University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Prototypical Implementation of ...

Author
City, Country
Student ID: 00-711-999

Supervisor: ...
Date of Submission: January 1, 2006

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Das ist die Kurzfassung...

Acknowledgments

Optional

Contents

Abstract **i**

Acknowledgments **iii**

1 Introduction **1**

1.1 Motivation 1

1.2 Description of Work 1

1.3 Thesis Outline 1

2 Related Work **3**

3 Methods and Prototype **5**

3.1 Methods used 5

3.1.1 MapReduce 5

3.2 Prototype 6

4 Evaluation **9**

5 Summary and Conclusions **11**

Bibliography **13**

Abbreviations **15**

Glossary **17**

List of Figures **17**

List of Tables **19**

A Installation Guidelines **23**

B Contents of the CD **25**

Chapter 1

Introduction

1.1 Motivation

1.2 Description of Work

1.3 Thesis Outline

Chapter 2

Related Work

Although the widest usage of the MapReduce programming model to date may be the one of Hadoop [1] with its centralised master-slave architecture, recently, there have also been a number of attempts to transport it to a more decentralised setting in an endeavour to support currently popular Cloud platforms, pervasive grids, and/or mobile environments.

[3] recognised the problem of centralised master-slave architectures to not cope well with dynamic Cloud infrastructures, where nodes may join or leave the network at high rates. Thus, they introduce a peer-to-peer model to manage node churn but also master failures and job recovery in a decentralised way. Each node may become a master or a slave at any given time dynamically, preserving a certain master/slave ratio. Slaves are assigned tasks to perform by the masters, which handle management, recovery, and coordination. To reduce job loss in case of master failures, each master may act as a backup master for a certain job, only executing it if the master primarily responsible for that job fails. Overall, the structure of different entities resembles very much the one of Hadoop, simply ported to a P2P setting. They were able to show that such an implementation provides better fault tolerance levels compared to a centralised implementations of MapReduce and only limited impact on network overhead.

In a very similar direction goes the idea of CloudFIT [4], which the authors advertise as a "Platform-as-a-Service (PaaS) middleware that allows the creation of private clouds over pervasive environments". The idea is to use private computers to set up a private "Cloud", and that MapReduce jobs are then distributed to this environment and executed in a P2P fashion. CloudFIT is, thus, built to support various P2P overlays and the authors show the performance of CloudFIT with both PAST and TomP2P against Hadoop. The results demonstrated CloudFIT to be able to achieve similar execution speeds as Hadoop while omitting the need of a dedicated cluster of computers. Data in CloudFIT is directly stored within the DHT of the corresponding overlay, such that it is replicated by a factor of k. Although such an implementation may not guarantee n-resiliency (meaning, replicating the data on each node), it is a reasonable assumption that fault tolerance may be ensured more than enough (provided at least a number of nodes stay alive) while drastically improving storage performance. Furthermore, not all the data is broadcasted but only the keys of each task, further reducing the network usage the authors showed in another

study [5] to be one of the main problems in distributed environments for the performance of MapReduce tasks.

Chapter 3

Methods and Prototype

In this section, the used methods are laid out and the implemented prototype is described.

3.1 Methods used

3.1.1 MapReduce

At the very core of this thesis lies the idea of MapReduce, a programming model made popular by Google and presented in [2]. Having large data sets to analyse that may not be handled by only few computers, like the millions of Websites gathered by Google each day that have to be indexed fast and reliably, MapReduce provides a way of automatic parallelisation and distribution of large-scale computations. Users only need to define two types of functions: a map and a reduce function. Each computation expressed by these functions takes a set of input key/value pairs and produces a set of output key/value pairs. A map function takes an input pair and produces a set of *intermediate* key/value pairs. Once the intermediate values for each intermediate key I are grouped together, they are passed to the reduce function. The reduce function then takes an intermediate key I and the corresponding set of intermediate values (usually supplied as an iterator to the reduce function) and merges them according to user-specified code into a possibly smaller set of values. The resulting key and value are then written to an output file, allowing to handle lists of values that are too large to fit in memory.

A very simple yet often used MapReduce operation is to count words in a collection of documents, the so called *WordCount*. Below pseudocode demonstrates how one can count all the words in documents using map and reduce functions.

```
function MAP(String key, String value)
    // key: document name, value: document content
    for word w in values do
        EmitIntermediate(w, "1");
    end for
end function
```

```
function REDUCE(String key, Iterator values)
    // key: a word, values: a list of "1"'s
    int result = 0;
    for v in values do
        result += ParseInt(v);
    end for
    Emit(AsString(result));
end function
```

The map function splits the received text (*value*) into all corresponding words and for each word, emits its occurrence count (simply a 1). The reduce function then takes, for each word individually (*key*), these occurrences (*values*) and sums them up, eventually emitting for each word the sum of occurrences (*result*). Although above pseudocode suggests inputs and outputs to be strings, the user specifies the associated types. This means that input keys and values may be of a different domain than the output keys and values (note that the *intermediate* keys and values are from the same domain as the output keys and values). Conceptually, this is expressed as follows:

map	(k1, v1)	→ list(k2, v2)
reduce	(k2, list(v2))	→ list(v2)

The traditional way to view the workflow in a MapReduce program is indicated by Figure 3.1. A master schedules execution and assigns tasks to workers, which then perform the actual map or reduce on the assigned data. Input data is partitioned into M splits with a size of around 16-64 MB per piece to be processed in parallel by different machines, and the intermediate key space is partitioned into R pieces, where R usually is the number of available workers for executing reduce but may also be specified by the user. To evenly distribute these pieces, a partitioning function like *hash(key) mod R* may be used. In such a setting, same intermediate keys may not be grouped together, which is done before the reduce action is performed. Additional combination in between may be applied to reduce the amount of data to be distributed among the workers.

3.2 Prototype

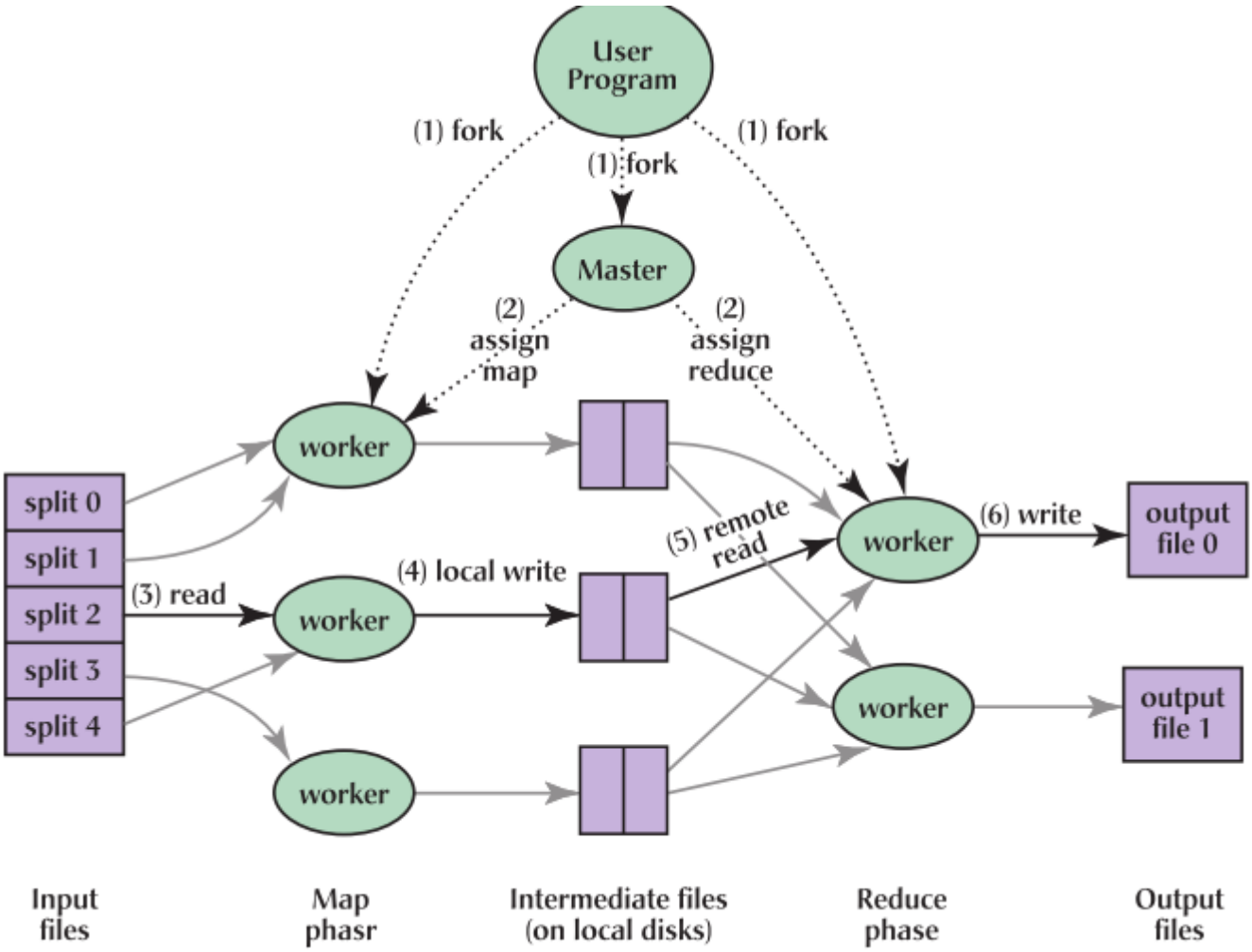

Figure 3.1: Execution overview in a typical MapReduce program

Chapter 4

Evaluation

Chapter 5

Summary and Conclusions

Bibliography

[1] Autoren: Titel, Verlag, `http://...`, Datum.

Abbreviations

AAA	Authentication, Authorization, and Accounting

Glossary

Authentication

Authorization Authorization is the decision whether an entity is allowed to perform a particular action or not, e.g. whether a user is allowed to attach to a network or not.

Accounting

List of Figures

3.1 Execution overview in a typical MapReduce program 7

List of Tables

Appendix A

Installation Guidelines

Appendix B

Contents of the CD

Bibliography

[1] Welcome to Apache Hadoop!

[2] Jeffrey Dean and Sanjay Ghemawat. MapReduce. *Communications of the ACM*, 51(1):107, jan 2008.

[3] Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. P2P-MapReduce: Parallel data processing in dynamic Cloud environments. *Journal of Computer and System Sciences*, 78(5):1382–1402, sep 2012.

[4] Luiz Angelo Steffenel and Manuele Kirch Pinheiro. CloudFIT , a PaaS platform for IoT applications over Pervasive Networks. 2015.

[5] Luiz Angelo Steffenel and Manuele Kirch Pinheiro. Leveraging Data Intensive Applications on a Pervasive Computing Platform: The Case of MapReduce. *Procedia Computer Science*, 52:1034–1039, 2015.