PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
COMPUTER SCIENCE GRADUATE PROGRAM

A UNIFIED MAPREDUCE PROGRAMMING INTERFACE FOR MULTI-CORE AND DISTRIBUTED ARCHITECTURES

DANIEL COUTO ADORNES

Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fullfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Luiz Gustavo Leão Fernandes

Porto Alegre
2015

Dados Internacionais de Catalogação na Publicação (CIP)

A241u Adornes, Daniel Couto
A unified mapreduce programming interface for multi-core and distributed architectures / Daniel Couto Adornes. – Porto Alegre, 2015.
139 p.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Luiz Gustavo Leão Fernandes.

1. Informática. 2. Processamento Distribuído. 3. Processamento Paralelo. 4. MapReduce. 5. DSL. 6. Memória Compartilhada. I. Fernandes, Luiz Gustavo Leão. II. Título.

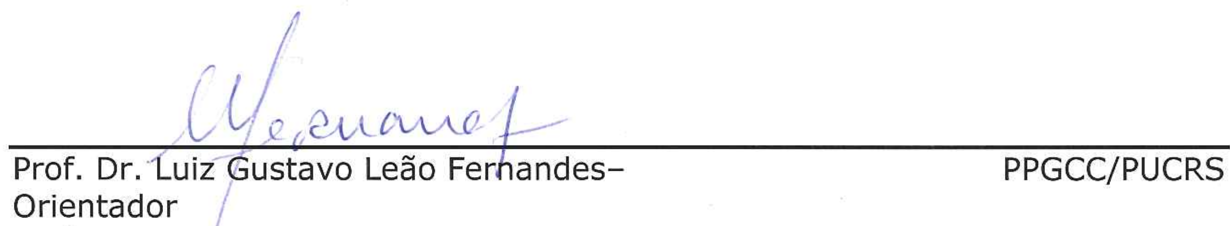
CDD 005.4

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "*A Unified Mapreduce Programming Interface for Multi-Core and Distributed Architectures*" apresentada por Daniel Couto Adornes como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, aprovada em 31/03/2015 pela Comissão Examinadora:

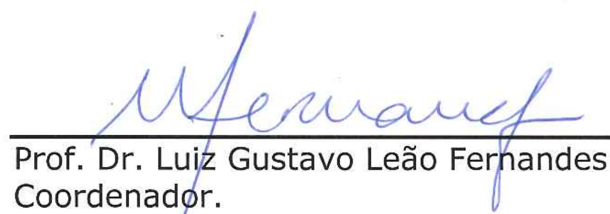


Prof. Dr. Luiz Gustavo Leão Fernandes– Orientador — PPGCC/PUCRS

Prof. Dr. César Augusto Fonticielha De Rose– — PPGCC/PUCRS

Prof. Dr. Rodrigo da Rosa Righi– — UNISINOS

Homologada em 18/12/2015, conforme Ata No. 023 pela Comissão Coordenadora.



Prof. Dr. Luiz Gustavo Leão Fernandes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320–3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

This work is dedicated to my precious wife Juliana and my lovely son Pietro.

"Blessed are those who find wisdom, those who gain understanding, for she is more profitable than silver and yields better returns than gold. She is more precious than rubies; nothing you desire can compare with her. Long life is in her right hand; in her left hand are riches and honor. Her ways are pleasant ways, and all her paths are peace."
(Proverbs 3:13-17)

ACKNOWLEDGMENTS

I thank God, who takes my charges and keeps being faithful and giving me His peace, which exceeds all understanding. To my dear wife, always understanding and supporting this important stage of my life and career, despite many difficult moments. To my lovely son, who still not even talks, but gives me very special moments with every little thing he does. And to my parents, who always taught me to strive for excellence in everything I do.

I thank my advisor professor Luiz Gustavo for advising me along each decision and step to build this work, my research partner Dalvan Griebler who contributed almost as much as an advisor, the Pontifícia Universidade Católica do Rio Grande do Sul and Faculdade de Informática for accepting me in the program and providing such a great structure, team and equipments, and CAPES for providing me with financial support for the whole program.

UMA INTERFACE DE PROGRAMAÇÃO MAPREDUCE UNIFICADA PARA ARQUITETURAS MULTI-CORE E DISTRIBUÍDA

RESUMO

Visando melhoria de performance, simplicidade e escalabilidade no processamento de dados amplos, o Google propôs o padrão paralelo MapReduce. Este padrão tem sido implementado de variadas formas para diferentes níveis de arquitetura, alcançando resultados significativos com respeito a computação de alto desempenho. No entanto, desenvolver código otimizado com tais soluções requer conhecimento especializado na interface e na linguagem de programação de cada solução. Recentemente, a DSL-POPP foi proposta como uma solução de linguagem de programação de alto nível para programação paralela orientada a padrões, visando abstrair as complexidades envolvidas em programação paralela e distribuída. Inspirado na DSL-POPP, este trabalho propõe a implementação de uma interface unificada de programação MapReduce com regras para transformação de código para soluções otimizadas para arquiteturas multi-core de memória compartilhada e distribuída. A avaliação demonstra que a interface proposta é capaz de evitar perdas de performance, enquanto alcança uma redução de código e esforço de programação de 41,84% a 96,48%. Ademais, a construção do gerador de código, a compatibilidade com outras soluções MapReduce e a extensão da DSL-POPP com o padrão MapReduce são propostas para trabalhos futuros.

Palavras-Chave: mapreduce, dsl, shared-memory, multi-core, parallel, distributed.

A UNIFIED MAPREDUCE PROGRAMMING INTERFACE FOR MULTI-CORE AND DISTRIBUTED ARCHITECTURES

ABSTRACT

In order to improve performance, simplicity and scalability of large datasets processing, Google proposed the MapReduce parallel pattern. This pattern has been implemented in several ways for different architectural levels, achieving significant results for high performance computing. However, developing optimized code with those solutions requires specialized knowledge in each framework's interface and programming language. Recently, the DSL-POPP was proposed as a framework with a high-level language for patterns-oriented parallel programming, aimed at abstracting complexities of parallel and distributed code. Inspired on DSL-POPP, this work proposes the implementation of a unified MapReduce programming interface with rules for code transformation to optimized solutions for shared-memory multi-core and distributed architectures. The evaluation demonstrates that the proposed interface is able to avoid performance losses, while also achieving a code and a development cost reduction from 41.84% to 96.48%. Moreover, the construction of the code generator, the compatibility with other MapReduce solutions and the extension of DSL-POPP with the MapReduce pattern are proposed as future work.

Keywords: mapreduce, dsl, shared-memory, multi-core, parallel, distributed.

LIST OF FIGURES

2.1 MapReduce job execution flow [DG08] 26
2.2 Key/value pairs distribution [Ven09] 27
2.3 Shuffle and Sort steps with Hadoop [Whi09] 30
2.4 The basic data flow for the Phoenix runtime [RRP+07] 34
2.5 Phoenix data structure for intermediate key/value pairs 34
2.6 Execution Flow of Tiled-MapReduce [CCZ10] 38
2.7 NUCA/NUMA-aware mechanism [CCZ10] 40
2.8 Task Parallelism through pipeline [CCZ10] 41
2.9 Experiment of 1 GB *word count* using Phoenix++ and Hadoop on a multi-core architecture. The y-axis is in a logarithmic scale. [CSW13] 45
2.10 Hybrid structure with mixed patterns [GF13] 46
3.1 The relationship graph between abstraction and performance on the programming interface of analyzed researches. 51
4.1 Transformation Process 58
4.2 Compilation flow 69
5.1 Mean execution time in seconds for original and generated Phoenix++ code 75
5.2 Mean execution time in seconds for original and generated Hadoop code . . 75
5.3 SLOC count for the interface version without curly braces 77
5.4 Cost estimate for the interface version without curly braces 78
5.5 SLOC count for the interface version with curly braces 78
5.6 Cost estimate for the interface version with curly braces 78
B.1 General Purpose GPU Architecture. (SP: Streaming Multi-Processor) [BK13] 136

LIST OF TABLES

2.1 Phoenix API functions 32
2.2 Phoenix scheduler attributes 33
2.3 Key distributions and optimized *data containers* proposed by Phoenix++ [TYK11] 42
3.1 Overview of Related Work 51
4.1 Java *imports* and C++ *includes* always required by MapReduce applications 60
4.2 Variable types, code transformation and additional *includes/imports* 61
4.3 Built-in functions, code transformation and additional C++ *includes* 62
4.4 Built-in functions for emit operations 62
4.5 Code transformation rules for indirect replacements. 64
5.1 Software category as suggested by COCOMO model 73
5.2 Suggested parameters according to COCOMO model and software category 73
5.3 Mean execution time in seconds for original and generated Phoenix++ code 74
5.4 Mean execution time in seconds for original and generated Hadoop code . . 75
5.5 SLOC count for the version without curly braces 76
5.6 Cost estimate for the version without curly braces 76
5.7 SLOC count for the version with curly braces 77
5.8 Cost estimate for the version with curly braces 77
B.1 Comparisons of the GPU-based MapReduce frameworks [BK13] 137

CONTENTS

1 INTRODUCTION . . . **23**

2 BACKGROUND . . . **25**
2.1 MAPREDUCE . . . 25
2.2 AN OVERVIEW OF MAPREDUCE IMPLEMENTATIONS . . . 27
2.3 HADOOP MAPREDUCE . . . 29
2.4 PHOENIX . . . 31
2.5 PHOENIX 2 . . . 35
2.6 TILED-MAPREDUCE . . . 37
2.7 PHOENIX++ . . . 41
2.7.1 LIMITATIONS OF PREVIOUS VERSIONS . . . 42
2.7.2 PHOENIX++: A COMPLETE REVISION OF PHOENIX . . . 43
2.8 PHOENIX++ AND HADOOP PERFORMANCE COMPARISON . . . 44
2.9 DSL-POPP . . . 45
2.10 SUMMARY OF THE CHAPTER . . . 47

3 RELATED WORK . . . **49**
3.1 THE HONE PROJECT . . . 49
3.2 SCALE-UP VS SCALE-OUT FOR HADOOP . . . 49
3.3 THE AZWRAITH PROJECT . . . 50
3.4 RELATED WORK OVERVIEW . . . 50

4 UNIFIED MAPREDUCE PROGRAMMING INTERFACE . . . **53**
4.1 JUSTIFICATION . . . 53
4.1.1 REQUIRED CODE BY HADOOP AND PHOENIX++ . . . 54
4.1.2 ADVANTAGES OF A UNIFIED MAPREDUCE PROGRAMMING INTERFACE . . . 54
4.2 RESEARCH QUESTIONS AND HYPOTHESIS . . . 55
4.3 PROPOSED INTERFACE . . . 55
4.3.1 UNIFIED INTERFACES’S STRUCTURE . . . 56
4.3.2 TRANSFORMATION PROCESS . . . 58
4.3.3 INTERFACE COMPONENTS AND TRANSFORMATION RULES . . . 59
4.3.4 SPECIAL COMPONENTS FOR TEXT PROCESSING . . . 65
4.3.5 PERFORMANCE COMPONENTS . . . 67

4.3.6 INTERFACE ACHIEVEMENTS AND LIMITATIONS 68
4.4 DEVISED CODE GENERATOR 69

5 EVALUATION 71
5.1 SAMPLE APPLICATIONS 71
5.2 SLOCCOUNT 72
5.2.1 PROGRAMMING LANGUAGE SUPPORT 72
5.2.2 COCOMO MODEL 72
5.3 PERFORMANCE EVALUATION 74
5.4 SLOC AND EFFORT EVALUATION 76

6 CONCLUSION AND FUTURE WORK 81

REFERENCES 83

APPENDIX A – Evaluated applications and generated code **89**
A.1 WORD COUNT 89
A.2 WORD LENGTH 93
A.3 HISTOGRAM 99
A.4 K-MEANS 106
A.5 LINEAR REGRESSION 120
A.6 MATRIX MULTIPLICATION 128
A.7 PRINCIPAL COMPONENT ANALYSIS 130

APPENDIX B – MapReduce for Many-core GPU architectures **135**
B.1 BACKGROUNDS OF GPGPUS 135
B.2 PRECEDENT RESEARCHES 136
B.3 MAPREDUCE PHASES IN GREX 137
B.3.1 BUFFER MANAGEMENT IN GREX 138
B.3.2 EVALUATION 139

1. INTRODUCTION

In order to improve performance, simplicity and scalability of large datasets processing, Google proposed the MapReduce parallel pattern [DG08] based on two simple operations, *map* and *reduce*, originally from functional programming languages. Since then, the MapReduce model has originated many implementations by both industry and academia. Some of these implementations have achieved great importance worldwide, such as Hadoop[1], which is suited for large-clusters architectures and Phoenix++ [TYK11] for low level multi-core architectures.

Meanwhile, Griebler et al. [Gri12, GF13] proposed a Domain Specific Language for Patterns-oriented Parallel Programming (DSL-POPP), aimed at providing a more intuitive and structured language for developing parallel applications.

Both research trends, DSL for parallel programming and MapReduce parallel pattern, converge to similar goals while aimed at disseminating parallel programming among not so specialized programmers and researchers, with optimal resource usage on modern architectures. Besides similar goals, both have also shown solid results and maturity in the field of high performance computing.

Inspired on DSL-POPP, this work proposes a unified MapReduce programming interface, from which it also proposes detailed rules for code transformations toward generating Phoenix++ [TYK11] and Hadoop [Whi09, Ven09, Lam10] MapReduce code, for shared-memory multi-core and distributed architectures, respectively. High-performance MapReduce solutions with highly optimized resource usage were chosen for the proposed code generation. Moreover, the coverage of other MapReduce frameworks as well as the construction of the code generator and the extension of DSL-POPP with the addition of the proposed unified MapReduce programming interface are proposed as future work.

The work is organized as follows.

Chapter 2 presents the most important MapReduce implementations and their evolution through more specialized architectures and improved resource usage. Chapter 3 gives continuity to chapter 2 but covering research projects more focused on unified MapReduce programming interfaces for different architectural levels, which comes to be more tightly related to our research. Additionally, Appendix B discusses some important MapReduce implementations for heterogeneous architectures, with GPGPUs, which were studied though are proposed to be addressed by future work.

Chapter 4 then provides a detailed description of the proposed unified MapReduce programming interface, the code transformation rules for Hadoop and Phoenix++, the research justification, research questions and evaluated hypothesis.

[1] http://hadoop.apache.org

Chapter 5 in turn provides a detailed analysis of the evaluation process, metrics and obtained results based on the sample applications described in the Appendix A.

Finally, chapter 6 presents the conclusions and future work.

2. BACKGROUND

This chapter presents the result of a study carried out on the MapReduce model and a set of important implementations of this model for distributed and shared-memory multi-core architectures.

It also presents a Domain Specific Language for Patterns-Oriented Parallel Programming (DSL-POPP), whose progress motivated our research.

Additionally, a brief overview of MapReduce implementations for heterogeneous architectures with GPGPUs is covered in Appendix B.

2.1 MapReduce

In order to facilitate constant *petascale* data processing, Google introduced the MapReduce model [DG08] based on two operations, *map* and *reduce*, originally from functional programming languages.

The MapReduce model aims at improving the simplicity of developing software for large datasets processing, as well as improving scalability by harnessing the distributed architecture in a seamless way. One of its main objectives is allowing programmers to focus on the business rules of the data processing, without worrying about data distribution to and among job instances, network communication, and other complex details required by distributed processing.

The model's definition avoids excessive prescriptions, since the basic expected structure consists of *map* and *reduce* tasks assigned to a set of worker units, which generate key/value pairs, which are in turn reduced according to some user-provided logic. It does not determine that this set of worker units must be run in a cluster, nor that the architecture must be independent commodity machines, though this is the environment in which the model originally showed its big performance advantages.

Dean and Ghemawat (2004, p. 3) clearly visualized it while originally proposing the model:

> Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

To illustrate the applicability of this model, Dean and Ghemawat present a sample implementation. The execution flow of a MapReduce job (illustrated in figure 2.1) starts by dividing the input dataset among the nodes, so that each node has early access to a chunk of it. The next step consists of starting up a single execution of the program per node, also assigning the role each node is to play during the execution, whether master or worker. The

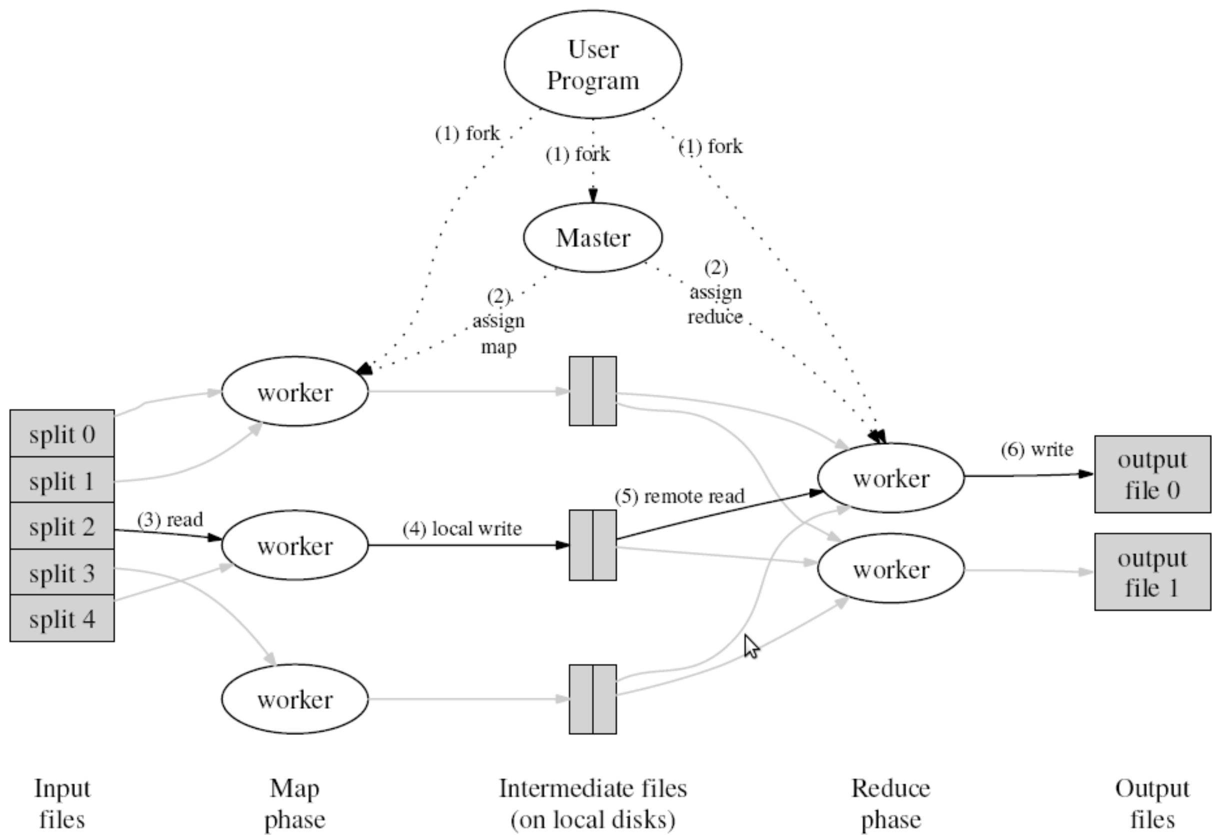


Figure 2.1: MapReduce job execution flow [DG08]

master node is given the responsibility of assigning *map* and *reduce* tasks for the worker nodes. It also monitors eventual idle nodes, so as to assign a new task, and failed nodes, so as to assign the failed task to some working and available node.

When a worker node is assigned a *map* task, it is also delegated a piece of the dataset in order to execute over it the user defined *map* function. According to MapReduce model, this function must receive a key/value pair, execute some process over it, and produce key/value pairs which will be the intermediate data to be received by some *reduce* task assigned to some other node.

When the map task is completely executed by a node, the intermediate key/value pairs are written to local disk, in order to avoid volatility. Through this persistence process, the values are sorted by key and organized into partitions in order to be subsequently grouped by key. The information about the locations of these partitions of key/values pairs is passed back to the master node.

At the end of the execution of all *map* tasks, the master node forwards the location of the grouped key/values pairs to the nodes who were assigned the *reduce* tasks. These nodes, in turn, call the *reduce* function provided by the user passing key and values in order to run the reduction code over it.

At last, when all the reduce tasks are completed, one or more output files will contain the conclusive results of the MapReduce job. The figure 2.2 shows in more details

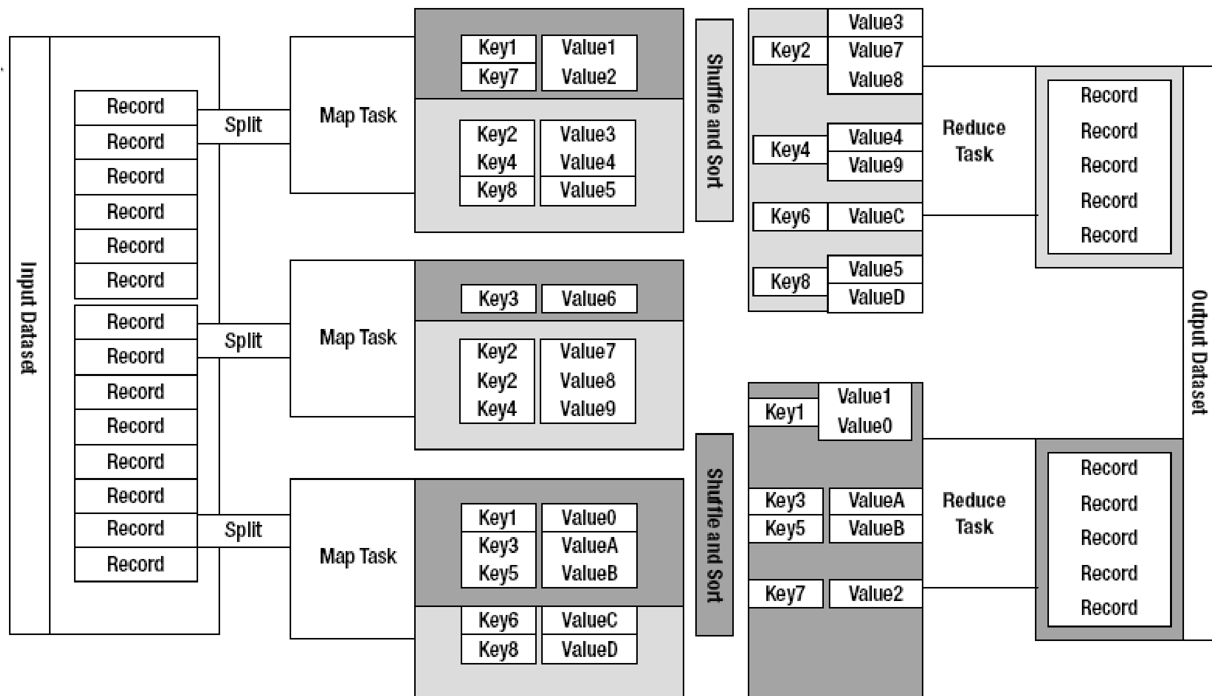


Figure 2.2: Key/value pairs distribution [Ven09]

the way the key/value pairs are passed through *map* and *reduce* functions so as to produce the final results of the job.

Additionally to the *reduce* and the *map* function, whenever running in a distributed environment, a *combine* function can be defined by the user, which takes place by the partitioning phase and runs a reduction algorithm over the data before creating the partition. This is specially useful to harness each node's computation power and save network bandwidth when passing partitions content to final *reduce* tasks assigned to other nodes across the cluster.

2.2 An Overview of MapReduce Implementations

Throughout a thorough research over MapReduce implementations since its first publication by Dean and Ghemawat [DG08], a set of works were analyzed [RRP+07, HFL+08, YRK09, HCC+10, JRA10, CCZ10, TYK11, SO11, GL11, JA12, CC13, BK13, BKX13], which includes implementations for distributed-memory, shared-memory and heterogeneous architectures.

The following list provides a chronological visualization of the most relevant MapReduce implementations.

- **2004** MapReduce original publication [DG08]
- **2005** Hadoop [Whi09, Ven09, Lam10]

- **2007** Phoenix [RRP+07]
- **2008** Mars [HFL+08]
- **2009** Phoenix Rebirth [YRK09]
- **2010** MapCG [HCC+10]
- **2010** Tiled-MapReduce [CCZ10]
- **2011** Phoenix++ [TYK11]
- **2013** Grex [BK13]
- **2013** Moin [BKX13]
- **2014** Glasswing [EHHB14]

The most widely adopted implementation for distributed systems, namely Hadoop, achieved the highest level of programming abstraction among the analyzed implementations previously mentioned. It lets programmers ignore almost completely the underlying architectural aspects and focus only on its specific application and the way to represent it through a MapReduce job.

By other hand, the shared-memory implementations, such as Phoenix++ [TYK11] and Tiled MapReduce [CCZ10], aimed at achieving maximum optimization, force programmers to deal in some level with details concerned to memory allocators, underlying data structures, fixed or dynamic sizes in memory, among other. Otherwise, the programmer is unable to minimize memory pressure, fit levels of cache memory and optimally use the processing units.

Finally, implementations for heterogeneous architectures such as Mars [HFL+08], MapCG [HCC+10], Grex [BK13], and Moin [BKX13] force programmers to deal with specific APIs through which to choose the more suitable components and data structures according to the size and volatility of data, in order to optimally exploit the different memory levels of GPU devices.

More recently, Glasswing [EHHB14] proposed a highly optimized MapReduce implementations for distributed, shared-memory and heterogeneous architectures, outperforming Hadoop on a 64-node multi-core CPU cluster (VU Amsterdam cluster of DAS4, each node equipped with a dual quad-core Intel Xeon 2.4GHz CPU, 24GB of memory and two 1TB disks configured with software RAID0) by a factor of 1.8 to 4 and by a factor from 20 to 30 on 16 of these nodes each one equipped with an NVidia GTX480 GPU. The work, though, does not provide details about the programming interface, presenting only some specific functions signatures. For this reason we did not use it in our research and did not include in the further studies of next sections.

2.3 Hadoop MapReduce

This section explains the logic and organization behind the Hadoop Core, the main subproject of Apache Hadoop project [1], which implements the MapReduce model described in section 2.1 and the Hadoop Distributed File System (HDFS) aimed at distributed-memory architectures.

According to Venner (2009, p. 4) [Ven09]:

> Hadoop is the Apache Software Foundation top-level project that holds the various Hadoop subprojects that graduated from the Apache Incubator. The Hadoop project provides and supports the development of open source software that supplies a framework for the development of highly scalable distributed computing applications. The Hadoop framework handles the processing details, leaving developers free to focus on application logic.

White (2009, p. 42) highlights the focus of Hadoop on Clusters of commodity machines [Whi09]:

> Hadoop doesn't require expensive, highly reliable hardware to run on. It's designed to run on clusters of commodity hardware (commonly available hardware available from multiple vendors) for which the chance of node failure across the cluster is high, at least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

When running a MapReduce job on Hadoop, each node in the cluster is given a specific role represented in a piece of code built from a set of pre-defined components. Hadoop provides two components for managing the MapReduce tasks distribution and execution, JobTracker and TaskTracker, respectively. [Lam10]

By analyzing the execution flow of a MapReduce job in Hadoop, it is possible to clearly visualize the way the components work together and play each one its role contributing to the process as a whole. It also becomes clearer where the programmer contribution takes place, by specially implementing the *map* and *reduce* tasks, and preview how data shall be shuffled, sorted and, finally, returned to the user application. [Whi09]

A JobClient instance, running a client application on a JVM in an external node, starts the process by communicating with a JobTracker node, asking it for a Job identifier, splitting the input dataset across the HDFS and, finally, calling submitJob() on the JobTracker instance. Once submitted, the job holds references to all *map* tasks, which in turn reference each input split processed by the JobTracker instance. The *reduce* tasks are also created at this stage with a configured number of instances. [Whi09]

Once created, the tasks are assigned to TaskTracker nodes, which communicate their availability by periodically sending heartbeat signals to the JobTracker. It is possible for a TaskTracker to be assigned more than one *map* or *reduce* task, so as to run in parallel in multi-core systems [Whi09].

[1] http://hadoop.apache.org

While assigning a *map* task to a TaskTracker the JobTracker looks for data locality attempting to make the TaskTracker as close as possible from its targeted data split. Once everything is ready for task execution, the TaskTracker looks for the Java code library containing the *map* or *reduce* user defined code and runs it over the target data. The execution of a TaskRunner is started by default within a new JVM instance for fault tolerance purposes. If the JVM instance executing the TaskRunner code breaks or becomes hung, for any reason, this process can be killed without impacting the TaskTracker process, which will be running in a different JVM instance. [Whi09]

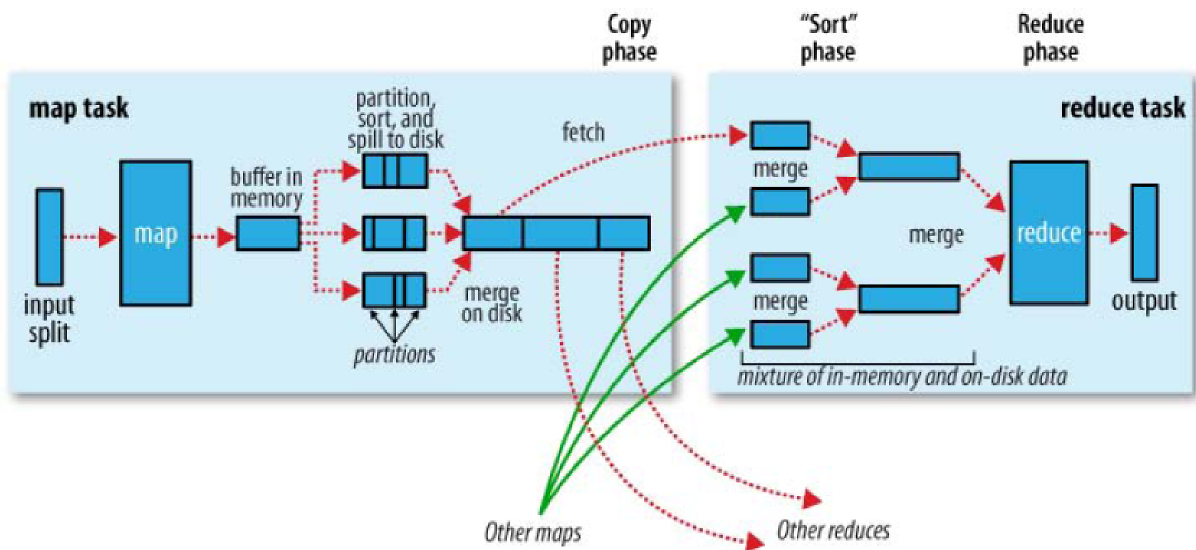


Figure 2.3: Shuffle and Sort steps with Hadoop [Whi09]

As previously described, the MapReduce model states that the *map* tasks yield key/value pairs, which are afterwards grouped by key and then processed by *reduce* tasks so that, while processing a given key, the *reduce* task is guaranteed to hold all possible values for such key. This process is illustrated in figure 2.3.

As the TaskRunner's execution of a *map* operation produces key/value pairs, these pairs are buffered in memory to be, after a configured threshold, grouped in partitions and persisted to disk. The partitions are built so as to correspond to the *reduce* tasks that will process it. During the partitioning process, the partitions content is sorted by key. Within this phase, if a *combine* function was provided, such function is run over the key/values pairs in order to decrease the amount of data to be persisted to local disk and to be after transferred to *reduce* nodes across the network.

After successfully completing a *map* operation, the TaskRunner process reports it and the partitions locality to TaskTracker parent process, which in turn communicates these informations to JobTracker service.

Instead of waiting all *map* operations to be successfully completed, the TaskTracker nodes responsible for *reduce* operations keep watching over the cluster for partitions that are ready to be copied. The locality information about the data to be reduced is obtained through

the JobTracker. Thereby, when all *map* operations are completed, almost all partitions will be already copied and locally available to *reduce* operations.

Finally, as *reduce* operations finish their execution, the JobTracker is notified and sends commands to erase mapping output data from the nodes which executed the *map* tasks.

2.4 Phoenix

In 2007, Ranger et al. proposed an implementation of MapReduce (section 2.1) named Phoenix [RRP+07] aimed at evaluating the suitability of MapReduce model to multi-core and SMP architectures.

Compared to Cluster architectures, for which there are MapReduce implementations such as Hadoop (section 2.3), shared-memory multi-core and multi-processor architectures bring a set of peculiarities which in turn require different development skills. Phoenix provides an API for easy development of functional-style *map*, *reduce* and related operations which, once implemented, can be run over a provided Runtime which in turn dynamically manages all parallelism mechanisms, resource management and fault recovery. This way, the whole execution is adapted to the specific system characteristics and scalability, such as number of cores, processors and memory details.

Phoenix Runtime provides a scheduling mechanism which keeps track of processors availability, delegating tasks in parallel among the processing units so as to optimize the load balance and task throughput. To improve locality, the tasks granularity is adjusted according to memory specific hierarchy and capacity.

By managing low-level details, one of the main goals is to simplify parallel programming by allowing programmers to focus on their specific problem assuming, though, that there are problems not suitable to a MapReduce paradigm. For these, the ease of programming would only be provided by some other parallel pattern. On the other hand, Phoenix allows the programmer to customize almost any low-level default mechanism it provides, as soon as it is found a more optimized way to deal with the system architecture in an specific scenario.

The Phoenix API

The API provided by Phoenix is written in C, and thus compatible with *C* and *C++*, however can be extended to *Java* and *C#*. It provides a set of functions to be used by the programmer's application, and another set of functions definitions to be implemented and provided by the programmer. The first set includes functions to initialize the process and emit intermediate and output values, and the second set defines functions that comprehend

the Map and Reduce logic, apart from partitioning logic and key comparison. Table 2.1 shows the two sets of functions, being the three first the set of Runtime provided functions, and the remaining the functions defined by the programmer.

Table 2.1: Phoenix API functions

Function	Description
int phoenix_scheduler (scheduler_args_t * args)	Initializes the runtime system. The scheduler_args_t struct provides the needed function and data pointers.
void emit_intermediate(void *key, void *val, int key_size))	Used in Map to emit an intermediate output <key,value> pair. Required if the Reduce is defined.
void emit(void *key, void *val)	Used in Reduce to emit a final output pair.
int (*splitter_t)(void *, int, map_args_t *)	Splits the input data across Map tasks. The arguments are the input data pointer, the unit size for each task, and the input buffer pointer for each Map task.
void (*map_t)(map_args_t*)	The Map function. Each Map task executes this function on its input.
int (*partition_t)(int, void *, int)	Partitions intermediate pair for Reduce tasks based on their keys. The arguments are the number of Reduce tasks, a pointer to the keys, and a the size of the key. Phoenix provides a default partitioning function based on key hashing.
void (*reduce_t)(void *, void **, int)	The Reduce function. Each reduce task executes this on its input. The arguments are a pointer to a key, a pointer to the associated values, and value count. If not specified, Phoenix uses a default identity function.
int (*key_cmp_t)(const void *, const void*)	Function that compares two keys, used for merge phase.

The process is started by an application call to the *phoenix_scheduler()* function, which from there on takes care of the whole process, controlling load balancing, fault tolerance and all other details involved in the MapReduce job. Many configuration details, though, must be parametrized by the programmer through a *scheduler_args_t* type whose attributes are described in Table 2.2.

The Phoenix Runtime

As previously described, the execution process starts when the application calls the *phoenix_scheduler()* function. From that moment on, the scheduler takes place and starts to create and manage threads that run the MapReduce mechanism, having as its basis the configurations and pointers provided by the *scheduler_args_t* parameter. For each available core, or each thread of a multithreaded core, the scheduler spawns a worker thread which runs all over the process being dynamically assigned *Map* and *Reduce* tasks along the execution.

Through the provided *Splitter*, the running *Map* tasks look for data to process, using pointers so as to avoid copying data. As each *Map* task is being completed, it emits intermediate <key,value> pairs.

Table 2.2: Phoenix scheduler attributes

Attribute	Description
Input_data	Input data pointer; passed to the Splitter by the runtime.
Data_size	Input dataset size
Output_data	Output data pointer; buffer space allocated by user.
Splitter	Pointer to Splitter function.
Map	Pointer to Map function.
Reduce	Pointer to Reduce function.
Partition	Pointer to Partition function.
Key_cmp	Pointer to key compare function.
Unit_size	Pairs processed per Map Reduce task.
L1_cache_size	L1 data cache size in bytes.
Num_Map_workers	Maximum number of threads (workers) for Map tasks.
Num_Reduce_workers	Maximum number of threads (workers) for Reduce tasks.
Num_Merge_workers	Maximum number of threads (workers) for Merge tasks.
Num_procs	Maximum number of processors cores used.

As each <key,value> pair is emitted, the *Partition* function ensures it is grouped within a unit with all values of the same key. When all *Map* tasks are finished, these units are ready to be processed by *Reduce* tasks.

Each worker is dynamically assigned *Reduce* tasks until all intermediate pairs have been processed. Each *Reduce* task processes reduction on all values of a given key. Depending on the input dataset and the application specific logic, this peculiarity of *Reduce* tasks can generate some load imbalance, since some keys might have many more values than others (i.e. Word Count, described in the Appendix A).

A preliminary sorting is performed by *Partition* function and, when all *Reduce* tasks are finished, the merge phase ensures the final sorting. There are applications which do not require the output pairs to be ordered, however Phoenix always executes this step accordingly to the originally proposed implementation (section 2.1).

Figure 2.4 shows the basic data flow of a MapReduce application with Phoenix, as explained in previous paragraphs.

Along the process, the data is stored in buffers allocated in shared memory. The main type of buffers, named *Map-Reduce buffers*, are buffers organized in a two-dimensional array strategy. These buffers are hidden from user application and are internally used by Phoenix Runtime to store in memory the intermediate <key,value> pairs generated from the *Map* tasks and consumed by *Reduce* tasks. The figure 2.5 shows how the buffers behave while running a WordCount application (Appendix A).

During the execution of a *Map* task, data is written into a *keys-array* represented by a column in the 2D-array, storing on it one key per position and a *vals-array* which contains all values emitted for that key during that specific *Map* task. This *keys-array* starts with a default height of 256 positions that can be expanded and acts as a hash table for <key,values> pairs,

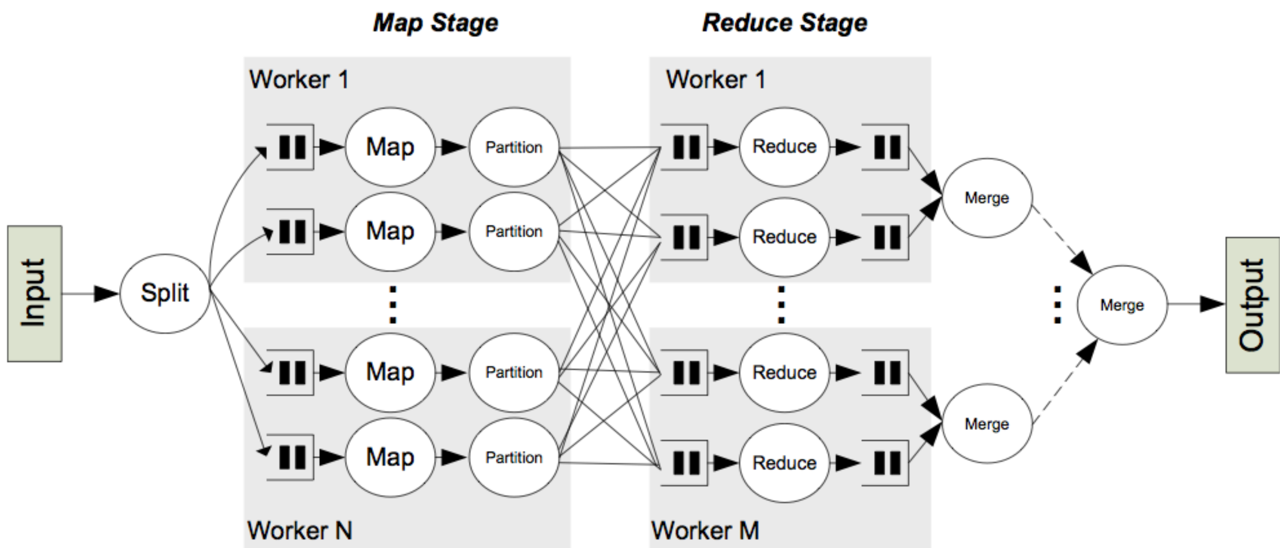


Figure 2.4: The basic data flow for the Phoenix runtime [RRP+07]

where the values are stored as a *vals-array*. The pairs of the *keys-array* are dynamically sorted by key as each one is inserted.

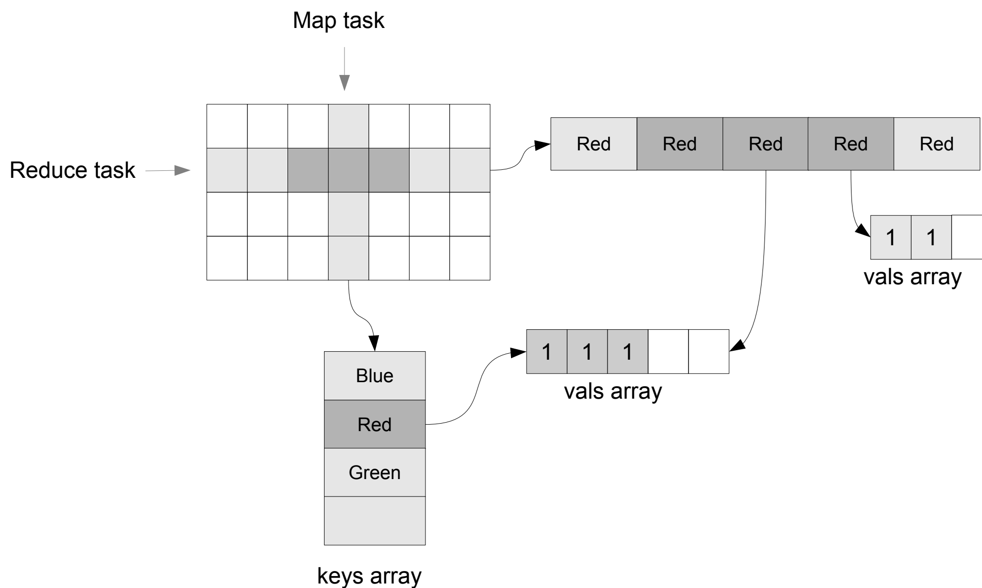


Figure 2.5: Phoenix data structure for intermediate key/value pairs

During the *Map* phase, the Map tasks work in a column-oriented approach. After that, when the *Reduce* phase takes place, the orientation of *Reduce* tasks turns to be row-wise. Each task goes through a given row which is guaranteed to contain all values emitted for a given key during all *Map* tasks. After merging these values, the *Reduce* task runs the provided reduction function storing the reduced pairs in another set of buffers named *Reduce-merge buffers*.

When the *Reduce* phase is completed, the *Reduce-merge buffers* are sorted by key and, finally, the Runtime writes the result in the user provided *Output_data* buffer.

As previously described, each *Map* task processes a unit, whose size can optionally be parametrized by the user through the *scheduler_args_t* type. The default size, though, is adjusted by Phoenix to fit the L1 data cache in order to improve efficiency. If the system also offers an L2 data cache, the Phoenix Runtime can use it to *prefetch* data for the task to be executed after the current executing task.

While evaluating Phoenix through a set of common applications and three different datasets, the results showed important differences concerning suitability to MapReduce model, code complexity and overall performance.

In order to evaluate performance, two categories of architecture, CMP and SMP, were used. The hardware support for multithreading was exploited when using a CMP based on the UltraSPARC T1 multi-core chip with 8 multithreaded cores sharing the L2 cache [KAO05], and the speedup achieved was high when increasing the number of cores. With SMP architecture based on the UltraSPARC II multi-processor with 24 processors, the performance was also considerably improved when scaling the system.

Some applications such as *Matrix Multiplication* can achieve a higher granularity and thus better exploit the cache memory available in each core, achieving even superlinear speedups. Some others such as *Word Count*, though, present suboptimal load balance at the Reduce phase when a higher number of cores was used. An also superlinear speedup is achieved by *Reverse Index*, which shows its heavy processing at the merge phase, taking high advantage of cache memory.

Summarizing, applications that are naturally key-based and some tailored to this model, show good results and resource usage with Phoenix, such as *Word Count*, *Matrix Multiplication*, *String Match* and *Linear Regression*. Others, although tailored to MapReduce model, present no advantage but only avoidable overheads which evinces a better suitability with some other parallel programming pattern.

2.5 Phoenix 2

The original Phoenix (described in section 2.4) performed well on small-scale CMP and SMP systems with uniform memory access and 24 to 32 hardware threads, but when benchmarked over large-scale systems with NUMA characteristics it underperformed.

In 2009, Yoo at el. published a new version of Phoenix System for MapReduce implementation on shared-memory systems [YRK09], this time with resource usage improvements for NUMA architectures.

At this second work, the system used for evaluation was a Sun SPARC Enterprise T5440, quad-chip, 32-core, 256-thread UltraSPARC T2+, with NUMA characteristics [SPA10]. The 256 supported hardware threads are distributed among the 4 chip. Each chip

has 4 channels of locally attached main memory (DRAM). Whenever a remote access to some other chip's memory is required, the access is 33% slower than the access to locally attached memory. This way, when a program uses at most 64 threads, it is able to use only one chip and avoid remote memory access, but when it needs more than 64 threads the cost of non-uniform access cannot be avoided.

The authors realized that three issues should be specially addressed in order to overcome the challenges presented by this category of architecture. First, the runtime must hide the non-uniform accesses, through locality enhancement. Second, use specialized data structures for input, output and intermediate data for different kind of datasets in order to improve performance with large datasets. Lastly, dynamically identify mechanisms provided by the operating system for memory management and I/O operations.

In order to address these issues and improve performance on large-scale NUMA systems, the second version of Phoenix proposes a three-layered optimization focused on algorithm, implementation and OS interaction.

First of all, in a NUMA architecture, the runtime cannot assign tasks without taking into consideration the locality of the targeted data chunks, it must be NUMA-aware. Such locality must be taken into account, otherwise it can end up having tasks running locally but continuously working on remote data, causing additional latency and unnecessary remote traffic.

To solve the locality issue, Phoenix 2 introduces a task queue per locality group, where each group is assigned tasks that work with local data chunks, sharing memory and processor, avoiding remote access as much as possible. The limit is when a given local queue becomes empty, when it happens the algorithm starts stealing tasks from neighbor queues. As load balance is improved, this stealing of tasks happens less frequently.

The data management mechanism of the original Phoenix, presented in section 2.4, performed well with medium-sized datasets, but when brought to larger datasets and large-scale architectures, goals of Phoenix 2, it was dramatically suboptimal.

The algorithmic optimizations include the locality groups, in which tasks are allocated in a way to work on local data. But after *Map* phase is concluded, some keys might have been emitted by tasks located in different locality groups, causing some *Reduce* tasks to be unable to avoid remote accesses in a NUMA system.

While working with larger datasets, the buffers reallocation also appeared as a critical bottleneck whenever some buffer ran out of default space. Each *keys-array* hash table is dynamically ordered during *Map* phase to enable binary search, what shows a downside of forcing reallocation of all keys coming lexicographically after some new key. Similar reallocation problem affects *vals-array*.

Phoenix 2 overcomes the reallocation issue of *keys-array* by increasing the number of hash buckets allowing each *keys-array* to store only one key most of the times. This

approach significantly optimizes *Map* tasks. In order to also optimize *Reduce* tasks, an iterator interface is added to hide the way *vals-arrays* are arranged in memory, allowing it to be arranged in the way that better optimizes locality, as well as allowing prefetching mechanism that mitigates the latency of NUMA remote accesses.

Other approaches such as linked-lists, tree structure and combiners were evaluated to optimize *keys-array* and *vals-array* but offered little or no optimization.

MapReduce systems tend to be I/O and memory intensive, and so is Phoenix systems. Some peculiarities need to be addressed, such as the memory allocations for significant amount of intermediate data at *Map* phase and the deallocation of this memory by some other thread running a *Reduce* task. The input and output datasets also represent large I/O operations.

In order to address these specific needs, a suited memory allocator must be chosen. *mmap()* showed great scalability while the number of threads increased, as well as a mechanism for thread stacks. The only downside was the increased thread join time due to calls to *munmap()*, what was addressed by a thread pool implementation, which allowed threads to be reused across many phases and iterations.

Phoenix 2 improved the original Phoenix's Runtime, and enabled significant speedups in some scenarios. With single-chip machines of 64 threads the average improvement was 1.5x with a maximum of 2.8x. For large-scale NUMA systems the average was 2.53x with a maximum of 19x. It evinces the significant improvements achieved in NUMA systems.

Operating System interaction optimizations had the most impact, but still represented the main bottlenecks. Once both *sbrk()* and *mmap()* presented limited scalability, no memory allocator successfully scaled up to the 256 threads offered by UltraSPARC T2+ [SPA10], even though many options have been experimented.

In order to confirm the assumption of OS being the main bottleneck, it was tested the pre-allocation of all memory needed for a WordCount workload and the reading of all the data into the user address space before the Runtime started executing, and as result the system scaled well up to the 256 threads.

In summary, this second version of Phoenix system proved that optimizing a Runtime for a large-scale, shared-memory NUMA environment can be non-trivial but great speedups can be achieved.

2.6 Tiled-MapReduce

In 2010, Chen et al. proposed a MapReduce implementation based on Phoenix 2 (section 2.5) with an innovative *tiling strategy*.

Named Tiled-MapReduce [CCZ10], it considers that the core limitation of a MapReduce implementation for multi-core is that the available memory in such architectures cannot process more than some gigabytes of data, while a distributed implementation such as Hadoop (section 2.3) can smoothly process terabytes or even petabytes. The reason of this limitation is that MapReduce model is designed to work with the same input and intermediate data along the entire process, what demands available memory in the same proportion of such data.

The *tiling strategy* [CM95] turns to be a feasible way to overcome such limitation, by dividing the whole job in sub-jobs and iteratively executing, optimally harnessing the resources provided by the architecture and avoiding input and intermediate data to persist along the entire life cycle of a processing phase.

For more optimal use of architecture, the proposed mechanism enhances reuse of data structure allocation among execution iterations, includes NUCA/NUMA awareness (based on Phoenix 2 enhancements, section 2.5) and employs pipelined execution among sub-job's reduce phase and the successive sub-job's map phase.

The iterative process allows partial results to be persisted and eventually restored when some failure breaks the system. When resumed, the process continues from the state achieved by the latest sub-job completion, what turns to be also an effective fault tolerance mechanism.

Partial results can also be used to provide real-time information about process progress, such as explored by Condie et al. in a project called MapReduce Online [CCA+10].

Figure 2.6 shows the execution flow for a Tiled-MapReduce job and highlights the extension over Phoenix 2 implementation.

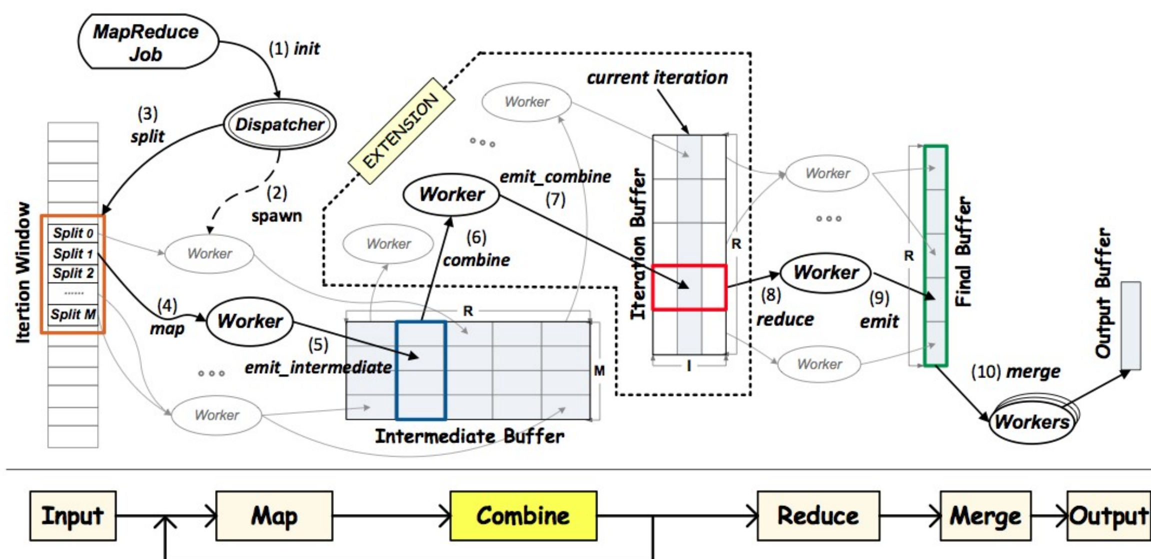


Figure 2.6: Execution Flow of Tiled-MapReduce [CCZ10]

The *mr_dispatcher* function starts the process splitting the input data in the Iteration Window. For each iteration, input data chunk is split into M pieces distributed among Map tasks. Each available CPU core is bound to a worker job which keeps looking for pending tasks whenever idle.

The *Map* phase invokes user-provided *map* function, in which the runtime-provided *emit_intermediate* function is executed and inserts *key/value* pairs into the Intermediate Buffer.

Similarly to Phoenix 2, the Intermediate Buffer is organized as an M by R matrix, where R is the number of reduce tasks and the column orientation searched for *combine* phase, which executes *emit_combine* immediately after *map* execution, populating the *Iteration Buffer*. Following MapReduce model's original definition (section 2.1), *combine* phase is a forehand *reduce* phase, usually executed internally to a node in a cluster, before sending results back through the network. In Tiled-MapReduce context, *combine* phase is a forehand *reduce* phase executed internally to each iteration process.

After the *combine* phase, all resulting *Iteration Buffers* are provided as input for the final *reduce* phase, along which the workers invoke the programmer-provided *reduce* function. This function in turn populates the *Final Buffer*, which finally is sorted and merged into the *Output Buffer*.

As previously mentioned, the adoption of *tiling strategy* by Tiled-MapReduce model not only allows the input data to be much larger by splitting the execution in multiple sub-jobs, but also allows a set of optimizations on resources usage by circumventing limitations of original MapReduce model, such as the need of *map* phase completion before starting *reduce* phase. This set of optimizations covers memory reuse, data locality and task parallelism.

The proposed implementation for Tiled-MapReduce, namely Ostrich, enhances memory efficiency by reusing memory allocations for input data and intermediate data among sub-jobs. Once an *Iteration Buffer* has been allocated for the first iteration, it is reused by the subsequent iterations. The same happens with *Iteration Window* over Input Data.

For Input Data, the original model suggests pointers to every piece of the split data, however this strategy may cause excessive pressure on memory. Instead of taking this approach, Ostrich is designed to adapt its execution to the concerned data and application, applying different strategy according to keys duplication and available memory. If there are abundant duplicated keys, memory copy is allowed, and memory efficiency is improved by harnessing cache levels. Otherwise, if there are not abundant duplicated keys, pointers are used instead. Behind this, a single fix-sized memory buffer is allocated for the first iterations and reused for all subsequent iterations, being remapped to the corresponding *Iteration Window* whenever a new sub-job starts.

For Intermediate Data, a single global allocation is also exploited. While running, each sub-job populates the global *Intermediate Buffer* with the result of *Map* phase and starts reducing such result through *Combine* phase which in turn runs in parallel (pipelined) with the subsequent sub-job's *Map* phase. Each iteration's *Combine* phase generates an *Iteration Buffer* which is limited to a threshold in order to fit into the last-level cache. Whenever the limit is exceeded, the runtime runs an internal *Compress* phase which, similarly to *Combine* phase, runs the same user-provided *reduce* function, thus improving memory efficiency also with Intermediate Data.

As larger the Input Data, more accesses are required from *map* and *reduce* functions to Input Data and Intermediate Buffer, causing poor temporal and spatial data locality.

Again, Ostrich circumvents the problem by splitting Input Data and, hence, decreasing input to *Iteration Windows*. By working with a shortened set of data, the sub-jobs can fit the cache and achieve great speedups.

Similarly to Phoenix [YRK09], Ostrich also was designed to be suitable for architectures with non-uniform access to cache or memory (NUCA and NUMA). In order to be optimized for such environments, a *Dispatcher* thread spawns one *Repeater* thread for each chip. The *Dispatcher* delegates sub-jobs to *Repeaters* as soon as they become idle.

The *Repeaters* play the role of reproducing the entire execution flow described earlier inside each chip, hence avoiding accesses beyond the chip boundaries. However, cross-chip accesses can only be avoided until *Combine* phase is completed, becoming unavoidable from final *Reduce* phase.

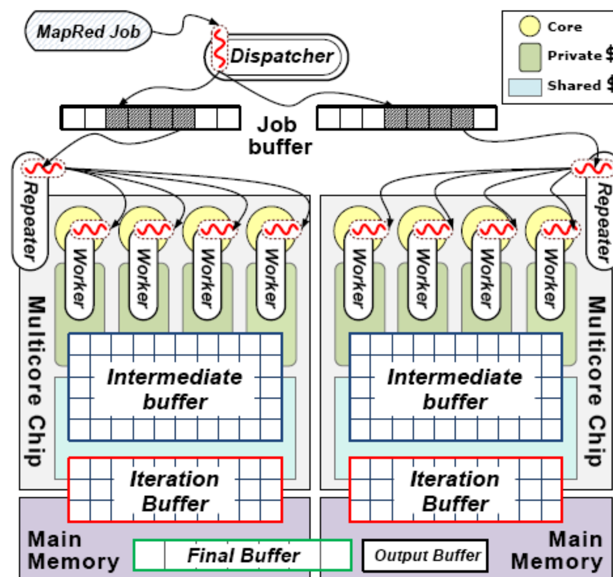


Figure 2.7: NUCA/NUMA-aware mechanism [CCZ10]

Figure 2.7 illustrates Ostrich's NUCA/NUMA-aware mechanism.

In Tiled-MapReduce, there is no need for a strict barrier between the end of *Map* phase and the beginning of *Reduce* phase, as it is with original MapReduce model. The *tiling*

strategy allows runtime to pipeline tasks and mitigate idle threads avoiding the imbalance among tasks when the amount of keys and values per key is unpredictable.

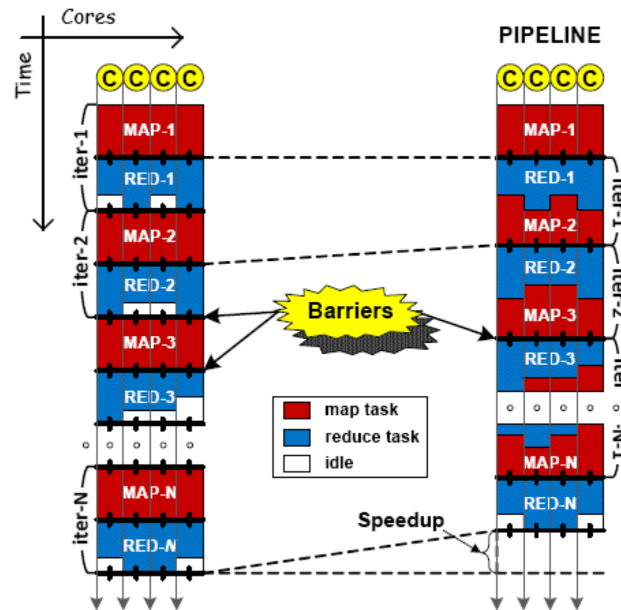


Figure 2.8: Task Parallelism through pipeline [CCZ10]

Figure 2.8 illustrates the tasks pipeline, where the *Map* task of subsequent iteration is able to start even if the *Reduce* task has not finished. Due to *Intermediate Buffer* allocation reuse among iterations, this buffer needs to be duplicated when pipeline is enabled, thus increasing memory consumption. Depending on the application, this may cause excessive memory pressure and loss of performance. Ostrich, for that reason, allows user to disable pipeline in order to avoid specific suboptimal cases.

Experimental results showed that Ostrich outperforms Phoenix 2 due to the mentioned optimizations, made possible through Tiled-MapReduce. Experiments using different types of data-parallel applications show that Ostrich can save up to 87.6% memory, cause less cache misses, and make more efficient use of CPU cores, resulting in a speedup from 1.86x to 3.07x.

All experiments were conducted on a 48-core machine with eight 2.4 GHz 6-core AMD Opteron chips. Each core has its own private 64KByte instruction and data caches, and a 512KByte L2 cache. The cores on each chip share a 5MByte L3 cache. The size of the physical memory is 128GByte.

2.7 Phoenix++

In 2011, the Phoenix project evolved to a third version and a completely new implementation of MapReduce for shared-memory multi-core architectures, this time built over

C++ programming language aiming to exploit object-oriented facilities to modularity and *in-line* compilation.

The new Phoenix++ Runtime [TYK11] focuses on overcoming a set of limitations noted in previous versions (sections 2.4 and 2.5). Talbot et al. mentioned that many users reported the need to rewrite and circumvent some parts of Phoenix engines in order to achieve a reasonable optimization for specific applications.

The static MapReduce pipeline adopted by Phoenix until its second version showed suboptimal execution for some types of applications and the need of an adaptable Runtime.

Besides that, recent researches (section 2.6) achieved up to 3.5x speedup over Phoenix 2 (section 2.5).

2.7.1 Limitations of previous versions

The table 2.3 describes the three possible key distributions for MapReduce applications as well as the more optimal *data container* for each type of distribution.

Table 2.3: Key distributions and optimized *data containers* proposed by Phoenix++ [TYK11]

Key distribution	Description	Sample applications	Container type
:	any map task can emit any key, where the number of keys is not known before execution	Word Count	variable-size hash table
***:k**	any map task can emit any of a fixed number of keys *k*	Histogram, Linear Regression, K-means, String Match	array with fixed mapping
1:1	each task outputs a single, unique key	Matrix Multiplication, PCA	shared array

Precedent versions ignored such peculiarities, proposing a static and uniform way to execute all kind of applications. This approach exposed inefficiencies on intermediate key-value storage, combiner implementation and task chunking.

Inefficient Intermediate key-value storage

The use of map thread-specific fixed-width hash tables, with a constant number of hash buckets (described in section 2.4), allows each single *reduce* task to traverse the same bucket in each hash table in a cross-cutting fashion, avoiding locking. Nevertheless, this strategy limits the performance for **:** workloads with large number of keys, and performs

unnecessary processing and memory usage in **:k* and *1:1* workloads. For instance, a *picture histogram* application fits better with fixed size arrays, instead of hash tables, since the number of pixels is known in advance.

Ineffective Combiner

In previous versions, the *Combiner* phase is always run at the end of *Map* phase. This strategy is an inheritance from implementations of MapReduce for clusters, where it is used in order to minimize network traffic among nodes. In SMP architectures, though, there is greater cost in memory allocation than in data traffic. The *Combiner* execution at the end of *Map* phase forces the data to remain allocated for a longer time causing memory pressure, and also suffers from cache-misses, which greatly affects performance. In such cases, users tend to implement *Combiner* calls inside the *Map* function.

Exposed Task Chunking

Phoenix exposes data chunks to *Map* function implementation, bringing extra code into user code and allowing the user to manipulate such chunks in ways that can break Runtime capabilities of managing data.

2.7.2 Phoenix++: A Complete Revision of Phoenix

Phoenix++ is based on a totally new source code written in C++, which promotes capabilities toward modularization through *templates* and *inline* functions.

Two core abstractions compose the new modularized Runtime: *containers* and *combiner objects*. Besides that, Phoenix++ allows users to alternate between memory allocators and choose whether the final key-value sort is required or avoidable.

Containers

By knowing in advance the desired workload characteristics, the user is able to choose the more suitable *container* to hold intermediate key-value pairs, rather than being forced to use a hash table structure.

For **:** workloads, the *hash container* implements a data structure similar to the one used in Phoenix, except for making hash tables resizable. For **:k* workloads, the *array container* implements a fixed-size, thread-local array. And for *1:1* workloads, the *common array container* implements a non-blocking array structure shared across all threads.

The resizable hash tables in *hash container* causes the correspondence between keys and bucket indices to be lost. Nevertheless, Phoenix++ overcomes this issue by copying the result of *Map* phase into a new fixed-width hash table with the same number of buckets as *reduce* tasks. Surprisingly, such copy overhead still allows execution to achieve speedup over Phoenix. The *array container* easily increases overall performance by avoiding the cost of hashing keys and repeatedly checking whether the hash table must be resized. The *common array container* avoids many verifications involving hashing and synchronizing, since it is known that each key produces a single value. Finally, a *container interface* is also provided in order to allow users to implement their own *container* without circumventing Phoenix++ mechanisms.

Combiner objects

In Phoenix++, a new *stateful* object takes place to incorporate the *combiner* behavior and is called after each key-value pair is emitted by the *map* function.

There are two standard implementations and an interface for customized implementations. The *buffer_combiner* buffers all emitted values in order to run the *Combine* phase only until *reduce* function is called, following the original model, whereas the *associative_combiner* in turn does not buffer the values but executes *combine* logic incrementally after each value is emitted.

Evaluation

For all types of workloads, Phoenix++ achieves improved performance and more scalability up to 32 threads. Alternative *containers* and *combiners* evince that there is no single option suitable for all workloads. The *template* and *inline functions* capabilities of C++ also eliminate function calls in the *map* and *reduce* inner loops. The overall speedup achieved an average of 4.7x over Phoenix 2.

Experiments were conducted in a multi-core system with 4 Nehalem-EX chips, 8 cores per chip, 2-way Hyper-Threading per core, a total of 64 hardware contexts on system and 32 GB of RAM memory. Per core data / instruction L1, 32 KB. Per core L2, 256 KB. Shared L3, 24 MB.

2.8 Phoenix++ and Hadoop performance comparison

In order to implement a high-performance recommendation system with MapReduce for multi-core architectures, Cao et al. [CSW13] evaluated Phoenix++ against Hadoop, achieving 28.5x speedup over Hadoop for a *word count* sample application, as shown in the

Figure 2.9 (note the log-scale in the y-axis). Their final recommendation system performed 225% faster than the Hadoop counterpart without losing recommendation quality.

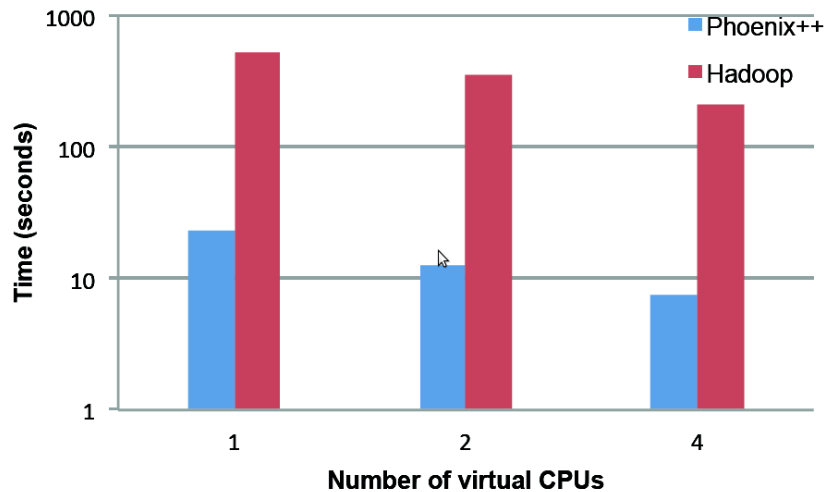


Figure 2.9: Experiment of 1 GB *word count* using Phoenix++ and Hadoop on a multi-core architecture. The y-axis is in a logarithmic scale. [CSW13]

Wittek and Darányi [WD11] perform a case study of high-performance computing and cloud technologies for applications of digital libraries. Their work observes that Hadoop is able to run into a single shared-memory multi-core machine executing multiple jobs simultaneously by launching multiple Java Virtual Machines (JVM), and hence exploiting internal parallelism of multicore nodes. However, such solution performs poorly, particularly due to JVM instances management, being this the reason why Hadoop emphasizes its focus on data-intensive computing. In other words, a large volume of data has to be processed reliably and the execution time is considered of secondary importance.

Tiwari and Solihin [TS12] carefully analyze key performance factors of shared memory MapReduce implementations. Their work particularly observes that Hadoop uses disk-based file system to store intermediate key-value pairs, which is required for distributed execution and contingency for fault tolerance. However, such requirements are inexistent for multi-core environments, in which the MapReduce intermediate data can be stored in the main memory and be still optimized to better resource usage.

2.9 DSL-POPP

After an overview of MapReduce implementations for different levels of architectures in the previous sections, this section analyzes the DSL-POPP, which, proposed by Griebler et al. [Gri12, GF13, GAF14], aims at providing both specialized syntax and compiler based on C for patterns-oriented parallel programming through a Domain Specific Language [Fow10, Gho11]. Programmers are, thus, supplied with special coding blocks through which

they are able to define the chosen parallel pattern(s) [MSM04, McC10] more suitable for the parallel application being developed.

From the chosen pattern(s) (i.e. *Master/Slave*, *Pipeline*, *Divide and Conquer*) the compiler is able to generate a parallel code with *threads* management, load balancing and any other low-level detail needed for optimal parallel execution.

DSL-POPP also provides the possibility of nested pattern structures, which is represented by more than one level of the same pattern (i.e. *Master/Slave*) or different patterns, for instance, each stage of a *Pipeline* process can be structured as a *Master/Slave* sub-process. Figure 2.10 demonstrates a nested hybrid patterns structure with *Pipeline* and *Master/Slave*.

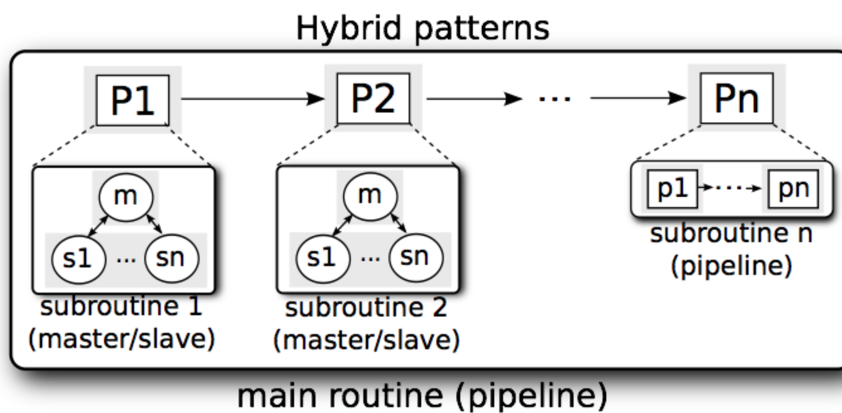


Figure 2.10: Hybrid structure with mixed patterns [GF13]

Currently, DSL-POPP supports only shared memory multi-core architectures, however its coding structure aims at being futurely portable for different architectures, such as distributed or even modern heterogeneous systems with GPGPUs, in a seamless way for programmers.

The Listing 2.1 shows an example of Master/Slave pattern implementation with DSL-POPP and how the code blocks are structured for this parallel pattern.

```
1 $MasterSlavePattern int func_name(){
2   @Master(void *buffer, const int size){
3     //pure C/C++ code
4     @Slave(const int num_threads, void *buffer,const int size, const
          policy){
5       //pure C/C++ code
6     }
7     //pure C/C++ code
8   }
9 }
```

Listing 2.1: Master/Slave programming interface.

The DSL-POPP project is an ongoing research, which motivated our current research with the aim of including the MapReduce model as a new parallel pattern to be supported by the DSL, however some challenges arose. DSL-POPP currently does not support built-in functions, such as those required by MapReduce to emit intermediate key/value pairs and finally emit the reduced values per key. The MapReduce implies very specialized underlying stages of execution, such as *shuffle*, *group* and *sort*, whereas the patterns currently supported by DSL-POPP deal with a more high level of simply dividing the execution of almost any parallelizable algorithm.

Moreover, to implement a whole new MapReduce mechanism would be to neglect the well evolved projects already widely used for shared and distributed environments.

Nevertheless, Griebler et al. successfully evaluated key concepts on developing and evaluating domain specific languages for abstracting parallel programming, which served as an essential basis for our research on a unified programming interface for MapReduce.

2.10 Summary of the chapter

As introduced in the beginning of the chapter, a comprehensive study was performed over the MapReduce model and a set of important implementations of this model for different architectural levels.

This chapter also presented performance evaluations among these works and how MapReduce implementations evolved to state-of-the-art solutions, with optimal performance and resource usage, even in very low-level architectures.

Such evaluations led our research to focus on Hadoop (section 2.3) for distributed systems and Phoenix++ (section 2.7) for shared-memory multi-core architectures. Heterogeneous architectures with GPGPUs (Appendix B) are indicated for future work, as described in chapter 6.

Finally, DSL-POPP served as an essential basis for our research.

3. RELATED WORK

Separately from chapter 2, this chapter focuses on some researches which recently explored the power of combining multi-architecture MapReduce implementations with a single and simple programming interface, thus more directly related to the objetives of our research.

We searched for MapReduce implementations of unified MapReduce programming interfaces through the concept of a DSL and also aimed at providing seamless optimization for different architectural levels. Important researches were found on improving existing MapReduce implementations for distributed systems, particularly Hadoop, in order to achieve high performance in MapReduce implementations at the single-node level. Nevertheless, no research was found on building a unified MapReduce programming interface for shared memory and distributed architectures through a DSL.

3.1 The Hone project

The Hone project [KGDL13, KGDL14], by Kumar et al., advocates that the exclusivity of large clusters for data processing can be re-evaluated, once shared-memory multi-core systems have increased in memory capacity and several common MapReduce applications do not deal with *petascale* datasets. Kumar et al. also observe the unfeasibility of adapting MapReduce implementations like Phoenix++ [TYK11] to be used with Hadoop, once they do not share a single programming interface and programming language. To achieve such integration, several reimplementations would be required in the Hadoop project.

Hone is aimed at providing an efficient MapReduce implementation for multi-core systems with the Java language, sharing thus compatibility with Hadoop MapReduce programming interface and requiring no code changes on MapReduce applications previously developed for distributed systems with Hadoop.

Experiments conducted in a server with dual Intel Xeon quad-core processors (E5620 2.4 GHz) and 128 GB RAM showed, however, that Phoenix++ still outperforms Hone, with speedup from 2X to 4X depending on the application.

3.2 Scale-up vs Scale-out for Hadoop

Appuswamy et al. [AGN+13] also argue about the increasing relevance of shared-memory multi-core environments for several MapReduce applications (section 3.1). Several categories of common MapReduce workloads fit well in the memory capacity of multi-core

environments, however current popular implementations, such as Hadoop, perform poorly on such environments.

Their research proposes a reimplementation of many Hadoop internals in order to improve the execution performance on multi-core systems in a completely seamless fashion.

Finally, authors present the results of experiments with a set of MapReduce applications in order to measure advantages of the refactored internals concerning execution time, financial costs of cloud resources and energy consumption. Nevertheless, no source code or enough information is provided to support comparison with Phoenix++.

3.3 The Azwraith project

Xiao et al. [XCZ11] argue that the disregarded efficiency of Hadoop for single-node level fails to exploit data locality, cache hierarchy and task parallelism for multi-core architectures. The JVM-based runtime causes extra objects creation and destroys overhead as well as memory footprint. The authors also argue that such resource wasting causes avoidable costs in *pay-as-you-go* cloud systems such as Amazon's Elastic MapReduce [1].

In order to increase Hadoop efficiency on multi-core, Xiao et al. proposes Azwraith, an integration of Hadoop with an efficient MapReduce runtime (namely Ostrich, section 2.6) for multi-core. Both Ostrich and Hadoop are adapted to conform to the same workflow and communication protocols, causing most components of Hadoop (e.g., TaskTracker, HDFS) to be left untouched, and avoiding thus impacts on existing Hadoop applications.

Experiments were conducted on a small-scale cluster with 1 master node and 6 slave nodes. Each machine was equipped with two AMD Opteron 12-core processors, 64 GB main memory and 4 SCSI hard drives. Each machine connected to the same switch through a 1Gb Ethernet link.

Azwraith gains a considerable speedup over Hadoop with different input sizes, ranging from 1.4x to 3.5x. Computation-oriented tasks like *Word Count* and *Linear Regression* gain larger speedup than the I/O-intensive applications such as *GigaSort*. With the support of the cache system, Azwraith gains a 1.43X to 1.55X speedup over a Azwraith instance without the cache scheme, and 2.06X to 2.21X over Hadoop.

3.4 Related work overview

As previously mentioned in this chapter, no research was found on building a unified MapReduce programming interface for shared memory and distributed architectures

[1] http://aws.amazon.com/pt/elasticmapreduce/

through a DSL. The researches described in this chapter and summarized in Table 3.1 aim at improving Hadoop, in order to achieve high performance in MapReduce implementations at the single-node level.

Table 3.1: Overview of Related Work

Work	Approach	Evaluation
Hone	A MapReduce implementation in Java optimized for multi-core systems with the Java language and compatible with Hadoop MapReduce programming interface	Phoenix++ outperforms Hone with speedup from 2x to 4X
Appuswamy	Reimplementation of many Hadoop internals in order to optimize execution time, financial costs of cloud resources and energy consumption.	No source code or enough information is provided to support comparison with Phoenix++
Azwraith	Integration of Ostrich and Hadoop, bringing Ostrich's multi-core optimizations to Hadoop internals without imposing a different MapReduce programming interface	Speedups from 1.4x to 3.5x over Hadoop, although still presenting less performance improvements compared Phoenix++

Concerning programming interface, such solutions make use of the same interface of Hadoop, making the underlying improvements transparent to the programmers. However, such improvements are still far from the performance achieved by Phoenix++ on shared memory multi-core level.

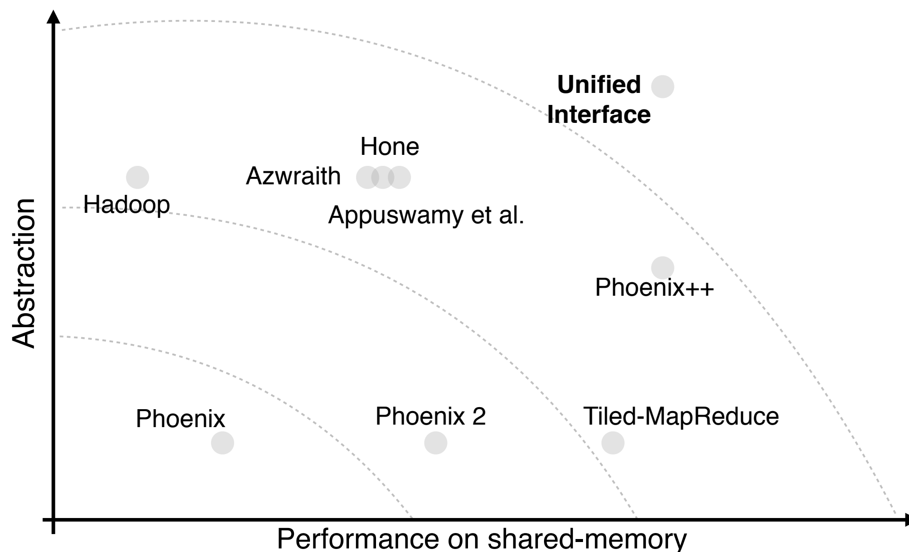


Figure 3.1: The relationship graph between abstraction and performance on the programming interface of analyzed researches.

Figure 3.1 also justifies our choices for generating MapReduce code for Hadoop and Phoenix++. These are the best alternatives for our design principles in terms of abstraction and performance.

4. UNIFIED MAPREDUCE PROGRAMMING INTERFACE

This chapter starts by presenting the justification for this research (section 4.1), which also covers the disadvantages of implementing MapReduce applications directly from the selected frameworks (section 4.1.1) and the advantages of a unified MapReduce programming interface for both shared-memory multi-core and distributed architectures (section 4.1.2). It continues with the research questions and the advocated hypothesis (section 4.2).

The proposed unified MapReduce programming interface is then described in detail (section 4.3), which starts by providing an overview of the transformation process in section 4.3.2. This section continues covering the interface's structure in section 4.3.1 and the interface's components in section 4.3.3, which in turn describes the equivalence among the interface's components and those from Hadoop and Phoenix++ code. Section 4.3.4 also describes some special components and code transformation rules for text processing, which comprehend most common MapReduce applications.

Section 4.3.5 describes performance mechanisms provided through Phoenix++ and Hadoop interfaces, which are also guaranteed through the proposed unified interface.

Section 4.3.6 describes programming complexities the unified interface is able to abstract and some very specialized features from Phoenix++ which are not supported since no possible abstraction was identified.

Finally, section 4.4 describes the devised code generator and its code transformation flow.

4.1 Justification

MapReduce pattern brings modularization and abstraction of data processing to an innovative level, what turns it to be an intuitive and simple model. In order to achieve high performance, it is mandatory an optimized resources usage for either shared-memory multi-core or distributed architectures. Current researches have exploited it in order to maximize optimization and have achieved considerable speedups. MapReduce solutions though have implied more complex interfaces and required more advanced skills from programmers.

Therefore, it is opportune to propose a unified MapReduce programming interface to minimize coding complexities and still optimally use the available resources by generating optimized code for consolidated solutions according to the architectural level, namely Hadoop (described in section 2.3) and Phoenix++ (described in section 2.7).

Complexities involving MapReduce programming with Hadoop and Phoenix++ are further described in section 4.1.1 and advantages of a unified MapReduce programming interface are discussed in section 4.1.2.

4.1.1 Required code by Hadoop and Phoenix++

The implementation of MapReduce applications with Hadoop requires the writing of two classes, which must inherit from *Mapper* and *Reducer* in order to override the respective methods which hold the *mapping* and *reduction* logics. Such methods also are required to throw specific Java and Hadoop *exceptions*. Phoenix++, in turn, requires the definition of a subclass of MapReduce or MapReduceSort class with functions for mapping and reduction, or the definition of a *combiner*.

In Phoenix++, the class definition also requires the parametrization of many configuration aspects, which may include explicit definition of data sizes, since optimized multicore implementations deal directly with memory allocators. Concerning optimized memory usage, Phoenix++ requires the programmer to specify both the *container* to manage the intermediate key-value pairs emitted during the *Map* phase (table 2.3) and the *combiner* to perform early reduction.

Besides such complex structures, inside which *mapping* and *reduction* logics are defined, there also are many required lines of code outside such structures and not related to the relevant MapReduce logic. For instance, both Hadoop and Phoenix++ always require many *imports* and *includes* which are thus totally repetitive among any kind of MapReduce application implemented with one of these two frameworks. Considering that a given *import* or *include* is required due to some variable type being used along the code, this also represents some unneeded code redundancy.

Appendix A shows the code for all sample applications, providing thus a complete and detailed visualization of the code required from Phoenix++ and Hadoop, as well as complexities imposed by the hosting languages C++ and Java respectively.

4.1.2 Advantages of a unified MapReduce Programming interface

The proposed unified MapReduce interface is not only able to significantly reduce the code required for MapReduce applications, by following transformation rules for equivalent Phoenix++ and Hadoop code, but also to keep performance optimizations and hereafter be evolved to generate code for different MapReduce frameworks.

Implementations for heterogeneous parallel architectures were considered and studied along our research (Appendix B), but were left as indication for future work (chapter 6).

Most of all, the interface provides code reuse among different architectural levels and the respective MapReduce implementations, being also able to be hereafter extended to comprehend new solutions and architectures.

4.2 Research questions and hypothesis

- **Q1** Is it possible for a unified MapReduce programming interface to generate effective code for Hadoop and Phoenix++?
- **Q2** Does the unified MapReduce programming interface significantly reduce programming effort?

The effectiveness questioned by *Q1* implies a working code for both Phoenix++ and Hadoop with negligible or no performance loss.

The significance questioned by *Q2* implies a significant reduction in lines of code and programming effort.

Experiments are based on hypothesis, which in turn must provide a way of being proved either false or true [Oat06].

The hypothesis *H1* and *H2* are proposed to answer the research question *Q1*, whereas *H3* is aimed to answer question *Q2*.

- **H1** The code resultant from the transformation rules produces the same output results with the same input datasets for code implemented directly from Phoenix++ and Hadoop.
- **H2** Both the Phoenix++ and Hadoop code, resultant from the transformation rules, execute with negligible or no performance loss.
- **H3** The lines of code and the programming effort required by the unified interface are significantly less than those required by Phoenix++ and Hadoop programming interfaces.

4.3 Proposed interface

It is proposed a unified MapReduce programming interface, in conjunction with code transformation rules for shared-memory multi-core and distributed architectures. The code transformation rules cover Phoenix++ (section 2.7) and Hadoop (section 2.3) MapReduce code, being up to the user to choose either one or the other according to the target architecture, whether shared-memory multi-core or distributed, in order to obtain high-performance with highly optimized resource usage.

The proposed interface is inspired by the syntax proposed by the Patterns-oriented Parallel Programming (POPP) model from Griebler et al. (section 2.9). Our interface however is not built over a third-part language as C or C++, being based on an own language

instead. The different programming languages (Java and C++) and the very specific syntaxes for Phoenix++ and Hadoop code led us to decide for an own language in order to maximize abstraction. A C++ programming interface provided by the Hadoop project, called Hadoop Pipes[1], was initially considered but later discarded due to absence of documentation and for looking as a discontinued project.

Moreover, the parallel patterns already comprehended by DSL-POPP, namely *Master/Slave* and *Pipeline*, allow the implementation of a wide diversity of algorithms, whereas a MapReduce approach is more peculiar. Algorithms adapted to MapReduce always deal with key/value pairs and its (*map* and *reduce*) blocks imply very specific code for managing the intermediate and final key/value pairs.

4.3.1 Unified Interfaces's structure

Our unified MapReduce programming interface's structure consists of an outer *@MapReduce* block and two inner *@Map* and *@Reduce* blocks, as detailed on listing 4.1 and grammar 4.1.

```
1 @MapReduce<NAME,
2             K_IN, V_IN,
3             K_OUT, V_OUT,
4             K_DIST>{
5
6   @Map(key, value){
7       \\ Map code logic
8   }
9
10  @Reduce(key, values){
11      \\ Reduce code logic
12  }
13}
```

Listing 4.1: Interface's structure.

The *@MapReduce* block always requires six parameters, namely *NAME*, *K_IN*, *V_IN*, *K_OUT*, *V_OUT* and *K_DIST*.

The *NAME* parameter is any user-defined name, which is used for identifying the MapReduce process and further transforming the code for Java and C++ classes, for Hadoop and Phoenix++ respectively.

The *K_IN*, *V_IN*, *K_OUT* and *V_OUT* parameters are used to define the *<key/-value>* input and output types, respectively. In other words, these parameters define which

[1] http://wiki.apache.org/hadoop/C%2B%2BWordCount

⟨*Map*⟩ ::= '`@Map(key, value){`' { ⟨*cmd*⟩* ⟨*EmitCall*⟩ ⟨*cmd*⟩* }+ '`}`'

⟨*Reduce*⟩ ::= '`@Reduce(key, values){`' { ⟨*cmd*⟩* ⟨*EmitCall*⟩ ⟨*cmd*⟩* }+ '`}`' | '`@SumReducer`' | '`@IdentityReducer`'

⟨*MapReduce*⟩ ::= '`@MapReduce<`' ⟨*mapreduce-params*⟩ '`>{`' ⟨*Map*⟩ ⟨*Reduce*⟩ '`}`'

Grammar 4.1: Structure's grammar

type of raw data is initially read by the MapReduce process and which type of reduced data is produced by it at the end.

Finally, the *K_DIST* parameter is used for defining the keys distribution (see table 2.3) of each specific MapReduce application. Section 2.7.1 describes how useful can this information be in order to employ more optimized data structure for intermediate *<key/value>* pairs. In other words, this parameter is critical for the generation of Phoenix++ optimized code. If not provided, **:** is assumed as the key distribution.

The inner blocks, *@Map* and *@Reduce*, must be programmed by the user in order to define the core logic of the given MapReduce application. The *@Map* block receives a *<key/value>* input pair from which to compute the *<key/value>* intermediate pairs. Finally, the *@Reduce* block receives all mapped values for each key, this is a *<key/values>* pair, and computes the final reduced *<key/value>* pair by key. Both blocks are also provided with an *@emit* function (grammar 4.2), which for *@Map* block represents the function to emit intermediate *<key/value>* pairs, whereas for *@Reduce* block represents the function to emit the final reduced value for a given key.

⟨*EmitCall*⟩ ::= '`emit(`' ⟨*key*⟩ '`,`' ⟨*value*⟩ '`)`'

Grammar 4.2: The emit function's grammar

One additional characteristic of the *@Reduce* block is that it can be replaced by a single *@SumReducer* directive with no block code (grammar 4.1), which indicates that a simple sum operation must be performed over all values of each key. Another option is the *@IdentityReducer* directive, which indicates that no reduction needs to be performed. Both default options are provided by Hadoop and Phoenix++, since these are the most common reduce logics for MapReduce applications. Nevertheless, whenever the provided default reducers do not fit the need, a customized reducer can be implemented, as demonstrated in the Listing 4.2 for a hypothetical multiplicand reducer.

```
1 @Reduce(key, values){
2     double product = 1
3
4     for(int i=0; i < length(values); i++)
5         product *= values[i]
```

```
6
7     emit(key, product)
8}
```

Listing 4.2: Multiplicand reducer example.

The code developed for *@Map* and *@Reduce* logics can employ many language components such as variable declarations and loops, further described in section 4.3.3. Every code declared inside a *@Map* or *@Reduce* block is private and will not be accessible from other code blocks. Global variables and functions may be used for data and operations to be accessible from different blocks.

4.3.2 Transformation process

Figure 4.1 provides an overview of the transformation process through which the transformation rules defined in sections 4.3.1, 4.3.3 and 4.3.4 are to be applied.

	Transformation process
First Stage	imports/includes
Second Stage	@MapReduce, global variables, @Map, @Reduce, @Type
Third Stage	unsolved keywords
Fourth Stage	variables types
Fifth Stage	functions

Figure 4.1: Transformation Process

- **First stage** - The process starts by generating all *imports/includes* always required by Phoenix++ and Hadoop applications. This stage follows the rules detailed in table 4.1.
- **Second stage** - The process then continues by transforming the *@MapReduce* block and its *@Map* and *@Reduce* inner blocks. At this same stage, custom types *@Type*

may have been provided by the programmer being then also transformed. At last, global variables, external to the *@MapReduce* block, may have also been defined by the programmer and are also transformed in this second stage. This stage follows the rules defined in listings 4.1, 4.2, 4.3 and 4.4.

- **Third stage** - This stage addresses the transformations of the unsolved keywords included by the second stage and defined in table 4.5.
- **Fourth stage** - This stage transforms the variable types defined in the blocks' signature and also internally to these blocks. The transformation rules for this stage are defined in table 4.2.
- **Fifth stage** - At last, the fifth stage transforms the functions defined internally to the MapReduce blocks and internally to custom types. The transformation rules for this final stage are defined in tables 4.3 and 4.4.

4.3.3 Interface components and transformation rules

This section provides a detailed description of the proposed interface's components, which comprehends variable types, built-in functions, flow control and, finally, the MapReduce blocks variants.

As each component is detailed, it is also described the correspondent code transformation for Hadoop and Phoenix++, which in turn is the base for the proposed code generator (section 4.4).

Imports and Includes

Concerning code transformation, the first important topic to describe is the set of *imports* (for Hadoop's Java code) and *includes* (for Phoenix++'s C++ code) always included in the generated code. Each one of these *imports*/*includes* is always required for any MapReduce application, being for this reason automatically included by our interface. Table 4.1 shows the two set of Java *imports* and C++ *includes*.

Additional *imports*/*includes* are inferred from the components used by the programmer along the code, as detailed in the following sections.

Variable types

The proposed interface is typed. This way, for the programmer to be able to define variables in any of the allowed contexts, it is provided a set of variable types.

Table 4.1: Java *imports* and C++ *includes* always required by MapReduce applications

Framework	import/include
Hadoop	import java.io.IOException;
Hadoop	import java.util.*;
Hadoop	import org.apache.hadoop.fs.FileSystem;
Hadoop	import org.apache.hadoop.fs.Path;
Hadoop	import org.apache.hadoop.conf.*;
Hadoop	import org.apache.hadoop.mapreduce.*;
Hadoop	import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
Hadoop	import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
Hadoop	import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
Hadoop	import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
Phoenix++	#include <sys/mman.h>
Phoenix++	#include <sys/stat.h>
Phoenix++	#include <fcntl.h>
Phoenix++	#include “map_reduce.h”
Phoenix++	#include “tbb/scalable_allocator.h”

Specifically for contexts that imply communication among MapReduce tasks, Hadoop requires a Writable[2] type, instead of primitive/wrapper[3] types, and Phoenix++ requires pointers. The proposed interface, in turn, prevents the programmer from this distinction, parsing the variable types to the right format (Writable, primitive or pointer) according to the context.

Table 4.2 describes the set of variable types, the Hadoop/Phoenix++ correspondent code transformation and any additional *include*/*import* required. For those types for which there are more than one possible Hadoop/Phoenix++ equivalent code, the code transformation rule considers the suitable option according to the context, whether it implies communication beyond the task's scope, as previously described.

Custom variable types

The programmer is also able to define custom types, which are translated to C++ *structs* for Phoenix++ and Java classes implementing the *Writable* interface for Hadoop. The resulting Java classes, particularly, include *getters* and *setters* methods, besides some other methods whose implementation is required by *Writable* interface.

Moreover, whenever a custom type is defined for input data in Hadoop, a complete implementation of a subclass of FileInputFormat[4] and another subclass of RecordReader[5] is required. It is particularly needed in order to instruct Hadoop on how to split and distribute the input data among *Map* tasks. Nonetheless, it causes applications developed with Hadoop to reach a considerable amount of code.

[2]https://hadoop.apache.org/docs/current/api/org/apache/hadoop/io/Writable.html
[3]http://docs.oracle.com/javase/tutorial/java/data/numberclasses.html
[4]https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/FileInputFormat.html
[5]https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/RecordReader.html

Table 4.2: Variable types, code transformation and additional *includes/imports*

Type	Phoenix++	Hadoop	C++ *includes*	Java *imports*
string	ph_word	String	<string.h> <ctype.h>	org.apache.hadoop.io.Text
text	ph_string	Text		
boolean	bool	BooleanWritable Boolean boolean		org.apache.hadoop.io.BooleanWritable
float	float	FloatWritable Float float		org.apache.hadoop.io.FloatWritable
double	double	DoubleWritable Double double		org.apache.hadoop.io.DoubleWritable
longdouble	long double	DoubleWritable Double double		org.apache.hadoop.io.DoubleWritable
short	int8_t intptr_t	ShortWritable Short short	<stdint.h>	org.apache.hadoop.io.ShortWritable
ushort	uint8_t uintptr_t	ShortWritable Short short	<stdint.h>	org.apache.hadoop.io.ShortWritable
int	int16_t intptr_t	IntWritable Integer int	<stdint.h>	org.apache.hadoop.io.IntWritable
uint	uint16_t uintptr_t	IntWritable Integer int	<stdint.h>	org.apache.hadoop.io.IntWritable
long	int32_t intptr_t	LongWritable Long long	<stdint.h>	org.apache.hadoop.io.LongWritable
ulong	uint32_t uintptr_t	LongWritable Long long	<stdint.h>	org.apache.hadoop.io.LongWritable
longlong	int64_t intptr_t	LongWritable Long long	<stdint.h>	org.apache.hadoop.io.LongWritable
ulonglong	uint64_t uintptr_t	LongWritable Long long	<stdint.h>	org.apache.hadoop.io.LongWritable
vector<T>	vector<T>	ArrayWritable List<T>		org.apache.hadoop.io.ArrayWritable

In such cases, the transformation rules provide a subclass of RecordReader which considers each line of a input file to represent a single instance of the given custom type, in which the values for its attributes are separated by space.

The sample Histogram, K-means and Linear Regression applications (sections A.3, A.4 and A.5) provide complete demonstrations of custom types and the consequently required additional code.

Functions

For a wide set of common operations, built-in functions are provided with the unified interface, as shown in Table 4.3. Loops and control flow structures follow the same pattern as in Java and C++, since their structures do not differ between these two languages.

Table 4.3: Built-in functions, code transformation and additional C++ *includes*

Function	Phoenix++	Hadoop	C++ *include*
toUpper(str)	for (uint64_t i = 0; i << s.len; i++) s.data[i] = toupper(s.data[i]);	str = str.toUpperCase()	<string.h> <ctype.h>
toLower(str)	for (uint64_t i = 0; i << s.len; i++) s.data[i] = tolower(s.data[i]);	str = str.toLowerCase()	<string.h> <ctype.h>
length(str)	strlen(str)	st.length()	<string.h> <ctype.h>
length(vector)	vector.size()	vector.size()	
rand()	rand()	Math.random()	
min_int()	std::numeric_limits<int>::min()	Integer.MIN_VALUE	<limits>
max_int()	std::numeric_limits<int>::max()	Integer.MAX_VALUE	<limits>
min_double()	std::numeric_limits<double>::min()	Double.MIN_VALUE	<limits>
max_double()	std::numeric_limits<double>::max()	Double.MAX_VALUE	<limits>
tokenize(value)	*Code in listing 4.9*	*Code in listing 4.10*	

Additionally, Table 4.4 shows the syntax for emiting intermediate and final key/value pairs in the contexts of *Map* and *Reduce* operations.

Table 4.4: Built-in functions for emit operations

Function	Context	Phoenix++	Hadoop
emit(k, v)	Map	emit_intermediate(out, k, v)	context.write(k, v)
emit(k, v)	Reduce	out.push_back(kv)	context.write(k, v)

Code blocks

The code for the MapReduce structure, which is defined as previously demonstrated in the listing 4.1, is proposed to be transformed to Phoenix++ and Hadoop as shown in the listings 4.3 and 4.4, respectively.

```
1 SPECIAL_TYPES
2
3 class NAME : public MapReduceSORT<NAME, V_IN, K_OUT, V_OUT,
4   CONTAINER<K_OUT, V_OUT,
5   COMBINER, K, HASH> >{
6       CLASS_VARS
7
8 public:
9
10      CONSTRUCTOR_FUNC
11
12      void map(data_type const& data, map_container& out) const{
13          \\ Map code logic
14      }
15
16      void reduce(key_type const& key, reduce_iterator const& values,
17                              std::vector<keyval>& out) const {
18          \\ Reduce code logic
19      }
20
21      SPLIT_FUNC
22
23      SORT_FUNC
24 }
```

Listing 4.3: MapReduce structure's code transformation for Phoenix++

```
1 public class NAME{
2
3   CLASS_VARS
4
5   public static class Map extends Mapper<K_IN, V_IN, K_OUT, V_OUT>{
6       @Override
7       public void map(K_IN key, V_IN value, Context context)
8               throws IOException, InterruptedException {
9             \\ Map code logic
10      }
11  }
```

```
12
13  public static class Reduce extends Reducer<K_IN, V_IN, K_OUT, V_OUT> {
14      @Override
15      public void reduce(K_IN key, Iterable<V_IN> values, Context context
            )
16              throws IOException, InterruptedException {
17          \\ Reduce code logic
18      }
19  }
20}
```

Listing 4.4: MapReduce structure's code transformation for Hadoop

The keywords in upper case are replaced as defined through the code written with the unified interface. However, as it can be observed, some keywords (i.e., *CONTAINER* have no direct correspondents in the unified interface, being dependent on the rules defined in table 4.5.

Table 4.5: Code transformation rules for indirect replacements.

Condition	Keyword	Replacement for Phoenix++	Replacement for Hadoop
K_DIST = "*:*"	CONTAINER	hash_container	
K_DIST = "*:k"	CONTAINER	fixed_hash_container	
K_DIST = "*:k"	K	k	
K_DIST = "1:1"	CONTAINER	hash_container	
K_OUT = int	HASH	std::tr1::hash<intptr_t>	
K_OUT = string	HASH	ph_word_hash	
K_OUT = string	SORT_FUNC	*Code in listing 4.7*	
K_OUT = string	SORT	Sort	
K_OUT = string or V_IN = text	SPECIAL_TYPES	*Code in listing 4.5*	
V_IN = text	CLASS_VARS	char* data; uint64_t data_size; uint64_t chunk_size; uint64_t splitter_pos;	private Text token = new Text();
V_IN = text	CONSTRUCTOR_FUNC	*Code in listing 4.6*	
V_IN = text	SPLIT_FUNC	*Code in listing 4.8*	
V_OUT = int	CLASS_VARS		private final static IntWritable one = new IntWritable(1);
@SumReducer	COMBINER	sum_combiner	
@IdentityReducer	COMBINER	one_combiner	

Finally, whenever a custom type is used for output values, Phoenix++ requires the implementation of a custom *associative_combiner*, which in turn is most likely to perform a simple sum for internal attributes of the custom type. By assuming this, the unified interface still allows *@SumReducer* directive even if output values are of a custom type. In this

case, the code transformation is defined for the correspondent *associative_combiner* and *Reducer* class of Phoenix++ and Hadoop respectively, as demonstrated in the K-means sample application (section A.4).

4.3.4 Special components for text processing

As it can be observed in Table 4.5, both the unified interface and the code transformation rules treat textual types in a special way, inferring many lines of code from it.

Some aspects particularly led us to such approach. The first one is that treating textual data with Phoenix++ requires a great amount of code, responsible for performing very elementary operations, like split input data and sort the final reduced keys. Moreover, even if the application logic greatly differ, the split and sort operations will remain the same.

The listings 4.5, 4.6, 4.7 and 4.8, show the code referenced in Table 4.5 for especially treating textual data.

```
1 struct ph_string {
2     char* data;
3     uint64_t len;
4 };
5
6 struct ph_word {
7     char* data;
8
9     ph_word() { data = NULL; }
10    ph_word(char* data) { this->data = data; }
11
12    bool operator<(ph_word const& other) const {
13        return strcmp(data, other.data) < 0;
14    }
15    bool operator==(ph_word const& other) const {
16        return strcmp(data, other.data) == 0;
17    }
18 };
19
20 struct ph_word_hash
21 {
22    size_t operator()(ph_word const& key) const
23    {
24        char* h = key.data;
25        uint64_t v = 14695981039346656037ULL;
```

```
26        while (*h != 0)
27            v = (v ^ (size_t)(*(h++))) * 1099511628211ULL;
28        return v;
29    }
30};
```

Listing 4.5: Transformation of special variable types for textual applications with Phoenix++

```
1   explicit NAME(char* _data, uint64_t length, uint64_t _chunk_size) :
2       data(_data), data_size(length), chunk_size(_chunk_size),
3           splitter_pos(0) {}
```

Listing 4.6: Constructor for textual applications with Phoenix++

```
1    bool sort(keyval const& a, keyval const& b) const
2    {
3        return a.val < b.val || (a.val == b.val && strcmp(a.key.data, b.
           key.data) > 0);
4    }
```

Listing 4.7: Sorting function for textual applications with Phoenix++

```
1    int split(ph_string& out)
2    {
3        if ((uint64_t)splitter_pos >= data_size)
4            return 0;
5
6        uint64_t end = std::min(splitter_pos + chunk_size, data_size);
7
8        while(end < data_size &&
9            data[end] != '␣' && data[end] != '\t' &&
10           data[end] != '\r' && data[end] != '\n')
11           end++;
12
13       out.data = data + splitter_pos;
14       out.len = end - splitter_pos;
15
16       splitter_pos = end;
17
18       return 1;
19   }
```

Listing 4.8: Split function for textual applications with Phoenix++

Finally, listings 4.9 and 4.10 show the code referenced in Table 4.3 for optimally splitting a *Map* task's textual input data.

```
uint64_t i = 0;
while(i < s.len)
{
    while(i < s.len && (s.data[i] < 'A' ||
           s.data[i] > 'Z'))
        i++;
    uint64_t start = i;
    while(i < s.len && ((s.data[i] >= 'A' &&
                         s.data[i] <= 'Z') ||
                         s.data[i] == '\''))
        i++;
    if(i > start)
    {
        s.data[i] = 0;
        ph_word token = { s.data+start };
        ...
    }
}
```

Listing 4.9: Optimized code for splitting a *Map* task's textual input data with Phoenix++

```
StringTokenizer tokenizer = new StringTokenizer(line);
while (tokenizer.hasMoreTokens()) {
   token.set(tokenizer.nextToken());
   ...
}
```

Listing 4.10: Optimized code for splitting a *Map* task's textual input data with Hadoop

4.3.5 Performance components

Both Phoenix++ and Hadoop provide good abstraction for the parallel and distributed processing, so that the programmer is able to concentrate his/her effort on *map* and *reduce* logics, which are sequential in nature. However, some components of Phoenix++ and Hadoop interfaces still play a performance role.

Phoenix++, particularly, requires the programmer to choose a *container* (table 2.3), which directly influences the memory consumption and the performance of reading and writing operations. For instance, a **:k* container must always be chosen whenever the number of keys is known in advance. In such cases, a **:** container could present much lower performance.

Hadoop, in turn, allows many performance configurations while the whole environment is configured. Afterwards, though, when the MapReduce application is indeed implemented, no such configurations are required. One good practice for better memory usage was identified as the instantiation of constant values as class variables, such as the *Writable* instance of the number 1 in MapReduce applications with numeric type for output values (sections A.1 and A.2).

These few performance controls still required or recommended at the programming stage are completely covered by the proposed unified interface, as defined in sections 4.3.3 and 4.3.4 and demonstrated with the sample applications in Appendix A.

4.3.6 Interface achievements and limitations

As it can be observed throughout sections 4.3.3 and 4.3.4, the proposed unified interface is able to abstract many aspects of Phoenix++ and Hadoop, by hiding from programmers complexities and prescriptions of both the frameworks and their hosting languages, C++ and Java.

For any application, *includes* and *imports* are totally abstracted. The same can be observed with object-orientation aspects, such as classes, inheritance and inner classes.

Both for Phoenix++ and Hadoop there are also some control objects, *map_container* and *Context* respectively, responsible for managing communication, which are required to be always parametrized to *map* and *reduce* functions. Such objects are also totally abstracted in the proposed interface.

For Hadoop, particularly, exceptions are also abstracted, as well as the *Writable* types which serve as wrappers for every primitive or custom types communicated through the process.

Finally, for Phoenix++, particularly, *sum combiners* for custom types are totally abstracted, as well as text processing specific types, functions, and hash.

However, while attempting to cover a wider range of MapReduce applications with Phoenix++, some limitations were identified.

The Phoenix++ project provides Matrix Multiplication (listing A.21) and Principal Component Analysis (listing A.22) sample applications, which demonstrate the applicability of *1:1* key distribution. However, both sample applications demonstrate the requirement of an explicit implementation of customized *split* logic, to which we could not devise an abstract representation suitable for both Phoenix++ and Hadoop, being thus not cover by our proposed interface.

Phoenix++ also enables programmers to implement a *locate* function, which is responsible for *data locality* optimization in NUMA architectures. Such function has also no

correspondent in Hadoop. Therefore, the proposed unified interface is not optimized for this category of architectures.

4.4 Devised code generator

A compiler and code generator for effectively applying the transformation rules detailed in section 4.3 through the process detailed in section 4.3.2 is indicated for future work (chapter 6). Nonetheless, this section describes the devised structure for a proper compiler and code generator, inspired on the work of Griebler et al. for the DSL-POPP [Gri12, GF13, GAF14].

It is aimed to be executable from the command line as *mapred-gen*. The first parameter would indicate the source file for the code written with the proposed interface, whereas optional parameters could be also included indicating whether to generate code specifically for Phoenix++ or Hadoop. If no target framework is defined, code would be separately generated for all supported frameworks. The syntax for the devised *mapred-gen* command is as follows.

mapred-gen input_source_file [–hadoop | –phoenixpp]

The compilation flow supporting both language recognition and code generation is demonstrated in figure 4.2

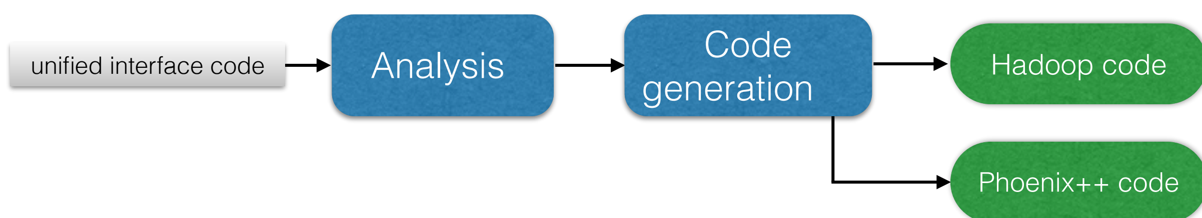


Figure 4.2: Compilation flow

Differently from DSL-POPP, our proposed unified interface is not based on a known host language, such as C or C++. A complete new language is proposed instead. However, since it is mainly aimed at developing *map* and *reduce* high-level logics, its components consist in a strict set, described in section 4.3, and a straightforward blocks structure is prescribed, as described in section 4.3.1.

The language recognition phase comprehends the interpretation through lexical, syntactic and semantic analysis. The lexical analysis validates the compliance with the proposed components described in sections 4.3.1, 4.3.3 and 4.3.4, then producing tokens.

The syntactic analysis uses the identified tokens to check the grammar language and report syntax errors. The semantic analysis in turn checks how components are disposed throughout the whole code.

Up to this point, the statements are saved and analyzed. This analysis verifies whether the provided code is syntactically correct and the overall organization of the code has full compliance with unified interface.

For an effective and proper implementation of a compiler and code generator, we indicate Lex and Yacc platforms.

5. EVALUATION

This chapter describes the approach used for evaluating the effectiveness of results and execution performance for both Phoenix++ and Hadoop code transformations, as well as the significance of the achieved programming abstraction and simplification (research questions and hypothesis are further detailed in section 4.2).

A set of applications with different peculiarities (section 5.1) was implemented with the proposed interface. Aimed at comparing it against the generated code in Phoenix++ and Hadoop, it was used the SLOCCount[1] suite, also used by Griebler et al. [GAF14] for the evaluation of DSL-POPP (section 2.9) and by a set of other researches [SET+09, LRCS14, HFB05, RGBMA06, KEH+09, VSJ+14, STM10]. SLOCCount is described in section 5.2.

Finally, the obtained results for performance evaluation are described and discussed in section 5.3, whereas simplicity and abstraction evaluation are addressed in section 5.4.

5.1 Sample applications

Five different applications with specific peculiarities, namely Word Count, Word Length, Histogram, K-means and Linear Regression, were implemented with the proposed interface and generated through the transformation rules for Phoenix++ and Hadoop.

The main peculiarity we looked for while choosing the sample applications was the key distribution (table 2.3). Word Count demonstrates the **:** distribution, whereas Word Length, Histogram, K-means and Linear Regression demonstrate the **:k* distribution. Additionally, other peculiarities are also covered by the selected sample applications, such as custom types and custom combiners.

As discussed in section 4.3.6, the Matrix Multiplication and Principal Component Analysis applications would fit the *1:1* distribution, however would also require more programming controls for Phoenix++ generated code beyond the abstraction aimed by the proposed interface and with no equivalent functionality in Hadoop.

The complete code for all the evaluation applications can be found in the Appendix A.

[1] http://www.dwheeler.com/sloccount/sloccount.html

5.2 SLOCCount

SLOCCount[1] is a software measurement tool, which counts the physical source lines of code (SLOC), ignoring empty lines and comments. It also estimates development time, cost and effort based on the original Basic COCOMO[2] model.

As previously mentioned, SLOCCount is a common tool for software measurement, also used by other similar researches such as DSL-POPP [GAF14].

Section 5.2.1 discusses the programming languages supported by SLOCCount and how it fits the evaluation of the unified programming interface.

Finally, section 5.2.2 describes how SLOCCount uses the COCOMO model for evaluating development time, cost and effort.

5.2.1 Programming Language support

The suite supports a wide range of both old and modern programming languages, which include C, C++, Objective-C, C#, Pascal, Java, Python, Ruby, among many others. For Phoenix++ and Hadoop, C++ and Java are naturally inferred by SLOCCount and thus used for measurement, however, for our proposed unified interface, there are only languages with similar syntax. Considering this, it was chosen C++ when using the interface with curly braces, and Python when using the interface structured by indentation.

5.2.2 COCOMO model

COCOMO provides three models for software measurement, namely *Basic*, *Intermediate* and *Detailed*. The *Basic* model performs the measurement based on the counted source lines of code and the *effort*, *schedule* and *personcost* parameters when provided. In contrast, the *Intermediate* and *Detailed* models require a wide range of configuration parameters in order to achieve a customized and more accurate measurement.

Besides the model selection, it is also suggested a value for each of the three previously mentioned parameters according to the categorize of the software being measured. There are three possible categories, namely *Organic*, *semidetached* and *embedded*, which are described in table 5.1.

Table 5.2 shows the suggested parameters according to COCOMO model and software category.

[2]http://www.dwheeler.com/sloccount/sloccount.html#cocomo

Table 5.1: Software category as suggested by COCOMO model

Category	Description
Organic	Relatively small software teams develop software in a highly familiar, in-house environment. It has a generally stable development environment, minimal need for innovative algorithms, and requirements can be relaxed to avoid extensive rework.
Semidetached	This is an intermediate step between organic and embedded. This is generally characterized by reduced flexibility in the requirements.
Embedded	The project must operate within tight (hard-to-meet) constraints, and requirements and interface specifications are often non-negotiable. The software will be embedded in a complex environment that the software must deal with as-is.

Table 5.2: Suggested parameters according to COCOMO model and software category

COCOMO Model	Software Category	Parameter	Factor	Exponent
Basic	Organic	Effort	2.4	1.05
Basic	Organic	Schedule	2.5	0.38
Basic	Semidetached	Effort	3.0	1.12
Basic	Semidetached	Schedule	2.5	0.35
Basic	Embedded	Effort	3.6	1.20
Basic	Embedded	Schedule	2.5	0.32
Intermediate	Organic	Effort	2.3	1.05
Intermediate	Organic	Schedule	2.5	0.38
Intermediate	Semidetached	Effort	3.0	1.12
Intermediate	Semidetached	Schedule	2.5	0.35
Intermediate	Embedded	Effort	2.8	1.20
Intermediate	Embedded	Schedule	2.5	0.32

For our evaluation, the COCOMO model and the *effort*, *schedule* and *personcost* parameters were kept with the default values since for our evaluation the accuracy of these measurements is less relevant than the proportion of results among the languages. In other words, the default options of SLOCCount already provide a sufficient measurement for comparing the code programmed with the proposed unified interface against the generated code for Phoenix++ and Hadoop.

5.3 Performance evaluation

The workload for Word Count and Word Length was a 2Gb text file, for Histogram, a 1.41 Gb image with 468,750,000 pixels and for Linear Regression the workload was a 500Mb file. For Kmeans, no input file is required, since number of points, means, clusters and dimensions are parametrized through command line or assumed to the default values of 100,000, 100, 3 and 1,000, respectively, which were considered for our tests. All workloads are available at the Phoenix++ project's on-line repository[3].

For performance evaluation of Phoenix++ generated code compared to original code (developed directly with Phoenix++) for the sample applications (appendix A), we used a multi-core system equipped with a 2.3 GHz Intel Core i7 processor with Hyper-Threading, which sums 4 cores (8 threads), and 16Gb of DRAM.

In order to obtain the arithmetic means (used in table 5.3 and figure 5.1), 30 execution times were collected for each sample application.

Table 5.3 shows the mean execution time for each sample application for the code transformed from our proposed unified interface and for the code developed directly from Phoenix++. Figure 5.1 graphically demonstrates this same measurements.

Table 5.3: Mean execution time in seconds for original and generated Phoenix++ code

	WC	WL	Histogram	Kmeans	LR
Original	5.38	4.02	2.83	5.98	0.62
Generated	5.37	3.99	2.87	6.09	0.63
Difference	-0.27%	-0.9%	1.4%	1.7%	0.3%

For performance evaluation of Hadoop generated code compared to original code (developed directly with Hadoop), we used a Dell PowerEdge M1000e distributed system equipped with 16 Blades Dell PowerEdge M610, each node with a 2.4 GHz Intel Xeon Six-Core E5645 processor with Hyper-Threading, which sums 12 cores (24 threads), and 24Gb of DRAM. The cluster sums 192 cores (384 threads). Nodes are linked through 2 Gigabit-Ethernet networks and 2 InfiniBand networks.

[3]https://github.com/kozyraki/phoenix

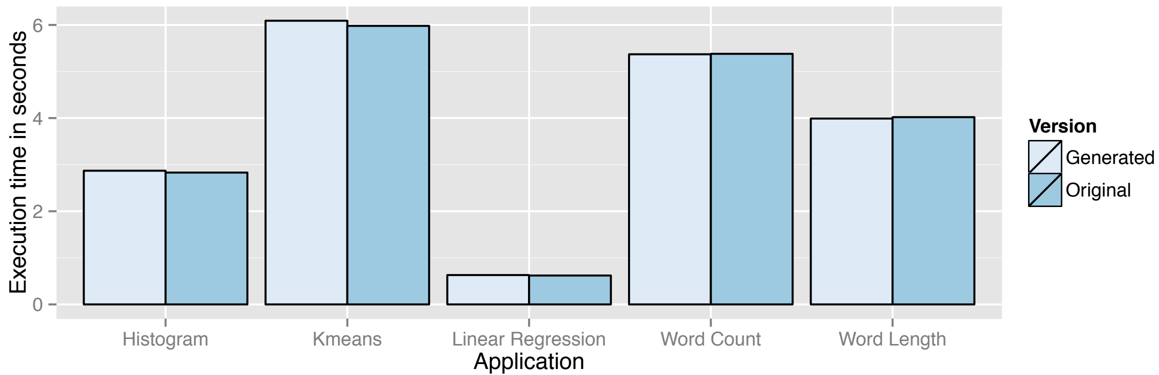


Figure 5.1: Mean execution time in seconds for original and generated Phoenix++ code

For the evaluation tests, 8 nodes were allocated and 30 execution times were collected for each sample application in order to obtain the arithmetic means used in table 5.4 and figure 5.2. The whole iteration with the cluster was performed through the work of Neves et al. [**?**], whose source code is available at github[4].

Table 5.4 shows the mean execution time for each sample application for the code transformed from our proposed unified interface and for the code developed directly from Hadoop. Figure 5.2 graphically demonstrates this same measurements.

Table 5.4: Mean execution time in seconds for original and generated Hadoop code

	WC	WL	Histogram	Kmeans	LR
Original	36.24	26.36	21.87	51.36	5.97
Generated	37.22	26.48	22.42	50.59	6.01
Difference	2.63%	0.45%	2.45%	-1.52%	0.76%

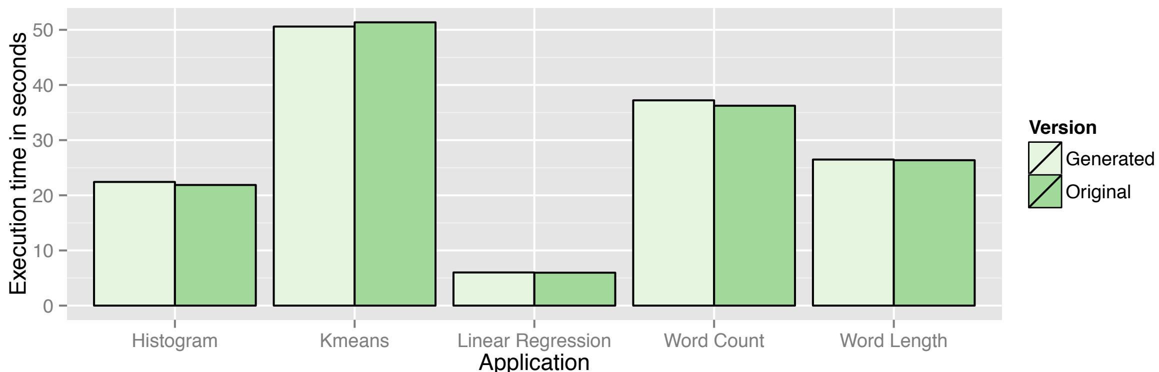


Figure 5.2: Mean execution time in seconds for original and generated Hadoop code

Through tables 5.3 and 5.4 and figures 5.1 and 5.2 it is possible to visualize the negligible difference between the execution time of the generated and original versions for

[4]https://github.com/mvneves/hadoop-deploy

the two frameworks, being it below the 3% aimed by the hypothesis *H2*. Moreover, output results were the same between the two versions for all sample applications, confirming hypothesis *H1* as well.

Performance losses are considerably avoided as a direct result of the effective coverage of performance components by the transformation rules, as described in section 4.3.5.

5.4 SLOC and effort evaluation

The SLOC and effort evaluation was performed considering the proposed unified interface in a first version without curly braces, which makes it close to a Python-like syntax, and a second version with curly braces, thus keeping it close to a C-like syntax. This second version is aimed at minimizing the bias due to some SLOC being considered as valid lines of code while only holding some opening or closing curly brace.

Tables 5.5 and 5.6 present the results for the version without curly braces for SLOC count and cost estimate, respectively.

Tables 5.7 and 5.8, in turn, present the results for the version with curly braces for SLOC count and cost estimate, respectively.

Table 5.5: SLOC count for the version without curly braces

Application	**Phoenix++**	**Hadoop**	**Unified Interface**	**Reduction compared to Phoenix++**	**Reduction compared to Hadoop**
WordCount	89	27	6	93.26%	77.78%
WordLength	95	33	11	88.42%	66.67%
Histogram	22	170	7	68.18%	95.88%
K-means	98	244	52	46.94%	78.69%
Linear Regression	31	171	16	48.39%	90.64%
	67	**129**	**18.4**	**69.04%**	**81.93%**

Table 5.6: Cost estimate for the version without curly braces

Application	**Phoenix++**	**Hadoop**	**Unified Interface**	**Reduction compared to Phoenix++**	**Reduction compared to Hadoop**
WordCount	$2,131.00	$609.00	$126.00	94.09%	79.31%
WordLength	$2,282.00	$752.00	$237.00	89.61%	68.48%
Histogram	$491	$4,204.00	$148.00	69.86%	96.48%
K-means	$2,357.00	$6,143.00	$1,212.00	48.58%	80.27%
Linear Regression	$704.00	$4,229.00	$352.00	50.00%	91.68%
	$1,593.00	**$3,187.40**	**$415.00**	**70.43%**	**83.24%**

Table 5.7: SLOC count for the version with curly braces

Application	**Phoenix++**	**Hadoop**	**Unified Interface**	**Reduction compared to Phoenix++**	**Reduction compared to Hadoop**
WordCount	89	27	8	91.01%	70.37%
WordLength	95	33	14	85.26%	57.58%
Histogram	22	170	9	59.09%	94.71%
K-means	98	244	57	41.84%	76.64%
Linear Regression	31	171	18	41.94%	89.47%
	67	**129**	**21.2**	**63.83%**	**77.75%**

Table 5.8: Cost estimate for the version with curly braces

Application	**Phoenix++**	**Hadoop**	**Unified Interface**	**Reduction compared to Phoenix++**	**Reduction compared to Hadoop**
WordCount	$2,131.00	$609.00	$170.00	92.02%	72.09%
WordLength	$2,282.00	$752.00	$306.00	86.59%	59.31%
Histogram	$491	$4,204.00	$192.00	60.90%	95.43%
K-means	$2,357.00	$6,143.00	$1,334.00	43.40%	78.28%
Linear Regression	$704.00	$4,229.00	$398.00	43.47%	90.59%
	$1,593.00	**$3,187.20**	**$480.00**	**65.28%**	**79.14%**

Figures 5.3, 5.4, 5.5 and 5.6 show a graphical representation for the reduced SLOC count and cost estimate, according to the tables 5.5, 5.6, 5.7 and 5.8, respectively.

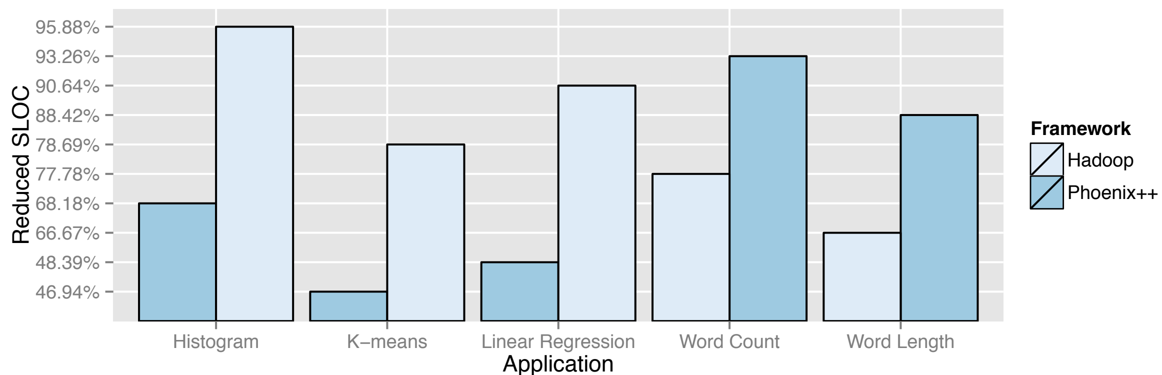


Figure 5.3: SLOC count for the interface version without curly braces

By comparing the obtained measurements for SLOC and Cost of the same interface version it is possible to notice that the difference between these two measurements is negligible, which confirms the approach used by COCOMO model (section 5.2.2).

The difference among versions with curly braces and without it is little, but not negligible. There is a subtle SLOC reduction while structuring code by indentation, instead of

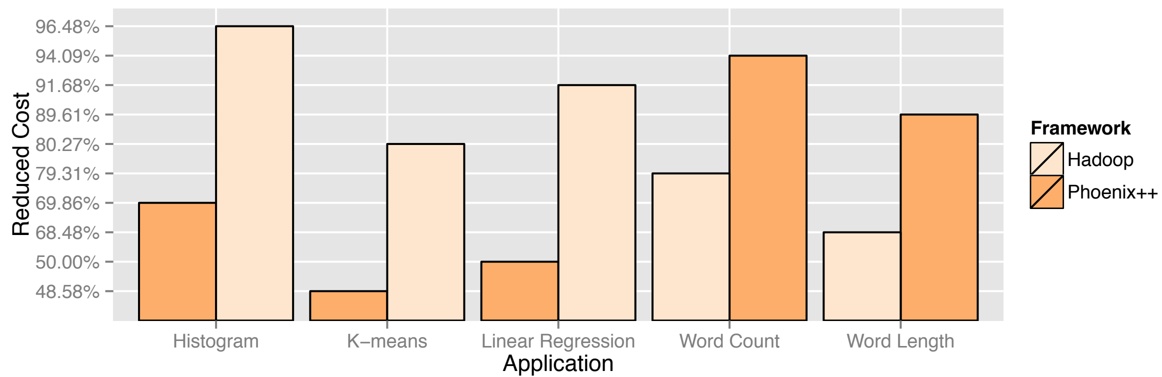


Figure 5.4: Cost estimate for the interface version without curly braces

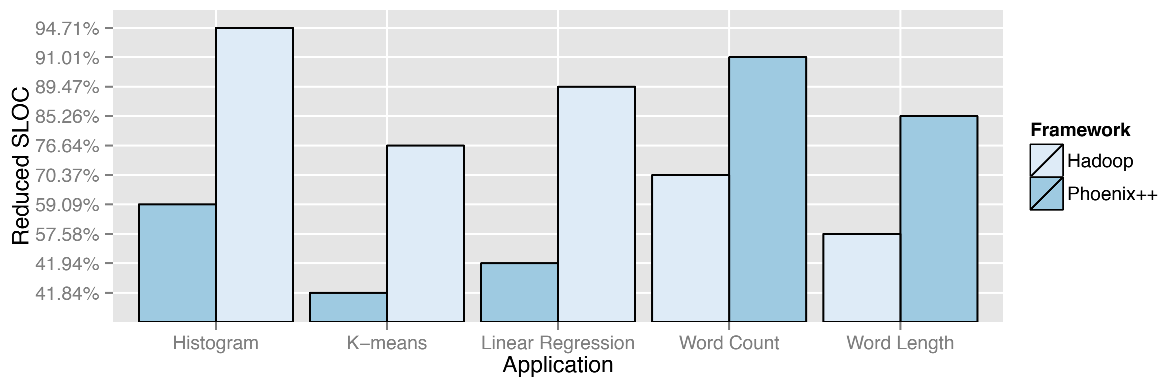


Figure 5.5: SLOC count for the interface version with curly braces

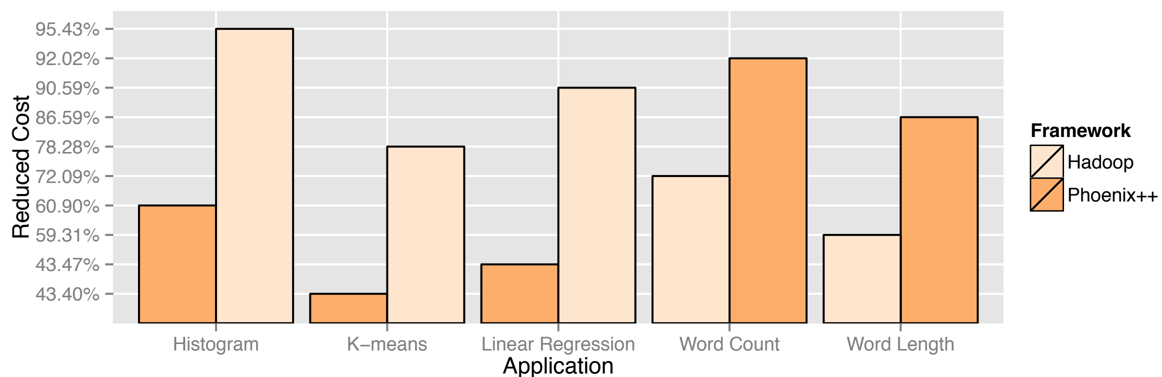


Figure 5.6: Cost estimate for the interface version with curly braces

using curly braces. Nevertheless both interface versions show great SLOC a effort reduction for all sample applications when compared to Phoenix++ and Hadoop.

A significant SLOC reduction can be observed for *Word Count* and *Word Length* applications compared to Phoenix++ code, which take advantage of the specific components for text processing, covered in section 4.3.4.

Compared to Hadoop, the *Histogram*, *K-means* and *Linear Regression* applications achieved greater SLOC reduction mainly for the amount of code required to create custom types in Java, as detailed in section 4.3.3.

K-means also takes advantage over Phoenix++ by completely avoiding code for custom combiner, however many functions are required for the custom *CPoint* type, which causes little SLOC reduction with the unified interface.

6. CONCLUSION AND FUTURE WORK

The ease of programming and the abstraction intended by the MapReduce model are impaired by some of its implementations, particularly for more low-level architectures.

From selecting Phoenix++ and Hadoop as the state-of-the-art solutions for shared-memory multicore and distributed architectures, respectively, this work proposes a solution for keeping the MapReduce programming easy without losing the performance optimizations provided by these selected implementations. Such objective is achieved through a unified MapReduce programming interface, proposed through a comprehensive set of transformation rules among a proposed interface and the correspondent manually generated code for Phoenix++ and Hadoop.

Concerning the first research question (section 4.2), the transformation rules are effective in covering from custom types to custom functions, custom combiners, default reducers, different key distributions and text processing, covering thus all components needed from the selected sample applications. The same output results are guaranteed and performance losses are successfully avoided (difference of less than 3%, as detailed in section 5.3).

And concerning the second research question, special skills are not required from programmers while developing applications for different architectural levels, being the target architecture totally abstracted. The evaluation demonstrates that a code and a development effort reduction from 41.84% to 96.48% can be achieved (section 5.4). The highest reduction is attributed to code required by Hadoop for custom types, being followed by text processing applications, which for Phoenix++ requires a wide set of mechanisms whose need is then identified in advance by the proposed transformation rules, being it transparent while developing with the proposed unified interface.

Some advantages and main contributions are the code reuse among different architectures and the possibility to expand the coverage of the transformation rules to other MapReduce solution, such as Grex for GPGPU (described in Appendix B). Besides that, the programmer is required to learn a single language and programming interface.

Some limitations were also identified while addressing some more specific applications for Phoenix++, such as Matrix Multiplication and PCA. The unified interface is also not compatible to the NUMA support provided by Phoenix++. These limitations are further described in section 4.3.6.

As future work we identified the effective construction of the compiler and code generator based on the proposed transformation rules, the compatibility with other MapReduce solutions (such as Grex) and the extension of DSL-POPP with the MapReduce pattern.

REFERENCES

[AGN+13] Appuswamy, R.; Gkantsidis, C.; Narayanan, D.; Hodson, O.; Rowstron, A. “Scale-up vs Scale-out for Hadoop: Time to Rethink?” In: Proceedings of the Annual Symposium on Cloud Computing, 2013, pp. 20:1–20:13.

[BK13] Basaran, C.; Kang, K.-D. “Grex: An Efficient MapReduce Framework for Graphics Processing Units”, *Transactions on Parallel Distributed Computing*, vol. 73–4, May 2013, pp. 522–533.

[BKX13] Basaran, C.; Kang, K.-D.; Xie, M. “Moin: A Multi-GPU MapReduce Framework”. In: Proceedings of the International Symposium on MapReduce and Big Data Infrastructure, 2013.

[CA12] Chen, L.; Agrawal, G. “Optimizing MapReduce for GPUs with Effective Shared Memory Usage”. In: Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing, 2012, pp. 199–210.

[CC13] Chen, R.; Chen, H. “Tiled-MapReduce: Efficient and Flexible MapReduce Processing on Multicore with Tiling”, *ACM Transactions on Architecture and Code Optimization*, vol. 10–1, April 2013, pp. 3:1–3:30.

[CCA+10] Condie, T.; Conway, N.; Alvaro, P.; Hellerstein, J. M.; Elmeleegy, K.; Sears, R. “MapReduce Online”. In: Proceedings of the Symposium on Networked Systems Design and Implementation, 2010, pp. 21–21.

[CCZ10] Chen, H.; Chen, R.; Zang, B. “Tiled-MapReduce: Optimizing Resource Usages of Data-Parallel Applications on Multicore with Tiling”. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2010, pp. 523–534.

[CM95] Coleman, S.; McKinley, K. S. “Tile Size Selection Using Cache Organization and Data Layout”, *ACM Transactions on SIGPLAN Not.*, vol. 30–6, June 1995, pp. 279–290.

[CSW13] Cao, C.; Song, F.; Waddington, D. “Implementing a High-Performance Recommendation System Using Phoenix++”. In: Proceedings of the International Conference for Internet Technology and Secured Transactions, 2013.

[DG08] Dean, J.; Ghemawat, S. “MapReduce: Simplified Data Processing on Large Clusters”, *ACM Transactions on Commun.*, vol. 51–1, Jan 2008, pp. 107–113.

[EHHB14] El-Helw, I.; Hofman, R.; Bal, H. E. “Glasswing: Accelerating Mapreduce on Multi-core and Many-core Clusters”. In: Proceedings of the International Symposium on High-performance Parallel and Distributed Computing, 2014, pp. 295–298.

[Fow10] Fowler, M. “Domain-Specific Languages”. Boston, USA: Addison-Wesley, 2010, 460p.

[GAF14] Griebler, D.; Adornes, D.; Fernandes, L. G. “Performance and Usability Evaluation of a Pattern-Oriented Parallel Programming Interface for Multi-Core Architectures”. In: The 26th International Conference on Software Engineering & Knowledge Engineering, 2014, pp. 25–30.

[GF13] Griebler, D.; Fernandes, L. G. “Towards a Domain-Specific Language for Patterns-Oriented Parallel Programming”. In: Programming Languages - 17th Brazilian Symposium - SBLP, 2013, pp. 105–119.

[Gho11] Ghosh, D. “DSLs in Action”. Stamford, CT, USA: Manning publications Co., 2011, 377p.

[GL11] Gordon, A. W.; Lu, P. “Elastic Phoenix: Malleable Mapreduce for Shared-Memory Systems”. In: Proceedings of the International Conference on Network and Parallel Computing, 2011, pp. 1–16.

[Gri12] Griebler, D. J. “Proposta de uma Linguagem Específica de Domínio de Programação Paralela Orientada a Padrões Paralelos: Um Estudo de Caso Baseado no Padrão Mestre/Escravo para Arquiteturas Multi-Core”, Master’s Thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, 2012, 168p.

[HCC+10] Hong, C.; Chen, D.; Chen, W.; Zheng, W.; Lin, H. “MapCG: Writing Parallel Program Portable Between CPU and GPU”. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2010, pp. 217–226.

[HFB05] Hertz, M.; Feng, Y.; Berger, E. D. “Garbage Collection Without Paging”. In: Proceedings of the Conference on Programming Language Design and Implementation, 2005, pp. 143–153.

[HFL+08] He, B.; Fang, W.; Luo, Q.; Govindaraju, N. K.; Wang, T. “Mars: A MapReduce Framework on Graphics Processors”. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2008, pp. 260–269.

[JA12] Jiang, W.; Agrawal, G. “MATE-CG: A Map Reduce-Like Framework for Accelerating Data-Intensive Computations on Heterogeneous Clusters”. In: Proceedings of the International Parallel and Distributed Processing Symposium, 2012, pp. 644–655.

[JM11] Ji, F.; Ma, X. “Using Shared Memory to Accelerate MapReduce on Graphics Processing Units”. In: Proceedings of the International Parallel & Distributed Processing Symposium, 2011, pp. 805–816.

[JRA10] Jiang, W.; Ravi, V. T.; Agrawal, G. “A Map-Reduce System with an Alternate API for Multi-core Environments”. In: Proceedings of the International Conference on Cluster, Cloud and Grid Computing, 2010, pp. 84–93.

[KAO05] Kongetira, P.; Aingaran, K.; Olukotun, K. “Niagara: A 32-Way Multithreaded Sparc Processor”, *IEEE Transactions on Micro*, vol. 25–2, March 2005, pp. 21–29.

[KEH+09] Klein, G.; Elphinstone, K.; Heiser, G.; Andronick, J.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; Sewell, T.; Tuch, H.; Winwood, S. “seL4: Formal Verification of an OS Kernel”. In: Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles, 2009, pp. 207–220.

[KGDL13] Kumar, K. A.; Gluck, J.; Deshpande, A.; Lin, J. “Hone: "Scaling Down" Hadoop on Shared-Memory Systems”, *Transactions on the International Conference on Very Large Data Bases*, vol. 6–12, August 2013, pp. 1354–1357.

[KGDL14] Kumar, K. A.; Gluck, J.; Deshpande, A.; Lin, J. “Optimization Techniques for "Scaling Down" Hadoop on Multi-Core, Shared-Memory Systems”. In: Proceedings of the International Conference on Extending Database Technology, 2014, pp. 13–24.

[Lam10] Lam, C. “Hadoop in Action”. Greenwich, CT, USA: Manning Publications Co., 2010, 1st ed..

[LRCS14] Li, K.; Reichenbach, C.; Csallner, C.; Smaragdakis, Y. “Residual Investigation: Predictive and Precise Bug Detection”, *ACM Transactions on Software Engineering and Methodology*, vol. 24–2, December 2014, pp. 7:1–7:32.

[McC10] McCool, M. D. “Structured Parallel Programming with Deterministic Patterns”. In: Proceedings of the Conference on Hot Topics in Parallelism, 2010, pp. 5–5.

[MSM04] Mattson, T.; Sanders, B.; Massingill, B. “Patterns for Parallel Programming”. Addison-Wesley Professional, 2004, first ed..

[NBGS08] Nickolls, J.; Buck, I.; Garland, M.; Skadron, K. “Scalable Parallel Programming with CUDA”, *ACM Transactions on Queue*, vol. 6–2, March 2008, pp. 30–53.

[Ngu07] Nguyen, H. “GPU GEMS 3”. Addison-Wesley Professional, 2007.

[Oat06] Oates, B. “Researching Information Systems and Computing”. SAGE, 2006.

[RGBMA06] Robles, G.; Gonzalez-Barahona, J. M.; Michlmayr, M.; Amor, J. J. “Mining Large Software Compilations over Time: Another Perspective of Software Evolution”. In: Proceedings of the International Workshop on Mining Software Repositories, 2006, pp. 3–9.

[RRP+07] Ranger, C.; Raghuraman, R.; Penmetsa, A.; Bradski, G.; Kozyrakis, C. “Evaluating MapReduce for Multi-core and Multiprocessor Systems”. In: Proceedings of the International Symposium on High Performance Computer Architecture, 2007, pp. 13–24.

[SET+09] Shinagawa, T.; Eiraku, H.; Tanimoto, K.; Omote, K.; Hasegawa, S.; Horie, T.; Hirano, M.; Kourai, K.; Oyama, Y.; Kawai, E.; Kono, K.; Chiba, S.; Shinjo, Y.; Kato, K. “BitVisor: A Thin Hypervisor for Enforcing I/O Device Security”. In: Proceedings of the International Conference on Virtual Execution Environments, 2009, pp. 121–130.

[SO11] Stuart, J. A.; Owens, J. D. “Multi-GPU MapReduce on GPU Clusters”. In: Proceedings of the International Parallel & Distributed Processing Symposium, 2011, pp. 1068–1079.

[SPA10] “Sun Sparc Enterprise T5440 Server Architecture”, 2010.

[STM10] Siefers, J.; Tan, G.; Morrisett, G. “Robusta: Taming the Native Beast of the JVM”. In: Proceedings of the Conference on Computer and Communications Security, 2010, pp. 201–211.

[TS12] Tiwari, D.; Solihin, Y. “Modeling and Analyzing Key Performance Factors of Shared Memory MapReduce”. In: Proceedings of the Second International Workshop on MapReduce and Its Applications, 2012, pp. 1306–1317.

[TYK11] Talbot, J.; Yoo, R. M.; Kozyrakis, C. “Phoenix++: Modular MapReduce for Shared-Memory Systems”. In: Proceedings of the International Workshop on MapReduce and Its Applications, 2011, pp. 9–16.

[Ven09] Venner, J. “Pro Hadoop”. Berkely, CA, USA: Apress, 2009, 1st ed..

[VSJ+14] Vazou, N.; Seidel, E. L.; Jhala, R.; Vytiniotis, D.; Peyton-Jones, S. “Refinement Types for Haskell”. In: Proceedings of the International Conference on Functional Programming, 2014, pp. 269–282.

[WD11] Wittek, P.; Darányi, S. "Leveraging on High-Performance Computing and Cloud Technologies in Digital Libraries: A Case Study". In: Proceedings of the Third International Conference on Cloud Computing Technology and Science, 2011, pp. 606–611.

[Whi09] White, T. "Hadoop: The Definitive Guide". O'Reilly Media, 2009, original ed..

[XCZ11] Xiao, Z.; Chen, H.; Zang, B. "A Hierarchical Approach to Maximizing MapReduce Efficiency". In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2011, pp. 167–168.

[YRK09] Yoo, R. M.; Romano, A.; Kozyrakis, C. "Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System". In: Proceedings of the International Symposium on Workload Characterization, 2009, pp. 198–207.

APPENDIX A – EVALUATED APPLICATIONS AND GENERATED CODE

This appendix describes and shows the programming code for each sample application in four versions: Two implementations with the proposed unified MapReduce programming interface, one with curly braces, following a C-like syntax, and another without it, following an indented syntax similar to Python, and the code for Phoenix++ and Hadoop according to the transformation rules defined in sections 4.3.1, 4.3.3 and 4.3.4.

It is important to observe that the *generated* terminology used here means a manual process performed accordingly to the transformation rules defined in Chapter 4. The effective compiler and code generator are indicated as future work in Chapter 6.

At the end, sections A.6 and A.7 are dedicated to the Matrix Multiplication and PCA applications, with code provided by the Phoenix++ project. These two applications demonstrate the need of implementing specific *split* logic for some applications, which has no correspondent component in our proposed interface.

A.1 Word Count

The Word Count application consists of a MapReduce implementation for counting the total number of occurrences of each word in a given text file.

It is a very common application for MapReduce evaluating, once it is not possible to know in advance how many keys (distinct words) will be mapped.

In the *map* phase, each *map* execution receives a chunk of the input data, splits it in order to obtain the words, and emits an intermediate key/value pair for each word, in which the key is the word found, and the value is 1, indicating 1 occurrence found.

In the *reduce* phase, all values of a given key are summed up, which means summing up all values 1 of each given word producing thus the total number of occurrences for each distinct word in the input text file.

```
1 @MapReduce<WordCountMR, long, text, string, int>{
2
3     @Map(key, value){
4         toupper(value)
5         tokenize(value)
6             emit(token, 1)
7     }
8
9     @SumReducer
```

```
10}
```

Listing A.1: Implementation with proposed interface with curly braces.

```
1@MapReduce<WordCountMR, long, text, string, int>
2
3    @Map(key, value)
4        toupper(value)
5        tokenize(value)
6            emit(token, 1)
7
8    @SumReducer
```

Listing A.2: Implementation with proposed interface without curly braces.

```
1#include <sys/mman.h>
2#include <sys/stat.h>
3#include <fcntl.h>
4#include <string.h>
5#include <ctype.h>
6
7#include "map_reduce.h"
8
9struct ph_string {
10    char* data;
11    uint64_t len;
12};
13
14struct ph_word {
15    char* data;
16
17    ph_word() { data = NULL; }
18    ph_word(char* data) { this->data = data; }
19
20    bool operator<(ph_word const& other) const {
21        return strcmp(data, other.data) < 0;
22    }
23    bool operator==(ph_word const& other) const {
24        return strcmp(data, other.data) == 0;
25    }
26};
27
28struct ph_word_hash
29{
```

```
30    size_t operator()(ph_word const& key) const
31    {
32        char* h = key.data;
33        uint64_t v = 14695981039346656037ULL;
34        while (*h != 0)
35            v = (v ^ (size_t)(*(h++))) * 1099511628211ULL;
36        return v;
37    }
38};
39
40class WordCountMR : public MapReduceSort<WordCountMR, ph_string, ph_word
     , uint64_t, hash_container<ph_word, uint64_t, sum_combiner,
     ph_word_hash
41#ifdef TBB
42    , tbb::scalable_allocator
43#endif
44> >
45{
46    char* data;
47    uint64_t data_size;
48    uint64_t chunk_size;
49    uint64_t splitter_pos;
50public:
51    explicit WordCountMR(char* _data, uint64_t length, uint64_t
        _chunk_size) :
52        data(_data), data_size(length), chunk_size(_chunk_size),
53            splitter_pos(0) {}
54
55    void map(data_type const& s, map_container& out) const
56    {
57        for (uint64_t i = 0; i < s.len; i++)
58        {
59            s.data[i] = toupper(s.data[i]);
60        }
61
62        uint64_t i = 0;
63        while(i < s.len)
64        {
65            while(i < s.len && (s.data[i] < 'A' || s.data[i] > 'Z'))
66                i++;
67            uint64_t start = i;
```

```
68            while(i < s.len && ((s.data[i] >= 'A' && s.data[i] <= 'Z')
                 || s.data[i] == '\''))
69                i++;
70            if(i > start)
71            {
72                s.data[i] = 0;
73                ph_word token = { s.data+start };
74                emit_intermediate(out, token, 1);
75            }
76        }
77    }
78
79    int split(ph_string& out)
80    {
81        if ((uint64_t)splitter_pos >= data_size)
82        {
83            return 0;
84        }
85
86        uint64_t end = std::min(splitter_pos + chunk_size, data_size);
87
88        while(end < data_size &&
89            data[end] != '␣' && data[end] != '\t' &&
90            data[end] != '\r' && data[end] != '\n')
91            end++;
92
93        out.data = data + splitter_pos;
94        out.len = end - splitter_pos;
95
96        splitter_pos = end;
97
98        return 1;
99    }
100
101   bool sort(keyval const& a, keyval const& b) const
102   {
103       return a.val < b.val || (a.val == b.val && strcmp(a.key.data, b.
              key.data) > 0);
104   }
105 };
```

Listing A.3: Code for Phoenix++.

```
1 import java.io.IOException;
2 import java.util.*;
3
4 import org.apache.hadoop.fs.Path;
5 import org.apache.hadoop.conf.*;
6 import org.apache.hadoop.io.*;
7 import org.apache.hadoop.mapreduce.*;
8 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
9 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
10 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
11 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
12 import org.apache.hadoop.mapreduce.lib.reduce.IntSumReducer;
13
14 public class WordCountMR {
15
16     public static class Map extends Mapper<LongWritable, Text, Text,
           IntWritable> {
17
18         private final static IntWritable one = new IntWritable(1);
19         private Text word = new Text();
20
21         @Override
22         public void map(LongWritable key, Text value, Context context)
23                 throws IOException, InterruptedException {
24             String line = value.toString();
25             StringTokenizer tokenizer = new StringTokenizer(line);
26             while (tokenizer.hasMoreTokens()) {
27                 word.set(tokenizer.nextToken());
28                 context.write(word, one);
29             }
30         }
31     }
32 }
```

Listing A.4: Code for Hadoop.

A.2 Word Length

The Word Length application consists of a MapReduce implementation for counting the total number of occurrences of different word lengths in a given text file.

The lengths are predefined in ranges, for instance, small (up to 4 characters), medium (between 5 and 8 characters) and large (9 or more characters). This way, it is known in advance how many keys (distinct length ranges) may be mapped.

In the *map* phase, each *map* execution receives a chunk of the input data, splits it in order to obtain the words, and emits an intermediate key/value pair for each word, in which the key is the range description (small, medium or large) of the word found, and the value is 1, indicating 1 occurrence found for that length range.

In the *reduce* phase, all values of a given key are summed up, which means summing up all values 1 of each length range producing thus the total number of occurrences of words with each length range in the input text file.

```
1 @MapReduce<WordLengthMR, long, text, string, int, "*:3">{
2
3     @Map(key, value){
4         toupper(value)
5         tokenize(value){
6             if(length(token) <= 4)
7                 emit("short", 1)
8             else if(length(token) >= 5 && length(token) <= 8)
9                 emit("medium", 1)
10            else
11                emit("large", 1)
12        }
13    }
14
15    @SumReducer
16 }
```

Listing A.5: Implementation with proposed interface with curly braces.

```
1 @MapReduce<WordLengthMR, long, text, string, int, "*:3">
2
3     @Map(key, value)
4         toupper(value)
5         tokenize(value)
6             if(length(token) <= 4)
7                 emit("short", 1)
8             else if(length(token) >= 5 && length(token) <= 8)
9                 emit("medium", 1)
10            else
11                emit("large", 1)
12
```

```
13     @SumReducer
```

Listing A.6: Implementation with proposed interface without curly braces.

```
1#include <sys/mman.h>
2#include <sys/stat.h>
3#include <fcntl.h>
4#include <string.h>
5#include <ctype.h>
6
7#include "map_reduce.h"
8
9struct ph_string {
10    char* data;
11    uint64_t len;
12};
13
14struct ph_word {
15    char* data;
16
17    ph_word() { data = NULL; }
18    ph_word(char* data) { this->data = data; }
19
20    bool operator<(ph_word const& other) const {
21        return strcmp(data, other.data) < 0;
22    }
23    bool operator==(ph_word const& other) const {
24        return strcmp(data, other.data) == 0;
25    }
26};
27
28struct ph_word_hash
29{
30    size_t operator()(ph_word const& key) const
31    {
32        char* h = key.data;
33        uint64_t v = 14695981039346656037ULL;
34        while (*h != 0)
35            v = (v ^ (size_t)(*(h++))) * 1099511628211ULL;
36        return v;
37    }
38};
39
```

```
40 class WordLengthMR : public MapReduceSort<WordLengthMR, ph_string,
     ph_word, uint64_t, fixed_hash_container<ph_word, uint64_t,
     sum_combiner, 3, ph_word_hash
41 #ifdef TBB
42     , tbb::scalable_allocator
43 #endif
44 > >
45 {
46     char* data;
47     uint64_t data_size;
48     uint64_t chunk_size;
49     uint64_t splitter_pos;
50 public:
51     explicit WordLengthMR(char* _data, uint64_t length, uint64_t
         _chunk_size) :
52         data(_data), data_size(length), chunk_size(_chunk_size),
53             splitter_pos(0) {}
54
55     void map(data_type const& s, map_container& out) const
56     {
57         for (uint64_t i = 0; i < s.len; i++)
58         {
59             s.data[i] = toupper(s.data[i]);
60         }
61
62         uint64_t i = 0;
63         while(i < s.len)
64         {
65             while(i < s.len && (s.data[i] < 'A' || s.data[i] > 'Z'))
66                 i++;
67             uint64_t start = i;
68             while(i < s.len && ((s.data[i] >= 'A' && s.data[i] <= 'Z')
                 || s.data[i] == '\''))
69                 i++;
70             if(i > start)
71             {
72                 s.data[i] = 0;
73                 ph_word token = { s.data+start };
74                 if(strlen(token.data) <= 4){
75                     emit_intermediate(out, ph_word("short"), 1);
76                 }else if(strlen(token.data) >= 5 && strlen(token.data)
                     <= 8){
```

```
77                    emit_intermediate(out, ph_word("medium"), 1);
78                }else{
79                    emit_intermediate(out, ph_word("large"), 1);
80                }
81            }
82        }
83    }
84
85    int split(ph_string& out)
86    {
87        if ((uint64_t)splitter_pos >= data_size)
88        {
89            return 0;
90        }
91
92        uint64_t end = std::min(splitter_pos + chunk_size, data_size);
93
94        while(end < data_size &&
95            data[end] != ' ' && data[end] != '\t' &&
96            data[end] != '\r' && data[end] != '\n')
97            end++;
98
99        out.data = data + splitter_pos;
100       out.len = end - splitter_pos;
101
102       splitter_pos = end;
103
104       return 1;
105   }
106
107   bool sort(keyval const& a, keyval const& b) const
108   {
109       return a.val < b.val || (a.val == b.val && strcmp(a.key.data, b.
            key.data) > 0);
110   }
111};
```

Listing A.7: Code for Phoenix++.

```
1 import java.io.IOException;
2 import java.util.*;
3
4 import org.apache.hadoop.fs.Path;
```

```
 5 import org.apache.hadoop.conf.*;
 6 import org.apache.hadoop.io.*;
 7 import org.apache.hadoop.mapreduce.*;
 8 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
 9 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
10 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
11 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
12 import org.apache.hadoop.mapreduce.lib.reduce.IntSumReducer;
13
14 public class WordLengthMR {
15
16     public static class Map extends Mapper<LongWritable, Text, Text,
           IntWritable> {
17
18         private final static IntWritable one = new IntWritable(1);
19         private Text word = new Text();
20
21         @Override
22         public void map(LongWritable key, Text value, Context context)
23                 throws IOException, InterruptedException {
24             String line = value.toString();
25             StringTokenizer tokenizer = new StringTokenizer(line);
26             while (tokenizer.hasMoreTokens()) {
27                 String token = tokenizer.nextToken();
28                 if (token.length() <= 4) {
29                     context.write(new Text("short"), one);
30                 } else if (token.length() >= 5 && token.length() <= 8) {
31                     context.write(new Text("medium"), one);
32                 } else {
33                     context.write(new Text("large"), one);
34                 }
35             }
36         }
37     }
38 }
```

Listing A.8: Code for Hadoop.

A.3 Histogram

The Histogram application consists of a MapReduce implementation for counting the total number of occurrences of different levels of red, green and blue in the RGB pixels of a given image file.

The possible RGB colors are always in a range of 0 to 255 for red, green and blue. This way, it is known in advance how many keys (distinct RGB ranges) may be mapped.

In the *map* phase, each *map* execution receives the data corresponding to a given pixel in the input image file, from each it emits one key/value pair for the red level, one other pair for the green level and finally one pair for the blue level. Given that the red, green and blue varies from 0 to 255, it sums 256 to the green level and 512 to the blue level in order to distinguish the keys for each one when the reduction is performed. For each one of the three key/value pairs, the value is 1, indicating 1 occurrence found for that RGB level.

In the *reduce* phase, all values of each key are summed up, which means summing up all values 1 of each red, green or blue level producing thus the total number of occurrences of each possible red, green and blue levels in the input image file.

```
1@type pixel(r: ushort, g: ushort, b: ushort)
2
3@MapReduce<Histogram, long, pixel, int, ulonglong, "*:768">{
4
5    @Map(key, p){
6        emit(p.b, 1)
7        emit(p.g+256, 1)
8        emit(p.r+512, 1)
9    }
10
11   @SumReducer
12}
```

Listing A.9: Implementation with proposed interface with curly braces.

```
1@type pixel(r: ushort, g: ushort, b: ushort)
2
3@MapReduce<Histogram, long, pixel, int, ulonglong, "*:768">
4
5    @Map(key, p)
6        emit(p.b, 1)
7        emit(p.g+256, 1)
8        emit(p.r+512, 1)
9
```

```
10    @SumReducer
```

Listing A.10: Implementation with proposed interface without curly braces.

```
1#include <sys/mman.h>
2#include <sys/stat.h>
3#include <fcntl.h>
4
5#include "map_reduce.h"
6
7struct pixel {
8     uint8_t b;
9     uint8_t g;
10    uint8_t r;
11};
12
13class HistogramMR : public MapReduceSort<HistogramMR, pixel, intptr_t,
     uint64_t, array_container<intptr_t, uint64_t, sum_combiner, 768
14#ifdef TBB
15    , tbb::scalable_allocator
16#endif
17 > >
18{
19public:
20    void map(data_type const& value, map_container& out) const {
21        emit_intermediate(out, value.b, 1);
22        emit_intermediate(out, value.g+256, 1);
23        emit_intermediate(out, value.r+512, 1);
24    }
25};
```

Listing A.11: Code for Phoenix++.

```
1import java.io.DataInput;
2import java.io.DataOutput;
3import java.io.IOException;
4import org.apache.hadoop.conf.Configuration;
5import org.apache.hadoop.fs.FSDataInputStream;
6import org.apache.hadoop.fs.FileSystem;
7import org.apache.hadoop.fs.Path;
8
9import org.apache.hadoop.io.*;
10import org.apache.hadoop.mapred.LineRecordReader;
11import org.apache.hadoop.mapreduce.*;
```

```
12 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
13 import org.apache.hadoop.mapreduce.lib.input.FileSplit;
14 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
15 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
16 import org.apache.hadoop.mapreduce.lib.reduce.IntSumReducer;
17
18 public class HistogramMR {
19
20     public static class Pixel implements Writable {
21
22         private int r;
23         private int g;
24         private int b;
25
26         public int getR() {
27             return r;
28         }
29
30         public int getG() {
31             return g;
32         }
33
34         public int getB() {
35             return b;
36         }
37
38         public void setR(int r) {
39             this.r = r;
40         }
41
42         public void setG(int g) {
43             this.g = g;
44         }
45
46         public void setB(int b) {
47             this.b = b;
48         }
49
50         @Override
51         public void write(DataOutput out) throws IOException {
52             out.writeInt(r);
53             out.writeInt(g);
```

```
54                out.writeInt(b);
55            }
56
57            @Override
58            public void readFields(DataInput in) throws IOException {
59                this.r = in.readInt();
60                this.g = in.readInt();
61                this.b = in.readInt();
62            }
63
64            @Override
65            public String toString() {
66                return r + ", " + g + ", " + b;
67            }
68        }
69
70        public static class PixelRecordReader
71                extends RecordReader<LongWritable, Pixel> {
72
73            private long start;
74            private long pos;
75            private long end;
76            private LineRecordReader.LineReader in;
77            private int maxLineLength;
78            private LongWritable key = new LongWritable();
79            private Pixel value = new Pixel();
80
81            @Override
82            public void initialize(
83                    InputSplit genericSplit,
84                    TaskAttemptContext context)
85                    throws IOException {
86
87                FileSplit split = (FileSplit) genericSplit;
88
89                Configuration job = context.getConfiguration();
90                this.maxLineLength = job.getInt(
91                        "mapred.linerecordreader.maxlength",
92                        Integer.MAX_VALUE);
93
94                start = split.getStart();
95                end = start + split.getLength();
```

```
96
97              final Path file = split.getPath();
98              FileSystem fs = file.getFileSystem(job);
99              FSDataInputStream fileIn = fs.open(split.getPath());
100
101             boolean skipFirstLine = false;
102             if (start != 0) {
103                 skipFirstLine = true;
104                 --start;
105                 fileIn.seek(start);
106             }
107
108             in = new LineRecordReader.LineReader(fileIn, job);
109
110             if (skipFirstLine) {
111                 Text dummy = new Text();
112                 // Reset "start" to "start + line offset"
113                 start += in.readLine(dummy, 0,
114                         (int) Math.min(
115                                 (long) Integer.MAX_VALUE,
116                                 end - start));
117             }
118
119             this.pos = start;
120         }
121
122         @Override
123         public boolean nextKeyValue() throws IOException {
124
125             key.set(pos);
126
127             int newSize = 0;
128
129             Text txValue = new Text();
130
131             while (pos < end) {
132
133                 newSize = in.readLine(txValue, maxLineLength,
134                         Math.max((int) Math.min(
135                                         Integer.MAX_VALUE, end - pos),
136                                 maxLineLength));
137
```

```
138                 this.value.setR(Integer.parseInt(txValue.toString().
                        split(" ")[0]));
139                 this.value.setG(Integer.parseInt(txValue.toString().
                        split(" ")[1]));
140                 this.value.setB(Integer.parseInt(txValue.toString().
                        split(" ")[1]));
141
142                 if (newSize == 0) {
143                     break;
144                 }
145
146                 pos += newSize;
147
148                 if (newSize < maxLineLength) {
149                     break;
150                 }
151             }
152
153             if (newSize == 0) {
154                 key = null;
155                 value = null;
156                 return false;
157             } else {
158                 return true;
159             }
160         }
161
162         @Override
163         public LongWritable getCurrentKey() throws IOException,
164                 InterruptedException {
165             return key;
166         }
167
168         @Override
169         public Pixel getCurrentValue() throws IOException,
                InterruptedException {
170             return value;
171         }
172
173         @Override
174         public float getProgress() throws IOException,
                InterruptedException {
```

```
175             if (start == end) {
176                 return 0.0f;
177             } else {
178                 return Math.min(1.0f, (pos - start) / (float) (end -
                        start));
179             }
180         }
181 
182         @Override
183         public void close() throws IOException {
184             if (in != null) {
185                 in.close();
186             }
187         }
188     }
189 
190     public static class PixelInputFormat extends FileInputFormat<
            LongWritable, Pixel> {
191 
192         public PixelInputFormat() {
193         }
194 
195         @Override
196         public RecordReader<LongWritable, Pixel>
197                 createRecordReader(InputSplit split,
198                         TaskAttemptContext context) throws IOException,
199                 InterruptedException {
200 
201             return new PixelRecordReader();
202         }
203     }
204 
205     public static class Map extends Mapper<LongWritable, Pixel,
            IntWritable, IntWritable> {
206 
207         private final static IntWritable one = new IntWritable(1);
208 
209         @Override
210         public void map(LongWritable key, Pixel p, Context context)
211                 throws IOException, InterruptedException {
212             context.write(new IntWritable(p.getR()), one);
213             context.write(new IntWritable(p.getG() + 256), one);
```

```
214             context.write(new IntWritable(p.getB() + 512), one);
215         }
216     }
217}
```

Listing A.12: Code for Hadoop.

A.4 K-means

The K-means application consists of a MapReduce implementation for K-means clustering algorithm, which aims at iteratively decreasing the squared distante among N clusters and a set of points until a totally converged status.

The given MapReduce application represents each iteration, in which it calculates the distances and the need of repositioning the clusters. The K-means algorithm states that the number of clusters needs to be defined beforehand, which in the MapReduce context means to know in advance how many keys (distinct clusters) will be mapped.

In the *map* phase, each *map* execution receives the data corresponding to one cluster and iterates over every points in order to find the closest one. Once it has been found, the cluster is set a new position according to it. Finally, it emits one key/value pair where the key is the closest distance found, and the value is the resulting coordinates for the given cluster.

In the *reduce* phase, all values of each key are summed up, which particularly for this application means summing up the values of the attribute *x*, *y*, *z* and *cluster* of the custom *CPoint* type producing thus the input data for the next K-means iteration.

```
1@type cpoint(x: int, y: int, z: int,
2       cpoint(){
3           x = 0
4           y = 0
5           z = 0
6           cluster = −1
7       },
8       cpoint(x: int, y: int, z: int, cluster: int){
9           x /= cluster
10          y /= cluster
11          z /= cluster
12          cluster = 1
13          this
14      },
15      normalize(): cpoint {
```

```
16            x /= cluster
17            y /= cluster
18            z /= cluster
19            cluster = 1
20            this
21        },
22        sq_dist(p: cpoint): int {
23            int sum = 0
24
25            int diff = x - p.x
26            sum += diff * diff
27
28            diff = y - p.y
29            sum += diff * diff
30
31            diff = z - p.z
32            sum += diff * diff
33
34            sum
35        },
36        generate(): void {
37            x = rand()
38            y = rand()
39            z = rand()
40        })
41
42boolean modified = false
43vector<cpoint> means
44
45@MapReduce<KmeansMR, long, cpoint, int, cpoint, "*:100">{
46
47    @Map(key, value){
48        int min_dist = max_int()
49        int min_idx = 0
50
51        for (int j = 0; j < means.size(); j++){
52            int cur_dist = value.sq_dist(means[j])
53            if (cur_dist < min_dist){
54                min_dist = cur_dist
55                min_idx = j
56            }
57        }
```

```
58
59          if (value.cluster != min_idx){
60              value.cluster = min_idx
61              modified = true
62          }
63
64          emit(min_idx, cpoint(value.x, value.y, value.z, 1))
65      }
66
67      @SumReducer
68}
```

Listing A.13: Implementation with proposed interface with curly braces.

```
1@type cpoint(x: int, y: int, z: int,
2       cpoint()
3           x = 0
4           y = 0
5           z = 0
6           cluster = -1
7       ,
8       cpoint(x: int, y: int, z: int, cluster: int)
9           x /= cluster
10          y /= cluster
11          z /= cluster
12          cluster = 1
13          this
14      ,
15      normalize(): cpoint
16          x /= cluster
17          y /= cluster
18          z /= cluster
19          cluster = 1
20          this
21      ,
22      sq_dist(p: cpoint): int
23          int sum = 0
24
25          int diff = x - p.x
26          sum += diff * diff
27
28          diff = y - p.y
29          sum += diff * diff
```

```
30
31              diff = z - p.z
32              sum += diff * diff
33
34              sum
35          ,
36          generate(): void
37              x = rand()
38              y = rand()
39              z = rand()
40          )
41
42boolean modified = false
43vector<cpoint> means
44
45@MapReduce<KmeansMR, long, cpoint, int, cpoint, "*:100">
46
47     @Map(key, value)
48          int min_dist = max_int()
49          int min_idx = 0
50
51          for (int j = 0; j < means.size(); j++)
52                  int cur_dist = value.sq_dist(means[j])
53                  if (cur_dist < min_dist)
54                      min_dist = cur_dist
55                      min_idx = j
56
57          if (value.cluster != min_idx)
58                  value.cluster = min_idx
59                  modified = true
60
61          emit(min_idx, cpoint(value.x, value.y, value.z, 1))
62
63     @SumReducer
```

Listing A.14: Implementation with proposed interface without curly braces.

```
1#include <sys/mman.h>
2#include <sys/stat.h>
3#include <fcntl.h>
4#include <limits>
5
6#include "map_reduce.h"
```

```
7
8int grid_size = 1000;
9
10struct cpoint
11{
12     int x;
13     int y;
14     int z;
15     int cluster;
16
17     cpoint() {
18         x = 0;
19         y = 0;
20         z = 0;
21         cluster = -1;
22     }
23
24     cpoint(int x, int y, int z, int cluster) {
25         this->x = x;
26         this->y = y;
27         this->z = z;
28         this->cluster = cluster;
29     }
30
31     cpoint& normalize() {
32         x /= cluster;
33         y /= cluster;
34         z /= cluster;
35         cluster = 1;
36         return *this;
37     }
38
39     unsigned int sq_dist(cpoint const& p)
40     {
41         unsigned int sum = 0;
42
43         int diff = x - p.x;
44         sum += diff * diff;
45
46         diff = y - p.y;
47         sum += diff * diff;
48
```

```
49          diff = z - p.z;
50          sum += diff * diff;
51
52          return sum;
53      }
54
55      void generate() {
56          x = rand();
57          y = rand();
58          z = rand();
59      }
60 };
61
62 bool modified = false;
63 std::vector<cpoint> means;
64
65 template<class V, template<class> class Allocator>
66 class KmeansMR_combiner : public associative_combiner<KmeansMR_combiner<
       V, Allocator>, V, Allocator>
67 {
68 public:
69     static void F(cpoint& a, cpoint const& b) {
70         a.cluster += b.cluster;
71         a.x += b.x;
72         a.y += b.y;
73         a.z += b.z;
74     }
75     static void Init(cpoint& a) {
76         a.cluster = 0;
77         a.x = 0;
78         a.y = 0;
79         a.z = 0;
80     }
81     static bool Empty(cpoint const& a) {
82         return a.cluster == 0 && a.x == 0 && a.y == 0 && a.z == 0;
83     }
84 };
85
86 class KmeansMR : public MapReduce<KmeansMR, cpoint, intptr_t, cpoint,
       array_container<intptr_t, cpoint, KmeansMR_combiner, 100
87 #ifdef TBB
88     , tbb::scalable_allocator
```

```
89#endif
90> >
91{
92
93public:
94
95    void map(data_type& value, map_container& out) const
96    {
97        unsigned int min_dist = std::numeric_limits<unsigned int>::max()
            ;
98        unsigned int min_idx = 0;
99
100       for (size_t j = 0; j < means.size(); j++)
101       {
102           unsigned int cur_dist = value.sq_dist(means[j]);
103           if (cur_dist < min_dist)
104           {
105               min_dist = cur_dist;
106               min_idx = j;
107           }
108       }
109
110       if (value.cluster != (int)min_idx)
111       {
112           value.cluster = (int)min_idx;
113           modified = true;
114       }
115
116       emit_intermediate(out, min_idx, cpoint(value.x, value.y, value.z
            , 1));
117   }
118};
```

Listing A.15: Code for Phoenix++.

```
1import java.io.DataInput;
2import java.io.DataOutput;
3import java.io.IOException;
4import java.util.ArrayList;
5import java.util.List;
6
7import org.apache.hadoop.fs.Path;
8import org.apache.hadoop.conf.*;
```

```
 9 import org.apache.hadoop.fs.FSDataInputStream;
10 import org.apache.hadoop.fs.FileSystem;
11 import org.apache.hadoop.io.*;
12 import org.apache.hadoop.mapred.LineRecordReader;
13 import org.apache.hadoop.mapreduce.*;
14 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
15 import org.apache.hadoop.mapreduce.lib.input.FileSplit;
16 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
17 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
18
19 public class KmeansMR {
20
21     public static boolean modified = false;
22     public static int grid_size = 1000;
23     public static List<Cpoint> means = new ArrayList<>();
24
25     public static class Cpoint implements Writable {
26
27         private int x;
28         private int y;
29         private int z;
30         private int cluster;
31
32         public Cpoint() {
33             this.x = 0;
34             this.y = 0;
35             this.z = 0;
36             this.cluster = -1;
37         }
38
39         public Cpoint(int x, int y, int z, int cluster) {
40             this.x = x;
41             this.y = y;
42             this.z = z;
43             this.cluster = cluster;
44         }
45
46         public Cpoint normalize() {
47             this.x /= this.cluster;
48             this.y /= this.cluster;
49             this.z /= this.cluster;
50             this.cluster = 1;
```

```
51             return this;
52         }
53 
54         public int sq_dist(Cpoint p) {
55             int sum = 0;
56 
57             int diff = x - p.x;
58             sum += diff * diff;
59 
60             diff = y - p.y;
61             sum += diff * diff;
62 
63             diff = z - p.z;
64             sum += diff * diff;
65 
66             return sum;
67         }
68 
69         public void generate() {
70             x = (int) (Math.random() * grid_size);
71             y = (int) (Math.random() * grid_size);
72             z = (int) (Math.random() * grid_size);
73         }
74 
75         public int getX() {
76             return x;
77         }
78 
79         public int getY() {
80             return y;
81         }
82 
83         public int getZ() {
84             return z;
85         }
86 
87         public int getCluster() {
88             return cluster;
89         }
90 
91         public void setX(int x) {
92             this.x = x;
```

```
93          }
94
95          public void setY(int y) {
96              this.y = y;
97          }
98
99          public void setZ(int z) {
100             this.z = z;
101         }
102
103         public void setCluster(int cluster) {
104             this.cluster = cluster;
105         }
106
107         @Override
108         public void write(DataOutput out) throws IOException {
109             out.writeInt(x);
110             out.writeInt(y);
111             out.writeInt(z);
112             out.writeInt(cluster);
113         }
114
115         @Override
116         public void readFields(DataInput in) throws IOException {
117             this.x = in.readInt();
118             this.y = in.readInt();
119             this.z = in.readInt();
120             this.cluster = in.readInt();
121         }
122
123         @Override
124         public String toString() {
125             return x + ", " + y + ", " + z + ", " + cluster;
126         }
127     }
128
129     public static class CPointRecordReader
130             extends RecordReader<LongWritable, Cpoint> {
131
132         private long start;
133         private long pos;
134         private long end;
```

```
135         private LineRecordReader.LineReader in;
136         private int maxLineLength;
137         private LongWritable key = new LongWritable();
138         private Cpoint value = new Cpoint();
139 
140         @Override
141         public void initialize(
142                 InputSplit genericSplit,
143                 TaskAttemptContext context)
144                 throws IOException {
145 
146             FileSplit split = (FileSplit) genericSplit;
147 
148             Configuration job = context.getConfiguration();
149             this.maxLineLength = job.getInt(
150                     "mapred.linerecordreader.maxlength",
151                     Integer.MAX_VALUE);
152 
153             start = split.getStart();
154             end = start + split.getLength();
155 
156             final Path file = split.getPath();
157             FileSystem fs = file.getFileSystem(job);
158             FSDataInputStream fileIn = fs.open(split.getPath());
159 
160             boolean skipFirstLine = false;
161             if (start != 0) {
162                 skipFirstLine = true;
163                 --start;
164                 fileIn.seek(start);
165             }
166 
167             in = new LineRecordReader.LineReader(fileIn, job);
168 
169             if (skipFirstLine) {
170                 Text dummy = new Text();
171                 // Reset "start" to "start + line offset"
172                 start += in.readLine(dummy, 0,
173                         (int) Math.min(
174                                 (long) Integer.MAX_VALUE,
175                                 end - start));
176             }
```

```
177
178             this.pos = start;
179         }
180
181         @Override
182         public boolean nextKeyValue() throws IOException {
183
184             key.set(pos);
185
186             int newSize = 0;
187
188             Text txValue = new Text();
189
190             while (pos < end) {
191
192                 newSize = in.readLine(txValue, maxLineLength,
193                         Math.max((int) Math.min(
194                                         Integer.MAX_VALUE, end - pos),
195                                 maxLineLength));
196
197                 this.value.setX(Integer.parseInt(txValue.toString().
                        split(" ")[0]));
198                 this.value.setY(Integer.parseInt(txValue.toString().
                        split(" ")[1]));
199                 this.value.setZ(Integer.parseInt(txValue.toString().
                        split(" ")[2]));
200                 this.value.setCluster(Integer.parseInt(txValue.toString
                        ().split(" ")[3]));
201
202                 if (newSize == 0) {
203                     break;
204                 }
205
206                 pos += newSize;
207
208                 if (newSize < maxLineLength) {
209                     break;
210                 }
211             }
212
213             if (newSize == 0) {
214                 key = null;
```

```
215                 value = null;
216                 return false;
217             } else {
218                 return true;
219             }
220         }
221 
222         @Override
223         public LongWritable getCurrentKey() throws IOException,
224                 InterruptedException {
225             return key;
226         }
227 
228         @Override
229         public Cpoint getCurrentValue() throws IOException,
               InterruptedException {
230             return value;
231         }
232 
233         @Override
234         public float getProgress() throws IOException,
               InterruptedException {
235             if (start == end) {
236                 return 0.0f;
237             } else {
238                 return Math.min(1.0f, (pos - start) / (float) (end -
                       start));
239             }
240         }
241 
242         @Override
243         public void close() throws IOException {
244             if (in != null) {
245                 in.close();
246             }
247         }
248     }
249 
250     public static class CPointInputFormat extends FileInputFormat<
           LongWritable, Cpoint> {
251 
252         public CPointInputFormat() {
```

```
253         }
254
255         @Override
256         public RecordReader<LongWritable, Cpoint>
257                 createRecordReader(InputSplit split,
258                         TaskAttemptContext context) throws IOException,
259                 InterruptedException {
260
261             return new CPointRecordReader();
262         }
263     }
264
265     public static class Map extends Mapper<LongWritable, Cpoint,
            IntWritable, Cpoint> {
266
267         public void map(LongWritable key, Cpoint value, Reducer.Context
                context)
268                 throws IOException, InterruptedException {
269             int min_dist = Integer.MAX_VALUE;
270             int min_idx = 0;
271
272             for (int j = 0; j < means.size(); j++) {
273                 int cur_dist = value.sq_dist(means.get(j));
274                 if (cur_dist < min_dist) {
275                     min_dist = cur_dist;
276                     min_idx = j;
277                 }
278             }
279
280             if (value.getCluster() != (int) min_idx) {
281                 value.setCluster((int) min_idx);
282                 modified = true;
283             }
284
285             context.write(new IntWritable(min_idx), new Cpoint(value.
                    getX(), value.getY(), value.getZ(), 1));
286         }
287     }
288
289     public static class Reduce extends Reducer<IntWritable, Cpoint,
            IntWritable, Cpoint> {
290
```

```
291         public void reduce(IntWritable key, Iterable<Cpoint> values,
                Reducer.Context context)
292                 throws IOException, InterruptedException {
293
294             Cpoint reduced = new Cpoint();
295             reduced.setX(0);
296             reduced.setY(0);
297             reduced.setZ(0);
298             reduced.setCluster(0);
299
300             for (Cpoint val : values) {
301                 reduced.setX(reduced.getX() + val.getX());
302                 reduced.setY(reduced.getY() + val.getY());
303                 reduced.setZ(reduced.getZ() + val.getZ());
304                 reduced.setCluster(reduced.getCluster() + val.getCluster
                        ());
305             }
306
307             context.write(key, reduced);
308         }
309     }
310}
```

Listing A.16: Code for Hadoop.

A.5 Linear Regression

The Linear Regression application consists of a MapReduce implementation for Linear Regression algorithm for statistical inference and learning, which aims at finding the best coefficients for a linear correlation among variables whose values are represented as points in a cartesian plane.

The given MapReduce application represents an initial step in the calculus of the linear regression, which calculates a set of new values based on the x and y coordinates, which implies a number of keys known in advance.

In the *map* phase, each *map* execution receives the data corresponding to one point (x and y coordinates) and calculates x *squared*, y *squared* and x times y. Finally, it emits one key/value pair where the key is a constant indicating the operation and the value is the result of that operation, additionally the original x and y values are also emitted.

In the *reduce* phase, all values of each key are summed up, which particularly for this application means summing up the values produced by each operation.

```
1@type point(x: uint, y: uint)
2
3uint KEY_SX = 0
4uint KEY_SY = 1
5uint KEY_SXX = 2
6uint KEY_SYY = 3
7uint KEY_SXY = 4
8
9@MapReduce<LinearRegressionMR, long, Point, uint, ulonglong, "*:5">{
10
11    @Map(key, value){
12        ulonglong px = p.x
13                ulonglong py = p.y
14
15                emit(KEY_SXX, px*px)
16                emit(KEY_SYY, py*py)
17                emit(KEY_SXY, px*py)
18                emit(KEY_SX,  px)
19                emit(KEY_SY,  py)
20        }
21
22    @SumReducer
23}
```

Listing A.17: Implementation with proposed interface with curly braces.

```
1@type point(x: uint, y: uint)
2
3uint KEY_SX = 0
4uint KEY_SY = 1
5uint KEY_SXX = 2
6uint KEY_SYY = 3
7uint KEY_SXY = 4
8
9@MapReduce<LinearRegressionMR, long, Point, uint, ulonglong, "*:5">
10
11    @Map(key, value)
12        ulonglong px = p.x
13                ulonglong py = p.y
14
15                emit(KEY_SXX, px*px)
16                emit(KEY_SYY, py*py)
17                emit(KEY_SXY, px*py)
```

```
18                emit(KEY_SX,   px)
19                emit(KEY_SY,   py)
20
21     @SumReducer
```

Listing A.18: Implementation with proposed interface without curly braces.

```
1#include <sys/mman.h>
2#include <sys/stat.h>
3#include <fcntl.h>
4
5#include "map_reduce.h"
6
7struct point {
8     uint16_t x;
9     uint16_t y;
10};
11
12uint16_t const KEY_SX = 0;
13uint16_t const KEY_SY = 1;
14uint16_t const KEY_SXX = 2;
15uint16_t const KEY_SYY = 3;
16uint16_t const KEY_SXY = 4;
17
18class LinearRegressionMR : public MapReduce<LinearRegressionMR, point,
     uint16_t, uint64_t, array_container< uint16_t, uint64_t, sum_combiner
     , 5
19#ifdef TBB
20     , tbb::scalable_allocator
21#endif
22> >
23{
24public:
25     void map(data_type const& value, map_container& out) const
26     {
27         uint64_t px = value.x;
28         uint64_t py = value.y;
29
30         emit_intermediate(out, KEY_SXX, px*px);
31         emit_intermediate(out, KEY_SYY, py*py);
32         emit_intermediate(out, KEY_SXY, px*py);
33         emit_intermediate(out, KEY_SX,  px);
34         emit_intermediate(out, KEY_SY,  py);
```

```
35    }
36};
```

Listing A.19: Code for Phoenix++.

```
 1import java.io.DataInput;
 2import java.io.DataOutput;
 3import java.io.IOException;
 4
 5import org.apache.hadoop.fs.Path;
 6import org.apache.hadoop.conf.*;
 7import org.apache.hadoop.fs.FSDataInputStream;
 8import org.apache.hadoop.fs.FileSystem;
 9import org.apache.hadoop.io.*;
10import org.apache.hadoop.mapred.LineRecordReader;
11import org.apache.hadoop.mapreduce.*;
12import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
13import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
14import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
15import org.apache.hadoop.mapreduce.lib.reduce.LongSumReducer;
16
17import org.apache.hadoop.mapreduce.InputSplit;
18import org.apache.hadoop.mapreduce.RecordReader;
19import org.apache.hadoop.mapreduce.TaskAttemptContext;
20import org.apache.hadoop.mapreduce.lib.input.FileSplit;
21
22public class LinearRegressionMR {
23
24    public static final int KEY_SX = 0;
25    public static final int KEY_SY = 1;
26    public static final int KEY_SXX = 2;
27    public static final int KEY_SYY = 3;
28    public static final int KEY_SXY = 4;
29
30    public static class Point implements Writable {
31
32        private int x;
33        private int y;
34
35        public int getX() {
36            return x;
37        }
38
```

```
39        public int getY() {
40            return y;
41        }
42
43        public void setX(int x) {
44            this.x = x;
45        }
46
47        public void setY(int y) {
48            this.y = y;
49        }
50
51        @Override
52        public void write(DataOutput out) throws IOException {
53            out.writeInt(x);
54            out.writeInt(y);
55        }
56
57        @Override
58        public void readFields(DataInput in) throws IOException {
59            this.x = in.readInt();
60            this.y = in.readInt();
61        }
62
63        @Override
64        public String toString() {
65            return x + ", " + y;
66        }
67    }
68
69    public static class PointRecordReader
70            extends RecordReader<LongWritable, Point> {
71
72        private long start;
73        private long pos;
74        private long end;
75        private LineRecordReader.LineReader in;
76        private int maxLineLength;
77        private LongWritable key = new LongWritable();
78        private Point value = new Point();
79
80        @Override
```

```
81      public void initialize(
82              InputSplit genericSplit,
83              TaskAttemptContext context)
84              throws IOException {
85
86          FileSplit split = (FileSplit) genericSplit;
87
88          Configuration job = context.getConfiguration();
89          this.maxLineLength = job.getInt(
90                  "mapred.linerecordreader.maxlength",
91                  Integer.MAX_VALUE);
92
93          start = split.getStart();
94          end = start + split.getLength();
95
96          final Path file = split.getPath();
97          FileSystem fs = file.getFileSystem(job);
98          FSDataInputStream fileIn = fs.open(split.getPath());
99
100         boolean skipFirstLine = false;
101         if (start != 0) {
102             skipFirstLine = true;
103             --start;
104             fileIn.seek(start);
105         }
106
107         in = new LineRecordReader.LineReader(fileIn, job);
108
109         if (skipFirstLine) {
110             Text dummy = new Text();
111             start += in.readLine(dummy, 0,
112                     (int) Math.min(
113                             (long) Integer.MAX_VALUE,
114                             end - start));
115         }
116
117         this.pos = start;
118     }
119
120     @Override
121     public boolean nextKeyValue() throws IOException {
122
```

```
123             key.set(pos);
124
125             int newSize = 0;
126
127             Text txValue = new Text();
128
129             while (pos < end) {
130
131                 newSize = in.readLine(txValue, maxLineLength,
132                         Math.max((int) Math.min(
133                                         Integer.MAX_VALUE, end - pos),
134                                 maxLineLength));
135
136                 this.value.setX(Integer.parseInt(txValue.toString().
                        split(" ")[0]));
137                 this.value.setY(Integer.parseInt(txValue.toString().
                        split(" ")[1]));
138
139                 if (newSize == 0) {
140                     break;
141                 }
142
143                 pos += newSize;
144
145                 if (newSize < maxLineLength) {
146                     break;
147                 }
148             }
149
150             if (newSize == 0) {
151                 key = null;
152                 value = null;
153                 return false;
154             } else {
155                 return true;
156             }
157         }
158
159         @Override
160         public LongWritable getCurrentKey() throws IOException,
161                 InterruptedException {
162             return key;
```

```
163         }
164 
165         @Override
166         public Point getCurrentValue() throws IOException,
                InterruptedException {
167             return value;
168         }
169 
170         @Override
171         public float getProgress() throws IOException,
                InterruptedException {
172             if (start == end) {
173                 return 0.0f;
174             } else {
175                 return Math.min(1.0f, (pos - start) / (float) (end -
                        start));
176             }
177         }
178 
179         @Override
180         public void close() throws IOException {
181             if (in != null) {
182                 in.close();
183             }
184         }
185     }
186 
187     public static class PointInputFormat extends FileInputFormat<
            LongWritable, Point> {
188 
189         public PointInputFormat() {
190         }
191 
192         @Override
193         public RecordReader<LongWritable, Point>
194                 createRecordReader(InputSplit split,
195                         TaskAttemptContext context) throws IOException,
196                 InterruptedException {
197 
198             return new PointRecordReader();
199         }
200     }
```

```
201
202     public static class Map extends Mapper<LongWritable, Point,
            IntWritable, LongWritable> {
203
204         @Override
205         public void map(LongWritable key, Point value, Context context)
206                 throws IOException, InterruptedException {
207
208             int px = value.getX();
209             int py = value.getY();
210
211             context.write(new IntWritable(KEY_SXX), new LongWritable(px
                    * px));
212             context.write(new IntWritable(KEY_SYY), new LongWritable(py
                    * py));
213             context.write(new IntWritable(KEY_SXY), new LongWritable(px
                    * py));
214             context.write(new IntWritable(KEY_SX), new LongWritable(px))
                    ;
215             context.write(new IntWritable(KEY_SY), new LongWritable(py))
                    ;
216         }
217     }
218}
```

Listing A.20: Code for Hadoop.

A.6 Matrix Multiplication

The Matrix Multiplication code presented in the listing A.21 is the original code suggested by the Phoenix++ project, which demonstrates the need of implementing specific *split* logic not compatible with the proposed unified interface.

```
1#include <sys/mman.h>
2#include <sys/stat.h>
3#include <fcntl.h>
4#include <assert.h>
5
6#include "map_reduce.h"
7
8struct mm_data_t {
```

```
9     int row_num;
10    int rows;
11    int *matrix_A;
12    int *matrix_B;
13    int matrix_len;
14    int* output;
15};
16
17class MatrixMulMR : public MapReduce<MatrixMulMR, mm_data_t, int, int>
18{
19    int *matrix_A, *matrix_B;
20    int matrix_size;
21    int row;
22    int *output;
23
24public:
25    explicit MatrixMulMR(int* _mA, int* _mB, int size, int* out) :
26        matrix_A(_mA), matrix_B(_mB), matrix_size(size), row(0), output(
            out) {}
27
28    void map(mm_data_t const& data, map_container& out) const
29    {
30        int* a_ptr = data.matrix_A + data.row_num*data.matrix_len + 0;
31        int* b_ptr = data.matrix_B + 0;
32
33        int* output = data.output + data.row_num*data.matrix_len;
34        for(int i = 0; i < data.matrix_len ; i++) {
35            for(int j=0;j<data.matrix_len ; j++) {
36                output[j] += a_ptr[i] * b_ptr[j];
37            }
38            b_ptr += data.matrix_len;
39        }
40    }
41
42    int split(mm_data_t& out)
43    {
44        if (row >= matrix_size)
45        {
46            return 0;
47        }
48
49        out.matrix_A = matrix_A;
```

```
50          out.matrix_B = matrix_B;
51          out.matrix_len = matrix_size;
52          out.output = output;
53          out.rows = 1;
54          out.row_num = row++;
55
56          return 1;
57      }
58};
```

Listing A.21: Original code suggested by Phoenix++ project.

A.7 Principal Component Analysis

The Principal Component Analysis code presented in the listing A.22 is the original code suggested by the Phoenix++ project, which demonstrates the need of implementing specific *split* logic not compatible with the proposed unified interface.

```
1#include <sys/mman.h>
2#include <sys/stat.h>
3#include <fcntl.h>
4
5#include "map_reduce.h"
6
7typedef struct {
8    int row_num;
9    int const* matrix;
10} pca_map_data_t;
11
12typedef struct {
13    int const* matrix;
14    long long const* means;
15    int row_num;
16    int col_num;
17} pca_cov_data_t;
18
19#define DEF_GRID_SIZE 100
20#define DEF_NUM_ROWS 10
21#define DEF_NUM_COLS 10
22
23int num_rows;
```

```
24int num_cols;
25int grid_size;
26
27class MeanMR : public MapReduce<MeanMR, pca_map_data_t, int, long long,
     common_array_container<int, long long, one_combiner, 1000
28#ifdef TBB
29     , tbb::scalable_allocator
30#endif
31> >
32{
33     int *matrix;
34     int row;
35
36public:
37     explicit MeanMR(int* _matrix) : matrix(_matrix), row(0) {}
38
39     void map(data_type const& data, map_container& out) const
40     {
41          long long sum = 0;
42          int const* m = data.matrix + data.row_num * num_cols;
43          for(int j = 0; j < num_cols; j++)
44          {
45               sum += m[j];
46          }
47          emit_intermediate(out, data.row_num, sum/num_cols);
48     }
49
50     int split(pca_map_data_t& out)
51     {
52          if (row >= num_rows)
53          {
54               return 0;
55          }
56
57          out.matrix = matrix;
58          out.row_num = row++;
59
60          return 1;
61     }
62};
63
```

```
64 class CovMR : public MapReduceSort<CovMR, pca_cov_data_t, intptr_t, long
       long, common_array_container<intptr_t, long long, one_combiner,
       1000*1000
65 #ifdef TBB
66     , tbb::scalable_allocator
67 #endif
68 > >
69 {
70     int *matrix;
71     long long const* means;
72     int row;
73     int col;
74
75 public:
76     explicit CovMR(int* _matrix, long long const* _means) :
77         matrix(_matrix), means(_means), row(0), col(0) {}
78
79     void map(data_type const& data, map_container& out) const
80     {
81         int const* v1 = data.matrix + data.row_num*num_cols;
82         int const* v2 = data.matrix + data.col_num*num_cols;
83         long long m1 = data.means[data.row_num];
84         long long m2 = data.means[data.col_num];
85         long long sum = 0;
86         for(int i = 0; i < num_cols; i++)
87         {
88             sum += (v1[i] - m1) * (v2[i] - m2);
89         }
90         sum /= (num_cols-1);
91         emit_intermediate(out, data.row_num*num_cols + data.col_num, sum
              );
92     }
93
94     int split(pca_cov_data_t& out)
95     {
96         if (row >= num_rows)
97         {
98             return 0;
99         }
100
101        out.matrix = matrix;
102        out.means = means;
```

```
103         out.row_num = row;
104         out.col_num = col;
105
106         col++;
107         if(col >= num_rows)
108         {
109             row++;
110             col = row;
111         }
112
113         return 1;
114     }
115 };
```

Listing A.22: Original code suggested by Phoenix++ project.

APPENDIX B – MAPREDUCE FOR MANY-CORE GPU ARCHITECTURES

Computers have exploited higher levels of parallelism, more than faster processing units.

Meanwhile, Graphics Processing Units (GPUs) have evolved to General Purpose Graphics Processing Units (GPGPUs), bringing parallelism of general purpose applications to thousands of *lightweight threads* in a single device. Within the last 5-6 years, GPUs have emerged as the means for achieving *extreme-scale*, *cost-effective* and *power-efficient* high performance computing [CA12].

In this context, applications developed to run in shared-memory environments stay far below of an optimal resource usage if not using GPGPUs, except for algorithms that require sequential processing.

Dean and Ghemawat [DG08] first presented MapReduce as a prominent model for leveraging parallelism. Afterwards, some researches (sections 2.4 to 2.7) showed the MapReduce's applicability for shared-memory multi-core environments. Finally, more recent researches [HFL+08, HCC+10, BK13] found out the need of exploiting GPGPU parallelism and also proved MapReduce's applicability for such heterogeneous environments.

After briefly showing some state-of-the-art shared-memory multi-core implementations in sections 2.4 to 2.7, this section presents Grex [BK13], a MapReduce implementation for heterogeneous architectures with CUDA [NBGS08].

B.1 Backgrounds of GPGPUs

Modern GPGPUs provide hundreds of SIMT (Single Instruction Multiple Threads) multiprocessors, which can accommodate thousands of executing *threads*. Threads are grouped in *thread blocks* which in turn are scheduled dynamically to run in the physical multiprocessors. Due to the underlying SIMT architecture, all *threads* in a *block* running in a multiprocessor execute each instruction of the same code simultaneously. This behavior enhances parallelism, however it may be lost when there is *control divergence* which is caused by control flow conditions along the code of a *kernel*.

In order to support efficient memory access and locality, modern GPUs employ a sophisticated memory hierarchy shown in figure B.1 and described as follows:

- **Registers per thread** - Each *thread* owns its private set of registers, which represent the memory of more efficient access.

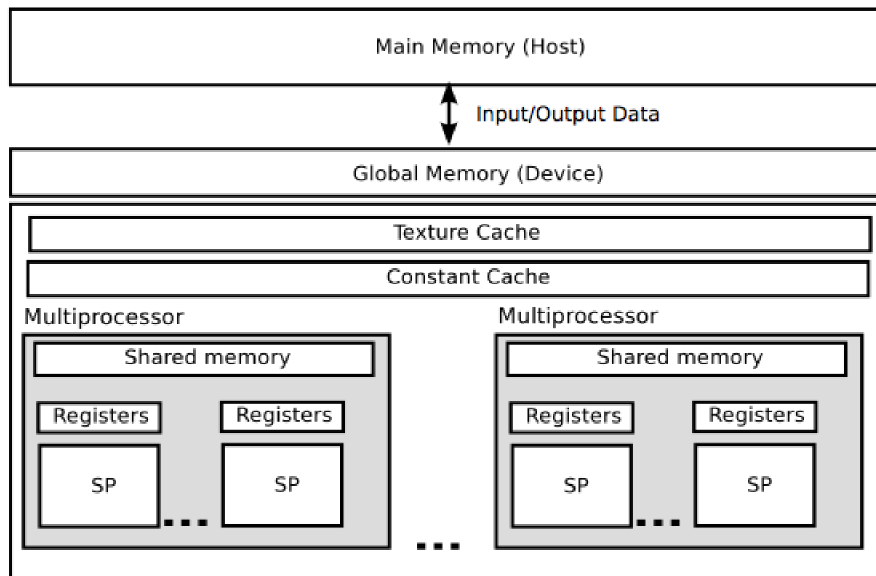


Figure B.1: General Purpose GPU Architecture. (SP: Streaming Multi-Processor) [BK13]

- **Shared Memory** - All *threads* in a *thread block* share a memory region called *shared memory*, which offers an efficiency of access close to the *registers*.
- **Constant Memory** - Any *thread* in any *thread block* has access to a unique read-only memory region called *constant memory*, which has short access latency upon a *cache* hit and is aimed at storing shorter fixed-size data structures frequently accessed by all *threads*. Its allocation is static and must be know in advance at compile time.
- **Texture Cache** - Also read-only and accessible from any *thread*, the *texture cache* is a hardware managed cache for the global memory, which can be used to store frequently accessed read-only data.
- **Global Memory** - The *global memory* is the largest memory region in the GPU device, and also the slowest. Is accessible from any *threads* and also is the only region accessible from the *host* CPU. Before and after calling a *kernel* function in order to run at the GPU, the code running at the *host* CPU uses this region to exchange data with the GPU device.

B.2 Precedent researches

Grex references some precedent researches which similarly are designed for heterogeneous architecture with a single GPU device (not cluster of GPUs), namely Mars [HFL+08], MapCG [HCC+10], [JM11] and [CA12].

Mars shows up to 16x speedup over Phoenix [RRP+07], but also shows suboptimal approaches in its implementation, such as a double pass execution of MapReduce pipeline.

MapCG achieves an average of 2-3x speedup over Phoenix 2 [YRK09], but forces sequential steps while using *locks* and *atomics*, same problem in [JM11] and [CA12]. [JM11] has an average of 2.67x speedup over Mars. [CA12] has a speedup between 5x and 200x over MapCG, and between 2x and 60x over [JM11].

Grex attempts to overcome the weaknesses of the mentioned precedent researches by presenting new approaches to MapReduce implementation on heterogeneous systems, such as *Parallel Split* and *Lazy Emit* achieving thus strong load balancing. Table B.1 summarizes some relevant characteristics and the differences among researches.

Table B.1: Comparisons of the GPU-based MapReduce frameworks [BK13]

Property	Grex	Mars	MapCG	[JM11]	[CA12]
Single pass execution	yes	no	yes	yes	yes
Locks and atomics	no	no	yes	yes	yes
Parallel split	yes	no	no	no	no
Load balancing	yes	no	no	no	no
Lazy emit	yes	no	no	no	no
GPU caching	yes	no	no	no	no

B.3 MapReduce Phases in Grex

Next, it is described in detail contributions made by Grex and how each one leverages resources usage along execution phases, achieving optimized and load-balanced execution of MapReduce in heterogeneous architectures.

There are five execution phases in Grex: 1) parallel split, 2) map, 3) boundary merge, 4) group, and 5) reduce. Each one is described as follows.

1. **Parallel Split phase** - In the *split* phase, rather than sequentially processing *input data*, Grex performs this task in parallel, copying equally distributed data chunks from *global memory* to the *shared memory* of *thread blocks*. After the data has been copied to *shared memory*, *threads* are able to start identifying the keys in the data chunk.

   By applying the *parallel prefix sum* algorithm [Ngu07] on equally distributed data chunks among *blocks*, all *threads* complete *split* phase simultaneously, with optimal load balancing and without requiring synchronization, since chunks are independent and both locks and atomics are, thus, avoidable.

   At the end of *parallel split* phase, all *threads* of each *block* point to each byte of the data chunks and indicates which bytes point to the beginning of a valid key.

2. **Map phase** - The function to perform mapping is defined by the user in User::map(). The *map* phase executes in the same *kernel* as *parallel split* phase, thus sharing

keys indications previously stored in *shared memory*. For each *thread* pointing to the beginning of a valid key, *map* function copies the position indication from *shared memory* to *global memory* with a thread-unique identifier, thus again without relying on locks or atomics.

Until this stage, the runtime may has emitted only *keys*, avoiding *values* generation and maintenance, and as well memory pressure. The *lazy emit* concept introduced by Grex provides an API function called User::init() so that the user can specify some behavior to precede each phase, and also define in which phase the values must be emitted. The function User::emit(key) is also user-defined.

3. **Boundary merge phase** - The optimization approaches of precedent phases, aimed at equally distributing data chunks among *blocks* to improve load balance, can cause a key to be splitted between two *blocks*, with its beginning in one *block* and its remaining part in another. *Boundary merge* phase is thus introduced by Grex in order to eliminate splitted keys in *global memory*. At its conclusion, this phase ensures a complete and consistent reference to the keys to be used by next phases.

4. **Grouping phase** - This phase groups the intermediate <key,value> pairs by similar keys, and then sorts all groups by key. The user has to provide implementation for the User::cmp() function to indicate the sorting logic. In order to harness parallelism and enhance load balancing, the *parallel bitonic sorting* is used.

   Toward achieving highly efficient access to input data, Grex copies these data to the *texture cache*, managed by hardware and high efficient. During sorting, the positions are referenced by pointers, and finally copied back to *global* memory.

5. **Reduce phase** - In order to define the algorithm to perform reduction, the user must implement the function User::reduce() accordingly to the specific application. Similarly to the *map* phase behavior, Grex distributes intermediate <key,value> pairs equally among the *thread blocks* ensuring load balancing. After each *block* has performed reduction, a second *boundary merge* is run in order to treat pairs whose key is present in more than one *block*.

B.3.1 Buffer management in Grex

In order to optimally exploit the memory hierarchy without imposing complexity to user code, Grex provides two types of data structures through which the user's code can create buffer instances: *constant buffer* and *buffer*.

The *constant buffer* type is always assigned to *constant memory*, requiring its size to be fixed and known at compilation time. The type called simply *buffer* requires a *read-only*

attribute which helps underlying mechanism to find the more suitable memory level. When the attribute *read-only* is set to true, Grex tries to allocate it at *texture cache*, otherwise, *shared-memory* is chosen.

The User::init() function, mentioned in section B.3 as useful for specifying in which phase the *lazy-emit* must be performed, is also used to specify the phase or phases in which each buffer must be enabled or disabled. Thus, this function also provides a high-level API to manage memory usage according to application specific requirements.

B.3.2 Evaluation

The performance of Grex was evaluated on an AMD quad-core CPU using two different generations of NVIDIA GPUs. Results indicate up to 12.4x speedup over Mars and up to 4.1x speedup over MapCG using the GeForce GTX 8800 GPU and GeForce GTX 460 GPU that supports atomic operations needed by MapCG, respectively.