HTW Dresden Fakultät Informatik/Mathematik

Bericht Projektseminar Parallele Programmierung

Paralleles Text-Retrieval auf einem Rechnercluster

Markus Klemm

WS 2016/2017

Die Verarbeitung und Ausgabe großer Datenmengen ist im heutigen Zeiten von Internet-Suchmaschinen oder globalen Buchungssystemen von Fluglinien nicht mehr Wegzudenken. Die Lösung dieser Aufgaben innerhalb eines Computer-Systems ist weder aus räumlichen Gründen noch in Hinblick auf Datenspeichermengen noch Leistung praktikabel. Ein häufig genutztes Mittel hierfür ist die Verteilung und Parallelisierung, hier durch Nutzung einer Implementierung des Message Passing Interface (MPI) Standards. Es wurden die simple Implementierung einer einfachen Textsuche im Modell der Vektor-Retrival dokumentiert und deren konkrete Laufzeiten im Hinblick auf Knoten-Mengen im zweistelligen Bereich verglichen. Die Gewinne durch die Parallelisierung sowie die aufgetretenen Probleme wurden debattiert und mögliche Verbesserungsmöglichkeiten der Implementierung [2] aufgezeigt.

Inhaltsverzeichnis

1. Zielstellung

Im Rahmen des Projektseminars unter Prof. Dr. Sobe, war es Ziel eine parallele Suche nach Stichworten innerhalb einer Menge von Textdokumenten zu entwickeln. Die Dokumente und Suchanfrage sollten „gemäß dem Vector-Space-Modell repräsentiert werden“.

Für die Implementierung stellte sich damit folgende Aufgaben:

1. Indizieren der Dokumente
2. Vergleich einer Suchanfrage mit den indizierten Dokumenten
3. Sortierung der Vergleichswerte
4. Ausgabe der Vergleichswerte zur Suchanfrage

Durch Messen der benötigten Zeit einer Suchanfrage sollten letztendlich die Geschwindigkeits- und ggf. Durchsatz-Gewinne, in Abhängigkeit zum Verteilungsgrad, in Versuchsreihen festgestellt werden.

2. Modell: Vector retrival

Grundlage des Vefahren ist es, aus allen Wörtern, hier auch Terme genannt, aller erfassten Dokumente, dem sog. Korpus, einen Vektorraum aufzuspannen. Zu jedem Dokument im Korpus wird ein Vektor zugeordnet, bestehend aus je einer Komponente bzw. Dimension die ein Wort bzw. Term repräsentiert. Diese hat einen Faktor $w \in \mathbb{R}$ normalisiert auf Werte im Intervall $[0; 1]$.

In diesem kann der Abstand der Vektoren der Dokumente zueinander oder jeweils zum Vektor einer Suchanfrage gebildet werden. Der Vorteil des Verfahren begründet sich aus dem Zipfschen Gesetz, welches selten auftretenden Wörtern eine höhrere Relevanz beimisst, dazu [13, Seite 54]:

> The constrained relationship between the frequency of a word in a corpus and its rank gained wide attention in the 1930s and 1940s through the work of George Kingsley Zipf (1902-1950),a professor of philology at Harvard University. The name “Zipf’s law” has been given to the following approximation of the rank-frequency relationship:

$$rf = c \tag{1}$$

> where r is the rank of a word-type, f is the frequency of Occurrence of the word-type, and c is a constant, dependent on the corpus (often around one-tenth of the total size of [i.e., number of word-tokens in] the corpus).

[12, vergl. Seite 123]

Als Modell für die Dokumentenähnlichkeit wurde das tf-idf—term frequency-inverse document frequency Modell gewählt. Wir definieren den Gewichtsvektor eines Dokumentes d als $\vec{v}_d = [w_{1,d}, w_{2,d}, \ldots, w_{N,d}]^T$, mit:

$$w_{t,d} = \mathrm{tf}_{t,d} \cdot \mathrm{idf}_{t,D} \tag{2}$$

mit der lokalen Häufigkeit des Wortes t im Dokument d

$$\mathrm{tf}_{t,d} = |\{t \in d\}| \tag{3}$$

und der globalen inversen Häufigkeit des Wortes im Korpus D, logarithmisch skaliert:

$$\mathrm{idf}_{t,D} = \log \frac{|D|}{|\{d' \in D \,|\, t \in d'\}|} \tag{4}$$

Die Ähnlichkeit zwischen zwei Dokumenten bestimmt schließlich aus dem Winkel zwischen zugehören Vektoren:

$$\theta = \arccos \frac{\mathbf{d_1} \cdot \mathbf{d_2}}{\|\mathbf{d_1}\| \, \|\mathbf{d_2}\|} \tag{5}$$

Weit verbreitet ist jedoch direkt das Funktionsargument von arccos zu nutzen, auch Kosinus-Ähnlichkeit oder Cosine similarity genannt.

$$\mathrm{sim}(d_1, d_2) = \cos\theta = \frac{\mathbf{d_1} \cdot \mathbf{d_2}}{\|\mathbf{d_1}\| \, \|\mathbf{d_2}\|} = \frac{\sum_{i=1}^{N} w_{i,d_1} w_{i,d_2}}{\sqrt{\sum_{i=1}^{N} w_{i,d_1}^2} \sqrt{\sum_{i=1}^{N} w_{i,d_2}^2}} \tag{6}$$

Eine Ähnlichkeit von 0 bedeutet somit, dass beide Dokumente keinen Bezug zu einander haben, eine Ähnlichkeit von 1 hingegen bedeutet maximale Ähnlichkeit. Abfragen, auch Querys genannt, können ebenso wie ein Dokument verglichen werden. Der einzige Unterschied besteht im Rechenmodell darin, sie nicht in den Korpus einzubeziehen. Relevant wäre dies also nur bei den Seiteneffekten der globalen Bewertungsfunktion (2).

3. Entwurf

3.1. Indizierung

Da die geplante Implementierung mittels C++ erfolgte, wurde ein objektorientierter Entwurf gewählt 1. Erste Entscheidung war, das IO-intensive Zählen der Wörter als ersten Prozess zu separieren. Hierfür wurde die Klasse word_counter benannt. Diese nimmt ein Dokument oder Query in ihrem Konstruktor entgegen. Dabei ist es irrelevant ob das

Dokument als Datei oder Nutzereingabe vorliegt, da jede von std :: istream abgeleitete Eingabe akzeptiert wird.

Da in diesem Abschnitt des Projekts noch nicht festgelegt war, ob bereits die Arbeit des Zählens und Stemmings parallelisiert werden sollte, sind all diese schwergewichtigen Funktionen vom Konstruktor getrennt und idempotent. Um die Datenhaltung möglichst du reduzieren, ist die Funktion des Stemming bereits fest in den Zählprozess, als Funktionsaufruf, eingebaut. Intern wird von word_counter für die Zuordnung Wort $\rightarrow$ Anzahl eine std :: map benutzt. Dies halt den Vorteil, dass die Wörter, bzw. nach dem Stemming, Wortäquivalenzklassen, einzigartig und sortiert, sowie der Zugriff auf den Zähler in $\mathcal{O}(\log n)$ garantiert ist. Der Code ist allerdings Container-agnostisch geschrieben, so dass auch std :: unordered_maps benutzt werden könnten, welche die Garantie des sortiert Seins gegen eine annähernd $\mathcal{O}(1)$ Laufzeit-Komplexität eintauschen.

3.2. Gewichtung

Nach dem nun alle Wörter gezählt sind, also der Faktor aus Gleichung (3) bekannt ist, wird eine Referenz zu den Werten an den Konstruktor der Klasse weighter gegeben. Da dieser die gesamte Gewichtung inklusive (4) bewerkstelligen soll, benötigt er den Parameter D, also den Korpus. Hierbei wurde auf globale Variablen verzichtet, sondern eine weitere notwendige Referenz für den Konstruktor dieser Klasse hinzugefügt. Diese Referenz zeigt auf einen Typ abgeleitet von der Basisklasse global_weight_state. Aufgabe einer solchen Klasse ist die Verwaltung des nötigen Zustandes von D. In diesem Fall, die Anzahl aller Dokumente $|D|$ sowie die Anzahl aller Dokumente d' welche das Wort t beinhalten. Wie diese Bereitstellung bewerkstelligt wird, bleibt jener Klasse überlassen. Somit kann die spätere Sychronisationslogik zwischen den einzelnen Nodes, per Objektpolymorphie ersetzt werden, ohne Änderungen an der Dokumentenverwaltung oder der Berechnung zu erfordern. Im nicht parallelen Fall, wäre dies simpel das Inkrementieren eines Dokumenten-Counters sowie eines Wort-Dokumenten-Counters, vorzugsweise in einer std :: map ähnlich wie bereits bei word_counter.

3.3. Abstand

Die eigentliche Berechnung von (2) wäre somit letztendlich Aufgabe von weighter. Auch dessen Funktionen zur Berechnung wurden idempotent bezüglich den Rechenaufwandes implementiert und für spätere Optimierungsmöglichkeiten ein getrennter Start von lokalen und globalen Berechnungsteil ermöglicht. Denkbar wären Synergieeffekte bei zeitlich konzentrierter Berechnung mehrerer entweder lokaler oder globaler Gewichte.

Die Abstandsermittlung nach (6) erledigt schlussendlich die Funktion calc_distance, welche wie ihr mathematisches Vorbild, exakt 2 Dokumente (bzw. ein Dokument und eine Query) in Form eines weighters bzw. der Rückgabe eines Jenen finalen Berechnungs-Methode benötigt.

4. Implementierung

4.1. Werkzeuge

Ursprünglich geplant war eine Implementierung nach [**?**, ISO/IEC 14882:2014], doch aufgrund der Anforderung auf dem Server-Hosts bzw. Pool-PCs der HTW Dresden zu kompilieren, mussten einige Rückschritte in Richtung des vorherigen Standards, auch C++11 genannt, gemacht werden. Dies ergab sich aus den älteren Versionen des Compilers und C++–Standardbibliothek, welche nicht in allen Belangen dem aktuell gültigem Standard entsprachen. Dies waren unter anderem:

- Fehlende std :: make_unique<T>() Funktion
- Probleme bei Initialisierung von Membern in der member intitializer list mit Funktionen
- Unvollständige Unterstützung von generic lambda expressions

Auf dem Entwicklungsrechner wurde [8, Apple Clang] in der Version 8.0.0 mit der C++–Standardbibliothek [4, libc++] 3700 benutzt.

Die umfangreiche Bibliothek [1, Boost] in der Version 1.59 wurde zunächst für Unit-Tests benutzt, jedoch später auch als Abstraktionsschicht für MPI. Auch im Hinblick für eventuell später notwendigeres umfangreicheres und differenziertes Logging schien die Einbindung von Boost sinnvoll. Als MPI Implementierung wurde [5, MPICH] gewählt in der Version 3.2.

Als Buildsystem wurde [11, CMake] gewählt, welches aus einer Meta-Beschreibung des Projektes, folgende Aufgaben erfüllte:

- Ermittlung und Ersetzung der Pfade von
  - Bibliotheken
  - Kompiler
  - Linker
- Ermittlung der nötigen Flags und Argumente
- Erzeugung von Projektdateien oder wahlweise Make-Scripten, incl. Unittests

CMake respektive seinen Modulen gelang es selbst Boost und MPI im lokalen Benutzer-Verzeichnis auf dem Server-Host der HTW zu erkennen, da jene Bibliotheken mangels Existenz und Root-Rechte, lokal gebaut und installiert werden mussten. Somit konnten die Schritte zum Kompilieren und Ausführen der Unit-Tests auf

```
cmake
make
make test
```

minimiert werden. Als integrierte Entwicklungsumgebung wurde [10, CLion] in der 2016. Version benutzt. Zur Versionsverwaltung wurde [3, Git] benutzt. Als Distributionsplattform und Projektunterstützung wurde [9, Github] genutzt.

4.2. Verlauf

Der Beginn der Implementierung konzentrierte sich um Experimente ob denn MPI den Erwartungen entsprechend einsatzfähig war. Im Anschluss wurden allgemeine Verwaltungs- und Diagnoseaufgaben einer MPI Applikation, insbesondere einer in C++ entwickelter eingerichtet. Dazu gehörten nach der Installation der C-Errorhandler, als auch eines entsprechenden Handlers der anstelle von std :: terminate installiert wurde. Somit war garantiert, das im Fehlerfall als auch bei u.a. nicht gefangenen Exceptions, stets der saubere Abbruch im gesamten Schwarm mittels mpi_abort durchgeführt wurde.

Als erste problemdienliche Funktion ist exemplarisch die Funktionalität alle Dateien in einem Verzeichnis zu indizieren. C++ besitzt leider erst ab dem voraussichtlichem Standard C++17 Funktionen in der Standardbibliothek für Verzeichnismanipulationen, doch wurde für eben jene Funktion Boost genutzt, welche diese bereits im voraussichtlichem zukünftigem Stil anbietet. So muss später nur die Zeile des namespaces als die des Headerprefixes im Code ausgetauscht werden.

Eben jener Stil, des möglichst einfachen Veränderung von Implementierungs–Details wurde bei der weiteren Entwicklung stets verfolgt. So sind alle konkreten Typen der Indizierungsklasse durch einen zentralen Typalias in der Klasse defniert. Die darauf aufbauenden Klassen wiederum benutzen ebenfalls nur transitive Ableitungen dieses Types. Diese Meta-Programming Schreibweise kann allerdings auch zu hartnäckigen Fehlern durch relativ unbewusste implizite Typkonversionen führen. Dies war bei der Entwicklung der Fall, wurde aber durch das konsequente Unit-Testing früh abgefangen. Diese Fälle liesen sich auch durch die Erweiterung der Typaliase, zu Typklassen mittels Typsystem abfangen. Nachteilig ist dabei allerdings der Mehraufwand durch dann zu implementierende Operatorüberladungen.

Auch zeigte sich der Vorteil als zur Vereinfachung der Berechnung des Abstandes (6), nicht alle Dimensionen der Korpus, sondern nur der Vereinigung aller Dimensionen der 2 Vektoren berechnet wurden. Dabei unterlief dem Autor in der ersten Implementierung der Fehler statts der Vereinigung, die Schnittmenge der Dimensionen zu bilden. Durch konsequente Nutzung des Iterator und Algorithm Konzeptes der C++-Standardbibliothek, belief sich spätere Korrektur des Quelltextes auf exakt den Austausch des Namens eines Funktionsaufrufes, von std :: set_intersection zu std :: set_union.

4.3. Parallelisierung

Probleme bei der Benutzung von MPI stellten sich bei der Benutzung des MPI Async-Interfaces dar. Ungewohnt war auch der inkonsistente Umgang der Bibliothek mit NULL bzw. MPI_REQUEST_NULL. Nach einigen Experimenten mit der synchronen und kollektiv synchronen API, wurde der Versuch einer eigenen C++ Abstraktion für individuelle asynchrone MPI Kommunikation entwickelt. Jedoch tauchten dabei weitere Probleme auf. Einerseits die Zerlegung von Objekten in einen Binärstrom, auch Serialisierung genannt, als auch dem Autor unerklärliche Fehler, bei Benutzungsversuchen zur Laufzeit (Segmentationfaults). Trotz Fehlersuche schien ein weiteres Verfolgen dieses Ansatzes nicht praktikabel, insbesondere als geklärt werden konnte, dass die Benutzung einer be-

reits existierenden MPI Abstraktionsschicht, nicht gegen den Sinn der Aufgabenstellung verstößt.

So wurde die weitere Entwicklung mithilfe von boost::mpi fortgesetzt. Als erstes Bestachen die Ähnlichkeiten mit dem Entwurf der Abstraktion des Autors, wie z.B. die fast wortlaute Definition einer mpi_exception. Größte Vorteile waren das typsichere Interface als auch die Fähigkeit, einfache Container wie std :: vector von quasi willkürlichen Typen entgegen zu nehmen. Wobei die akzeptieren Typen von Haus aus die sogenannten Built-In-Typen, wie **double** als auch Typen zu denen passende Template-Spezialisierungen für boost:: serialize , welches von boost::mpi benutzt wird, bereitgestellt werden, sind.

5. Laufzeit-Messungen

5.1. Vorbetrachtung

Gemessen wurde die Laufzeit innerhalb eines PC-Labors innerhalb der HTW Dresden, bestehend aus über 20 x86-64 Dell Optiplex 7010 mit Intel Core i5-3570 Quadcores getaktet nominell bei 3,4 GHz, und ca. 16 GB RAM. Betrieben wurden diese mit Linux der Distribution open Suse Leap42.1 mit Kernel 41.36-44-default. Test-Abfrage war stets „Roosevelt made this quote to Washington with Germany“. Zur Netzwerkverbindung wurden keine genauen Informationen ermittelt, jedoch scheint es ausreichend geswitchtes Gigabit-Ethernet in einer Stern-Topologie anzunehmen.

Durchgeführt wurden 3 Messreihen, welche sich durch die zu durchsuchende Dokumentenbasis unterschieden:

1. 1084 Textdokumente aus dem öffentlichen Gutenberg Projekt
2. Eine Sammlung von Artikeln und Büchern konvertiert zu Textdateien, 49 Stück
3. 30 Modulbeschreibungen bereitgestellt vom Themen–Betreuer

Hierbei stellte sich heraus das die kleineren 2 Datenmengen kaum für eine akkurate Messung der Laufzeit geeignet waren, da sich die Laufzeiten bereits auf der Entwicklungsmaschine auf unter 100ms bzw 2ms einstellen. Im Hinblick auf den zufälligen und systematischen Einfluss von Laufzeitsumgebungs–Variablen und Roundtrip–Zeiten selbst innerhalb von lokalen Netzwerken schienen erst die Größenordnungen einiger Sekunden der größten Dokumentenbasis als praktikabel. Daher wird bei der weiteren Betrachtung vorrangig auf diese Bezug genommen und auf einen Plot der Kleinsten verzichtet.

Die eigentliche reale Zeit wurde mithilfe der std::chrono:: high_resolution_clock bestimmt, welche eine in [6, §20.12.7.3] definiert ist. Diese bietet die portable Möglichkeit in C++ Zeiten mit der höchsten der Laufzeitumgebung zur Verfügung stehenden Auflösung zu messen. Während angenommmen wird, dass der Mehraufwand für Funktionsaufrufe und Systemcalls hierfür kleiner als eine einstellige Millisekunden-Anzhal ist, so ist dies nachweißbar nicht der Fall für ein ggf. sofortiges Vermerken des Starts der Zeitmessung auf die Standardausgabe des Programms, auch nicht in der gepufferten Version im

Sinne von std :: cout. Um die Benutzung der Zeitmessung und der Ausgabe der Zeitmessung außerhalb der eigentlich Messung zu vereinfachen wurde eine Hilfsklasse namens time_utility entwickelt und benutzt.

Jede Messreihe setzte sich aus einer schrittweisen Erhöhung der Anzahl der genutzten Nodes, genauer des -n Parameters von mpiexec, mit 3-facher Wiederholung zu Minimierung von zufälligen Fehlern zusammen. Dem Leser wird jedoch bis auf Ausnahmen das scheinbare Fehlen der Error-Bars auffallen. Dies ist durch die geringe Standardabweichung von teilweise weniger als einem Digit begründet. Mpiexec wurde dabei eine Liste mit 17 Hostnames übergeben, weshalb eine Nodeanzahl > 17 Parallelität auf Hostebene bedeutet, und folglich bei den eingesetzt Quadcores bei einer Anzahl $> 17 * 4 = 68$ Oversubscription, also eine größere Anzahl von Prozessen als CPU-Kerne. Diese Schwellen wurden in den Diagrammen mit vertikalen Strichen gekennzeichnet. Der Host welcher mpiexec Ausführt und die Benutzerschnittstelle zur Verfügung stellt und die Ergebnisse der anderen Nodes vergleicht ist nicht Teil der Hostliste und nimmt damit eine Sonderrolle gegenüber den anderen Nodes ein.

5.2. Diskussion

Der Verlauf von Abbildung 2 zeugt von einem hyperbolischem Abfall der Abfragezeiten zum Wachstum der benutzen mpi-Prozesse. Auch scheint sich die Zeit schon merklich vor der Schwelle 1 Node pro physische Node nicht mehr signifikant zu verringern. Wie Erwartet erhöhte sich die Ausführungszeit ab der Oversubscription-Schwelle wieder.

Auffällig ist ein ausgeprägtes Plateau im Verlauf von Abbildung 3, für welches der Autor keine sichere Erklärung hat. Ein Erklärungsversuch liegt in einer ungünstigen Dateigrößen-Verteilung auf den Nodes. Die Zugehörigkeit Datei zu Node, wird durch eine einfache Datei-Position mudolo Node-Anzahl gebildet, welches beim Anblick von Abbildung 5 problematisch sein könnte. Auch scheinen Schedeuling-Anomalien wahrscheinlich.

5.3. Auswertung

Im Hinblick der Verringerung der Anfragezeit durch Erhöhung der beteiligten physischen Knoten konnte eine Anfangs fast lineare Leistungssteigerung mit einem allerdings recht früh eintretendem Sättigungspunkt festgestellt werden. Dieser korreliert auch recht gut mit der Größe der Dokumentenbasis. Jedoch gibt es anscheinend, je nach Dokumentenbasis, einen Grenzwert gegen welchen die Ausführungszeit trotz Node-Anzahl-Erhöhung strebt. Dieser ist auch signifikant höher als die zu erwartende Latenz bzw. Round-Trip-Time des Netzwerkes.

Mögliche Ursachen wären

- zeitlich konstante Aufwendungen für jede Suche auf jeder Node wie
  - RAM-Seitenauslagerungen des Linux-Kernels durch die relativ große Datenstruktur, bei 4 Nodes, ca. 4.5 GibiByte per Node für die 1. Messreihe
  - Sequentielles Abarbeiten der einzelnen Wörter

- zeitlich konstante Latenzen in der Schwarm Koordination, durch Kollektive MPI Kommunikation oder verteilte Locks

Anhand der Datenlage scheint es gerechtfertigt anzunehmen, dass offensichtlich größere Datenmengen bei angenommener konstanter Leistung pro Knoten mit einer Vergrößerung des Schwarm bewältigt werden können. Doch nähert man sich schnell einem Grenzwert, welcher seinen Ursprung in den inhärenten Eigenschaften des Gesamtsystems zu haben scheint. Allerdings ist auch ein größeres hier globales Plateau möglich. Messreihen mit signifikant größeren Datenmengen scheint deshalb die nächst dringeste Fragestellung.

5.4. Ausblick und Zusammenfassung

Im Bezug auf diese Studie sieht der Autor folgende weiterführende Aufgaben- bzw. Fragestellungen:

1. Messreihen mit wie bereits erwähnt signifikant größeren Datenmengen für sinnvolle Bewertungen des Durchsatz-Gewinns
2. Messreihen mit einer größeren Anzahl an physischen Knoten
3. Messreihen mit variierenden Netzwerktechnologien und Typologien
4. Variationen im Ablauf der globalen Korpus-Modellierung mittels MPI, durch unterschiedliche Klassen-Spezialisierungen von global_weight_state_t

Hierbei kann das bestehende Projekt inklusive [2, Quellcode], quasi frei (MIT-Lizenz) genutzt werden.

In Bezug auf die hohe Latenzverbesserung schon bei einer geringen Knoten-Anzahl stellt sich die Parallelisierung der Aufgabenstellung allgemein als auch die Nutzung von MPI im Speziellen trotz Mehraufwand als lohnenswert herraus. Jedoch war praktikable Lösung erst mit weiteren Hilfsmitteln, hier boost::mpi im modernen Maßstab möglich. Eine direkte Nutzung der MPI-Schnittstelle scheint nicht mehr zeitgemäß noch Vorteile irgendeiner Art einher zu bringen. Dies fügt sich gut in der Sprachdiskussion von C im Vergleich zu C++, wobei hier stellvertretend [7, Technical Report on C++ Performance von Dave Abrahams et al.] genannt sei, welche Geschwindigkeits-Gewinne durch Sprachen-inhärente Vorteile in [7, 5.3.4] von C++ zeigte. Die Verminderung von Fehlern durch die neuen sprachlichen Mittel von C++ zur Schnittstellen-Gestaltung in Form von boost::mpi hat 4.3 gezeigt.

Abbildungsverzeichnis

Literatur

[1] *Boost C++ Libraries*. http://www.boost.org

[2] *Code-Repository der Implementierung*. https://github.com/Superlokkus/dist_vec_ret

[3] *Git –fast-version-control*. https://git-scm.com

[4] *libc++ C++ Standard Library*. https://libcxx.llvm.org

[5] *MPICH*. https://www.mpich.org

[6] 22, ISO/IEC JTC 1.: Information technology — Programming languages — C++ / International Organization for Standardization. Geneva, CH, 12 2014 (ISO/IEC 14882:2014). – Standard

[7] 22, ISO/IEC WG 2.: Technical Report on C++ Performance / International Organization for Standardization. Geneva, CH, 02 2006 (ISO/IEC TR 18015:2006(E)). – Technical Report

[8] Apple: *LLVM Compiler Overview*. https://developer.apple.com. Version: 2017. – [Online; accessed 2017-02]

[9] Github: *Github*. https://github.com

[10] Jetbrains: *CLion A cross-platform IDE for C and C++*. https://www.jetbrains.com/clion/

[11] Kitware: *CMake*. https://cmake.org

[12] Salton, G. ; Wong, A. ; Yang, C. S.: A Vector Space Model for Automatic Indexing. In: *Commun. ACM* 18 (1975), November, Nr. 11, 613–620. http://dx.doi.org/10.1145/361219.361220. – DOI 10.1145/361219.361220. – ISSN 0001–0782

[13] Wyllys, Ronald E.: *Empirical and theoretical bases of Zipf's law*. 1981-01-01

A. Abbildungen

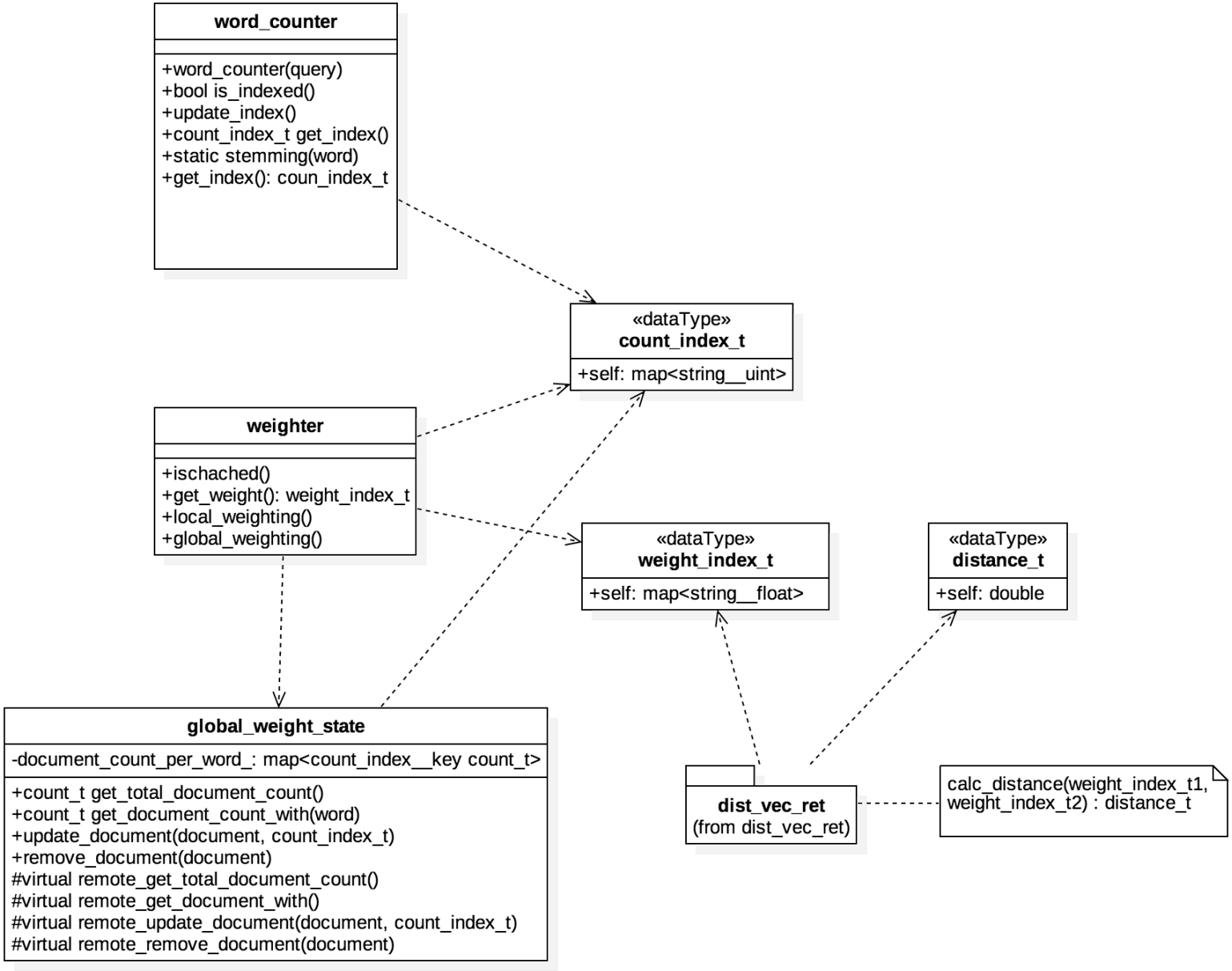

Abbildung 1: Klassendiagramm

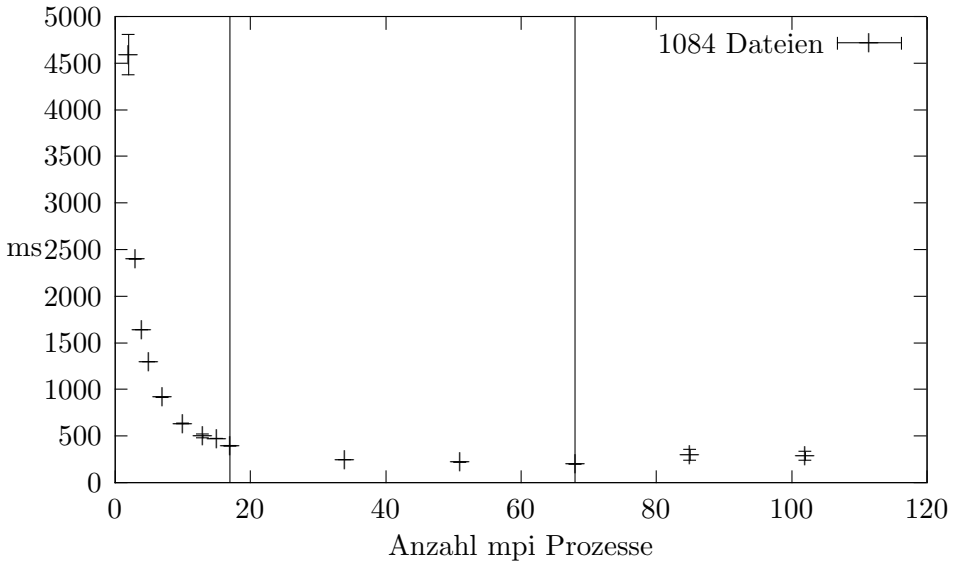

Abbildung 2: Query Laufzeiten 1. Messreihe

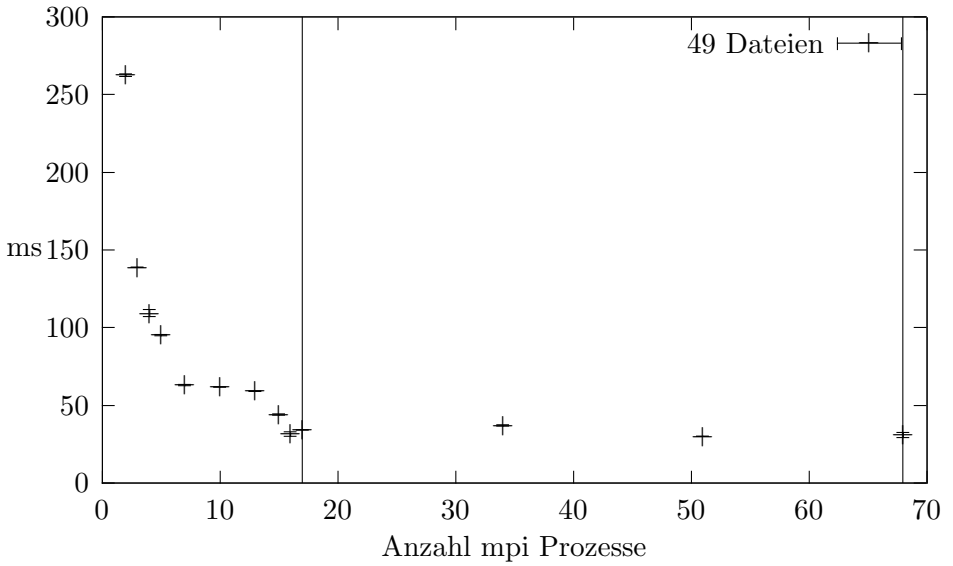

Abbildung 3: Query Laufzeiten 2. Messreihe

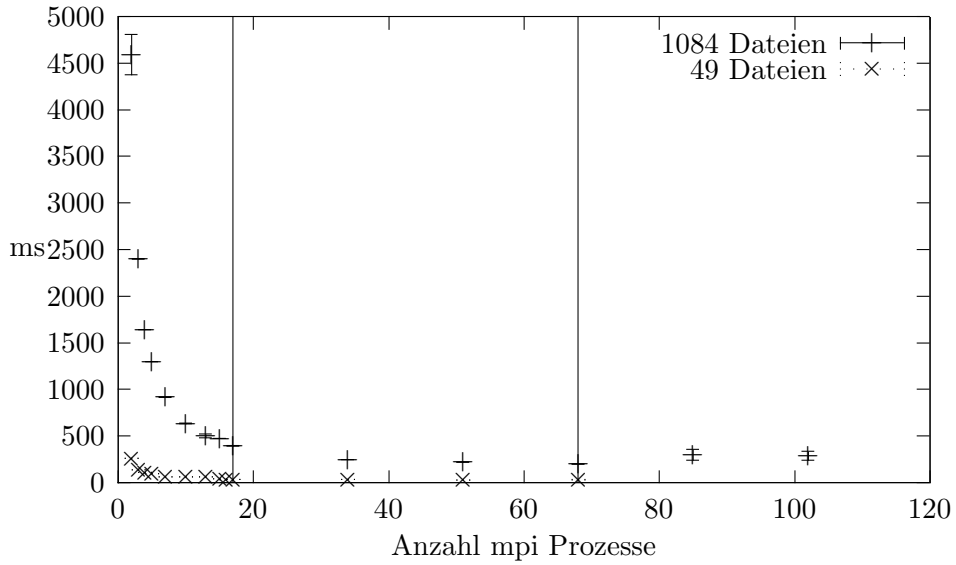

Abbildung 4: Query Laufzeiten 1. und 2. Messreihe im Vergleich

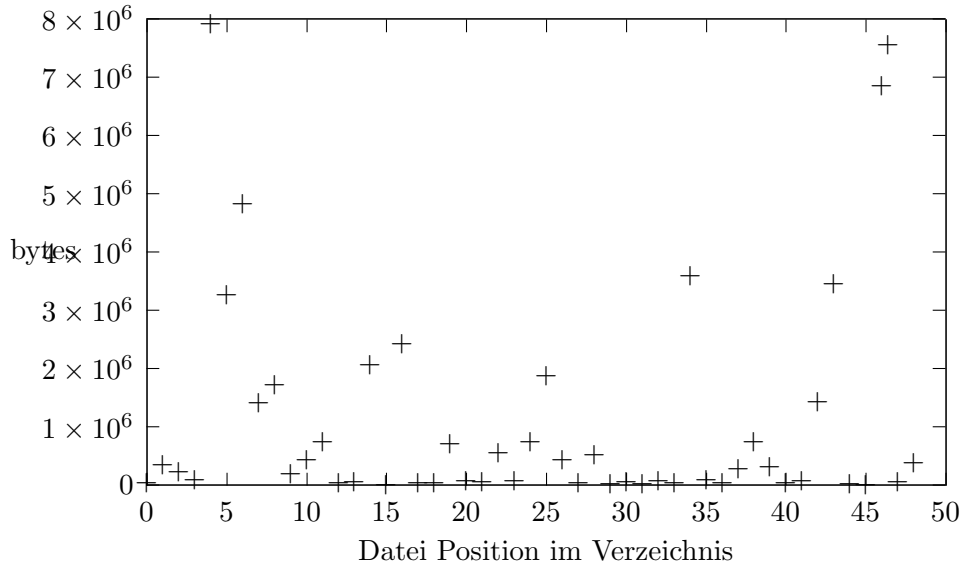

Abbildung 5: Größe der Dateien der 2. Messreihe