MapReduce

Dhyan Blum
Betreuer: Dirk Haage
Seminar Innovative Internettechnologien und Mobilkommunikation SS2010
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: blumd@in.tum.de

KURZFASSUNG

Die Firma Google verarbeitete im Jahr 2007 täglich 20 Petabyte an Daten [8]. Diese Leistung wäre ohne den massiven Einsatz verteilter, paralleler Systeme nicht möglich. Die Parallelisierung von Programmen ist jedoch eine komplexe Aufgabe, der zusätzliche Entwicklungsaufwand fällt zudem bei jeder weiteren verteilten Anwendung erneut an.

Diese Arbeit beschreibt das von Google entwickelte Modell *MapReduce* und dessen Implementierung. Mit *MapReduce* können extrem große Datensätze in verteilten Umgebungen verarbeitet oder generiert werden, ohne dass sich der Entwickler mit Details der Parallelisierung beschäftigen oder über Vorkenntnisse in diesem Bereich verfügen muss. Er hat lediglich die Funktionen *map* und *reduce* zu implementieren, da sich das *MapReduce*-Framework bei der Programmausführung automatisch um Parallelisierung, Lastverteilung, Fehlerbehandlung und die Kommunikation der Rechner untereinander kümmert.

Vielfältige Anwendungsmöglichkeiten ergeben sich vor allem in den Bereichen Indexierung, Tokenisierung, Suche, Erzeugung von Graphen und Datenstrukturen, maschinelles Lernen sowie *Data Mining* und Stapelverarbeitung. Bislang kaum dokumentiert ist hingegen die Verarbeitung und Analyse von Netzwerkmessdaten, was gerade deshalb verwundert, da hier mitunter sehr große Mengen an Verkehrsdaten anfallen. Heute wird das Modell *MapReduce* von vielen Entwicklern erfolgreich eingesetzt, vor allem große IT-Firmen sind in diesem Bereich stark vertreten.

1. EINLEITUNG

In den letzten beiden Jahrzehnten fand Datenverarbeitung in zunehmend größeren Dimensionen statt, gerade in Netzwerken fallen inzwischen riesige Datenmengen an. Gleichzeitig lässt sich eine Entwicklung hin zu einer immer umfassenderen Verarbeitung und Analyse dieser Daten beobachten, etliche Firmen operieren hier bereits im Petabyte-Bereich. Die Notwendigkeit, derart große Datenmengen verarbeiten zu können, wird speziell bei Suchmaschinen im Internet sehr deutlich. Um Suchanfragen möglichst schnell und umfassend bedienen zu können, müssen Milliarden von Internetseiten eingelesen, aufbereitet und analysiert werden.

Um Probleme dieser Größenordnung in annehmbarer Zeit lösen zu können, reichen einzelne Rechenmaschinen nicht mehr aus. Es müssen daher hunderte bis tausende von Computern zu einem Rechnerverbund zusammengeschlossen werden. Damit diese kooperativ arbeiten können, müssen die Eingabedaten sinnvoll verteilt und die eigentliche Berechnung parallelisiert werden. Anschließend müssen die Ergebnisse noch überprüft, zusammengeführt und unter Umständen dauerhaft gespeichert werden.

Dabei werden Entwickler vor eine ganze Reihe von Herausforderungen gestellt: Wie wird ein Problem richtig aufgeteilt? Sind überhaupt alle Probleme und ihre Berechnung parallelisierbar? Wie können die Rechner im Verbund koordiniert werden? Wie reagiert man auf Hardwareausfälle und Fehler bei der Verarbeitung? Nicht zuletzt stellt sich die Frage, wie Ergebnisse gespeichert werden, die zu groß für einzelne Computer sind.

Obwohl das ursprüngliche Problem und die nötigen Berechnungen häufig einfach sind, muss der Entwickler also eine vergleichsweise komplexe Lösung finden. Ein Großteil der Anstrengungen wird dabei in die Parallelisierung statt in die Lösung der Aufgabe investiert. Und das immer wieder, obwohl sich nur die Anwendung, nicht aber der Unterbau für die Ausführung auf verteilten Systemen ändert.

Entwickler der Firma Google haben daher im Jahr 2004 eine Lösung in Form eines Programmiermodells und einer zugehörigen Implementierung entwickelt. Sie ließen sich dabei von den in funktionalen Programmiersprachen verbreiteten *map*- und *reduce*-Funktionen inspirieren und nannten das Verfahren daher *MapReduce*. *MapReduce* kann als Framework aufgefasst werden, dessen Abstraktionsschicht alle Details der Parallelisierung vor dem Entwickler verbirgt. Dieser muss nur mehr die beiden vorgegebenen Funktionen *map* und *reduce* implementieren und kann sich somit voll auf die Lösung seines eigentlichen Problems konzentrieren.

Diese Arbeit gibt zunächst eine Einführung in die grundlegende Funktionsweise von *MapReduce*. Anschließend werden einige Beispiele vorgestellt und es wird ausführlich auf den Programmablauf eingegangen. In Abschnitt 4 folgt eine Beschreibung der Architektur und Details der Google-Implementierung von *MapReduce*. In Abschnitt 5 werden Anwendungsmöglichkeiten des Modells beschrieben, mögliche Anforderungen an Umgebung und Daten besprochen und schließlich auch die Grenzen von *MapReduce* aufgezeigt. Abschnitt 8 widmet sich verschiedenen alternativen Implementierungen des Modells. Zuletzt folgt ein Verweis auf wichtige verwandte Arbeiten und ein zusammenfassendes Fazit dieser Arbeit und ihrer Ergebnisse.

2. FUNKTIONSWEISE

Abbildung 1 veranschaulicht die grundlegende Funktionsweise von *MapReduce*. Eingabe und Ausgabe aller Berechnungen sind Mengen von `(key,value)`-Paaren. Die vom Benutzer zu implementierenden Funktionen *map* und *reduce* sind wie folgt definiert:

- *map* nimmt ein Paar `(key1,value1)` entgegen und gibt eine Liste von Paaren `list(key2,value2)` zurück. Diese Liste stellt das Zwischenergebnis dar. *MapReduce* gruppiert alle Zwischenergebnisse nach ihren Schlüsseln `key2` und reicht diese an *reduce* weiter.

- *reduce* nimmt einen Schlüssel `key2` sowie eine Liste von Werten `list(value2)` entgegen und gibt eine potentiell kleinere Liste von Werten - in der Regel nur einen oder gar keinen Wert - zurück.

3. BEISPIELE

3.1 Worthäufigkeit

Ein Problem, das sich sehr einfach für die Lösung mittels *MapReduce* formulieren lässt, ist die Ermittlung der Worthäufigkeit jedes eindeutigen Wortes in einer Menge von Dokumenten. Der Quelltext, den der Entwickler schreiben würde, sähe vereinfacht wie in Listing 1 gezeigt aus. Die *map*-Funktion gibt jedes gefundene Wort zusammen mit seiner Häufigkeit zurück, wobei die Häufigkeit in diesem einfachen Beispiel schlicht eine Eins für jedes Auftreten des Wortes ist. Die *reduce*-Funktion erhält anschließend die gruppierten Ergebnisse der *map*-Funktion, also ein einzelnes Wort und eine Liste von Einsen. Alles was die *reduce*-Funktion jetzt noch zu tun hat, ist die Einsen für jedes Wort zu summieren. Abbildung 2 illustriert den Ablauf des Programms.

```
map(String key, String value):
    // key:   document name
    // value: document contents
    for each word w in value:
        emitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each count c in values:
        result += ParseInt(c);
        emit(AsString(result));
```

Listing 1: Worthäufigkeit

3.2 Weitere Beispiele

Verteiltes Suchen

Eine *grep*-ähnliche Suchfunktion für die verteilte Suche in großen Datenmengen lässt sich ebenfalls einfach formulieren. Die *map*-Funktion gibt dabei eine Zeile zurück, wenn diese auf das Suchmuster zutreffende Zeichenketten enthält. Da weiter kein *reduce*-Vorgang nötigt ist, stellt *reduce* einfach nur die Identitätsfunktion dar und reicht die Zwischenergebnisse zur Ausgabe durch.

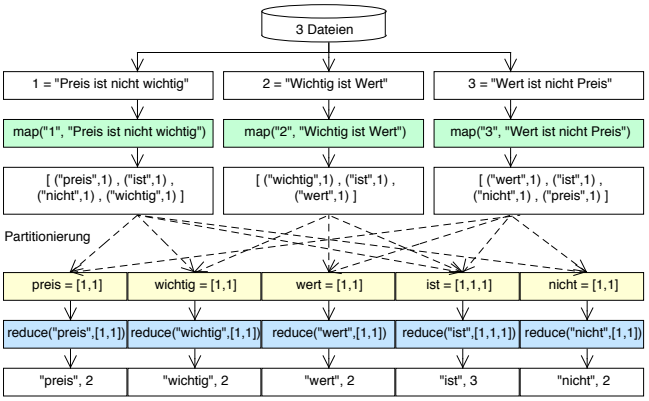

Abbildung 2: Ermittlung der Häufigkeitsverteilung von Wörtern mittels MapReduce

Zugriffsstatistiken

Um Zugriffe auf eine Menge von Internetseiten auszuwerten, lässt man die *map*-Funktion Logdateien verarbeiten und ein `(URL,"1")`-Paar für jeden Zugriff, also für jedes einzelne Vorkommen einer URL, zurückgeben. Die *reduce*-Funktion summiert alle zu einer URL gehörenden Einsen auf und gibt dann ein `(URL,totalCount)`-Paar zurück.

Verteiltes Sortieren

Beim verteilten Sortieren extrahiert die *map*-Funktion den Schlüssel eines jeden Datums und gibt ein `(key,data)`-Paar zurück. Die *reduce*-Funktion gibt die Ergebnisse erneut unverändert weiter. Die eigentliche Sortierung geschieht durch Ausnutzen der Ordnungsgarantie von *MapReduce* (vgl. Abschnitt 4.7).

4. ARCHITEKTUR

Der folgende 4. Abschnitt beschreibt die Architektur von *MapReduce*. Sofern nicht anders angegeben, ist dabei stets Googles Implementierung des Frameworks gemeint. Diese wurde in C++ geschrieben und zielt auf große Rechnerverbunde mit gewöhnlicher Hardware ab. *MapReduce* als Modell ermöglicht jedoch Implementierungen in beliebigen Sprachen und die Anpassung an unterschiedlichste Zielsysteme. Seit 2004 sind viele weitere Implementierungen veröffentlicht worden, von denen einige in Abschnitt 8 vorgestellt werden.

4.1 Google File System

Google verwendet insbesondere für seine Suchmaschine, aber auch für *MapReduce*, ein selbst entwickeltes, verteiltes Dateisystem namens *Google File System (GFS)*. GFS ist ein skalierbares System für verteilte Anwendungen und große Datenmengen. Es wurde daher mit besonderem Augenmerk auf Fehlertoleranz, Performanz und den Einsatz günstiger Standardhardware konzipiert [18]. Alternative *MapReduce* Implementierungen bauen in der Regel auf vergleichbaren Dateisystemen auf.

4.2 Vorbereitung

Bevor ein Entwickler *MapReduce* Operationen durchführen kann, muss er ein Benutzerprogramm schreiben. Dieses im-

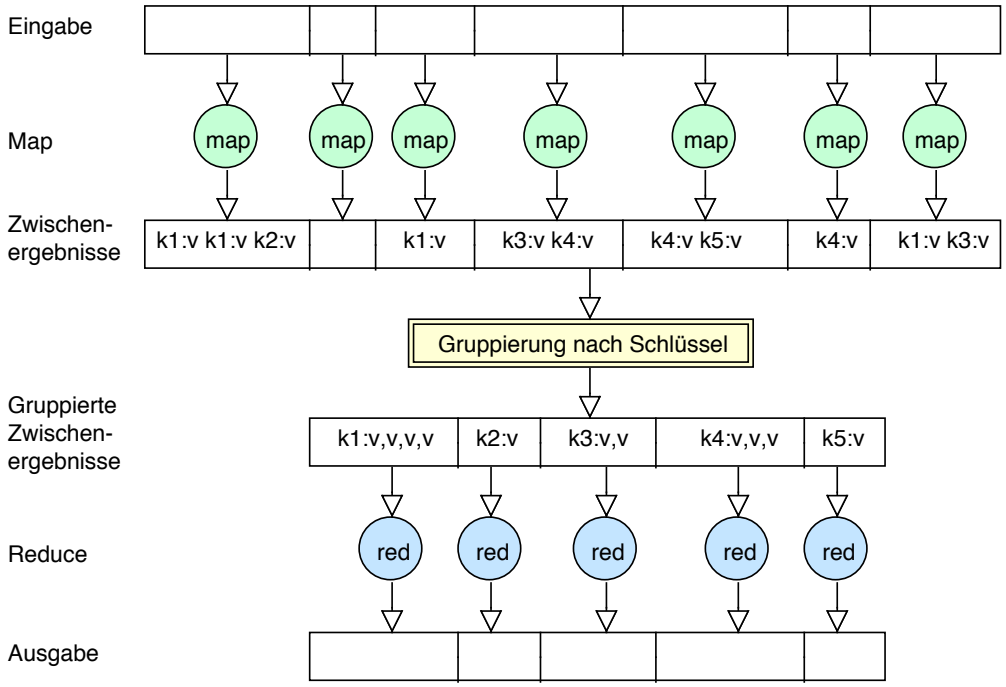

Abbildung 1: Grundlegende Funktionsweise von MapReduce

plementiert die vorgeschriebenen Funktionen *map* und *reduce* und erzeugt ein Objekt vom Typ `MapReduceSpecification`. Diesem Objekt werden die Namen der Ein- und Ausgabedateien sowie optional einige Parameter zur Optimierung der Berechnungen übergeben. Zuletzt ruft das Programm die Funktion `MapReduce()` auf und übergibt ihr das Spezifikationsobjekt. Von nun an übernimmt die *MapReduce*-Bibliothek die Kontrolle, das Benutzerprogramm wartet.

4.3 Ausführung

Abbildung 3 zeigt die einzelnen Phasen, die während der Ausführung durchlaufen und im folgenden beschrieben werden:

1. Die *MapReduce*-Bibliothek teilt die Eingabedateien automatisch in gleich große Teile auf, wobei die Stückgröße mittels optionaler Parameter vom Benutzer bestimmt werden kann. Anschließend startet die Bibliothek auf jedem angeschlossenen Rechner eine Kopie des Benutzerprogramms.

2. Eine spezielle Programmkopie ist das Masterprogramm. Genau ein Rechner erhält eine solche Kopie und wird damit zum Master, alle anderen Rechner agieren fortan als Worker. Worker, die sich im Leerlauf befinden, erhalten vom Master Aufgaben zugeteilt, die entweder aus *map*- oder *reduce*-Schritten bestehen.

3. Worker, denen eine *map*-Aufgabe zugewiesen wurde, lesen die zugehörigen Eingabedaten ein, extrahieren `(key,value)`-Paare und geben diese an die *map*-Funktion des Benutzerprogramms weiter. Die Zwischenergebnisse der *map*-Funktion werden anschließend im Arbeitsspeicher des Workers gepuffert.

4. In regelmäßigen Abständen werden die gepufferten Zwischenergebnisse gemäß einer Partitionsfunktion (vgl. Abschnitt 4.7) aufgeteilt und auf die lokale Festplatte des Workers geschrieben. Der Speicherort dieser Zwischenergebnisse wird dann dem Master mitgeteilt, der für die Weiterleitung der Information an die *reduce*-Worker verantwortlich ist.

5. Wenn ein Worker vom Master über diese Speicherorte informiert wurde, verwendet er einen *Remote Procedure Call* um die Daten von der lokalen Festplatte des *map*-Workers zu lesen. Sobald alle Zwischenergebnisse eingelesen wurden, sortiert sie der *reduce*-Worker nach ihren Schlüsseln, so dass gleiche Schlüssel gruppiert werden. Die Sortierung ist notwendig, da in der Regel viele verschiedene Schlüssel an den gleichen *reduce*-Task weitergegeben werden. Sind die Zwischenergebnisse zu groß für den Arbeitsspeicher, wird eine externe Sortierung durchgeführt.

6. Der *reduce*-Worker durchläuft nun alle sortierten Zwischenergebnisse und reicht für jeden eindeutigen Schlüssel ein Paar bestehend aus eben diesem Schlüssel und einer Liste der zugehörigen Werte an die *reduce*-Funktion weiter. Wie schon am Beispiel der Häufigkeitsverteilung von Wörtern gesehen, schrumpft die Datenmenge

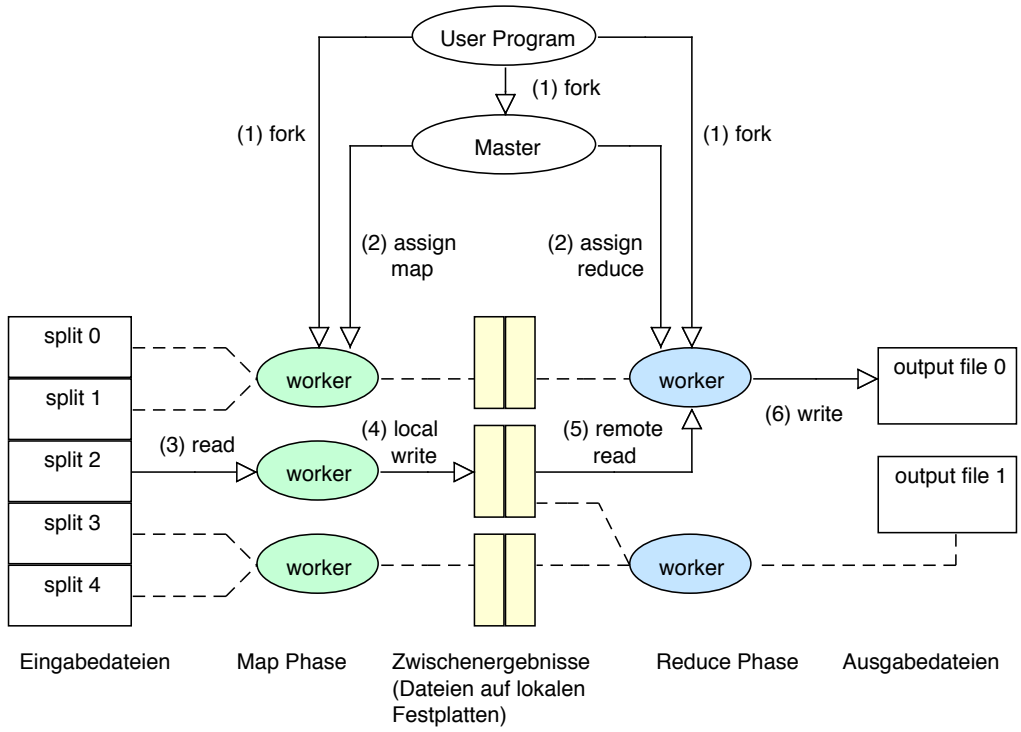

Abbildung 3: Ausführungsübersicht

nach einem Aufruf der *reduce*-Funktion in der Regel erheblich. Das Ergebnis der Funktion wird nun in eine zum aktuellen Schlüssel gehörende Ausgabedatei geschrieben. Die Namen der Ausgabedateien entsprechen der Spezifikation des Benutzers.

7. Wenn alle *map*- und *reduce*-Tasks abgeschlossen sind, weckt der Master das Benutzerprogramm wieder auf und gibt die Kontrolle an selbiges zurück. Die Ausführung wird dann nach dem Aufruf von *MapReduce* fortgesetzt.

4.4 Attribute des Masters

Der Master ist ein gewöhnlicher Rechner des Clusters, der vor der eigentlichen Ausführung eine spezielle Masterkopie des Benutzerprogramms erhalten hat. Um seine Rolle als Koordinator ausfüllen zu können, muss er eine Reihe von Datenstrukturen mit Informationen über den Zustand der aktuellen Operation verwalten. So speichert er für jeden *map*- und *reduce*-Task dessen jeweiligen Status (*idle*, *in progress* oder *completed*) sowie die Identität des zugehörigen Workers, falls der Task gerade ausgeführt wird. Außerdem muss er die Speicherorte von Zwischenergebnissen verwalten, um diese Information später entsprechenden *reduce*-Tasks zur Verfügung stellen zu können.

4.5 Fehlerbehandlung

Googles Implementierung von *MapReduce* ist auf einen Rechnerverbund zugeschnitten, der aus hunderten bis tausenden Computern mit handelsüblicher Hardware besteht. Ausfälle von Rechnern sind daher an der Tagesordnung und müssen von der Bibliothek toleriert werden. Solange die benutzerdefinierten Funktionen *map* und *reduce* deterministisch sind, garantiert *MapReduce* außerdem identische Ergebnisse wie bei vollständig sequentieller Ausführung in einer ausfallsicheren Umgebung. Um diese Eigenschaft sicherstellen zu können, wird für konkurrierende Programmpunkte auf atomare Operationen des darunter liegenden Dateisystems zurückgegriffen.

Fehler der Worker

Worker fallen häufig aus, daher muss unbedingt eine konstruktive Fehlerbehandlung stattfinden. In Googles Implementierung senden die Worker in regelmäßigen Abständen einen *ping* an den Master, um zu signalisieren, dass sie noch aktiv sind. Meldet sich ein Worker über einen festgelegten Zeitraum nicht mehr, so markiert ihn der Master als *failed* und versetzt alle vom Worker aktuell in Arbeit befindlichen und bereits verarbeiteten *map*-Tasks in den Zustand *idle*. Ein Task, der sich im Zustand *idle* befindet, kann vom Master wieder einem freien Worker zugewiesen werden. Auch die vollständig ausgeführten *map*-Tasks müssen erneut ausgeführt werden, da ihre Ergebnisse lokal auf dem Rechner des ausgefallenen Workers liegen und nicht mehr zugreifbar sind.

Fehler des Masters

Da ein Ausfall des Masters unwahrscheinlich ist, findet in Googles Implementierung keine besondere Fehlerbehandlung

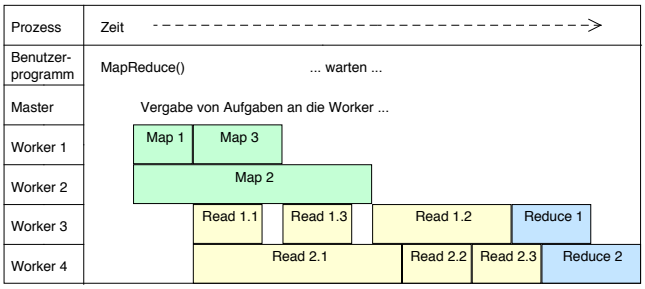

Abbildung 4: Pipelining

statt. Die gesamte Berechnung wird abgebrochen und der Benutzer hiervon in Kenntnis gesetzt. Er kann den Job dann gegebenenfalls neu starten. Eine mögliche konstruktive Fehlerbehandlung bestünde darin, den Zustand des Masters, d.h. die von ihm verwalteten Datenstrukturen, in regelmäßigen Abständen zu sichern und bei einem Ausfall einfach eine Kopie des zuletzt gesicherten Masterprogramms zu starten.

4.6 Optimierungen

Im folgenden werden einige Optimierungen beschrieben, die entweder Teil der *MapReduce*-Bibliothek sind oder vom Benutzer durch Verwendung optionaler Parameter vorgenommen werden können.

Googles verteiltes Dateisystem teilt Dateien in Blöcke auf und speichert diese redundant auf mehreren Rechnern des Clusters. Diese Tatsache kann genutzt werden, um Bandbreite im Netzwerk zu sparen. *MapReduce* versucht daher, *map*-Tasks auf Rechnern auszuführen, die eine lokale Kopie der entsprechenden Eingabedatei besitzen. Ist dies nicht möglich, wird zumindest versucht, einen Knoten mit möglichst geringer Entfernung im Netzwerk zu finden.

Ein Faktor, der die Ausführungszeit signifikant verlängern kann, sind sogenannte Nachzügler, also Rechner die überdurchschnittlich viel Zeit für die letzten verbleibenden *map*- und *reduce*-Operationen benötigen. Gründe hierfür können unter anderem defekte Festplatten mit verringerten Leseraten sein. Um das Problem zu lösen, stößt die Bibliothek gegen Ende der *MapReduce* Operation Kopien aller Tasks zur Ausführung an, die sich aktuell im Zustand *in progress* befinden. Anschließend werden die Ergebnisse all jener Task-Kopien verwendet, die zuerst fertiggestellt wurden.

Kleine Aufgabenpakete während der *map*-Phase verbessern nicht nur die Lastverteilung, sie minimieren auch die Zeit, die im Fehlerfall für die Neuausführung des *map*-Tasks benötigt wird. Abbildung 4 veranschaulicht diesen Pipelining-Effekt. Gängige Werte liegen bei Google etwa im Bereich von 200.000 *map*- sowie 5.000 *reduce*-Tasks auf einem Verbund von 2.000 Rechnern.

4.7 Erweiterungen

Standardmäßig werden die Zwischenergebnisse anhand ihrer Schlüssel in R Teile zerlegt und auf R *reduce*-Tasks verteilt. Der Benutzer kann jedoch eine eigene Partitionsfunktion für den Fall angeben, dass er zuvor eine Funktion auf die Schlüssel der Zwischenergebnisse anwenden möchte. Werden beispielsweise URLs als Schlüssel verwendet und man möchte statt einer Ausgabedatei pro URL lieber eine Datei pro Host, so kann durch die Partitionsfunktion zunächst der Hostname aus der URL extrahiert und anschließend als Schlüssel verwendet werden. *MapReduce* garantiert weiterhin, dass innerhalb einer Partition alle `(key,value)`-Paare aufsteigend nach ihren Schlüsseln verarbeitet werden. Diese Eigenschaft ist unter anderem beim Sortieren nützlich.

In bestimmten Fällen, zum Beispiel bei der Häufigkeitsverteilung von Wörtern, tritt der Schlüssel in den Ergebnispaaren der *map*-Funktion immer wieder auf. So würden im Deutschen unzählige `("und",1)`-Paare in die Ausgabedatei geschrieben und später über das Netzwerk übertragen. In solchen Fällen kann der Benutzer eine *combine*-Funktion bereitstellen, die auf die Ergebnisse der *map*-Funktion angewendet wird, bevor deren Zwischenergebnisse in lokale Dateien geschrieben werden. In der Regel erfüllen *combine* und *reduce* die gleiche Aufgabe. Der Unterschied liegt darin, dass die Ergebnisse von *reduce* in finale Ausgabedateien geschrieben, die von *combine* jedoch wie Zwischenergebnisse von *map* behandelt werden.

Weiterhin existieren einige vordefinierte Funktionen zur Verarbeitung verschiedener Eingabeformate, Mechanismen um unerwünschte oder fehlerhafte Datensätze zu überspringen, eine sequentielle Version von *MapReduce* für die lokale Ausführung und lokales *Debugging* sowie eine *counter*-Schnittstelle, durch die der Entwickler das Auftreten bestimmter Ereignisse während der Berechnung global zählen kann. Letzteres kann beispielsweise von Nutzen sein, um die Einhaltung bestimmter Toleranzen zu überwachen. Darüber hinaus beinhaltet Googles Implementierung von *MapReduce* auch einen internen HTTP-Server, der während der gesamten Operation Statusinformationen im Browser zur Verfügung stellt.

5. ANWENDUNGEN

Grundsätzlich ist *MapReduce* für nebenläufige Berechnungen über große Datenmengen auf Rechnerverbunden entwickelt worden. Die meisten Anwendungen fallen dabei in eine der folgenden drei Kategorien:

1. Tokenisierung, Indexierung und Suche
2. Erzeugung von Datenstrukturen wie z.B. Graphen
3. Data Mining und maschinelles Lernen

Dean und Ghemawat nennen in ihrer Arbeit [6] und den zugehörigen Vortragsfolien [7] zu *MapReduce* einige Anwendungsbeispiele: verteiltes Sortieren, verteiltes Suchen, die Erstellung von gewöhnlichen und umgedrehten Weblinkgraphen, *Term Vector Per Host*-Berechnungen (ein Teilgebiet bei der Erstellung von Suchmaschinen-*Rankings*), die Auswertung von Zugriffsstatistiken zu Internetseiten, invertierte Indexerstellung, *Document Clustering*, maschinelles Lernen sowie *Statistical Machine Translation*. Googles wichtigste Anwendung ist dabei zweifellos die Erstellung des Suchindexes, für die allein 24 verschiedene, hintereinander ausgeführte *MapReduce*-Operationen nötig sind. Michael Kleber erwähnt in [13] zudem Berechnungen im Zusammenhang mit Googles *PageRank*.

Auf der Projektseite des alternativen *MapReduce*-Frameworks *Hadoop* (vgl. Abschnitt 8, Implementierungen) findet sich eine Liste von Nutzern und deren *Hadoop*-Anwendungen. Die meisten Anwendungen gehören dabei zu einer der oben genannten Kategorien, am häufigsten fallen die Begriffe Web, Suche, Analyse, Statistik, Werbung, Indexierung, Graphen und Bildverarbeitung. Vieles davon steht im Zusammenhang mit Benutzern, deren Verhalten auf Webseiten und den dabei anfallenden Daten. Unter den Nutzern befinden sich etliche bekannte Unternehmen, so z.B. Amazon, Adobe, AOL, Baidu, Facebook, IBM, Microsoft und viele weitere. Die New York Times nutzte *Hadoop* 2007 beispielsweise, um 11 Millionen eingescannte Zeitungsartikel — 4 Terabyte — aus dem Zeitraum von 1851 bis 1980 in PDF-Dateien zu konvertieren. Die Operation wurde auf 100 *Amazon Elastic Compute Cloud* Instanzen durchgeführt und nahm 24 Stunden in Anspruch [10].

Die Verwendung von *MapReduce* im Zusammenhang mit Netzwerkmessdaten ist derzeit kaum dokumentiert, obwohl gerade hier große Datenmengen anfallen können und durchaus entsprechende Analyseprogramme existieren, die das Sammeln und Auswerten dieser Daten in verteilten Umgebungen unterstützen. Münz und Carle beschreiben in [15] die verteilte Netzwerkanalyse unter Verwendung der Programme *TOPAS* und *Wireshark*. Weiterhin haben Schneider et al. in [19] gezeigt, dass *packet capturing* mittels Standardhardware selbst in voll ausgelasteten 10-Gigabit-Netzwerkumgebungen möglich ist, sofern ein verteiltes System zum Einsatz kommt und der Datenverkehr auf mehrere einzelne Systeme verteilt wird. Gerade der Einsatz von Standardhardware wird jedoch seitens der *MapReduce*-Entwickler sehr hervorgehoben, da hier das Verhältnis von Nutzen zu Kosten besser sei, als beim Einsatz hochverfügbarer Spezialhardware [5].

Zumindest Kang et al. beschreiben in einem Lehrgang [20] den erfolgreichen Einsatz von *Hadoop* bei der Analyse von Netzwerkflussdaten. Primäres Ziel dabei sei, die benötigte Rechenzeit zu senken und die Fehlertoleranz der Analyse zu erhöhen. Die verwendete Hardwarearchitektur besteht aus mehreren Netzwerken, deren Router jeweils Flussdaten an Computer eines Rechnerverbunds übertragen. Auf der Softwareseite steht eine mehrschichtige Architektur, die in Abbildung 5 veranschaulicht wird. Aufbauend auf *Hadoop* kommen das verteilte Dateisystem *HDFS* und eine *MapReduce*-Bibliothek mit den obligatorischen *map/reduce*-Funktionen zum Einsatz. Eingehende Flussdaten werden im Binärformat gespeichert, von einem Flussdatenkonverter in ein Textformat umgewandelt und anschließend im *HDFS* abgelegt. Die *map*-Tasks lesen Daten zeilenweise ein, extrahieren den Zielport und die Länge der Daten und geben anschließend ein `(port,size)`-Paar zurück. Die *reduce*-Tasks lesen die nach Ports gruppierten Zwischenergebnisse ein und summieren die zugehörigen Datenlängen. Bei der Evaluierung vergleichen die Autoren die benötigte Rechenzeit ihrer Lösung mit der konventioneller Analyseprogramme und erreichen dabei eine um 72 Prozent kürzere Rechenzeit. Die Entwickler kommen außerdem zu dem Schluss, dass eine direkte Unterstützung binärer Eingabeformate seitens der Bibliothek die benötigte Rechenzeit noch weiter senken könnte und wollen daher ein entsprechendes Eingabemodul für *Hadoop* entwickeln.

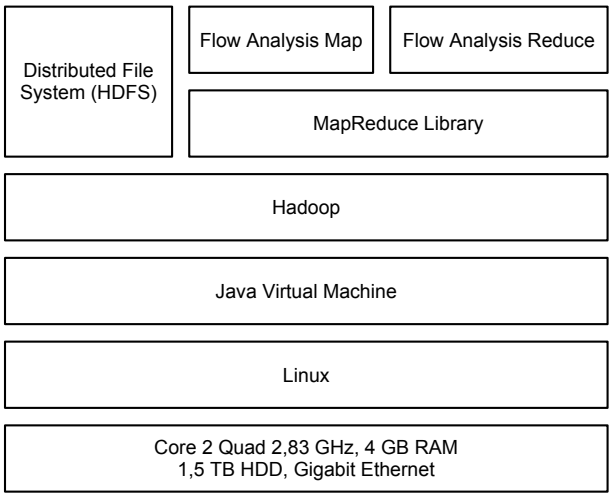

Abbildung 5: Hadoop Architektur für die Analyse von Netzflussdaten

6. ANFORDERUNGEN

MapReduce stellt keine besonderen Anforderungen an die verwendete Hardware oder die zu verarbeitenden Daten. Googles Implementierung in C++ arbeitet zwar ausschließlich mit Strings, der Konvertierung zwischen Strings und den tatsächlich benötigten Datentypen steht jedoch nichts im Wege. Auch ist der Entwickler nicht auf Dateien als Datenquelle angewiesen, durch die Implementierungen einer einfachen *reader*-Schnittstelle können Eingabedaten auch aus einer Datenbank oder aus Datenstrukturen im Speicher gelesen werden. Gleiches gilt für das Ausgabeformat. Im Zusammenhang mit Datentypen wäre lediglich zu erwähnen, dass die Ergebnisse der *map*-Funktion der gleichen Domäne wie die Ein- und Ausgaben der *reduce*-Funktion entstammen. Auch wenn die Schlüssel für das endgültige Ergebnis bedeutungslos sind, muss diese Beziehung bestehen bleiben.

7. EINSCHRÄNKUNGEN

Googles Entwickler stellten *MapReduce* 2004 auf einer Konferenz in San Francisco vor. Im Anschluss an den Vortrag wurde aus dem Publikum die Frage gestellt, welche Probleme nicht mit *MapReduce* zu lösen seien. Die Antwort darauf lautete, dass „join operations“ nicht durchgeführt werden können [17]. Michael Kleber gibt in [12] das Denkspiel *Boggle* als anschauliches Beispiel für diese Einschränkung an. Bei *Boggle* müssen Spieler in einem rechteckigen Feld mit Buchstaben möglichst viele und möglichst lange Wörter finden, wobei die Buchstaben benachbart aber nicht zwingend in einer Linie angeordnet sein müssen. Man stelle sich nun ein Spielfeld vor, das zu groß ist, um vollständig in den Speicher geladen zu werden. Um das Problem mit Hilfe von MapReduce lösen zu können, teilt man das Spielfeld in mehrere kleinere Felder auf und lässt diese durchsuchen. Wie findet man dann aber Wörter, die über die Grenzen der kleineren Felder hinausgehen? Und wie stellt man sicher, sie nur einmal zu zählen? Probleme dieser Art sind mit MapReduce nicht lösbar, da die Teilschritte während der *map*-Phase nicht unabhängig voneinander durchgeführt werden können.

8. IMPLEMENTIERUNGEN

Die bisherigen Abschnitte bezogen sich in erster Linie auf Googles Implementierung von *MapReduce*. Im folgenden werden nun einige alternative Implementierungen des *MapReduce*-Modells vorgestellt. Mit *Dryad* wird zudem eine konzeptionelle Alternative präsentiert, die zwar ähnliche Ziele, jedoch auch neuartige Ansätze verfolgt.

Das von der *Apache Software Foundation* betreute Open-Source-Framework *Hadoop* bietet unter den Namen *Hadoop MapReduce* und *Hadoop Distributed File System (HDFS)* vergleichbare Java-Implementierungen zu *MapReduce* und Googles verteiltem Dateisystem *GFS* an. Größter Unterstützer und einer der vielen namhaften Anwender von *Hadoop* ist die Firma Yahoo. Yahoo führt nach eigenen Angaben *Hadoop*-Operationen vor allem in den Bereichen Werbesysteme und Internetsuche auf mehr als 36.000 Rechnern mit 100.000 Rechenkernen durch [1].

Yahoo Pig baut auf *Hadoop* auf, bietet jedoch mit *Pig Latin* eine neuartige Abfragesprache, die die Lücke zwischen dem deklarativen Stil von SQL und dem als *low-level* empfundenen, prozeduralen Stil von *MapReduce* füllen soll. In [16] geben die *Pig*-Autoren einige Beispiele dafür an, wie stark *Pig* die benötigte Entwicklungs- und Ausführungszeit gegenüber der direkten Nutzung von *Hadoop* senken kann.

Skynet ist eine in Ruby geschriebene Implementierung des *MapReduce*-Frameworks, die als adaptives, fehlertolerantes und vollständig verteiltes System ohne *single point of failure* beworben wird. Die Open-Source-Implementierung kommt völlig ohne Master aus, stattdessen überwachen sich die Worker gegenseitig und übernehmen bei Bedarf den aktuellen Task eines anderen, ausgefallenen Workers. Zudem kann jeder Worker zu jeder Zeit in die Rolle eines koordinierenden Masters schlüpfen und wird auch in diesem Zustand von anderen Workern überwacht und gegebenenfalls ersetzt [3].

Disco ist eine von Nokia entwickelte und in Python geschriebene Implementierung von *MapReduce*. Auch *Disco* baut auf einem eigenen Dateisystem auf, dem *Disco Distributed Filesystem (DDFS)*. Mit *Discodex* wird zudem eine verteilte, auf `(key,value)`-Paaren basierte Datenbank mit entsprechenden Abfragemöglichkeiten angeboten [2].

Dryad ist eine von Microsoft entwickelte, allgemeinere Variante von *MapReduce*. *Dryad*-Anwendungen werden als gerichtete, azyklische Graphen modelliert. Der Graph bestimmt dabei den Datenfluss, die Knoten bestimmen die Operationen auf den Daten. Knotenprogramme werden streng sequentiell geschrieben und zur Ausführung automatisch auf mehrere Rechenmaschinen oder Rechenkerne verteilt. *Dryad* wurde als Plattform entwickelt, auf der weitere Schnittstellen aufbauen können, z.B. in Form von Abfragesprachen. Ein Beispiel hierfür ist die eigens entwickelte *nebula scripting language*, die einfachere, wenn auch stärker limitierte Operationen auf großen Datenmengen ermöglicht. Eine Erweiterung für den *Microsoft SQL Server* ermöglicht es zudem, SQL-Operationen im Kontext von *Dryad* durchzuführen. *Dryad* verfolgt im Grunde die gleichen Ziele wie MapReduce, soll es dem Entwickler also möglichst einfach machen, möglichst beliebige Anwendungen in verteilten Umgebungen auszuführen, ohne sich dabei um die Parallelisierung kümmern zu müssen. Die *Dryad*-Entwickler opfern jedoch nach eigenen Angaben ein Stück weit die Einfachheit von *MapReduce*, um den Entwickler im Gegenzug vom als zu starr empfundenen *map/sort/reduce*-Paradigma zu befreien [11].

9. PERFORMANZ

Google hat die Performanz von *MapReduce* in vielen experimentellen und realen Anwendungen untersucht und mitunter beeindruckende Ergebnisse erzielt. So wurde beispielsweise 2008 ein neuer Rekord im Benchmark *Terasort* aufgestellt, bei dem Daten im Umfang von einem Terabyte möglichst schnell sortiert werden müssen. Google gelang diese Aufgabe mit *MapReduce* in 68 Sekunden, während der bisherige Rekordhalter Yahoo mit 209 Sekunden dreimal langsamer war. Yahoo setzte mit *Hadoop* ebenfalls auf eine *MapReduce*-Implementierung, beide Seiten hatten etwa 1000 Server im Einsatz. Fairerweise sei jedoch gesagt, dass Google dreimal mehr Festplatten in seinen Rechnern einsetzte, was bei E/A-intensiven Operationen wie dem Sortieren entsprechende Auswirkungen hat. Die Google Entwickler gingen jedoch noch einen Schritt weiter und sortierten mit 4000 Rechnern einen Petabyte an Daten, also 1.000 Terabyte. Der Vorgang nahm sechs Stunden in Anspruch, das sortierte Ergebnis wurden auf 48.000 Festplatten gespeichert [4]. Für ausführliche Leistungsanalysen sei an dieser Stelle jedoch auf [6] verwiesen.

10. VERWANDTE ARBEITEN

Grundlage dieses Dokuments ist die 2004 erschienene Arbeit *MapReduce: Simplified Data Processing on Large Clusters* [6] sowie die zugehörigen Vortragsfolien [7] der beiden Google Entwickler Jeffrey Dean und Sanjay Ghemawat. Dean beschreibt 2006 in einem weiteren Vortrag [5] die bisher bei Google gemachten Erfahrungen mit *MapReduce*. Microsoft Entwickler Ralf Lämmel zerlegt in seiner grundlegenden Untersuchung [14] das *MapReduce*-Programmiermodell in sämtliche Einzelteile, um es dann Stück für Stück und mit starkem Bezug zu den funktionalen Sprachen wieder zusammenzusetzen.

11. ZUSAMMENFASSUNG

Diese Arbeit hat Googles *MapReduce*-Framework vorgestellt, das Programme automatisch parallelisiert und so die einfache Verarbeitung großer Datenmengen auf verteilten Systemen ermöglicht. Ausführlich wurde auf Implementierung, Programmablauf und mögliche Anwendungsgebiete eingegangen, zudem alternative Frameworks vorgestellt.

Da sich die *MapReduce*-Bibliothek um alle Details der Parallelisierung, Fehlerbehandlung, Lastverteilung und Kommunikation der Rechner untereinander kümmert, kann sie helfen den Aufwand bei der Entwicklung verteilter Anwendungen zu reduzieren. Weiterhin wurde gezeigt, dass es in bestimmten Bereichen vielfältige Anwendungsmöglichkeiten für *MapReduce* gibt. Zumindest bei der Verarbeitung und Analyse von Netzwerkmessdaten scheint das Framework jedoch noch kaum verbreitet zu sein.

Kritiker monieren, die Nutzungsmöglichkeiten seien trotz allem zu begrenzt, das Programmiermodell sei unnötig einschränkend, rückwärts gewandt und auf zu niedriger Ebene

angesiedelt. Darüber hinaus würden die *MapReduce*-Entwickler die Erfahrungen aus 40 Jahren Datenbankentwicklung ignorieren [9]. Diese Kritik mag nicht völlig unzutreffend sein, die Erfolge von *MapReduce* sind jedoch nicht zu übersehen. Erstmals wurde die Idee eines Frameworks für verteilte Anwendungen und der effiziente Ansatz zur Fehlerbehandlung durch Neuausführung einem breiten Publikum zugänglich gemacht. Nicht zu unterschätzen ist außerdem der dadurch ermöglichte Einsatz kostengünstiger Standardhardware in großen, verteilten Systemen. Vor allem aber hat *MapReduce* neben etlichen Implementierungen auch eine ganze Reihe von Weiterentwicklungen des ursprünglichen Konzepts hervorgebracht. Einige hiervon nähern sich stark dem Datenbankbereich an und können so gewisse Kritikpunkte an *MapReduce* entkräften. Andere, wie beispielsweise Microsofts *Dryad*, bringen neuartige Ansätze und bei Bedarf höhere Abstraktionsschichten für den Entwickler mit.

MapReduce war letztlich auch deshalb ein Erfolg, weil es verteilte Anwendungen bei Entwicklern populärer gemacht und ein Stück weit *demokratisiert* hat. Auf Grund der vielen aktuellen Entwicklungen in diesem Bereich, darf man gespannt sein, wie es in Zukunft weitergeht.

12. LITERATUR

[1] Apache hadoop project website. http://hadoop.apache.org/, June 2010.

[2] Disco project website. http://discoproject.org/, June 2010.

[3] Skynet project website. http://skynet.rubyforge.org/, June 2010.

[4] G. Czajkowski. Sorting 1 pb with mapreduce. http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html, November 2008.

[5] J. Dean. Experiences with mapreduce, an abstraction for large-scale computation. Proc. 15th International Conference on Parallel Architectures and Compilation Techniques, 2006.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI '04*, pages 137–150, December 2004.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. Slides from http://labs.google.com/papers/mapreduce.html, December 2004.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[9] D. J. DeWitt and M. Stonebraker. Mapreduce: A major step backwards. http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/, January 2008.

[10] D. Gottfrid. Self-service, prorated super computing fun! http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/, November 2007.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[12] M. Kleber. Mapreduce nature: Not everything is a nail. Slides from http://sites.google.com/site/mriap2008/lectures, January 2008.

[13] M. Kleber. What is mapreduce? Slides from http://sites.google.com/site/mriap2008/lectures, January 2008.

[14] R. Lämmel. Google's mapreduce programming model — revisited. *Sci. Comput. Program.*, 68(3):208–237, 2007.

[15] G. Münz and G. Carle. Distributed network analysis using topas and wireshark. In *Proceedings of IEEE Workshop on End-to-End Monitoring Techniques and Services (E2EMon 2008)*, April 2008.

[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[17] OSDI. Conference reports. http://www.usenix.org/publications/login/2005-04/openpdfs/osdi04.pdf, December 2004.

[18] H. G. Sanjay Ghemawat and S.-T. Leung. The google file system. 19th ACM Symposium on Operating Systems Principles, October 2003.

[19] F. Schneider, J. Wallerich, and A. Feldmann. Packet capture in 10-gigabit ethernet environments using contemporary commodity hardware. In *Proceedings of the 8th International Conference on Passive and Active Network Measurement*, volume 4427 of *Lecture Notes in Computer Science*, pages 207–217, New York, NY, USA, Apr. 2007. Springer-Verlag Berlin Heidelberg.

[20] Y. L. Wonchul Kang, Yeonhee Lee. Netflow analysis with mapreduce. 3rd CAIDA-WIDE-CASFI Joint Measurement Workshop, April 2010.