Konzept & Implementierung eines MapReduce Frameworks in Erlang

Filip Martinovský
FB Informatik
FH Zittau/Görlitz
safimart@stud.hs-zigr.de

Philipp Wagner
FB Informatik
FH Zittau/Görlitz
siphwagn@stud.hs-zigr.de

Abstract

Große Datensätze sind nicht länger esoterische Bedenken weniger Unternehmen, sondern werden mehr und mehr zum alltäglichen Geschäft. Die Explosion der Datenmenge stellt das Informationszeitalter vor die Aufgabe relevante Informationen aus Daten zu extrahieren. Eine sequentielle Verarbeitung dieser Datenmengen ist undenkbar und die parallele Verarbeitung von Daten erzwingt neue Programmiermodelle für Mehrkernprozessoren und Cluster. MapReduce [DG04] stellt ein Programmierparadigma und ein Framework für die parallele Berechnung dar und findet breite Anwendung in der Praxis. Dieser Beleg stellt das Konzept und die Implementierung eines MapReduce Frameworks in Erlang[1] vor. Das Framework bietet dem Anwender durch ein einfaches Interface die Möglichkeit eigene Map und Reduce Funktionen zu definieren und auszuführen. Der Aufbau eines Cluster[2] erfolgt auf Basis von Linux[3].

1 Einführung

Das Datenvolumen auf der Welt explodiert. Schon vor dem Aufkommen der sozialen Netzwerke und Plattformen wie Youtube lag im Jahr 2000 das monatliche Datenaufkommen, welches die Backbones der USA im Monat Dezember durchläuft, bei bis zu 35,000 Terabyte. [CO01] Die Jahre zuvor verdoppelte sich das Datenvolumen bereits, siehe Tabelle 1.

year	TB/(Monat=Dezember)
1990	1.0
1991	2.0
1992	4.4
1993	8.3
1994	16.3
1995	?
1996	1,500
1997	2,500 - 4,000
1998	5,000 - 8,000
1999	10,000 - 16,000
2000	20,000 - 35,000

Tab. 1. Traffic in Terabyte/Monat.

Unsere moderne Informationsgesellschafft ist hungrig nach Information. Walmart verkauft in 6,000 Filialen rund 267 Millionen Artikel pro Tag und lässt sich von HP ein Data Warehouse mit 4 Petabyte Kapazität bauen. Sloan Digital Sky Survey, welche die Bilder des New Mexiko Teleskops analysieren, generieren 200GB Bilder pro Tag und ihr letztes veröffentlichtes Dataset hatte die Größe von 10TB. Der günstige Festplattenspeicher, die Automatisierung und das Internet machen es einfach Daten zu speichern. Und diese Daten enthalten potentiell wichtige Informationen, sei es in Form von Benutzerverhalten, Markttrends, Preisstrategien oder DNA Sequenzen. Es wird ein Ozean von Daten generiert, der Durchsatz von Festplatten jedoch ist dünn. Ein Terabyte von Daten zu verschieben benötigt bei einer handelsüblichen Festplatte, wie der Seagate Barracuda bereits 3.6 Stunden, wie Tabelle 2 zu entnehmen ist.

Da selbst der Hauptspeicher zu klein ist die Daten zu halten, ist die einzige Möglichkeit diese enormen Datenmengen zu Verarbeiten die Partitionierung in berechenbare Teile und die verteilte Berechnung. Dies erzwingt neue Modelle für verteiltes Rechnen und impliziert auch neue Programmiermodelle um den Programmierer vor der Komplexität von paralleler Berechnung zu schützen.

Funktionale Sprachen bieten die Möglichkeit seiteneffektfreie Funktionen zu schreiben, die resultierenden Programme sind implizit parallelisierbar. Inspiriert vom funktionalen Modell und den beiden Funktionen höherer Ordnung *map* und *reduce* entwickelte Google das MapReduce Framework. Dabei ist das Konzept intuitiv, wie [Lä07] beschreibt: *(i) Iteration* über die Eingabedaten, *(ii) Berechnung* von Schlüssel/Wert Paaren für jeden Teil der ingabedaten, *(iii) Gruppierung* der Zwischenergebnisse nach dem Schlüssel, *(iv) Iteration* über die resultierenden Gruppen und *(v) Reduzierung* jeder Gruppe. Der Benutzer definiert zwei pure, d.h. seiteneffektfreie Funktionen als Parameter für map und reduce. Hierbei zeigen sich die Unterschiede hinsichtlich der Semantik [Lä07] zwischen dem funktionalen map und reduce und Googles $\mathcal{MAP}$ und $\mathcal{REDUCE}$. In dieser Arbeit werden nicht die semantischen Unterschiede zwischen map und $\mathcal{MAP}$ und reduce und $\mathcal{REDUCE}$ herausgearbeitet, die Bedeutung von map und reduce ist im Kontext ersichtlich.

Seit der Publikation sind neben Googles eigenem Framework viele freie Implementierungen entstanden. Im Opensource Bereich ist Hadoop, durch Yahoo!, IBM und Google unterstützt, die bekannteste Implementierung und das erste Java Programm überhaupt, welches mit dem Terasort 2008 einen Performancebenchmark gewinnen konnte. Auf der PoweredBy Seite von Hadoop[4] findet sich eine ausführliche Liste von Anwendern, wie beispielsweise Yahoo!, Facebook, A9.com (Amazon) oder Baidu. Zur Verbesserung des AdSystems und der Websuche verwendet Yahoo!, als größter Unterstützer von Hadoop, nach eigenen Angaben 25,000 Computer mit mehr als 100,000 CPUs um Hadoop Jobs auszuführen. Die soziale Plattform Facebook stellte 2008 die Hadoop-Erweiterung Hive unter Apache Lizenz und betreibt nach eigenen Angaben 600 Rechner mit 4,800 Kernen und 7 Petabyte Speicher für MapReduce Aufgaben. Die möglichen Anwendungsgebiete für MapReduce sind breit gefächtert. Google selbst beschreibt in [DG04] ein verteiltes grep, eine verteilte Sortierung, Erstellung invertierter Indexe, Dokumenten Clustering, maschinelles Lernen oder auch statistische maschinelle Übersetzung.

Ein weiteres Beispiel für den Einsatz von MapReduce ist die

[1] http://www.erlang.org

[2] Rechnerverbund

[3] http://www.knoppix.org

[4] http://wiki.apache.org/hadoop/PoweredBy

Disk	MB/s	ReadIn 1TB
Seagate Barracude	78	3.6h
Seagate Cheetah	125	2.2h

Tab. 2. Traffic in Terabyte/Monat.

verteilte Datenbank CouchDB[5]. Während dieser Belegarbeit wurde das Konzept von MapReduce durch Google Inc. zum Patent[6] angemeldet.

2 Einführung in Erlang

Erlang ist eine funktionale Programmiersprache die von Ericssons Computer Science Laboratory in den späten 1980er Jahren entwickelt wurde. Die Sprache, ähnlich wie Java von einer VM interpretiert, bringt ein Framework (*OTP*[7]) für die Entwicklung von parallelen, verteilten und fehlertoleranten Systemen mit. 1998 wurde die Sprache und die VM von Ericsson als *Open Source* zur Verfügung gestellt. Die Charakteristika von Erlang sind:

- Funktionen höherer Ordnung
- Prozesse und Message-Passing
- leichte Skalierbarkeit
- Soft Real-Time Fähigkeit

Variablen

In Erlang beginnt ein Name einer Variable immer mit einem Großbuchstaben. Werte werden mit Hilfe des Zuweisungsoperators = an Variablen gebunden. Es ist zu beachten, dass in Erlang *single assignment* gilt, das heisst eine Variable kann nur einmal an einen Wert gebunden werden:

```
1> Q = 4683.
4683
2> A = 30484.
30484
3> Q = A.
** exception error: no match of right
hand side value 30484
```

Atoms

Atoms sind durch Namen fest definierte IDs. Die Idee kommt aus der Programmiersprache *Prolog*. Atomare Datentypen müssen in Erlang immer mit einen Kleinbuchstaben beginnen oder werden in quotes geschrieben:

```
> atom1.
atom1
> 'Atom2'.
'Atom2'
> test@web.de.
'test@web.de'
```

Boolescher Datentyp

In Erlang existiert kein expliziter Datentyp für Wahrheitswerte. Wahrheitswerte werden durch die Atome `true` und `false` dargestellt.

Zahlen

Integer

Erlang bietet die Möglichkeit Integer mit verschiedenen Basen zu verwenden. Diese sind im Format `BASIS#WERT` einzugeben:

```
> -10.
-10
> 2#101010101010110.
21846
> 16#CAFEBABE.
3405691582
```

Float

Gleitkommazahlen werden gemäß dem 64-bit Format des IEEE754-1985 Standards abgespeichert (11 bit Exponent, 52 bit Mantisse):

```
> 1.2E10 - 1.2E-10.
1.2e10
> 1.231.
1.231
```

Tupel

Tupel dienen zur Speicherung einer festen Anzahl von Elementen die meist in Beziehung zueinander stehen. Ein Tupel kann unterschiedliche Datentypen enthalten und wird durch geschweifte Klammern gekennzeichnet:

```
> { 'Map', 16#BABE }.
{ 'Map', 47806 }
> tuple_size({ 'Map', 16#BABE }).
2
```

Um die Arbeit mit Tuplen zu vereinfachen gibt es in Erlang die Funktionen `element`, `tuple_size` und `setelement`.

Listen

Listen werden wie Tupel für das Abspeichern von Daten genutzt. Im Unterschied zu Tupeln haben Listen eine variable Anzahl an Elementen. Listen werden durch eckige Klammern `[]` gekennzeichnet.

```
> [ 'Map', 16#BABE ].
[ 'Map', 47806 ]
> length([ 'Map', 16#BABE ]).
2
> [ test1 | [ test2 | [] ] ].
[ test1, test2 ]
```

Pattern Matching

Erlang nutzt bei der Bindung von Variablen *Pattern-Matching* und ermöglicht es so Werte aus komplexen Datenstrukturen zu extrahieren:

```
> {person, Name, en} = {person, 'Thomas', de}.
** exception error: no match of
right hand side value {person,'Thomas',de}
>  {person, Name, de} = {person, 'Thomas', de}.
{person,'Thomas',de}
> Name.
'Thomas'
```

Es gibt auch spezielle Operatoren für das Pattern-Matching bei Listen:

```
> [Hd | Tl] = [test1, test2, test3].
> Hd.
test1
erl> Tl.
[test2, test3]
```

Funktionen

Funktionen können in Erlang ausschliesslich in Modulen definiert werden und können erst nachdem sie kompiliert wurden geladen werden. Die meisten rekursiven Funktionen nutzen das Pattern-Matching:

```
mymap(Fun,[]) ->
 [];
mymap(Fun,[Hd|Tl]) ->
 [Fun(Hd) | mymap(Fun,Tl)].
```

Dieses kurze Programm definiert die *map* Funktion. Als Funktion höherer Ordnung ruft *map* die Funktion `Fun` (eine anonyme Funktion) mit jedem Element in der Liste auf und liefert anschließend eine Ergebnisliste gleicher Kardinalität.

[5] http://couchdb.apache.org
[6] Patent Number: 7,650,331 angemeldet.
[7] Open Telecommunication Platform

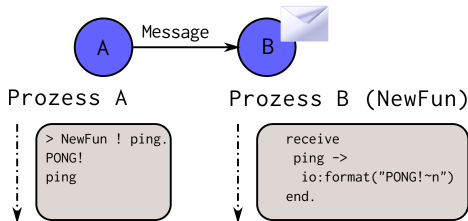


Abb. 1. Message Passing in Erlang

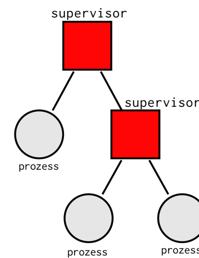


Abb. 2. Supervisor Tree in Erlang

Nebenläufigkeit in Erlang

Die große Stärke von Erlang ist die Unterstützung von Nebenläufigkeit. Jede Funktion kann mittels der Funktion `spawn` aufgerufen werden und als ein neuer Prozess in der Erlang VM fungieren. Die Erlang VM verwaltet die Prozesse unabhängig vom Betriebssystem und verteilt selbstständig die Erlang VM Threads auf Betriebssystem-Prozesse.[8] Die Erlangprozesse sind sehr leichtgewichtig, da sie innerhalb der Erlang VM lediglich Funktionen sind und so der Overhead durch Kontextwechsel und Process Control Blocks entfällt. Der Rückgabewert von `spawn` ist die *Pid*[9] des neu erstellten Prozesses.

Message Passing

Weil ein Aufruf der Funktion `spawn` asynchroner Natur ist wird ein realisiert Erlang die Kommunikationen zwischen den Prozessen mit Hilfe des Message Passing. Nachrichten, Erlang Ausdrücke, werden an Prozesse mit dem `!` Operator gesendet. Die Nachricht wird in der Mailbox des empfangenden Prozesses gespeichert und mittels Pattern-Matching innerhalb der `receive` Klausel ausgelesen.

```
> NewFun = spawn(fun()-> receive
> ping -> io:format("PONG!~n")
> end end ).
<0.48.0>
> NewFun ! ping.
PONG!
ping
```

In diesem Beispiel wird ein Prozess erstellt, der innerhalb seiner *receive* Klausel auf die Nachricht `ping` wartet und bei Erfolg die Nachricht `PONG!` herausgibt und beendet. Dies ist dargestellt in Abbildung 1.

Processlinks

Ausfallsichere und fehlertolerante Systeme lassen sich nur dann erfolgreich implementieren, falls die Möglichkeit existiert einzelne Abschnitte, Funktionen bzw. Module des Programms zu überwachen. In Erlang bieten *Monitore* und *Links* die gewünschte Funktionalität.

Ein *Link* ist dabei eine bidirektionale Verbindung zwischen zwei Prozessen. Tritt während der Ausführung eines Prozesses ein fataler Fehler auf, stürzt dieser Prozess ab und es wird eine `'EXIT'` Nachricht an alle verlinkten Prozesse gesendet. Jedem Prozess steht es anschliessend frei die Nachricht mit Hilfe des Befehls `process_flag(trap_exit, true)` abzufangen oder sie werden von der VM automatisch beendet[10].

Ein *Monitor* hingegen ist eine unidirektionale Verbindung zwischen zwei Prozessen. Im Gegensatz zu einem *Link* wird bei einem *Monitor* im Falle des Absturzes nur eine Nachricht an den beobachtenden Prozess gesendet. Dem beobachtenden Prozess wird selbst überlassen wie er diese Nachricht behandelt.

Mit Hilfe von *Monitoren* und *Links* können ganze Prozesshierarchien in sogenannte Workerprozesse und Supervisors geteilt werden, siehe Abbildung 2. Der Workerprozess führt die Berechnungen durch und der Supervisor ist lediglich für die Kontrolle der Prozesse zuständig. Im Falle eines Fehlers startet der Supervisor den Workerprozess gemäß einer *restart strategy*.

Cluster mit Erlang

Die `spawn` Funktion kann nicht nur Funktionen auf dem aktuellen Node starten, das heisst lokal auf der eigenen Erlang VM, sondern auch Funktionen auf entfernten Nodes invokieren.

```
alpha@127.0.0.1> NewFun = spawn('beta@127.0.0.1',
alpha@127.0.0.1> fun()-> receive
alpha@127.0.0.1> ping -> io:format("PONG!~n")
alpha@127.0.0.1> end end ).
<32.48.0>
alpha@127.0.0.1> NewFun ! ping.
PONG!
ping
```

Dieses Beispiel demonstriert den `spawn` Vorgang einer Funktion von einem lokalen Node *alpha@127.0.0.1* ausgehend, auf einem entfernten Node *beta@127.0.0.1*. Hinzu kommt dabei als zusätzlicher Parameter für `spawn` lediglich der Name des Nodes. Diese Eigenschaft der Sprache ermöglicht es leicht skalierbare Software zu entwickeln.

3 MapReduce

Eine abstrakte Beschreibung von MapReduce ist in [DG04] gegeben. [Lä07] arbeitet die grundlegende Abstraktion von MapReduce im Bezug zum funktionalen map und reduce heraus. Allgemein kann der Ablauf eines MapReduce Vorgangs so beschrieben werden, daß das Framework die mapper worker, reducer worker und den master worker identifiziert und startet. Die Anzahl an gespawnten Prozessen ist bedingt durch die Infrastruktur des Clusters. Es ist möglich und wünschenswert dass mapper und reducer worker entweder auf dem gleichen oder nahem Workernode arbeiten, um etwaige Zeitverzögerungen durch Netzwerkkommunikation zu vermeiden. Die Eingabedaten werden in M Teile partitioniert und von mapper workern verarbeitet. Die vom Programmierer für den map Schritt definierte Funktion nimmt <key,value> Paare als Eingabe und produziert eine neue Zwischenliste an <key,value> Paaren, in [DG04] *Intermediate Results* genannt. Die reduce Funktion hat Zugriff auf die Liste an Werten für die Schlüssel der Zwischenergebnisse, beispielsweise durch ein verteiltes Dateisystem. Sie wird einmalig für jeden Schlüssel der Zwischenliste aufgerufen und produziert eine neue Ergebnisliste, die üblicherweise aus 0 oder 1 Wert besteht.

Laut [Lä07] ist die map Funktion eine pure Funktion, so daß die Reihenfolge der Verarbeitung keine Auswirkungen auf das Ergebnis des map Schrittes hat und die keine Kommunikation zwischen

[8] http://www.defmacro.org/ramblings/concurrency.html

[9] Process identifier in der Erlang VM.

[10] Und dadurch werden wieder alle Prozesse benachrichtigt, die mit diesem Prozess verlinkt sind.

den Prozessen notwendig ist. Es können auch nicht MapReduce-konforme Komponenten einen MapReduce Vorgang beeinflussen können um Ergebnisse zwischenzuspeichern.[11]

map und reduce sind vom polymorphen Typ:

```
map      (key1,value1)     →   list(k2,v2)
reduce   (k2, list(v2))    →   list(v2)
```

Die Verteilung der Daten, eine Backup Strategie und die Verteilung der Aufgaben muss von einem Framework übernommen werden und ist teilweise schon durch ein verteiltes Dateisystem abgedeckt. Darüber hinaus muss das Framework die nötigen failover Strategien implementieren, falls ein Workernode unerwartet abbricht.

map und reduce in der Programmierung

map ist in vielen Programmiersprachen als eine Funktion höherer Ordnung verfügbar, welches eine gegebene Funktion auf eine Liste von Elementen anwendet und eine Ergebnisliste zurückgibt. Im folgenden wird zur Illustration Erlang verwendet. Um Beispielsweise eine Liste von Elementen zu quadrieren, kann map mit einer anonymen Funktion der Form $f(x) = x * x$ aufgerufen werden:

```
1> lists:map(fun(X) -> X*X end, [1, 2, 3, 4]).
[1,4,9,16]
```

fold, auch bekannt als reduce (clisp) oder accumulate (Scheme), ist eine Familie von Funktionen höherer Ordnung die eine Datenstruktur in einer gewissen Reihenfolge, entweder als left oder right fold, verarbeitet und einen Rückgabewert besitzt. fold nimmt eine combiner function und eine Datenstruktur und kombiniert die Elemente der Datenstruktur mit Hilfe der combine function.[12]

```
2> lists:foldl(fun(X,Sum)-> X + Sum end, 0,
   [1,4,9,16]).
30
```

Mit diesen Grundlagen kann bereits das in [DG04] vorgestellte Beispiel von Worthäufigkeiten von gegebenen Dokumenten, im Geschäftsbereich von Google vermutlich Webseiten, nachvollzogen werden.

Die map Funktion produziert als Zwischenergebnis eine Liste an von Wörtern mit ihrer zugehörigen Häufigkeit, in Googles Beispiel 1.

```
map(String document, String content):
  for each word w in document:
    EmitIntermediate(w, "1");
```

In Erlang gleichermaßen möglich, als ein map über eine Liste von Token. *MapData* wird an die Ergebnisliste des Map gebunden, welche die Eingabe für den Reduzierungsschritt darstellen.

```
4>MapData=lists:map(fun(W)->{W,1} end,
  string:tokens("Ich bin ein Text, ein Text.","., ")).
[{"Ich",1},
{"bin",1},
{"ein",1},
{"Text",1},
{"ein",1},
{"Text",1}]
```

Die Reduce Funktion summiert die Worthäufigkeiten der Zwischenergebnislisten.

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

[11] Cluster Computing & MapReduce: Lecture 1-4; Google Lab Tutorials

[12] Im Appendix ist dies noch einmal für den Desinteressierten Leser visualiert.

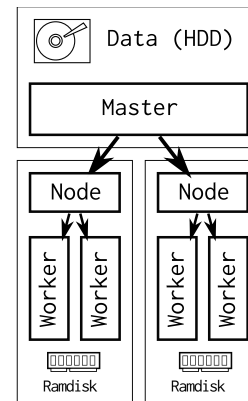


Abb. 3. MapReduce Framework des Belegs

Es ist zu erkennen, daß bei Google ein Schlüssel übergeben wird und ein Iterator von Werten. Da es vorkommen kann, daß Zwischenergebnisse größer als der verfügbare Hauptspeicher sind, wurde der Eingabeparameter der Liste von Zwischenwerten als Iterator implementiert.

Die Reduce Funktion ist etwas komplizierter, da an dieser Stelle des Belegs kein Framework vorhanden ist um die Ergebnisse des Reduzierungsschrittes zu verwalten.

```
5> Reduce = fun({Key, Val}, Dict) ->
  case lists:keymember(Key, 1, Dict) of
    true ->
      { Key, OldVal } = lists:keyfind(Key, 1, Dict),
        lists:keyreplace(Key, 1, Dict,
          { Key, OldVal + Val});
    false ->
      lists:keystore(Key, 1, Dict, { Key, Val })
        end end.
#Fun<erl_eval.12.113037538>
```

Die combiner function ist nun an die Variable *Reduce* gebunden. Diese Funktion nimmt 2 Parameter: ein Tupel bestehend aus key, value und ein assoziatives Array in Form eines Dictionary. *Reduce* (combiner function) und *MapData* (intermediate results) sind die Eingabe für das abschliessende left fold:

```
6>lists:foldl(Reduce, [], MapData).
[{"Ich",1},{"bin",1},{"ein",2},{"Text",2}]
```

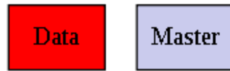
In einem verteilten System würden die worker nodes ihre Reduce Ergebnisse an die übergeordnete Instanz zurückgeben und die Dictionaries werden zusammengeführt. Es ist in diesen Beispielen ist zu erkennen, daß die Mapping Phase und die Reduce Phase komplett isoliert für mehrere Teilprobleme evaluiert werden können, ohne das Ergebnis der Berechnung zu verfälschen.

4 Implementierung

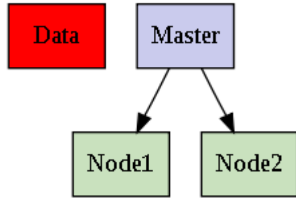
4.1 MapReduce Framework im Beleg

Für den Beleg wurde aus Zeitgründen ein einfaches Framework ohne verteiltes Dateisystem und Fehlertoleranz implementiert, Abbildung 3. Es wird ein gesamter MapReduce Vorgang durchgearbeitet, diese Schritte können auch in den Modulen nachvollzogen werden.

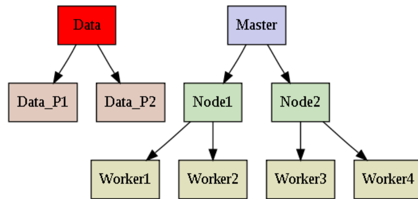
1. Starten des Master Node, als Parameter erhält dieser die im Netzwerk verfügbaren Nodes. Im Quellcode ist dies im Modul mr zu finden.



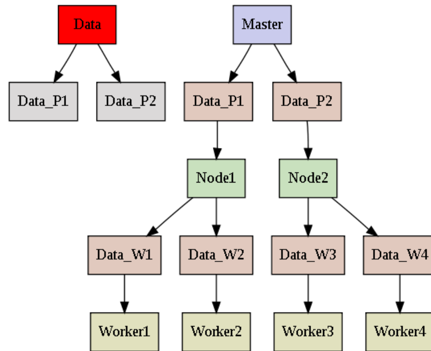
2. Master verbindet sich zu den Nodes.



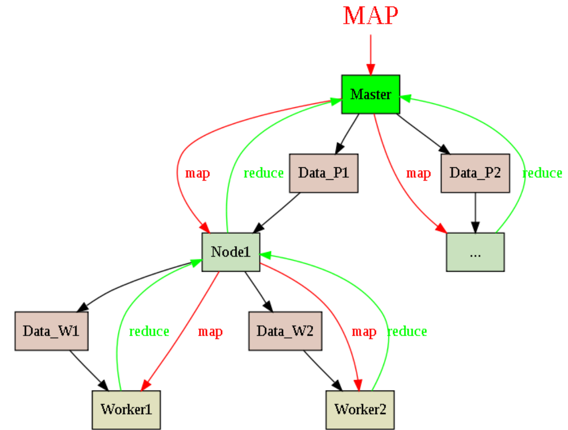
3. Die Eingabedaten werden in die Anzahl an Nodes *#NumNodes* partitioniert. Im Quellcode *Modul mr*.



4. Die Eingabedaten werden im Cluster verteilt, jeder Node teilt seine Daten in die Anzahl an WorkerNodes *#NumWorker*. Diese kann beispielsweise die Anzahl an verfübaren CPUs auf dem Node sein. Im Quellcode im *Modul map*.



5. Der MapReduce Schritt wird angestossen (*Modul mr*), wenn der Master den MapReduce Befehl erhält. Jeder Node erhält die Anweisung die Daten zu mappen (*Modul node*). Anschliessend werden die mit reduce Funktion reduzierten Ergebnisse (*Modul node*) an die übergeordnete Instanz zurückzugeben.



6. Im letzten Schritt erhält der Master Node die von den Instanzen reduzierten Ergebnisse.

5 Rechnerverbund

Die LiveCD basiert auf Knoppix[13], welches seinerseits aus einer Mischung zwischen Debian *unstable* und *testing*[14] basiert. Knoppix ist als Live-System entwickelt, welches sich direkt von CD, DVD oder Flashdisk starten lässt und in Version 6.2 einen aktuellen 2.6.31 Linux Kernel mitbringt. Erlang/OTP kann entweder mit dem Paketmanager apt oder mit Hilfe von Quellen zur LiveCD hinzugefügt werden. Sollte kein DHCP-Server für die Verteilung der Netzwerkadressen zur Verfügung stehen, ist es darüber hinaus nötig die IP-Adressen manuell zu setzen.

6 Experimente

Experimente wurden in einem Cluster von 18 handelsüblichen Computern durchgeführt. Jeder Computer verfügt über einen Intel®Xeon®E5520 (2.26Ghz), welcher 8 logische CPUs abstrahiert, mit 8Mb Cache, 6144MB Arbeitsspeicher und 250GB SATAII Festplatten. Die Computer sind über Gigabit Ethernet verbunden, der Chip ist ein Realtek RTL8111 und der Switch ein Cisco Systems Catalyst 2960-48-TC. Die Rechner wurden während der Experimentalphase ausschliesslich für die Experimente verwendet und betrieb ein angepasstes, also mit Erlang versehenes, Knoppix 6.2.

Es müssen zwei Dinge angemerkt werden:

1. Es war keine Installation von Linux auf dem Cluster erlaubt. Der Festplattenspeicher wird somit nicht in die Betrachtung einbezogen.

2. Vorerst wird nur eine 32bit Knoppix Distribution verwendet. Da in dem Kernel nicht die bigmem Erweiterung kompiliert ist, ist der maximal adressierbare Speicher auf 3072MB limitiert.

6.1 Experiment1: Summierung

6.1.1 MapReduce

Um die lineare Skalierbarkeit des entstandenen Erlangclusters zu testen wurde eine Summierung von Daten als reduce Funktion implementiert, der map Schritt ist lediglich das Erstellen von Zufallszahlen.

6.1.2 Experimentkonfiguration

Da im Experimentalaufbau kein Linux auf den PCs installiert werden konnte und auch keine Partition auf der Festplatte zur

[13] http://www.knoppix.net
[14] http://www.debian.org

Verfügung gestellt wurde, ist die obere Grenze des Experimentdatensatzes durch die Größe der *ramdisk* definiert. Bei dem verwendeten Knoppix beträgt die Größe der *ramdisk* 1GB. Da das Betriebssystem sich komplett in die *ramdisk* lädt und somit wiederum Speicher alloziert wird, ergab sich für diesen Test pro involvierter Einheit die obere Grenze von rund 700MB künstlich erzeugter Werte.

Der homogene Cluster enthält somit für jede Einheit 80 Dateien der Größe 9.2Mb. Für den initialen Schritt eines Erlangclusters von 4 PCs sind somit 320 Dateien und 2.875GB Daten im Cluster verteilt. Da diese Dateigröße zu klein ist wurde entschieden, dass jeder Workernode die gesamten, dem Node verfügbaren, Daten einliest. Für 4 Nodes mit 8 Cores ergibt sich somit die zu verarbeitende Datenmenge von $4*8*80*9.2MB = 23,552MB = 23GB$.

6.1.3 Messung

Mit Hilfe des *timer* Moduls kann eine Erlang Funktion hinsichtlich ihrer Lauzeit untersucht werden. Die Methode *timer:tc(Modul, Funktion, Parameter)* gibt die verwendete CPU Zeit als Tupel `{microseconds, value}` zurück.

6.1.4 Ergebnisse

Die Ergebnisse zeigen deutlich die lineare Skalierbarkeit des Clusters:

```
%%% 4 Nodes a 8 Core %%%
(master@141.46.202.75)23> timer:tc(mr,map, [320]).
{49933174,[32008393175872]}

%%% 8 Nodes a 8 Core %%%
(master@141.46.202.75)6> timer:tc(mr, map, [640]).
{25105170,[32008393175872]}

%%% 16 Nodes a 8 Core %%%
(master@141.46.202.75)6> timer:tc(mr, map, [1280]).
{12647182, [32008393175872]}
```

Während für die initiale Berechnung mit 4 Nodes rund 50 Sekunden zur Verarbeitung benötigt wurden, verwendeten 8 Nodes das 0.503-fache der Zeit und 16 Nodes nur noch das 0.253-fache. Der Cluster skaliert für das gegebene Problem somit superlinear.

7 MapReduce Beispiele

7.1 K Nearest Neighbor

Die Methode der K nächsten Nachbarn klassifiziert ein Element aus einer Menge von Anfragen L_q gegen die Elemente aus der Trainingsmenge L_t. Das theoretische Fundament von k-nächsten Nachbarn ist einer reichhaltigen Literatur zu entnehmen.[WF99]

Die in [MPF09] vorgeschlagene Version eines parallelen K nächsten Nachbarn stellt sich als eine Folge von 2 Mapschritten dar: Die erste Map Funktion erstellt für jede Anfrage q eine Ergebnisliste von Tupeln (q,t,r) der Kardinalität $|L_t|$, mit $t \in L_t$ und $r = dist(q,t)$.[15] Der zweite Map Vorgang gibt die k Nächsten Nachbarn zu q zurück.[16]

```
L_q ← List of Queries (Testset)
L_t ← List of Training Instances (Trainingset)
L_d ← lists:map(fun->(Q) ->
        lists:map(fun(T)-> {q,t, distance(q,t)} end, L_t)
        end, L_q.
L_k ←
       lists:map(fun(q) ->
           lists:foldl(fun(q, dist) ->
              findKNN(q,dist) end, [], L_d) end, Query)
       return L_k
```

[15] Um Herrn Martinovský glücklich zu machen, alternativ: $[q,t,distance(q,t)||q \rightarrow L_q, t \rightarrow L_t]$

[16] Um Herrn Martinovsky ein zweites Mal glücklich zu machen: $lists : foldl(fun(Dist,Results) - > findKNN(Dist,Results) end, [], Distances)$

Dieser Algorithmus sollte mit steigender Anzahl an verfügbaren Prozessoren nahezu linear skalieren, da die Schritte in Isolation voneinander ausgeführt werden können und keine Kommunikation zwischen den Prozessen notwendig ist. Die Ergebnisse in [MPF09] belegen dies.

7.2 Google Beispiele

Im folgenden sollen noch einmal ausgewählte Beispiele aus [DG04] erläutert werden.

Verteiltes Grep

Die Map Funktion emmitiert nur eine Zeile falls sie gegen ein gesuchtes Pattern validiert. Die Reduzierungsfunktion ist lediglich die Identität.

Invertierter Index

Die Map Funktion liest alle gegebenen Dokumente ein und erzeugt eine Zwischenergebnisliste mit $\langle word, documentid \rangle$ Paaren. Die Reduce Funktion sortiert für ein gegebenes Wort die Dokumenten Ids, und gibt eine Liste mit $\langle word, list(documentid) \rangle$ Paaren zurück. Die Menge an Ergebnispaaren ergeben einen einfachen invertierten Index.

8 Verwandte Arbeiten

[Lä07] arbeitet die formalen Grundlagen von MapReduce heraus, besonders im Bezug auf die aus der funktionalen Programmierung bekannten Funktionen höherer Ordnung *map* und *reduce*.

[Chu06] untersucht den Aspekt des maschinellen Lernens auf Multicore Architekturen. Sie untersuchen k-means, logistische Regression (LR), Naive Bayes (NB), lokal gewichtete lineare Regression (LWLR), Gaussche Diskriminative Analyse (GDA), Neuronale Netze (NN), Principal Component Analysis (PCA), Expectation Maximazation (EM) und Support Vector Maschinen (SVM). Neben der Zeitkomplexität der jeweiligen Algorithmen erarbeiten sie auf Basis von MapReduce ein generisches Framework für die Ausführung von Multicore Berechnungen. Es ist zu erwähnen, daß die bekannten Algorithmen des maschinellen Lernens gut skalieren. [Chu06]: "*NN speedup was [16 cores, 15.5x], [32 cores, 29x], [64 cores, 54x]. LR speedup was [16 cores, 15x], [32 cores, 29.5x], [64 cores, 53x]*".

In [MPF09] wird versucht kanonische Schleifen automatisch zu parallelisieren und als Beispiele werden k-means und die k nächste Nachbarn Methode verwendet. Sie stellen zwar nicht explizit ihr verwendetes Dataset vor, berichten jedoch von guter Skalierbarkeit der k nächster Nachbarn Methode. Geschuldet ist dies dem Umstand, daß die Semantik der Methode eine Gleichverteilung der Eingabedaten erlaubt.

[JLR$^+$08] beschreibt die Implementierung eines Schemeinterfaces für MapReduce Aufgaben. Während die eigentlichen MapReduce Jobs in Hadoop erfolgen, entwickeln sie eine JNI-Schnittstelle zwischen beiden Technologien. Das Schemeinterface verwenden sie für Einsteigerkurse der parallelen Datenverarbeitung an der Universität von Kalifornien in Berkley. Die ersten Ergebnisse der Performance zeigen jedoch, daß ein Scheme Programm durchschnittlich 10 mal langsamer als ein äquivalentes Java Programm ist. Die Autoren führen dies auf ein Problem im I/O Handling der clusterseitigen Software zurück und geloben Besserung.

9 Fazit

In diesem Beleg wurden die theoretischen Grundlagen von MapReduce aufgearbeitet und anhand eines Frameworks für die Programmiersprache Erlang demonstriert. Es konnte gezeigt werden, dass ein auf Linux basierender Erlangcluster linear skaliert, insofern die Map und Reduce Funktion dies zulassen. Die Belegarbeit kann somit als Erfolg angesehen werden.

Im Bezug auf Datamining sind die relevanten verwandten Arbeiten herausgearbeitet, welche sich mit der Skalierbarkeit der

wichtigsten Algorithmen des maschinellen Lernens beschäftigen. [Chu06], [MPF09]

Ein weiterer Beitrag dieser Arbeit ist die erstmalige Darstellung von MapReduce als Endlicher Automat, welche in der folgenden Version des Frameworks einen ausfallsicheren und fehlertoleranten Erlangcluster zur Verfügung stellen wird.

A map(...) und reduce(...)

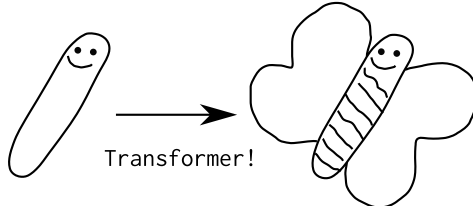
A.1 In C++

Die traditionellen map und reduce Funktionen sind keinesfalls ausschliesslich der funktionalen Programmierung zugänglich, dies soll an keiner Stelle des Beleges die Überlegenheit von funktionalen Sprachen andeuten. Auch in Sprachen wie C++, D, JavaScript, Perl, PHP, Groovy und C# ist es möglich Map und Fold Funktionen als Funktionen höherer Ordnung zu verwenden. In C++ stellen sich die Funktionen dar als:

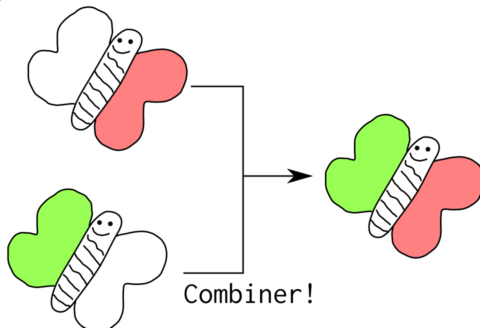
- **Map**: *begin*, *end*, *result* im Header <*algorithm*> definierte Iteratoren und *func* ein Funktionszeiger
  - `Map` - std::transform(*begin, end, result, func*)
  - `Map 2 Lists` - std::transform(*begin1, end1, begin2, result, func*)
- **Reduce** (als **Fold**): *begin*, *end*, *rbegin*, *rend* im Header <*numeric*> definierte Iteratoren und *func* ein Funktionszeiger.
  - `Left Fold` - std::accumulate(*begin, end, initval, func*)
  - `Right Fold` - std::accumulate(*rbegin, rend, initval, func*)

A.2 Visualisiert

map



reduce



B Literatur

[Chu06] CHU, Cheng-Tao: Map-Reduce for Machine Learning on Multicore / CS. Department, Stanford University. 2006. – Forschungsbericht

[CO01] COFFMAN, K. G. ; ODLYZKO, A. M.: Internet growth: Is there a "Moore's Law" for data traffic? / AT&T Labs - Research. 2001. – Forschungsbericht

[DG04] DEAN, Jeffrey ; GHEMAWAT, Sanjay: MapReduce: Simplified Data Processing on Large Clusters / Google, Inc. 2004. – Forschungsbericht. – Google: http://labs.google.com/papers/mapreduce-osdi04.pdf

[JLR+08] JOHNSON, Matthew ; LIAO, Robert H. ; RASMUSSEN, Alexander ; SRIDHARAN, Ramesh ; GARCIA, Dan ; HARVEY, Brian K.: Infusing Parallelism into Introductory Computer Science Curriculum using MapReduce / EECS Department, University of California, Berkeley. Version: Apr 2008. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-34.html. 2008 (UCB/EECS-2008-34). – Forschungsbericht

[Lä07] LÄMMEL, Ralf: Google's MapReduce Programming Model — Revisited / Data Programmability Team, Microsoft Corp. 2007. – Forschungsbericht

[MPF09] MATA, Leonardo L. P. ; PEREIRA, Fernando Magno Q. ; FERREIRA, Renato: Automatic Parallelization of Canonical Loops / Universidade Federal de Minas Gerais. 2009. – Forschungsbericht

[WF99] WITTEN, Ian H. ; FRANK, Eibe: *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999 http://www.amazon.de/Data-Mining-Techniques-Implementation-Management/dp/1558605525%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D1558605525. – ISBN 1558605525