")"
```

Listing B.2: haskell/Helpers.hs

```
1   module Types
2       ( KeyValuePair(..)
3       , Record(..)
4       , makeKey
5       , makeTuple
6       ) where
7
8   import qualified Data.Map as M
9   import qualified Data.Text as T
10  import Data.Maybe (fromJust)
11
12  type KeyValuePair = (String, String)
13
14  data Record = Record
15      {metas      :: M.Map String String
16      ,attributes :: M.Map String String
17      }
18
19  instance Show Record where
20      show r = "\nRecord:\nMetas:\n" ++
21               concat (M.elems (M.mapWithKey showKVP $ metas r)) ++
22               "Attributes:\n" ++
23               concat (M.elems (M.mapWithKey showKVP $ attributes r))
24
25  showKVP :: String -> String -> String
26  showKVP k v = "\t" ++ k ++ " -> " ++ v ++ "\n"
27
28  makeKey :: Record -> String
29  makeKey x = dt ++ " " ++ ek ++ " " ++ es ++ " " ++ pf ++ " " ++ tz where
30      dt  = fromJust $ M.lookup "AggregationsDatum" (attributes x)
31      ek  = fromJust $ M.lookup "Eventklasse" (attributes x)
32      es  = fromJust $ M.lookup "Eventsource" (attributes x)
33      pf  = fromJust $ M.lookup "Plattform"  (attributes x)
34      tz  = fromJust $ M.lookup "Tarifzone" (attributes x)
35
36  makeTuple :: Record -> [(Int,Int)]
37  makeTuple x = [(dau', vol')] where
38      dau' = read dau :: Int
39      vol' = read vol :: Int
40      dau  = fromJust $ M.lookup "Dauer" (attributes x)
41      vol  = fromJust $ M.lookup "Volumen" (attributes x)
```

Listing B.3: haskell/Types.hs

```
module Grammar (parseRecords) where

import qualified Data.Map.Lazy as M
import qualified Data.Text as T
import Control.Applicative ((*>), (<*))
import Text.Parsec
import Text.Parsec.Text
import Types (KeyValuePair(..), Record(..))

parseRecords :: T.Text -> Either ParseError [Record]
parseRecords input = parse recordFile "" input

recordFile :: GenParser st [Record]
recordFile = do
    contents <- record `sepEndBy1` eol
    eof
    return contents

record :: GenParser st Record
record = do
    recordBegin <?> "start of record"
    metas       <- count 7 (char '#' *> spaces *> metaKvp <* eol)
    attribs     <- count 32 (char 'F' *> spaces *> attribKvp <* eol)
    recordEnd   <?> "end of record"
    return $! Record (M.fromList metas) (M.fromList attribs)

metaKvp :: GenParser st KeyValuePair
metaKvp = do
    key   <- validMetaName <* space
    value <- option "" valueChars
    return $! (key, value)

attribKvp :: GenParser st KeyValuePair
attribKvp = do
    key   <- validAttribName <* space
    value <- option "" valueChars
    return $! (key, value)

eol :: GenParser st String
eol = string "\r\n"
    <|> string "\n"
    <?> "end of line"

recordBegin :: GenParser st String
recordBegin = string "RECORD" <* eol

recordEnd :: GenParser st Char
recordEnd = char '.' <* eol

valueChars :: GenParser st String
valueChars = many1 $ oneOf ".:/_-" <|> alphaNum

validMetaName :: GenParser st String
validMetaName = choice (map (try . string) all_metas)
                <?> "valid meta field name"

validAttribName :: GenParser st String
validAttribName = choice (map (try . string) all_attribs)
                  <?> "valid attribute field name"

all_metas = ["addkey",
    "filename",
    "input_id",
    "input_type",
    "output_id",
    "output_type",
    "source_id"]

all_attribs = ["AccountingStatusType",
    "AggregationsDatum",
    "Anschlussvariante",
    "CFSID",
    "CFSSID",
    "CycleDate",
    "Dauerstatus",
    "Dauer",
```

```
77      "EventTypeID",
78      "Eventklasse",
79      "EventsourcePraefix",
80      "Eventsource",
81      "FlatStatus",
82      "Flatklasse",
83      "InInterfaceID",
84      "InternalID",
85      "LogicalInterface",
86      "MRSEventType",
87      "NASID",
88      "Plattform",
89      "ProcessingDate",
90      "QoS_Class",
91      "RFSID",
92      "Selektionsparameter",
93      "SubscriptionID",
94      "Tarifzone",
95      "Timestamp",
96      "UnixTimestamp",
97      "VBSLKennzeichen",
98      "Verkehrsrichtung",
99      "Volumen",
100     "ZirkCounter"]
```

Listing B.4: haskell/Grammar.hs

Anhang C

C++ Quellcode

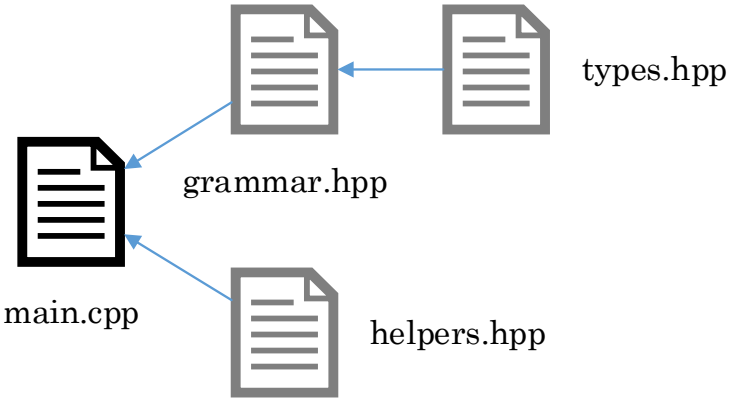

Abbildung C.1: C++ Programmstruktur

```
#include <iostream>
#include <fstream>              /* ifstream             */
#include <algorithm>            /* std::transform       */
#include <numeric>              /* std::accumulate      */
#include <parallel/algorithm>   /* parallel transform   */
#include <parallel/numeric>     /* parallel accumulate  */
#include "grammar.hpp"          /* spirit::qi grammar   */
#include "helpers.hpp"          /* helper functions     */

namespace spirit = boost::spirit;
namespace qi   = spirit::qi;
namespace par = std::__parallel;

using namespace std;

aggregateMap map_(string input);
pair<string,intPair> reduce_(aggregatePair input);
void group_(vector<aggregateMap>* input, aggregateMap* output);

int main (int argc, char** argv)
{
    /* check if argument count is ok */
    if (argc != 2) {
        cerr << "Wrong number of arguments given\nExiting..\n";
        printUsage(argv[0]);
        return -1;
    }

    string path = normalizePath(argv[1]);
    vector<string>* file_list = new vector<string>();

    /* get directory contents */
    if (getDirectoryContents(path, file_list)) {
        cerr << "Failed to read file list.\nExiting..\n";
        return -1;
    }

    /* parse and validate records from files */
    vector<aggregateMap>* dir_result =
        new vector<aggregateMap>(file_list->size());
```

```
41      par::transform(file_list->begin(), file_list->end(),
42                     dir_result->begin(), map_);
43      delete file_list; /* no longer needed at this point */
44
45      /* merge several maps with file contents into one global one */
46      aggregateMap* groups = new aggregateMap();
47      group_(dir_result, groups);
48      delete dir_result; /* no longer needed at this point */
49
50      /* sum up dauer and volume */
51      vector<pair<string,intPair> >* result =
52          new vector<pair<string,intPair> >(groups->size());
53      par::transform(groups->begin(), groups->end(),
54                     result->begin(), reduce_);
55      delete groups; /* no longer needed at this point */
56
57      /* print results */
58      for (const auto& elem : *result) {
59          cout << elem.first << " => ("
60               << elem.second.first << ", "
61               << elem.second.second << ")\n";
62      }
63
64      delete result; /* no longer needed at this point */
65      return 0;
66  }
67
68  aggregateMap map_(string input)
69  {
70      /* create a new parser instance */
71      RecordParser<spirit::istream_iterator> parser;
72      ifstream in;
73      /* container for parsed results of one file */
74      vector<Record> file_result;
75      aggregateMap small_result;
76      in.open(input);
77      /* disable automatic skipping of whitespaces    */
78      /* otherwise this clashes with the parser rules */
79      in.unsetf(ios::skipws);
80      spirit::istream_iterator begin(in);
81      spirit::istream_iterator end;
82
83      /* run <parser> from <begin> to <end>, skip whitespaces with <qi::blank>*/
84      /* store the parsed results in the variable <file_result>               */
85      bool ok = qi::phrase_parse(begin, end, parser, qi::blank, file_result);
86      in.close();
87
88      /* check if parsing succeeded and all input was consumed */
89      if (!ok || (begin != end))
90          cerr << "Failed to parse file: " << input << '\n';
91
92      /* filter out all records that do not satisfy the checks in validate */
93      /* new iterator is returned, pointing to the last valid record       */
94      vector<Record>::iterator validated_end =
95          validate(file_result.begin(), file_result.end());
96
97      for (auto record = file_result.begin(); record != validated_end; ++record) {
98          string key    = make_key(*record);  /* get concatenated key values  */
99          intPair value = make_value(*record);/* get pair of aggregate values */
100         vector<intPair> value_container = { value };
101
102         /* insert() will return a pair of an iterator and a boolean value */
103         /* if false, another element with this key was already existent   */
104         pair<aggregateMap::iterator, bool> insert_result =
105             small_result.insert(make_pair(key, value_container));
106         /* if key already existed, add current pair of values to its vector */
107         if (insert_result.second == false)
108             insert_result.first->second.push_back(value);
109     }
110
111     /* insert map into global vector */
112     return small_result;
113 }
114
115
116
```

```
117 void group_(vector<aggregateMap>* input, aggregateMap* output)
118 {
119     /* neutral element for the left fold */
120     aggregateMap empty = aggregateMap();
121     /* merge seperate file vectors together */
122     *output = par::accumulate(input->begin(), input->end(), empty);
123 }
124
125 pair<string,intPair> reduce_(aggregatePair input)
126 {
127     /* neutral element for the left fold */
128     intPair init = pair<int, int>(0,0);
129     /* sum up integer pairs of the vector */
130     return make_pair(input.first,
131         accumulate(input.second.begin(), input.second.end(), init));
132 }
```

Listing C.1: cpp/main.cpp

```
#include <string>
#include <algorithm>    /* remove_if   */
#include <dirent.h>     /* DIR, dirent */

void printUsage (const char* program);
std::string normalizePath (const char* path);
int getDirectoryContents (std::string path, std::vector<std::string>* result);
intPair make_value (const Record r);
bool containsDuplicateFields (Record r);
bool containsEmptyFields (Record r);
bool containsNullValues (Record r);
std::vector<Record>::iterator validate(std::vector<Record>::iterator begin,
                                       std::vector<Record>::iterator end);

void printUsage (const char* program)
{
    std::cout << "Usage: " << program << " <directory>" << std::endl;
}

std::string normalizePath (const char* path)
{
    int length = strlen(path);
    if (path[length - 1] == '/') {
        return std::string(path);
    }
    return (std::string(path) + "/");
}

int getDirectoryContents (std::string path, std::vector<std::string>* result)
{
    DIR* directory;
    struct dirent* entry;

    /* get directory contents */
    if ((directory = opendir(path.c_str())) != NULL) {
        while ((entry = readdir(directory)) != NULL) {
            /* make sure it's a regular file */
            if ((strcmp(entry->d_name, ".")) ||
                (strcmp(entry->d_name, "..")) ) {
                /* construct complete filename */
                std::string filename = path + entry->d_name;
                result->push_back(filename);
            }
        }
        closedir(directory);
    } else {
        return -1;
    }
    return 0;
}

/* returns a Record's key values as concatenated string */
std::string make_key (const Record r)
{
    return r.attributes.at("AggregationsDatum") + " " +
           r.attributes.at("Eventklasse") + " " +
           r.attributes.at("Eventsource") + " " +
           r.attributes.at("Plattform") + " " +
           r.attributes.at("Tarifzone");
}

/* returns a Record's aggregate values as pair */
intPair make_value (const Record r)
{
    int dauer   = atoi(r.attributes.at("Dauer").c_str());
    int volumen = atoi(r.attributes.at("Volumen").c_str());
    return std::make_pair(dauer, volumen);
}

/* checks if all meta and attribute fields are contained in the Record */
bool containsDuplicateFields (Record r)
{
    if ((r.metas.size() != 7) ||
       (r.attributes.size() != 32))
        return true;
    return false;
```

```
77 }
78
79 /* checks if key and aggregate values are non-empty */
80 bool containsEmptyFields (Record r)
81 {
82     if (r.attributes.at("AggregationsDatum").empty() ||
83         r.attributes.at("Eventklasse").empty()       ||
84         r.attributes.at("Eventsource").empty()       ||
85         r.attributes.at("Plattform").empty()         ||
86         r.attributes.at("Tarifzone").empty()         ||
87         r.attributes.at("Dauer").empty()             ||
88         r.attributes.at("Volumen").empty()           )
89         return true;
90     return false;
91 }
92
93 /* checks if both aggregate values are non-zero */
94 bool containsNullValues (Record r)
95 {
96     if ((r.attributes.at("Dauer").compare("0") == 0) &&
97         (r.attributes.at("Volumen").compare("0") == 0))
98         return true;
99     return false;
100 }
101
102 /* checks if a vector of Records conforms to certain rules                */
103 /* filters out all that do not satisfy those and returns a new end-pointer */
104 std::vector<Record>::iterator validate(std::vector<Record>::iterator begin,
105                                        std::vector<Record>::iterator end)
106 {
107     std::vector<Record>::iterator no_duplicate_fields =
108         std::remove_if(begin, end, containsDuplicateFields);
109     std::vector<Record>::iterator no_empty_fields =
110         std::remove_if(begin, end, containsEmptyFields);
111     return std::remove_if(begin, no_empty_fields, containsNullValues);
112 }
```

Listing C.2: cpp/helpers.hpp

```
#include <map>              /* std::map container */

typedef std::pair<int, int> intPair;
typedef std::pair<std::string, std::string> stringPair;
typedef std::pair<std::string, std::vector<intPair> > aggregatePair;
typedef std::map<std::string, std::vector<intPair> > aggregateMap;
typedef std::map<std::string, std::string> stringMap;

typedef struct Record {
    stringMap attributes;
    stringMap metas;
} Record;

bool operator > (const aggregatePair& a, const aggregatePair& b)
{
    return a.first > b.first;
}

bool operator < (const aggregatePair& a, const aggregatePair& b)
{
    return a.first < b.first;
}

bool operator == (const aggregatePair& a, const aggregatePair& b)
{
    return a.first == b.first;
}

/* merges two sorted vectors of aggregatePairs with linear complexity */
std::vector<aggregatePair> mergeVectors (std::vector<aggregatePair>::iterator first1, std
    ::vector<aggregatePair>::iterator last1,
                                         std::vector<aggregatePair>::iterator first2, std::
    vector<aggregatePair>::iterator last2)
{
    std::vector<aggregatePair> result;
    /* run as long as there're elements left */
    while ((first1 != last1) || (first2 != last2)) {
        /* first vector empty but second non-empty */
        if (first1 == last1) {
            result.push_back(*first2);
            ++first2;
        }
        /* first vector non-empty but second empty */
        else if (first2 == last2) {
            result.push_back(*first1);
            ++first1;
        }
        /* first element equal zu second */
        else if (*first1 == *first2) {
            std::vector<intPair> tmp;
            /* concatenate aggregate value vectors */
            tmp.reserve(first1->second.size() + first2->second.size());
            tmp.insert( tmp.end(), first1->second.begin(), first1->second.end() );
            tmp.insert( tmp.end(), first2->second.begin(), first2->second.end() );
            result.push_back(std::make_pair(first1->first, tmp));
            /* increment both iterators, since we took both head elements */
            ++first1;
            ++first2;
        }
        /* first element smaller */
        else if (*first1 < *first2) {
            result.push_back(*first1);
            ++first1;
        }
        /* second element smaller */
        else if (*first1 > *first2) {
            result.push_back(*first2);
            ++first2;
        }
    }
    return result;
}

/* those two need to be defined in the std namespace */
/* otherwise the compiler won't find them when called from within std */
namespace std {
```

```
75      intPair operator + (const intPair& a, const intPair& b)
76      {
77          return std::make_pair(a.first + b.first, a.second + b.second);
78      }
79
80      aggregateMap operator + (const aggregateMap& a, const aggregateMap& b)
81      {
82          /* convert maps to sorted vectors */
83          std::vector<aggregatePair> vec_a(a.begin(), a.end());
84          std::vector<aggregatePair> vec_b(b.begin(), b.end());
85
86          /* merge vectors together */
87          std::vector<aggregatePair> vec_result =
88              mergeVectors(vec_a.begin(), vec_a.end(), vec_b.begin(), vec_b.end());
89
90          /* create new map from vector */
91          aggregateMap result(vec_result.begin(), vec_result.end());
92          return result;
93      }
94  }
```

Listing C.3: cpp/types.hpp

```
#include <boost/fusion/include/std_pair.hpp>
#include <boost/fusion/include/adapt_struct.hpp>
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/include/qi_repeat.hpp>
#include <boost/spirit/include/support_istream_iterator.hpp>
#include <boost/config/warning_disable.hpp>
#include "types.hpp"

namespace spirit = boost::spirit;
namespace qi = spirit::qi;

/* Adapt Struct for Spirit Parsing */
BOOST_FUSION_ADAPT_STRUCT(
        Record,
        (stringMap, metas)
        (stringMap, attributes)
    )

/* Grammar Definition */
template <typename Iterator>
struct RecordParser: qi::grammar<Iterator, std::vector<Record>(), qi::blank_type> {
    RecordParser() : RecordParser::base_type(start) {
        using qi::string;
        using qi::eol;              /* end-of-line              */
        using qi::eoi;              /* end-of-input             */
        using spirit::repeat;       /* repeat semantic          */

        start = +(record >> eol)    /* std::vector<Record>      */
              >> eoi;

        record =  recordBegin
               >> metaSection       /* stringMap                */
               >> attrSection       /* stringMap                */
               >> recordEnd;

        recordBegin = "RECORD"      /* literal has no attribute */
                    >> eol;         /* eol has no attribute     */
        recordEnd   = '.'           /* literal has no attribute */
                    >> eol;         /* eol has no attribute     */

        metaSection = repeat(7)[metaLine];
        attrSection = repeat(32)[attrLine];

        metaLine = '#'              /* literal has no attribute */
                 >> metaKvp         /* stringPair               */
                 >> eol;            /* eol has no attribute     */

        attrLine = 'F'              /* literal has no attribute */
                 >> attrKvp         /* stringPair               */
                 >> eol;            /* eol has no attribute     */

        metaKvp = validMetaKey      /* string                   */
                >> -value;          /* string (optional)        */
        attrKvp = validAttrKey      /* string                   */
                >> -value;          /* string (optional)        */

        validMetaKey = string("addkey")
                     | string("filename")
                     | string("input_id")
                     | string("input_type")
                     | string("output_id")
                     | string("output_type")
                     | string("source_id")
                     ;

        validAttrKey = string("AccountingStatusType")
                     | string("AggregationsDatum")
                     | string("Anschlussvariante")
                     | string("CFSID")
                     | string("CFSSID")
                     | string("CycleDate")
                     | string("Dauerstatus")
                     | string("Dauer")
                     | string("EventTypeID")
                     | string("Eventklasse")
                     | string("EventsourcePraefix")
```

```
                    | string("Eventsource")
                    | string("FlatStatus")
                    | string("Flatklasse")
                    | string("InInterfaceID")
                    | string("InternalID")
                    | string("LogicalInterface")
                    | string("MRSEventType")
                    | string("NASID")
                    | string("Plattform")
                    | string("ProcessingDate")
                    | string("QoS_Class")
                    | string("RFSID")
                    | string("Selektionsparameter")
                    | string("SubscriptionID")
                    | string("Tarifzone")
                    | string("Timestamp")
                    | string("UnixTimestamp")
                    | string("VBSLKennzeichen")
                    | string("Verkehrsrichtung")
                    | string("Volumen")
                    | string("ZirkCounter")
                    ;

        value = +qi::char_("a-zA-Z0-9_.:/-"); /* value charset  */

        /* Uncomment for debug logs                                  */
        /* start.name("start"); debug(start);                        */
        /* metaSection.name("metaSection"); debug(METAsEction);      */
        /* metaLine.name("metaLine"); debug(metaLine);               */
        /* metaKvp.name("metaKvp"); debug(metaKvp);                  */
        /* validMetaKey.name("validMetaKey"); debug(validMetaKey);   */
        /* attrSection.name("attrSection"); debug(attrSection);      */
        /* attrLine.name("attrLine"); debug(attrLine);               */
        /* attrKvp.name("attrKvp"); debug(attrKvp);                  */
        /* validAttrKey.name("validAttrKey"); debug(validAttrKey);   */
        /* value.name("value"); debug(value);                        */
    }

    /* Rule declarations */
    qi::rule<Iterator, std::vector<Record>(), qi::blank_type> start;
    qi::rule<Iterator, Record(), qi::blank_type> record;
    qi::rule<Iterator, qi::blank_type> recordBegin;
    qi::rule<Iterator, qi::blank_type> recordEnd;
    qi::rule<Iterator, stringMap(), qi::blank_type> metaSection;
    qi::rule<Iterator, stringMap(), qi::blank_type> attrSection;
    qi::rule<Iterator, stringPair(), qi::blank_type> metaLine;
    qi::rule<Iterator, stringPair(), qi::blank_type> attrLine;
    qi::rule<Iterator, stringPair(), qi::blank_type> metaKvp;
    qi::rule<Iterator, stringPair(), qi::blank_type> attrKvp;
    qi::rule<Iterator, std::string(), qi::blank_type> validMetaKey;
    qi::rule<Iterator, std::string(), qi::blank_type> validAttrKey;
    qi::rule<Iterator, std::string(), qi::blank_type> value;
};
```

Listing C.4: cpp/grammar.hpp

Anhang D

Messergebnisse des Desktop-Systems

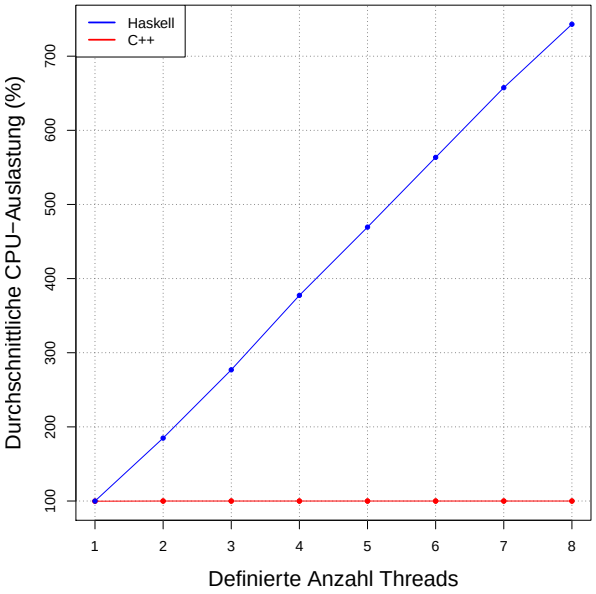

Abbildung D.1: Durchschnittliche CPU-Auslastung bei Verarbeitung von 100, 250, 500 und 750 Dateien (Desktop)

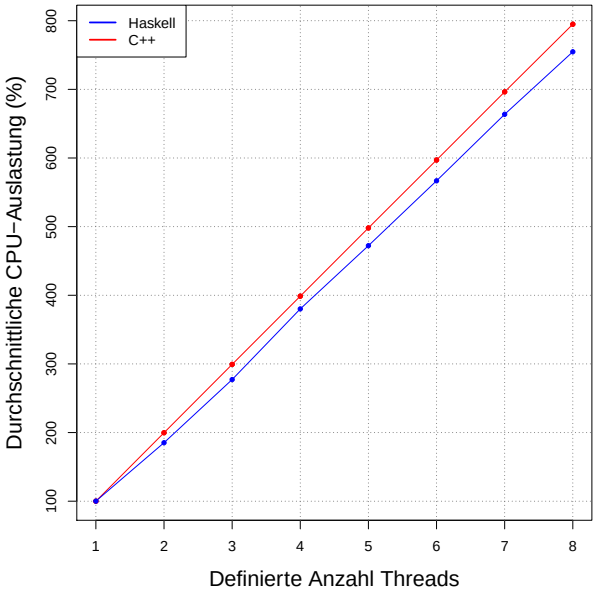

Abbildung D.2: Durchschnittliche CPU-Auslastung bei Verarbeitung von 1.000 Dateien (Desktop)

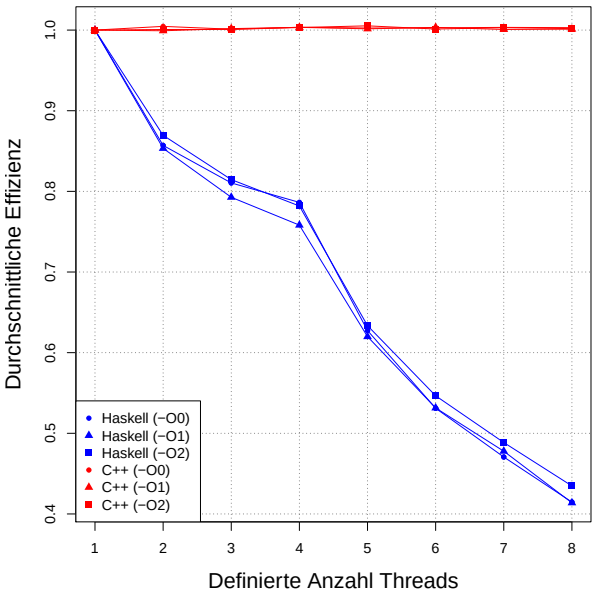

Abbildung D.3: Durchschnittliche Effizienz bei Verarbeitung von 100, 250, 500 und 750 Dateien (Desktop)

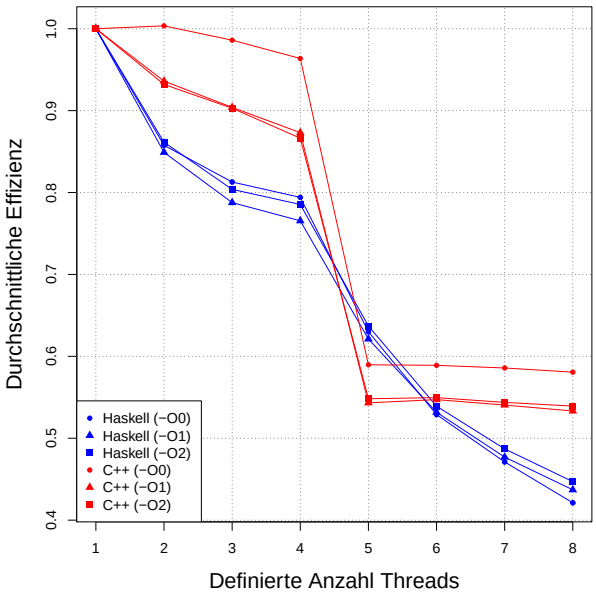

Abbildung D.4: Durchschnittliche Effizienz bei Verarbeitung von 1.000 Dateien (Desktop)

Die Kosten und der parallele Overhead für C++ bei Problemgrößen mit weniger als 1.000 Dateien werden allein der Vollständigkeit halber abgebildet. Aufgrund der sequenziellen Ausführung der Implementierung liefern diese Merkmale keine zusätzlichen Informationen zur Leistung des Programms. Die Kosten in Abbildung D.5 entsprechen der Antwortzeit des Programms, weil für die Anzahl der effektiv genutzten Prozessoren $P = 1$ gilt. Der parallele Overhead in Abbildung D.7 wurde für diesen Fall fest auf den Wert Null gesetzt[27], da keine Parallelisierung stattfindet.

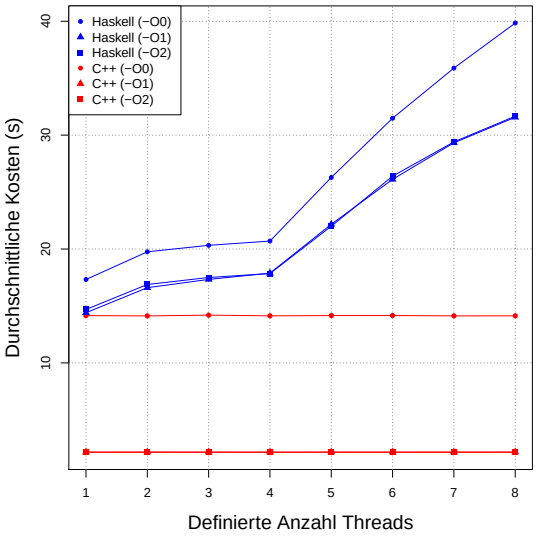

Abbildung D.5: Durchschnittliche Kosten bei Verarbeitung von 100, 250, 500 und 750 Dateien (Desktop)

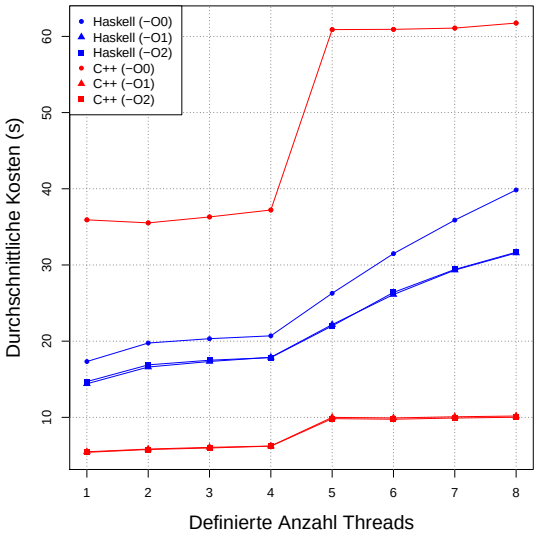

Abbildung D.6: Durchschnittliche Kosten bei Verarbeitung von 1.000 Dateien (Desktop)

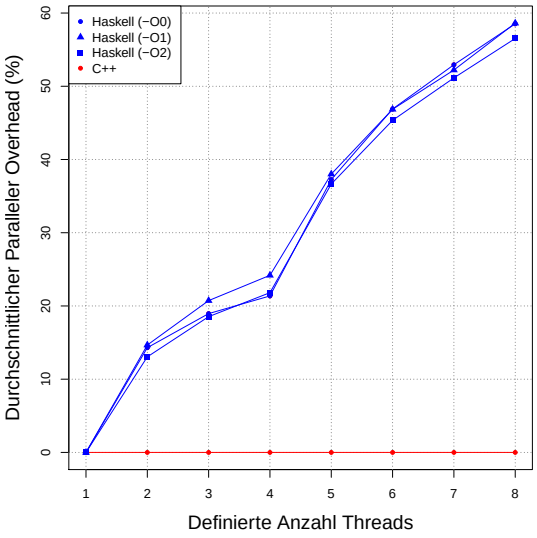

Abbildung D.7: Durchschnittlicher paralleler Overhead bei Verarbeitung von 100, 250, 500 und 750 Dateien (Desktop)

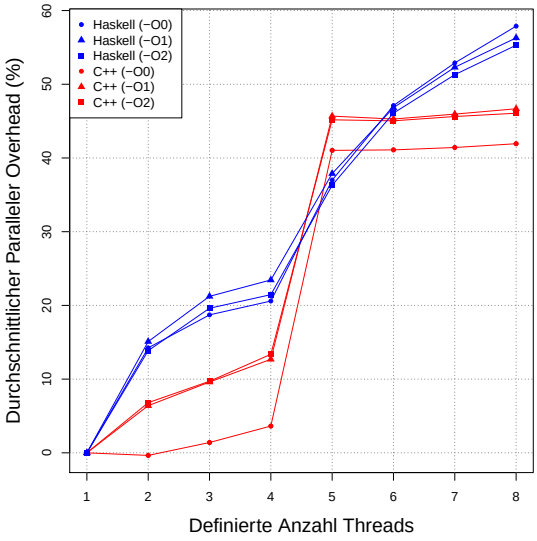

Abbildung D.8: Durchschnittlicher paralleler Overhead bei Verarbeitung von 1.000 Dateien (Desktop)

[27] Die Werte des parallelen Overheads wurden von dem Perl-Skript, welches die Testwerte in die SQLite-Datenbanken einfügt, mit einer statischen Threadanzahl berechnet. Diese Zahlenwerte sind für das C++ programm unterhalb von 1.000 Dateien also nicht repräsentativ.

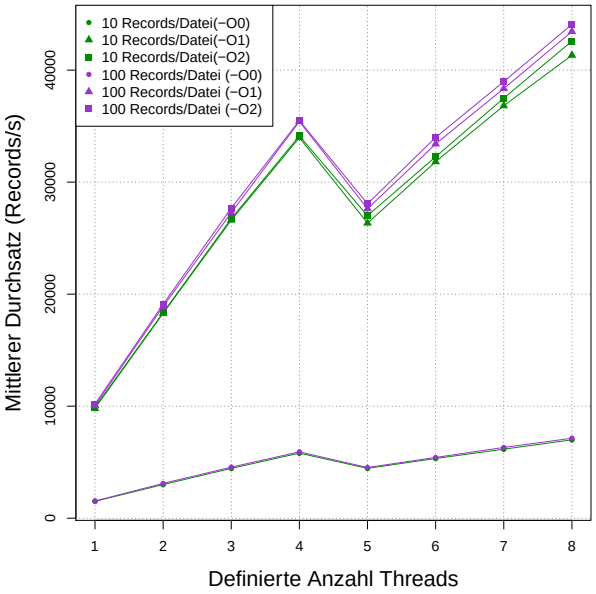

Abbildung D.9: Mittlerer Durchsatz des C++ Programms bei Verarbeitung von 1.000 Dateien (Desktop)

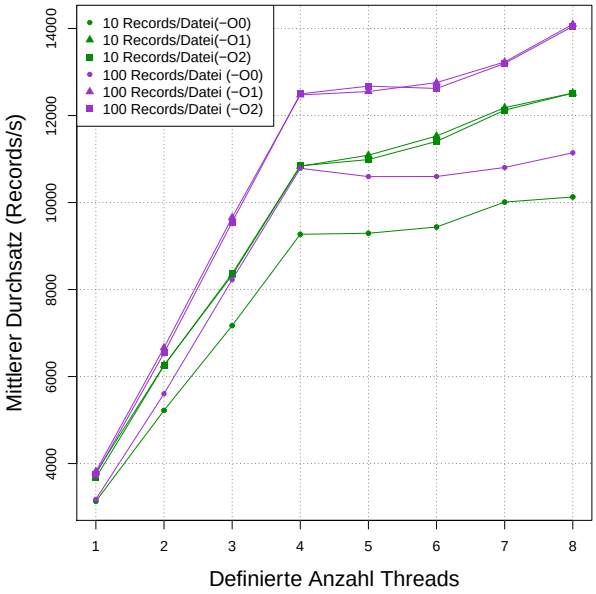

Abbildung D.10: Mittlerer Durchsatz des Haskell Programms bei Verarbeitung von 1.000 Dateien (Desktop)

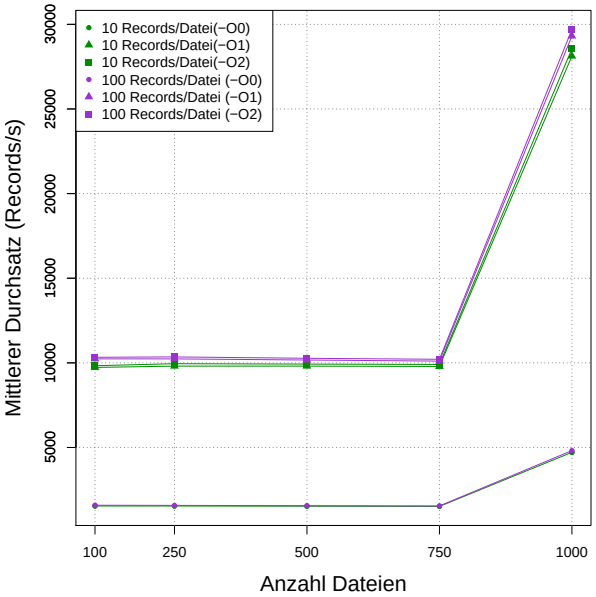

Abbildung D.11: Mittlerer Durchsatz des C++ Programms im Verhältnis zur Dateianzahl (Desktop)

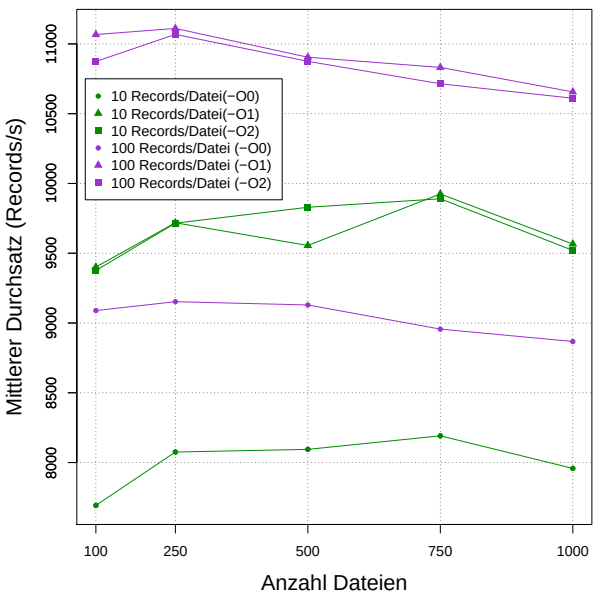

Abbildung D.12: Mittlerer Durchsatz des Haskell Programms im Verhältnis zur Dateianzahl (Desktop)

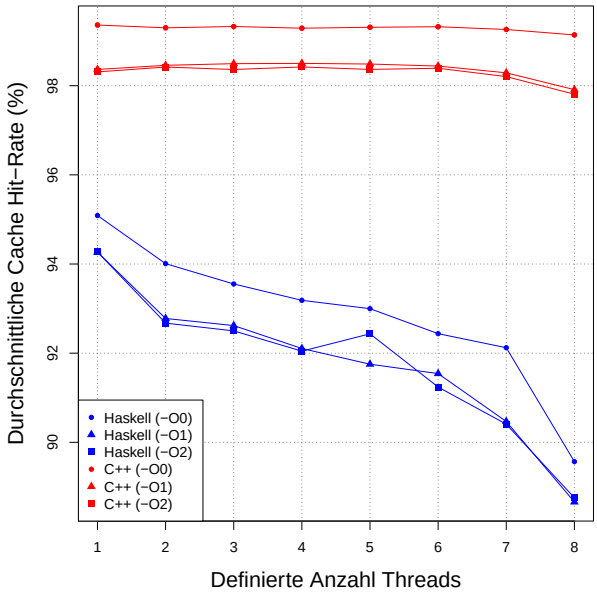

Abbildung D.13: Durchschnittliche Cache Hit-Rate bei Verarbeitung von 1.000 Dateien

Anhang E

Messergebnisse des EC2-Servers

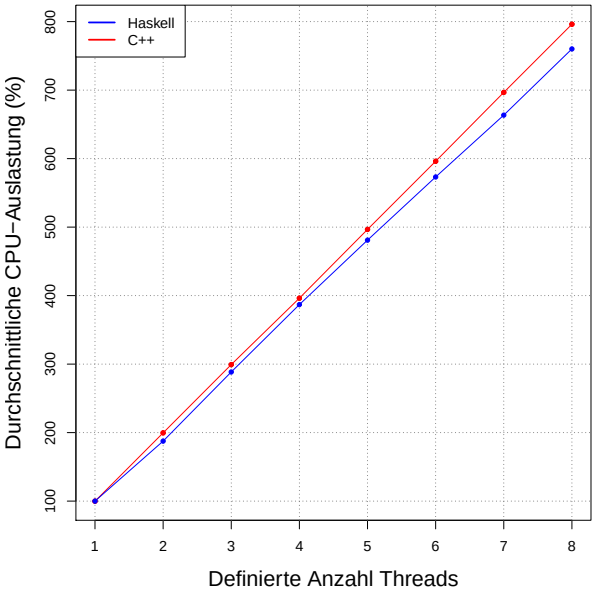

Abbildung E.1: Durchschnittliche CPU-Auslastung (EC2-Server)

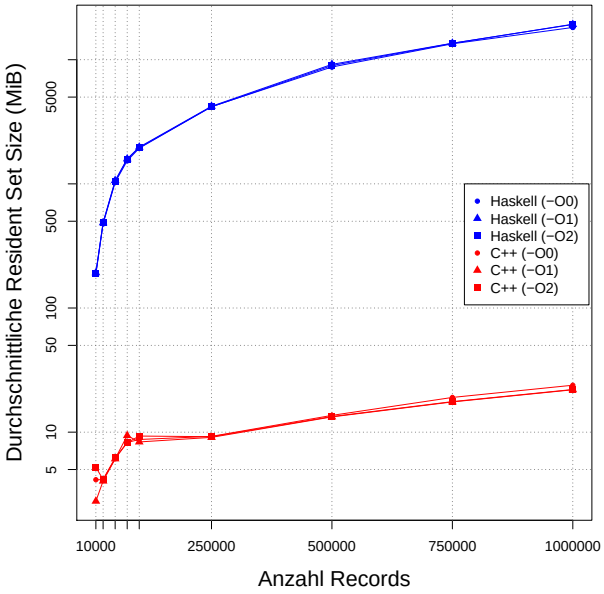

Abbildung E.2: Durchschnittliche Resident Set Size (EC2-Server)

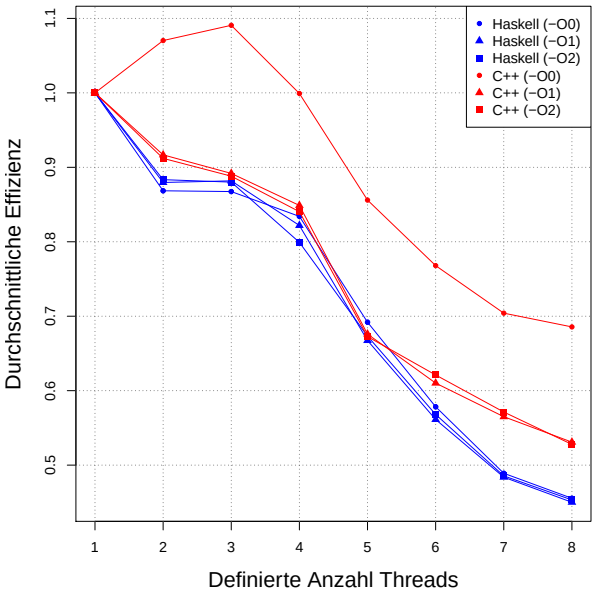

Abbildung E.3: Durchschnittliche Effizienz (EC2-Server)

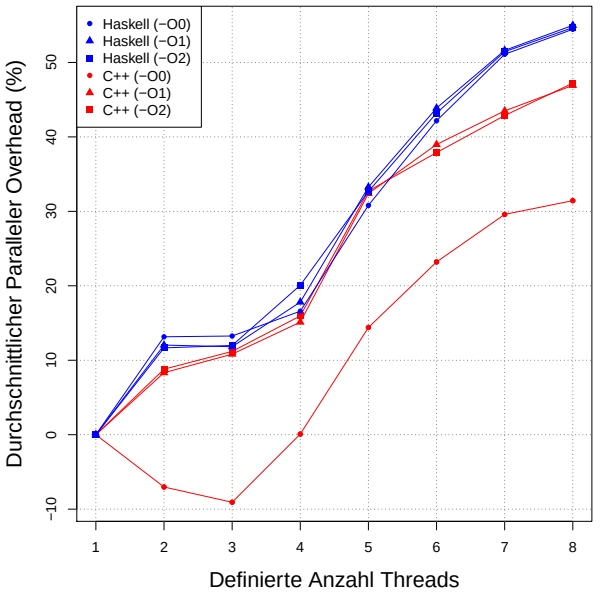

Abbildung E.4: Durchschnittlicher paralleler Overhead (EC2-Server)

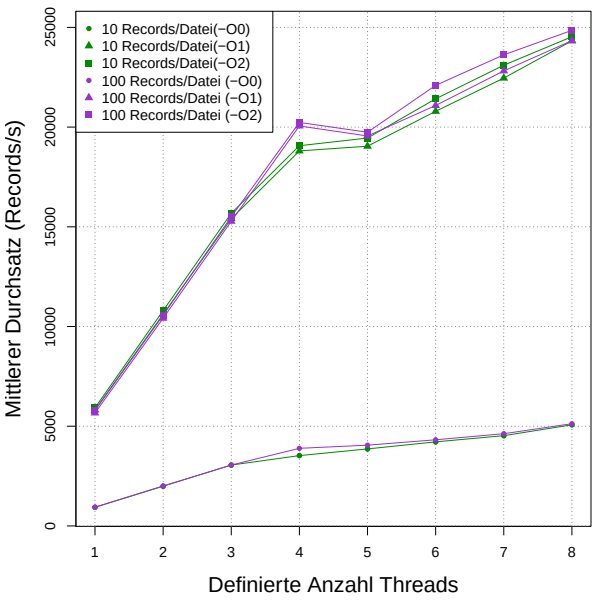

Abbildung E.5: Mittlerer Durchsatz des C++ Programms im Verhältnis zur Threadanzahl (EC2-Server)

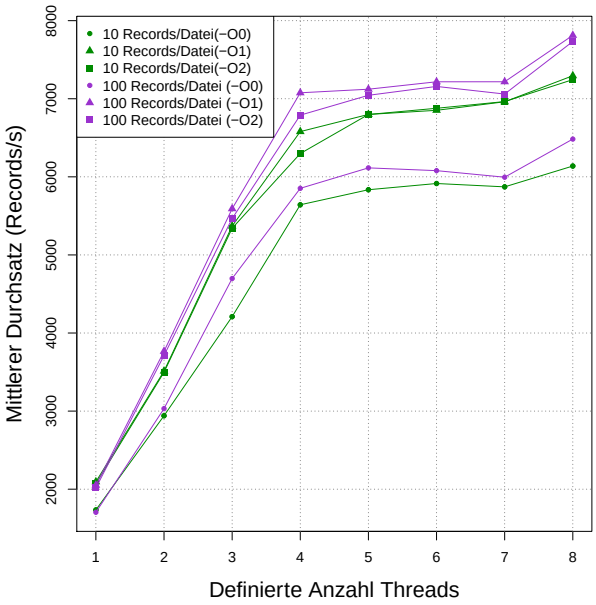

Abbildung E.6: Mittlerer Durchsatz des Haskell Programms im Verhältnis zur Threadanzahl (EC2-Server)

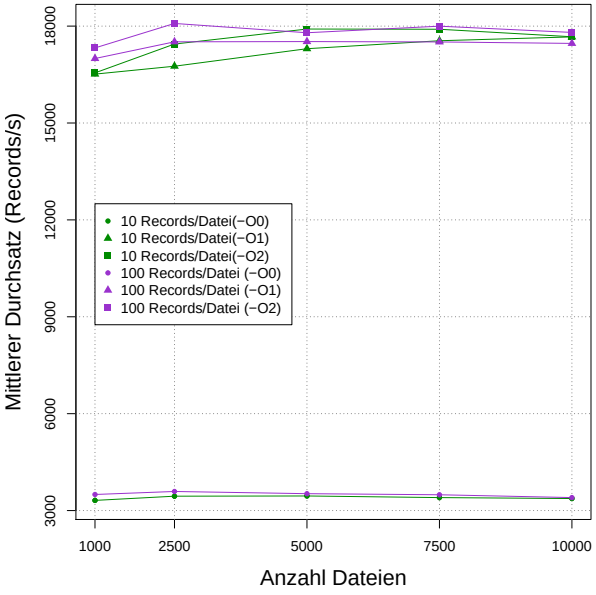

Abbildung E.7: Mittlerer Durchsatz des C++ Programms im Verhältnis zur Dateianzahl (EC2-Server)

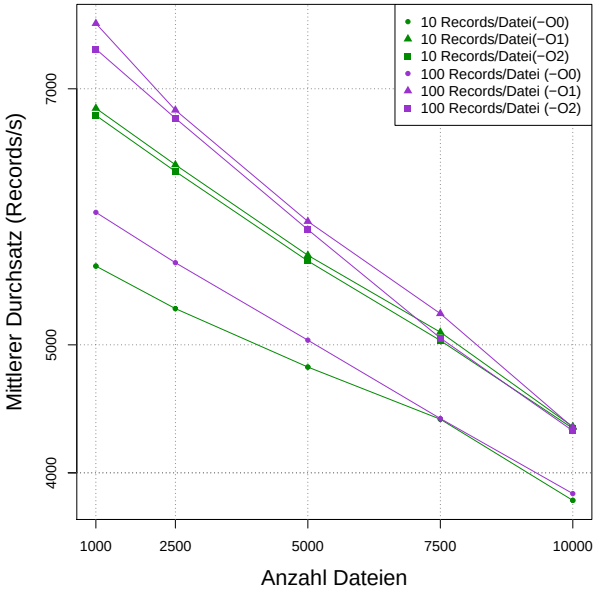

Abbildung E.8: Mittlerer Durchsatz des Haskell Programms im Verhältnis zur Dateianzahl (EC2-Server)

Selbständigkeitserklärung

Hiermit erkläre ich, dass die von mir an der Hochschule für Telekommunikation Leipzig eingereichte Abschlussarbeit zum Thema

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

vollkommen selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Abbildungen in dieser Arbeit sind von mir selbst erstellt oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Hochschule/Universität eingereicht worden.

Leipzig, den 10.10.2016

Hans Christian Rudolph