Universitat Autònoma de Barcelona

Escola d Enginyeria

Departament d Arquitectura de Computadors i Sistemes Operatius

Improving Memory Hierarchy Performance on MapReduce Frameworks for Multi-Core Architectures

Submitted by **Tharso de Souza Ferreira** for the degree of Philosophiae Doctor by the Universitat Autònoma de Barcelona, under the supervision of Dr. Juan Carlos Moure and Dr. Antonio Espinosa, done at the Computer Architecture and Operating System Department, PhD. in High performance Computing

Barcelona, September 2013

Improving Memory Hierarchy Performance on MapReduce Frameworks for Multi-Core Architectures

Submitted by Tharso de Souza Ferreira for the degree of Philosophiae Doctor by the Universitat Autònoma de Barcelona, under the supervision of Dr. Juan Carlos Moure and Dr. Antonio Espinosa, at the Computer Architecture and Operating Systems Department, Ph.D in High-performance Computing.

Barcelona, September 2013

Supervisors: Author:

Dr. Juan Carlos Moure Tharso de Souza Ferreira

Dr. Antonio Espinosa

Abstract

The need of analyzing large data sets from many different application fields has fostered the use of simplified programming models like MapReduce. Its current popularity is justified by being a useful abstraction to express data parallel processing and also by effectively hiding synchronization, fault tolerance and load balancing management details from the application developer.

MapReduce frameworks have also been ported to multi-core and shared memory computer systems. These frameworks propose to dedicate a different computing CPU core for each map or reduce task to execute them concurrently. Also, Map and Reduce phases share a common data structure where main computations are applied.

In this work we describe some limitations of current multi-core MapReduce frameworks. First, we describe the relevance of the data structure used to keep all input and intermediate data in memory. Current multi-core MapReduce frameworks are designed to keep all intermediate data in memory. When executing applications with large data input, the available memory becomes too small to store all framework intermediate data and there is a severe performance loss.

We propose a memory management subsystem to allow intermediate data structures the processing of an unlimited amount of data by the use of a disk spilling mechanism. Also, we have implemented a way to manage concurrent access to disk of all threads participating in the computation.

Finally, we have studied the effective use of the memory hierarchy by the data structures of the MapReduce frameworks and proposed a new implementation of partial MapReduce tasks to the input data set. The objective is to make a better use of the cache and to eliminate references to data blocks that are no longer in use.

Our proposal was able to significantly reduce the main memory usage and improves the overall performance with the increasing of cache memory usage.
Keywords. MapReduce; Multi-core; Threads; Main Memory; Virtual Memory

Resumen

La necesidad de analizar grandes conjuntos de datos de diferentes tipos de aplicaciones ha popularizado el uso de modelos de programación simplicados como MapReduce. La popularidad actual se justifica por ser una abstracción útil para expresar procesamiento paralelo de datos y también ocultar eficazmente la sincronización de datos, tolerancia a fallos y la gestión de balanceo de carga para el desarrollador de la aplicación.

Frameworks MapReduce también han sido adaptados a los sistema multi-core y de memoria compartida. Estos frameworks proponen que cada core de una CPU ejecute una tarea Map o Reduce de manera concurrente. Las fases Map y Reduce también comparten una estructura de datos común donde se aplica el procesamiento principal.

En este trabajo se describen algunas limitaciones de los actuales frameworks para arquitecturas multi-core. En primer lugar, se describe la estructura de datos que se utiliza para mantener todo el archivo de entrada y datos intermedios en la memoria. Los frameworks actuales para arquitecturas multi-core han estado disenado para mantener todos los datos intermedios en la memoria. Cuando se ejecutan aplicaciones con un gran conjunto de datos de entrada, la memoria disponible se convierte en demasiada pequena para almacenar todos los datos intermedios del framework, presentando así una grave pérdida de rendimiento.

Proponemos un subsistema de gestión de memoria que permite a las estructuras de datos procesar un número ilimitado de datos a través del uso de un mecanismo de spilling en el disco. También implementamos una forma de gestionar el acceso simultáneo al disco por todos los threads que realizan el procesamiento.

Por último, se estudia la utilización eficaz de la jerarquía de memoria de los frameworks MapReduce y se propone una nueva implementación de una tarea MapReduce parcial para conjuntos de datos de entrada. El objetivo es hacer un buen uso de la caché, eliminando las referencias a los bloques de datos que ya no están en uso.

Nuestra propuesta fue capaz de reducir significativamente el uso de la memoria principal y mejorar el rendimiento global con el aumento del uso de la memoria caché.

Palabras clave. MapReduce; Multi-core; Threads; Memoria Principal; Memoria Virtual.

Resum

La necessitat d analitzar grans conjunts de dades per part de diferents tipus d aplicacions ha popularitzat la utilització d entorns de programació simplificats com ara MapReduce. La seva popularitat actual es justifica pel fet de proporcionar una abstracció útil per expressar processament paral·lel de dades i per oferir una forma efectiva de fer transparent per al programador d aplicacions detalls sobre la implementació de les tècniques de sincronització, tolerància a errors i balanceig de càrrega.

Els entorns d execució MapReduce també han estat adaptats per als sistemes multi-core i de memòria compartida. Aquests entorns proponen dedicar un core de CPU diferent a cada tasca map i reduce i així executar-les de forma concurrent. Per una altra part, les etapes Map i Reduce comparteixen una estructura de dades comuna on s aboquen la gran part de computacions realitzades.

En aquest treball descrivim les limitacions més importants dels entorns actuals d execució d aplicacions MapReduce en entorns multi-core. Primer descrivim la rellevància de les estructures de dades utilitzades per mantenir les dades intermitges en memòria. Els entorns actuals estan dissenyats per mantenir tots els resultats intermitjos en memòria principal. Quan les aplicacions en execució han de tractar grans quantitats de dades d entrada, la quantitat de memòria disponible no és capaç d emmagatzemar la quantitat d informació necessària i es produeix un greu problema de rendiment.

Proposem un nou subsistema de gestió de memòria per permetre que aquestes estructures de dades puguin processar una quantitat il·limitada de dades amb l ús d un mecanisme de spill al disc. També presentem un altre mecanisme per a gestionar els accessos a disc que generen tots els threads que participen al còmput.

Finalment, també hem estudiat l ús efectiu de la jerarquia de memòria per part de les estructures de dades dels entorns MapReduce multi-core i proposem una nova implementació de l execució parcial de tasques MapReduce a les dades d entrada. L objectiu és fer un millor ús de la memòria cau, així com eliminar les referències als blocs de dades que ja no es fan servir.

La nostra proposta ha estat capaç de reduir significativament la utilització de la

memòria principal, així com millorar el rendiment general de les aplicacions amb un increment en la utilització de la memòria cau.
Paraules claus. MapReduce; Multi-core; Threads; Memòria principal; Memòria virtual.

Resumo

A necessidade de analizar largos conjuntos de dados desde distintos tipos de aplicaçoes tem promovido o uso de simplicados modelos de programaçao como MapReduce. A atual popularidade é justificada por ser uma abstraçao útil para expressar processamento paralelo de dados e também por efetivamente esconder detalhes de sincronizaçao, tolerancia a falhas, e gerenciamento de balanceio de carga do desenvolvedor da aplicaçao.

Frameworks MapReduce também foram portados para sistema multi-core e de memória compartilhada. Esses frameworks propoe dedicar os diferentes cores de uma CPU para cada tarefa Map ou Reduce a ser executada concorrentemente. As fases Map e Reduce também compartilham uma estrutura de dados comum onde o processamento principal é aplicado.

Em este trabalho nós descrevemos algumas limitaçoes dos atuais frameworks para arquiteturas multi-core. Primeiro, nós descrevemos o relevante da estrutura de dados utilizada para manter todo o arquivo de entrada e dados intermédios em memória. Atuais frameworks para arquiteturas multi-core foram desenhados para manter todos os dados intermédios em memória. Quando se executam aplicaçoes com grande conjunto de dados de entrada, a memória disponível torna-se muito pequena para armazenar todos os dados intermédios do framework apresentando assim uma séria perda de desempenho.

Nós propomos um subsistema de gestao de memória que permita as estruturas de dados processar um ilimitado conjunto de dados através do uso de um mecanismo de spilling no disco. Nós também implementamos uma maneira de gerenciar o acesso concorrente ao disco de todos os threads que estao participando do processamento.

Finalmente, nós estudamos o uso eficaz da hierarquia de memória dos frameworks MapReduce e propomos uma nova implementaçao de uma tarefa Mapreduce parcial para o conjunto de dados de entrada. O objetivo é fazer um bom uso da cache em elimiar as referencias para os blocos de dados que nao estao mais em uso.

Nossa proposta foi capaz de significativamente reduzir a memória principal utilizada e melhorar o desempenho geral com o incrememento do uso da memória cache.
Palavras chave. MapReduce; Multi-core; Threads; Memória Principal; Memória Virtual.

Acknowledgements

Four years ago i started this adventure across the ocean, away from home, with a whole new world ahead. First and foremost, i would not be here without the support of my parents and my sisters. For all the confidence, patience and faith deposited in me since the first moment. I would not be here without the support of you.

To Alexandre Strube, for introducing me to the world of research, for all the encouragement in the beginning of this journey, essential to be able to reach the end, i will be eternally grateful.

To Emilio and Lola, for receive me so well and make me feel safe so far from home, for all the help and advice, permitting believe and always continue. No thanks here would be enough to express my gratitude.

To Juan Carlos, Toni and Porfi, for all patience, guidance and productive discussions. I have learned so much from you, my whole way of looking at things has changed. Thank you for everything, especially for the patience with my immaturity in the research world.

To Carlos, who four years ago was the first to understand me, for advices, for long hours listening to me and helping solve any problems, thank you for the immense help throughout this journey. Thank you for becoming this great friend, i hope to have your friendship forever.

To Sandra, Andrea, Eduardo, Aprigio, César, Joao, Julio, Hugo, Marcela, Leo, Gemma, the best friends i could have here, were my family away from home. I will miss every minute by your side, thank you for making these past four years the best i could have.

To Sophie, for all the patience, always listening to me talk about work, care and worry about me. By always pushing me forward and believing in me.

To Bruno, the brother that life gave me. These last four years would never be complete without you always close.

To all my friends and partners at CAOS, for all i have learned of each of you.

Thank you all!

Contents

List of Figures

List of Tables

List of Algorithms

List of Equations

Chapter 1

Introduction

In the early years of modern computing, when the first electronic computer was developed, the famous ENIAC in 1946 occupied a huge room at the University of Pennsylvania in the United States. At that time many of the most experienced technologists thought, except for advanced military applications, such as breaking encryption codes or calculate trajectories of projectiles, no one would want something as big as that. As technology has advanced, the machines invaded the civilian world. Today there are computers everywhere, it is hard to look at a side where there is not one. The evolution of technology has reduced the physical size of computers and increased performance on a single workstation, however the increased speed of the processors found growing barriers, as the physical limits were reached. Today it is not possible to increase the frequency of processors as was observed in the 90. The need for processing capacity is no longer achieved by faster processors and now being obtained by increasing the number of operations performed simultaneously, i.e. parallel processing.

Parallel systems are computer environments composed of a set of processing units that work together in a simultaneous way with the objective of solving a problem. Parallel environments are attractive because of their processing throughput, and their larger amount of main memory and storage capacity. Such systems have emerged as a solution to solve problems in totally different areas [12]. The set of typical applications of these systems include, physical systems simulations, and technical computations in the diverse areas of the chemistry, physics, biology, materials science, climatology, astrophysics and subsurface transport [5].

However, the use of parallel systems becomes useless if the necessary applications are not properly designed. In general, the idea of processing a data set consists of simple strategies, but becomes complicated when it is necessary to process large data sets in a distributed way.

MapReduce[11] is a programming model for expressing distributed computations and a distributed execution framework for processing large amounts of data [29]. In the MapReduce programming model, the user defines an algorithm using only two functions, Map and Reduce.

The Map function takes an input data set and produces a set of intermediate key/value pairs. The Reduce function receives an intermediate key K and a set of values for that key. It aggregates together these values to create an output for the problem. Typically just zero or one output value is produced per Reduce operation. In general, reductions must wait until all Map operations are completed.

The MapReduce Model uses combiner functions to allow the local aggregation of values of each Map. The complete MapReduce model scheme can be seen in Figure 1.1.

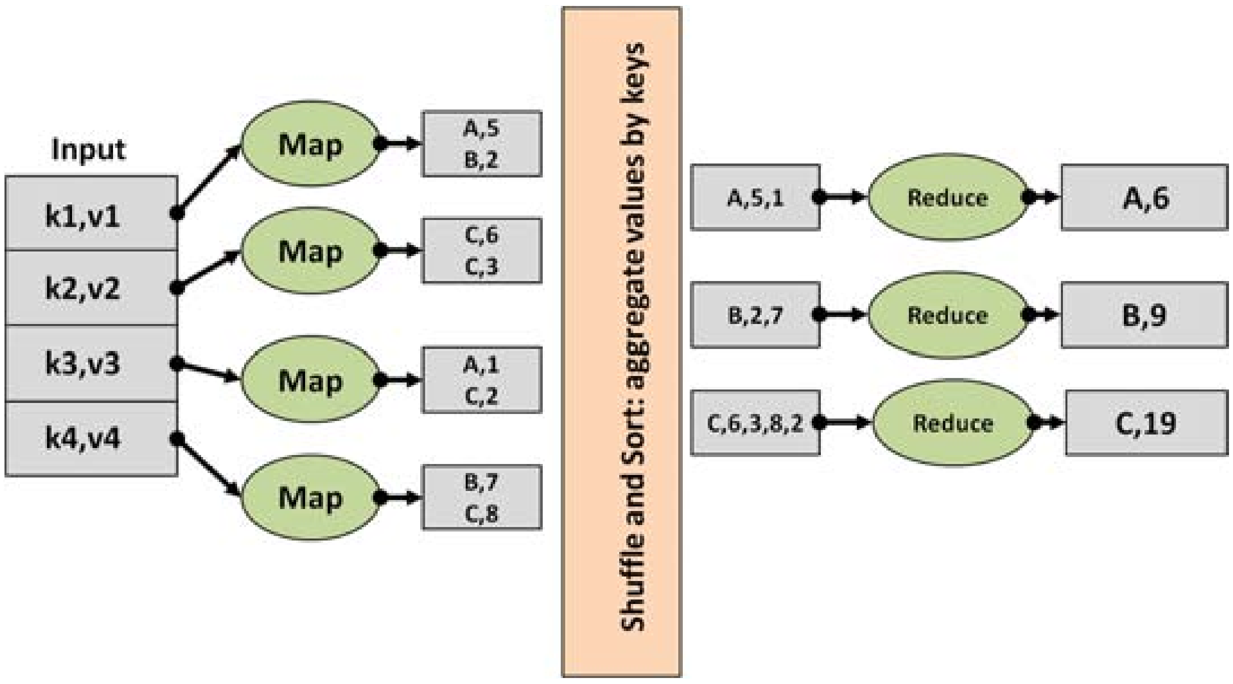

Figure 1.1: Basic data flow for MapReduce programming model.

The main advantage of MapReduce is to provide a simple abstraction for the developer, transparently handling most of the internal implementation details in a scalable, robust, and efficient manner[29].

The goal of MapReduce was focused on hiding complex problems of parallel execution like fault tolerance, data distribution and load balancing [2][26]. The MapReduce usage allows programmers to exclusively search for solutions to specific problems, as opposed to how those computations are actually carried out or how to get the data to the processes that need them.

MapReduce execution gets parallel speedup by running many Maps concurrently, each on a different part of the input file, and by running many Reduces concurrently on different keys. In short the charm of MapReduce is that the programming model hides all the concurrency management problems from the programmer[30].

In cluster-based MapReduce frameworks like Hadoop[44], the idea is to spread data across the local disks of nodes and run processes on nodes that hold the data. The complex issues of task managing storage in such environment are handled by using a distributed file system assembled over MapReduce. In most Hadoop executions, a distributed file system is used to store both the input to the Map phase and the output of the Reduce phase [10].

In Hadoop, each Map task is executed on the node that holds the input split, so that the Map will be processing local data. If it is not possible to run a Map task on local data, it becomes necessary to send input key/value pairs across the network.

Map and Reduce tasks run in isolation without any mechanism for direct communication. During the shuffle and sort phase, the intermediate key/value pairs are merged, sorted and then copied from the Map to the Reduce tasks nodes.

The need for faster processing required on a simple workstation in recent years, provided a growth of multi-core and multi-processor systems. The advances in technology, circuits integration and the limited performance of wide issue intruction execution, made multi-core and multi-processor shared memory architectures become the basic building block of parallel systems. They embeds multiple processing cores into a single system to exploit growing amounts of thread-level parallelism for achieving higher overall chip-level instruction-per-cycle rates.

By working over multi-core, cluster-based MapReduce frameworks exploit the parallelism offered by current parallel workstations by running multiple, independent MapReduce instances on the same node. The framework schedules tasks dynamically across the available processors in order to achieve parallelism, load balance and maximize task throughput.

Typical programming view of MapReduce frameworks can be divided into two set of functions, the user and system set. The Map user-defined function is applied in parallel over the input data, producing a list of intermediate key/value pairs. The Reduce user-defined function is applied in parallel to all intermediate pairs with the same key. The system set is composed of system-defined functions that are responsible for exploiting the available parallelism, data distribution and intermediate key/value management[24].

In shared-memory systems, the intermediate data produced by the Map phase and the Reduce phase output are stored in shared memory. Compared to cluster-based approaches, the multi-core has as main advantage its faster core-to-core communication through shared memory[7][34]. For multi-core MapReduce frameworks the most important aspect is that the Map and Reduce tasks handle with intermediate *key/values* through a shared memory. Unlike the cluster-based where the handling takes place through the distributed file system and network.

The use of a data structure organized in rows and columns makes Map and Reduce tasks independent. Each Map task has its own row in a hash table. Using a unique hash function, it finds the key position in the hash table. By processing the keys per column, each Reduce task has access to all the repeated keys produced from different Map tasks, as can be seen in Figure 1.2. Such a strategy avoids data copying, as reduce tasks reuse the data structure product of the Map phase.

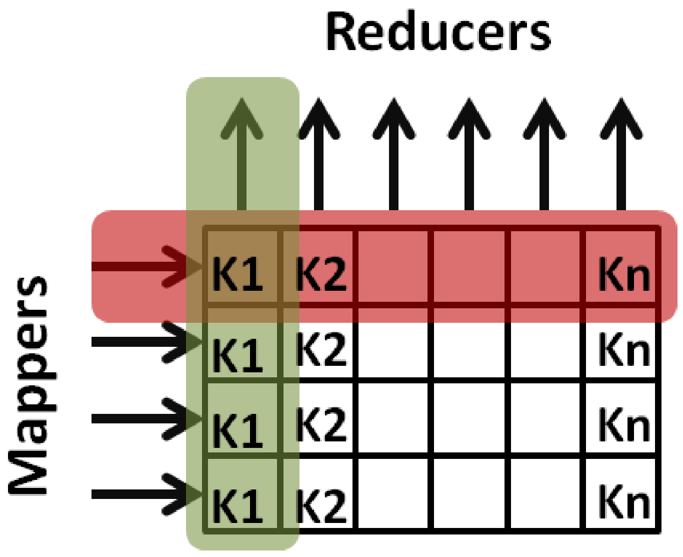

Figure 1.2: Intermediate data structure organization.

Metis[30] and Phoenix++[42] were implemented to support multi-core MapReduce implementations,

In general, Phoenix++ and Metis work the same way, by using hash tables as a base, which allows quick key lookup with O(1) complexity. Reduce tasks have simple and fast access to all values of the same key by accessing columns of the data structure. Moreover, Phoenix and Metis add rows to the hash table to increase the parallelism achieved, making each worker has its own memory space.

In order to minimize the number of repeated keys in the intermediate data structure, Phoenix++ and Metis implement a combiner function, which aims to make a local thread reduction, saving main memory. After running the combiner function, the framework no longer stores a set of values and stores only a combined value of a given key.

The main differences between Phoenix++ and Metis is how they allocate the intermediate data structure and how they store the keys/value pair per hash entry.

In order to allocate the intermediate data structure, Metis tries to predict its size. First samples a part of the input and calculates the number of different keys, then allocates the hash table to have a specific amount of entries so that each entry contains a bounded number of keys. Phoenix++ on the other hand uses a resizable hash table, which can scale depending on the key distribution.

To store the keys/value pairs in hash entries, Phoenix++ uses a vector and Metis uses a B+tree index. The vector usage is more advantageous to store sets of keys from non repeated keys input, that do not require to be sorted. On the other hand, the use of trees is most advantageous for large sets of repeated keys.

Current MapReduce frameworks deal with some major performance problems. MapReduce frameworks are not designed to be efficient if the intermediate data do not fit in main memory.

Large input MapReduce applications provide a key distribution which causes the intermediate data structure to grow. On multi-core environments, such data structures can consume all the available main memory. When the available memory is not large enough to store the data structure, it creates a paging file on the hard disk, starting the use of virtual memory. Continuous hard disk access makes the framework have a severe performance loss.

Moreover, dealing with large datasets in main memory does not make scalable MapReduce frameworks. Such behavior is presented because MapReduce frameworks present a high contention when using threads across multiple chips. Due to non-uniform memory access effects, the cost are increased when more chips are used, obtaining a worse performance with increasing number of threads[45].

Another major problem of multi-core MapReduce frameworks, it is the low cache memory data-locality, when it is necessary to process large data sets. The large number of different keys produced and the growth of the intermediate data structure makes it difficult to keep all relevant data in cache memory.

In order to solve the problems presented in multi-core MapReduce frameworks, we designed and implemented a solution that manages, handles and stores keys in a more efficient intermediate data structure.

In this work we describe an engineering technique for the MapReduce runtime library to minimize the memory footprint and execution-time overheads when it is necessary to manipulate large data sets.

The proposed approach is divided into two major parts: a memory management system for the intermediate data structure with a set of parameters and limits to synchronize spilling of multiple threads to the hard disk, and a partial MapReduce system optimized for better cache memory data-locality.

The memory management system for the intermediate data structure prevents that the amount of intermediate data generated consumes the entire main memory. Our proposed approach monitors the main memory usage on runtime, until a pre-defined limit is met. Then it moves key/value pairs between main memory and hard disk until memory is available again. Furthermore, using a set of parameters and limits, our mechanism synchronizes the threads, thus avoiding any competition when spilling on the hard disk. Our memory management mechanism adapts to changes in the available memory in the execution environment, caused by sharing the system between different applications.

The proposed partial MapReduce system aims to increase the cache memory data cache locality. The mechanism groups execution of Map and Reduce phases such that the Reduce phase begins as soon as the number of keys generated in the Map phase reaches a pre-defined limit. Such a strategy assures that the most relevant data is maintained into cache memory. The main issues of this proposed mechanism are:

- Partially stop the production of keys by the Map phase when reaching a cache size-fit limit;
- Reduce the data produced partially so that the relevant data can fit in the last-level cache, and only then return to run the Map phase;
- By restricting the amount of keys produced by the Map phase and run the Reduce phase before the end of all Map operations it is possible to keep fewer keys repeated in memory;
- Perform a global reduction when all the input has already been processed.

1.1 Objectives

The main objective of this work is to design, implement and evaluate a mechanism to improve the performance of multi-core MapReduce frameworks through efficient handling of intermediate data structures. The strategy to solve this problem focuses in improving the integration in memory hierarchy, hard disk, main memory and cache memory. The specific objectives of this work are numbered as follows:

1. Evaluate Metis and Phoenix++ performance using MapReduce applications that process large data sets and identify the potential bottlenecks in multi-core MapReduce frameworks;
2. Design and implement a memory management mechanism for multi-core MapReduce frameworks that deals with changes in the amount of available memory and reduce the main memory usage when the data do not fit in main memory;
3. Design and implement a mechanism to store and manipulate keys in a intermediate data structure on MapReduce frameworks, which achieves better performance by increasing data locality;
4. Propose a set of parameters that must be added to multi-core MapReduce frameworks to manage the use of cache memory, main memory and hard disk for each specific system;
5. Evaluate the performance improvement of the proposed solutions.

1.2 Work Organization

Through the objectives previously proposed the outline of the work follows this organization.

Chapter 2: Thesis Background. Presents the MapReduce programming model and the related work on multi-core architectures. Introduces a brief explanation of intermediate data structures and the influence of key distribution, in addition to the problem with the growth of intermediate structures.

Chapter 3: Proposal of Virtual Memory Management. Presents the proposed mechanism to improve Virtual Memory Management, as well as details in the implementation of the proposal.

Chapter 4: Proposal of Memory Hierarchy Management. Presents the proposed mechanism to improve the memory hierarchy integration, as well as details of the implementation, advantages and main points.

Chapter 5: Evaluation. Describes the used scenario and provides an explanation of the results of our proposal as well as the types of applications and workloads.

Chapter 6: Conclusions. Finally, we present a conclusion of this work, future work and open lines.

Chapter 2

Thesis Background

In this chapter we first introduce the MapReduce programming model describing its principles and some relevant details not visible to programmers. We start by introducing the MapReduce programming phases and the internal operations that frameworks must provide in between them. Then, we describe the existing multi-core MapReduce implementations attending their architectural differences, focusing our description in the specific details of the intermediate data structures they use. Of special importance are key distribution and how current data structures tolerate data growth and variations of key distributions. Finally, we discuss the influence of the memory hierarchy in multi-core MapReduce frameworks

2.1 MapReduce Programming Model and Framework

MapReduce[11] is a programming model originally developed by Google for expressing distributed computations on massive amounts of data. It is also a private framework for large-scale data processing on clusters of commodity servers [29][33][46]. Its main objective is to facilitate the task of parallel programming. It proposes a way of abstracting the details related with data distribution, load balancing and fault tolerance through a clear and simplified API of a distributed framework. The work of the programmers is to design a solution to their problems by applying the MapReduce model. Then, the framework is in charge of parallelizing the computation among the computing resources [14][15].

Since its introduction of MapReduce, apart from its intensive use by Google for tasks involving large datasets each day, it has gained popularity by the widely use of Hadoop [44], an Apache project MapReduce implementation. Hadoop has been used regularly by public and private institutions like Yahoo, eBay, Facebook, Twitter, Amazon and IBM [19].

All MapReduce computations are arranged along a set of operations that take advantage of a set of distributed computing resources in a cluster[35][31]. In Google and Hadoop implementations, there are two main subsystems inside the framework:

- Task management environment: in charge of forking, synchronization and elimination of computation worker tasks;
- Distributed file system: MapReduce frameworks use their own file systems like GFS [18][27] or HDFS [44] to store datasets. They provide read, write operations on a distributed data repository and allow the replication of blocks as a fault tolerance method.

Once a problem is specified in terms of the MapReduce programming model, we can also use that design to execute applications on any shared memory, multicore CPU computer [9]. Then, the programming model becomes our point in common with the cluster original implementations. This model is based on the following concepts [25]:

1. Iteration over the input.
2. Computation of key/value pairs from each piece of input.
3. Grouping of all intermediate values by key.
4. Iteration over the resulting groups.
5. Reduction of each group.

The terms *Map* and *Reduce* come from Lisp operations with the same name. In practice, Reduce here can be significantly different from the Lisp s in that the output does not need to be smaller than the input, and may in fact be higher [9].

The most viable approach to efficiently tackling large-data problems today is to divide and conquer, which is a fundamental concept in computer science that is introduced very early in typical undergraduate curricula. The basic idea is to partition a large problem into smaller sub-problems. To the extent that the sub-problems are independent [1], they can be tackled in parallel by different workers, represented by threads in a processor core, cores in a multi-core processor, multiple processors in a machine, or many machines in a cluster.

MapReduce frameworks perform parallel processing first by distributing the data among a group of nodes. Then, each worker applies the same computation to a different local input data block [36].

The MapReduce programming model has been effective in solving some problems because it easily fits many solutions to large data analysis. Expressing computation in terms of Map and Reduce operations exhibits the inherent parallelism of an application, allowing the efficient exploitation of distributed machines by executing many tasks in parallel [4]. The importance of MapReduce is providing an effective tool to tackle the problem of large-scale data processing. It represents an effective new way of organizing the computations needed at a very large scale [29].

2.1.1 MapReduce Programming Model

As previously mentioned, the MapReduce programming model is simple. The input data format is application-specific, and must be specified by the user [22][37][28]. In principle, programmers must design a solution using Map and Reduce operations. From the point of view of the programmer, the model can be expressed in the following stages that are shown in Figure 2.1.

- Input is split in blocks and each map task receives one block as its input;
- Map tasks receive a key/value pair (a record) from each entry of its input block. For text inputs, the key is the input file line number while the value is the text of the line;
- Map tasks apply some computation to each key/value input pair and generate zero, one, or several different key/value outputs;
- Reduce tasks input is a unsorted set of all values generated by all map tasks associated with a specific key;
- Reduce tasks apply some computation, usually some kind of reduction operation such as summation, to the set of values and generate an output as a solution to the problem;
- The disjoint set of keys and values received by each reduce task is sorted by key.

Some relevant consequences are derived from this model.

- Map tasks are executed on local blocks of data without any interdependency;
- At the end of the execution of the map tasks, they generate a set of data called intermediate key/value pairs. This set of values is automatically shuffled (merged) and sorted by the framework before providing input to reduce tasks;

Reduce tasks cannot generate a solution until all map tasks have finished their execution;

Reduce tasks do not have any interdependency.

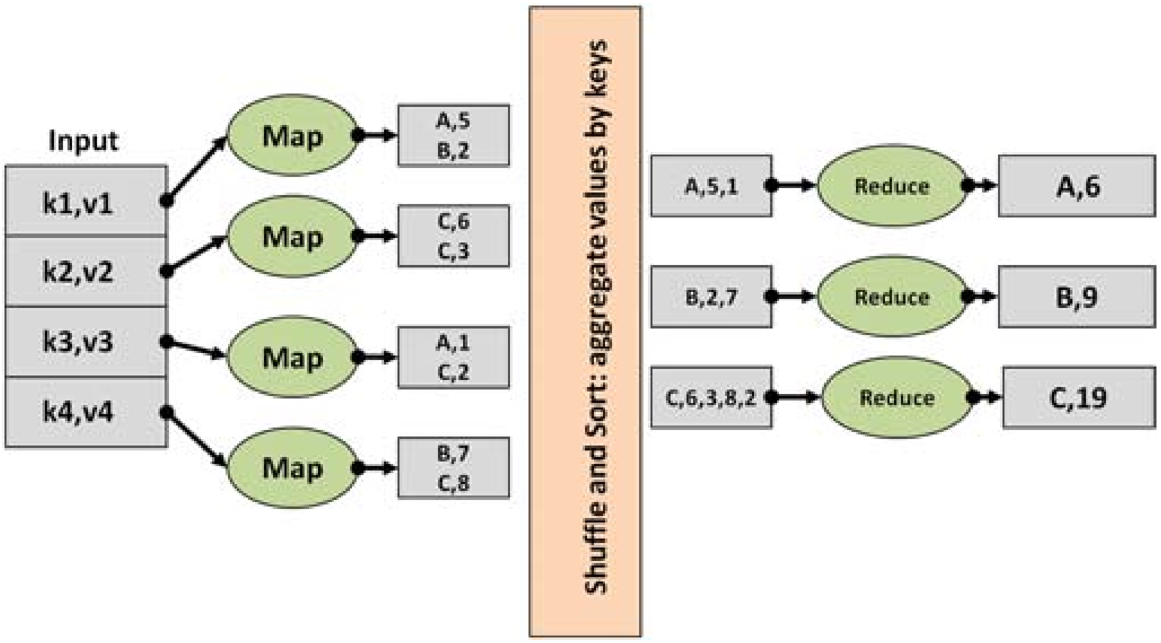

Figure 2.1: Full Programming Model Map Reduce Schema.

In addition to the Map and Reduce operations, there are helper functions to increase efficiency: combiner and partitioners, as shown in figure 2.2. The combiner functions allow the aggregation of values for the same key at the end of the map execution. The partitioner functions allow us to define the distribution of keys among the reducers.

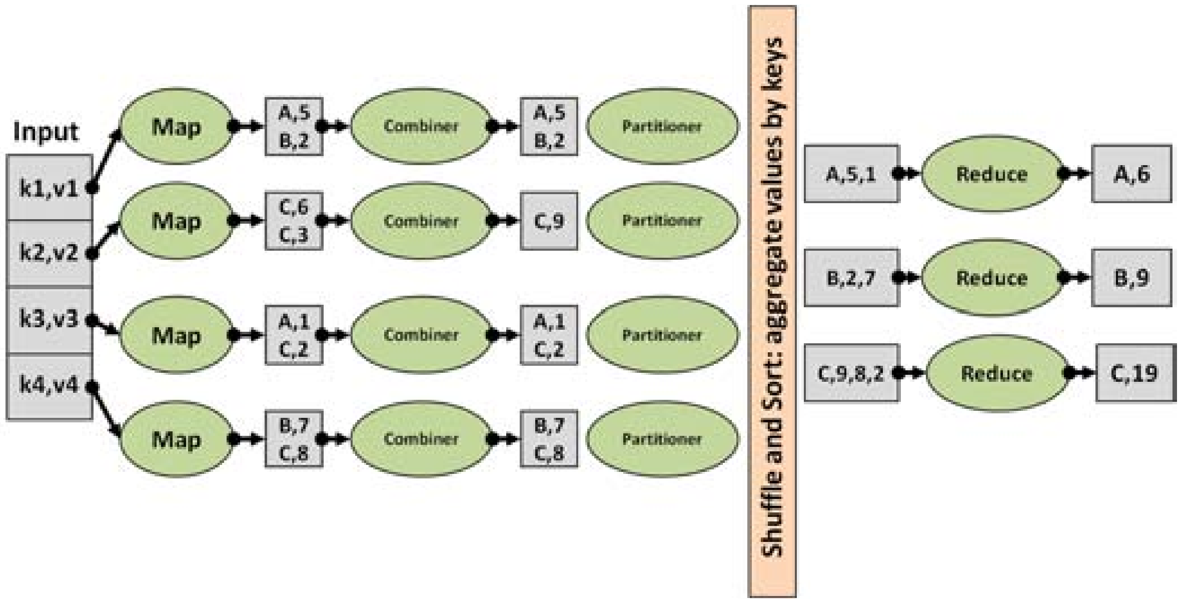

Figure 2.2: MapReduce Model With Combiner Functions.

2.1.2 Multi-core MapReduce Programming Model

There are different implementations of MapReduce for parallelizing computing heterogeneous and multi-core environments, for FPGA chips as the FPMR [40], GPU[4], [20], [21], [41], [13], [32], [6], [3] and especially for multi-core architectures and multi-processor shared memory as Phoenix[38][45], Phoenix++[42] and Metis[30], where we are focusing our work.

As for cluster systems, multi-core MapReduce implementations require the programmers to design their algorithms using two functions: map and reduce. Multi-core MapReduce frameworks are available for programmers in the form of libraries, while Hadoop implementations provide an API via a Java runtime system in the form of a set of classes and objects.

Typically, the library partitions the input into fixed size blocks and calls the specific user map implementation on each block. The framework usually executes one thread per CPU core for running map tasks.

Each map task generates a list of intermediate key/values as its output. When all input blocks are processed by map tasks, the library starts creating reduce tasks, again one per core, to call the specific user reduce implementation. This time, the library calls reduce once for each distinct key produced by the map tasks. Each reduce generates a set of output key/value pairs. Finally, there is a merge step where the framework sorts all reduce outputs by key.

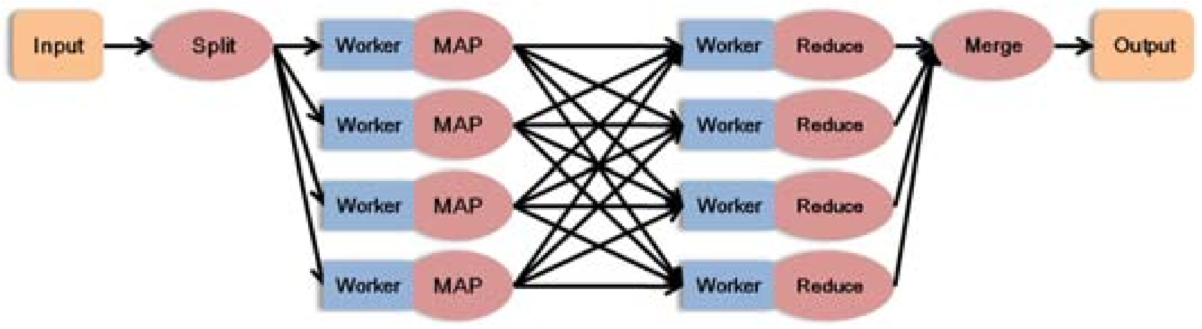

Figure 2.3: MapReduce Programming Model.

The main benefit of this model is simplicity. The programmer provides a simple description of the algorithm that focuses on functionality and not on parallelization [38]. Frameworks make use of multi-core resources by running many map tasks concurrently, each on a different input block, and by executing many reduce tasks on different keys.

2.2 Multi-core MapReduce frameworks

The performance of MapReduce frameworks such as Hadoop is managed by the configuration of a large number of parameters. System managers are responsible of modifying default values to increase the efficiency of the platform for running a specific set of applications.

It is very important to have a good implementation of the sort-merge algorithm that must join all intermediate key/values generated by map tasks, sort them and send the joined values to each reduce task. This operation depends on the actual bandwidth of both disk and network systems [11].

The multi-core MapReduce frameworks can be understood as interleaved models of parallel and serial computation operating in steps. At each step, the user-defined serial functions are executed independently in parallel. Each step consists of the consecutive execution of the main phases, Map and Reduce [19].

The multi-core MapReduce frameworks have been designed to use the shared main memory of the system. All map and reduce tasks operate in a shared data structure that must be stored in main memory. This approach takes advantage of the rapid communication between tasks through shared memory [7]. Therefore, the design and performance of the data structures used to store and retrieve intermediate data are crucial in overall system performance [45].

MapReduce frameworks for multi-core architectures typically make extensive use of pointers. All the framework main functions provide pointers to the input/output data buffers. The use of pointers allows the data manipulation through all the phases without the need for copying large amounts of data [38]. For instance, keys are not copied to intermediate data structures. Instead, pointers to keys in the original input are used as seen in Figure 2.4. An additional benefit is to simplify the implementation for any key data type and size.

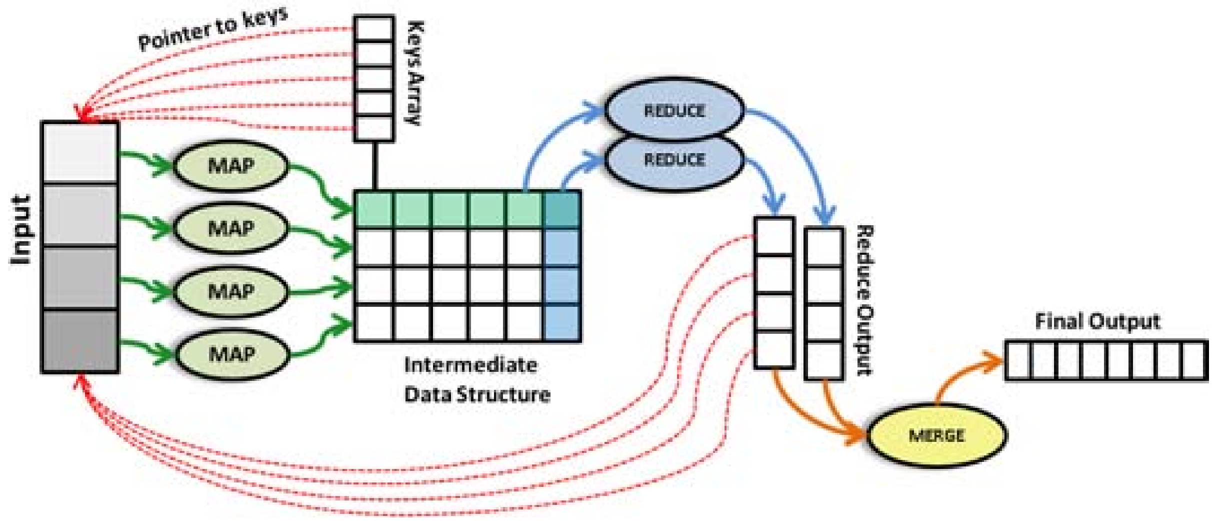

Figure 2.4: Multi Core MapReduce Framework.

2.2.1 Phoenix

Phoenix implements MapReduce for shared-memory systems [38][45]. It presents an API to the programmer through a framework that deals with the issues of parallelization. It consists of a set of functions that allows the programmer to initialize the framework and link it with the application to be executed. The Phoenix runtime system spawns multiple processes or threads that apply Map and Reduce functions concurrently across the elements of the input dataset [45]. The API guarantees that within a partition of the intermediate output, reducers will process the input pairs in key order, making it easy to produce a sorted final output that is often desired. The Map and Reduce functions are declared as void pointers wherever possible, providing flexibility in their declaration and fast use without conversion overhead.

To start the Map stage, the framework uses the Splitter function to divide input into equally sized units to be processed by the Map tasks. The Splitter function is called once per Map task and returns a pointer to the input data that will be processed. When a MapReduce application starts a job, it spawns multiple workers and binds them to different CPU cores. The Map tasks are allocated dynamically to workers and each one emits intermediate *key/value* pairs, which are stored in an intermediate data structure.

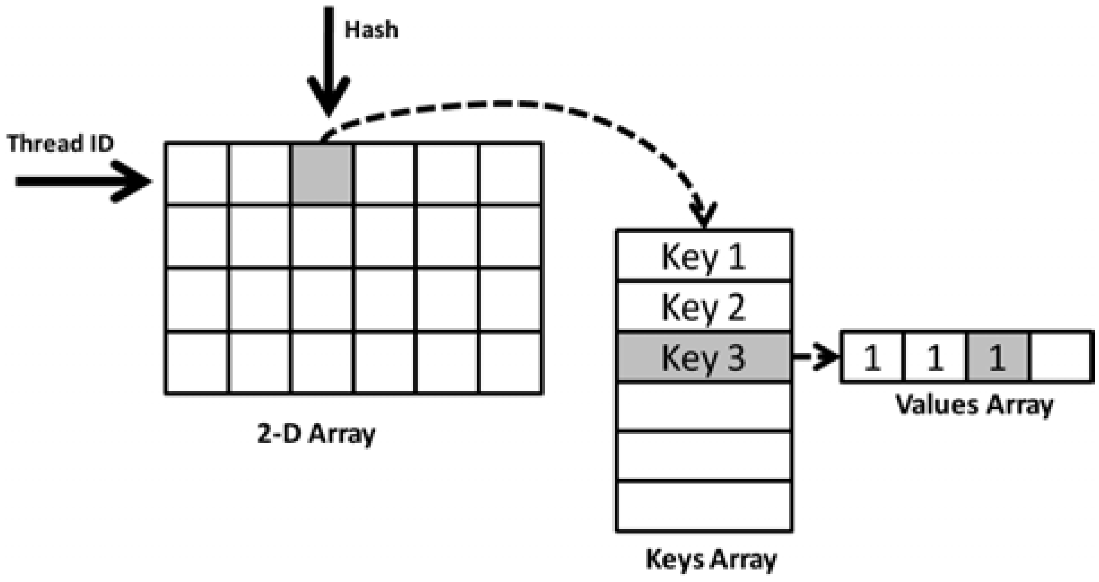

Figure 2.5: Phoenix intermediate data structure Map Phase.

Phoenix uses a matrix as intermediate data structure as shown in Figure 2.5. Each worker responsible of a map task accesses a different row of the matrix.

When the Map worker emits a key, it uses a hash function to determine a position in its own row. This position contains a pointer to a keys array where the emitted key is searched. The keys in the keys array are kept sorted to accelerate this search process. For each key contained in this keys array there is another array where values are stored. When a key already exists in the array of keys, the new value is just appended to the array of values. If the emitted key is not found, (a pointer to) the actual key is stored in the keys array. When a keys array is full, it is dynamically doubled. The same happens with the values array.

In summary, each map worker uses a hash function shared by all map workers. It accesses a row of pointers as a fixed-sized hash table. For each hash element, there is a hash bucket containing keys. Each array of keys is implemented as a contiguous buffer. The framework must wait for all Map tasks to complete before initiating the Reduce phase.

In the Reduce phase, aiming to quickly join all the equal keys and their values, each

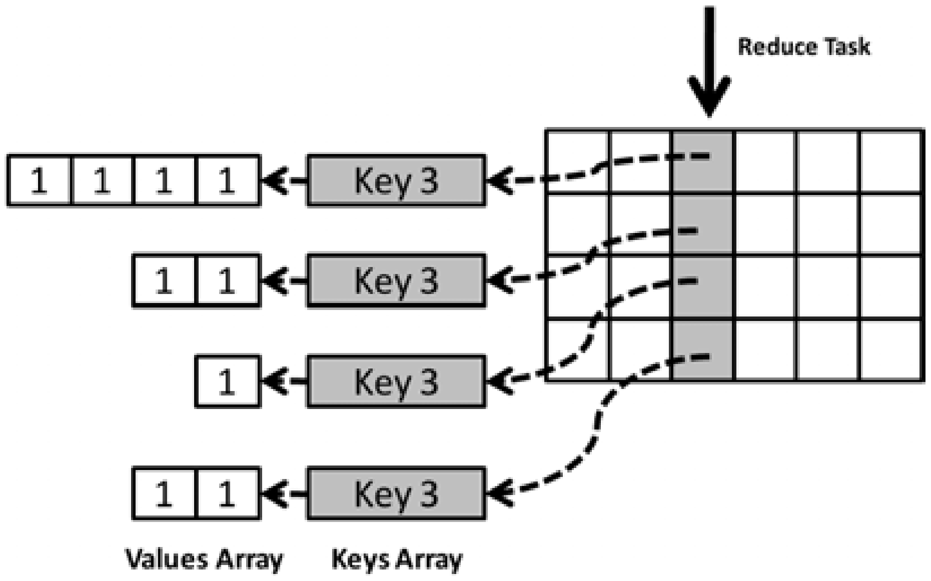

Figure 2.6: Phoenix intermediate data structure Reduce Phase.

reduce worker retrieves all the keys and values of a column of the matrix as shown in Figure 2.6. By assigning different workers to different columns all keys and its associated values are merged. When a reduce worker finishes reduction in one designated column, the framework is responsible of assigning another column to be processed.

The input of each reduce task is a set of arrays already sorted by key and at the end of the reduce phase, each key has only one value associated with it. In the last step the final output from all reduce tasks is merged into a single buffer, then sorted by keys. To Phoenix, this Merge phase is considered optional.

2.2.2 Phoenix++

As a complete revision of the original Phoenix, Phoenix++[42] was developed to improve execution speed through modularity in critical sections. First, it provides a flexible intermediate key/value storage abstraction that permits to adapt the specific implementation to the features of the workload. And second, it includes a more effective combiner implementation that can minimize memory usage.

As an improvement from the first implementation, Phoenix++ considers different types of key distributions and basic execution parameters, which directly influence the use of shared memory.

Phoenix++ classifies MapReduce applications by the number of keys that the map tasks will emit:

1:1 - each task outputs a single, unique key;

***:k -** any map task can emit any of a fixed number of keys, k;

***:* -** any map task can emit any key, where the number of keys is not known before execution.

The idea of knowing the type of key distribution is to allow the user to choose a type of intermediate data structure to be used from a list of options. Phoenix++ provides three types of intermediate data structure to choose from

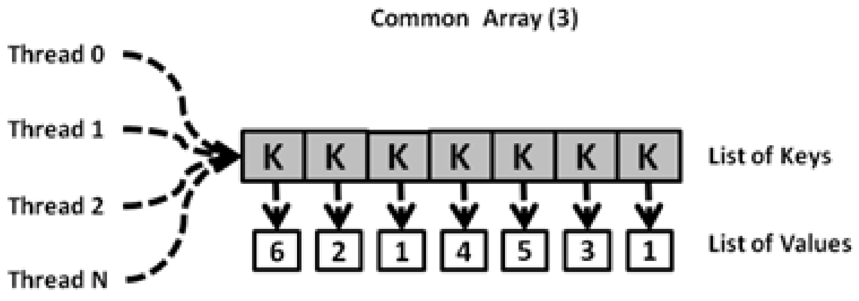

Figure 2.7: Phoenix++ non-blocking shared array.

Common Array (1:1): a non-blocking array structure shared across all threads, as it is shown in Figure 2.7. The common array leverages the fact that each emitted key is unique and all threads can write into the same array without any synchronization.

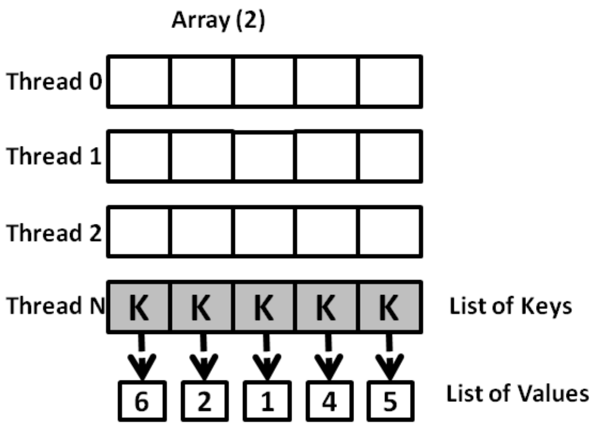

Figure 2.8: Phoenix++ fixed-size thread-local array.

Array (*:k): a fixed-size thread-local array implementation, as it is shown in Figure 2.8, which requires the keys to be integers within an a priori known range. For tasks with a known, small key cardinality, insertion into a fixed size array avoids the cost of hashing keys and repeatedly checking if the hash table should be resized. Provides an increase in performance in cases where there is a small computation per task.

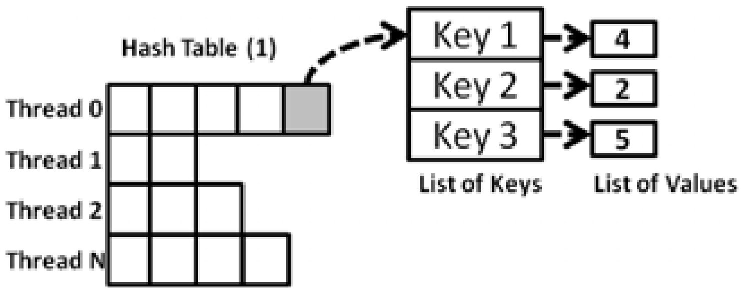

Figure 2.9: Phoenix++ variable-width hash table.

Hash table (*:*):This is the main improvement from the previous Phoenix version with the objective to improve the search in large keys arrays. It is a variable-width hash table implementation where each map thread can resize its own hash table, as it is described in Figure 2.9. This ensures that insertion complexity is kept in the order of O(1) even in the presence of an unexpectedly large number of keys. With this structure, Phoenix must prevent the loss of correspondence between the number of keys of each row, which would make it difficult to group values of the same key, through the different threads. At the end of the Map phase all keys and values are copied to a fixed-size hash table, where the number of columns is the same as the number of reduce tasks to launch.

The use of distinct data structures to adjust the framework to each workload key distribution improves the efficiency of handling specific known applications. In those cases, the volume of data to be processed may not be necessarily large.

In addition to using different intermediate data structures Phoenix++ also has a combiner function that performs a local reduction per thread, after each key/value pair is emitted by the map function. The combiner function reduces the amount of memory required when having multiple values for the same key inside each Map task.

2.2.3 Metis

Using Phoenix as a base, Metis[30] is an improved multi-core MapReduce framework. To increase the performance with most types of workloads, Metis makes use of three main strategies

The use of combiner function;

A new data structure to store intermediate key/values;

A key/value distribution prediction phase.

All Map workers have a hash table to store keys and values and use the same hash function to ensure a good distribution of keys throughout the hash table. To avoid competition between different threads on the same memory region, each thread has its own Hash-tree, which is represented by each row of the matrix shown in Figure 2.10.

Each hash table entry points to a B+tree to store keys and values. The idea is to achieve a theoretical O(1) search complexity accessing the hash table. Then, assuming N keys stored in each hash bucket, traversing the tree to find the key position has O(logN) complexity. If the key already exists, the worker adds a new value to the key list. If the key does not exist, then the new key is inserted into the tree.

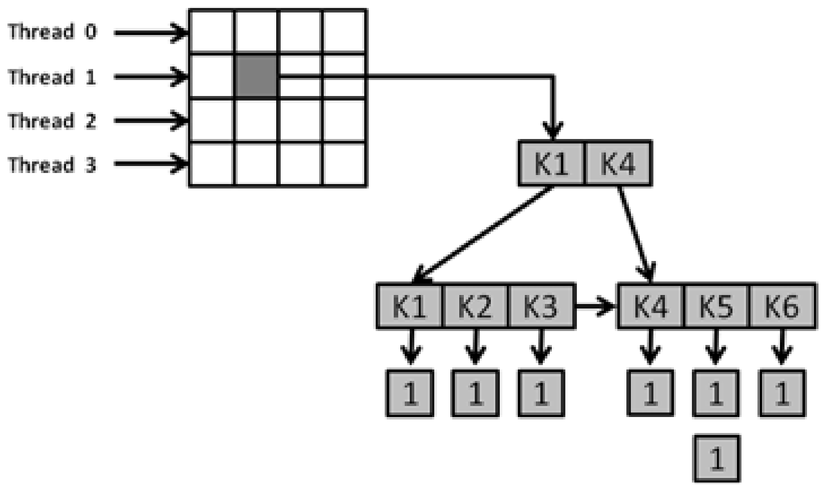

Figure 2.10: Metis intermediate data structure.

Some types of workload, exhibit a homogeneous key distribution in the Map output. This predictability helps obtaining an approximate sizing of the intermediate data structure, providing an equal distribution of keys across the hash table. Metis uses a prediction phase to define the amount of entries in the hash table. The idea is to predict the total number of distinct keys that a Map task can generate. Metis makes this prediction counting the total number of distinct keys issued over the first 10% of the input file, and extrapolating these results to the total execution.

Then, Metis allocates a hash table that is big enough to contain a certain average number of keys per hash entry. In this way, search and insert complexity remains bounded. By default, Metis tries to keep 10 distinct keys per hash entry.

2.2.4 Comparing Multicore MapReduce Frameworks: Intermediate Data Structures

Generally, the different implementations of MapReduce work in the same way, following an execution timeline divided between Map and Reduce phases. In large collections of computers that form the clusters, the entire operation of communication between different phases is dominated by network performance. However, in multi-core MapReduce frameworks there is no actual communication between different phases as map and reduce tasks share the same intermediate data structure.

The big difference between distinct multi-core MapReduce frameworks is the data structure they implement and the way that they manage it. The management process needs to consider multiple factors in order to obtain acceptable performance on multi-core architectures.

Each Map task must avoid accessing the same positions when inserting keys and values, saving as much locking and cache contention costs as possible. An intermediate data structure must be able to maintain the keys sorted, allowing quick search and insertion, and accelerating the combination of equal keys. Different MapReduce applications pressure in different ways intermediate data structure. In this context the Map phase represents most of the insertion work on the intermediate data structure for processing the input data, producing and organizing keys. Also, Map tasks usually emit repeated keys in their own output, which makes it relatively important to the intermediate data structure the ability to perform a fast search for already existing keys.

The Reduce phase must consider all instances of a given key at the same time, grouping the output of different Map tasks. The Reduce tasks should generate an output organized with the objective of minimizing the cost of the Merge phase [30].

Comparing the current main implementations of MapReduce for multi-core architecture, Phoenix++ and Metis exhibit significant differences in the intermediate data structure used. Phoenix++ promotes the use of three adaptations of intermediate data structure on the same framework. The user becomes responsible for deciding what type of intermediate data structure should be used. This procedure makes it necessary to know about the type of workload of the application.

The use of a hash table is attractive for workloads with many repeated keys. Phoenix++ chooses to use a resizable hash table to store map outputs. The use of this strategy requires Phoenix++ to make an extra copy between map and reduce phases, to retrieve the indices that allow reduce tasks process equal keys. Variable-width hash table allows the framework to avoid waste of main memory, since usually different Map workers may emit different sets of keys, ranging from many repeated keys to many different keys.

Metis uses in a different data structure, freeing the user from the responsibility to know the workload characteristics. Metis uses a hash table as well as Phoenix++. After the end of the prediction phase, Metis defines the hash table size that is shared by all workers. Instead of arrays, Metis uses B+ trees for each hash entry. The objective is to take advantage from the sorting of keys that the B+ tree provides, in addition to O(logN) complexity for insertion and search. Although the use of B+ trees is more attractive for workloads where there are many repeated keys, they can also store a certain number of distinct keys without sacrificing performance. The sorting of the keys in the B+ tree reduces the Merge phase execution time to make the final global sorting.

In summary, the Phoenix++ resize strategy adds some extra work to the framework, thereby increasing execution time. Metis gets better performance by using a more general intermediate data structure. However, the use of a prediction phase may end up compromising the use of resources, by generating a waste of memory.

2.2.5 Influence of Key Distribution

Understanding factors that affect positively or negatively MapReduce frameworks, provides an opportunity to optimize current strategies. In current multi-core MapReduce frameworks, a factor that affects performance is the key distribution.

Shared memory MapReduce performance is highly affected by the number of intermediate output *key/value* pairs relative to the number of distinct keys [43]. As can be seen in Figure 2.11 item 1 in accordance with a distribution key *X*, for each key in the key list, it was kept in memory a list of values. In this case the shared memory is affected by the number of keys, independently of the repetition of the keys. Using the combiner, which makes the reduction of keys per thread, repeated keys no longer cause major impact on the use of shared memory, as shown in Figure 2.11 item 2. However, in a key distribution which is predominately high the number of unique keys, shared memory performance is directly affected. Such a situation is caused because the combiner becomes useless, making the key list grow thus increasing the size of the intermediate data structure, as shown in

Figure 2.11 item 3.

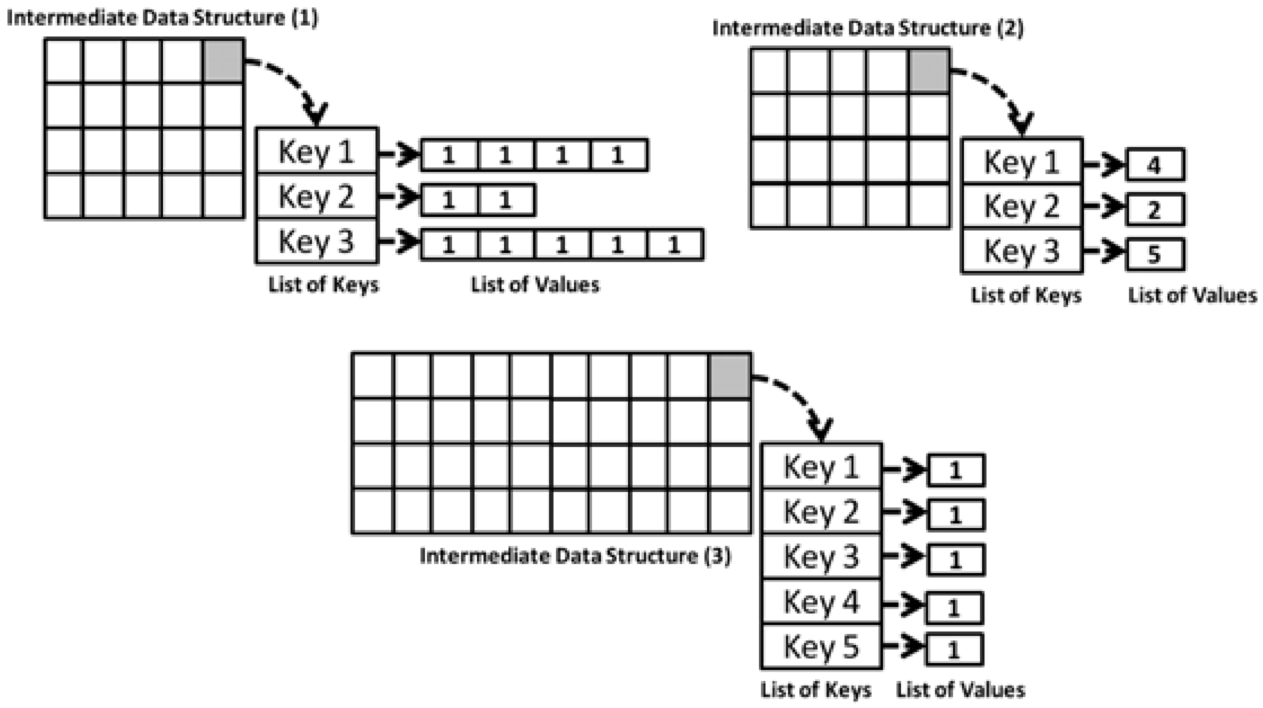

Figure 2.11: MapReduce Framework Key Distribution.

On the other hand, the number of keys per hash entry is also an important factor in the performance of multi-core MapReduce frameworks. By the way that frameworks works, sharing main memory between different *Map* and *Reduce* tasks, all space allocated in main memory is important. When an area of main memory allocated does not have the amount of keys needed to cover the allocation cost, the framework is penalized in unnecessary memory usage, reduced spatial locality, and increasing the work of all the MapReduce framework phases.

Using as an example Metis[30], which by default tries to keep 10 keys trees in each hash entry, although the proximity of this value depends on the distribution through the hash table. The amount of keys per hash entry is an important parameter for Metis in the decision of the hash table size. If Metis[30] makes a prediction that a given input file contains 10 million of distinct keys, the number of columns required would be 1 million. In short, for a prediction of 10 million distinct keys in a environment with 2 threads running, it would create a table of 2 rows per 1 million columns. For the same input file with 10 million distinct keys, but in an environment with 24 threads it would be necessary a table of 24 rows per 1 million columns. This situation creates a waste of memory space, since that the creation of a large table to ensure a memory space for each thread, allow the increase of empty hash entries. The increase in the number of empty hash entries also reflected in unnecessary work to the *Reduce* tasks, that must go through all the column reducing keys.

Chapter 3

Proposal of Virtual Memory Management

3.1 Challenges of Managing Large Data Sets in Multicore MapReduce Frameworks and Work Proposals

MapReduce is a paradigm that was designed to facilitate data parallel computing. Initially developed for cluster environments, MapReduce applications can also be executed in multi-core architectures by the use of specialized frameworks. One of the main proposals of MapReduce implementations is the storage of the intermediate data in a specific data structure

In cluster environments, the Map and Reduce tasks are usually executed in different machines and data communications between tasks are sent through the network. But compared to multi-core shared memory environments, the major performance bottlenecks processing for large MapReduce applications is the large memory requirements of shared data structures and its management.

We have already described how the current implementation strategy is to keep all input and intermediate data into memory during the entire execution. This makes necessary the use of a large amount of main memory of the system [7]. Large data sets usually have difficult to predict key-value distributions that generate very large intermediate data structures. In environments with limited amount of main memory, the whole performance of the application is affected by the unmanaged use of virtual memory and disk.

The intermediate data structures used by multi-core MapReduce frameworks are hash tables. Each of their entries points to a second data structure, where keys and values are actually stored. The size of the hash table is normally set at the beginning of execution, while key-value buffers must grow to allocate map key and value outputs. There is an ideal hash table where each entry contains a similar number of keys and values so that search cost is evenly distributed. Structures where some entries are empty and others store many values should be avoided as they waste memory space.

The main reason to use a MapReduce framework in a multi-core environment is the possibility of executing concurrent map and reduce tasks using the available CPU cores. This promise of parallelism is compromised by the limitations of the data structure needed to store all intermediate data of all concurrent tasks. As each Map task will have its own hash table to insert its keys values without any task dependence the space needed grows linearly with the number of tasks.

For example, if a computing environment has just one thread per map task, the MapReduce framework would allocate space for a hash table with only one row, with a fixed number of N columns. If, for example, each map task generates a key-value set of 200 megabytes in main memory, now let's suppose that the same execution be done in an environment with 32 threads for 32 map tasks. The framework must distribute 32 main memory spaces, one for each thread in use. That means a hash table with 32 rows for the same fixed amount of N columns. In summary, memory needed makes a total of 6.4 gigabytes of main memory just to allocate space for the same workload used in the execution with just one thread as shown in Figure 3.1.

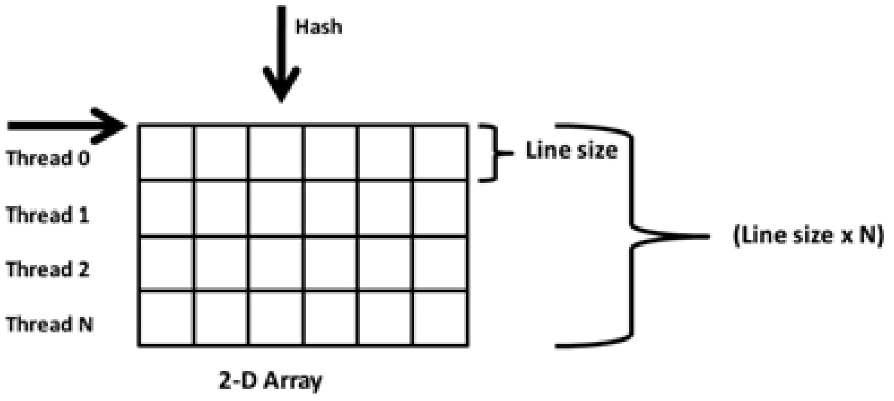

Figure 3.1: Growth of intermediate data structure depending on the parallelism.

In conclusion, current designs of MapReduce frameworks data structures do not scale well when dealing with large data set applications using a large number of threads.

When a Map task, in its key-value insertion process, runs out of main memory the framework relies on the operating system to create a paging file in the hard disk, starting the use of virtual memory. Framework disk usage through virtual memory is not different from main memory. That is, Map tasks request scattered memory references to different hash entries actually mapped in the disk. As the disk latency can be a hundred times slower than main memory, the performance is dramatically decreased.

3.2 Memory Management Proposals

In this section we present the main objectives of our proposal to improve the performance multi-core MapReduce frameworks. During the execution time, the frameworks control the flow of the application. Then, they are responsible of managing the use of resources. Any improvement in the performance of the frameworks affects directly the whole application execution times.

Initially multi-core MapReduce frameworks have been developed for applications which data fits in main memory[23]. When data sets grow bigger than the available memory available, frameworks should consider strategies to manage the extra space needed. Dealing with large data sets requires framework environments to be prepared to manage their needs. Multi-core environments deal with a limited static amount of resources and more demanding memory needs are not easily solved by adding more nodes to the system.

In order to solve the problems presented in multi-core MapReduce frameworks, we designed and implemented a solution that manages, handles and stores keys and values in a more efficient intermediate data structure.

Our proposal objective is to improve the intermediate data structure memory management to be able to accept an indeterminate large number of key-values. We propose to use a mechanism to monitor the memory needs of the MapReduce application. We present a spilling subsystem to save the memory data structure to disk without affecting the performance of the application. Finally, we also add a mechanism to manage the concurrent access to disk by several concurrent threads.

For this work presented, we are using Metis as our platform of analysis. We have designed and implemented some add-on modules to control certain routines of the original Metis, adding some changes to the original code. As seen in Figure 3.2, we have added a monitoring layer between the framework and the operating system, in order to measure the resources used by the framework as a basis for taking decisions in higher-level modules added.

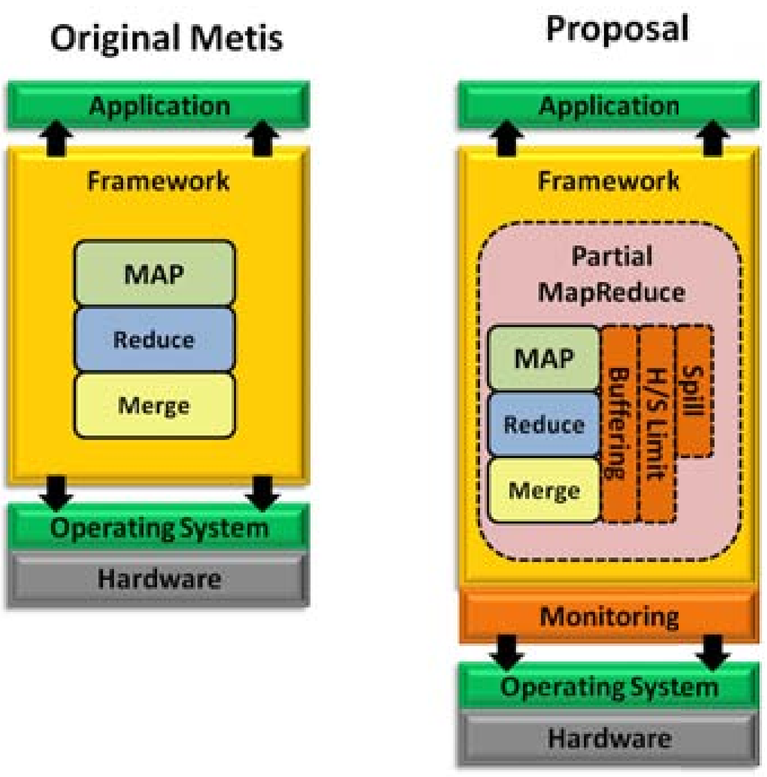

Figure 3.2: Proposed scheme to improve memory hierarchy usage.

3.3 Dynamic Monitoring of Resource Utilization

Our objective is to use a memory management system inside the multi-core frameworks to hide the latency of the secondary memory accesses. First, the framework needs to be aware of how resources are being used. To enhance the control of the framework over the amount of resources used, we present a set of functions that allow the monitorization of the main memory in use.

To collect the necessary information, we use the sysconf function, which allows us to have access to several counters with less overhead than using the common sysinfo library. The monitoring package collects two main system parameters. The total amount of memory in the environment is collected during the start of the monitoring package. The amount of available memory is dynamically monitorized during the application execution. To know the amount of total memory available in the environment represented by Tmem in equation 3.1, we calculate Tpages that represents the total number of physical pages of the system by the page size, represented by Psize.

$$T_{mem} = T_{pages} \times P_{size} \quad (3.1)$$

To calculate the amount of memory available in the environment, represented by Fmem, we calculate the amount of available pages represented by Fpages by the page size represented by Psize, as can be seen in equation 3.2.

$$F_{mem} = F_{pages} \times P_{size} \quad (3.2)$$

As seen in the algorithm 1 at the beginning of the execution we start the monitoring package, together with the MapReduce scheduler, to collect the total memory present in the execution environment. To amortize the overhead caused by monitoring the memory usage, each measure of the amount of memory available is only made at the beginning of a new task.

Algorithm 1 Pseudo-code of the monitoring package.

```
 1: init: MapReduce Scheduler
 2: init: Monitoring Package
 3: Monitoring Package: Check Total Memory
 4: while not finished loading input data do
 5:    while not finished processing the input block do
 6:       scheduler: call phase MAP
 7:       while not finished MAP phase do
 8:          Monitoring Package: Check Free Memory
 9:          scheduler: call task MAP
10:       end while
11:    end while
12: end while
13: while not finished REDUCE phase do
14:    Monitoring Package: Check Free Memory
15:    scheduler: call phase REDUCE
16: end while
17: scheduler: call phase MERGE
18: end: MapReduce Scheduler
```

3.4 Managing the Use of Memory of the Intermediate Data Structure

In multi-core MapReduce frameworks, Map phase is responsible for the production and insertion of keys in the intermediate data structure. After having added the monitoring package as can be seen in Figure 3.3, we have also designed and developed a spill and buffering mechanism. The objective of this subsystem is to move a specific set of keys and values of the intermediate data structure from memory to the hard disk in an organised way. This system will avoid the complete exhaustion of main memory space during execution and uncontrolled paging.

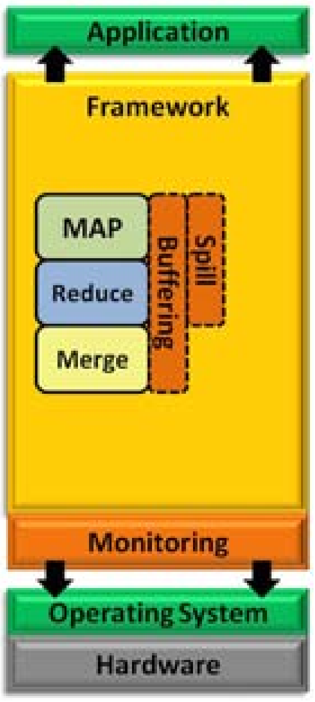

Figure 3.3: MapReduce Framework Buffering and spill scheme.

In the Map phase, our mechanism starts by identifying the actual amount of main memory in the system to adjust memory usage to a specific limitation described as a parameter in the framework user settings. The focus of this strategy is to allow the framework to decide when keys and values stored in the intermediate data structure should be spilled to the hard disk. Also, it also provides a specific amount of data to be spilled.

When a key is emitted by the Map task as can be seen in Figure 3.4 items 1 and 2, the map thread searches the key for an entry in the hash table. When the hash entry is found the thread traverses down the tree contained in the entry, until a position is found for the key to be inserted. The memory consumed by the Map Task is variable, depending on the amount and distribution of unique keys stored in the hash table. Knowing that the B+tree is an order 3 tree, storing up to 7 keys per node. Each time that a new key is inserted into a new node, the system allocates space for seven keys. At the end of the Map phase, the same thread that produced the keys checks if the memory has reached the predefined memory occupation threshold. If the limit has been reached, the same thread copies the keys stored in the last column of its own line on the hash table to an auxiliary buffer, as can be seen in Figure 3.3 item 3. The key-value spill process must traverse every leaf of the B+ tree, retrieving all keys and values contained. The goal is to serialize the data structure before copying it in the hard disk. Serialization is the process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage [44].

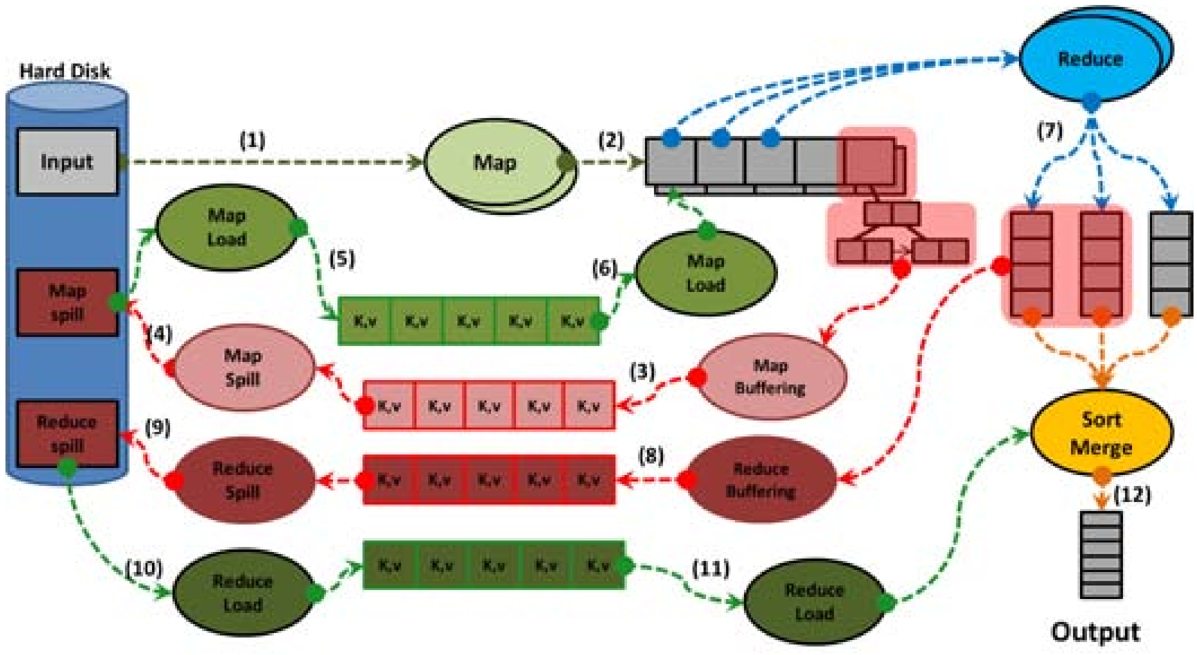

Figure 3.4: Buffering and spill scheme.

To manage the keys and values, we create a spill buffer, which has the same number of rows as the original hash table, i.e., the amount of threads that are running, and simply perform sequential storage. Since this is only an intermediate buffer, with the purpose of inserting sequentially, each line consists of an array that stores only key and value. When the copying of keys is completed, if memory usage is not reduced sufficiently, the same thread continues copying keys from the last column, in direction to first column. The thread resumes the execution of to the next map task only if the memory is reduced enough or column zero is reached. The buffer where the keys and values are copied spills the keys on the hard disk when the task is completed or the buffer is full, as seen in figure 3.4, item 4. Each thread running generates a file on the hard disk, where it stores its keys and values.

In Reduce phase the needed task is to quickly retrieve keys and values that are on the hard disk, to be merged with keys that are in main memory. In Reduce phase, the framework must ensure that the predefined limit of the main memory in use will not be exceeded, moving keys and values between main memory and hard disk when necessary. To avoid the risk of overcoming the memory usage limits, the keys and values must be brought from the hard disk on demand. Before the Reduce phase starts, a certain amount of key-value pairs is moved from the hard disk to fill the first group of columns which will be reduced, as shown in Figure 3.4, items 5 and 6.

When the Reduce workers process the first group of columns, the management system checks if the memory threshold is reached. Then, the workers copy the keys of the first group of columns already reduced to spill buffer, as shown in Figure 3.4, item 8. When the spill buffer is full, the keys are spilled to the hard disk, as shown in item 9, and freeing up memory space. After all columns are processed by Reduce tasks, and before starting the Merge phase, the thread zero retrieves all key-value sets stored on hard disk back to memory. Then, they can be sorted and joined by merge workers, as is shown in items 10 and 11 of Figure 3.4. To start the Merge phase, all the keys that were on the hard disk need to be brought back to memory.

The multi-core MapReduce frameworks do not store actual keys, but store pointers to the input data. In this way, any specific type of keys can be used without modifying map and reduce processing. To perform the strategy of buffering and spill, the framework must lose some of this generality, since it is necessary to make the buffering and spill of the keys and stop manipulating pointers. By bringing the keys back to memory, the buffer created to temporary store the keys is not freed until the end of the execution. Now, the pointers no longer point to the keys of the input and begin to point to the keys stored in the buffer created by the spill mechanism

Performance optimization is directly linked to the occupation of the hash table entries, i.e., the number of keys stored in each entry when the workers begin to move keys to the hard disk. If workers begin to move keys to the hard disk too early, they may find a few keys stored in the hash entries, increasing the overhead time of the workers, and making little reduction of main memory usage. The decision to start taking the keys to the hard disk should directly of the actual main memory consumption, and the memory limit set.

If other applications are sharing the main memory with the framework, it still should be able to move keys to the hard disk in the execution of the Map tasks without compromising the memory utilization limit set. This scenario is not favourable to our mechanism, since the many hash entries may still be empty. When the amount of moved keys to the hard disk is small, it creates an overhead for each spill operation.

Spill operations made over the hard disk done by the last Map tasks of the execution have less overhead since there is a higher number of already spilled keys. The number of keys to be moved is limited by the spill buffer size. This should be also a parameter that the users should adapt to the system resources.

3.5 Synchronizing Multiple Threads Spilling to Hard Disk

Multi-core MapReduce frameworks are designed to achieve parallelism running Map and Reduce tasks over different threads. In our first proposal of a memory management system for intermediate data structure, we tryied to avoid that the excessively hard disk usage would harm the overall execution time.

We suffered a performance problem when the number of threads increased. By analyzing deeper this performance loss, we found a significant competition between threads to spill keys to the hard disk during the Map phase. The problem was that the workstation owned only one local hard disk and N threads were trying to spill keys to the hard disk at the same time.

To spill keys on the hard disk, a particular thread should stop producing or processing keys for the framework. Taking into account that a certain set of keys needs 200MB of main memory, in a common hard disk a thread would take 2 seconds to spill all the keys. On the other hand if there are 24 threads trying to spill keys in the hard disk, and each thread has a set of 200MB keys, that means that the thread number 23 would have to wait about 46 seconds to be able to spill the keys. During these 46 seconds of waiting, the thread number 23 does not emit or process any keys for the framework. Such a problem becomes even worse when the spill process needs to be done more than once. Initially, our implementation used a global threshold value of memory usage to start the spill process that was shared to all threads. We call that value, the hard limit as it represents the limit amount of memory to use.

As a solution to avoid competition between different threads to spill keys to hard disk, we first defined a new threshold limit, called soft limit. The goal is to treat each thread differently, so that disk access is done at different times.

The value of the hard limit is set by the user, and is calculated in percent of the total amount of main memory. To calculate the base of the soft limit, the running thread divides the hard limit by the total number of threads. To find the corresponding soft limit, the running thread uses as a reference equation 3.3 , where N_t corresponds to the number of the running thread, and S_{limit} corresponds to the base of the soft limit previously calculated.

$$B_{SL} = (N_t \times S_{limit}) + S_{limit} \tag{3.3}$$

Using as an example an environment with 6 threads, where the total amount of available main memory is 24 gigabytes, and using a hard limit of 50% would result in the following soft limits shown in table 3.1.

Table 3.1: Memory Limit.

Thread	Soft Limit	Hard Limit
0	2GB	12GB
1	4GB	12GB
2	6GB	12GB
3	8GB	12GB
4	10GB	12GB
5	12GB	12GB

Chapter 4

Proposal of Memory Hierarchy Management

4.1 Optimizing Cache Utilization on Shared-Memory MapReduce Frameworks

It is well known that the memory access pattern of an application can have a very important effect on performance. In the worst case, the performance degradation can be up to two orders of magnitude (around 100x). This worst case occurs for memory patterns that exhibit no temporal or spatial locality, and when dependent data loads must be issued sequentially to the memory system. Lack of locality occurs when the application accesses its data from a large memory area and with an unpredictable scattered pattern.

Multithreaded applications running on top of multiprocessor, multi-core systems provide additional parallelism at the thread-level: each thread issues independent memory requests. This parallelism helps exploiting the available system memory bandwidth. H/W multithreading is a latency-hiding strategy that can alleviate the effect of long-latency cache misses that must be solved in main memory. However, several concurrent threads competing for the scarce cache memory capacity may exacerbate contention and increase cache miss rates.

MapReduce applications typically generate lots of data accesses, and the proportion between data accesses and computation often grows with the size of the data input. Most of these data accesses are performed by the system runtime, while the user-defined map and reduce functions mainly compute using local data provided to the functions by the runtime. The performance of applications with large data sets that require relatively low computation operations is very sensitive to the management of the input data, and especially to the management of the intermediate key/value pairs.

Next we will analyse how current shared-memory MapReduce implementations handle the memory hierarchy for applications with relatively large data sets. We will identify performance problems originated on the design of the data structures used to store the intermediate *key/values* generated by the Map phase. There are also problems when scaling the execution to systems with a large number of cores (multiple sockets and several cores per socket).

Based on the analysis, we will present a proposal, which we call partial MapReduce execution, aimed to improve the performance of the system s memory hierarchy, and then decrease overall execution time. The main idea behind the proposal is to distribute the *key/value* pairs generated by the map tasks in a two-level data structure, where each level is designed to meet different performance requirements.

4.1.1 Memory Access Behaviour on Current MapReduce Frameworks

MapReduce applications concentrate their memory accesses on reading input data, and especially on reading and writing the intermediate *key/value* pairs generated by map tasks, and consumed by reduce tasks. We will assume applications with relatively large data sets, and applications that provide a user-defined combiner function and, therefore, assume that reduction operations are associative. Associativity implies that the system can apply the combiner function to the list of values associated to each key in any order, either with values generated by the same map task or values from different map tasks, without affecting the final result.

Reading the input data efficiently is a relatively straight forward problem. Since data must be read just once and input data blocks can be processed concurrently, the most effective strategy is to process large contiguous blocks of input data, so that spatial locality is fully exploited. Shared-memory MapReduce frameworks handle this issue by partitioning the input data into relatively large blocks and assigning a contiguous block to each active map task (executed concurrently by a thread worker). For applications that generate few intermediate keys and are mostly bounded by the input read time, the size of those blocks can have a performance impact. In this case, the optimal size of the block will depend on the total number of threads competing for the input data, and the organization of the memory hierarchy (NUMA versus UMA organization, size of Last Level Cache (LLC), and so on).

For MapReduce applications that generate a relatively large number of distinct intermediate keys, the most important performance issue on shared-memory MapReduce frameworks is the management of intermediate key/value pairs,both for searching and inserting emitted keys. The organization of the intermediate data structure must accomplish two goals, often conflicting: (1) minimize the number of operations and data accesses for the search and insert operations, and (2) maximize data access locality to improve memory performance. We will next explain how existing frameworks (Phoenix++ and Metis) handle this question. In brief, they prioritize the first goal and assume a relatively small number of distinct keys that assures a good memory performance.

4.1.2 Memory Access Behaviour for Intermediate Data Structures

Both Phoenix++ and Metis use dynamically sized hash tables, whose size grows with the number of distinct intermediate keys. In both systems, the hash table stores a pointer to another data structure, devoted to store the actual keys and values. Therefore, a single algorithmic step provides a pointer to a reduced list of keys where the search can continue. However, as the size of the hash table increases, the locality of accessing the hash table decreases: accesses to the hash buckets tend to be scattered and the bigger hash table does not fit in some of the cache levels. Also, the use of pointers requires two strictly sequential accesses: one to the pointer and one to the pointed data, then reducing memory-level parallelism.

Phoenix++ and Metis use different data structures to store the actual keys and values. Phoenix++ stores keys and values in contiguous arrays of memory, which enhances spatial locality for sequential search. However, when the arrays of keys grow bigger and a binary search algorithm is used, then locality suffers. Resizing the array to accommodate a bigger set of keys, although costly in terms of number of operations, exhibits good spatial locality.

Metis B+trees are more efficient for insert operations when the number of keys in the hash bucket is relatively large. However, both search and insert operations exhibit the low data access locality and the low memory-level parallelism typical of a pointer chasing access pattern.

There are two main drawbacks shared by both strategies: (1) the usage of pointers to keys, instead of the actual key contains, and (2) the wasted storage space when keys do not fully populate the data structures. By large, the first drawback is the most important one.

Storing pointers to keys instead of keys simplifies the search (and insert) algorithm and memory management. This strategy releases algorithms to depend on the actual key data type, and memory storage size is fixed. This is especially helpful when keys are variable-size strings. However, reading the stored key to be compared with the emitted key requires two chained memory accesses and, what is worse, accessing the memory address space devoted to the input data. The first occurrence of each key in the input data may be referenced several times by means of the pointer contained in the data structure holding intermediate key/value pairs. Those occurrences are scattered along the input data area and accesses to them have very low spatial locality.

Having a large hash table reduces the average number of keys per bucket, and then the total number of instructions needed for searching a key. The problem is that many buckets may be empty, reducing the efficiency of memory usage. Metis and Phoenix++ allocate relatively large blocks to accommodate keys and values. This improves spatial locality when accessing consecutive keys, but the larger the granularity of the block, the more space will be unused on average.

4.1.3 Memory Access Behaviour When Scaling to a Higher Number of Threads

Both Metis and Phoenix++ allocate independent data storage for each concurrent thread executing a map task. This avoids including synchronization locks, which will hamper performance scalability as more threads are added in the system. The cost of this simple solution is to duplicate many data structures, sometimes increasing memory requirements almost linearly with the number of threads, and putting more pressure on the processor caches.

Applications that generate a large number of keys with many repetitions tend to store the same key in each of the data structure corresponding to several of the running threads. The pointers representing the same key point to different positions in the input data, since keys are inserted by each thread from its corresponding input blocks. This behaviour exacerbates the problem of access locality: in the worst case, all the virtual pages containing the input data are referenced from the intermediate data structures. This can easily generate an extremely high amount of TLB misses, apart from multiple data cache misses.

4.1.4 Memory Access Behaviour on the Merge Stage

Metis prioritizes the performance of the Map and Reduce stages, and allocates a very large hash table to maintain a small average number of different keys per bucket. This generates a very large number of small sorted lists, which must be merged in the final stage of the framework in order to produce the final sorted output. We have measured that applications with large input footprints expend a lot of their execution time in the final merge phase. Merging a lot of small lists is not optimized for exploiting cache locality.

In summary, Metis and Phoenix++ are designed considering a relatively small number of distinct keys, and prioritize the reduction of operations in the Map stage. For large workloads, and for an increasing number of concurrent threads, this strategy achieves bad memory performance.

4.2 Proposal: Partial MapReduce Execution

We want to attack the performance problems suffered on current share-memory MapReduce frameworks with applications that generate a large number of intermediate key/value pairs. The main idea is to provide a two-level storage organization for *key/value* pairs. The primary data structure is small and static, sized to fit into the last-level cache (LLC) of the hierarchy, and it is optimized for concurrent searches and insertions of the key/value pairs emitted from map tasks, and for concurrent reduction of these data. The secondary data structure is dynamically allocated, and is optimized for a final reduce and merge sort phase with large data sets.

For this strategy to work, we need to group the execution of Map and Reduce phases such that the reduce phase begins as soon as the primary data structure is filled. We call this mechanism partial MapReduce, since the Reduce phase is partially applied, before all the input has been processed. Reduced *key/value* pairs are stored sequentially and sorted by key in the secondary data structure, which is basically a collection of sorted vectors of key/value pairs. The partially reduced data must be reduced at the end, when all the input has been processed, in a final stage that is combined with a final sorting operation.

Our proposal relies on the associativity and commutability of the reduction operation. This is a property that accomplish a very large number of MapReduce applications, and implies that the reduction operation can be applied in any order to the values associated to a given key. This property is also needed to use combiner functions.

Figure 4.1 shows a scheme of the intermediate data structure used by Metis. When an application generates a large number of distinct keys, the prediction phase of Metis resizes the original hash table to a very large table, which will be poorly used. The B+tree building blocks are dynamically allocated, and expand along the processor memory. B+tree nodes must occupy a full cache line, or otherwise false cache sharing can occur. Metis estimates the size of the hash table so that, on average, around 10 keys are stored per bucket. Both facts contribute to have a low occupancy of the B+tree.

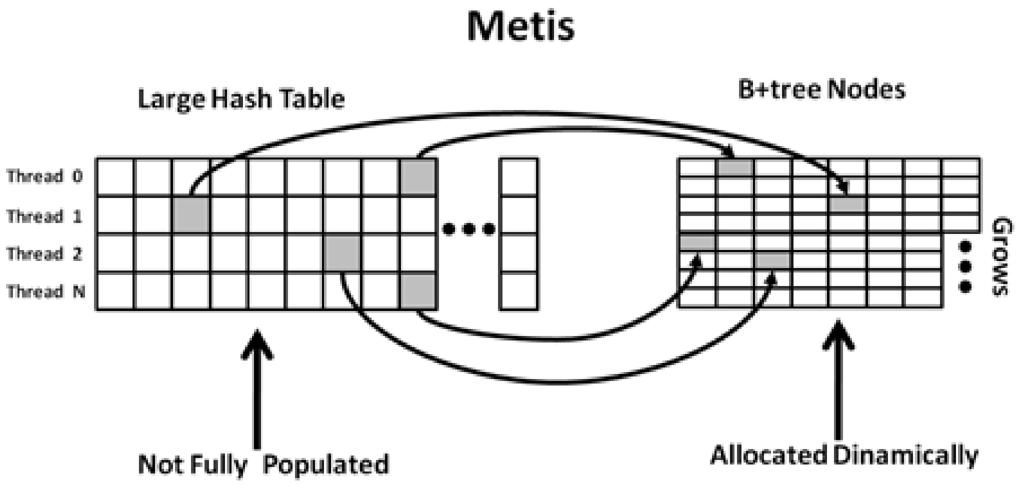

Figure 4.1: Intermediate data structures used by Metis.

Figure 4.2 shows our proposed two-level intermediate data structure. The hash matrix and the B+tree nodes are statically allocated. The blocks containing partial reduced key/value pairs are dynamically allocated. The first-level structure contains the active emitted keys, those generated in the current partial MapReduce phase. Searches and insertions are performed using this structure. The second-level structure contains vectors of sorted, partially-reduced *key/value* pairs. These data can expand along memory.

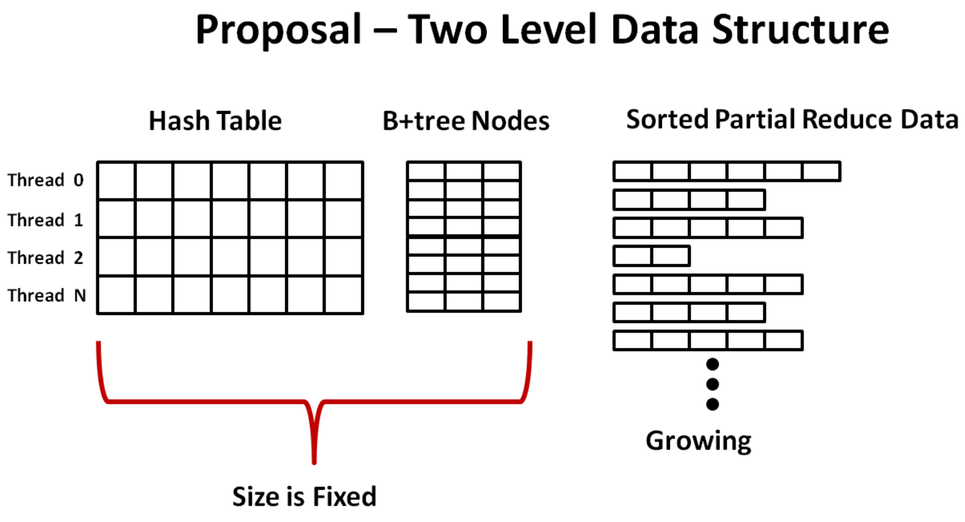

Figure 4.2: Two-level hierarchy for storing intermediate key/value pairs.

The proposal involves the following issues:

1. The size of the primary-data structure must be calculated depending on the size

of the Last Level Cache (LLC) of the processor. The amount of threads sharing the LLC in the same socket must also be considered. Also, it is necessary to consider if the table contains actual keys, or pointers to keys;

2. It is necessary to implement a mechanism for detecting when the primary data structure is full, and then to synchronize all the threads and start the reduction phase;

3. After the partial reduction, a merge operation generates the partial output, where the *key/value* information is serialized (copied in contiguous positions). If pointers to keys are used, then at this point the key information is copied into the final output, so that no more references to the input processed so far are maintained;

4. The partially reduced and sorted data must be stored until the input data is completely processed. At the end, these vectors must be reduced and merged to generate the final output.

The main advantage of our proposal is that the partial Map and Reduce phases are faster, because accesses to the intermediate data are mostly local into the LLC. Also, the final Merge operation is more efficient, because it operates with a small number of long lists that are sequentially stored into memory, instead of a large number of short chained lists that include pointers to the input keys.

The price to pay is the overhead of copying from the primary to the secondary structure. This overhead should be small, because data is copied sequentially into main memory. Also, keys are actually copied and the pointers can be discarded, effectively limiting the memory footprint of the map and reduce phases. Also, the synchronization required between partial Map and Reduce phases may reduce parallelism and hurt performance.

In order to prove that our proposal is effective, we have implemented a design that works on top of Metis. This has the advantage that the proposed mechanism does not include additional overhead when the application generates a small number of distinct keys: a single execution processes all the input without filling the primary data structure, and then the additional mechanisms are not used.

4.3 Proposal: Partial MapReduce Implementation

To support the use of a small intermediate data structure, we reorganized the framework to partial reduction. This strategy allows us to support the fixed size of intermediate data structure, processing and reducing keys quickly. The principle of partial MapReduce includes joining all the Map, Reduce and Merge tasks in a partial execution, and replicate through of the framework execution. Each partial execution allows the reduction of keys that are on the intermediate data structure and make room for new keys, maintaining structure within the limits of the LLC.

The Figure 4.3 shows an execution, that has been divided into two partial MapReduce tasks. In part 0, the framework makes the distribution of the first block of the input file to be processed by the map tasks, as shown in Figure 4.3 items 1 and 2. Each map task processes a block of the input file on the first part shown in Figure 4.3 by item 2. As map tasks process the blocks of the input file, the generated keys are stored in the intermediate data structure shown in Figure 4.3 by item 3.

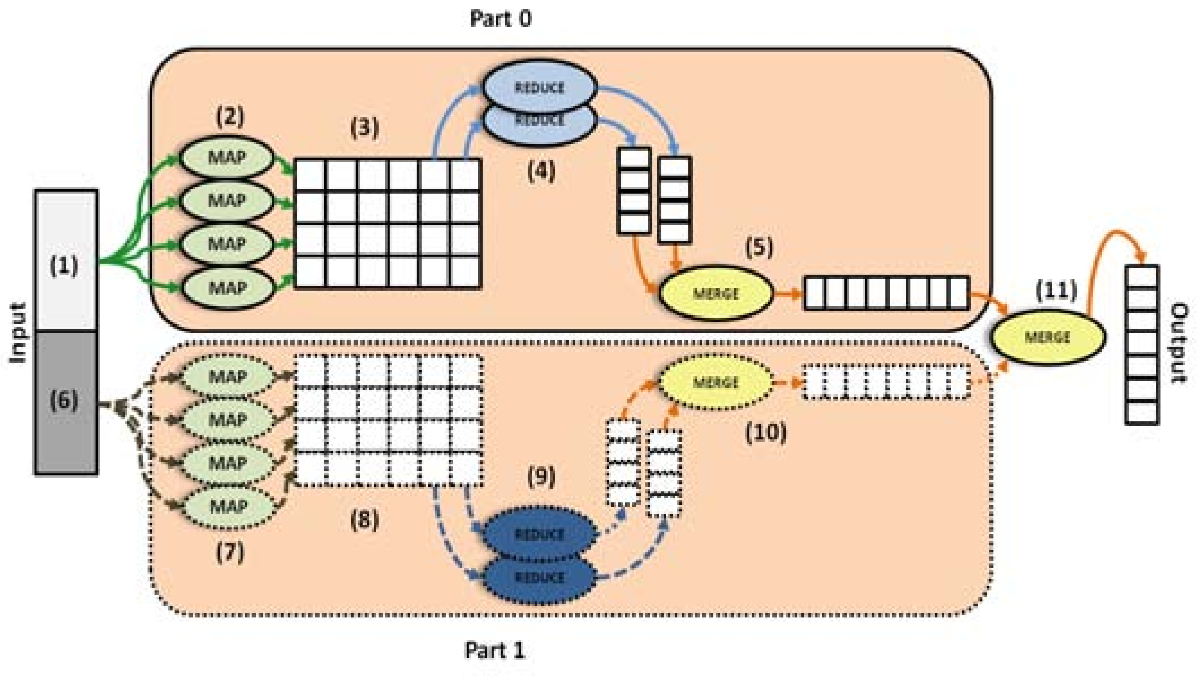

Figure 4.3: Partial MapReduce scheme.

In our implementation, the framework maintains a counter of the number of keys per hash entry. When the counter reaches a preset limit, the framework stops processing the input file through the map tasks and the reduce phase begins. The counter of keys per hash entry aims to prevent the intermediate data structure grow sufficiently as to exceed the size of LLC.

At the beginning of the reduce phase, shown in Figure 4.3 item 4, the framework executes a reduce task per column. The goal is to aggregate all the values associated to equal keys generated by distinct map tasks. Each reduce task stores the values in a new data structure, where the keys are sorted.

When the reduce phase reaches the last column of the hash table, the framework releases the space taken by tasks map to store the keys, freeing up space in memory and starts execution of the merge phase that can be seen in Figure 4.3 item 5. The merge phase aims to merge all outputs generated by the reduce phase and perform a partial final sort. The more sorted the keys are, the faster the execution of the merge phase will be. When the merge phase is completed, all reduced and sorted keys are stored in a final data structure, and the framework releases the data structure used by the reduce phase.

For our implementation, at the end of the merge phase, the framework checks if the pointer to the input file has reached the end of the file, if not, the framework performs another part of the execution.

In execution of the second part, the framework reruns all phases to process the next block of the input file shown in Figure 4.3 item 7. On reaching the end of the merge phase shown in Figure 4.3 by item 10, the keys generated by the merge phase of the second part are stored in a final data structure. With the end of all parts, the framework checks the need to perform any more partial MapReduce. If the pointer to the input file has reached the end, the framework performs an extra merge phase that aims to merge and sort the output of all execution parts, making a possible reduction if there are repeated keys in different parts of the execution. At the end of the execution it generates a final output, as shown in Figure 4.3 item 11.

The idea is to divide the input load in blocks so that all phases can interact on each block. At the end of execution of each MapReduce part, the result will be a number of keys already reduced. In the original Metis, the Map phase had to process all the input load before starting the Reduce phase, i.e., maintain a large intermediate data structure and millions of keys stored in this intermediate data structure. In our proposal, we process a part of the input load using N Map tasks, we reduce the keys stored in the intermediate data structure of this small block using N reduce tasks. In the end we join and sort using Merge phase.

Each block of the data set is processed as shown in Figure 4.4 item 1, until finally there is only one set of key buffers, reduced and sorted, as can be seen in Figure 4.4 item 2. For the framework to emit a final output, it is necessary to make extra an call to the Merge phase, which aims to merge and sort all the output buffers of MapReduce parts, as can be seen in Figure 4.4 item 3.

The development of the overall execution is simple, as can be see in the algorithm 2, which consists of blocks of MapReduce parts that perform calls to the Map, Reduce and Merge phases. To reduce the use of main memory, we establish more procedures for

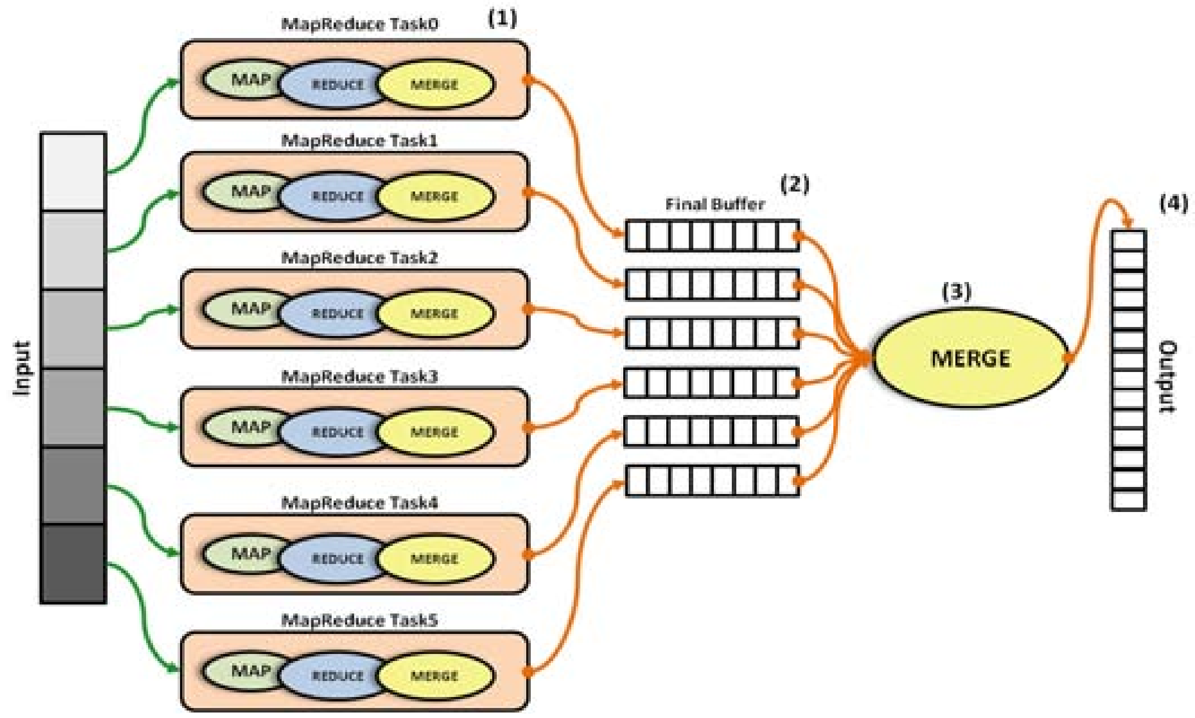

Figure 4.4: Complete Partial MapReduce scheme.

allocation and freeing memory.

Partial MapReduce aims to process small blocks of the input file, generating small key blocks, when the final result will always be a set of reduced keys. As shown in Figure 4.5, different from Metis where the phases are called few times and generate all possible data, our proposal calls phases more often and generate smaller sets of data.

4.3.1 Proposal: Map Phase

In the original Metis, for each existing thread in the environment, the framework launches 16 Map tasks, for an environment with 24 threads, Metis needs to launches by default 384 Map tasks. To organize the blocks of MapReduce parts, our proposal uses the original Metis tasks as a reference. To try to increase the spatial locality of the data, the framework should stop running the Map phase, even if the input file has not finished being processed, and initiate execution of the Reduce phase. The idea is that a part of MapReduce execution never finishs a Map task by half, which could cause loss of keys.

To know the moment at Map phase execution should be interrupted, our proposal uses the LLC size as a reference. When the size of intermediate data structure gets closer to

Algorithm 2 Partial MapReduce overview.

```
Require: Input, StructureSize
 1: init: MapReduce Scheduler
 2: while not finished loading input data do
 3:    scheduler: call task MAPREDUCE
 4:    mapreduce: task N
 5:    malloc: intermediate data structure
 6:    while not finished processing the input block do
 7:       scheduler: call phase MAP
 8:       malloc: final data structure
 9:       while not finished processing the map block N do
10:          scheduler: call task MAP
11:       end while
12:       scheduler: call phase REDUCE
13:       free: intermediate data structure
14:       scheduler: call phase MERGE
15:    end while
16:    end mapreduce: task N
17: end while
18: scheduler: call phase MERGE
19: free: final structure
```

overcoming the LLC size, the framework does not launch new Map tasks and waits for the finalization of Map tasks that are already running, to finally call the Reduce phase.

If the framework interrupts the execution of the Map phase too early, it would increase the number of parts of execution. Each time that the Map phase is interrupted, the finished threads are forced to wait for threads that are still running Map tasks. A number of unnecessary MapReduce parts comes down as a unnecessary thread interruption.

If the framework takes too long to interrupt the execution of Map phase, or if the Map tasks are long, the intermediate data structure will grow sufficiently may surpass the LLC size reference.

Every new part of MapReduce execution, a new intermediate data structure is allocated to support the data generated by the Map tasks, in other words, each part of MapReduce execution has a individually intermediate data structure, as can be seen in Figure 4.6. In the original Metis, a large intermediate data structure was allocated to support all data generated by all Map tasks.

In the original version of Metis, the entire input file was splitted through all the Map tasks needed to be processed. In our proposal, the input file is splitted into parts of the MapReduce execution, after the first split, the block that belonging to each part task is splitted between all Map tasks. Arrange the blocks in small sizes prevents that the Map

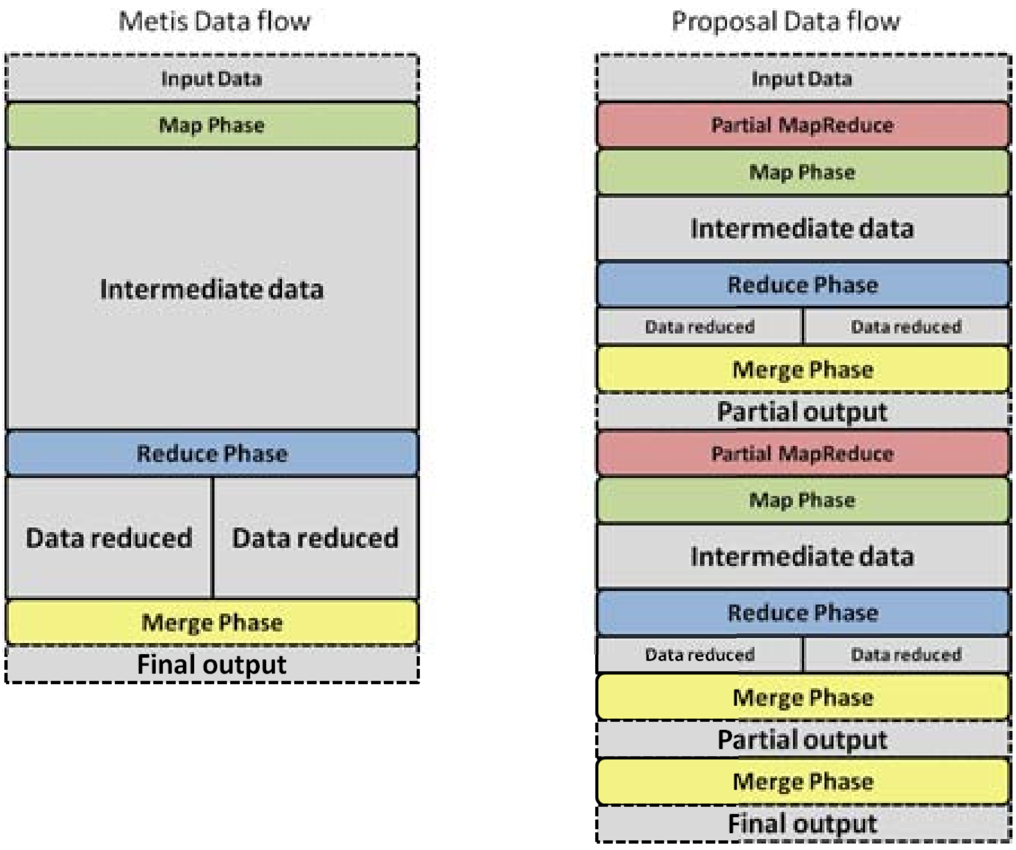

Figure 4.5: Metis Data Flow versus Proposal Data Flow.

tasks, becomes large as to make the intermediate data structure can surpass the LLC size.

4.3.2 Proposal: Reduce Phase

When the Map phase that belongs to a particular part of MapReduce execution is interrupted, the framework makes a call to Reduce phase. The keys reduction, of a particular part of MapReduce execution keeps intermediate data structure size under control.

In the original version of Metis, the intermediate data structure can grow unchecked, and the Reduce phase is started only when the entire input file ends being processed. In our proposal, a small set of keys is generated and reduced directly on the LLC. This strategy allows reduce tasks become even faster. According to keys are reduced by the Reduce tasks, the result is stored in another memory region, as shown in Figure 4.7, allowing the release of the intermediate data structure at the end of the Reduce phase.

In the original Metis, to process 10 million of different keys, we would be needed 1

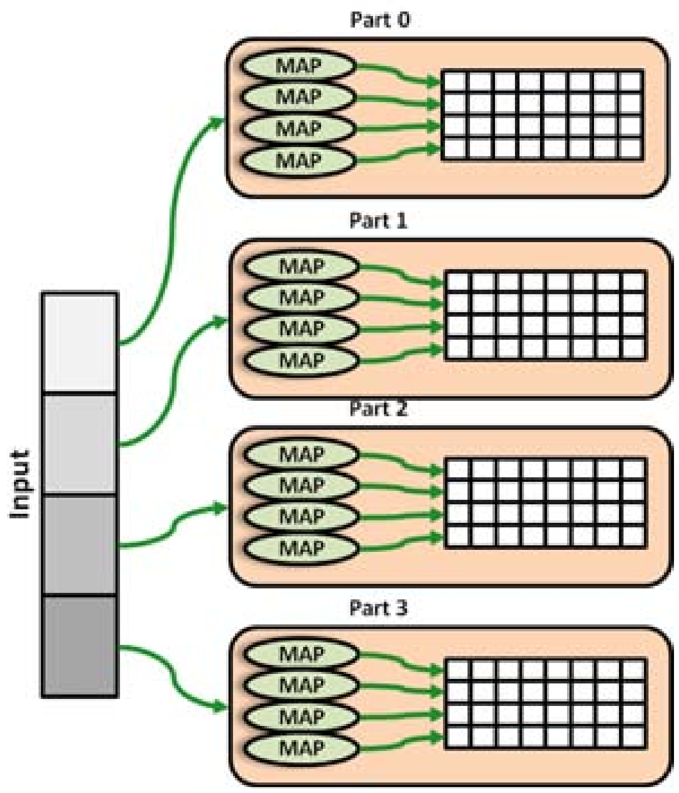

Figure 4.6: Partial MapReduce Map phase scheme.

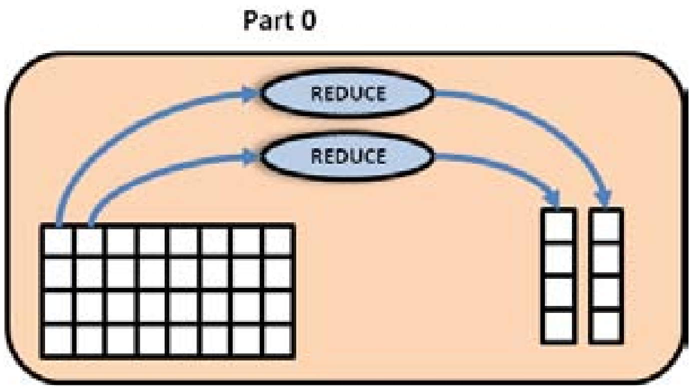

Figure 4.7: Partial MapReduce Reduce phase scheme.

million of Reduces task, which is the number of columns in the hash table. In our proposal, the number of reduce tasks is directly related to the amount of LLC available. As each part of MapReduce execution uses a small intermediate data structure, the number of reduce tasks should take into consideration the amount of MapReduce parts produced by the framework. If the number of columns in the hash table be set to 10K for example, and the framework requires 10 MapReduce parts to process all the input file, it would give us a total of 100K of reduce tasks, only 10% of amount of Reduces tasks of the original Metis.

Different of the Map phase, where the total number of Map tasks is divided over the number of MapReduce parts, in the Reduce phase the process is reversed. To know the total number of Reduce tasks, we use the number of Reduce tasks multiplied by the number of MapReduce parts.

4.3.3 Proposal: Merge Phase

When the Reduce phase is completed, the framework begins the Merge phase related with the MapReduce part in execution. The Merge Phase of each MapReduce part aims to generate only one set of keys reduced and sorted, which is stored in a final buffer, as can be see in Figure 4.8.

Each sorted output generated by each MapReduce part, allows to accelerate the final merge and sort of all the output generated by distinct MapReduce parts. At the end of the execution of all MapReduce parts, an extra Merge phase is called by the framework, which aims to merge and sort the data generated by all MapReduce parts of the framework, as can be see in Figure 4.8.

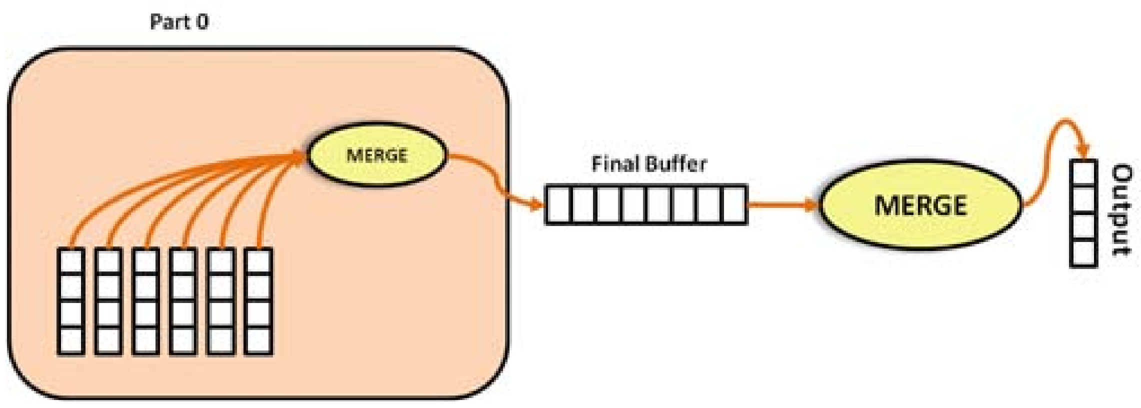

Figure 4.8: Partial MapReduce Merge phase scheme.

During the sorting keys in the Merge phase, if they are found repeated keys coming from distinct MapReduce parts outputs, the Merge phase makes another small reduction in the repeated keys. This procedure becomes simple because the sorted keys provide the nearest equal keys. However depending on the key distribution across the different blocks and the number of MapReduce parts, there may be a significant number of keys to be reduced. This extra reduction may cause an extra processing in the Merge phase.

Chapter 5

Evaluation

In this section we describe the results obtained with the different parts of our proposal. Throughout this chapter, all comparative experiments consider the latest available version of Metis in C, as well as the latest available version of Phoenix++.

5.1 Experimental Environment

Measurements have been taken in three different environments: (I) a multi-core processor with Intel Core Duo 3GHz and 6GB of main memory in 64-bit Linux. (II) A dual-socket Intel Xeon E5645 2.4GHz with 6 cores/12 threads each one, with 96GB of main memory in 64-bit Linux. Each of the 12 cores has a 32 KB L1 data cache, and 256KB L2 cache. The 6 cores on each chip share a 12 MB L3 cache. (III) A four sockets AMD Opteron Processor 6376 2.3GHz with 16 cores each one, with 128GB of main memory in 64-bit Linux. Each of the 64 cores has 16kB L1 data cache, and 2MB L2 cache. The 16 cores on each chip share a 12 MB L3 cache. All executions are done using the maximum number of H/W threads available in the system (I) 2 threads; (II) 24 threads; and (III) 64 threads.

5.2 Benchmark Applications Description

As shown in previous studies[30], different applications push the framework in different ways. On the one hand, certain applications spend most of the execution time in the MapReduce libraries, manipulating the Map tasks output and sorting the Reduce task outputs. On the other hand, there are applications that make little use of the MapReduce library, guided by the application purpose and the type of output produced. Of the applications originally available with the original Metis, we can divide into two main groups, represented by tables 5.1 and 5.2. In the first group represented by table 5.1, the applications that make high use of MapReduce library, are the Word Count and Inverted Index benchmarks. In the second group represented by table 5.2, the applications that make lower use of MapReduce library, are the PCA, kmeans, histogram, Linear Regression, String Match and Matrix Multiply benchmarks.

Table 5.1: Benchmark applications that make higher use of the MapReduce library.

Application	Description	Keys	Pairs per key
Word Count	Counts the number of unique word occurrences	Unique words	N^o words
Inverted Index	Build reverse index for words	Unique words	N^o words

Table 5.2: Benchmark applications that make lower use of MapReduce library.

Application	Description	Keys	Pairs per key
PCA	Principle component analysis on a matrix	Input matrix	1
Kmeans	Classifying 3D points into groups	K	Points/K
Histogram	Computes the RGB histogram of an image	3X256	N^{o}map input splits
Linear Regression	Calculate the best t line for points	5	N^{o}map input splits
String Match	Find an encrypted word from les with keys	4	N^{o}map input splits
Matrix Multiply	Multiply two integer matrices	0	0

In Figure 5.1 it is possible to see the execution time distribution between the application and the MapReduce library. Our proposed improvement is based on modifications made over the original Metis framework, this situation makes the improvements in memory usage and execution time can be best viewed in benchmark applications that use MapReduce library more extensively. Based on the previously mentioned points, we decided to use the benchmark application presented in table 5.1, since these applications make hight use of the MapReduce library, they can demonstrate the problems pointed out in this work. In short, the kind of MapReduce applications used in this work, are applications where any Map task can emit any number of keys with any distribution.

To show the improvement obtained with our proposal, we use the original Word Count benchmark application of Metis[30] and Phoenix++[42], both already included in the original frameworks. The use of an Word Count application is significant for three main reasons: (I) it has a scenario where relatively large input loads are used; (II) it can produce a number of intermediate keys that exceed even the size of the input load; (III) it has a key distribution which can be significantly reduced. Based-cluster Frameworks of previous work[39] showed such behavior for Word Count applications.

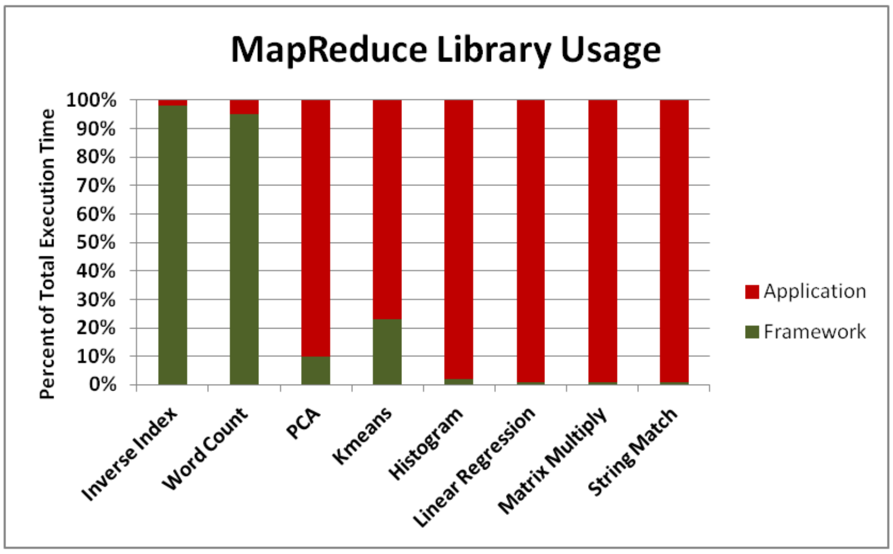

Figure 5.1: MapReduce library usage.

For the Inverted Index case, we use it only for the comparisons against the original Metis[30], since Phoenix++[42] does not provide an Inverted Index benchmark application.

5.2.1 Word Count

The application counts the frequency of occurrence of each word in a given input file. The Map task processes different parts of the input file and returns intermediate data consisting of word(key) and the value 1, which indicates that the word was found. Reduce tasks unify and add values to each word (Key) on the global set of all keys found.

5.2.2 Inverted Index

The application traverses a set of occurrences in a document, extracts all keys, and compiles an index from keys to files. Each Map task analyzes a set of files. For each key that is found, it generates an intermediate pair with the key and the information of the file as the value. The Reduce task combines all the files that reference the same key in a single set.

5.3 Workload

When we analized the type of workload initially used by Metis and Phoenix++ papers, we verified that both were too small to exemplify the problem of memory consumption that we described.

To demonstrate the framework behavior attending its memory consumption issue presented, we chose to make use of synthetic input files, so that we could push the framework to substantiate such problems. For aspects of comparison, the original Metis uses as maximum workload an input file that has 300MB, while Phoenix++ works with an input file maximum of 1GB, while we use the workloads shown in table 5.3.

Table 5.3 is divided into *Input Size*, *Keys* and *Pairs*. *Input Size* represents the size in bytes of each input, *Keys* represents the number of unique keys present in the input and *Pairs* represents the number of words present in the input, i.e., the amount of *key/value* pairs.

Table 5.3: Repeated Keys Workload.

Input Size	Keys	Pairs
3.9GB	10M	200M
7.8GB	20M	400M
11.7GB	30M	600M
15.6GB	40M	800M
19.6GB	50M	1.0G
23.5GB	60M	1.2G
27.4GB	70M	1.4G

5.4 Memory Management Performance

5.4.1 Buffering and Spill Execution Time

Using Intel Core Duo with 6GB of main memory, the goal in this experiment was to verify the benefits obtained at execution time using the Spill/Buffering Mechanism in a limited environment.

Using the inputs described in Table 5.3, it is possible to see in Figure 5.2 the result obtained. For inputs of 10M to 40M keys it is possible to see that there is no difference in Map phase execution time compared to original Metis. Such a situation occurs because the memory usage limit set was not exceeded and the framework does not need to make use of the buffering/spill mechanisms. On the other hand, for the 50M, 60M and 70M inputs, it is possible to see a speedup of 2.2, 2.4 and 3.6 respectively.

When the memory usage limit is exceeded, the mechanism moves keys between main memory and hard disk, avoiding the use of virtual memory. Although such a mechanism use the hard disk, the entire procedure is done in an orderly way, designed to be efficient with the framework, which allows to hide the hard disk natural latency.

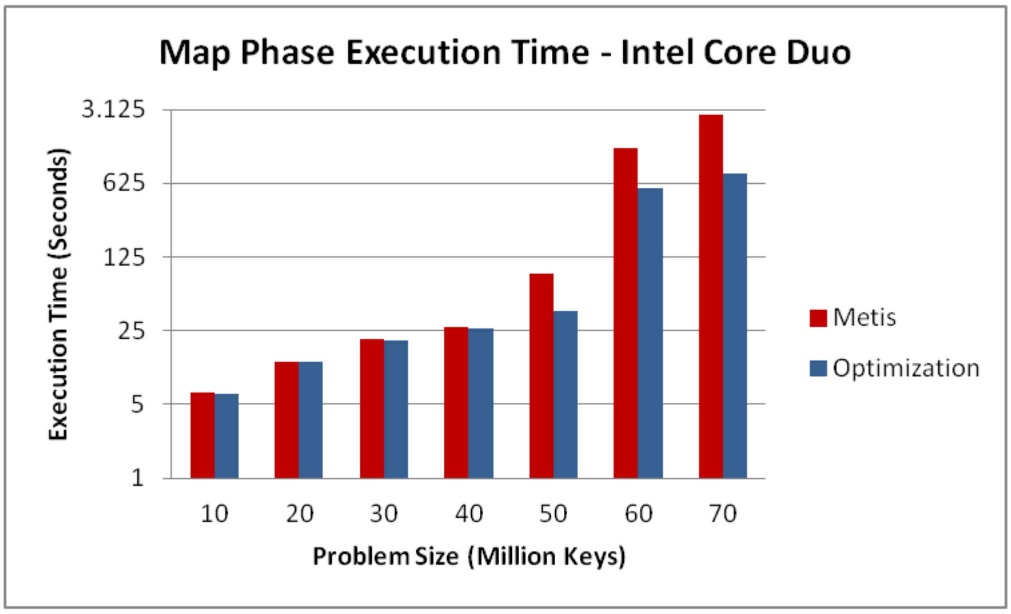

Figure 5.2: Map phase Execution Time using Buffering/Spill mechanism.

As seen in Figure 5.3, in the Reduce phase the 40M input uses the Buffering/Spill mechanism. Such a situation occurs because the memory usage limit was exceeded in the last Map task, forcing the first Reduce task move keys between the hard disk and memory.

In contrast with the Map phase, where only keys and values were moved from main memory to the hard disk, the Reduce phase can make two types of operations, move keys and values to the hard drive or bring keys and values from hard disk to the main memory. While the Map phase only produces keys, the Reduce phase needs to process keys and values, which makes it necessary to retrieve keys from the hard disk to be processed.

For 40M, 50M, 60M, and 70M inputs it is possible to see a speedup of 1.4, 2.1, 2.2 and 2.5 respectively, which shows an improvement as the workload increases.

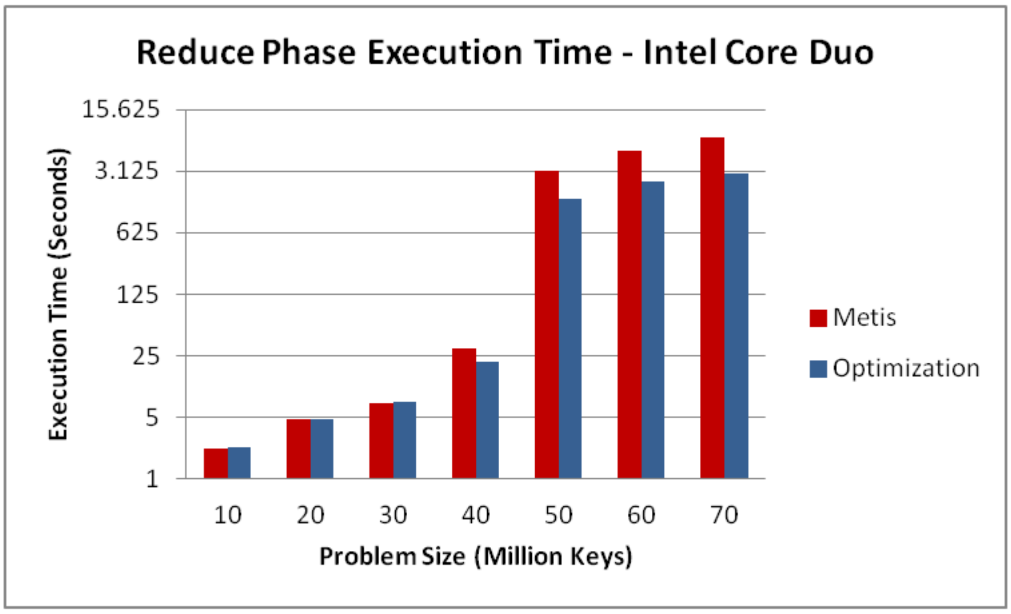

Figure 5.3: Reduce phase Execution Time using Buffering/Spill mechanism.

5.4.2 Buffering and Spill Main Memory Usage

Using the Intel Xeon environment with 96GB of main memory, our goal in this experiment was to verify the reduction in main memory usage making use of our *Buffering/Spill* mechanism.

Figures 5.4 and 5.5 show the reduction in main memory usage obtained in the *Map* and *Reduce* phases for the optimized version. For the input files of 10M, 20M and 30M of keys, there is no change from the original Metis, because for these types of workloads the memory limit is not reached. As the memory consumption does not exceed the limit, there is no change on the way that the MapReduce framework works in each of the implementation compared.

On the other hand, as seen in Figures 5.4 and 5.5, for input files of 40M, 50M, 60M and 70M of keys, memory usage is exceeded and the framework begins to make use of our mechanism. Our *Buffering/Spill* mechanism has reduced the use of main memory for the workloads of 40M, 50M, 60M and 70M keys. As explained in section 3.4, each time the main memory usage limit is reached, our mechanism move keys and values between main memory and hard disk. Each keyset brought to hard disk allows the release of new memory space, preventing the main memory to be fully consumed, which would force the system to use the swap area, increasing the number of page faults.

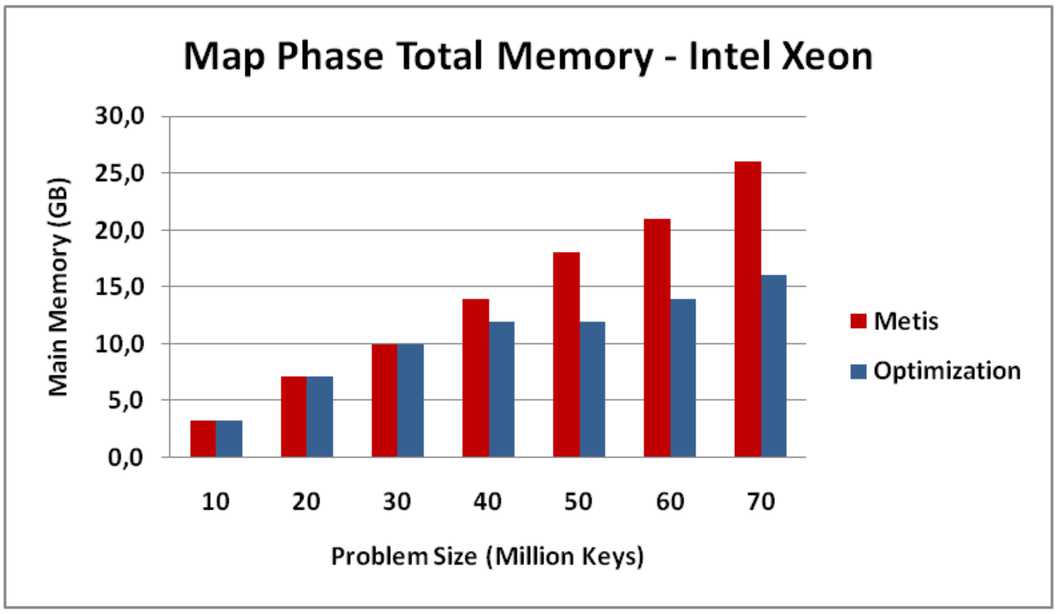

Figure 5.4: Main memory usage on Map phase using Buffering/Spill mechanism.

Also in Figure 5.5 we can observe the same behavior as Figure 5.4, but with a smaller reduction in main memory. This situation is justified by the fact that the Reduce phase needs more data in memory to make the reduction, in addition to produce more intermediate data in main memory before the old data can be freed. Unlike the Map phase where the main work consists of producing and storing keys in a intermediate data structure, during the Reduce phase the data must be read from an intermediate structure, processed and stored in a second structure, forcing the framework to hold more keys at the same time into main memory.

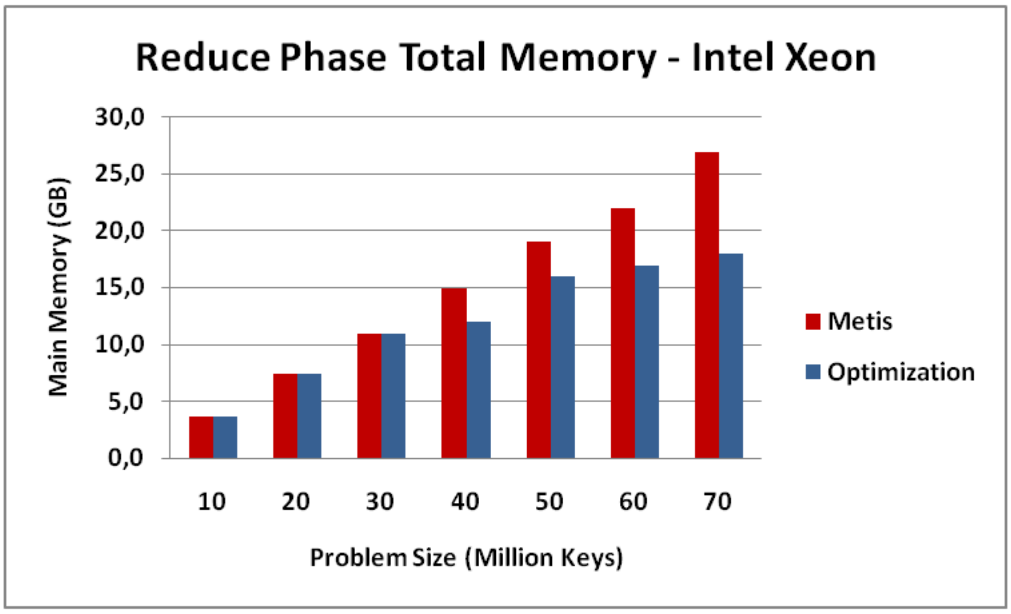

Figure 5.5: Main memory usage on Reduce phase using Buffering/Spill mechanism.

5.4.3 Buffering and Spill Major Page Faults

Using the Intel Core Duo environment with 6GB of main memory, our goal in this experiment was to verify the possible reduction in page faults making use of our *Buffering/Spill* mechanism.

A page fault is an interrupt triggered by the hardware when an application accesses a page mapped into the virtual memory space, but that was not loaded into main memory. Page faults, by its own characteristic, degrade the performance of an application. Optimizations that reduce the number of page fault occurrences improve application performance and also system performance. In general, an optimization can go through the reduction of total main memory usage as shown in section 5.4.2.

In a page fault occurrence, the system uses the swap as an extension of memory. In contrast, we define in this mechanism the need to reduce these pages faults, since the constant interaction of the system with the hard disk becomes costly to the execution time. Moving a set of keys to the hard disk before the total amount of main memory is consumed, prevents the system to use the swap. In this way it is possible to decrease the number of page faults, also reducing the constant and disordered hard disk access. Figures 5.6 and 5.7 the decrease of page faults, both in the *Map* and *Reduce* phase. Moving the keys to the hard disk in groups, before the main memory be fully consumed, reduces the latency influence of hard disk access at execution time, effect that can be observed by the use of swap.

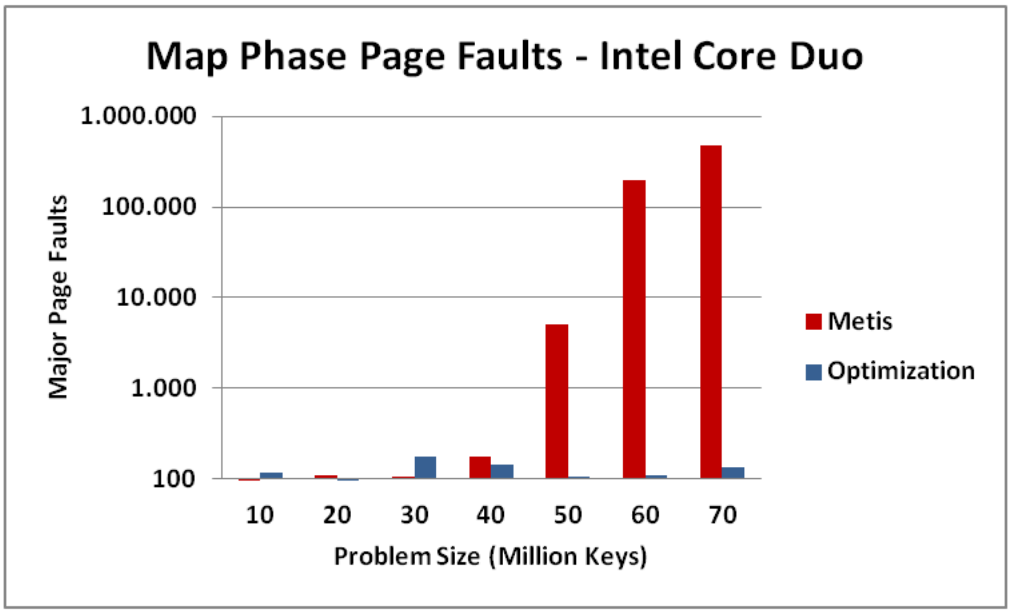

Figure 5.6: Page faults Reduction on Map phase using spill/buffering mechanism.

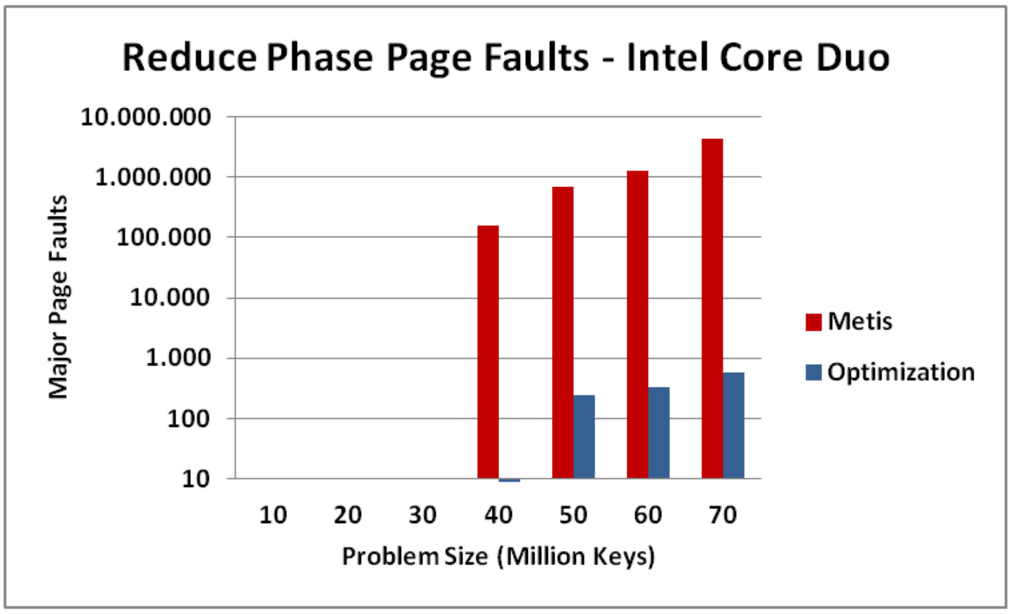

Figure 5.7: Page faults Reduction on Reduce phase using spill/buffering mechanism.

5.4.4 Thread Disk Acess Execution Time

Using the Intel Xeon environment with 96GB of main memory, our goal in this experiment was to verify the efficiency of using the hard and soft limit mechanism in order to avoid competition between different threads to move keys to hard disk. For this, we have designed an experiment using the workloads described in Table 5.3. The objective is to verify the improvement in execution time, caused by less competition among threads for access to the hard disk.

In the Figure 5.8 it shows that for execution with a limit of 10%, where only a hard limit is used, there is a significant increase in execution time. This situation is the consequence of using the 24 threads available to move large amounts of keys to the hard disk at the same time. When more than one thread tries to write a set of keys to the hard disk, it makes the hard disk receive a long list of requests. Concurrent writes from different threads will issue seek operations, introducing an overhead which could slow down the system, depending on the number of keys written.

Keeping the 10% hard limit, and now making use of the soft limit strategy, we can see an execution time improvement of up to 2x, because fewer threads access the hard disk at the same time. The actual value of the hard limit, combined with the type of input and the memory usage, reflects in more keys moved to hard disk and a higher reduction of memory usage, or fewer keys moved to hard disk and a lower memory reduction. The knowledge about the workload type in use, combined with the configuration of these parameters may provide better performance of the multi-core MapReduce framework.

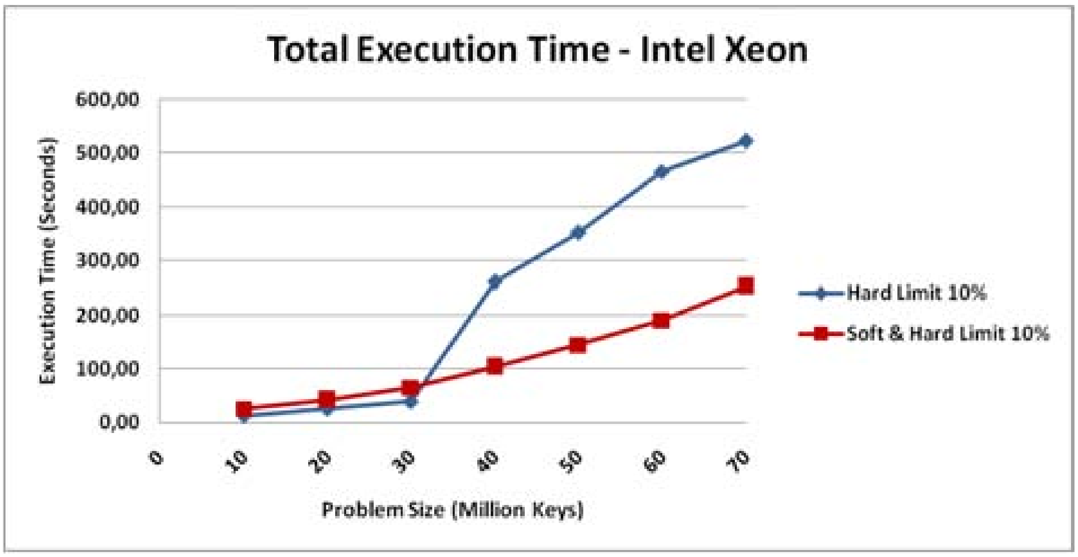

Figure 5.8: Total execution time using hard and soft limit.

5.5 Partial MapReduce

In this section, the idea was to determine the efficiency of our partial MapReduce proposal. In addition to supporting a small intermediate data structure, our proposal allows to organize the framework operation in different ways to reduce the keys and values as soon as possible, reducing the main memory consumption.

For this experiment we used both the Intel Xeon and the AMD Opteron, since both have different amount of cores sharing the cache memory. While in the Intel Xeon each 6 cores share 12MB of L3 cache memory, in the AMD Opteron the same 12MB of L3 cache memory are shared among 16 cores. Knowing that the use of partial MapReduce provides a cache memory locality increase, we expected that such difference between architectures also would provide different performance results.

In this experiment we use the Word Count benchmark running over our proposal, in comparison with Phoenix++ and the original Metis. We also show the performance of Inverted Index benchmark. A comparison is made between the original Metis and our proposal, since Phoenix++ does not provide a Inverted Index benchmark.

5.5.1 Word Count Total Execution Time on Intel Xeon

Considering total execution time, our implementation achieves better performance than Metis and Phoenix++. The results in the total execution time of our implementation are significantly smaller compared to Metis and Phoenix++.

We designed an experiment where resizing the intermediate data structure and processing the keys as soon as possible, allowed us to reduce the total execution time. In the Figure 5.9, it is possible to see that our implementation get a significant improvement in execution time, both against Metis as Phoenix++. The goal of our implementation, was to seek the best possible configuration, to improve the results preventing the growth of the intermediate data structure, and keeping the locality with the cache memory.

Among the three frameworks used, as seen in Figure 5.9 Phoenix++ has the worst performance. We can partially conclude, that the cost of resizing the hash table size and copy of the keys to the new hash table represent an important part of the total execution time. On the other hand, the large amount of distinct keys produced, makes it difficult for Phoenix++ to keep cache memory locality. This situation makes the execution of Phoenix++ to be dominated by non-uniform main memory access.

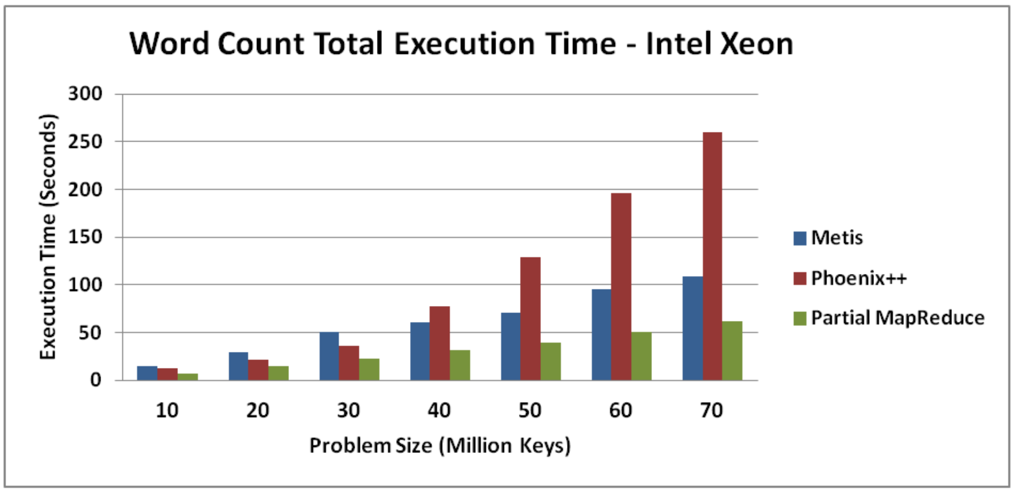

Figure 5.9: Intel Xeon Total Time using Word Count Benchmark.

The improvements presented in Figures 5.9 can be seen in the table 5.4. Our implementation has significant improvements over the original version of Metis and Phoenix++. Over Metis can be seen that in total execution time over Intel Xeon environment our implementation can be up to 2,3 times faster, while against Phoenix++ our implementation is up to 4.2 times faster. It is also possible to see in table 5.4, that the reduction in execution time is not uniform, since the distribution of keys directly influences the pressure over the framework.

Table 5.4: Total Execution Time Speedup on Intel Xeon using Word Count Benchmark.

Input Size	Keys	Over Metis	Over Phoenix++
3.9GB	10M	2.2x	1.8x
7.8GB	20M	2.1x	1.5x
11.7GB	30M	2.3x	1.6x
15.6GB	40M	2.0x	2.5x
19.6GB	50M	1.8x	3.3x
23.5GB	60M	1.9x	3.9x
27.4GB	70M	1.8x	4.2x

In Figure 5.10, it is possible to see that our proposal achieves significantly lower L3 Cache miss rate than Phoenix. The reduction is smaller compared with Metis, since we use Metis as the basis for our implementation. Among the main points of this reduction we can include using a relatively smaller intermediate data structure, and always keep reduced keys to avoid keys repetition and the allocation cost of keeping the repeated keys .

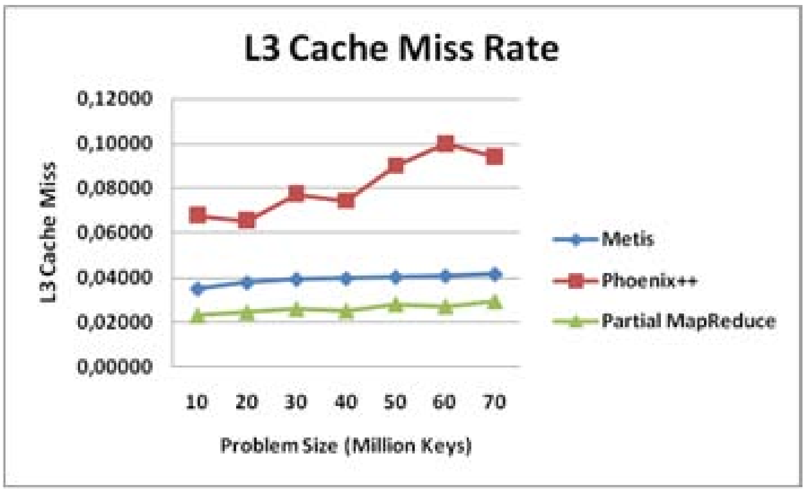

Figure 5.10: Word Count L3 Cache Memory miss rate on Intel Xeon.

By maintaining a smaller intermediate data structure, each instruction requires less CPU cycles to complete, as viewed in Figure 5.11. Keeping a smaller intermediate data structure results in fewer cycles to process and sort keys.

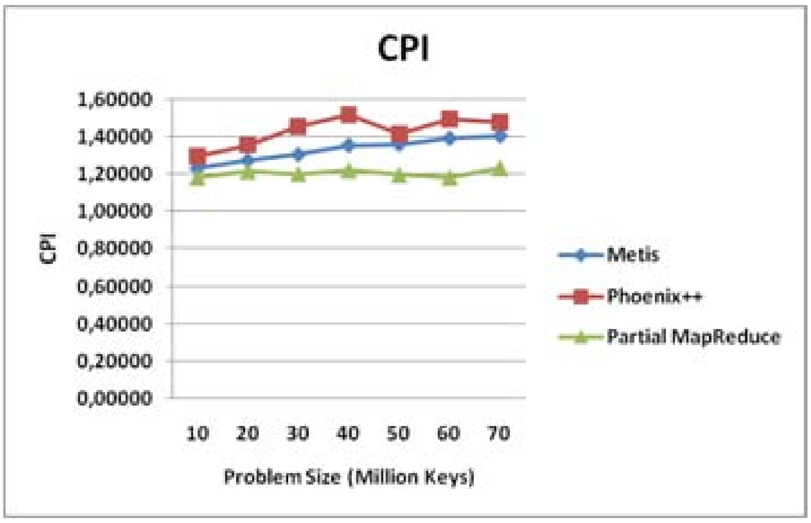

Figure 5.11: Word Count Benchmark CPI on Intel Xeon.

5.5.2 Word Count Map Phase Total Execution Time on Intel Xeon

Our objective is to reduce the cost of the Map tasks execution by taking advantage of spatial locality between the processor and the cache memory, allowed a reduction of the costly work done by the map tasks. When executing a map task, the framework deals with three main operations, seeking position in the hash table, get the position in the tree and balance resultant trees.

Search the hash entry corresponding with key is the less problematic, such operation has constant access and complexity $O(1)$. After finding the hash entry, the map task must find the key position in the tree with the complexity $O(logN)$. In the original version of Metis, trees with 10 keys did not offer a high cost in balance, but when using smaller hash tables, consequently it was necessary to use more large trees. This decision forced us to consider the cost of balancing the trees during their growth. In short, we conclude that the use of cache memory has benefited mainly the operations done on the trees of each hash entry. For workloads where the rate of key repetition is low, the replacement of trees per array can provide improved insertion time, but it should be considered that maintaining sorted arrays may be more costly.

On the other hand Phoenix++ does not use B+trees, leaves the work of sorting the keys to Merge phase, avoiding the cost of sorting in the Map phase.

As mentioned in section 2.2.3, Metis uses a prediction phase to define the main parameters of intermediate data structure. Also during the prediction phase Metis also allocates the intermediate data structure. Phoenix++ on the other hand has an initialization phase which takes the responsibility of cost allocation. In our implementation, due to using a fixed size intermediate data structure, the original prediction phase becomes unnecessary, in this way on our implementation we establish all memory allocation for intermediate data structure in the Map phase.

Knowing that multi-core MapReduce frameworks extensively use the main memory, and we also considered memory allocation to have a cost in execution time. For purposes of comparison at the execution time, in Phoenix++ the Map phase was counted together with the cost of allocation in the initialization phase, as well as Metis, for the allocation of the intermediate data structure made in the prediction phase.

As seen in Figure 5.12, the Map phase execution time follows more or less the same behavior of the total execution time. It is still possible to see a significant Map phase reduction in execution time of our implementation over the original Metis and Phoenix++, showing the advantage provided by the spatial locality of data.

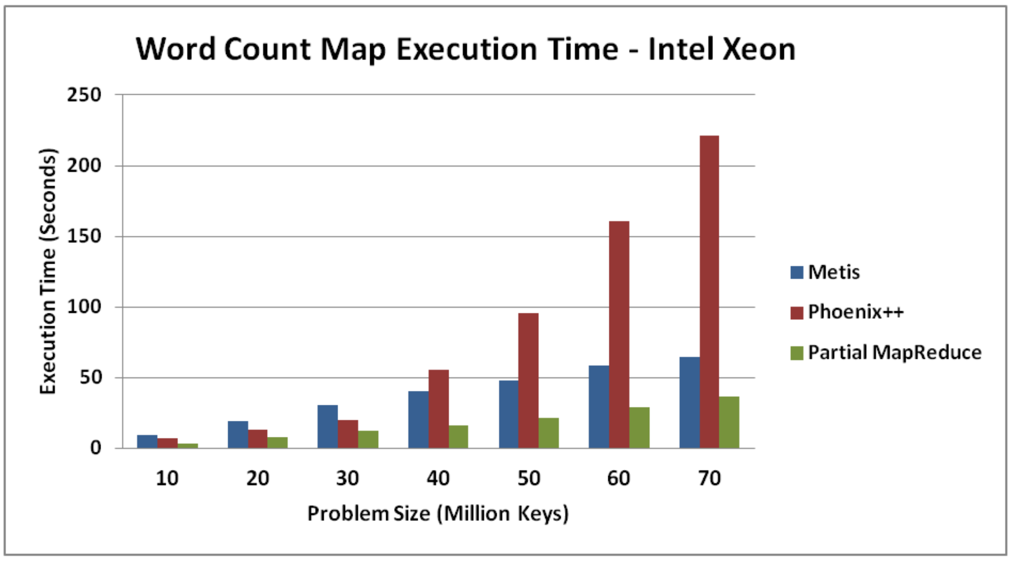

Figure 5.12: Intel Xeon Map phase execution time using Word Count Benchmark.

In the table 5.5 it can be seen in detail the results shown in Figure 5.12, where we achieved a speedup between 1.8 and 3 compared against Metis, and between 1.6 and 6.1 compared against Phoenix++.

Table 5.5: Map Phase Execution Time Speedup on Intel Xeon using Word Count Benchmark.

Input Size	Keys	Over Metis	Over Phoenix++
3.9GB	10M	3.0x	2.2x
7.8GB	20M	2.4x	1.6x
11.7GB	30M	2.5x	1.6x
15.6GB	40M	2.5x	3.6x
19.6GB	50M	2.2x	4.4x
23.5GB	60M	2.0x	5.5x
27.4GB	70M	1.8x	6.1x

5.5.3 Word Count Reduce Phase Total Execution Time on Intel Xeon

In original Metis execution, regardless of the size of the intermediate data structure, a reduce task would do the work of reducing a given column once. By dividing the original implementation in MapReduce parts, reduce tasks are forced to fall back on all the intermediate data structure for each MapReduce part generated. However the reduction of the intermediate data structure to a size that can be working on cache memory, allowed this extra work did not create a significant overhead.

Another advantage that reduced the overhead created by the extra work of reduce tasks, was the number of columns used in our intermediate data structure. In the original version of Metis, for an input file with 10M of different keys, a hash table with about 1M columns would be required. For overhead in reduce phase became significant at execution time, it would be necessary that our implementation utilizes about 100 MapReduce parts, much higher than the 10 needed for the current input file 70M keys.

As can be seen in Figure 5.13, the Reduce phase already has a relatively short execution time, however our implementation has better execution time than the original Metis and Phoenix++, due to the small intermediate data structure and spatial locality.

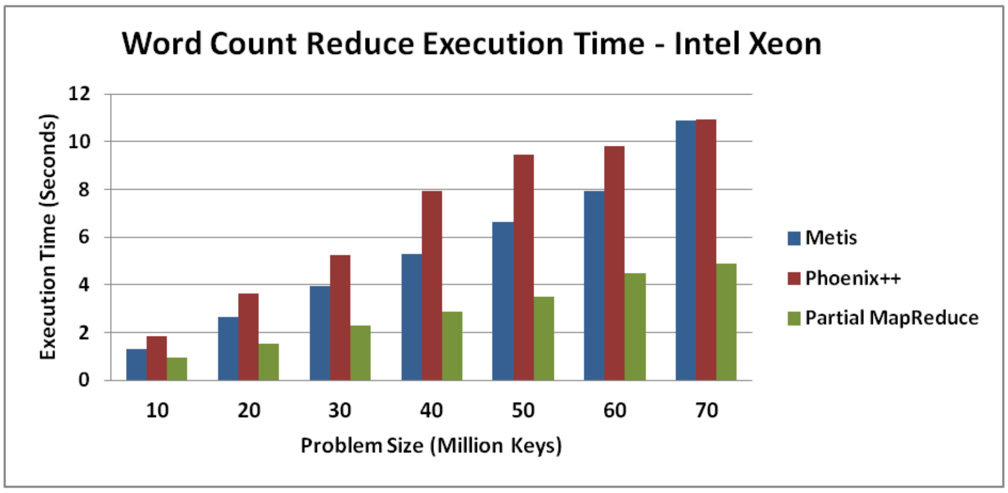

Figure 5.13: Intel Xeon Reduce phase execution time using Word Count Benchmark.

In Table 5.6, it is possible to see the speedup obtained in the Reduce phase, caused by the use of our proposal. Unlike the Map phase, the Reduce phase presents a relatively stable speedup compared Metis and Phoenix++.

Among the three main phases, Map, Reduce and Merge, the Reduce is the phase that consumes smaller execution time, making it relatively difficult to achieve large execution time reductions.

Table 5.6: Reduce Phase Execution Time Speedup on Intel Xeon using Word Count Benchmark.

Input Size	Keys	Over Metis	Over Phoenix++
3.9GB	10M	1.4x	1.9x
7.8GB	20M	1.7x	2.4x
11.7GB	30M	1.7x	2.3x
15.6GB	40M	1.8x	2.8x
19.6GB	50M	1.9x	2.7x
23.5GB	60M	1.8x	2.2x
27.4GB	70M	2.2x	2.2x

5.5.4 Word Count Merge Phase Total Execution Time on Intel Xeon

In Metis and Phoenix++, the Merge phase is called only after all keys having been processed and reduced. For these implementations, the Merge phase is only responsible for sorting and join the different key blocks.

On the other hand, the Merge phase of our implementation beyond sort and join different key blocks, is also responsible for reducing possible repeated keys. When our implementation generates many MapReduce parts, depending on the distribution of input keys, it is possible to arrive repeated key on the Merge phase. If there are many repeated keys in the Merge phase, our implementation can have lower performance than Metis and Phoenix++.

As can be seen in Figure 5.14 compared with Phoenix++, our implementation always has better execution time. On the other hand, when comparing against Metis. In such comparison, it is only possible to get better performance in the larger input files, because it is directly influenced by the key distribution these input files that has no extra work to merge phase.

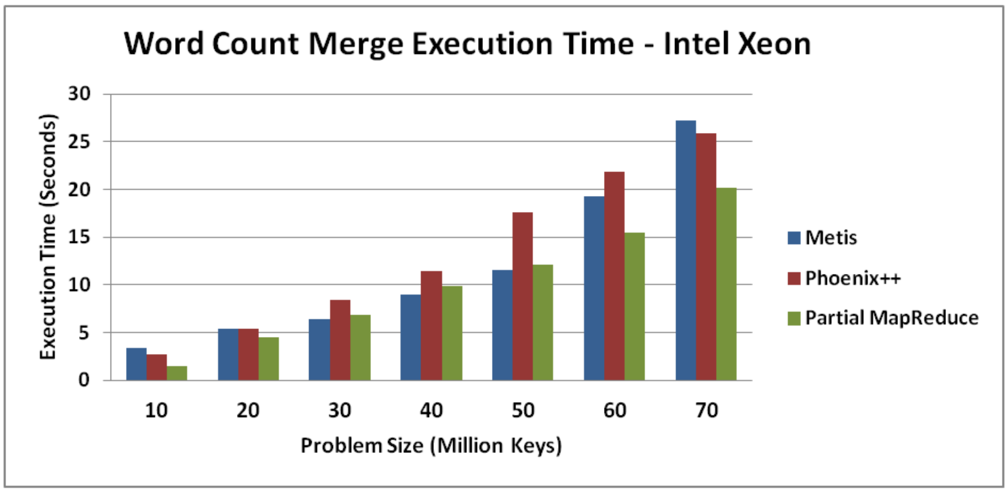

Figure 5.14: Intel Xeon Merge phase execution time using Word Count Benchmark.

In Table 5.7, it is possible to see that for the 30M and 40M input files , our implementation introduces a slight worsening versus Metis. This situation is caused by the existence of possible repeated keys, and the need to make reductions in the Merge phase.

Table 5.7: Merge Phase Execution Time Speedup on Intel Xeon using Word Count Benchmark.

Input Size	Keys	Over Metis	Over Phoenix++
3.9GB	10M	2.2x	1.8x
7.8GB	20M	1.2x	1.2x
11.7GB	30M	0.9x	1.2x
15.6GB	40M	0.9x	1.2x
19.6GB	50M	1.0x	1.5x
23.5GB	60M	1.2x	1.4x
27.4GB	70M	1.3x	1.3x

5.5.5 Word Count Total Main Memory Usage on Intel Xeon

A characteristic present in Metis was the high main memory usage, pushed forward with the parallelism of the environment, which made the prediction phase used by Metis became the main responsible. Using a simple calculation, Metis tries to predict the number of distinct keys based on the 10% of the initial input file, and scale the size of the hash table.

The main fault on the prediction phase, is not take into account the parallelism factor, and eventually allocate memory space much larger than would be needed, causing a high waste of main memory in parallel environments. A major question was to understand if it was actually necessary for example, for a input with 10M keys in an environment of two threads to have 2M hash entries, while in an environment of 64 threads 64M hash entries were necessary . Resizing the intermediate data structure was clearly the change that could bring many benefits to the framework. Knowing that the main memory is a limited resource, the possibility of using fewer resources is always a good option. Such a change offers an advantage to all phases, even if different types of workload pressure intermediate data structure in different ways.

Phoenix++ already works with the idea that different Map tasks produce different amounts of keys, using hash tables of varying size and achieving a better use of main memory.

In Figure 5.15 it can be seen the reduction in main memory achieved, using our implementation against Metis and Phoenix++. Our implementation using partial MapReduce, allows us to make best use of main memory available space. Against the original version of Metis, we managed to get a significant reduction in use of main memory. However as it is possible to see in Figure 5.15, the use of partial MapReduce and a fixed size intermediate data structures, gets more or less the same main memory usage than Phoenix++, although the strategies are quite different.

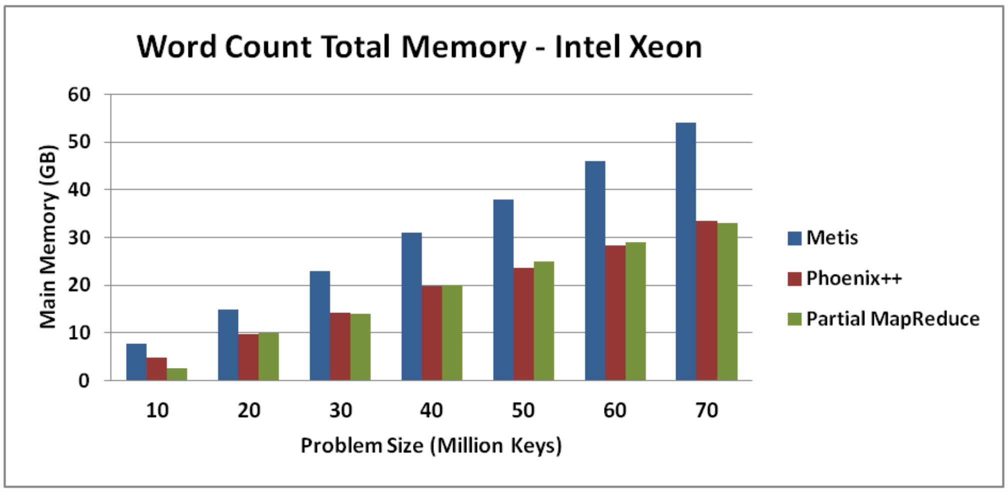

Figure 5.15: Intel Xeon main memory usage using Word Count Benchmark.

The reduction in main memory usage of our implementation against Metis and Phoenix++, can be seen in detail in table 5.8 for the Intel Xeon environment.

Table 5.8: Reduction in main memory usage on Intel Xeon using Word Count Benchmark.

Input Size	Keys	Over Original Metis	Over Phoenix++
3.9GB	10M	-67%	-46%
7.8GB	20M	-34%	+1%
11.7GB	30M	-39%	-1%
15.6GB	40M	-35%	+1%
19.6GB	50M	-34%	+6%
23.5GB	60M	-37%	+2%
27.4GB	70M	-39%	-1%

5.5.6 Inverted Index Total Execution Time on Intel Xeon

As a second option, we decided to prove the efficiency of our proposal using Inverted Index benchmark. The Inverted Index maps terms and their occurrences in a document or set of documents. It is an indexing strategy that allows performing quick and accurate searches in exchange for greater difficulty in the act of inserting and updating documents. It is usually built on a traditional list of documents, is so called because reverse the hierarchy of information.

Unlike the Word Count application, where Map tasks simply must emit all found words. In Inverted index, each key emited requires more processing than is necessary in word count.

As it is possible to see in Figure 5.16, the use of partial MapReduce show an improvement even when using an different application from the Word Count initially used. Unlike the original Metis, our proposal is able to maintain a low growth at execution time when increase the workload, as can been see in Figure 5.16 mainly for 60M and 70M inputs .

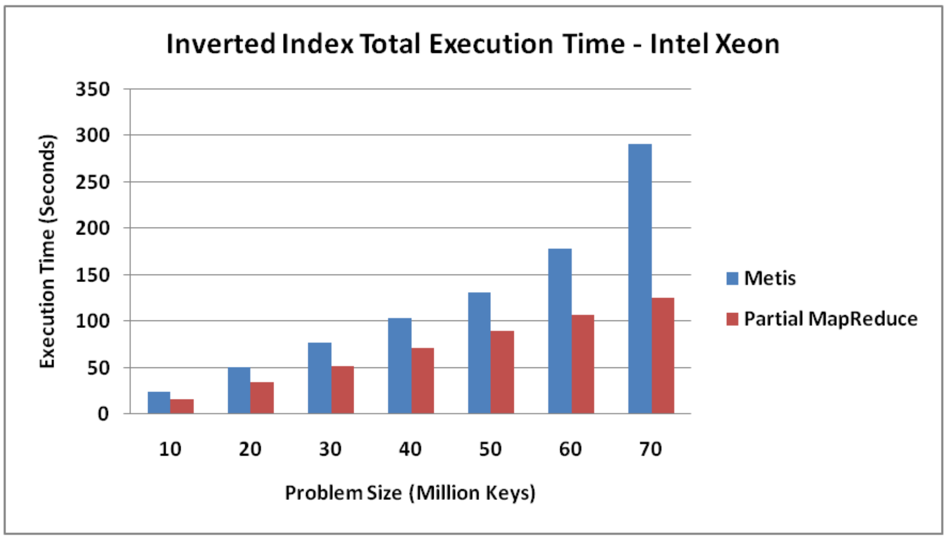

Figure 5.16: Intel Xeon Total Time using Inverted Index Benchmark.

In table 5.9 it can be seen that our proposal was able to achieve a speedup between 1.5 and 2.3 times faster than the original Metis, as was also shown in Figure 5.16.

Table 5.9: Total Execution Time Speedup on Intel Xeon using Inverted Index Benchmark.

Input Size	Keys	Over Metis
3.9GB	10M	1.5x
7.8GB	20M	1.5x
11.7GB	30M	1.5x
15.6GB	40M	1.5x
19.6GB	50M	1.5x
23.5GB	60M	1.7x
27.4GB	70M	2.3x

5.5.7 Inverted Index Map Phase Total Execution Time on Intel Xeon

Looking with more detail, it can been seen in Figure 5.17 compared with the figure 5.16 that Map phase has significant influence on the framework execution time. Such a situation is justified because the Map phase is responsible for reading all the input file information, produce and organize all the keys in the intermediate data structure. As seen in Figure 5.17, it is possible to obtain a significant improvement in execution time for the 70M input file.

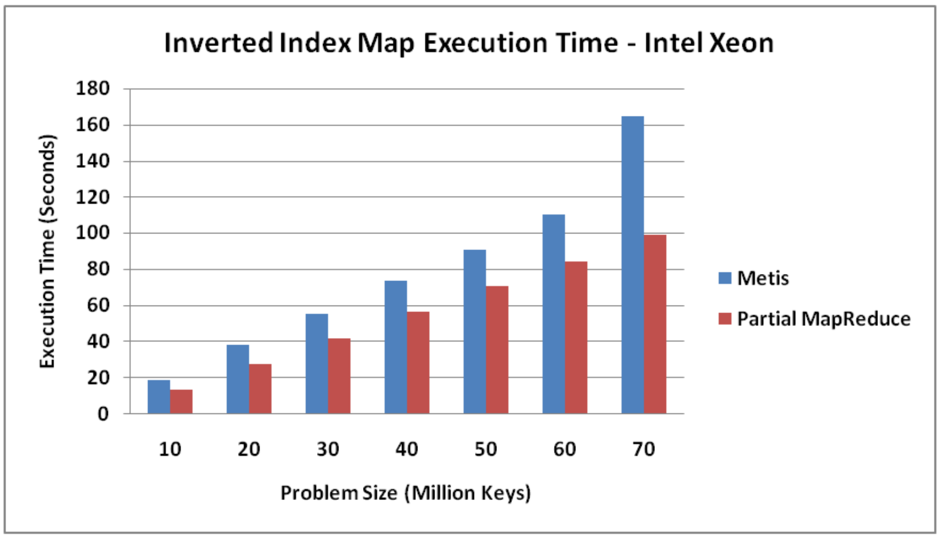

Figure 5.17: Intel Xeon Map phase execution time using Inverted Index Benchmark.

The details of the executions shown in Figure 5.17 can be appreciated with more detail in the table 5.10, where it can be seen that our proposal led to a speedup between 1.3 to 1.7 times faster than the original Metis.

Table 5.10: Map Phase Execution Time Speedup on Intel Xeon using Inverted Index Benchmark.

Input Size	Keys	Over Metis
3.9GB	10M	1.4x
7.8GB	20M	1.4x
11.7GB	30M	1.3x
15.6GB	40M	1.3x
19.6GB	50M	1.3x
23.5GB	60M	1.3x
27.4GB	70M	1.7x

5.5.8 Inverted Index Reduce Phase Total Execution Time on Intel Xeon

Unlike the Map phase where the keys are mapped from an input file and organized into a intermediate data structure, the Reduce phase only needs to process the keys already organized in intermediate data structure. Such a situation allows the reduce phase have a linearly stable execution time, as can be seen in figure 5.18. Even so, our proposal is able to show improvement for all workloads.

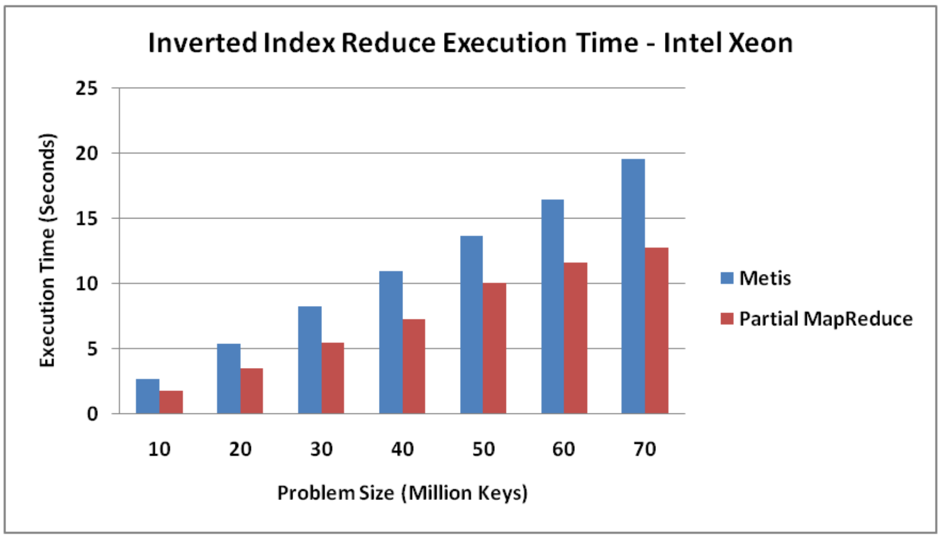

Figure 5.18: Intel Xeon Reduce Phase Execution Time using Inverted Index Benchmark.

As can be seen in table 5.11, the use of our proposal allowed a small improvement in the execution time of the Reduce phase, using as a benchmark the Inverted Index application. With a stable behavior, we managed to get speedup between 1.4 and 1.6 faster than the original Metis.

Table 5.11: Reduce Phase Execution Time Speedup on Intel Xeon using Inverted Index Benchmark.

Input Size	Keys	Over Metis
3.9GB	10M	1.6x
7.8GB	20M	1.6x
11.7GB	30M	1.5x
15.6GB	40M	1.5x
19.6GB	50M	1.4x
23.5GB	60M	1.4x
27.4GB	70M	1.5x

5.5.9 Inverted Index Merge Phase Total Execution Time on Intel Xeon

Some large workloads have a Reduce phase that emits many items, so that the final Merge has significant cost. For these types of workloads, sorting the keys in final output may reduce the cost of the Merge. In short, for these workloads it may be beneficial that the intermediate data structure store the keys in sorted order.

In the original Metis, the Merge phase receives a large number of short vectors ordered from 1 to 10 keys to make a global sorting. In our proposal, by make a partial reduction, the Merge phase receives a smaller set of ordered vectors, where the number of keys sorted by vector is significantly higher than the original Metis. By receives a higher number of sorted keys, the cost of making a global sort becomes relatively smaller. This advantage can be seen in Figure 5.19, mainly for 70M input, where we achieved speedup that reaches 6.1 times faster than the original Metis.

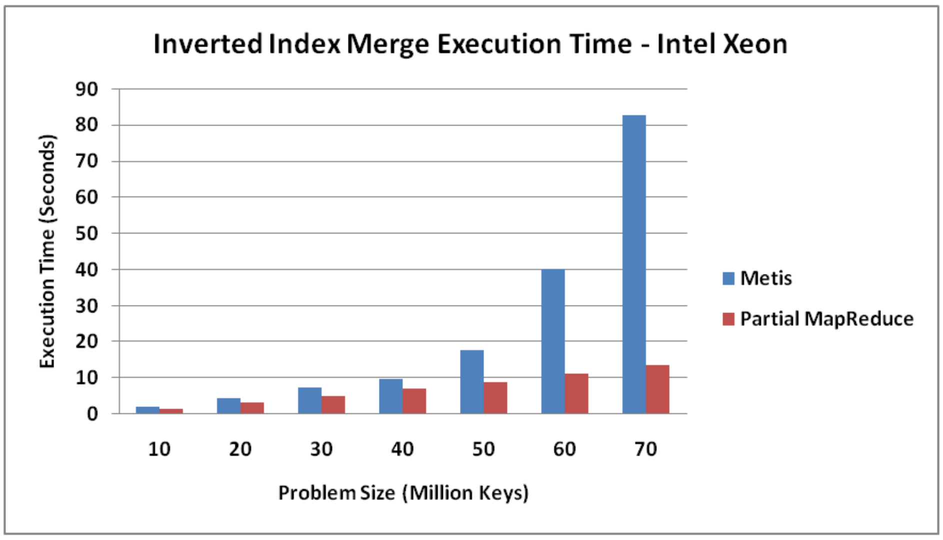

Figure 5.19: Intel Xeon Merge Phase Execution Time using Inverted Index Benchmark.

The speedup details of the Figure 5.19 can be seen with more detail in the table 5.12 below.

Table 5.12: Merge Phase Execution Time Speedup on Intel Xeon using Inverted Index Benchmark.

Input Size	Keys	Over Metis
3.9GB	10M	1.4x
7.8GB	20M	1.4x
11.7GB	30M	1.5x
15.6GB	40M	1.4x
19.6GB	50M	2.0x
23.5GB	60M	3.6x
27.4GB	70M	6.1x

5.5.10 Inverted Index Total Main Memory Usage on Intel Xeon

In the original Metis, to process large number of different keys it would need a large intermediate data structure. In our proposal, the size of the intermediate data structure related to the amount of LLC available. As each part of MapReduce execution uses a small intermediate data structure, it is possible to always keep the smallest possible number of keys in memory.

By taking advantage of a small data structure intermediate as well as improvements in execution time, it is also possible to see an improvement over the amount of main memory usage. Making partial data reduction prevents the intermediate data structure to grow without control. As a result of our proposal it can be seen in Figure 5.20 that using Inverted Index, our framework can achieve a reduction up to 25% in main memory usage compared to Metis.

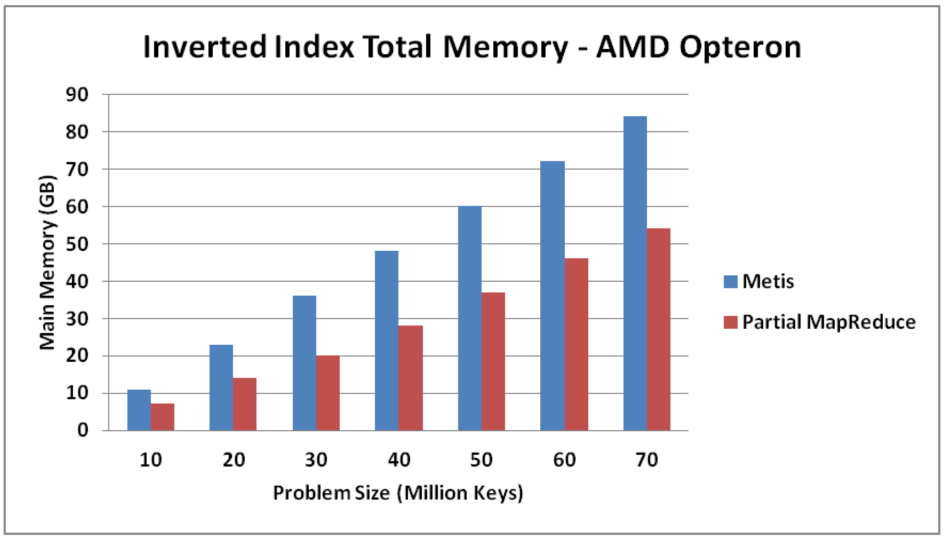

Figure 5.20: Intel Xeon Main Memory Usage using Inverted Index Benchmark.

In the table 5.13 it is also possible to see that our proposal achieves the reduction in memory usage ranging between 19% and 25% for all types of workloads.

Table 5.13: Reduction in Main Memory Usage on Intel Xeon using Inverted Index Benchmark.

Input Size	Keys	Over Original Metis
3.9GB	10M	-19%
7.8GB	20M	-25%
11.7GB	30M	-23%
15.6GB	40M	-24%
19.6GB	50M	-24%
23.5GB	60M	-23%
27.4GB	70M	-23%

5.5.11 Word Count Total Execution Time on AMD Opteron

In Figure 5.21, it can be seen that the original Metis lose performance when handling a large volume of keys in a more parallel environment. On the other hand, our implementation are able to maintain roughly the same performance. While the Intel Xeon has 6 cores and 12 threads sharing the 12MB of cache memory, the AMD Opteron keeps 16 cores sharing the same 12MB of cache memory, which makes it difficult to improve the execution time performance on AMD Opteron.

Previous work reported that accesses to the main memory are 33% slower than accesses to cache memory. Any program that uses more than 64 threads will experience such non-uniform latency[38]. This may partially explain the drop in Metis performance, since it is necessary to handle with large datasets in main memory through the many threads.

To keep the spatial locality in AMD, the path to follow is to reduce the size of the intermediate data structure. However the decrease of the intermediate data structure, requires that our implementation launches many MapReduce parts to prevent the intermediate data structure to grow sufficiently to lose locality with cache memory. As explained in section 4.3.1, a small structure causes more interruptions in the execution, which may cause loss of performance.

In terms of execution time, for our implementation, the number of threads that are sharing the cache memory on the Intel Xeon processor provides better locality with cache memory and hence better performance, as shown in Figure 5.9.

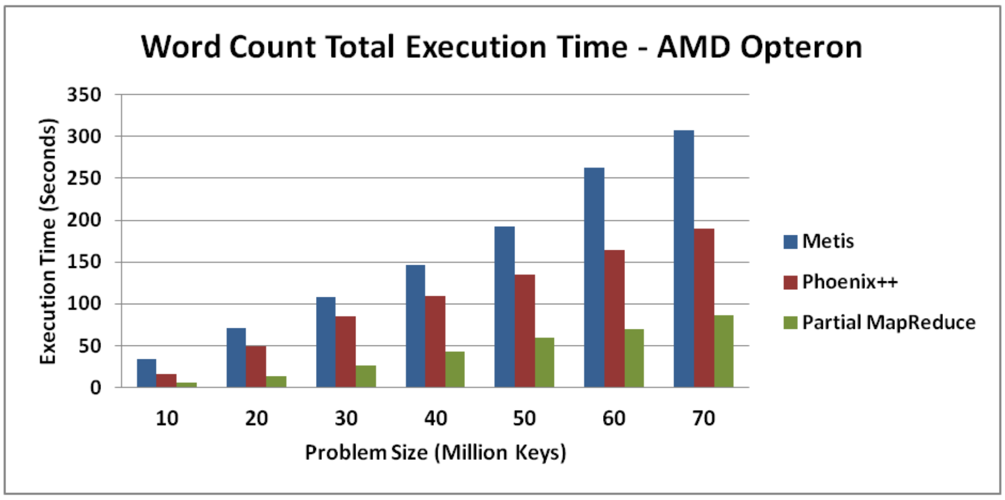

Figure 5.21: AMD Opteron Total Time using Word Count Benchmark.

The improvements presented in Figures 5.21 can be see in the table 5.14. As can be seen in table 5.14, our implementation can be up to 5.6 times faster than Metis, and up to 3.5 times faster than Phoenix++.

Table 5.14: Total Execution Time Speedup on AMD Opteron using Word Count Benchmark.

Input Size	Keys	Over Metis	Over Phoenix++
3.9GB	10M	5.6x	2.6x
7.8GB	20M	5.0x	3.5x
11.7GB	30M	4.1x	3.3x
15.6GB	40M	3.4x	2.5x
19.6GB	50M	3.2x	2.2x
23.5GB	60M	3.8x	2.4x
27.4GB	70M	3.6x	2.2x

5.5.12 Word Count Map Phase Total Execution Time on AMD Opteron

Following the pattern of Figure 5.21, over the total execution time in Figure 5.22 it is possible to see that the Map phase execution time follows the same pattern as the total execution time. By making use of arrays instead of trees B+tree, Phoenix++ achieves better spatial locality than Metis, although there are more threads sharing the same memory cache. However our implementation follows the same intermediate data structure style and can get better Map phase execution time performance than both.

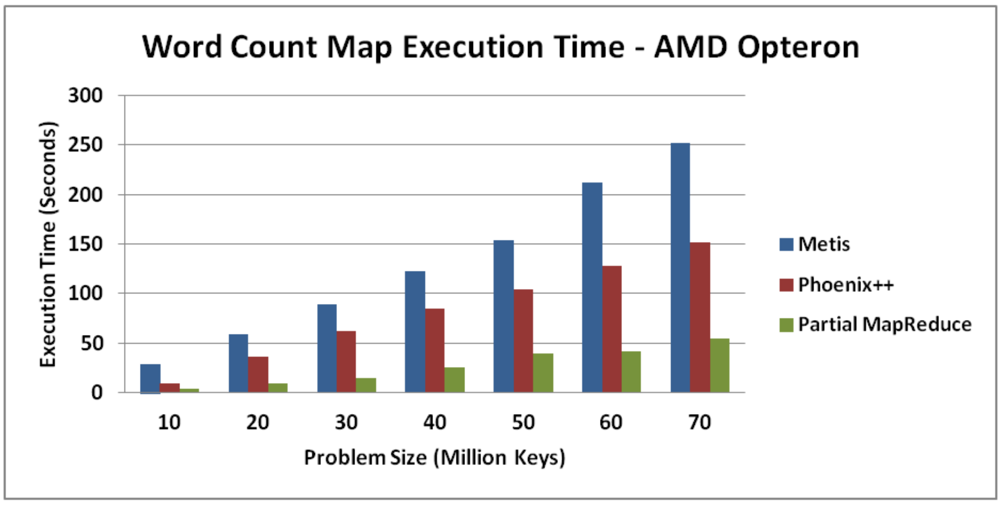

Figure 5.22: AMD Opteron Map phase execution time using Word Count Benchmark.

Looking in more detail through the table 5.15, as can be see that our implementation can be up to 7.2 times faster than Metis. Against the time taken in Intel Xeon, while

Metis lose performance, Phoenix++ can still be a little faster. Yet our implementation can be up to 4.1 times faster than Phoenix++ at the Map phase execution time.

Table 5.15: Map Phase Execution Time Speedup on AMD Opteron using Word Count Benchmark.

Input Size	Keys	Over Metis	Over Phoenix++
3.9GB	10M	7.2x	2.3x
7.8GB	20M	6.4x	3.9x
11.7GB	30M	5.9x	4.1x
15.6GB	40M	4.9x	3.4x
19.6GB	50M	3.9x	2.7x
23.5GB	60M	5.1x	3.1x
27.4GB	70M	4.6x	2.8x

5.5.13 Word Count Reduce Phase Total Execution Time on AMD Opteron

As well as shown by using the Intel Xeon environment, our implementation takes advantage of the small intermediate data structure usage to perform smaller number of reduce tasks, and to be relatively faster. As can be see in Figure 5.23, our implementation outperforms Metis and Phoenix++.

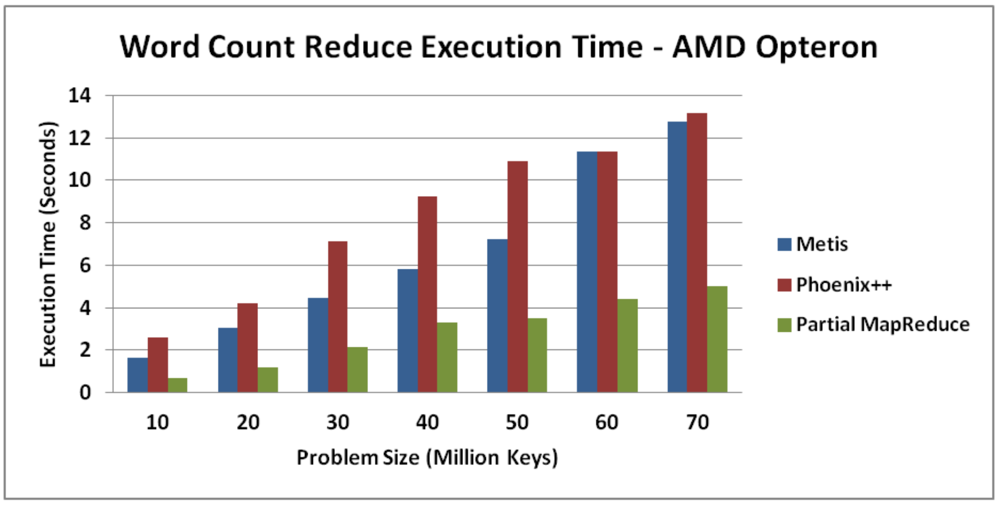

Figure 5.23: AMD Opteron Reduce Phase Execution Time using Word Count Benchmark.

Looking at table 5.16, we can see that our implementation can be up to 2.6 times faster than Metis and up to 3 times faster than Phoenix++ in the Reduce phase using AMD Opteron.

Table 5.16: Reduce Phase Execution Time Speedup on AMD Opteron using Word Count Benchmark.

Input Size	Keys	Over Metis	Over Phoenix++
3.9GB	10M	2.4x	2.8x
7.8GB	20M	2.6x	3.0x
11.7GB	30M	2.1x	2.5x
15.6GB	40M	1.8x	2.4x
19.6GB	50M	2.1x	2.7x
23.5GB	60M	2.6x	2.2x
27.4GB	70M	2.6x	2.2x

5.5.14 Word Count Merge Phase Total Execution Time on AMD Opteron

As explained in section 5.5.4, in our implementation the Merge phase execution time is greatly influenced by the existence of repeated keys in the output of each MapReduce part. This situation makes necessary to make reduction of the repeated keys during the Merge phase. As can be seen in Figure 5.24, unlike the results shown in Intel Xeon, on the AMD Opteron the influence of repeated keys becomes even higher, so that our implementation have worse or equal execution times than Metis and Phoenix++.

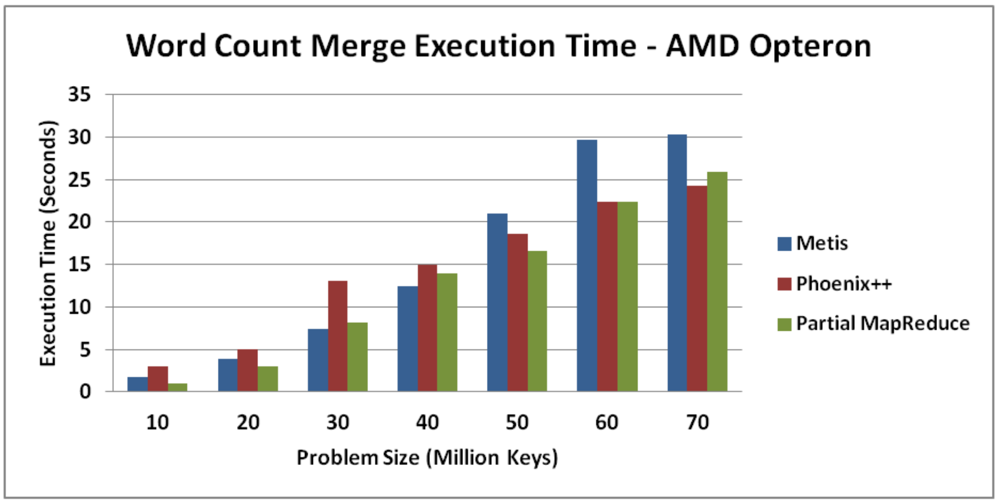

Figure 5.24: AMD Opteron Merge Phase Execution Time using Word Count Benchmark.

As can be seen in table 5.17, compared with Metis is possible to see that the best result does not exceed 1,8 times faster, and for input with 30M and 40M of keys our implementation has worse performance. Compared with Phoenix++, while the input file keys 10M has a key distribution allowing our implementation to be 3.2 times faster. For the input of 70M keys our implementation delivers worse performance than Phoenix++.

Table 5.17: Merge Phase Execution Time Speedup on AMD Opteron using Word Count Benchmark.

Input Size	Keys	Over Metis	Over Phoenix++
3.9GB	10M	1.8x	3.2x
7.8GB	20M	1.3x	1.7x
11.7GB	30M	0.9x	1.6x
15.6GB	40M	0.9x	1.1x
19.6GB	50M	1.3x	1.1x
23.5GB	60M	1.3x	1.0x
27.4GB	70M	1.2x	0.9x

5.5.15 Word Count Total Main Memory Usage on AMD Opteron

As can be seen in Figure 5.25, now using an environment with increased parallelism, while Metis increases the memory usage for supporting the intermediate data structure and parallelism, we maintain the memory usage similar to that used in the environment of Figure 5.15. Now against Phoenix++, as can be seen in Figure 5.25, our implementation achieves slightly better results with less use of main memory.

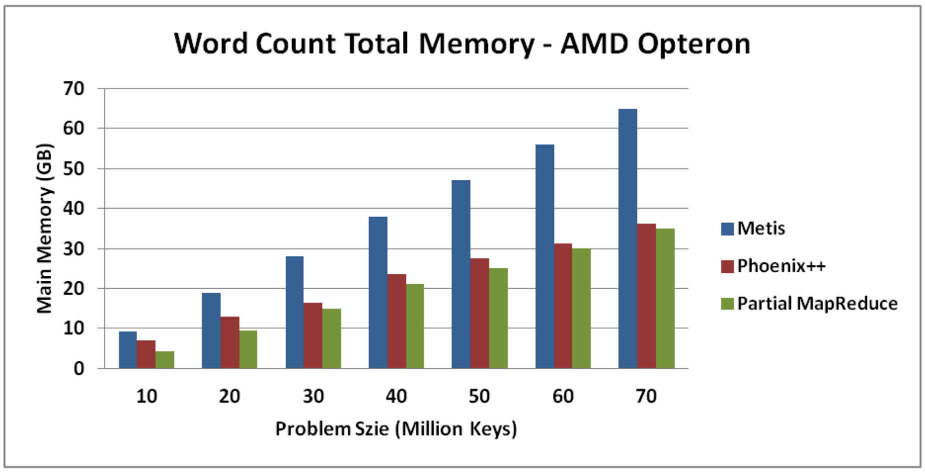

Figure 5.25: AMD Opteron main memory usage using Word Count Benchmark.

Compared with the results using the Intel Xeon, we can see in table 5.18 that our implementation introduces a slight reduction in main memory usage versus Phoenix. On the other hand, comparing versus Metis our implementation still presents significant improvements in the main memory usage, using up to 53% less main memory.

Table 5.18: Reduction in Main Memory Usage on AMD Opteron using Word Count Benchmark.

Input Size	Keys	Over Original Metis	Over Phoenix++
3.9GB	10M	-53%	-39%
7.8GB	20M	-50%	-26%
11.7GB	30M	-46%	-9%
15.6GB	40M	-44%	-10%
19.6GB	50M	-46%	-9%
23.5GB	60M	-46%	-4%
27.4GB	70M	-46%	-3%

5.5.16 Inverted Index Total Execution Time on AMD Opteron

As can be seen in Figure 5.26, comparing the word count application shown in Figure 5.21, the execution time is about 16% higher using inverted index.

Just like the original execution, our proposal follows the same linear execution that original Metis, However, our implementation can be 1.6 times faster. Relatively less than the 5.6 times faster obtained using word count with the same input data size.

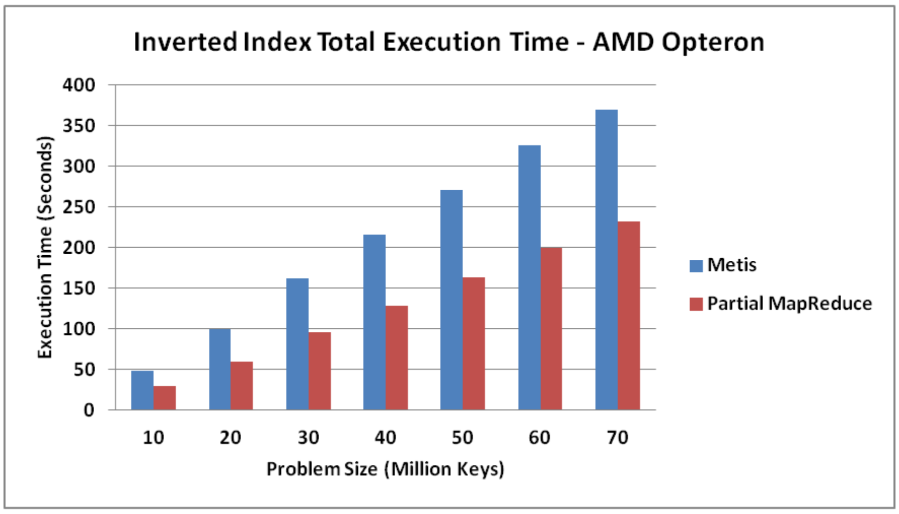

Figure 5.26: AMD Opteron Total Time using Inverted Index Benchmark.

When compared with the Word Count application used shown in table 5.14, the improvement with Inverted index shown in table 5.19 is relatively small. This situation shows the important influence exerted on the framework for different applications.

Table 5.19: Total Execution Time Speedup on AMD Opteron using Inverted Index Benchmark.

Input Size	Keys	Over Metis
3.9GB	10M	1.6x
7.8GB	20M	1.7x
11.7GB	30M	1.7x
15.6GB	40M	1.7x
19.6GB	50M	1.7x
23.5GB	60M	1.6x
27.4GB	70M	1.6x

5.5.17 Inverted Index Map Phase Total Execution Time on AMD Opteron

As in experiments done with Word Count benchmark, also in Inverted Index benchmark the Map phase suffers the highest influence to the overall behavior of the application. Can be seen in Figure 5.27 that the behavior is very similar to Figure 5.26, with an increase in execution time linearly due the size of the input.

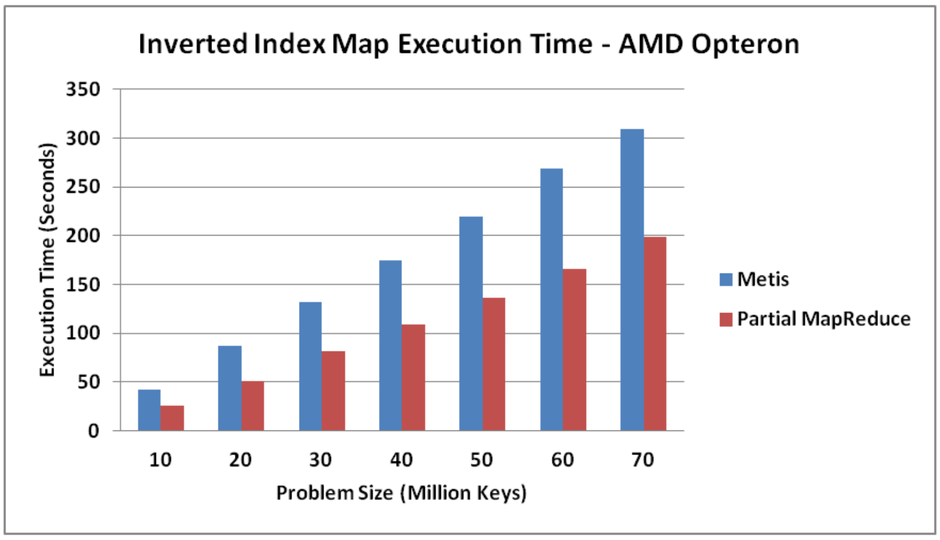

Figure 5.27: AMD Opteron Map phase execution time using Inverted Index Benchmark.

As the total execution time, it is possible to see in table 5.20 that the Map phase keeps between 1.6 and 1.7 times faster than Metis.

Table 5.20: Map Phase Execution Time Speedup on AMD Opteron using Inverted Index Benchmark.

Input Size	Keys	Over Metis
3.9GB	10M	1.7x
7.8GB	20M	1.7x
11.7GB	30M	1.6x
15.6GB	40M	1.6x
19.6GB	50M	1.6x
23.5GB	60M	1.6x
27.4GB	70M	1.6x

5.5.18 Inverted Index Reduce Phase Total Execution Time on AMD Opteron

Analyzing figure 5.28 compared with the results presented in phase Map, it is possible to see that the Inverted index has a strong linear behavior due of the size of input file. On the other hand we can see that unlike Word Count, the Map and Reduce phase in the inverted index the key distribution of the input does not produce large change in the behavior of the framework.

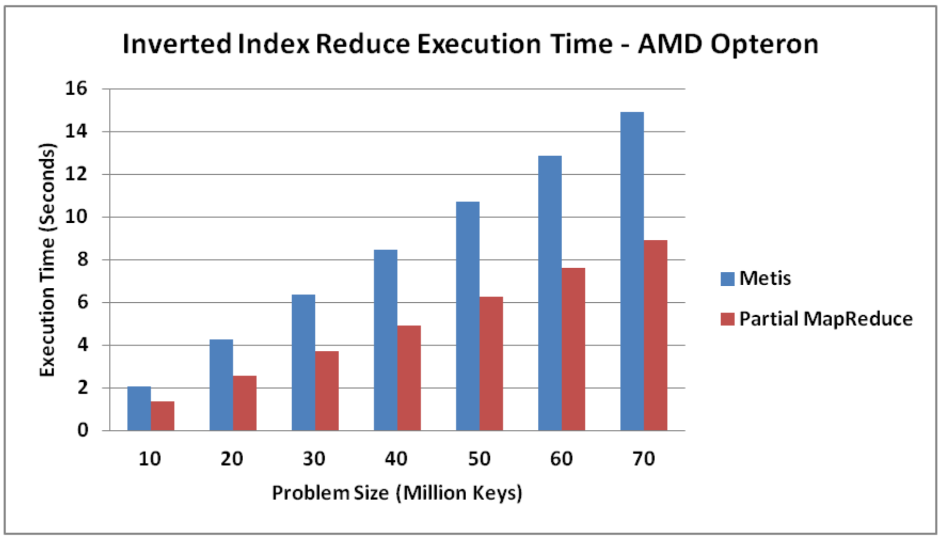

Figure 5.28: AMD Opteron Reduce Phase Execution Time using Inverted Index Benchmark.

As can be seen in table 5.21 the behavior between 1.5 and 1.7 times faster than Metis, is relatively similar to that shown in Map phase through the table 5.20.

Table 5.21: Reduce Phase Execution Time Speedup on AMD Opteron using Inverted Index Benchmark.

Input Size	Keys	Over Metis
3.9GB	10M	1.5x
7.8GB	20M	1.7x
11.7GB	30M	1.7x
15.6GB	40M	1.7x
19.6GB	50M	1.7x
23.5GB	60M	1.7x
27.4GB	70M	1.7x

5.5.19 Inverted Index Merge Phase Total Execution Time on AMD Opteron

The major difference in the behavior of Inverted Index as can be seen in Figure 5.29 happens when using the Merge phase. Between the inputs of 40M and 50M keys it is possible to see that in Merge phase the Inverted Index loses some of the linear characteristic shown so far in other phases. Also during Merge phase we can see that it is possible to get a speedup to 3.5 times faster than Metis, unlike the 1.7 obtained in the Map and Reduce phases.

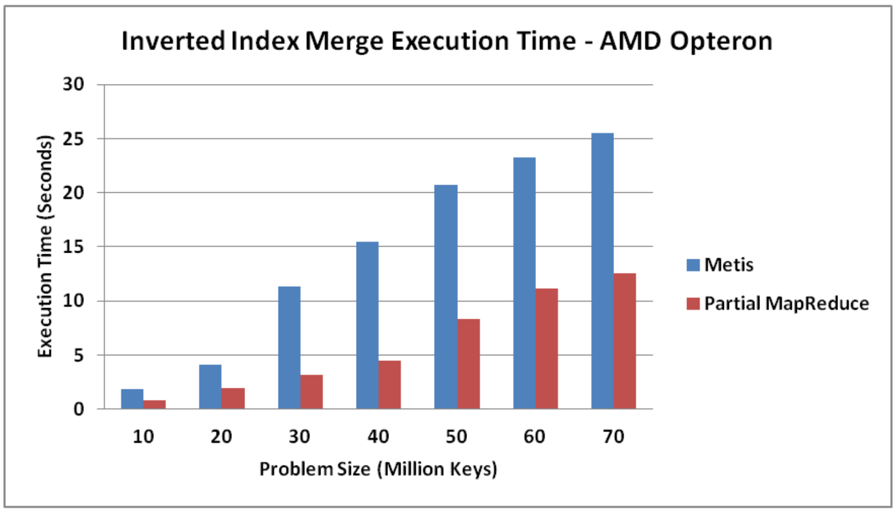

Figure 5.29: AMD Opteron Merge Phase Execution Time using Inverted Index Benchmark.

The table 5.22 presents in more detail the results shown previously in the Figure 5.29.

Table 5.22: Merge Phase Execution Time Speedup on AMD Opteron using Inverted Index Benchmark.

Input Size	Keys	Over Metis
3.9GB	10M	2.1x
7.8GB	20M	2.1x
11.7GB	30M	3.5x
15.6GB	40M	3.4x
19.6GB	50M	2.5x
23.5GB	60M	2.1x
27.4GB	70M	2.0x

5.5.20 Inverted Index Total Main Memory Usage on AMD Opteron

As well as in the use of Word Count benchmark, as shown in Figure 5.30 in the Inverted Index the reduction in the main memory usage follows the same behavior. This situation is influenced mainly by the use of an intermediate data structure that is referenced the cache memory size and effective keys reduction through the use of partial MapReduce.

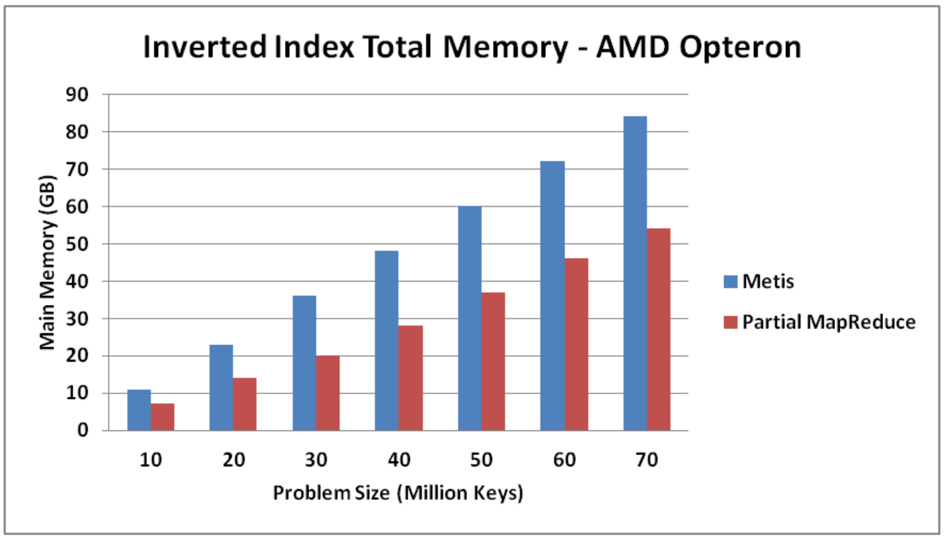

Figure 5.30: AMD Opteron Main Memory Usage using Inverted Index Benchmark.

The table 5.23 presents in more detail the results shown previously in the Figure 5.30, where it is possible to obtain a reduction in main memory usage up to 44%.

Table 5.23: Reduction in Main Memory Usage on AMD Opteron using Inverted Index Benchmark.

Input Size	Keys	Over Original Metis
3.9GB	10M	-35%
7.8GB	20M	-39%
11.7GB	30M	-44%
15.6GB	40M	-42%
19.6GB	50M	-38%
23.5GB	60M	-36%
27.4GB	70M	-36%

Chapter 6

Conclusions

6.1 Final Conclusions

This thesis analyzes current multi-core MapReduce frameworks and identifies and describes some of their limitations. In summary, these frameworks are designed to keep all intermediate data in memory. When executing applications with large data input, they fail to make an optimal usage of the cache memory, and when the available memory is too small to store all framework intermediate data, there is a severe performance loss.

This research shows the relevance of the data structure used to store the intermediate data for the global execution performance, and proposes a memory management subsystem to allow the processing of an unlimited amount of data by the use of a disk spilling mechanism. Key/value pairs are copied from memory to disk in a controlled way. Information is serialized and compacted before writing to disk. This mechanism reduces significantly the occurrence of page trashing due to the management of virtual memory performed by the operating system. System memory usage is monitored to launch the spilling process when meeting a certain threshold value. . This dynamic scheme allows the framework to adapt to varying availability of memory, for example due to existence of other concurrent applications sharing the same computer resources.

Experiments with an application making an intensive usage of the MapReduce library show a speedup with respect to the original Metis implementation of up to 3.6 and up to 2.5 in the Map and Reduce phases, respectively.Those results have been published in the international conference that is referenced below:

- **T. Ferreira, A. Espinosa, J.C. Moure, P. Hernández. *An Optimization for MapReduce Frameworks in Multi-core Architectures*, In Proceedings of The International Conference on Computational Science, ICCS**

2013, Barcelona, Spain, June 2013. [16]

Additionally, we have proposed and implemented a way to manage concurrent access to disk of all threads participating in the computation. We set different triggers for each thread so that there is less contention when moving data from memory to disk. The proposed mechanism reduces execution time up to a half on a dual-socket system executing a total of 24 threads. The results have been presented and published in the following international conference:

- **T. Ferreira, A. Espinosa, J.C. Moure, P. Hernández.** ***Optimizing the use of the Hard Disk in MapReduce Frameworks for Multi-core Architectures*, In Proceedings of The 2013 International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2013, Las Vegas, United States of America, July 2013.** [17]

Finally, we have studied the effective use of the memory hierarchy by the data structures of the MapReduce frameworks and proposed a new implementation of partial MapReduce tasks to the input data set. The objective was to make a better use of the cache and to eliminate references to data blocks that are no longer in use. Experiments using 2 applications and two computer systems show that execution time is always improved by at least 1.5x. Speedups range between 1.5 and 5.6 with respect to Metis, and between 1.5 and 4.2 with respect to Phoenix++, depending on the application, the input data size, and the computer system considered. An additional benefit of the partial MapReduce proposal is a reduction of main memory usage of up to 67

We have presented the previous proposal in a paper that has been rejected for publication and, with the help of the useful feedback from the referees we are currently making a final effort to improve the explanations and experimental setup to disseminate the results.

- **T. Ferreira, A. Espinosa, J.C. Moure, P. Hernández.** ***Improving Performance in Memory Hierarchy on MapReduce Frameworks for Multi-Core Architectures*, to be submitted.**

6.2 Future Work and Open Lines

There are two clear continuation lines for the work presented in this document. First, we have to integrate the two main proposals into a robust framework to allow the community to benefit from its use. Second, we have to extend the experimentation to a broader range of applications. In particular, we want to test applications from the bioinformatics field, with strong data-intensive requirements.

A unified shared memory MapReduce framework and a broader set of relevant applications would allow a more in-depth performance analysis to identify the key performance factors and the adequate key system parameters. We want to develop a tuning methodology that set those system parameters to let the framework adapt to the key performance characteristics of each application.

An open line is to extend shared memory MapReduce to cluster-based systems, using a hybrid approach with an in-memory version in each shared memory node and a disk-based MapReduce version across nodes. The memory management mechanisms studied and proposed in this work could be extended by considering remote nodes an additional level of the memory hierarchy.

A different open line is to extend the ideas described in this work to many-core platforms, like GPUs [20][21][32] or the intel Xeon Phi [8] [RefPhi: mirarabajo]. These processors are very complex to program, and can greatly benefit from a programming abstraction that hides the low-level details of thread management, synchronization and intermediate data management.

Bibliography

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS 67 (Spring), pages 483 485, New York, NY, USA, 1967. ACM.

[2] Bahman Bahmani, Kaushik Chakrabarti, and Dong Xin. Fast personalized pagerank on mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD 11, pages 973 984, New York, NY, USA, 2011. ACM.

[3] C. Basaran and K.D. Kang. Grex: An efficient mapreduce framework for graphics processing units. *Journal of Parallel and Distributed Computing*, 2013.

[4] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A map reduce framework for programming graphics processors. In *In Workshop on Software Tools for MultiCore Systems*, 2008.

[5] D. Chavarria-Miranda, Zhenyu Huang, and Yousu Chen. High-performance computing (hpc): Application amp; use in the power grid. In *Power and Energy Society General Meeting, 2012 IEEE*, pages 1 7, 2012.

[6] Linchuan Chen, Xin Huo, and Gagan Agrawal. Accelerating mapreduce on a coupled cpu-gpu architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC 12, pages 25:1 25:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[7] Rong Chen, Haibo Chen, and Binyu Zang. Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT 10, pages 523 534, New York, NY, USA, 2010. ACM.

[8] George Chrysos and Senior Principal Engineer. Intel® xeon phi coprocessor (codename knights corner). In *Proceedings of the 24th Hot Chips Symposium, HC*, 2012.

[9] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engg.*, 11(4):29 41, July 2009.

[10] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI 10, pages 21 21, Berkeley, CA, USA, 2010. USENIX Association.

[11] Jeffrey Dean, Sanjay Ghemawat, and Google Inc. Mapreduce: simplified data processing on large clusters. In OSDI 04: Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation, 2004.

[12] J. J. Dongarra and A. J. van der Steen. High-performance computing systems: Status and outlook. *Acta Numerica*, 21:379 474, 4 2012.

[13] Marwa Elteir, Heshan Lin, Wu chun Feng, and Tom Scogl. Streammr: An optimized mapreduce framework for amd gpus, 2011. 2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS).

[14] Zacharia Fadika, Elif Dede, Madhusudhan Govindaraju, and Lavanya Ramakrishnan. Mariane: Mapreduce implementation adapted for hpc environments. In *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing*, GRID 11, pages 82 89, Washington, DC, USA, 2011. IEEE Computer Society.

[15] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32 38, October 1997.

[16] Tharso Ferreira, Antonio Espinosa, Juan Carlos Moure, and Porfidio Hernndez. An optimization for mapreduce frameworks in multi-core. 2013. In Proceedings of The International Conference on Computational Science, ICCS 2013.

[17] Tharso Ferreira, Antonio Espinosa, Juan Carlos Moure, and Porfidio Hernndez. Optimizing the use of the hard disk in mapreduce frameworks for multi-core architectures. 2013. In Proceedings of The 2013 International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2013.

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29 43, October 2003.

[19] Gero Greiner and Riko Jacob. The efficiency of mapreduce in parallel external memory. In *Proceedings of the 10th Latin American international conference on Theoretical Informatics*, LATIN 12, pages 433 445, Berlin, Heidelberg, 2012. Springer-Verlag.

[20] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. pages 260 269, 2008.

[21] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. Mapcg: writing parallel program portable between cpu and gpu. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT 10, pages 217 226, New York, NY, USA, 2010. ACM.

[22] Chao Jin and Rajkumar Buyya. Mapreduce programming model for .net-based cloud computing. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par 09, pages 417 428, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA 10, pages 938 948, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

[24] George Kovoor. Mr-j: A mapreduce framework for multi-corearchitectures. Master s thesis, University of Manchester, 2009.

[25] Ralf Lammel. Google s mapreduce programming model: Revisited. *Sci. Comput. Program.*, 68(3):208 237, October 2007.

[26] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with mapreduce: A survey. *SIGMOD Rec.*, 40(4):11 20, January 2012.

[27] Heshan Lin, Xiaosong Ma, Jeremy Archuleta, Wu-chun Feng, Mark Gardner, and Zhe Zhang. Moon: Mapreduce on opportunistic environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC 10, pages 95 106, New York, NY, USA, 2010. ACM.

[28] Heshan Lin, Xiaosong Ma, and Wu-Chun Feng. Reliable mapreduce computing on opportunistic resources. *Cluster Computing*, 15(2):145 161, June 2012.

[29] Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, NAACL-Tutorials 09, pages 1 2, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[30] Yandong Mao, Robert Morris, and M. Frans Kaashoek. Optimizing mapreduce for multicore architectures. *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*, 2010.

[31] Rohith K. Menon, Goutham P. Bhat, and Michael C. Schatz. Rapid parallel genome indexing with mapreduce. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce 11, pages 51 58, New York, NY, USA, 2011. ACM.

[32] R. Mokhtari, A. Abbasi, F. Khunjush, and R. Azimi. Soren: Adaptive mapreduce for programmable gpus. In *Fourth Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-2011)*, page 118, 2011.

[33] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of mapreduce pipelines. In *Proc. of ICDE*, pages 681 684. IEEE, 2010.

[34] Gideon Nimako, E. J. Otoo, and Daniel Ohene-Kwofie. Cache-sensitive mapreduce dgemm algorithms for shared memory architectures. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, SAICSIT 12, pages 100 110, New York, NY, USA, 2012. ACM.

[35] Hirotaka Ogawa, Hidemoto Nakada, Ryousei Takano, and Tomohiro Kudoh. Sss: An implementation of key-value store based mapreduce framework. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM 10, pages 754 761, Washington, DC, USA, 2010. IEEE Computer Society.

[36] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD 11, pages 949 960, New York, NY, USA, 2011. ACM.

[37] Anastasios Papagiannis and Dimitrios S. Nikolopoulos. Rearchitecting mapreduce for heterogeneous multicore processors with explicitly managed memories. *Parallel Processing, International Conference on*, 0:121 130, 2010.

[38] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. pages 13 24. In HPCA 07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture, IEEE Computer Society, 2007.

[39] Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. Themis: an i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC 12, pages 13:1 13:14, New York, NY, USA, 2012. ACM.

[40] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. Fpmr: Mapreduce framework on fpga. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA 10, pages 93 102, New York, NY, USA, 2010. ACM.

[41] Jeff A. Stuart and John D. Owens. Multi-gpu mapreduce on gpu clusters. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS 11, pages 1068 1079, Washington, DC, USA, 2011. IEEE Computer Society.

[42] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. pages 9 16, 2011.

[43] Devesh Tiwari and Yan Solihin. Modeling and analyzing key performance factors of shared memory mapreduce. In *IPDPS*, pages 1306 1317, 2012.

[44] Tom White. *Hadoop: The Definitive Guide*. O Reilly Media, 1 edition, June 2009.

[45] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. pages 198 207. IISWC, IEEE, 2009.

[46] Junbo Zhang, Tianrui Li, and Yi Pan. Parallel rough set based knowledge acquisition using mapreduce from big data. In *Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, BigMine 12, pages 20 27, New York, NY, USA, 2012. ACM.