MapReduce

Vereinfachte Datenverarbeitung in großen Rechnerverbünden

Semesterarbeit

für den Kurs Wissenschaftliches Arbeiten im WS2012/13

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN

University of Applied Sciences

vorgelegt von:	Igor Marijanovic
Studienbereich:	Medieninformatik
Matrikelnummer:	794894
Dozent:	Prof. Dr. Henrik Tramberend

Zusammenfassung

MapReduce ist ein Programmiermodell und damit verbundene Implementierung, um große Datenbestände zu erzeugen und zu verarbeiten. Nutzer spezifizieren eine *Map*-Funktion, die Schlüssel/Wert-Paare verarbeitet, um Zwischendaten in Form von Schlüssel/Wert-Paaren zu erzeugen, und eine *Reduce*-Funktion, die alle Werte in den Zwischendaten zusammenfasst, die mit einem bestimmten Schlüssel assoziiert sind. Programme, die in diesem funktionalen Stil geschrieben sind, werden automatisch parallelisiert und auf großen Rechnerverbünden mit handelsüblicher Hardware ausgeführt. Mit MapReduce können Programmierer ohne Kenntnisse paralleler und verteilter Systeme leicht die Ressourcen großer verteilter Systeme nutzen.

Eine Forschungsgruppe bei Google hat 2004 eine MapReduce-Implementierung vorgestellt, die in großen Rechnerverbünden ausgeführt wird und hochskalierbar ist. Zu diesem Zeitpunkt wurden in Google's Rechnerverbünden bereits über tausend MapReduce-Jobs täglich ausgeführt.

Diese Arbeit basiert auf dem im Jahr 2004 veröffentlichten Artikel „MapReduce: Simplified Data Processing on Large Clusters“ von Jeffrey Dean und Sanjay Ghemawat.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Programs written in this functional style are automatically parallelized and executed on large clusters of commodity machines. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. With MapReduce programmers without any experience with parallel and distributed systems can easily utilize the resources of a large distributed system.

A group of scientists from Google presented a MapReduce implementation that is executed on large clusters and is highly-scalable in 2004. At this time over one thousand MapReduce jobs were already executed in Google's clusters daily.

This work is based on the article „MapReduce: Simplified Data Processing on Large Clusters“ by Jeffrey Dean and Sanjay Ghemawat published in 2004.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ASF	Apache Software Foundation
BAD-FS	Batch-Aware Distributed File System
GDB	The GNU Project Debugger
GFS	Google File System
HDFS	Hadoop Distributed File System
HTTP	Hypertext Transport Protocol
IDE	Integrated Drive Electronics
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
PC	Personal Computer
RPC	Remote Procedure Call
TACC	Texas Advanced Computing Center
UDP	User Datagram Protocol
URL	Uniform Resource Locator
WAN	Wide Area Network

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1. Einleitung

1.1. Motivation

Schon Jahre vor der Veröffentlichung der betrachteten Arbeit [DG04] sah man sich bei Google vor die Herausforderung gestellt, möglichst viele Rechner in einem Verbund für die Berechnung der Suchindizes effektiv zu verwenden. Ein eigenes verteiltes Dateisystem [GGL03] war bereits entwickelt worden, um die sehr großen Index-Dateien sicher und zuverlässig speichern und verwalten zu können. Da erschien es sinnvoll, auch die Berechnung dieser Dateien zu verteilen.

Hier stellt sich das Problem, dass die Verteilung einer solchen Berechnung eine sehr komplexe Aufgabe darstellt. Es muss entschieden werden, wie parallelisiert, Daten verteilt und bei Fehlern reagiert wird. Die Erfüllung solcher Anforderungen erfordert die Implementierung von komplexem Programmcode.

Diese Anforderungen in eine Bibliothek zu kapseln, war das Ziel bei Google, um es Nutzern zu erlauben, einfachen Programmcode zu schreiben, der in einem Rechnerverbund ausgeführt werden kann, ohne sich weiter um die Probleme von Parallelisierung, Distribution der Daten und Fehlertoleranz kümmern zu müssen.

1.2. Ziel der Arbeit

In dieser Arbeit soll die Implementierung einer MapReduce-Bibliothek bei Google betrachtet werden. Die Ergebnisse bezüglich der Performanz, Zuverlässigkeit und Versatilität des Einsatzes sollen im zeitlichen Kontext der Erstveröffentlichung 2004 bewertet werden.

1.3. Aufbau der Arbeit

Im folgenden Kapitel 2 werden die Grundlagen von MapReduce besprochen. Das daran anschließende Kapitel 3 stellt die MapReduce-Bibliothek von Google vor und Kapitel 4 geht dann näher auf die implementierten Verbesserungen ein. Kapitel 5 präsentiert Ergebnisse von Leistungstests und bewertet sie im zeitlichen Kontext. Das Kapitel 6 stellt die Erfahrungen vor, die bei Google im Zusammenhang mit der Verwendung der MapReduce-Bibliothek gemacht worden sind. Im abschließenden Kapitel 7 werden verwandte Arbeiten betrachtet und ein Fazit gezogen.

2. Grundlagen

Im Folgenden wird auf die Grundlagen des MapReduce-Programmiermodells eingegangen und diese anhand eines Beispiels verdeutlicht.

2.1. Programmiermodell

Die Berechnung nimmt einen Satz von Schlüssel/Wert-Paaren und liefert einen Satz solcher Paare zurück. Der Nutzer der MapReduce-Bibliothek drückt die Berechnung in zwei Funktionen aus: *Map* und *Reduce*.

Die durch den Nutzer implementierte *Map*-Funktion erhält eine Eingabe in Form eines Schlüssel/Wert-Paares und liefert einen Satz von *Intermediate*[1]-Schlüssel/Wert-Paaren. Die MapReduce-Bibliothek gruppiert alle *Intermediate*-Werte, die mit dem *Intermediate*-Schlüssel I assoziiert sind, und leitet diese an die *Reduce*-Funktion weiter.

Die *Reduce*-Funktion, die auch vom Nutzer geschrieben wurde, akzeptiert einen *Intermediate*-Schlüssel I und einen Satz von Werten für diesen Schlüssel. Sie führt diese Werte zusammen, um möglicherweise einen kleineren Wertesatz zu erzeugen. Typischerweise werden bei Aufruf der *Reduce*-Funktion null oder ein Ausgabewert erzeugt. Die *Intermediate*-Werte werden der *Reduce*-Funktion des Nutzers durch einen Iterator zur Verfügung gestellt. Dies ermöglicht es, mit Wertelisten umzugehen, die zu groß sind, um im Speicher abgelegt zu werden.

2.1.1. Beispiel

Gegeben sei das Problem, die Wortfrequenzen aus einer großen Sammlung von Dokumenten zu ermitteln. Der vom Nutzer zu implementierende Programmcode würde dem folgenden Pseudocode ähneln:

Listing 2.1: Pseudocode: Wortfrequenzen ermitteln mit MapReduce

```
1  map(String key, String value):
2  // key: documentname
3  // value: documentcontents
4    for each word w in value:
5      EmitIntermediate(w, "1");
6
7  reduce(Stringkey, Iterator values):
8    // key: a word
9    // values: a list of counts
10   int result = 0;
11   for each v in values:
12     result += ParseInt(v);
13   Emit(AsString(result));
```

Die Methode `map` gibt jedes enthaltene Wort im Dokument aus, gefolgt von einer Zahl, die die Anzahl der Vorkommnisse abbildet (nur 1 in diesem Beispiel). Die Methode `reduce` summiert alle Zählwerte, die für ein bestimmtes Wort zurückgegeben werden.

Ein „`mapreduce specification`“-Objekt wird durch den Nutzer mit Werten für Ein- und Ausgabedatei und optionalen Parametern zur Optimierung des Algorithmus befüllt. Der Nutzer ruft

[1] engl. Adj.: vermittelnd, schlichtend, zwischen zwei Phasen existierend

daraufhin die *MapReduce*-Funktion mit dem „`mapreduce specification`“-Objekt als Übergabewert auf. Der Programmcode des Nutzers wird an die MapReduce-Bibliothek gebunden[2].

2.2. Typen

Obwohl der vorherige Pseudocode Ein- und Ausgabewerte vom Typ *String* verwendet, haben *Map* und *Reduce* assoziierte Typen:

Map	`(k1,v1)`	→ `list(k2,v2)`
Reduce	`(k2,list(v2))`	→ `list(v2)`

Die Schlüssel/Wert-Paare der Eingabe stammen aus einer anderen Domäne als die der Ausgabe. Die *Intermediate*-Schlüssel/Wert-Paare stammen aber aus der gleichen Domäne wie die Ausgabepaare.

2.3. Weitere Beispiele

Ein paar Beispiele für interessante Programme, die leicht mit *MapReduce*-Berechnungen ausgedrückt werden können:

- **Verteiltes Grep:** Die *Map*-Funktion liefert eine Zeile, wenn sie dem regulären Ausdruck des Nutzers entspricht. Die *Reduce*-Funktion ist eine Identitätsfunktion, die nur die zur Verfügung stehenden Zwischendaten in die Ausgabe kopiert.
- **Zugriffsfrequenz einer URL[3]:** Die *Map*-Funktion verarbeitet Logs der Webseitenanfragen und der Ausgaben `<URL,1>`. Die *Reduce*-Funktion summiert alle Werte für die gleiche Webseite und gibt ein Paar `<URL, total count>` aus.
- **Reverse Web-Link Graph:** Die *Map*-Funktion gibt ein Paar `<target, source>` für jeden Link zu einem `target` aus, der auf einer Seite `source` gefunden wurde. Die *Reduce*-Funktion konkateniert die Liste aller Quell-*URLs* mit einer vorgegebenen Ziel-*URL*, und liefert ein Paar `<target, `*list*`(source)>`.
- **Termvektor pro Host:** Ein *Termvektor*[4] soll als eine Liste von Paaren `<word, frequency>` erzeugt werden. Die *Map*-Funktion liefert ein Paar `<hostname, term-vector>` für jedes eingegebene Dokument. Der Hostname wird aus der URL des Dokuments entnommen. Die *Reduce*-Funktion erhält alle Termvektoren für einen bestimmten Host. Sie fasst diese Termvektoren zusammen, verwirft Begriffe, die nicht häufig sind, und liefert letztendlich ein Paar `<hostname, term vector>`.
- **Invertierter Index:** Die *Map*-Funktion parst jedes Dokument und liefert eine Sequenz von Paaren `<word, document ID>`. Die *Reduce*-Funktion akzeptiert alle Paare für ein bestimmtes Wort, sortiert korrespondierende *document IDs* und gibt ein Paar `<word, list(document ID)>` aus. Die Zusammenstellung aller Ausgabepaare bildet einen einfachen invertierten Index. Es ist leicht diese Berechnung zur Überwachung von Wortpositionen zu erweitern.
- **Verteiltes Sort:** Die *Map*-Funktion extrahiert den Schlüssel und liefert ein Paar `<key, record>`. Die *Reduce*-Funktion gibt alle Paare unverändert aus. Diese Berechnung hängt von der Partitionierungsfunktion und der Sortierung ab, die in Kapitel 4.1 und in Kapitel 4.2 beschrieben werden.

[2] implementiert in C++; vollständiger Quellcode des Beispiels in Appendix A.1
[3] Abk.: Uniform Resource Locator
[4] Def.: Liste aller Worte, die in einer Webseite enthalten sind, für die Ermittlung von thematischer Breite

3. Implementierung

Es sind viele verschiedene Implementierungen einer MapReduce-Schnittstelle möglich. Sie hängt von der Umgebung, wie beispielsweise kleine Rechner mit gemeinsam genutzten Speicher oder NUMA-Großrechnern[5], ab und muss darauf abgestimmt sein. In diesem Abschnitt wird eine Implementierung beschrieben, die für eine Berechnungsumgebung bestimmt ist, die bei Google weitreichende Verwendung findet: große Rechnerverbünde mit handelsüblichen PCs[6] in einem geswitchten Netzwerk[BDH03]. Eigenschaften der Umgebung:

1. Dual-Prozessor-Systeme mit 2-4 Gigabyte Arbeitsspeicher und Linux Betriebssystem.
2. Handelsübliche Netzwerk-Hardware mit 100 Megabit oder 1 Gigabit pro Sekunde.
3. Ein Verbund besteht aus hunderten oder tausenden von Rechnern, weshalb Maschinenausfälle häufig auftreten.
4. Speicherkapazität wird über günstige IDE-Festplatten[7] bereitgestellt. Ein verteiltes Dateisystem [GGL03], das von Google entwickelt wurde, wird genutzt, um die Daten zu verwalten. Das Dateisystem nutzt Replikation, um Verfügbarkeit und Zuverlässigkeit auf nicht zuverlässiger Hardware zu gewährleisten.
5. Nutzer übermitteln Jobs an einen *Scheduler*[8]. Jeder Job besteht aus einer Reihe von Tasks und wird von dem *Scheduler* einer Gruppe verfügbarer Rechner im Verbund zugewiesen.

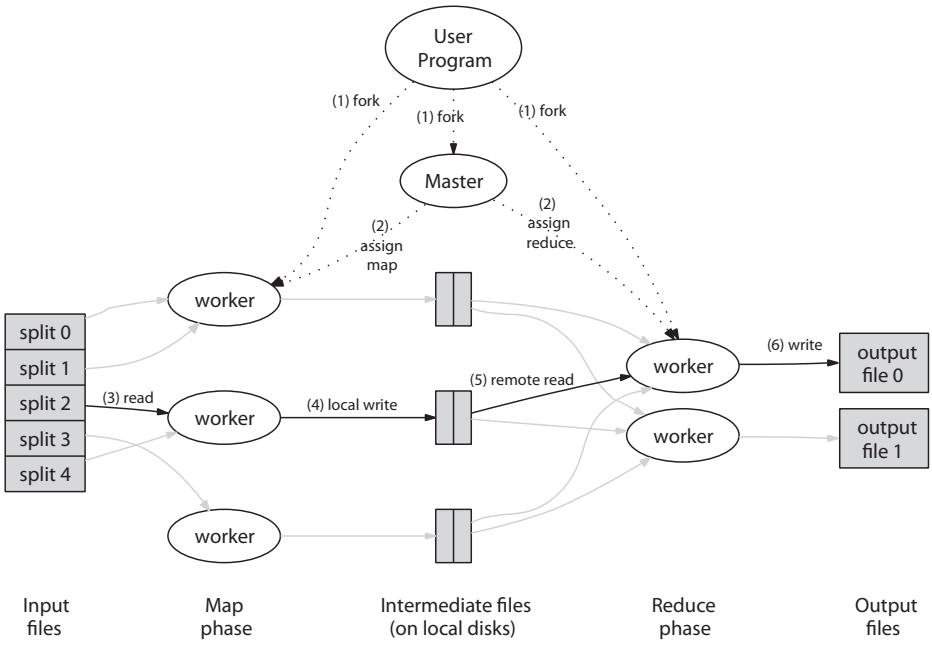

Abbildung 3.1.: Übersicht über die Ausführung

[5] Abk.: Non-Uniform Memory Access; Def.: Standard für optimierten Speicherzugriff in Großrechnern
[6] Abk.: Personal Computer
[7] Abk.: Integrated Drive Electronics
[8] engl. Subst.: (Zeit)Planer

3.1. Ausführungsübersicht

Die *Map*-Aufrufe sind durch automatische Partitionierung der Eingabedaten in M Teile über viele Rechner verteilt. Die Teile können parallel von verschiedenen Rechnern verarbeitet werden. Aufrufe von *Reduce* werden verteilt durch die Partitionierung der Zwischendaten in R Teile über eine Partitionierungsfunktion (z. B. , *hash(key) mod R*). Die Anzahl R der Teile und die Partitionierungsfunktion werden spezifiziert durch den Nutzer. Abbildung 3.1 zeigt den Gesamtfluss einer MapReduce-Operation in der vorgestellten Implementierung. Wenn das Programm des Nutzers die Funktion `MapReduce` aufruft, wird die folgende Sequenz von Aktionen ausgelöst (die Nummern in der folgenden Auflistung korrespondieren mit den Nummern in Abbildung 3.1):

1. Die MapReduce-Bibliothek im Programm des Nutzers zerteilt zuerst die Eingabedateien in M Teile mit einer Größe von 16 bis 64 Megabyte (vom Nutzer einstellbar). Sie startet daraufhin viele Kopien des Programms auf einem Rechnerverbund.

2. Es gibt eine spezielle Kopie des Programms: den Master. Alle anderen sind Worker[9] denen Aufgaben (M Map-Tasks und R Reduce-Tasks) vom Master zugewiesen werden müssen. Der Master vergibt jeweils einen Task an Rechner, die nicht ausgelastet sind.

3. Ein Worker, dem ein Map-Task zugewiesen wurde, liest den Inhalt des korrespondierenden Teils der Eingabedaten ein. Er parst Schlüssel/Wert-Paare aus den Eingabedaten und gibt jedes Paar an die vom Nutzer spezifizierte *Map*-Funktion weiter. Das *Intermediate*-Schlüssel/Wert-Paar wird im Speicher gepuffert.

4. Die gepufferten Schlüssel/Wert-Paare werden, durch die Partitionierungsfunktion aufgeteilt in R Regionen, regelmäßig auf die lokale Festplatte geschrieben. Die Speicherorte dieser gepufferten Paare auf den lokalen Festplatten werden an den Master übermittelt, der dafür verantwortlich ist, diese Daten an die Reduce-Worker weiterzuleiten.

5. Wenn ein Reduce-Worker vom Master über die Speicherorte informiert wird, nutzt es RPCs[10], um die gepufferten Daten von den lokalen Festplatten der Map-Worker zu lesen. Wenn ein Reduce-Worker die ganzen Zwischendaten eingelesen hat, sortiert er sie nach den *Intermediate*-Schlüsseln und gruppiert übereinstimmende Schlüssel. Eine Sortierung wird benötigt, da viele verschiedene Schlüssel zu dem gleichen Reduce-Task gehören. Wenn die Menge der Zwischendaten zu groß für den Arbeitsspeicher ist, wird eine externe Sortierfunktion genutzt.

6. Der Reduce-Worker iteriert über die sortierten Zwischendaten und gibt für jedes einzigartige Schlüssel/Wert-Paar, das er findet, den Schlüssel und die korrespondierenden *Intermediate*-Werte an die *Reduce*-Funktion des Nutzers weiter. Die Ausgabe der *Reduce*-Funktion wird an die finale Ausgabe für diese Partition angehängt.

7. Sind alle Map- und Reduce-Tasks abgeschlossen, weckt der Master das Programm des Nutzers. An dieser Stelle kehrt der Aufruf von `MapReduce` wieder zum Programmcode des Nutzers zurück.

Nach erfolgreicher Ausführung ist die Ausgabe in R Ausgabedateien verfügbar (eine je Reduce-Task mit Dateinamen wie vom Nutzer spezifiziert). Typischerweise müssen Nutzer diese R Ausgabedateien nicht in einer Datei vereinen, denn häufig werden sie als Eingabe für weitere MapReduce-Aufrufe oder von einer anderen verteilten Anwendung, die mit solchen Eingabedaten umgehen kann, genutzt.

[9] engl. Subst.: Arbeiter (hier Arbeitsrechner bzw. Arbeitsprozesse)
[10] Abk.: Remote Procedure Call

3.2. Master-Datenstruktur

Der Master hält mehrere Datenstrukturen. Er speichert für jeden Map- und Reduce-Task den Zustand (*idle*[11], *in-progress*[12]oder *completed*[13]) und die Identität des *Workers* (für Tasks, die nicht im Zustand *Idle* sind).

Der Master fungiert als Kanal über den die Speicherorte der Zwischendaten von Map- zu Reduce-Tasks übermittelt werden. Deshalb speichert der Master für jeden beendeten Map-Task die Speicherorte und Dateigrößen der R Dateien in denen die Zwischendaten gespeichert wurden. Updates bzgl. dieser Informationen werden empfangen, sobald Map-Tasks abgeschlossen werden. Die Information wird inkrementell an die *Worker* übertragen, die Reduce-Tasks im Zustand *in-progress* haben.

3.3. Fehlertoleranz

Da die MapReduce-Bibliothek dafür entworfen wurde, dabei zu helfen, sehr große Mengen von Daten unter Zuhilfenahme von hunderten oder tausenden Rechnern zu verarbeiten, muss sie tolerant gegenüber Rechnerausfällen sein.

3.3.1. Ausfall eines Workers

Der Master pingt die Worker regelmäßig an. Erhält er keine Antwort von einem Worker nach einer bestimmten Zeit, wird dieser vom Master als *failed*[14] markiert. Alle Map-Tasks, die von diesem Worker bearbeitet wurden, werden wieder auf den Zustand *idle* zurückgesetzt. Sie sind somit wieder frei, um vom *Scheduler* einem anderen Worker zugewiesen zu werden.

Abgeschlossene Map-Tasks werden neu ausgeführt, da die Ausgabe des Workers auf seiner lokalen Festplatte gespeichert war, auf die nach dem Ausfall nicht mehr zugegriffen werden kann. Abgeschlossene Reduce-Tasks müssen nicht erneut ausgeführt werden, weil ihre Ausgabe in einem globalen Dateisystem gespeichert wird.

Wenn ein Map-Task zuerst von Worker A und dann später von Worker B ausgeführt wird, weil A ausgefallen ist, werden alle Worker, die Reduce-Tasks ausführen, über die erneute Ausführung informiert. Jeder Reduce-Task, der noch nicht die Daten von Worker A gelesen hat, wird sie von Worker B einlesen.

MapReduce ist widerstandsfähig gegenüber Maschinenausfällen im großen Maßstab. Beispielsweise wurde bei Google berichtet, dass einmal während der Ausführung einer MapReduce-Operation eine Nichterreichbarkeit einer Gruppe von 80 Rechnern über mehrere Minuten auftrat. Ursache war die Wartung des Netzwerks des Rechnerverbunds, auf dem die Operation ausgeführt wurde. Der MapReduce-Master hat einfach die erneute Ausführung der Map-Tasks der nicht erreichbaren Maschinen eingeleitet und letztendlich die Operation abgeschlossen.

[11] engl. Adj.: brach liegend, nicht produktiv, frei
[12] engl. Adj.: in Bearbeitung, im Werden
[13] engl. Adj.: abgeschlossen
[14] engl. Adj.: ausgefallen

3.3.2. Ausfall eines Masters

Es ist leicht, den Master regelmäßig Wiederherstellungspunkte der oben beschriebenen Master-Datenstruktur erstellen zu lassen. Wenn der Master-Task ausfällt kann eine neue Kopie, vom letzten Wiederherstellungspunkt ausgehend, erstellt werden. Da es den Master nur einmal gibt, ist sein Ausfall nicht sehr wahrscheinlich. Sollte es dennoch passieren, bricht die vorgestellte Implementierung eine Berechnung ab. Fehlerquelle und Zustand des Masters können geprüft und abgebrochene MapReduce-Operationen ggf. fortgesetzt werden.

3.3.3. Fehlersemantik

Die von Nutzer zur Verfügung gestellten Map- und Reduce-Operatoren sind deterministische Funktionen ihrer Eingabewerte. Die verteilte Implementierung produziert die gleiche Ausgabe wie bei einer fehlerlosen sequentiellen Ausführung des gesamten Programms.

Die MapReduce-Implementierung ist angewiesen auf atomare *Commits*[15] von Map- und Reduce-Tasks, um diese Eigenschaft zu erreichen. Jeder in Bearbeitung befindliche Task schreibt seine Ausgabe in private, temporäre Dateien. Ein Reduce-Task produziert genau eine solche Datei während Map-Tasks R Dateien erzeugen (eine für jeden Reduce-Task). Wenn ein Map-Task abgeschlossen wurde, sendet der Worker eine Nachricht an den Master und schließt Informationen über die Namen der erzeugten Dateien ein. Erhält der Master eine solche Nachricht für einen Map-Task, der bereits abgeschlossen wurde, dann ignoriert er sie. Anderenfalls werden die Einträge der Namen der R Dateien in der Master-Datenstruktur abgelegt.

Wird ein Reduce-Task abgeschlossen, benennt der Worker die temporäre Ausgabedatei in die finale Ausgabedatei um. Wird derselbe Reduce-Task auf verschiedenen Rechnern ausgeführt, wird mehrfach zur Umbenennung in die finale Ausgabedatei aufgerufen. Die atomaren Umbenennungsoperationen des Dateisystems garantieren, dass nur die Ausgabe eines Reduce-Tasks gespeichert wird.

Die überwiegende Mehrheit der verwendeten Map- und Reduce-Operatoren sind deterministisch. Der Umstand, dass die Semantik sehr ähnlich einer sequentiellen Ausführung ist, macht es Programmierern leicht, das Verhalten ihrer Programme zu erfassen.

3.4. Lokalität

Netzwerkbandbreite ist eine wertvolle Ressource in den Rechenzentren von Google, die massiv eingespart werden kann, wenn die Eingabedaten auf den lokalen Festplatten der Rechner des Verbundes gespeichert werden (verwaltet von GFS [GGL03]). GFS teilt jede Datei in 64 Megabyte große Blöcke und speichert diverse Kopien der Blöcke auf unterschiedlichen Rechnern (normalerweise 3 Kopien pro Block). Der MapReduce-Master beachtet das und versucht dementsprechend Map-Tasks für solche Rechner zu planen, auf denen bereits die benötigten Eingabedaten gespeichert sind. Schlägt dieser Versuch fehl, wird versucht den Map-Task einem Rechner zuzuweisen, der sich in der Nähe eines Rechners mit den gespeicherten Daten befindet (z. B. ein Rechner, der am gleichen Netzwerkswitch ist wie der Rechner, der die Daten vorhält). Dies hat zur Folge, dass die Eingabedaten selbst bei sehr großen MapReduce-Operationen, die über einen Großteil des Rechnerverbundes ausgeführt werden, größtenteils lokal eingelesen werden und damit die Netzwerklast nicht erhöhen.

[15] Fachbegriff der Informatik; Def.: erfolgreicher Abschluss einer Transaktion in einem EDV-System bzw. einer Datenbank

3.5. Granularität der Aufgaben

Wie vorher beschrieben unterteilt die Implementierung die Map-Phase in M und die Reduce-Phase in R Teile. Idealerweise sollten sowohl M als auch R wesentlich größer sein als die Anzahl der Rechner, die als Worker zum Einsatz kommen. Durch die Abarbeitung vieler verschiedener Tasks unterstützen die Worker die dynamische Lastverteilung und beschleunigen die Wiederherstellung bei Ausfall eines Workers, da die vielen abgeschlossenen Map-Tasks über alle anderen Worker verteilt werden können.

Obwohl nur etwa 1 Byte pro Map-Task/Reduce-Task-Paar gespeichert werden muss, ist die Größe von M und R praktisch begrenzt, da der Master $O(M + R)$ Zuweisungsentscheidungen machen und $O(M * R)$ Statusinformationen speichern muss.

Außerdem wird R oft durch Nutzer beschränkt, da die Ausgabe jedes Reduce-Tasks in einer separaten Datei gespeichert wird. Bei Google wurde M so gewählt, dass die Eingabedaten für jeden Task zwischen 16 und 64 Megabyte groß waren. Dadurch wird erreicht, dass die oben beschriebene Lokalitätsoptimierung effektiv greift. R entspricht mit seiner Größe einem kleinen Vielfachen der Anzahl von Rechnern, die für die Aufgabe eingesetzt werden sollen (z. B. $M = 200.000$ und $R = 5000$ bei 2000 Workern).

3.6. Backup-Tasks

Ein üblicher Grund für eine Verlängerung der Gesamtausführungszeit einer MapReduce-Operation sind Nachzügler. Das sind Rechner, die besonders lange für den Abschluss einer der wenigen letzten Tasks in der Berechnung benötigen. Nachzügler können aus einer Reihe von Gründen resultieren. Beispielsweise könnte eine fehlerhafte Festplatte durch Fehlerkorrekturen für schwankende Transferraten sorgen oder ein Rechner könnte durch andere Aufgaben, die er vom *Scheduler* des Rechnerverbundes zugewiesen bekommen hat, ausgelastet sein.

Es gibt einen allgemeinen Mechanismus in der MapReduce-Bibliothek, um Nachzügler vorzubeugen. Steht eine MapReduce-Operation kurz vor der Vollendung, weist der Master die übrigen Tasks, die den Status *in-progress* haben, anderen Workern zur Backup-Ausführung zu. Der Task wird als abgeschlossen markiert sobald entweder die primäre oder die Backup-Ausführung fertig ist. Dieser Mechanismus ist soweit optimiert worden, dass nur wenige Prozent mehr Ressourcen für die Berechnung benötigt werden. Dafür wird die Berechnungszeit großer MapReduce-Operationen signifikant reduziert, wie beispielsweise aus den Ergebnissen in Kapitel 5.3 hervorgeht.

4. Verbesserungen

Obwohl die Grundfunktionalität, umgesetzt durch das einfache Schreiben der Map- und Reduce-Funktionen, für die meisten Bedürfnisse ausreicht, wurde bei Google festgestellt, dass es einige sinnvolle Erweiterung gibt. Diese werden in diesem Abschnitt beschrieben.

4.1. Partitionierungsfunktion

Benutzer von MapReduce legen die erwünschte Zahl von Ausgabedateien für Reduce-Tasks fest (`R`). Die Daten werden auf diese Tasks verteilt, indem eine Partitionierungsfunktion auf die *Intermediate*-Schlüssel angewendet wird. Es steht eine Standard-Partitionierungsfunktion zur Verfügung, die Hashing verwendet (z. B. „hash(key) mod R“). Dies führt zu ziemlich gut ausbalancierten Partitionen. In einigen Fällen ist es jedoch sinnvoll, die Daten nach einer anderen Funktion des Schlüssels zu partitionieren. Zum Beispiel sind die Ausgabeschlüssel manchmal URLs und man möchte, dass alle Einträge zu einem Host in derselben Ausgabe gespeichert werden. Deshalb können Anwender der MapReduce-Bibliothek eine spezielle Partitionierungsfunktion erstellen. Wird beispielsweise `hash(Hostname(urlkey)) mod R` als Partitionierungsfunktion verwendet, sorgt es dafür, dass alle URLs vom selben Host in derselben Ausgabedatei gespeichert werden.

4.2. Garantie der Reihenfolge

Es wird garantiert, dass innerhalb einer Partition die *Intermediate*-Schlüssel/Wert-Paare in aufsteigender Reihenfolge des Schlüssels verarbeitet werden. Die garantierte Reihenfolge macht es einfach, sortierte Ausgabedateien pro Partition zu erzeugen, was nützlich ist, wenn das Ausgabedatenformat einen effizienten zufälligen Zugriff auf Werte anhand des Schlüssels unterstützen muss. Außerdem ist der Umgang mit einer sortierten Ausgabedatei angenehmer für Nutzer.

4.3. Kombinierfunktion

In manchen Fällen gibt es signifikante Wiederholungen in den *Intermediate*-Schlüsseln, die von jedem Map-Task erzeugt werden. Ein gutes Beispiel hierfür ist das Wortfrequenz Beispiel in Appendix A.1. Da Wortfrequenzen meist einer Zipf-Verteilung[16] folgen, erzeugt jeder Map-Task hunderte oder tausende von Datensätzen in der Form `<the, 1>`. Die Ergebnisse werden über das Netzwerk an einen einzelnen Reduce-Task gesendet und dann von der *Reduce*-Funktion addiert, um eine Nummer zu erzeugen. Nutzer können eine optionale *Combiner*[17]-Funktion spezifizieren, die eine Teilzusammenfassung der Daten vornimmt bevor sie über das Netzwerk versandt werden.

Die *Combiner*-Funktion wird auf jedem Rechner ausgeführt, der Map-Tasks übernimmt. Typischerweise wird derselbe Code verwendet, um die Combiner- und die Reduce-Funktion zu implementieren. Der einzige Unterschied zwischen einer Reduce- und einer Combiner-Funktion ist, wie die Ausgabe der Funktion von der MapReduce-Bibliothek verarbeitet wird. Die Ausgabe einer Reduce-Funktion wird in die endgültige Ausgabedatei geschrieben. Die Ausgabe einer Combiner-Funktion wird in einer Zwischendatei gespeichert, die an den Reduce-Task geschickt wird.

[16] nach George Kingsley Zipf, amerikanischer Mathematiker (1902-1950)
[17] engl. Subst.: Kombinierer

Das partielle Kombinieren erhöht die Geschwindigkeit bestimmter Klassen von MapReduce-Operationen. Appendix A.1 enthält ein Beispiel, das einen *Combiner* verwendet.

4.4. Ein- und Ausgabetypen

Die MapReduce-Bibliothek unterstützt das Einlesen von Daten in vielen unterschiedlichen Formaten. Beispielsweise behandelt der Eingabetyp „text“ jede Zeile als Schlüssel/Wert-Paar: Der Schlüssel ist der Offset in der Datei und der Wert ist der Inhalt der Zeile. Ein weiteres häufig verwendetes Format speichert eine Sequenz von Schlüssel/Wert-Paaren sortiert nach dem Schlüssel. Im jeder Eingabetypimplementierung wird bestimmt, wie Daten in sinnvolle Bereiche unterteilt werden, um in verschiedenen Map-Tasks verarbeitet zu werden (z. B. die Bereichseinteilung des Modus „text“ sorgt dafür, dass Aufteilungen nur an Zeilengrenzen erfolgen).

4.5. Seiteneffekte

In einigen Fällen haben es Benutzer von MapReduce als sinnvoll erachtet, zusätzliche Hilfsdateien als Ausgabe ihrer Map- oder Reduce-Operatoren zu erzeugen. Es obliegt dem Entwickler der Anwendung, solche Seiteneffekte atomar und idempotent zu machen. Typischerweise schreibt die Anwendung in eine temporäre Datei und benennt sie nach vollständiger Generierung atomar um.

Es wird keine Unterstützung geboten für atomare Zwei-Phasen-*Commits* von multiplen Ausgabedateien, die von einen einzelnen Task erzeugt werden. Deshalb sollten Tasks, die mehrere Ausgabedateien mit dateiübergreifenden Konsistenzanforderung erzeugen, deterministisch sein. Diese Einschränkung hat sich in der Praxis nie als Problem erwiesen.

4.6. Überspringen fehlerhafter Einträge

Manchmal gibt es Bugs im Programmcode des Nutzers, der bei bestimmten Datensätzen die Map- oder Reduce-Funktion deterministisch abstürzen lässt. Solche Bugs verhindern, dass eine MapReduce-Operation fertiggestellt wird. Die übliche Vorgehensweise ist, den Fehler zu beheben. In manchen Fällen ist dies jedoch nicht umsetzbar, etwa wenn der Bug sich in einer Bibliothek eines Drittanbieters befindet, für den kein Quelltext zur Verfügung steht. Außerdem ist es manchmal akzeptabel, einzelne Datensätze zu ignorieren, beispielsweise dann, wenn eine statistische Analyse auf einem großen Datenbestand durchgeführt wird. Es wird ein optionaler Ausführungsmodus zur Verfügung gestellt, bei dem die MapReduce-Bibliothek Datensätze ermittelt, die deterministisch Abstürze verursachen und diese dann überspringt, um die Verarbeitung fortsetzen zu können.

Jeder Worker installiert einen *Signalhandler* der Segmentierungsverstöße (*Segmentation Violations*) und Busfehler verarbeitet. Bevor eine Map- oder Reduce-Operation des Nutzers aufgerufen wird, speichert die MapReduce-Bibliothek die Sequenznummer des Arguments in einer globalen Variablen. Wenn der Programmcode des Nutzers ein Signal erzeugt, schickt der Signalhandler ein „last gasp“[18]-UDP[19]-Paket an den MapReduce-Master, dass die Sequenznummer enthält. Wenn der Master mehr als einen Fehlversuch bei einem Datensatz erkennt, wird dieser markiert, damit er übersprungen werden kann, wenn er für die nächste neue Ausführung des jeweiligen Map- oder Reduce-Task ausgegeben wird.

[18] engl. Redewendung: letzter Atemzug
[19] Abk.: User Datagram Protocol

4.7. Lokale Ausführung

Debugging Probleme in Map- oder Reduce-Funktionen können sehr trickreich sein, da die eigentliche Berechnung in einem verteilten System häufig auf vielen tausend Maschinen ausgeführt wird und Entscheidungen über die Arbeitszuweisung dynamisch vom Master gefällt werden. Um zu helfen das Debugging, Profiling und Testen im kleinen Maßstab zu vereinfachen, wurde eine alternative Implementierung der MapReduce-Bibliothek entwickelt, die sequenziell eine MapReduce-Operation auf einem Rechner lokal durchführt. Es werden Kontrollmöglichkeiten zur Verfügung gestellt, so dass der Anwender die Berechnung auf einen einzelnen Map-Task beschränken kann. Benutzer rufen ihr Programm mit einem speziellen *Flag* auf und können so ihre Debugging- oder Testing-Werkzeuge einbinden (z.B. *gdb*[20]).

4.8. Statusinformationen

Der Master betreibt einen internen HTTP-Server[21] und exportiert eine Reihe von für den menschlichen Gebrauch bestimmten Statusseiten. Die Statusseite zeigt den Fortschritt der Berechnung. Dargestellt werden unter anderem folgende Informationen:

- die Anzahl abgeschlossener und gerade ausgeführter Tasks
- die Anzahl der Bytes in der Eingabe, den Zwischendaten und der Ausgabe
- Verarbeitungsraten

Die Seite enthält auch Links zu den Standardfehler- und Standardausgabedateien, die von jedem Task erzeugt werden. Diese Daten können verwendet werden, um vorherzusagen, wie lange die Berechnung dauern wird und ob mehr Ressourcen für die Berechnung benötigt werden. Diese Seiten können auch verwendet werden, um herauszufinden, wann die Berechnung viel langsamer ist als erwartet wurde.

Zusätzlich zeigt die Toplevel-Statusseite, welche Arbeitsprozesse mit welchen Map- und Reduce-Tasks fehlschlugen. Diese Informationen erweisen sich als nützlich bei der Diagnose von Bugs im Programmcode des Nutzers.

4.9. Zähler

Die MapReduce-Bibliothek stellt eine Zählerfunktionalität zur Verfügung, die es ermöglicht, verschiedene Events zu zählen. Durch den Programmcode könnte beispielsweise gefordert werden, die Anzahl der verarbeiteten Worte oder die Anzahl deutscher Dokumente, die indiziert wurden, zu ermitteln.

Um diese Funktionalität zu nutzen wird im Programmcode ein Zähler-Objekt angelegt, das in der Map- und/oder Reduce-Funktion entsprechend erhöht wird, wie im folgenden Codebeispiel:

Listing 4.1: Pseudocode: Zähler in Map-Funktion

```
1 Counter* uppercase;
2 uppercase = GetCounter("uppercase");
3 map(String name, String contents):
4   for each word w in contents:
5     if (IsCapitalized(w)):
6       uppercase->Increment();
7     EmitIntermediate(w, "1");
```

[20] Abk.: The GNU Project Debugger; http://www.gnu.org/software/gdb/
[21] Abk.: Hypertext Transport Protocol

Die Zählerwerte von einzelnen Workern werden periodisch an den Master übertragen (angehängt an einer Ping-Antwort). Der Master fasst die Zählerergebnisse von erfolgreichen Map- oder Reduce-Tasks zusammen und gibt sie an den Programmcode zurück, wenn die MapReduce-Operation abgeschlossen ist. Die aktuellen Zählerwerte werden ebenfalls auf der Hauptstatusseite so angezeigt, dass Menschen den Fortschritt der Berechnung live verfolgen können. Beim Zusammenfassen der Zählerwerte, eliminiert der Master Auswirkungen von mehrfacher Ausführung derselben Map- oder Reduce-Tasks, um eine Doppelzählung zu vermeiden (mehrfache Ausführung kann durch die Verwendung von Backup-Tasks und die fehlerbedingte Neuausführung eines Tasks auftreten).

Einige Zählerwerte werden automatisch von der MapReduce-Bibliothek verwaltet. Zähler sind hilfreich bei Plausibilitätsprüfungen des Verhaltens von MapReduce-Funktionen. Zum Beispiel kann so in einigen MapReduce-Operation vom Programmcode sichergestellt werden, dass die Zahl der Ausgabe Paare exakt der Zahl der verarbeiteten Eingabepaare entspricht, oder dass der Anteil an verarbeiteten deutschen Dokumenten innerhalb eines tolerierbaren Anteils der insgesamt verarbeiteten Dokumente liegt.

5. Performanz

In diesem Kapitel wird die Performanz von MapReduce anhand von zwei Berechnungen gemessen, die auf einem großen Rechnerverbund laufen. Eine Berechnung besteht darin, etwa ein Terabyte Daten nach einem bestimmten Muster zu durchsuchen. Die andere Berechnung sortiert etwa ein Terabyte Daten.

Diese beiden Programme repräsentieren zwei große Untergruppen der Programme, die von Nutzern von MapReduce implementiert werden:

- Programme, die Daten von einer Repräsentation in eine andere überführen.
- Programme, die einen kleinen Anteil interessanter Daten aus einem großen Datenbestand extrahieren.

5.1. Konfiguration des Rechnerverbundes

Alle Programme wurden auf einem Rechnerverbund ausgeführt der aus etwa 1800 Rechnern bestand. Jeder Rechner war bestückt mit zwei 2 Gigahertz Intel Xeon-Prozessoren mit aktiviertem Hyper-Threading, 4 Gigabyte Arbeitsspeicher, zwei 160 Gigabyte IDE-Festplatten und einem Gigabit-Netzwerkadapter. Die Rechner waren in einem baumartig über zwei Ebenen strukturierten Netzwerk miteinander verbunden, was in einer aggregierten Bandbreite von etwa 100-200 Gigabit pro Sekunde an der Wurzel resultiert. Da alle Rechner im gleichen Rechenzentrum untergebracht waren, war die Roundtrip-Zeit zwischen jeder möglichen Paarung von Rechnern unter einer Millisekunde.

Von den 4 Gigabyte Arbeitsspeicher wurden durchschnittlich zwischen 1-1,5 Gigabyte für andere Aufgaben auf dem Rechnerverbund verwendet. Die Programme wurden nachmittags an Wochenenden ausgeführt, da hier die Auslastung von Prozessoren, Festplatten und Netzwerk meist gering war.

5.2. Grep

Das Programm *grep* durchsucht 10^{10} Einträge mit einer Länge von 100 Byte nach einem relativ seltenen Muster, das aus drei Zeichen besteht und nur alle 92.337 Einträge vorkommt. Die Eingabe wird in Segmente zu jeweils etwa 64 Megabyte ($M = 15000$) zerstückelt und die gesamte Ausgabe dann in einer Datei gespeichert ($R = 1$).

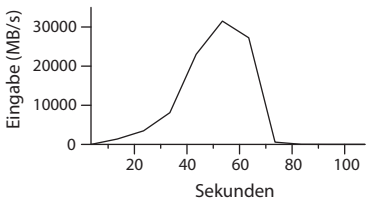

Abbildung 5.1.: Datentransferrate über Zeit

Bei Abbildung 5.1 zeigt sich der Fortschritt der Berechnung über die Zeit. Die Y-Achse zeigt die Rate, mit der die Eingabedaten durchsucht werden. Sie steigt stetig an mit der Anzahl der Rechner, die der MapReduce-Berechnung zugeteilt sind, und erreicht ihren Höchststand mit 30 Gigabit pro Sekunde über 1764 Rechner. Nach Beendigung des Map-Tasks fällt die Rate und erreicht null nach etwa 80 Sekunden Berechnungszeit. Die gesamte Berechnung benötigt

etwa 150 Sekunden von Start bis Ende. Das beinhaltet ca. eine Minute Overhead für den Start. Der Overhead ist begründet in der Verbreitung des Programms über alle beteiligten Rechner, Verzögerungen im Zugriff auf die 1000 Eingabedateien im GFS[22] und die Ermittlung der Informationen, die für die jeweilige lokale Anpassung benötigt werden.

5.3. Sort

Das Programm *sort* sortiert 10^{10} Einträge mit einer Länge von 100 Byte (ungefähr 1 Terabyte Daten). Dieses Programm ist der TeraSort-Benchmark[Gra12] nachempfunden.

Das Sortierprogramm besteht aus weniger als 50 Zeilen Programmcode. Eine dreizeilige *Map*-Funktion extrahiert einen Sortierschlüssel mit einer Länge von 10 Byte aus einer Textzeile und gibt diesen zusammen mit der jeweiligen Textzeile als *Intermediate*-Schlüssel/Wert-Paar aus. Als *Reduce*-Operator wurde eine vorhandene Indentitätsfunktion verwendet. Diese Funktion gibt die Zwischendaten unverändert weiter. Die sortierte Ausgabe wird in einem Satz replizierter GFS-Dateien gespeichert.

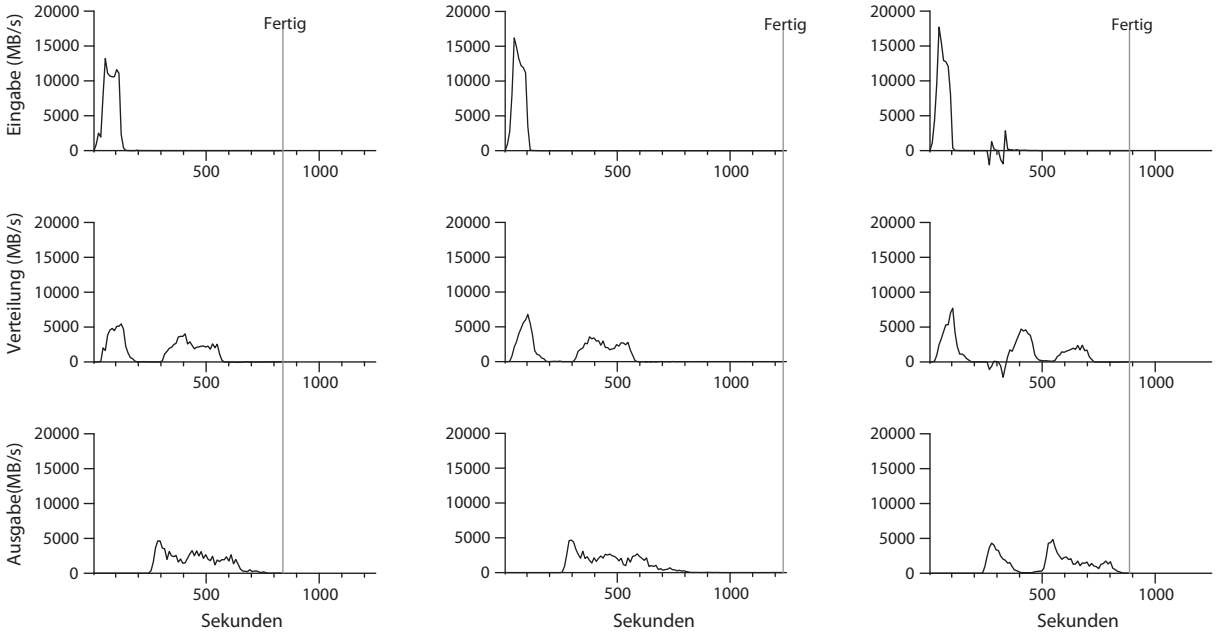

Abbildung 5.2.: Datentransferraten über Zeit verschiedener Ausführungen des Sortierprogramms

Wie bei *grep* wird die Eingabe in Segmente zu jeweils etwa 64 Megabyte ($M = 15000$) zerstückelt jedoch die Ausgabe dann über eine Partitionierungsfunktion in 4000 Dateien gespeichert ($R = 4000$). Die ersten Bytes des Schlüssels entscheiden, in welchem der R Teile er gespeichert wird.

Die verwendete Partitionierungsfunktion für diesen Benchmark verfügt bereits über Wissen über die Verteilung der Schlüssel. In einem allgemeinen Sortierprogramm würde eine MapReduce-Operation im Vorlauf eine zufällige Auswahl von Schlüsseln ermitteln und die Verteilung dieser Schlüssel dazu zu nutzen, sinnvolle Teilungsstellen für die finale Sortierung zu berechnen.

[22] Abk.: Google File System

In Abbildung 5.2 (a) sieht man den Fortschritt bei normaler Ausführung von *sort*. Der Graph oben links zeigt die Rate, mit der die Eingabe eingelesen wird. Sie erreicht ihren Höhepunkt bei 13 Gigabyte pro Sekunde und fällt daraufhin auf null ab, da alle Map-Tasks abgeschlossen sind bevor 200 Sekunden vergangen sind. Die Eingaberate ist deutlich geringer ist als bei *grep*. Die Map-Tasks von *sort* benötigen nur halb soviel Zeit für die Ausführung und schreiben die Zwischendaten, die bei grep aufgrund der Größe vernachlässigt werden können, in das Dateisystem.

Der mittlere Graph in Abbildung 5.2 (a) stellt die Rate dar, mit der Daten von Map-Tasks zu Reduce-Tasks über das Netzwerk versandt werden. Die Verteilung beginnt sobald der erste Map-Task abgeschlossen ist. Der Ausschlag dokumentiert die Verteilung von Reduce-Tasks über 1700 zugewiesene Rechner. Jeder Rechner kann nur jeweils einen Task verarbeiten. Die übrigen Reduce-Tasks werden danach begonnen, wie am zweiten Ausschlag ab etwa Sekunde 300 deutlich wird. Die gesamte Verteilung benötigt hier 600 Sekunden.

Der untere Graph in Abbildung 5.2 (a) zeigt die Rate, mit der Daten der Reduce-Tasks in die Ausgabedateien geschrieben werden. Die Verzögerung zwischen dem Ende der ersten Verteilungsphase und dem Beginn des Schreibens ist dadurch begründet, dass die Rechner die Zwischendaten sortieren. Die Schreibvorgänge werden für eine Weile mit einer Rate von 2-4 Gigabyte pro Sekunde fortgesetzt. Alle Schreibvorgänge sind nach 850 Sekunden abgeschlossen. Inklusive dem Start-Overhead benötigt die gesamte Berechnung 891 Sekunden. Das ist vergleichbar mit dem besten Ergebnis in der TeraSort-Benchmark [Gra12] 2004 (1057 Sekunden).

Es sind ein paar Dinge zu beachten: die Eingaberate ist höher als die Verteilungsrate und die Ausgaberate, da die meisten Daten von einer lokalen Festplatte gelesen werden und damit die Einschränkungen der Bandbreite des zur Verfügung stehenden Netzwerks umgangen werden. Die Verteilungsrate ist höher als die Ausgaberate, weil für die Ausgabe aus Gründen der Zuverlässigkeit und Verfügbarkeit zwei Kopien der sortierten Daten erstellt werden. Dieses Konzept ist auch auf dem verwendeten Filesystem GFS begründet. Die Anforderung für das Netzwerk, Daten zu Schreiben, würden stark reduziert werden, würde ein Dateisystem verwendet werden, das ein *Erasure Coding* [Rab89] statt Replikation einsetzt.

5.4. Effekt durch Backup-Aufgaben

In Abbildung 5.2 (b) wird die Ausführung des Sortierprogramms mit deaktiviertem Backup-Task gezeigt. Der Fluss der Ausführung ist ähnlich wie in Abbildung 5.2 (a) mit dem Unterschied, dass über längere Zeit keine Schreibvorgänge stattfinden. Nach 960 Sekunden sind alle bis auf 5 Reduce-Tasks beendet. Die letzten Nachzügler werden erst 300 Sekunden später abgeschlossen. Die gesamte Berechnung benötigt so 1283 Sekunden. Ohne Backup-Task wird also 44 Prozent mehr Zeit in Anspruch genommen als bei normaler Ausführung.

5.5. Maschinenausfälle

In Abbildung 5.2 (c) wird eine Ausführung demonstriert, in der einige Minuten nach Start der Berechnung absichtlich 200 der 1746 Arbeitsprozesse beendet wurden. Der Scheduler des Rechnerverbundes startet erneut Arbeitsprozesse auf den betroffenen Maschinen.

Der Abbruch der Arbeitsprozesse zeigt sich als negative Eingaberate, da bereits abgeschlossene Map-Daten verschwunden sind und neue Arbeitsprozesse ausgeführt werden müssen. Die Wiederaufnahme der Map-Tasks geht vergleichsweise schnell. Die gesamte Berechnung nimmt so letztendlich 933 Sekunden in Anspruch und damit nur 5 Prozent mehr als die normale Ausführungszeit.

6. Erfahrungen

Die erste Version der MapReduce-Bibliothek wurde im Februar 2003 bei Google entwickelt. Bereits im August 2003 wurden einige Verbesserungen implementiert, wie eine Lokalitätsoptimierung, eine dynamische Lastverteilung der Ausführung über beteiligte Rechnerknoten und weitere. Bis zur Veröffentlichung des Artikels, auf dem diese Arbeit basiert, im Jahr 2004, hat man bei Google mit freudigem Erstaunen festgestellt, wie breit die Einsatzmöglichkeiten der MapReduce-Bibliothek sind im Bezug auf die Probleme, an deren Lösung dort gearbeitet wird, wie beispielsweise:

- Maschinelles Lernen im großen Maßstab
- Verbundprobleme bei den Google News und Froogle Produkten
- Entnahme von Eigenschaften von Webseiten für neue Experimente und Produkte (z. B. Geodaten aus einer großen Menge von Webseiten für die lokalisierte Suche)
- Graph-Berechnungen im großen Maßstab

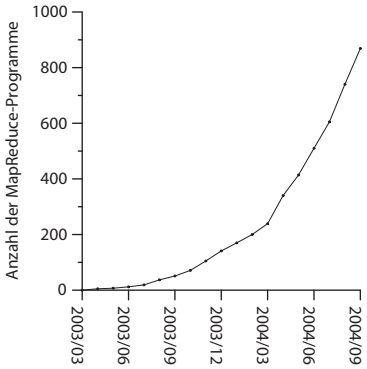

Abbildung 6.1.: MapReduce-Programme in der Google-Codebasis

Anzahl der Jobs	29.423
Durchschn. Ausführungszeit	634 Sekunden
Maschinentage	79.186 Tage
Gelesene Eingabedaten	3.288 TB
Produzierte Zwischendaten	758 TB
Geschriebene Ausgabedateien	193 TB
Durchschn. Worker pro Job	157
ausgefallene Worker pro Job	1,2
Durchschn. Map-Tasks pro Job	3.351
Durchschn. Reduce-Tasks pro Job	55
Map Implementierungen	395
Reduce Implementierungen	269
Map/Reduce Kombinationen	426

Tabelle 6.1.: Map Reduce Jobs bei Google August 2004

In der Abbildung 6.1 wird die Anzahl einzelner MapReduce-Programme dargestellt, die bei Google eingecheckt worden sind. Die Steigerung der Anzahl von 0 auf 900 über den Zeitraum von Anfang 2003 bis Ende 2004 ist rasant. MapReduce ist so erfolgreich, weil es ermöglicht, ein einfaches Programm zu schreiben und es effizient über tausende von Rechnern in sehr kurzer Zeit auszuführen. Dies hat Entwicklungs- und Prototypisierungszyklen bei Google massiv beschleunigt. Darüber hinaus haben Programmierer leichten Zugriff auf Ressourcen verteilter oder parallelisierter Systemen, ohne Erfahrung in der Softwareentwicklung für solche Systemen haben zu müssen.

Nach jedem MapReduce-Job werden statistische Daten über den Ressourcenverbrauch von der MapReduce-Bibliothek gespeichert. Tabelle 6.1 zeigt dies für einen Satz von MapReduce-Jobs die im August 2004 bei Google durchgeführt wurden.

6.1. Indizierung im großen Maßstab

Eines der wichtigsten Einsatzgebiete für die MapReduce-Bibliothek war Stand 2004 die komplette Neuprogrammierung des Indizierungsdienstes, der die Datenstrukturen liefert, die für die Google-Suchmaschine verwendet werden. Das Indizierungssystem verarbeitet als Eingabe einen großen Satz Dokumente, die vom *Crawling-System* stammen und als GFS-Dateien abgelegt

wurden. Der Rohinhalt dieser Dokumente umfasste mehr als 20 Terabyte Daten. Der Indizierungsprozess läuft als eine Sequenz von 5-10 MapReduce-Operationen. MapReduce hat dem vorher genutzten Verfahren gegenüber diverse Vorteile:

- Der Programmcode für die Indizierung ist einfacher, kürzer und leichter zu verstehen, da der komplexe Programmcode, der sich mit Fehlertoleranz, Verteilung und Parallelisierung auseinandersetzt, in der MapReduce-Bibliothek verborgen ist. Beispielsweise schrumpfte die Größe des Programmcodes in der Programmiersprache C++, der für eine Phase der Berechnung benötigt wird, von 3800 Zeilen auf 700 Zeilen ausgedrückt in MapReduce.
- Die Performanz der MapReduce-Bibliothek ist gut genug, dass konzeptionell nicht relevante Berechnungen separat ausgeführt werden können, anstatt zur Vermeidung weiterer Läufe direkt eingemischt zu werden. Dies erleichtert es, den Indizierungsprozess zu verändern. Beispielsweise hat eine bestimmte Veränderung im Indizierungssystem von Google nur Tage in Anspruch genommen, wo vor Einsatz von MapReduce noch Monate benötigt wurden.
- Der Indizierungsprozess ist viel leichter zu benutzen, da die meisten Probleme, die durch Maschinenausfälle, langsame Rechner und gestörte Netzwerke auftreten, durch die MapReduce-Bibliothek ohne Eingriff des Nutzers automatisch abgefangen werden. Darüber hinaus ist es leicht, die Performanz des Indizierungsprozesses durch das Hinzufügen weiterer Rechner in den Indizierungsverbund zu verbessern.

7. Fazit und kritische Bewertung

7.1. Zusammenfassung

Das Kapitel 2 vermittelte die Konzepte von MapReduce. In den Kapiteln 3 und 4 wurden die konkreten Ansätze von Jeffrey Deans und Sanjay Ghemawats Implementierung veranschaulicht, Optimierungen näher betrachtet und deren Implementierung klar begründet. Das Kapitel 5 hat die Testergebnisse der Implementierung anhand zweier Programme demonstriert.

Kapitel 6 hat die positiven Erfahrungen vorgestellt, die Google mit dem Einsatz der MapReduce-Bibliothek gemacht hat. Sie haben verdeutlicht, dass MapReduce, entsprechende Ressourcen vorausgesetzt, eine effektive Technologie zur verteilten Verarbeitung großer Datenmengen ist.

7.2. Verwandte Arbeiten

Viele Systeme stellen eingeschränkte Programmiermodelle zur Verfügung, um Berechnungen automatisch zu parallelisieren. Zum Beispiel können parallele Präfixberechnungen einer assoziativen Funktion über alle Präfixe von einem N Elemente großen Array in der Zeit log N auf N Prozessoren berechnet werden [Ble89; Gor96; LMF80]. Auf den Erfahrungen von Google basierend kann MapReduce als Vereinfachung und Destillat einiger dieser Modelle aufgefasst werden.

Bulk Synchronous Programming [Val90] und einige *MPI*[23]-Primitive [GLS94] stellen eine höhere Abstraktionsebene zu Verfügung und machen es für den Programmierer einfacher, parallele Programme zu schreiben. Ein wesentlicher Unterschied zwischen diesen Systemen und MapReduce ist, dass MapReduce ein eingeschränktes Programmiermodell zur automatischen Parallelisierung des Nutzerprogramms ausnutzt und eine transparente Fehlertoleranz bietet.

Die Lokalitätsoptimierung wurde von Techniken wie *Active Disks* [HSW+04; RFGN01] inspiriert, bei denen die Berechnungen in Verarbeitungselemente verschoben werden, die nah an der lokalen Festplatte sind. Dadurch wird die Datenmenge, die über das Netzwerk versandt wird, reduziert.

Der Backup-Task-Mechanismus ist dem „*Eager Scheduling*“ ähnlich, das im Charlotte-System [BKKW96] angewendet wird. Eine Schwäche von einfachem *Eager Scheduling* ist, dass die gesamte Berechnung nicht abgeschlossen wird, wenn eine gegebene Aufgabe wiederholt Fehler verursacht. MapReduce löst dieses Problem mit dem Mechanismus zum Überspringen fehlerhafter Datensätze.

Die MapReduce-Implementierung stützt sich auf ein „in-house Cluster Management“-System, dass für die Verteilung und den Ablauf von Benutzer-Tasks auf einer großen Sammlung geteilter Systeme verantwortlich ist und ähnelt Cluster-Management-Systemen wie Condor [TTL05].

Die Sortierfunktion, die Teil der MapReduce-Bibliothek ist, ähnelt in ihrer Verarbeitung NOW-Sort [ADADC+97]. Map-Worker partitionieren die zu sortierenden Daten und senden sie an Reduce-Worker, die alle ihre Daten lokal sortieren (wenn möglich im Speicher).

River [ADAT+99] verfügt über ein Programmiermodell, bei dem Prozesse miteinander kommunizieren, indem sie Daten über verteile Warteschlangen senden. Ähnlich wie MapReduce versucht das River-System durch sorgfältiges *Scheduling* der Festplatten- und Netzwerktransfers, eine gute gleichmäßige Performanz und ausgewogene Fertigstellungszeiten zu erreichen. MapReduce verfolgt mit der Beschränkung des Programmiermodells einen anderen Ansatz. Es ist in der

[23] Abk.: Message Passing Interface

Lage, das Problem in eine große Zahl kleiner Tasks zu zerlegen und dynamisch auf verfügbare Worker zu verteilen, so dass schnellere Worker mehr Tasks verarbeiten.

Das Programmiermodell von *BAD-FS*[24] [BTAD+04] ist im Vergleich zu MapReduce sehr unterschiedlich und hat im Gegensatz dazu die Ausführung von Jobs über ein WAN[25] zum Ziel. Fundamentale Gemeinsamkeiten sind jedoch, dass beide Systeme sowohl redundante Ausführung zur Datenwiederherstellung bei Verlust durch Ausfälle, als auch „*Locality-aware Scheduling*“ zur Reduzierung der Datenmenge, die über das Netzwerk gesendet werden muss, nutzen.

TACC[26] [FGC+97] ist ein System, um die Konstruktion von Hochverfügbarkeits-Netzwerkdiensten zu vereinfachen. Wie MapReduce stützt es sich auf die Wiederausführung als einen Mechanismus, um Fehlertoleranz zu erreichen.

7.3. Fazit

Das Forscherteam bei Google hat gezeigt, dass sich die Ressourcen von großen Rechnerverbünden über MapReduce nahezu beliebig nutzen und mit der Größe der Eingabedaten skalieren lassen. Die Gesamtgröße der zu verarbeitenden Daten als limitierender Faktor fällt damit weg.

Dadurch, dass sich viele praktische Probleme in MapReduce ausdrücken lassen, erschließen sich weitreichende Möglichkeiten der Datenverarbeitung mit einer Skalierbarkeit, die nur durch das darunterliegende Dateisystem begrenzt wird.

7.4. Entwicklungen seit Erstveröffentlichung

Seit Erstveröffentlichung der Arbeit von Jeffrey Dean und Sanjay Ghemawat 2004 hat sich die Technologie weit verbreitet. MapReduce-Implementierungen, wie seit 2008 Apache Hadoop[27], erleben eine rasante Entwicklung. Sie ermöglichen es Betreibern von Rechenzentren, ihre Ressourcen effektiver einzusetzen oder freie Ressourcen besser zu vermarkten. Diensteanbieter wie Yahoo oder Facebook können neue Nutzungsmöglichkeiten für ihre großen Datenbestände erschließen oder bestehende Prozesse, wie Indizierung, deutlich beschleunigen.

Analog zu GFS in MapReduce von Google, wird bei Hadoop HDFS[28] eingesetzt. Kernkomponenten von Hadoop wurden in der Programmiersprache Java implementiert. Obwohl Google 2010 ein Patent für die MapReduce-Technologie erhalten hat [DG10], konnte die ASF[29] die Hadoop-Nutzer kurz darauf mit der Nachricht beruhigen, dass die freie Nutzung der Technologie über ein Lizenzabkommen mit Google garantiert wird [Ros10].

7.5. Ausblick

In einer vernetzten Welt, in der immer mehr Vorgänge digital abgebildet werden, entsteht quasi nebenbei eine Fülle von Daten. Die Rate, mit der neues Datenmaterial hinzukommt, steigt stetig. Diese enormen Datenmengen verfügbar und auf vielfältige Weise analysierbar zu machen, wird heute unter dem Schlagwort *Big Data* zusammengefasst. Herausforderungen und Chancen dieser Entwicklung, die klar zu MapReduce zurückführt, sind vielfältig und können sich direkt auf die Lebensführung aller Menschen auswirken [BC11].

[24] Abk.: Batch-Aware Distributed File System
[25] Abk.: Wide Area Network
[26] Abk.: Texas Advanced Computing Center
[27] http://hadoop.apache.org
[28] Abk.: Hadoop Distributed File System
[29] Abk.: Apache Software Foundation

Literaturverzeichnis

ADADC+97 Arpaci-Dusseau, Andrea C. ; Arpaci-Dusseau, Remzi H. ; Culler, David E. ; Hellerstein, Joseph M. ; Patterson, David A.: High-performance sorting on networks of workstations. In: *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. New York and NY and USA : ACM, 1997 (SIGMOD '97). – ISBN 0–89791–911–4, 243–254 7.2

ADAT+99 Arpaci-Dusseau, Remzi H. ; Anderson, Eric ; Treuhaft, Noah ; Culler, David E. ; Hellerstein, Joseph M. ; Patterson, David ; Yelick, Kathy: Cluster I/O with River: Making the fast case common. In: *In Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99*, 1999, S. 10–22 7.2

BC11 Boyd, Danah ; Crawford, Kate: Six Provocations for Big Data. In: *Social Science Research Network Working Paper Series* (2011) 7.5

BDH03 Barroso, L.A ; Dean, J. ; Holzle, U.: Web search for a planet: the google cluster architecture. In: *IEEE Micro* 23 (2003), Nr. 2, S. 22–28. `http://dx.doi.org/10.1109/MM.2003.1196112`. – DOI 10.1109/MM.2003.1196112 3

BKKW96 Baratloo, Arash ; Karaul, Mehmet ; Kedem, Zvi ; Wyckoff, Peter: Charlotte: Metacomputing on the web. In: *In Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996 7.2

Ble89 Blelloch, G. E.: Scans as primitive parallel operations. In: *IEEE Transactions on Computers* (1989), Nr. 38, S. 1526–1538 7.2

BTAD+04 Bent, John ; Thain, Douglas ; Arpaci-Dusseau, Andrea C. ; Arpaci-Dusseau, Remzi H. ; Livny, Miron: Explicit control a batch-aware distributed file system. In: *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*. Berkeley and CA and USA : USENIX Association, 2004 (NSDI'04), 27 7.2

DG04 Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: simplified data processing on large clusters. In: *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. Berkeley and CA and USA : USENIX Association, 2004 (OSDI'04), 10 1.1

DG10 Dean, Jeffrey ; Ghemawat, Sanjay: *United States Patent: 7650331 - System and method for efficient large-scale data processing.* `http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=%2Fnetahtml%2FPTO%2Fsrchnum.htm&r=1&f=G&l=50&s1=7,650,331.PN.&OS=PN/7,650,331&RS=PN/7,650,331`. Version: 2010 7.4

FGC+97 Fox, Armando ; Gribble, Steven D. ; Chawathe, Yatin ; Brewer, Eric A. ; Gauthier, Paul: Cluster-based scalable network services. In: *Proceedings of the sixteenth ACM symposium on Operating systems principles*. New York and NY and USA : ACM, 1997 (SOSP '97). – ISBN 0–89791–916–5, 78–91 7.2

GGL03 Ghemawat, Sanjay ; Gobioff, Howard ; Leung, Shun-Tak: The Google file system. In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York and NY and USA : ACM, 2003 (SOSP '03). – ISBN 1–58113–757–5, 29–43 1.1, 4, 3.4

GLS94 Gropp, William ; Lusk, Ewing ; Skjellum, Anthony: *Using MPI: portable parallel programming with the message-passing interface*. Cambridge and MA and USA : MIT Press, 1994. – ISBN 0–262–57104–8 7.2

Gor96 Gorlatch, Sergei: Systematic Efficient Parallelization of Scan and Other List Homomorphisms. In: *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*. London and UK and UK : Springer-Verlag, 1996 (Euro-Par '96). – ISBN 3–540–61627–6, 401–408 7.2

Gra12 Gray, Jim: *sortbenchmark.org*. `http://sortbenchmark.org/`. Version: 2012 5.3, 5.3

HSW+04 Huston, Larry ; Sukthankar, Rahul ; Wickremesinghe, Rajiv ; Satyanarayanan, M. ; Ganger, Gregory R. ; Riedel, Erik: Diamond: A storage architecture for early discard in interactive search. In: *In Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04*, 2004 7.2

LMF80 Ladner, Richard E. ; Michael ; Fischer, J.: Parallel Prefix Computation. In: *Journal of the ACM* 27 (1980), S. 831–838 7.2

Rab89 Rabin, Michael O.: Efficient dispersal of information for security, load balancing, and fault tolerance. In: *J. ACM* 36 (1989), Nr. 2, 335–348. `http://dx.doi.org/10.1145/62044.62050`. – DOI 10.1145/62044.62050. – ISSN 0004–5411 5.3

RFGN01 Riedel, E. ; Faloutsos, C. ; Gibson, G.A ; Nagle, D.: Active disks for large-scale data processing. In: *Computer* 34 (2001), Nr. 6, 68–74. `http://dx.doi.org/10.1109/2.928624`. – DOI 10.1109/2.928624 7.2

Ros10 Rosen, Lawrence: *Re: License for Google's patent*. `http://mail-archives.apache.org/mod_mbox/hadoop-general/201004.mbox/%3C121803A3-CFB9-489B-96EF-027234E55D25@apache.org%3E`. Version: 2010 7.4

TTL05 Thain, Douglas ; Tannenbaum, Todd ; Livny, Miron: Distributed computing in practice: the Condor experience. In: *Concurrency and Computation: Practice and Experience* 17 (2005), Nr. 2-4, S. 323–356. `http://dx.doi.org/10.1002/cpe.938`. – DOI 10.1002/cpe.938. – ISSN 1532–0626 7.2

Val90 Valiant, Leslie G.: A bridging model for parallel computation. In: *Commun. ACM* 33 (1990), Nr. 8, 103–111. `http://dx.doi.org/10.1145/79173.79181`. – DOI 10.1145/79173.79181. – ISSN 0001–0782 7.2

A. Anhang

A.1. Codebeispiel

Listing A.1: C++: Wortfrequenzen ermitteln mit MapReduce

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
   const string& text = input.value();
   const int n = text.size();
   for (int i = 0; i < n; ) {
    // Skip past leading whitespace
    while ((i < n) && isspace(text[i]))
     i++;

   // Find word end
   int start = i;
   while ((i < n) && !isspace(text[i]))
    i++;
   if (start < i)
    Emit(text.substr(start,i-start),"1");
  }
 }
};

REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder : public Reducer {
 virtual void Reduce(ReduceInput* input) {
  // Iterate over all entries with the
  // same key and add the values
  int64 value = 0;
  while (!input->done()) {
   value += StringToInt(input->value());
   input->NextValue();
  }

  // Emit sum for input->key()
  Emit(IntToString(value));
 }
};

REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
 ParseCommandLineFlags(argc, argv);

 MapReduceSpecification spec;

 // Store list of input files into "spec"
 for (int i = 1; i < argc; i++) {
  MapReduceInput* input = spec.add_input();
  input->set_format("text");
  input->set_filepattern(argv[i]);
  input->set_mapper_class("WordCounter");
 }

 // Specify the output files:
 // /gfs/test/freq-00000-of-00100
 // /gfs/test/freq-00001-of-00100
 // ...
 MapReduceOutput* out = spec.output();
 out->set_filebase("/gfs/test/freq");
 out->set_num_tasks(100);
 out->set_format("text");
 out->set_reducer_class("Adder");

 // Optional: do partial sums within map
 // tasks to save network bandwidth
 out->set_combiner_class("Adder");

 // Tuning parameters: use at most 2000
 // machines and 100 MB of memory per task
 spec.set_machines(2000);
 spec.set_map_megabytes(100);
 spec.set_reduce_megabytes(100);

 // Now run it
 MapReduceResult result;
 if (!MapReduce(spec, &result)) abort();

 // Done: 'result' structure contains info
 // about counters, time taken, number of
 // machines used, etc.
 return 0;
}
```