Improving MapReduce Performance on Clusters

Sylvain Gault

N° attribué par la bibliothèque : 2015ENSL0985

École Normale Supérieure de Lyon
Laboratoire de l'Informatique du Parallélisme

THÈSE

en vue d'obtenention du grade de

Docteur de l'Université de Lyon, délivré par l'École Normale Supérieure de Lyon

Discipline : Informatique

au titre de l'École Doctorale Informatique et Mathématiques

présentée et soutenue publiquement le 23 mars 2015 par

M. Sylvain Gault

Improving MapReduce Performance on Clusters

Composition du jury

Directeur de thèse :	Frédéric Desprez *Directeur de Recherche au LIP / Inria*
Co-encadrant de thèse :	Christian Pérez *Directeur de Recherche au LIP / Inria*
Rapporteurs :	Jean-Marc Nicod *Professeur au LIFC / Université de Franche-Comté*
	Vincent Breton *Directeur de Recherche au Laboratoire de Physique Corpusculaire de Clermont-Ferrand / CNRS*
Examinateurs :	Christine Morin *Directeur de Recherche à l'IRISA / Inria*
	Michel Daydé *Professeur à l'IRIT*

Acknowledgments / Remerciements

N'étant pas expansif de nature, ces remerciements seront bref. J'aimerais remercier en premier lieu mon directeur de thèse pour son avis éclairé et son expérience. J'aimerais remercier Christian Pérez pour tout le temps qu'il m'a consacré pour m'avoir guidé et soutenu dans les passages difficiles de cette aventure qu'est une thèse. J'aimerais le remercier aussi en tant que chef d'équipe pour m'avoir accueilli. Je remercie aussi Jean-Marc Nicod et Vicent Breton pour le temps qu'ils ont consacré à lire et à rapporter ce manuscrit. Je les remercie aussi, ainsi que Christine Morin et Michel Daydé, pour avoir accepté de juger mes travaux lors de ma soutenance. Je remercie aussi tous les membres actuels et anciens de l'équipe Avalon pour la très bonne ambiance qui règne au sein de l'équipe.

Résumé

Beaucoup de disciplines scientifiques s'appuient désormais sur l'analyse et la fouille de masses gigantesques de données pour produire de nouveaux résultats. Ces données brutes sont produites à des débits toujours plus élevés par divers types d'instruments tels que les séquenceurs d'ADN en biologie, le Large Hadron Collider (LHC) qui produisait en 2012, 25 pétaoctets par an, ou les grands télescopes tels que le Large Synoptic Survey Telescope (LSST) qui devrait produire 30 téraoctets par nuit. Les scanners haute résolution en imagerie médicale et l'analyse de réseaux sociaux produisent également d'énormes volumes de données. Ce déluge de données soulève de nombreux défis en termes de stockage et de traitement informatique. L'entreprise Google a proposé en 2004 d'utiliser le modèle de calcul MapReduce afin de distribuer les calculs sur de nombreuses machines.

Cette thèse s'intéresse essentiellement à améliorer les performances d'un environnement MapReduce. Pour cela, une conception modulaire et adaptable d'un environnement MapReduce est nécessaire afin de remplacer aisément les briques logicielles nécessaires à l'amélioration des performances. C'est pourquoi une approche à base de composants est étudiée pour concevoir un tel environnement de programmation. Afin d'étudier les performances d'une application MapReduce, il est nécessaire de modéliser la plate-forme, l'application et leurs performances. Ces modèles doivent être à la fois suffisamment précis pour que les algorithmes les utilisant produisent des résultats pertinents, mais aussi suffisamment simple pour être analysés. Un état de l'art des modèles existants est effectué et un nouveau modèle correspondant aux besoins d'optimisation est défini. De manière à optimiser un environnement MapReduce la première approche étudiée est une approche d'optimisation globale qui aboutie à une amélioration du temps de calcul jusqu'à 47%. La deuxième approche se concentre sur la phase de shuffle de MapReduce où tous les nœuds envoient potentiellement des données à tous les autres nœuds. Différents algorithmes sont définis et étudiés dans le cas où le réseau est un goulet d'étranglement pour les transferts de données. Ces algorithmes sont mis à l'épreuve sur la plate-forme expérimentale Grid'5000 et montrent souvent un comportement proche de la borne inférieure alors que l'approche naïve en est éloignée.

Abstract

Nowadays, more and more scientific fields rely on data mining to produce new results. These raw data are produced at an increasing rate by several tools like DNA sequencers in biology, the Large Hadron Collider (LHC) in physics that produced 25 petabytes per year as of 2012, or the Large Synoptic Survey Telescope (LSST) that should produce 30 terabyte of data per night. High-resolution scanners in medical imaging and social networks also produce huge amounts of data. This data deluge raise several challenges in terms of storage and computer processing. The Google company proposed in 2004 to use the MapReduce model in order to distribute the computation across several computers.

This thesis focus mainly on improving the performance of a MapReduce environment. In order to easily replace the software parts needed to improve the performance, designing a modular and adaptable MapReduce environment is necessary. This is why a component based approach is studied in order to design such a programming environment. In order to study the performance of a MapReduce application, modeling the platform, the application and their performance is mandatory. These models should be both precise enough for the algorithms using them to produce meaningful results, but also simple enough to be analyzed. A state of the art of the existing models is done and a new model adapted to the needs is defined. On order to optimise a MapReduce environment, the first studied approach is a global optimization which result in a computation time reduced by up to 47%. The second approach focus on the shuffle phase of MapReduce when all the nodes may send some data to every other node. Several algorithms are defined and studied when the network is the bottleneck of the data transfers. These algorithms are tested on the Grid'5000 experiment platform and usually show a behavior close to the lower bound while the trivial approach is far from it.

Contents

Chapter 1

Introduction

Human is constantly seeking for knowledge. Technology development has always helped in that matter. Today, information technology can, more than ever, help building new knowledge by allowing to collect store and process more and more data. When processed with the right tool, the right algorithm, those data can tell a lot of useful information.

In the past decades, the amount of data produced by scientific applications has never stopped growing. Scientific instruments still make the rate of data production to continuously grow as well. DNA sequencers in biology, the Large Hadron Collider (LHC), the Large Synoptic Survey Telescope (LSST) in physics, high-resolution scanners in medical imaging, and digitalisation of archives in Humanities are few examples of scientific tools producing data at a high rate. Sensor Networks become quite common, while web indexing and social network analysis is a hot research topic.

With new orders of magnitude in data production come new challenges related to storage and computation. Depending on the kind of processing needed for these Big Data, several approaches can be considered. The users of the LHC are gathered into large international collaborations to analyze the data and perform long lasting simulation based on these data. Another approach is to rely on an easily scalable programming model. Google proposed to use MapReduce [1] for this purpose in order to handle the web indexing problems in its own data-centers. This paradigm, inspired by functional programming, distributes the computation on many nodes that can access the whole data through a shared file system. MapReduce system users usually only write a MapReduce application by providing a *map* and a *reduce* function.

1.1 Context

The context of Big Data raises several challenges. The first one is the storage of large amounts of data. Since large amount of data needs several physical support, they need to be replicated to face the failure rate that become non-negligible. All those data have to be managed to allow an easy and consistent usage. The second challenge is about moving the data. Large amount of data cannot be moved freely across a usual network infrastructure. And third, more data means that processing them takes more and more time.

Although the challenges raised by those amount of data are very real and practical, the notion of Big Data is not well defined. `META` Group (now Gartner)[1] defined Big Data [2] as a three-fold notion summarized in three words: *Volume*, *Variety*, and *Velocity*.

Volume The notion of volume in the context of Big Data is not directly about size of the data, it is mainly about the value of the data that lie in its size. The larger the data, the more

[1] Garner Inc. is an American information technology research and advisory company.

valuable it is.

Variety Unlike large relational databases that has been existing for decades, the data collected here may not be structured like text files and may be in various formats. This raise new challenges for the users and software developers.

Velocity The third key point of Big Data is the rate at which the data is produced and processed.

One of the solutions to ease the handling of those challenges and leverage the related opportunities is to use a framework like MapReduce to process those data. The MapReduce programming model is composed of 3 phases, *map*, *shuffle*, and *reduce* where the user provide a *map* function to transform one data chunk into another, while the user-provided *reduce* function merges several intermediate elements of data into one. Although this programming model only allow a limited set of computations to be done, it covers most Big Data use-cases, sometimes by chaining several MapReduce jobs. Additional tools may help create a sequence of MapReduce jobs for a particular task.

1.2 Challenges

Because of its nice properties, MapReduce has been largely adopted and several implementations has been designed and developed. However, several questions and challenges remain regarding the scalability and performance of the frameworks. MapReduce is designed to run on any kind of hardware and should therefore also run on low cost hardware. This thesis focus particularly on the performance challenges in the case of low cost network hardware and try to bring an answer to the following questions.

- How to make the execution of a MapReduce job as short as possible?
- How to deal with network contention?

1.3 Contributions

This thesis present a total of four contributions in Chapters 4 to 7. The first two contributions are the basis on which rely the following two contributions.

MapReduce with components. The first contribution is the extension of the design of a MapReduce framework named HoMR based on a software component model designed by Julien Bigot [3]. There are 3 main goals in the design of this framework. The first one is to build it mostly from reusable pieces of code. The second one is to make it easy to build another application with a similar computing structure. And last but not least, the third aim is to make it easy to change some parts of the framework so that other algorithms can be experimented in the context of this thesis.

Platform and performance modeling. The second contribution is about modeling the platform that runs a MapReduce framework, and the MapReduce application and its performance in order to be able to predict the time taken by each step. The models proposed in this thesis intend to be neither too coarse nor too detailed in order to be analytically tractable, but still accurate enough to be applied to actual applications.

Global optimization. Third, the problem of optimizing a MapReduce application as a whole is studied. This thesis proposes both a partitioning algorithm and a transfer scheduling algorithm that improve a previous work of Berlińska and Drozdowski [4] both in terms of computation time and in resulting schedule length.

Shuffle optimization. Finally, this thesis proposes several algorithms to handle the shuffle phase in a way that avoids network contention. Of those five algorithms, most show a behavior better than a naive approach and two of them show a behavior close to the theoretical lower bound.

1.4 Document Organisation

This document is divided in 6 main chapters. Chapter 2 presents an overview of the existing computing platforms both on the hardware and software levels and focus on the actual platforms relevant to this thesis. In Chapter 3, the MapReduce paradigm is presented as well as its implementations and variations. Then Chapter 4 presents the software component models used to design HoMR, as well as HoMR itself. Chapter 5 presents some existing work for modeling some distributed platforms and MapReduce applications with their performance and a proposition of some alternative models used in the next chapters. In Chapter 6 the problem of a global optimization is studied, and the focus is given to the *shuffle* phase in Chapter 7. Finally, Chapter 8 concludes this document and highlight some perspectives.

1.5 Publications

1.5.1 Publications in International Journal

[5] Gabriel Antoniu, Julien Bigot, Christophe Blanchet, Luc Bougé, François Briant, Franck Cappello, Alexandru Costan, Frédéric Desprez, Gilles Fedak, Sylvain Gault, Kate Keahey, Bogdan Nicolae, Christian Pérez, Anthony Simonet, Frédéric Suter, Bing Tang, and Raphael Terreux. "Scalable Data Management for Map-Reduce-based Data-Intensive Applications: a View for Cloud and Hybrid Infrastructures". In: *International Journal of Cloud Computing (IJCC)* 2.2 (2013), pp. 150–170

1.5.2 Publications in International Conference

[6] Gabriel Antoniu, Julien Bigot, Christophe Blanchet, Luc Bougé, François Briant, Franck Cappello, Alexandru Costan, Frédéric Desprez, Gilles Fedak, Sylvain Gault, Kate Keahey, Bogdan Nicolae, Christian Pérez, Anthony Simonet, Frédéric Suter, Bing Tang, and Raphael Terreux. "Towards Scalable Data Management for Map-Reduce-based Data-Intensive Applications on Cloud and Hybrid Infrastructures". In: *1st International IBM Cloud Academy Conference - ICA CON 2012*. Research Triangle Park, North Carolina, États-Unis, 2012. URL: http://hal.inria.fr/hal-00684866

1.5.3 Publications in International Workshop

[7] Sylvain Gault and Christian Perez. "Dynamic Scheduling of MapReduce Shuffle under Bandwidth Constraints". In: *Europar 2014 Workshop Proceedings, BigDataCloud*. Porto, Portugal, 2014

1.5.4 Publications in National Conference

[8] Sylvain Gault. "Ordonnancement Dynamique des Transferts dans MapReduce sous Contrainte de Bande Passante". In: *ComPAS'13 / RenPar'21 - 21es Rencontres francophones du Parallélisme*. 2013

1.5.5 Research Reports

[9] Frédéric Desprez, Sylvain Gault, and Frédéric Suter. *Scheduling/Data Management Heuristics*. Tech. rep. Deliverable D3.1 of MapReduce ANR project. URL: http://hal.inria.fr/hal-00759546

[10] Sylvain Gault and Frédéric Desprez. *Dynamic Scheduling of MapReduce Shuffle under Bandwidth Constraints*. Tech. rep. Inria/RR-8574

Chapter 2

Platforms

This chapter presents the computing platform most used today and is divided in 4 sections. Section 2.1 presents the hardware architecture of the computing platforms from the inner parallelism of a single node to the node aggregation of several nodes on a wide area network and to the storage. Section 2.2 presents the usual software tools that help making software exploit these architectures, while Section 2.3 focus on the management software for distributed systems. And finally, Section 2.4 concludes the chapter.

2.1 Architecture

2.1.1 Node Hardware

The most basic hardware that can run a program is a single computer. In its simplest form, it can be modeled as a Von Neumann architecture [11]. In the Von Neumann model, a computer consists in one Central Processing Unit (CPU), one memory to store the data and the programs, and some input / output devices attached to the CPU. The CPU is itself composed of a Control Unit that decides at each step what is the next instruction to execute, and an Arithmetical and Logical Unit (ALU) that actually performs the instructions.

In reality, a simple computer is very close to this model. The memory used is a Random Access Memory (RAM). The input and output devices may be any kind of usual peripheral, including a Hard Disk Drive (HDD) or Solid-State Drive (SSD) for storage.

The common operation of a computer is that the CPU reads its program from the main memory and executes it. The program may instruct to read or write data from / to the main memory, or from / to the mass storage device. From this, any computation is possible.

The throughput at which a given hardware can receive or send data vary a lot. As of 2013, a typical magnetic HDD can achieve a throughput of 100MB/s and exhibits a latency of roughly ten millisecond. The memory can provide more ten gigabytes per second with a latency in the order of approximately ten nanoseconds or less. While the processor could process data at a rate of several tens of gigabytes per second. For this discrepancy not to impair the performance, several strategies have been developed. There are two main techniques used to try to hide the slowness of those hardware: multitasking and caching.

Caching consist in using a *fast memory* put close to the hardware reading the data, that store a part of the data that are slow to access, so that subsequent accesses can hit[1] the cache and not access the slow storage. Typically, for the HDD access, the operating system allocates

[1]Successfully finding a data in a cache is said to be a *cache hit*, while needing to read it from the slow storage is said to be a *cache miss*.

a part of the main memory as a cache. While the processor have a piece of hardware memory that cache the main memory accesses.

Multitasking is the act of running several tasks during the same period of time. It cannot speedup the access to data, but it can hide the slowness of the data storage by running another task while the data is being transfered. To hide the hard disks slowness, the operating system can chose to execute another task on the processor. To overcome the slowness of the RAM, some processors can execute another thread if they support simultaneous multithreading (e.g. Intel's hyperthreading) or execute the instructions out-of-order. The latter can indeed be seen as a form of multitasking with a task being an instruction.

2.1.1.1 Multi-Processor

The single simple node processing power is roughly characterized by the frequency of the processor. Thus, at a given time, for a fixed state-of-the-art hardware, one way to increase the computing power is by using several computing units at the same time. Either by using several full CPUs, or by putting several processing units inside the same processor.

Symmetric Multiprocessing The Symmetric Multiprocessing (SMP) technology consists in using several processors in the same node. Every processor run independently and may execute a different program. They do however share all the rest of the computer hardware, especially the bus connecting the RAM. Since the processors run faster than the RAM, having several processors reading and writing the same memory only stress this phenomenon. The memory can indeed serve only one request for reading or writing data at a time. Thus, the more processors use the memory, the higher the risk of contention[2]. Moreover, all the processors have to communicate in order to keep their cache consistent. This limits even more the scalability. Figure 2.2 shows an architecture example of an SMP system.

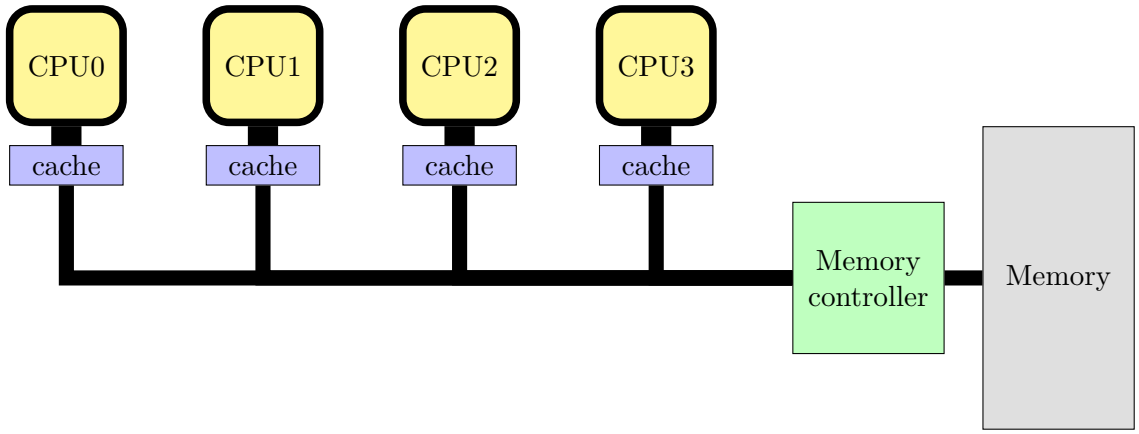


Figure 2.1: Schema of a symmetric multiprocessing architecture with 4 processors.

Non-Uniform Memory Access A way to alleviate the aforementioned memory bottleneck, is by splitting the main memory into several banks, each processor having one bank faster to access than the others. This creates what are called *NUMA nodes*. Figure 2.2 shows an example of NUMA architecture with 2 NUMA nodes and 4 processors per node. Each node roughly consists in one memory bank and one or several processors. Each processors inside a NUMA node can access all the memory of this node with a constant latency. Each node can

[2]Contention is a conflict for accessing a shared resource.

be seen as an SMP architecture. In order to communicate, these nodes are interconnected with a network. A processor can thus access the memory of the other nodes as well as the memory of its own node. However, accessing a memory location that belongs to another NUMA node is slower by a factor called *NUMA factor*.

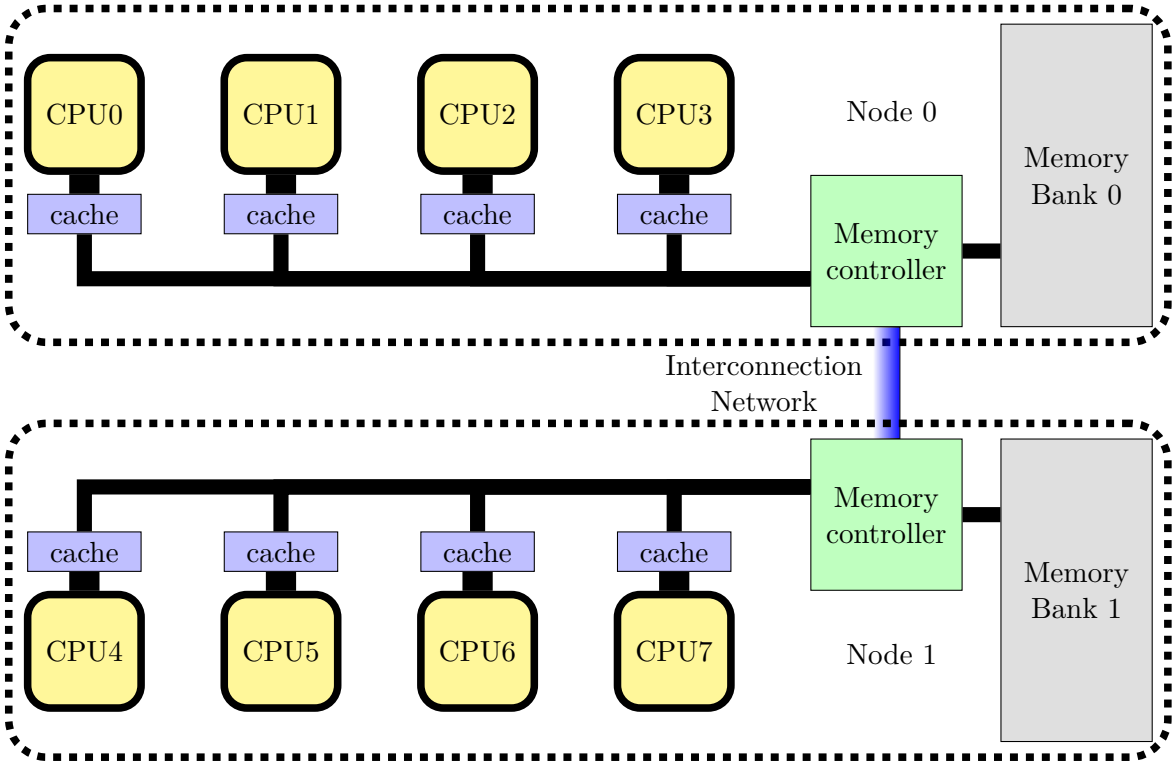


Figure 2.2: Schema of a NUMA architecture with 2 nodes of 4 processors.

Unlike SMP architecture, NUMA architecture need the software to be NUMA-aware in order not to suffer from the NUMA factor. The optimizations to perform are quite low level since the software must be aware of the of the location of the memory used.

2.1.1.2 Multicore Processors

Multicore processors are really close to SMP; they notably share the memory bank and the bus to the memory. The main difference is that instead of having several full processors, the processing units are located on the same chip. Moreover, they can share one or more levels of cache. From the operating system point of view, cores act like distinct processors. The multicore approach is orthogonal to the SMP / NUMA approach and both can exist at the same time inside a computer: a NUMA system can be made of multicore processors.

One step further into the integration of multiple processors is Simultaneous Multi-Threading (SMT). With SMT each core is composed of several hardware threads, each one has the minimal dedicated hardware to allow the execution of several threads at the same time. Thus, when the threads have to perform instructions that make use of distinct subsets of the elements of the processors, they can be executed at the same time. SMT also come in use when one execution thread wait for the memory to provide data, the other thread can continue its execution and the processor usage remain high. This architecture is also seen from the operating system as several processors.

2.1.1.3 GPU and Accelerators

Graphics Processing Units (GPU) are the processors embedded in video boards for accelerating the graphical rendering. They are massively parallel processors composed of hundreds of cores that can execute the same instructions at the very same time on distinct piece of data. This execution model is called SIMD (Single Instruction Multiple Data). As they are massively manufactured, their price is very low related to the computing power they provide. It is then not surprising that they are used to perform compute-intensive task outside of graphics rendering.

The new usage of GPUs, called GPGPU for *General Purpose Computing on Graphics Processing Units* has led to hardware modifications from the chip manufacturers. Especially, common sized integer have been added, as well as support for IEEE 754 [12] floating point numbers. Moreover, some GPU board are produced for the special purpose of performing non-graphics computations. Some graphic board do not even include an output port, making them unable to produce an image. The NVIDIA Tesla K40 boards [94] are one of a kind.

2.1.2 Network

Before connecting several computers together, the network need to be defined. A computer network is composed of several nodes which all have one or several Network Interface Cards (NIC) and some routing device connecting them. Over the years, several technologies have been employed to make the hardware layer of the network. The most common among consumers is Ethernet which has been improved to reach a theoretic bitrate of 1 Gbps in most new devices. The equipments supporting the 10 Gbps version of Ethernet are still expensive for the consumer market but are found in more and more new professional equipments.

On the high-end of network requirement in terms of bitrate and latency, there are some technologies like InfiniBand that can provide a bitrate higher than 10 Gbps per link and a latency less than a microsecond. Moreover, InfiniBand allows links to be aggregated, and can thus provide a bandwidth of several tens to a few hundreds of Gbps. Additionally This kind of low latency NIC usually support Remote Direct Memory Access (RDMA) meaning that a process on a node can directly access the memory of another process located on another node without invoking the operating system on the remote node. This feature makes the whole system close to a NUMA system.

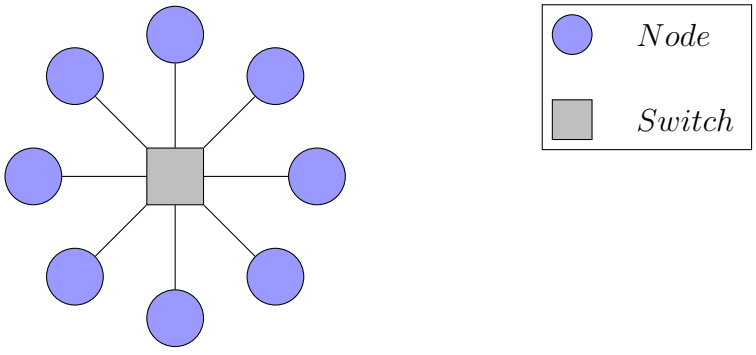


Figure 2.3: Star network topology.

The nodes and routing devices in a network form a graph. However, in order to make the performance predictable and, hopefully, good, the nodes are not randomly interconnected. The most basic network topology is the star as show in Figure 2.3 where all the compute nodes are connected to a single routing device. The obvious limitation of this approach is that it requires the routing device to have as many ports as there are nodes to interconnect. The

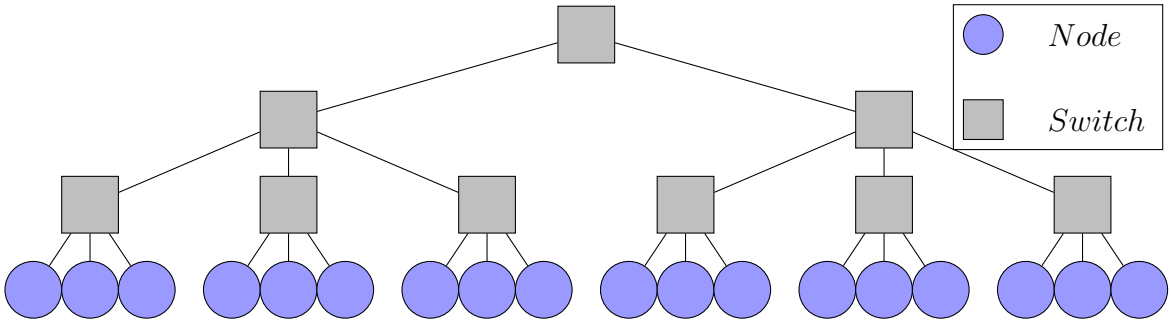


Figure 2.4: Tree network topology.

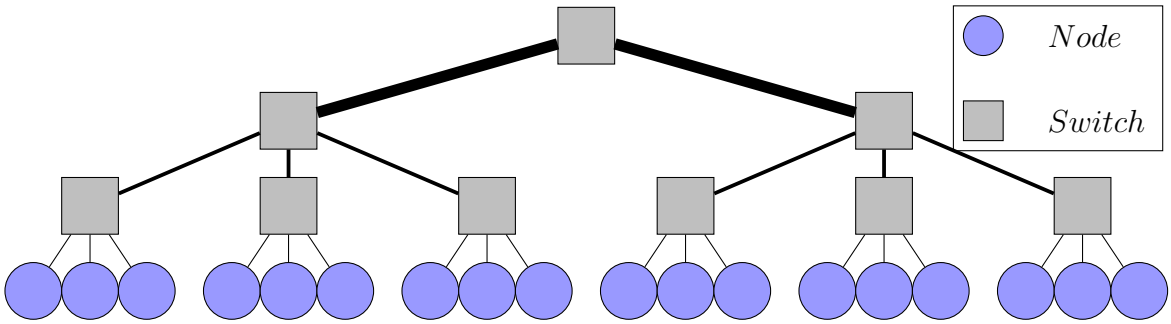


Figure 2.5: Fat tree network topology.

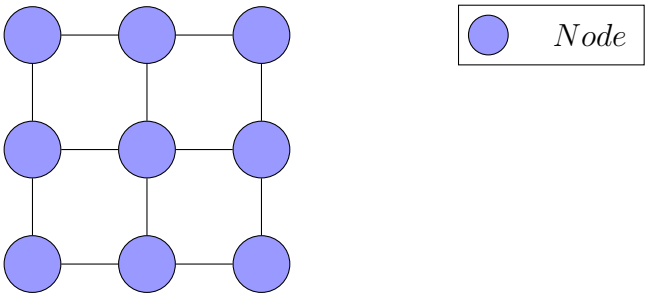


Figure 2.6: 3×3 grid network topology.

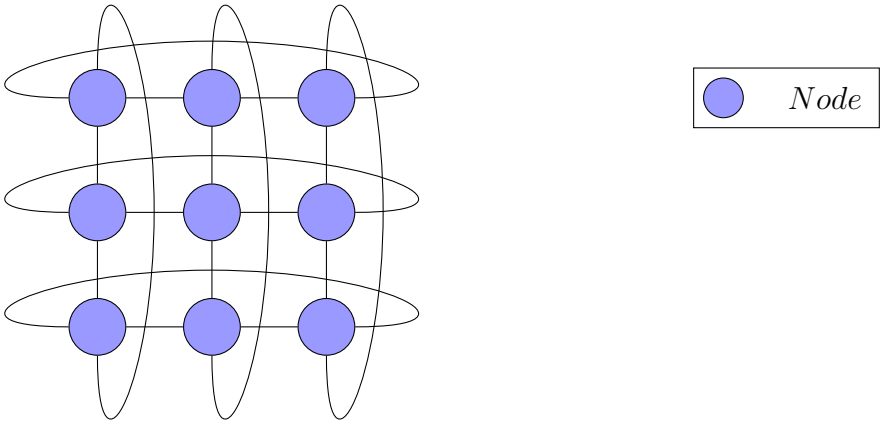


Figure 2.7: Torus network topology.

classical workaround is to make a tree as shown in Figure 2.4. This allows to interconnect as many nodes as needed, but the root node may become the bottleneck if several nodes try to communicate from one end to another of the tree. This issue, in turn, can be worked around by having links with a larger bandwidth as they are close to the center of the network, which is called a fat-tree [13]. Figure 2.5 shows an example of a fat tree network. Another solution is to change completely the network topology. Figure 2.6 show an grid network topology where each node is connected to its direct neighbors, either in 2D or in 3D. This allows a high global bandwidth and a is more resilient to failure somewhere in the network. This grid can, be closed to form a torus topology and thus eliminate the special cases on the sides where nodes have less than 4 or 6 neighbors and divide by 2 the maximal and average number of hops connecting a node to another. Figure 2.7 shows an example of network topology forming a 2D torus.

2.1.3 Node Aggregation

Another way to improve the computational performance is to aggregate nodes instead of building more complex processing units. The interconnected computers can be seen as three main groups: Clusters, clusters of clusters, and grids.

2.1.3.1 Cluster

In a cluster, all the nodes are usually homogeneous[3] and are interconnected within a Local Area Network (LAN) or a System Area Network and can be viewed as a single system. The nodes of a cluster are usually located in the same room and managed by the same entity. Increasing the computing power of a cluster is as simple as adding some nodes to the network. This makes the cluster a very scalable solution. Clusters can be made from any kind of nodes, from high-end computers to consumer computers.

The network hardware usually rely on classical Gigabit Ethernet technology. This network can however coexists with a low-latency high-bandwidth network technology such as InfiniBand or Myrinet. Fast network usually include features like Remote Direct Memory Access (RDMA).

[3]Consist of identical hardware.

RDMA allows a node to access directly the memory of another node. This leads to an easy implementation of a Distributed Shared Memory system.

The main use of clusters range from service duplication for high availability or load balancing to High Performance Computing (HPC), including the data-intensive computing. Clusters are thus found in many organizations. However, despite its easy scalability, the applications running on a cluster need to be aware of the network topology because the communication between the nodes is much higher than the communication between the processors of a large single computer.

Regarding hardware, there is not much difference between a cluster and a modern supercomputer. Supercomputers are usually composed of several nodes that contains high-end processors and other elements. Those nodes are interconnected with a high-performance network which may use proprietary technology. All those nodes act as single computer thanks to the operating system. This kind of cluster operating system is presented in Section 2.3.1.

2.1.3.2 Cluster of Clusters

Following this logic of aggregating the computing resources, the cluster of clusters comes naturally. In this kind of platforms, several clusters are connected through a fast network. All those clusters may belong to a single organization like a laboratory or belong to several entities collaborating.

This kind of platform raise new challenges related to the strong heterogeneity of the computing resources that may exists. The heterogeneity lies on several levels. First on the node level, where, despite each cluster being usually homogeneous, all clusters may have very different hardware brought several years apart. Second on the network level. The network interconnecting a single cluster may not have the same latency and bandwidth as the network connecting the clusters together. Moreover, not all the pairs of clusters have a dedicated link connecting them. Thus, not all the cluster-to-cluster communication have the same performance characteristics.

2.1.3.3 Data-Centers

Data centers are physical places where computing systems are housed. It usually offers all the facilities to run safely a large number of servers, including power supplies, cooling, and so on. Regarding the interconnection of the equipment, although everything is possible, the most common network topology between the computing nodes is a cluster of clusters. This allows to easily manage a large scale infrastructure.

The persistent storage can be handled by a SAN (Storage Area Network) or a NAS (Network Attached Storage). From the computing node's point of view, in a SAN, the storage device are accessed at a block level as if they were physically attached to the node. This, even thought the block storage may be virtual. The network link between the storage and the compute node accessing the block storage can be based on the Fibre Channel technology for a high throughput and a low latency. The NAS on the opposite, exposes to the user an interface allowing to retrieve some files directly. The storage techniques are presented in more details in Section 2.1.4.

2.1.3.4 Clouds

The cloud is a service-oriented infrastructure that hides most of the hardware to the user, thus allowing the cloud provider some freedom of implementation. As such, the most important part of the cloud infrastructures is at the software level and is described in more details in Section 2.3.3.

On the hardware level, depending on the size of the instrastructure, a cloud can be based on a cluster, a cluster of clusters, or even on several data-centers for the largest clouds. Since the

details are hidden to the user, virtualization plays an important role in the cloud. Virtualization is greately helped by the specialized CPU instructions for virtualization in the x86 processors in terms of performance.

2.1.3.5 Grid

Another kind of node aggregation is the computing Grid and can be seen as an extension of the cluster of clusters where the clusters are connected through a Wide Area Network (WAN). The nodes of a Grid can be of any kind, from the desktop computers to high end server. The computing resources of a Grid usually belong to several organizations or institutions and are geographically distributed. The WAN usually have a much larger latency and a lower throughput than a LAN that connects a cluster together. The WAN may even transit through the internet.

As the computing resources are managed by distinct entities, they need an agreement to collaborate and offer a consistent service. In order to do this, the organizations owning the computers are grouped into a *Virtual Organization*. The virtual organization define some rules and conditions for resource-sharing. The Virtual Organization is then all the user care about.

The idea of the computing Grid is loosely based on that of the electrical Grid. A user could benefit from computing power on demand without owning his own resources. And accessing computing power should be as simple as plugging a device to the mains.

Following this, Ian Foster proposed a three-point checklist to define a computer Grid. A computer Grid is a system that:

- coordinates resources that are not subject to centralized control
- using standard, open, general-purpose protocols and interfaces
- to deliver nontrivial qualities of services.

A special kind of Grid is one that consists mainly of Desktop computers. The computation usually only occur when the computer would be otherwise idle. This provide the platform a few unusual properties. Desktop Grids are extremely heterogeneous and have a high host churn rate implying a very different management approach. The nodes in the system are also untrusted, which mean that every computation has to be done several times before a result is trusted. However, a planetary Desktop Grid may provide a very high performance. The BOINC [14] infrastructure, for instance, reported 9.2 PFLOPS [95] in March 2013, which would make it the 5^{th} most powerful computer in the Top 500 list [96]. Those special properties of the Desktop Grids make them apart from the other computing platforms and are thus not addressed in this document.

2.1.4 Storage

When processing a large amount of data, the input data and the results have to be stored. Even more, the performance of data intensive applications is usually heavily dependent on both the hardware and software infrastructures used for storage.

2.1.4.1 Classical Storage

The main and easiest way to store data is with a simple magnetic hard disk drive attached directly to the node. Although solid-state drives become more and more widespread and allow a very high bandwidth, their price per gigabyte is still very high. This kind of storage system is sometimes called Direct-Attached Storage (DAS).

On those disks, data is stored using a classical hierarchical file system like ext3 or ReiserFS. This file system is usually implemented by a driver of the operating system as being a sensitive piece for security, performance, and reliability.

This kind of storage allow fast read and write operations since everything is done locally. It is also simple to use since it comes with any operating system. However, there is no easy way to exchange data among several nodes. Thus, in a cluster or a grid, the data should either be already located on the node they will be processed or the transfers should be handled by the application itself. In case the application decides to replicate some data, it has to manage itself the consistency of all the replicas, including the tracking of the replicas that can be evicted or not.

Although this kind of storage system is limited and has several issues, it is the basic layer of storage most of the other storage systems rely on.

One way to improve reliability and performance of the storage is the Redundant Arrays of Inexpensive Disks (RAID). RAIDs aggregate several storage devices and make them appear as a single one to the operating system. The operating system can then use its usual driver to read and write to the RAID device without knowing it is composed of several disks. There exists several *levels* of RAID which bring either more storage space, reliability or performance or a both. The most common levels of RAID are RAID 0, RAID 1 and RAID 5.

The RAID 0 setup scatters the blocks of data on all the disks in a Round Robin fashion. This provide more storage space a better read and write performance while the reliability is decreased since the failure of one disk make all the data unusable. RAID 1, on the opposite, copies the blocks of data on all the disks, thus, all the disks contains the exact same data. This provides better performance for reading the data and a better fault tolerance, but decreases the write performance and provide no more storage space. RAID 5 is a trade-off between RAID 0 and RAID 1. In a RAID 5 setup every time a block is written, one of the disk receives redundant data, so that the redundant data can be used to recover from a failed disk.

2.1.4.2 Centralized Network Storage

A second way to store data is with a centralized network storage usually called Network-Attached Storage (NAS). In this case, a node has one or several disks attached and allow other nodes to read and write files through a standard interface and serve them through the network. A widely known implementation of this scheme is the Network File System (NFS). NFS is mainly a protocol to access files through the network. Although the server is free to implement any way to access to the real data to provide through the network, most implementations simply rely on the data being directly accessible on the server.

One of the main advantage of this kind of architecture is that is makes it easy to share data among several compute nodes. Since the data is stored on one server, it is easily kept consistent. However, one of the major drawbacks that stems from the centralized nature of this model is that it does not scale . The concurrent read and write operations have to be supported by only one server. It also have the disadvantage of not being resilient to failures by itself. If the storage server crashes, all the data are unavailable. Thought, this last point may be mitigated in some setups with a fail-over and data replication.

2.1.4.3 Parallel and Distributed Storage

In order to overcome the limitations of a centralized network storage, the data can be distributed across several storage nodes. Using several nodes allows to access several files at the same time without contention. It also allow a better throughput for reading the same file when there are replicas on several nodes. A distributed file system is usually built to scale better than a

centralized storage. Also, in theory, distributed storage systems can avoid the single point of failure problem.

The distributed storage system often have a single entry point node that receives all the requests to read or write data. As its role is central and critical, its job must be kept to the minimum. This master node usually only allows a global collaboration among all the nodes involved in the storage system, and may store the meta-data (file name, file size, access attributes, ...). Thus, when a request for reading or writing a file is received, the client is redirected to another node that is going to actually process its request. However, if the meta-data may be stored on the master node, the real data are always stored on some other nodes, and may be replicated.

The drawback of a distributed storage is that to achieve the best performance, the application should take locality into account. Indeed, even thought the default behavior of the storage system might be quite good, it is usually preferable to read or write data from / to the closest node of the system storage than from / to a node with a high network cost.

An example of such storage system is the **Hadoop Distributed File System** (HDFS) [15] that is developed by Apache in the Hadoop [97] project that implements a MapReduce framework. It is designed to handle data in the order of terabytes. HDFS calls its central node the *NameNode* which manages de *DataNode*s and stores the meta-data. As a single point of failure, the meta-data stored on the *NameNode* are periodically replicated on a *SecondaryNameNode* so that in case of crash, not all the metadata are lost. The *DataNode*s of HDFS handle replication themselves, as well as rebalancing to spread the load across the nodes. As it has been developed to be used with the rest of the Hadoop project, it is optimized for BigData, which means it offers good read performance while the writing performance are considered of less importance. It is also designed to collaborate with the *JobTracker*s of Hadoop MapReduce so that the scheduler can take data locality into account.

BlobSeer [16] is another example of distributed storage system. It is not a file system per se as the notion of file name is missing. BlobSeer can be seen as a versioned storage system designed to handle data in the order of terabytes. However some basic file systems implementations on top of BlobSeer exists. As its name suggests, it stores Binary Large OBjects (BLOBs), which are nothing else than unstructured data. The master node of BlobSeer is called the *version manager* which, upon request, provide the id of the lastest version of a blob, or assign a unique id to a new version of a blob. The data a stored by the *metadata providers* while the data are kept by the *data providers*. BlobSeer is claimed to offer good performance for any combination of read or write operations under heavy concurrency.

Parallel Virtual File System (PVFS) is another example of distributed file system. The main difference with the previously presented file systems is that its API is compliant to POSIX, and thus usable transparently by any program. Its architecture is comparable to that of HDFS. It has several *IO daemons* that serve the data on demand, and a *manager daemon* that handles the metadata only.

2.2 Software

The hardware as described in Section 2.1.1 and Section 2.1.3 needs, of course, at least an operating system. It, however, would be a lot of work to write any non-trivial using only the operating system features. Some higher-level software are needed. That can be runtime libraries, frameworks, compilers or other tools. However, in the context of big data, those software frameworks must target performance on a large scale in addition to providing usability.

2.2.1 Component Models

Software component-based design is not a software per-se, but is still a useful tool to design applications. The applications to run on distributed platforms tend to be more and more complex and the components models offer an abstraction to manage the complexity while still keeping the software easily adaptable to other needs.

In the component models [17], a component is a piece of software that address a specific set of concerns. They are kind of modules that can, in some models, be composed of submodules. The components are, of course, connected together to allow them to communicate through method/procedure calls. In a component model, an application is fully defined as a set of components and connections. Chapter 4 go more into details about the components models and especially describe two models L^2C [18] and HLCM [19]. Note that all the software solutions hereafter can be encapsulated in components.

2.2.2 Message Passing Interface

One of the concerns with distributed infrastructures, is the network. If not properly used, it can become a bottleneck and impair the performance. The Message Passing Interface (MPI) [20] provide an API for sending and receiving messages from one process to another. Those process can either be located on the same computer or on the same network. It has been designed for efficiency, and usually tries to exploit the computing resources the most efficiently. Its more widespread implementations are MPICH and Open MPI.

MPI can be used either inside a node or across several nodes. Most implementations have optimizations to not use the network stack when it is not needed. It thus can be efficiently used in shared memory systems as well as in fast LAN, and WAN networks. The MPI API also provides some functions to perform collective operations like a all-to-all where every node has some distinct data to send to every other node, or a reduction where a simple operation has to be applied to an intermediate result and a data element to produce a new intermediate result. These collective operations can also benefit from advanced algorithms in an MPI implementation.

However, the performance comes at the cost of a programming effort. Indeed, MPI API is quite low level and only allow the program to send data. Any higher level feature, like calling a remote procedure and waiting for the result, has to be reimplemented.

2.2.3 CORBA

Common Object Request Broker Architecture (CORBA) [21] is a standard for inter-process communication. It provide a standard way to call a method on a remote object, and thus is an object-oriented platform-independent Remote Procedure Call (RPC) specification. CORBA specify several levels of the communications, from the network to the programming to some more general concepts of software architecture.

In the CORBA architecture, the ORB (Object Request Broker) or broker is a software component that connects the processes together. It handle the marshalling[4] and unmarshalling[5] as well as all the networking aspects. The ORB also handle the synchronousness of the method calls, which means that it build the request, send it *on the wires* and wait (or not) for the result.

On the network part, CORBA defines the General Inter-ORB Protocol (GIOP) which is an abstract protocol. This protocol is used for every communication between the processes. Its most common instanciation is Internet Inter-ORB Protocol (IIOP) which is a specification of

[4]Marshalling is the process of translating an object into a string of bytes while
[5]unmarshalling is the process of translating a string of bytes into an object.

GIOP designed to be used on top of TCP/IP[6] protocols. These specifications should allow any ORB implementation to communicate.

In the code, all the programmer has to manipulate are remote object references. The methods can be called on these references as if the object would be local. The ORB handles itself the marshalling and unmarshalling process, but it does not know about the classes and their methods. There is thus a layer of code on top of the ORB that allow the client process to perform a simple method call that is converted to calls to the ORB. On the server side there is also a layer of code that convert the methods calls from the ORB to calls to the implementation object method.

These two additional layers of codes are generated by a CORBA tool from an interface written with the Interface Description Language (IDL). An interface mostly consists in abstract data types and methods with their parameters and return type. The IDL language is part of the CORBA specification, as well as the mapping between the IDL and several languages. Interfaces written in IDL are thus portable among the CORBA implementations. The IDL compilers read the interface description and generate two pieces of code: the stub and the skeleton. The stub contains the code that makes the glue between the ORB API and the user code that perform a simple method call on a remote object reference. The skeleton is the symmetrical on the server side, it contains the code that makes the glue between the server-side ORB and the implementation object.

2.2.4 Processes and Threads

There are two main ways to make use of all the processors and cores inside a compute node: processes and threads. A process is the dynamic view of a program being executed. It has its own virtual memory space, its own data and has one or several execution threads. A thread (also called light-weight process) consists in a sequential code to run and some local memory. All the threads inside a process share the same memory space and can thus communicate through explicitly shared memory.

Although the memory isolation of the processes bring some safety, they are heavier to create and do not intrinsically share memory. Threads are thus usually the preferred way to run a parallel program on a multiprocessor / multi-core node.

POSIX threads (Pthread) is a standard that defines an API to create and synchronize execution threads. This specification is implemented by most UNIX-like operating systems, and some implementations are known for other operating systems like Microsoft Windows.

The Pthread interface is quite low level. All the thread creation and synchronization has to be explicit. Therefore, some tools try to provide a higher level abstraction of the threads by providing an interface for the common usage patterns. For instance, OpenMP extends the C language to include some `pragma` directives that instruct the compiler what section of code should be executed in parallel. Its method for parallelizing a code uses the *fork-and-join* model, therefore OpenMP is more adapted to the applications that fit into this model.

Another effort in simplifying thread usages is the BOOST threading library. It exposes an object-oriented API to be used by the programmer. In addition to the classical low-level thread management features, it provides some higher level abstraction and operations such as thread groups and Resource Acquisition Is Initialization (RAII) locks.

2.2.5 Graphics Processing Units

Using GPUs to compute something else than computer graphics is called General-Purpose computing on Graphics Processing Units (GPGPU). The GPUs have a great potential compu-

[6] Transmission Control Protocol / Internet Protocol

tational power. But their architecture induce a way to program them that is really different from a classical processor. GPUs are massively parallel processors counting several hundreds of cores which usually executes the individual instructions in a lockstep. Meaning that a given instruction is executed on several data at once. This induces that the code is written specifically for a GPU and using the GPU to its full potential need some specific low-level optimizations. Those optimizations can either be supported by a specific library, like BLAS[7] [22], which has a very specialized task, or by a language that includes enough meta-information for the compiler to generate code optimized for the target GPU.

In the area of GPGPU, the two main manufacturers are NVIDIA and ATI. Both developed their own solution for GPGPU, CUDA for NVIDIA and ATI Stream for ATI. As expected, both are incompatible. However, OpenCL is a third approach that can be compiled for NVIDIA or ATI GPUs as well as for CPUs, Field Programmable Gate Arrays (FPGA) and other kind of processors. But, even though OpenCL **can** be compiled to many type of processors, the code may have to be rewritten to achieve best performance on every platform.

Another orientation of third part languages is to augment C (or Fortran) with some *pragma* directives to add the meta-informations needed to compile the code for a GPU. StarSs [23], HMPP [24] and OpenACC [25] basically take the same approach as OpenMP and adapt it to target several processing units. For instance, in OpenACC, its directives instruct the compiler that some portions of the code are to be run on an accelerator, or that a given data have to be moved to the accelerator's memory. Even though the compiler could probably transform the code to obtain good performance on every accelerator, the limit is the cleverness of the compiler and obtaining good performance on distinct accelerators may require tweaking the code.

2.3 Distributed Infrastructure

All the platforms presented earlier need a software to make them easy to use as a consistent set of nodes. This can be done either as an operating system or as a higher level management software.

2.3.1 Cluster

The homogeneous set of nodes that make the cluster can be used in two different ways. One way is by using a cluster operating system that makes the cluster appear as a single system. Another way is by using a resource manager that allow to allocate a set of cores or nodes for a given amount of time.

The simplest way to use a cluster is probably through a Single System Image (SSI). A SSI is an operating system that act from the user point of view as a single NUMA computer although it is spread over several nodes. The point to which the nodes act as a single computer is variable among the systems. However, almost every SSI operating system share a single file system root. It is also quite common that the operating system allow process to be migrated, and to communicate through IPC mechanisms like they were on the same node. The goal of those mechanisms is to allow the programmers to picture the cluster as a distributed shared memory system. Kerrighed [26] and MOSIX [27] are two examples of such SSI.

Another, and more general, way to use a cluster is through a resource manager like OAR [98] or TORQUE [28] that allow to reserve a set of nodes for a given amount of time starting at a given date. Those resource managers hold a description of the available resources and allocate them to a job on demand. A cluster resource manager may also accept job submission that

[7] Basic Linear Algebra Subprograms

basically do not have a start date. There is thus a scheduler that chose the date at which a job should start and make the reservation at that date.

Some computing frameworks can also be conceived as a specialized cluster management systems when they include a runtime environment. The MapReduce Hadoop implementation, for instance, manages the nodes participating, it can remove a node from the list of active nodes if it is failing. It can also add some new nodes as they come alive. Finally, Hadoop MapReduce allow several users to submit their jobs and the framework automatically schedule them at the most suitable date and on the most suitable nodes.

2.3.2 Grid

Similarly to clusters, Grids can be operated in several ways. The most common way to operate a Grid is through a resource manager with a job scheduler. But distributed operating system that work on a Grid is not unheard of.

The task of a distributed operating system that would work on a Grid is much more challenging than on a simple cluster. It indeed need to handle the heterogeneity of nodes and the uncertainty of the WAN network links. XtreemOS [29] is a project that goes this way.

The most common way to operate a Grid remains the resource manager teamed with a job scheduler. This works exactly the same way as for clusters: The system receives a job with some constraints about the resources needed, it then find a set of resources and time where the job can run, and schedule it for that date. However, a Grid resource manager has to deal with the heterogeneity of the resources and the interconnection network. It also should deal with the changing availability state inherent to the Grid as well as the non-dependability of the network. And finally, a Grid resource manager has to deal with the resource allocation policies of each resource owner.

A special case of a Grid management is the Desktop Grid. Their volatility and untrusted nature make the usual Grid management systems unadapted to them. This kind of Grid is thus operated in the opposite way. Instead of waiting to receive a compute task, the nodes ask a server for task to compute, and once the computation is performed, the result is sent back to a server. Although very interesting, this way of working make the Desktop Grid apart from the other grids.

Although it is theoretically possible that a MapReduce framework operate on a Grid, there is no known implementation up to this point. Moreover, it would be arguable that a set of resources managed by a MapReduce framework would still be a Grid with respect to Ian Foster's definition. Especially on the second point of the check-list, the MapReduce programming interface may, or may not be seen as standard or general-purpose.

2.3.3 Cloud

The Cloud computing [30] as defined by the U.S. National Institute of Standards and Technology (NIST) is a model is a model where the user no longer has to own the computing resources but remotely accesses a pool of shared resources. There are five main characteristics that define Cloud computing. First the resource should be allocated on-demand upon user's request. Second it should be accessible through standard network. Third the resources should be pooled to serve several users at the same time. Fourth, the cloud should be capable to adapt rapidly to the users' demand, thus providing the illusion that the computing resources are unlimited. Finally, it should automatically monitor the resource usage and provide this information to the user so that she can control her use and the bill in the case of a pay-per-use system. The Cloud computing is therefore even closer to the ease of use of an electrical grid than the Grid computing.

The kind of service provided by a cloud can be divided in four main categories, each one has a name ending in *-aaS* meanning *as a Service*.

2.3.3.1 SaaS

Software as a Service (SaaS), provides the user with an application ready-to-use running on the cloud infrastructure. It is usually used through a web browser or through an API that can be used by a client program. In this kind of service, the user can only use the provided software and has no direct access to the network, storage system, or operating system. Web-mails or document managements like Google Docs could be classified as SaaS cloud services.

2.3.3.2 PaaS

Platform as a Service (PaaS) provide a working operating system with tools, libraries, and access to usual programming languages. The user then has to deploy and configure his software for the platform. A PaaS platform can be as general-purpose as freshly installed Linux distribution, or more specialized like a MapReduce framework waiting for jobs to be submitted. The first one give more freedom to the user, while the later handle load balancing and fault tolerance for the user. This kind of service is provided by Microsoft Azure [99] or Amazon Elastic MapReduce (EMR) [100] for instance.

2.3.3.3 IaaS

Infrastructure as a Service (IaaS) platforms allow the user to deploy virtual machines images and provide a virtual network between them. The user is then able to run any operating system and any software. A popular example is Amazon EC2 [101].

2.3.3.4 HaaS

Hardware as a Service (HaaS), also called Metal as a Service (MaaS) is one step further into allowing the user to control the resources. It indeed gives the user a full access to the node hardware. The user can then deploy an image with no virtualization layer. Grid'5000 [31] is an French experimental computing platform that provides HaaS.

2.3.3.5 Sky Computing

Sky computing [32] is a quite new computing paradigm that aggregates the resources from several Cloud providers. It provides single network with an all-to-all connectivity allow to build a larger virtual infrastructure. When a user requests some computing resources, the sky management software can allocate the resources from any Cloud provider.

2.4 Conclusion

Whether the applications are compute-intensive or data-intensive, the need for computational power motivates the creation of large computing platforms. Whatever the level at which it is applied, aggregation seems to be the key idea to build faster and larger computation platforms. But this come at the cost of a higher management complexity. Some management software exists to help leverage these platform by providing some abstraction. However, the classical management softwares provide a very limited support for recurring applications patterns, like data partitioning or load-balancing for instance. That's why some frameworks exists to ease the

building of distributed applications. And every MapReduce implementations are, by nature, required to handle and hide the complexity of the underlying platform.

Chapter 3

MapReduce Paradigm

The classical ways to operate a platform usually offers a low abstraction of the platforms. Thus some frameworks have been designed to help leveraging the true potential of large platforms. Several of those frameworks exists, some of them are based on the MapReduce paradigm. This paradigm has the nice property of being adaptable to a large number of platforms and allow the framework to perform a lot of optimizations while making the application writing simple.

This chapter first presents in Section 3.1 the abstract concept of MapReduce programming with its phases and some typical applications. Then Section 3.2 gives some details about some common implementations of MapReduce, especially the usual target properties and some extensions. Section 3.3 shows some common variations of the MapReduce model. Section 3.4 talks about a few overlay tools over MapReduce, and Section 3.5 concludes the chapter.

3.1 MapReduce Concepts

MapReduce can be seen as a programming paradigm to write applications that can be decomposed in a phase *Map* and a phase *Reduce*. This paradigm is intrinsically parallel and based on the high-order functions[1] *Map* and *Reduce*.

Although MapReduce could be generalized to any kind of platform, it is mostly used on highly parallel platforms, and the frameworks are usually designed with big data problematics in mind. The following section try to be as general as possible about the programming model, the next sections are more centered around implementations of MapReduce designed to run on distributed platforms.

3.1.1 Programming Model

MapReduce is a programming model inspired by the high-order functions *map* and *reduce* commonly found in functional languages. In the context of functional programming, the *map* function applies a given function to every element of a given list and return the new list. The *reduce* function aggregates all the elements of a list by applying a given function on an element and a partial result. In MapReduce, those high-order functions are not found as-it-is, however the concept of those data transformations remains.

While the functions *map* and *reduce* were probably invented with functional programming during the 60's, their use for processing large data sets has been popularized by Google in 2004. They published a paper [1] giving an insight about how their MapReduce implementation is designed and used. This paper also explains how they deal with some common issues for terascale computing such as node failure and minimizing the amount of data transfered.

[1] A high-order function is a function that can take a function as argument.

As its name suggests, MapReduce is composed of two main computation phases, the *Map* phase and the *Reduce* phase. There is, however, a lot more than this to make these phase to happen smoothly.

It all start with some kind data partitioning. Every computing node has to access the data it process. Depending on the platform, this may be done in several ways: The data may already be directly accessible to the nodes or may need to be scattered or re-balanced. The data is then split in key-value pairs. This part is usually not accounted when measuring the performance of a MapReduce application.

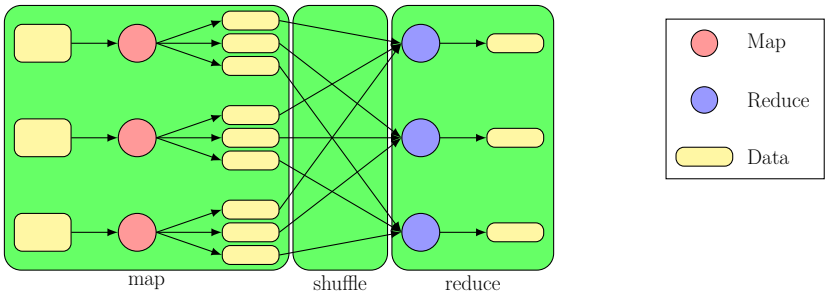


Figure 3.1: General workflow of a MapReduce execution.

Figure 3.1 show the global workflow of a MapReduce application. Every key-value pair is processed by *map* task to produce 0, 1 or n intermediate key-value pairs. Those pairs are then grouped by key in a phase called *Shuffle* to make pairs composed of a key and a list of values. The *reduce* function is then called on each key-list pairs to produce a result in the form of a set of values. For the sake of simplicity, some details are here omitted and are discussed in the next sections.

From the typing point of view, the component that read the files produces a set of key-value pairs of type $k1$, $v1$ written $\langle k1, v1\rangle$. The *map* function takes a key-value pair $\langle k1, v1\rangle$ and produces a $list(\langle k2, v2\rangle)$. The *reduce* function takes a $\langle k2, list(v2)\rangle$ pair and produces a $list(v2)$.

3.1.2 Distributed Implementation

3.1.2.1 Data Management

From a practical point of view, the user code that produces key-value pairs needs to be storage-independent, meaning that any piece of data can be processed on any node. This implies that the MapReduce implementation has to have an I/O component being able to bring the data to a node if needed. However, it is usually more common and cheaper to process the data where they are located instead of bringing the data to a node to process them.

If the storage system is centralized, the data management is simple since every computing node has access to all the data at the same speed. In this case, even the run-time partitioning is also simple since the transfer cost can be ignored.

However, if the storage system is distributed over several nodes, then the computing nodes may have access to some part of the data faster than some others. A common case is when the compute nodes are also the storage nodes and the network has a tree-shaped topology. To optimize the completion, the MapReduce framework should favor the fastest transfers. For instance, if the network has two levels of interconnection: a rack-level and a cluster-level, the framework should first chose to process the data on the node they are, if it is not possible, it should go to process them on another node in the same rack, and as last resort, another node

of the cluster may be chosen. There have been some work [33, 34, 35, 36, 37, 38, 39] around this optimization that showed good results.

This work of partitioning / balancing the data on the compute nodes is done at a block level which usually have a size of 64MB. The blocks are then transformed into a form usable for the *map* phase by splitting them into key-value pairs.

3.1.2.2 Map Phase

The *map* phase takes place right after the chunks of data have been split into key-value pairs. The user code is usually a simple function that takes one key-value pair and produces a new intermediate pair. However, unlike the `map` function in functional programming, it may here produce no output or several intermediate pairs, in case, for instance, of data filtering or when splitting data into chunks.

The set of calls to the *map* function for a given chunk of data is called a *map task*. These tasks are usually processed by a *mapper* process, but this is implementation-dependent. A *mapper* process is usually bound to one processor core. Some implementations of MapReduce may allow a state to survive across the *map* tasks in the same *mapper* process, but this is not provided by the model.

The intermediate data produced by the *map* function are usually not stored into the shared file system and are rather stored locally, in a local storage or in memory. Those intermediate data are then exchanged among all the computation nodes in preparation for the *reduce* phase.

3.1.2.3 Shuffle Phase

The goal of the *shuffle* phase is to group the key-value pairs by key to produce key-list pairs to be reduced. However there is often a coarser level of grouping: the partitions. A partition contains a set of key-list pairs and all the pairs inside a partition are usually processed in ascending key order by a single *reducer* process. This way, the *mapper* process only has to chose the destination node based on the partition id.

A classical way to chose the partition a key-value pair should go into, is by using a hash function. For instance, with p being the wanted number of reduce partition, $hash(key) \mod p$ should produce statistically quite well-balanced partitions. p should be greater than the number of available *reducers* so that every *reducer* has several partitions to reduce.

During the *shuffle* phase, all the *mapper* processes may send some data to all the *reducer* processes. Which is theoretically $\mathcal{O}(m \times n)$ transfers. This may be a problem if the network bandwidth is limited. This thesis tries to address this problem in Chapter 6 and Chapter 7.

3.1.2.4 Reduce Phase

Once the *shuffle* phase is done, the *reduce* phase can start. The *reduce* function takes as argument a pair with a key and a list of values and produces one value as result. For instance, a typical *reduce* function consists in adding all the values in the list. Similarly to the *mappers*, the *reduce tasks* are processed into a *reducer* process. One *reducer* only processes one partition at time and all the pairs inside this partition are processed in ascending key order.

Note that the reduce function does not need to be associative[2] or commutative[3] although this is usually the case. This allows some optimizations. The commutativity allows to not sort the list of values and the associativity allows to apply the parallel prefix computation

[2] An associative function is such that $f(f(a,b),c) = f(a,f(b,c))$ which is valid for operators such as $+$, $\times$ or concatenation.

[3] A commutative function is such that $f(a,b) = f(b,a)$ which is true for some operators such as $+$ or $\times$.

```
void map(String key, String value) {
    for each word w in value {
        emit(w, 1);
    }
}

void reduce(String word, List<int> values) {
    int result = 0;
    for each v in values {
        result += v;
    }

    emit(word, result);
}
```

Figure 3.2: Code example for a MapReduce word count application.

method [40]. If the reduction function is a hashing function like MD5 [41] or SHA-1 [42], then no optimization can be performed due to their non-associative and non-commutative nature.

Although this is not mandatory, a MapReduce implementation may allow a state to survive across the *reduce* function call. This may allow for some custom optimization or more complex computations.

3.1.2.5 Helpers

Some questions have been intentionally left unanswered in the previous sections because they are not essential to the global understanding of MapReduce. There are however some points to be aware of that extend the expressiveness of MapReduce or improve its performance. They are usually some places where the user can put some custom code.

It is usually useless to transfer all the intermediate keys during the *shuffle* phase as they could usually be partially reduced before the transfer. This, however, can only be done when the reduction function is associative and commutative. This is usually implemented as a *Combiner* that acts like a *reducer* on the output of the *mappers* and before they are transfered.

Although the partitions on the *reducers* side are not mandatory to the MapReduce programming model, they are usually supported and come very handy for some applications. However, this means that the way to chose the partition number from the keys have to be customizable by the user. This is usually done by allowing the user to provide his own partition choice function or class.

Another need for customization implied by the existence of the partition is the sorting order. This is again usually customizable by providing a comparison function.

3.1.3 Applications Examples

To better understand how data processing with MapReduce works, a few typical examples follow. Each example emphasis one or several specific features of MapReduce.

3.1.3.1 Word Count

The most common example for a MapReduce application is the *word count*. Figure 3.2 shows an example code in C++-like syntax for a word count. The goal is to count the global number of occurrence of each word in a set of documents. The `map` function takes for instance, a document id as key, and the document content as value. It then emits one pair $\langle word, 1 \rangle$ for every word

```
void map(String id, String content) {
    for each line in content {
        if (line contains the searched word) {
            emit(id, line);
        }
    }
}

void reduce(String id, List<String> lines) {
    for each line in lines {
        emit(id, line);
    }
}
```

Figure 3.3: Code example for a MapReduce grep application

in the document. The shuffle phase produces pairs $\langle word, list(1, 1, 1, ...)\rangle$ that are then reduced into an integer whose value is the number of occurrence of the given word. The same happens to every word of the intermediate data.

In this application a *Combiner* can be of great help to reduce the amount of intermediate data to be *shuffled*. There is, indeed, a lot of redundancy in the intermediate data since all the pairs have a value 1 and a word with a lot of occurrence would be repeated many times. Thus performing a partial reduction would help a lot.

3.1.3.2 Grep

MapReduce can be used as just a bag-of-tasks processing for a large amount of data. The application *grep* is an example like this. The goal of this application is to find a given word or pattern inside a collection of documents and output the text lines where it was found. Figure 3.3 shows an example code for this. The input of the *map* function can be a pair composed of a document id and the content of the document. Then the *grep* job is done and the *map* function emits one line of text for each match. And the *reduce* function does nothing else than repeating its input.

3.1.3.3 Sort

MapReduce can also be used to sort data. The *map* function can take data split in record to be sorted and produce key-value pairs with the key being the data to be compared during the actual sort. Then, the MapReduce model guaranty that withing a given partition, the *reduce* function is called by increasing order of intermediate keys. Thus, if every intermediate pairs are placed inside a single partition, they are processed in the right order. The *reduce* function then just needs to output the value out of the key-list pairs.

Note that the default partitioning function may not be the right choice and need to be customized in order to place all the intermediate data inside the same partition.

3.1.3.4 Matrix Multiplication

MapReduce is not restricted to data-intensive applications, it can also be applied to CPU-intensive applications. A matrix multiplication can be implemented in MapReduce as follow. The input of the *map* function is a pair of matrix element from each matrix. Every pair has to be processed by the *map* function and it just performs a multiplication and produces a pair with the key being the position of result element the value contribute to. The trick to make this work

is to define the function that split the data into key – value pairs such that it produces all the relevant pairs of matrix elements with their coordinate as key. Finally, the *reduce* function just sums the elements for every position of the result matrix. In order to optimize the processing and reduce the overhead of the MapReduce framework, the multiplication can be made block-wise instead of element-wise.

3.2 MapReduce Implementations

3.2.1 Common Wanted Properties

Since MapReduce is mainly used for processing large amount of data on a distributed platform, some properties are wished to hold. Scalability and fault tolerance are of course very important on a distributed platform, especially when it is built with commodity hardware which is unreliable as stated by Google [1]. As a special case of fault-tolerance, it happens that some nodes in a distributed platform takes an unusually long time to compute without completely failing. These case should be handled to provide best performance. Reducing the network utilization is also something wanted since it is probably shared with other applications.

3.2.1.1 Scalability

There are two kind of scalability: horizontal and vertical. The horizontal scalability characterizes the behavior of an application when adding more nodes to the platform while vertical scalability characterizes the behavior of an application when running on faster nodes. If for a given amount of data, the completion time decreases when nodes are added, the application is said to strongly scale. If for a given amount of data **per node**, the completion time of an application remains the same, the application is said to weakly scale.

In the context of distributed platforms, horizontal scaling is usually the most important since it is easier to add more nodes to a platform than to upgrade the existing ones. Moreover, in a MapReduce application, the *map* tasks usually performs an operation in constant time, which makes the *map* phrase run in linear time with respect to the amount of data. And the *reduce* tasks usually performs an operation in time linear to the list size. Thus, what may limit the scalability is the MapReduce framework used.

The framework must handle the partitioning of the initial data in a scalable way. This can only be done if the underlying storage system scales as well. The storage system scalability is also important if the final results is large. The shuffle phase also needs to scale since the trivial implementation performs $\mathcal{O}(r \times m)$ transfers, with m being the number of *mapper* process and r the number of *reducer* processes, which does not scale very well when m or r grow.

3.2.1.2 Fault Tolerance

With large scale distributed platforms the occurrence of fault is too large to be ignored. An usual way to provide some reliability on unreliable hardware is by having one master node that monitors all the worker nodes. As soon as a worker node has been unavailable for some amount of time, it is considered crashed and is no longer given any work. Additionally, any lost work has to be recomputed. What is saved and what has to be recomputed is implementation-dependant.

3.2.1.3 Performance

As usual in distributed computing, performance is very important since it motivates the use of distributed platforms. Beside classical optimization and scalability discussed earlier, a few points are important to consider, especially, stragglers resilience and locality.

Stragglers Resilience As the computing platform become larger, the hardware defect rate increase. This may result in a crashing node, but may also result in a node really slower than it should be. A slow node may also be caused by software misconfiguration. Those slow nodes are called stragglers. If these cases are not handled properly, a distributed MapReduce framework may suffer from the long trail effect, where most of the time is spent waiting for a few tasks to terminate.

Locality In order to improve performance in a distributed platform, it is essential to avoid using the network when possible. Thus if a platform has a tree-like network with 3 levels: nodes, racks, and cluster, it may be preferable to avoid inter-rack transfers and try to favor no transfer at all when possible. Exploiting locality to reduce the network usage can be done during both the *map* phase and the *shuffle* phase.

3.2.2 Classical Implementation

There are mostly two widely known MapReduce implementations. First is the Google's implementation presented in their paper. The second is Hadoop MapReduce.

3.2.2.1 Google's MapReduce

Although Google did not release the code of their implementation of MapReduce, they give some insight in their paper about the way they use it and how they resolved the common problems and wanted properties. Google's MapReduce target their usual computing environment at the time (2004): a large cluster of commodity hardware. This cluster runs the Google File System that bring availability and reliability on top of unreliable hardware.

Overview Google's MapReduce took the form of a library that a program can use. Every job is scheduled by a single *Master* process. The *Master* has a state `idle`, `in-progress` and `completed` for every *map* and *reduce* task along with a *mapper* or *reducer* process identifier if the task is not `idle`. It also keep track of the location of the intermediate data on the nodes running a *mapper* process.

Scalability The Google File System has replicas on several nodes that may be used to avoid some data transfers during the initial partitioning. However, there is apparently nothing that would help the *shuffle* phase to scale. The *Master* is also a single point of failure that has to keep track of all the tasks running and to be run. This theoretically limits the scalability. However, this has not been reported as a limitation.

Fault Tolerance The *Master* process pings every *mapper* and *reducer* on a regular basis to make sure they are still alive. After some time without a response, the *Master* considers that process as failed. If a running *mapper* process has been lost, then all tasks processed by this process are reset to the `idle` state, including the tasks that were completed. There is indeed no guaranty that the generated intermediate data could be read again. Moreover, when a *map* task has been completed twice because of the failure of a *mapper*, then, all the *reducers* are notified that they should read the data from the second *mapper* if they were to read some from the failed one.

If a running *reducer* process has been lost, then only the `in-progress` tasks are reset to the `idle` state, because the final results are written on the global file system.

Since a common cause of crash is the user code for *map* or *reduce* function failing on a bad input data, Google's MapReduce implemented a mechanism to deal with this kind of issue.

Every *mapper* and *reducer* process try to catch fatal errors like *segmentation faults* or *bus errors* and try to inform the *Master* which data was being processed during the crash. If the same data is seen to cause more than one crash, it is skipped in order to make progress in the MapReduce job.

It is apparently considered that the failure of the *Master* process is unlikely enough to be handled manually. It is however suggested that a master implementation could periodically write its state on the disk so that a new *Master* could be started again.

Furthermore, Google's MapReduce guaranty that if the *map* and *reduce* functions produce a deterministic output based on their input, then the computed result in case of failure is exactly the same as it would be without failure. This, however, needs the file system to support atomic file renaming.

Performance Performance is, of course, important for Google. But since their clusters are often based on commodity hardware, providing reliability on top of unreliable hardware is of first importance. Moreover, network bandwidth seems to be a scarce resource.

Stragglers Resilience To cope with the occurrence of stragglers, Google proposed a strategy that consists in running backup tasks at the end of the job. When there is no more task to run, the available *mappers* are used to execute the same tasks than those still running. That way, if one of the lagging task is due to a node-specific problem, the backup task completes first and the slow task can be killed. On the opposite, if the task was long because of the nature of the data, the original task completes first and the backup task can be killed. This way, straggler nodes have a reduced impact on the overall performance.

Locality In their paper, Dean Jeffrey and Ghemawat Sanjay from Google address locality for the *map* phase only. In order to exploit data locality, they suggest processing the data on a node that has a one replica locally. If this is not possible at the moment, they suggests that it is processed on another node connected to the same switch, or, in last resort, on a completely different node.

3.2.2.2 Apache Hadoop

Apache™ Hadoop® is a project that develops open-source software for reliable, scalable, distributed computing. It includes a MapReduce implementation that is usually just referred to by the name Hadoop. This implementation is the most widely used. Facebook and Yahoo! [102] have been reported to use it.

Most widely known, implements everything above and much more. Most difference are name changes. (Like Backup tasks = speculative execution.) However, Google's implementation is designed as a library a program can use to automatically distribute a computation, while Hadoop is designed as a daemon the jobs are submitted.

Since October 2013, a new version of Hadoop MapReduce is available [103]. It is a deep rework of the MapReduce framework which changes some parts of the architecture. Since this is quite new at the time of writing this document, it mainly focus on version 1.

Overview Unlike Google's MapReduce, Hadoop took the form of a framework to program in, and some daemons that run on each node of a cluster. The user then just submits his job to the master which is then scheduled. Hadoop implements everything that has been presented before, and most difference are just a matter of names.

Hadoop relies on a file system it must know about. The most widely used file system is Hadoop File System (HDFS). In order to run a HDFS, there must be some agents running on the nodes, named *DataNodes* which manage the actual data, and one file system master: the *NameNode* which handles the file names and other meta-data.

On top of the file system layer is the real MapReduce management. It is composed of a single *JobTracker* which is the master that receives the MapReduce jobs to be executed. And on every node run one or several *TaskTracker*s which manage a finite and configurable number of tasks at the same time.

MapReduce 2.x (or *MRv2*) is a complete overhaul of Hadoop MapReduce. The *JobTracker* is split between the resource management and execution flow management. The resource manager called *YARN* (for *Yet Another Resource Negotiator*) allocate resources on demand to the jobs. While every job has its own *ApplicationMaster* that manage the application life cycle and that can be run on any node. This new architecture should scale better and be more general since the execution workflow of Hadoop is no longer limited to MapReduce.

Scalability HDFS is known to scale to at least 3500 nodes [15]. Although it is quite hard to find some data about Hadoop MapReduce scalability, it is known [104, 105, 106, 107] that Hadoop finds its scalability limits around 4000 nodes. The reason for this limit is the centralized nature of the *JobTracker*.

Since the task management in MRv2 is done by the *ApplicationMaster*, which can be run on any node, it should scale better. The only remaining bottleneck when running several jobs is the *ResourceManager*.

Fault Tolerance In order to deal with process or node crash, Hadoop MapReduce uses the overall same technique as Google's implementation [108]. The *TaskTracker*s communicate constantly with the *JobTracker* which responds with some commands to be run. If one of the *TaskTracker*s take too long to check for some commands to execute, it is considered crashed. The policy for restarting the *map* and *reduce* tasks are the same as the Google's implementation.

Performance

Stragglers resilience Similarly to Google's MapReduce, Hadoop starts to backup tasks when it detects that some tasks are slower than expected. This is called *speculative execution*. However, where Google's MapReduce only run this mechanism at the end of the job, Hadoop has a metric for evaluating the progress of a task.

The progress of *map* tasks are measured with the percentage of data that has been read. The progress of *reduce* tasks is measured with by dividing it in 3 steps: *copy*, *sort*, *reduce*, each accounting for one third. And progress through each step is again measured by the amount of data processed. Hadoop decides a task is straggling if it is 20% slower than the average of its category and has run for more than one minute.

However, this metric only works fine if the platform is homogeneous and every task has roughly the same size. Moreover, it assumes that the amount of data that is read is proportional to the task progress. Deciding that the 3 steps of a *reduce* task takes one third of the progress is also arbitrary. Addressing these issues, the Longest Approximation Time to End (LATE) [43] scheduler has been introduced. It basically estimates the remaining time to complete a task by doing a linear extrapolation over the time of the amount of data processed. It then speculatively execute those that are supposed to finish last. Another approach called MonTool [44] monitors the system calls patterns to detect the stragglers and schedule the speculative execution. SkewTune [45] and Ussop [46] continuously repartition the unprocessed input

data in a way similar to work stealing. That way, the straggler nodes gets less and less data to process until the progress of the stragglers nodes and the work *stolen* from them meet. The History-based Auto-Tuning (HAT) [47] uses measured historical informations to replaces the arbitrary constants of the default method to detect straggler nodes.

Locality The default Hadoop scheduler uses the same techniques for locality optimizations as Google's MapReduce presented earlier.

Some studies [35] suggests that adding a small delay before scheduling a task on another node may improve the locality and the performance. The farther the node to move the task on, the longer the delay.

During the *shuffle* phase, exploiting locality can be done running a *reducer* process on the node that already has the largest volume of the data of the partition [46]. This can also be done by having some statistics about the distribution of the intermediate keys with respect to the input data [36, 48]. That way, the tasks can be scheduled so that most data that belong to the same partition end on the same node on which the *reducer* of this partition can be run.

There are many variations and optimizations of Hadoop for specific use-cases. The following paragraphs presents some of the most common.

Hadoop On Demand Hadoop on demand is a system built on top of Hadoop that allows to allocate some nodes in a large cluster and starts various Hadoop daemons (especially MapReduce and HDFS daemons) with the right configuration on the allocated nodes. For this, it uses the resource manager Torque [28] for the cluster resource management. All the instances of Hadoop are independent and may even be different versions.

The number of nodes to allocate is computed by the system based on the kind of virtual machine instance to start on the cloud and on the number of workers required by the user.

Elastic MapReduce Elastic MapReduce [100] is similar to Hadoop On Demand in the way it dynamically allocates some nodes in the cloud to deploy a Hadoop cluster. The deployed Hadoop cluster uses the usual HDFS storage system but can also use S3 or most of the storage systems available on Amazon. Just like Hadoop On Demand, it's up to the user to tell the number of nodes to be allocated.

HDInsight Microsoft also allows to allocate a cluster for Hadoop in its cloud Azure, this service is known as HDInsight [109]. The Hadoop clusters can be deployed with or without HDFS. Deploying a Hadoop cluster without HDFS is said to be much faster. Instead of HDFS, users are encouraged to use the Azure Blob storage that is persistent among the clusters creation.

3.2.2.3 Twister

Overview Twister [49] is another MapReduce implementation. It is based on a publish – subscribe infrastructure and data transfers of type *scatter* and *broadcast*. Its main features are about the support for iterative MapReduce which is discussed later in Section 3.3. However, despite being relatively known, this is not a *de facto* standard and has thus not been studied as extensively as Hadoop.

Scalability It relies on NaradaBrokering as communication layer, which is said to scale well. The experiments show that having a tree broker network reduces the time to broadcast some data to all nodes by a significant amount. A factor 4 for 4 leaf brokers with message size of

20MB where all the brokers serve a total of 624 Twister daemons. Twister has also shown an efficiency of 79% on 1632 CPU cores for a job that performs an All-pairs computation.

Fault Tolerance Since Twister is designed for iterative MapReduce it has been decided that the checkpoint granularity is one iteration and not a task unlike Hadoop and Google's MapReduce. This simplifies the implementation and hopefully, makes it more efficient. In case of failure, Twister just restarts the *map* and *reduce* tasks and schedules them based on data locality. If a data has been lost because of node failure, the job just fails.

Performance Twister does not seem to address the stragglers resilience problem at all. However, it seems to exploit locality in its scheduler by trying to process the data close to where one replica lives.

3.2.3 Extensions for Non-Commodity Hardware

Since Google's paper came out, the MapReduce paradigm has been adapted to many platforms. Notably, it can run on GPU, on fast-network with MPI or RDMA, or on multicore nodes.

3.2.3.1 MapReduce for Multi-Core

Phoenix [50] (and more recently Phoenix++ [51]) is an implementation of MapReduce designed to run on multi-core and multiprocessor systems. It is based on Google's MapReduce API and it uses threads to spawn its processes. While the distributed implementation of MapReduce has to take care of locality, it is of less importance for a shared memory implementation like Phoenix. This implementation demonstrates that by applying careful optimizations on every step, it is possible to achieve a low overhead on a multi-core implementation of MapReduce. Phoenix has also been run and optimized for NUMA architectures, thus exploiting the memory affinity. Phoenix also handles fault tolerance at the processor level and handles both transient fault and permanent fault. Phoenix++ globally just improves the performance of Phoenix and makes it more modular.

3.2.3.2 MapReduce for Fast Network

Not all the platforms MapReduce runs on are built around commodity network. In particular, some are built around fast networks like InfiniBand. Those fast networks not only allow a throughput of tens of Gbps and a low latency, they also usually offer RDMA[4]. The idea of leveraging the power of this feature to perform an in-memory merge has been studied [52] through the design of Hadoop-A. Its *network-levitated* algorithm that performs a merge operation without accessing the disk and design the *shuffle* + *reduce* as a single pipeline for the *reduce* tasks. The experiments shows that these techniques double the processing throughput of Hadoop and reduce the CPU utilization by 36% thanks to the RDMA mechanism that bypass the CPUs on both ends for the transfers and for the serialization / deserialization.

Another approach consists in focusing on a Remote Procedure Call (RPC) approach instead of the data transfers and implements an RPC interface based on RDMA [53]. This paper first redesigns the data communications and the buffer management of Hadoop as to eliminate the bottlenecks that would only appear on fast network. It then proposes a design of RPC over InfiniBand for Hadoop. The modified version achieves a latency reduction of 42% – 50% on both

[4]Remote Direct Memory Access

Ethernet 10G and InfiniBand QDR (32 Gbps with IPoIB[5] driver) and a throughput improved by 82% on Ethernet 10G and by 64% on InfiniBand.

The previous work have only focused on using fast networks inside MapReduce, while all the input data have to be read from a distributed file system and all the results have to be written there as well. To circumvent this bottleneck, some work [54] tried using InfiniBand on HDFS for the write operations. For this a hybrid design RDMA / socket is implemented. This solution provide a 30% gain in communication time and 15% with respect to the usage of IPoIB.

3.2.3.3 MapReduce on GPU Platforms

GPUs are around 10 times faster than CPUs and are highly parallel architectures. They are thus naturally suited to run MapReduce. Mars [55] is the reference implementation of MapReduce on GPU. It supports NVIDIA and AMD/ATI graphic chips. Among its particularities, it can be noted that Mars allow the user to **not** run some phases when they are not used. For instance, the sorting phase can be suppressed if the ordering is not important, and the *reduce* phase can be completely omitted instead of running a no-operation function on the data. Given the architecture of GPUs, Mars runs a tree-like reduction in order to use the intrinsic parallelism of the GPU during the *reduce* phase. This constraint the *reduce* function to be associative. Mars has also been integrated into Hadoop in order to make it use efficiently the nodes with GPU and shows a speedup factor up to 2.8.

3.2.4 Extension to Non-Standard Storage

3.2.4.1 BitDew-MapReduce

BitDew [56] is a data management system that allows to aggregate the storage of several platforms and keep track of the location of the data and associate some attributes to them. Its fault tolerance features make it especially suitable for Desktop Grids. A MapReduce programming model has been implemented on top of BitDew [57]. The key features of this implementation are that it is massive fault tolerance, replica management, barrier-free execution, latency-hiding and result checking.

Basically, almost everything was pre-existing in BitDew to make an easy implementation of MapReduce. Especially, BitDew always keep a copy of the data in a master node, and it allows to trigger some computation on the nodes upon data arrival. It however loosen the model of MapReduce computation by only allowing associative and commutative reduction function and avoiding the creation of partitions of the intermediate data. This allows to perform the reduction as soon as some data are available. BitDew-MapReduce actually send the intermediate results with the same key to a node as soon as it has two of them. This means that the reduction is performed in a tree-like computation.

3.2.4.2 BlobSeer-Hadoop

BlobSeer [16] is a storage system that has been optimized to provide best performance under heavy concurrency. It especially provide good performance under heavy concurrency, both for read and write operations. This has been exploited in a modified version of Hadoop that replaces HDFS with BlobSeer [58]. BlobSeer shows better raw performance and scalability than HDFS for read and write operations. When compared to a real data intensive MapReduce job like *Grep*, the modified version perform better by approximately 38%.

[5] IP over InfiniBand

3.3 MapReduce Model Variations

3.3.1 Iterative MapReduce

Some algorithms, to run in on top of a MapReduce programming model, would need several iterations of the *Map* and *Reduce* phases. This is true for instance for the K-Means or PageRank algorithms.

Hadoop supports this by allowing the user to write a `main` function that can submit as many MapReduce jobs as needed. There is, however, no optimization at all. Using this approach means that one iteration must be fully completed before the next one can start. To compute the halting condition during the MapReduce job, Hadoop offers `Counters`. However, if this is not sufficient, another full MapReduce job may be necessary to compute the condition.

3.3.1.1 Twister

Twister [49] is a MapReduce implementation designed to be iterative. The way it handles classical MapReduce jobs has already been discussed earlier. It optimizes the iterative execution by identifying the static and dynamic data. The static data are needed for the computation but are not modified. They thus do not need to be updated, while the dynamic data is modified at each iteration. And facts suggests that most iterative MapReduce jobs show an important amount of static data. Like K-Means would have the (possibly large) set of points as static data and would only produce the clusters parameters at each iteration.

Since it is designed to handle iterative jobs it only provides fault tolerance at the iteration level and not at the task level. The static data also remains in the nodes' memory thus avoiding to re-transfer them.

3.3.1.2 HaLoop

HaLoop [59] is a modified version of Hadoop to natively support iterative jobs. In addition to providing a programming model that handles iterative MapReduce natively, HaLoop provides a loop-aware task-scheduling, caching for loop-invariant data, caching for fixed-point evaluation.

HaLoop does not provide a general way for deciding whether a new iteration should be started. Instead, it provides a way to make either a predefined number of iteration or to search for a fixed-point in the data. For the later case, HaLoop provides an inter-loop caching and automatically calls a user-defined function to compute a distance between the previous and current iteration. For the task scheduling, HaLoop tries to place the tasks that uses the same data on the same nodes, even if the tasks belong to distinct iterations.

On average, HaLoop reduces the job execution time by a factor 1.85% compared to Hadoop and the amount of data transfered during the *shuffle* phase is only 4% of what it was.

3.3.1.3 iMapReduce

iMapReduce [60] is very similar to HaLoop. The main difference being that HaLoop does not identify explicitly the static data from the dynamic data and relies on its scheduler and caching techniques to avoid moving the data unnecessarily. iMapReduce instead uses long running tasks whose lifetime is that of the data they are associated to instead of dying just after the processing is complete. Those persistent tasks sleep when their data is processed and are reactivated when new data arrives. The loop termination is handled the same way as HaLoop does. iMapReduce shows a speedup of a to nearly 5 times when compared to the unmodified Hadoop on a *Single Source Shortest Path* [61] algorithm.

3.3.1.4 iHadoop

iHadoop [62], as its name suggest, is another modification of Hadoop to support iterative MapReduce. The main contributions of this implementation compared to the previous ones is that it allows asynchronous execution of the loops, and of the iterating condition.

The overlapping of the iterations means that the tasks for the next iteration can start before all the tasks of the previous ones are terminated. In addition to this, it tries to schedule on the same node the tasks that shows a producer-consumer pattern to avoid transferring a large amount of data. The asynchronous computation of the continuation condition imply that the compute time of the next iteration may be wasted since it already started.

iHadoop has been measured to perform 25% better than the unmodified Hadoop on iterative jobs. It has also been integrated with HaLoop and shows 35% improvement over Hadoop.

3.3.2 Stream Processing

3.3.2.1 MapReduce Online

The MapReduce model is designed to be used in a batch fashion. Some work extended it to process streams of data. MapReduce Online [63] is an attempt in this direction. The software is called MapReduce Online Prototype and is based on Hadoop. It does this by pipelining the data between the tasks *Map* and *Reduce*. This allows more parallelism, but makes the additional assumptions that the reduction function is associative and commutative. It also prevents some jobs to run properly since there are no longer partitions where the data are sorted by key. Another limitation is that it cannot pipe the data between the output of a job and the input of the next one. This is because the right final result cannot be computed before all the *reduce* tasks are done.

On the technical side, this pipeline is implemented by having the *mapper* send the data to the right *reducer* as soon as they are produced, meaning they opened a socket to every running *reducer*. This also means that, in the naïve implementation, no combiner (*mapper* side reduction) can be performed. However, the intermediate data is still written to disk on the *mapper* side to ensure fault tolerance of the *reduce* tasks. In order to scale better, not all live *mappers* open a socket to all live *reducers*. Some transfers will still happen in a batch fashion. On the *reducer* side, the reduction happen on-the-fly.

3.3.2.2 M^3

MapReduce Online is a first step toward continuous MapReduce jobs that process a real stream of data. However, having HDFS as data source forces it to process a finite set of data. This limitation is addressed by M^3 [64] (Main-Memory MapReduce) which aims at providing a real stream processing on MapReduce.

Since M^3 is oriented toward stream processing, it removes all the occurrences of storage in a file system in the MapReduce workflow. Not only the input is a stream of data, but the output is also kept in memory, as well as the intermediate data. In order to ensure fault tolerance without a distributed file system, the input data are replicated in the memory of several nodes and is kept alive as soon as they are not guaranteed to be processed up to the *reduce* tasks. The intermediate data is also kept in memory of the *mapper* that produced them as long as the *reducer* processing those data does not ensure the data is successfully processed.

All the processing in M^3 has to be incremental. This means that the reduction function performs addition, deletion and update on the current final result. While this broaden the scope of applications that MapReduce applicable to on some aspects, this also restricts its applicability to the reduction functions that are able to update a partial result.

Jobs in M^3 have a few interesting properties. The load balancing is not based on the amount of data to process, but rather on the rate at which they arrive. Instead of processing the data in a new job each time new data come as it would be in a classical MapReduce, there is a single running job that process all the data already there and all the future data. And finally, M^3 is able to pipeline several jobs.

3.3.3 MapReduce on Desktop Grids

In addition to BitDew-MapReduce presented earlier which targets desktop grids, MOON [65] took another approach. First, as a modification of Hadoop it does not break the model of MapReduce and allows the *reduce* operations to be non-associative and non-commutative and to rely on the partition sorting. However, like BitDew-MapReduce, it also assumes that a subset of the nodes are stable to simplify the data loss prevention.

To tune the data management, MOON classifies the data in two categories, those that cannot be lost (mostly the input data) and those that can be recomputed and place those data on the right nodes in HDFS. Moreover, given the unstability of the nodes in a desktop grid the replication factor has to be adapted. Contrary to MapReduce implementations targeting clusters, here the intermediate data also have to be saved on the distributed file system since it can rapidly disappear.

The high unavailability rate of such a platform makes the default stragglers resilience and failure detection of Hadoop not adapted to deal with those. MOON designed a system where a task can be in state *suspended* when the node is believed to be temporarily unavailable, or can be restarted if the node is believed to not return soon. Of course, it also takes advantage of the stable nodes to run a few tasks, especially the last ones.

3.4 Language Approaches

Despite its apparent simplicity, writing MapReduce programs can be fastidious and error-prone. Especially when the dataset is very large and each test-run can take several hours. That's why several other tools exists, only three of them are presented here. All those are related to Hadoop.

3.4.1 Pig

Pig [66], is a high level language designed to simplify the writing of MapReduce jobs. Its aim is to provide a language less imperative than Java or C++, but less declarative than SQL. The language of Pig is called Pig Latin and allows a small number of operators to deal with data.

The language allows to load and parse data into a structured form given by the user to handle a set of tuples afterwards. There are then several ways of processing the data. They can be for instance transformed with a construct that generates some new data from every item of the input data or filtered on some criteria. There are also some SQL-like operators like `GROUP BY`, `JOIN` and `UNION` working like their SQL counterpart, and more. The functions to be applied on the data by the operators are not restricted to a predefined set, they can as well be user-defined.

For instance the word count job could be written as presented in Figure 3.4. The first line loads the file and parse them line by line into a set of array of characters named `inputlines`. These lines are then split into words with the `FOREACH` construct. The `GROUP` operator groups the set of words into a set of pairs containing the grouping key and the set of values (here several times the same word). The second `FOREACH` counts the number of word in each group and returns a set of pairs containing the number of words and the word. And finally, these pairs are written to the output file.

```
inputlines = LOAD '/path/to/file' AS (line:chararray);
words = FOREACH inputlines GENERATE FLATTEN(TOKENIZE(line)) AS word;
wordgroups = GROUP words BY word;
wordcount = FOREACH wordgroups GENERATE COUNT(words) AS count, group AS word;
STORE wordcount INTO '/path/to/newfile';
```

Figure 3.4: Word count expressed in Pig Latin.

Pig then compiles this file into a set of Hadoop MapReduce jobs and coordinates their execution. This tool allows to write complex workflow of MapReduce jobs in a few lines of code without the burden of the pure declarative style of SQL. Pig handles the creation of the jobs and can thus perform some optimizations compared to a straightforward translation of what has been described.

3.4.2 Hive

Hive [67] basic language is HiveQL. It is really just an extension to the standard SQL language. The queries operate on tables similar to usual relational tables, stored in HDFS. However, in addition to the pure declarative syntax, some imperative code can be plugged into the queries. Those queries are then transformed into Hadoop MapReduce jobs.

Since the queries in HiveQL operate on a table, the data either has to be loaded in a table, or has to be converted on the fly into a virtual table. But this allows more freedom to Hive to build the execution plan of the queries.

3.5 Conclusion

MapReduce is a nice programming model designed for parallel and distributed computing. Being adapted and tuned for several platforms from the Desktop Grids to GPU only shows the versatility of the model. Some work has also been conducted to expand the expressivity of MapReduce, notably by allowing iterative jobs and by allowing to express MapReduce jobs with a higher level syntax.

However, there are some optimization approaches that have not been explored yet. Especially, taking a cost-model approach to try to optimize a MapReduce job as a whole. The shuffle phase is also one of the bottleneck of the MapReduce jobs but have been largely left out and deserve some attention. The next chapters lay the basis to study these questions.

Chapter 4

Designing MapReduce with Components

The main contributions of this thesis regards performance and scheduling. In order to implement and test the algorithms, a MapReduce framework is needed. Several of them exist as presented in Chapter 3. However, most of them are large software that are not easy to modify and tune. Especially Hadoop, the reference implementation of MapReduce account for almost $300,000$ lines of *Java* code and, for instance, does not allow easy modifications to control the time at which each transfer of the shuffle phase occurs.

Another MapReduce framework was needed. The work conducted for this thesis happened in the context of the ANR project MapReduce for which a MapReduce framework has been developed. This framework is named *HoMR* (HOme-made MapReduce) and is based on software components approach and rely on two components models: L^2C and HLCM.

This chapter is divided in five sections. Section 4.1 describes in more details what is the problem. Section 4.2 introduces components and gives more details about the component models L^2C and HLCM that are used hereafter. The way those models are used to implement a MapReduce framework is presented in Section 4.3. Section 4.4 discusses the limitations of this approach and Section 4.5 concludes this chapter.

4.1 Problem Description

What makes apart a MapReduce framework from a simple program performing its computation in a map-reduce structure is the reusability of most of the code without much effort. In addition to that, reducing the effort to write a new MapReduce application by reusing some code is really welcome, as much as making the framework easy to port to another platform. Also the framework being efficient is not only desirable, it may be mandatory to some applications.

Being adaptable to other needs by making most of the code reusable is a key property of a framework. In fact, Google's MapReduce [1] can be used as a library, and Hadoop MapReduce can be used as a daemon to which the jobs are submitted. Actually all the machinery that manages the tasks, performs the intermediate data partitioning and the shuffle will rarely need to be modified. Therefore, a good MapReduce framework should allow the user to provide a custom *map* and *reduce* functions as well as allowing to customize the way the data are split into chunks, how to read and write the data, and so on. In addition to that, it is desirable that the MapReduce framework can be easily adapted to several platforms. Thus adding the need for easy modifications of some parts of the code whose performance is platform-dependant.

Regarding the work of this thesis, some unusual parts of a framework had to be modified, especially a transfer scheduler had to be added and the instantaneous throughput of the data

during the *shuffle* phase had to be regulated. Therefore not only the usual parts have to be easily modifiable, but all the parts of the application has to be easy to modify. No known implementation of MapReduce allows to do this. This need of adaptability to others concerns implies that some clear interfaces have to be defined at every level of the application.

Having everything configurable is only worth it if the pieces of code can be reused. Not every MapReduce application is completely different from the others. For instance, it is quite common that the reduce function just performs an addition or a concatenation and most applications read their data from HDFS. The framework should allow to reuse them.

None of the above-mentioned flexibility should be done at the expense of efficiency. Performance is, indeed sometime, a strong requirement for some applications that needs some jobs to be finished by a given deadline. Performance can also be understood as minimizing the energy consumption.

4.2 Component Models

A way to meet the aforementioned requirement is through the use of software components, which, given their property of modularity make it straightforward to adapt an application to similar needs or to another platform and to reuse code. Efficiency is achievable with component models that allow some optimizations to be made at the component level.

```
ps aux | grep foo
```

Figure 4.1: A shell command that list the process and only keep the lines containing the word `foo`.

A rough example to give the intuition of what are components is the shell pipeline as show in Figure 4.1. In the Unix command line interface every command performs a specific task. Those commands can then be combined with a pipe symbol | meaning that the standard output of the command on the left of the pipe is connected to the standard input of the command on the right. For this system to work properly, several mechanisms need to be defined outside of the commands. A manager software (the shell) needs to create the pipes by themselves and to connect the commands that needs to be connected. But also, the protocol for communicating among the processes has to be defined system-wide. The notion of standard streams is defined by the Unix standard, and it is common practice to make commands that produce output and expect input in a line-wise text format.

A component model [17] is a definition of what are software components and how they interact with each others. Components can be seen as an extension the notion of classes in object oriented models. Components are not restricted to expose one interface, they may expose several of them under the notion of port for other components to use. But, in addition to providing interfaces, components can also require interfaces, thus making the component only depend on an interface instead of depending directly on a specific implementation of a feature. Those well-defined interfaces make the component-based software very modular. This modularity makes very straightforward to achieve the aforementioned requirement of adaptability to similar needs and platforms as well as code reusability.

One of the key principle of the component approach is the separation of concerns. This principle dictates that a given aspect of the global program should not be handled by several components, just like a shell command only performs one task. This emphasizes the need for a clear separation between the components and a well-defined way to communicate. The decoupling between the main code addressing a given concern and the features it uses is also made necessary by this principle. In the remaining of this thesis, two component models are

used, a low-level component models L^{2}C [18] and a higher-level component model HLCM [19] based on L^{2}C.

4.2.1 L^{2}C

L^{2}C is a component model and its implementation. The model it defines is *low level* and thus only deals with primitive components and primitive connections. The only implemented primitive connections are a C++ *use – provide*, a CORBA *use – provide* and MPI communicators. A *use – provide* connection is defined as a component exposing an interface, and another component using this interface. In the Object-Oriented world, this is the equivalence of an object implementing an interface when another object uses this interface. The implementation of L^{2}C is based on C++ and every component and every interface is a class.

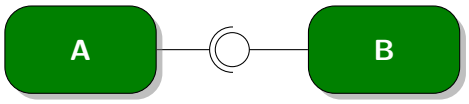


Figure 4.2: Simple connection between two components.

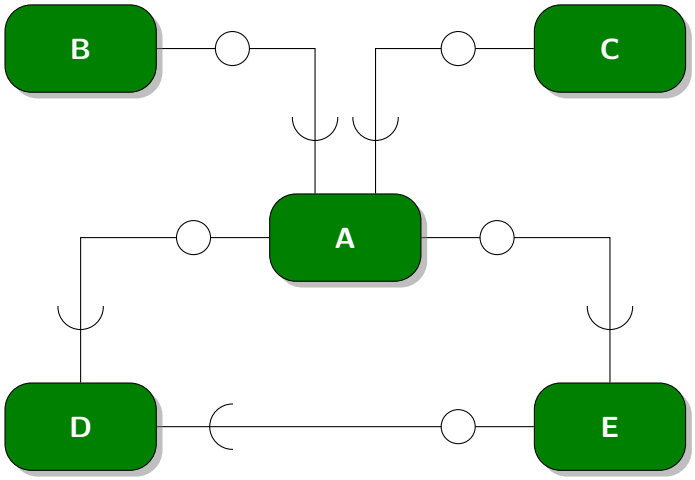


Figure 4.3: Non-trivial assembly between 5 components.

As stated before, the components only communicate through clearly defined interfaces named port. There are two classical kinds of ports as defined by UML [68]. The *use* ports and *provide* ports. A *use* port can be connected only to *provide* ports, and *vice-versa*. But this is not the only restriction. Every port has a type, and only ports with matching types can be connected. Just like the object oriented programming only allows an instance of a subclass of a type `T` to be used as an object of type `T`. The simplest *use – provide* relation is shown in Figure 4.2 which uses the UML notation where the —(shape represent a *use* port and the —○ shape represents a *provide* port. In this figure, the component `A` can then perform some method call on those that are exposed by the component `B`. Figure 4.3 illustrates the case where a component can export several *use* and *provide* ports.

Listing 4.1 shows a concrete example, a simple component `Server_HW` that *provides* an interface `Hello` and Listing 4.2 a component `Client_HW` that *uses* this interface. Providing an interface is done by implementing a C++ abstract class and then using the macro `L_CPP_PROVIDE` to declare that the class implements the interface. Another syntax exists that allows to *provide* several ports with interfaces that would conflict, but it has not been useful in this work. Also

```
class Server_HW:
    virtual public Hello {
public:
    virtual void greet(string name) {
        cout << "[Server_HW] hello " << name << "!" << endl;
    }
};

LCMP(Server_HW)
    L_CPP_PROVIDE(Hello, greeter);
LEND
```

Listing 4.1: Code of the server Hello component that provides an interface `Hello` through a `greet` port.

```
class Client_HW {
public:
    Hello *greetservice;

    Client_HW(): greetservice(0)
    {}
};

LCMP(Client_HW)
    L_CPP_USE(Hello, greetservice);
LEND
```

Listing 4.2: Code of the client Hello component that uses an interface `Hello` through a `greetservice` port.

in Listing 4.2, the component `Client_HW` declares that it has a *use* port by using the macro `L_CPP_USE` that will make the framework set the attribute `greetingservice` of the instance to a pointer to the instance of the `Server_HW` component. L^{2}C can also easily manage *use* – *provide* connections through CORBA by using two macros `L_CORBA_USE` and `L_CORBA_PROVIDE`.

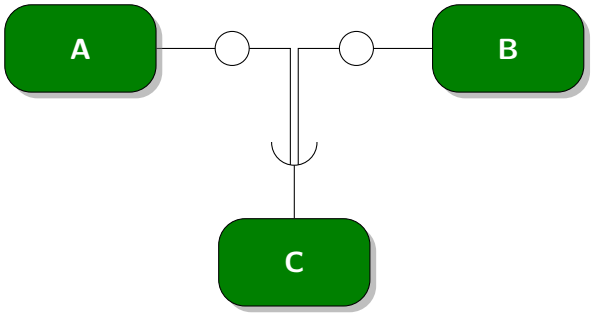


Figure 4.4: Example of assembly with a several *provide* for a single *use*.

An extension of the simple connection is when a single *use* port can be connected to several *provide* ports as shown on Figure 4.4. This is usually implemented by having a list of component references on the user side. This may be useful when a lot of identical components are created and have to be treated the same way. The user component then just iterates over the list of references to call the same function. The *multiple use* is implemented in L^{2}C using the same macro `L_CORBA_USE` but giving it as second argument a method name that is called once per connected component with the corresponding pointer as argument. A typical implementation

```
class Client_HW {
public:
    virtual void greetservice(Hello* service) {
        m_greetservice.push_back(service);
    }

    Client_HW(): m_greetservice()
    {}

private:
    std::vector<Hello *> m_greetservice;
};

LCMP(Client_HW)
    L_CPP_USE(Hello, greetservice);
LEND
```

Figure 4.5: Example code of a component using several *provide* ports.

saves the pointers in a list. Figure 4.5 shows another client component that could be used with several instances of the component `Server` in Listing 4.1.

```
<lad xmlns="http://avalon.inria.fr/llcmcpp3">
    <process>
        <!-- it is started through its "go" port (of type ::llcmcpp::Go) -->
        <start_property instance="client" property="go"/>
        <instance id="serv1" type="Server_HW"/>
        <instance id="serv2" type="Server_HW"/>
        <instance id="client" type="Client_HW">
            <property id="greetservice">
                <cppref instance="serv1" property="greeter"/>
                <cppref instance="serv2" property="greeter"/>
            </property>
        </instance>
    </process>
</lad>
```

Figure 4.6: Example L^{2}C assembly description in LAD for the hello world example with *multiple use*.

In L^{2}C, the assembly of components has to be extensively described in a XML dialect called LAD. This file list all the components and connections between the components. Figure 4.6 shows the assembly of the example code above with *multiple use*. There is, indeed, in L^{2}C no notion of component that would be composed of several smaller components. There is no way of expressing repetition either. These issues are addressed by HLCM.

In addition to being given references to other components, the components in L^{2}C can be configured with a static value written in the assembly file. This can allow some global configuration known during the deployment of the application. Listing 4.3 shows how this can be used with a component that just print a given string when the method hello is called.

4.2.2 HLCM

HLCM (High-Level Component Model) [19], as its name suggest, allows to express higher level concepts. It uses L^{2}C as a basic level of components and allows to define composite components

```
class HelloWorldProp {
public:
    void hello() {
        cout << "[HelloWorldProp] " << v << endl;
    }

    string v;
};

LCMP(HelloWorld)
    L_PROPERTY(string, v);
LEND
```

Listing 4.3: Example L^{2}C assembly description in LAD for the hello world example with *multiple use*.

and composite connectors. Its assembly language allows to express some patterns in the component instantiation with the help of looping constructions and genericity similar to the C++ templates.

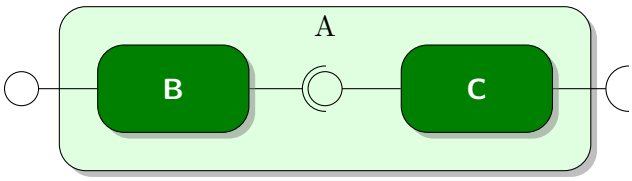


Figure 4.7: Composite component made from 2 components using the *use – provide* connection semantics.

In HLCM, the definition of a component is hierarchical. A component is either a primitive component implemented in L^{2}C, or a composite that contains several components. This allows to address a higher level concern without worrying about some details of implementation. Figure 4.7 shows an example of a component A made by assembling two components B and C. The exposed ports of the composite are, here, the unconnected ports of the subcomponents.

```
component StringProcessing
exposes {
    in: InputString;
    out: OutputString;
}
```

Listing 4.4: Example of component declaration in HLA.

The language used to describe the components in the HLCM model is named HLA (High-Level Assembly). In HLA, every component, primitive or composite, have an *interface* expressed as in Listing 4.4 which declares that the component `StringProcessing` that exposes two connections named `in` and `out` with the interface `InputString` and `OutputString` respectively. It can be seen that a HLA component only declares its available connections with their names and types. A composite `A` implementing that component is shown in Listing 4.5. The subcomponents are declared into the `components` section. Here, two subcomponents are declared with the names `compB` and `compC` which are instances of component `B` and `C`. The section `connections` connects the subcomponents together using the `merge` function. And finally, the exposed connections of the component this composite implements are declared in the `exposes` section. The

```
composite A
implements StringProcessing {
components:
    compB: B;
    compC: C;
connections:
    merge(compB.out, compC.in);
exposes:
    in = compB.in;
    out = compC.out;
}
```

Listing 4.5: Example of composite definition in HLA.

```
//#implements=GoProxy
LCMP(GoProxy)
    L_CPP_PROVIDE(Go, go);
    L_CPP_USE(Go, proxyfied);
LEND
```

Listing 4.6: Example of `l2c` file used by HLCM.

names must match those declared in the component. Alternatively, a component in HLCM can be implemented directly by a primitive component in L^{2}C. For this a `l2c` file must be provided, Listing 4.6 shows an example of it. This file is actually both valid C++ code using the macros provided by L^{2}C and also part of the HLA language. It is thus a common practice to include the `l2c` file after the definition of the class implementing a component. The main difference with the pure L^{2}C component declaration is the presence of a `//#implements` line that makes the link between the C++ class and the HLA component name.

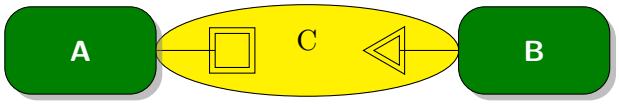


Figure 4.8: Example of simple connection between two components A and B using a connector C.

The links between components supported by a *use* and *provide* ports in L^{2}C are generalized in an abstract way by HLCM through the notion of connectors. With connectors, there are no more ports on the component side, they are replaced by *connections* which, in the HLCM terminology may be *open* if at least one role can still be fulfilled, or *closed*. With the connectors, there is no *use* – *provide* port duality. The connectors can have an arbitrary number of *roles*, which, in HLCM have an interface and a protocol. Figure 4.8 shows an example of 2 components connected with a connector with 2 roles which may have a different interface and a different protocol of communication.

Each role in a connector has a cardinality. It means that several connections of the components can be connected to the same role of a connector. This provides an alternative model for the *multiple use* stated earlier. Figure 4.9 shows an example equivalent to the *multiple use* using a connector.

Connectors are not forced to be simple primitive connectors implemented directly in the component framework. Like the components can be made by assembling other components, connectors can be made from components and connectors, thus allowing more complex behaviors to happen inside the connectors. For instance a connector could have two roles, one for a C++

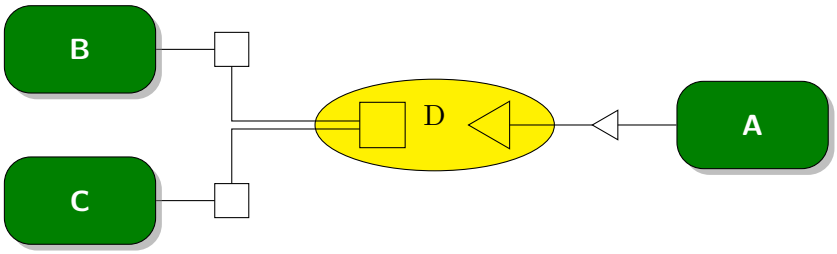


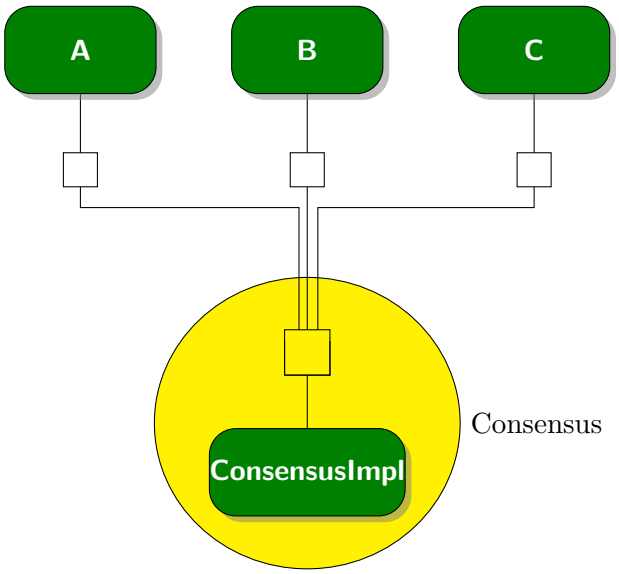
Figure 4.9: Example of a connection between 3 components equivalent to the *multiple use*.

use and one for a C++ *provide*, and internally convert those method calls to CORBA in order to work over a network connection. A connector could also implement a consensus algorithm through a single role with a cardinality 1..*n* and its implementation would apply a consensus algorithm. Conversely, a connector could implement a mandatory access control to allow (or not) a method call to succeed, thus exposing a `lock` role in addition to a `use` and `provide` pair of roles as before. Figure 4.10 shows both cases.

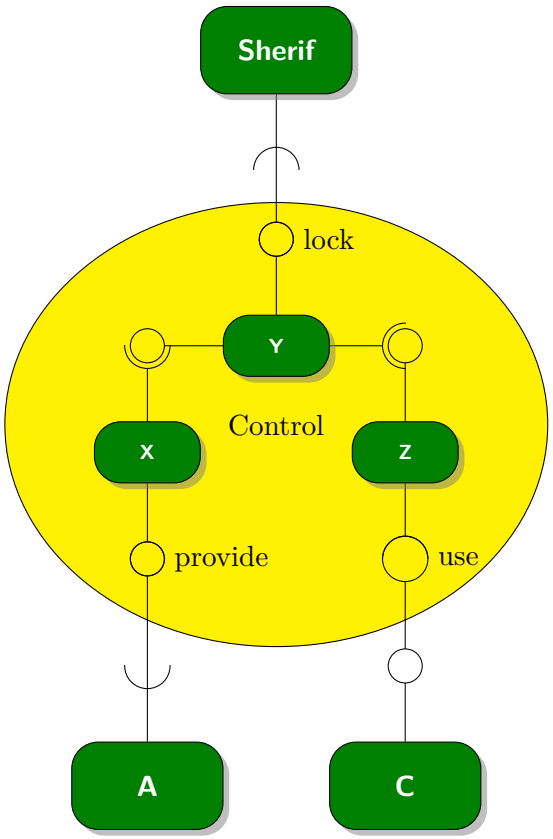
```
component GoProxyArrays
exposes {
    go_provides: MultiProvide<Go>;
    go_uses:     MultiUse<Go>;
}

composite GoProxyArraysImpl
implements GoProxyArrays
{
components:
    goproxy1: [each (i | [1 .. 10]) { GoProxy }];
    goproxy2: [each (i | [1 .. 10]) { GoProxy }];
connections:
    each(i | [1 .. 10]) {
        merge(goproxy1[i].go_use, goproxy2[i].go_provide);
    }
exposes:
    go_provides = merge({
        part_provider = [each (i | [1 .. 10]) {
            goproxy1[i].go_provide.provider
        }]
    });
    go_uses = merge({
        part_provider = [each (i | [1 .. 10]) {
            goproxy2[i].go_use.user
        }]
    });
}
```

Listing 4.7: Example of arrays of component usage in HLA.

HLCM also allow to express arrays of components and arrays of exposed connections. Listing 4.7 shows an example of a component `GoProxyArrays` exposing two connections named `go_uses` and `go_provides`. This component is implemented by the composite `GoProxyArraysImpl` which in composed of two arrays of components `goproxy1` and `goproxy2` made of 10 instances of the component `GoProxy`. The component `GoProxy` is assumed to expose two connections named `go_use` and `go_provide`. Those arrays are connected together one to one using the `each` construct. In the `exposes` section, the open connections of the compo-



(a) 3 components connected through a given implementation of a consensus connector.



(b) 2 components A and C connected through an implementation of a Control connector under the supervision of the Sherif component.

Figure 4.10: Example of a connector with more or less than 2 roles.

nents are merged to build the `go_provides` and `go_uses` connections. However, these arrays of components are more useful when used with the genericity.

```
composite GoProxyArraysImplGen<Integer len>
implements GoProxyArrays
{
components:
    goproxy1: [each (i | [1 .. len]) { GoProxy }];
    goproxy2: [each (i | [1 .. len]) { GoProxy }];
connections:
    each(i | [1 .. len]) {
        merge(goproxy1[i].go_use, goproxy2[i].go_provide);
    }
exposes:
    go_provides = merge({
        part_provider = [each (i | [1 .. len]) {
            goproxy1[i].go_provide.provider
        }]
    });
    go_uses = merge({
        part_provider = [each (i | [1 .. len]) {
            goproxy2[i].go_use.user
        }]
    });
}
```

Listing 4.8: Example of arrays of component usage in HLA with length as arugment.

```
class GoProxyId: virtual public GoId {
public:
        GoProxyId(): thisid(-1), proxyfied((Go *)-1)
        {}
        virtual int32_t get_id() const {
                return thisid;
        }
        virtual void go() {
                proxyfied->go();
        }
        int32_t thisid;
        Go *proxyfied;
};
#include "goproxyid.l2c"
```

Listing 4.9: L^{2}C component declaration with a property.

HLCM components can have arguments like the C++ templates. Those arguments can be values with one the basic types of HLA (integers, strings, ...), or the types themselves, or other components or connectors. When the argument is a value, it can be used by an HLA composite as the length of an array of components as seen in Listing 4.8. The value can also be used directly by a primitive component in L^{2}C through a property port. Listing 4.9 shows a primitive L^{2}C component and Listing 4.10 shows its `l2c` declarations using the value genericity to make the array length more flexible.

When the composite argument is a component, it can be used by a composite to instantiate some subcomponents. This feature makes it possible to express templates of applications where not everything is completely defined by a composite. This is simply done by specifying the type of the argument as `component` instead of a primitive type. This genericity with components as

```
//#implements=GoProxyId<thisid>
LCMP(GoProxyId)
        L_PROPERTY(int32_t, thisid);
        L_CPP_PROVIDE(GoId, goid);
        L_CPP_USE(Go, proxyfied);
LEND
```

Listing 4.10: Example of a L^{2}C component using a value provided by the genericity in HLA.

arguments allows to define skeleton of applications, which proves very useful when designing a MapReduce application.

```
composite GoProxyIdLocImpl<Integer thisid, String sl>
with {Process(sl)}
implements GoProxyIdLoc<thisid, sl> {
components:
    proxy: GoProxyId<thisid>;
exposes:
    goid      = proxy.goid;
    proxyfied = proxy.proxyfied;
}
```

Figure 4.11: Example of use of the `with` construct in HLA.

Additionally, HLCM is able to state in which process a component should be. This allows to generate an assembly that can be run on a distributed platform. The way this is expressed in HLA is through a `with` construct that apply some constraints on the component which can state that the component should be placed on a process with the given name with the construct `Process`. Figure 4.11 shows an usage of this. However, this become useful only when several components have a distinct process constraint. This feature is however, still experimental and under research.

4.3 Implementing MapReduce

A MapReduce framework named HoMR (HOmemade MapReduce) has been implemented by Julien Bigot and Christian Perez in L^{2}C and HLCM. However, it did not exactly fit for the needs of this thesis. Especially, several transfer scheduler needed to be implemented. Some of them should control the moment where each of the $m \times r$ transfer start. Some should also control the data rate of this transfer on the *mapper* side. This section shows the modifications brought to the software both on L^{2}C and HLCM levels.

4.3.1 HoMR in L^{2}C

4.3.1.1 Overview of the Architecture

Figure 4.12 shows the HoMR component architecture in L^{2}C for two processes. Components in green are considered part of the MapReduce architecture, those in blue are those always provided by the user. And those in gray are actually just CORBA bridges that transform a C++ call to a CORBA on one side, and back to a C++ call on the other side.

There is currently no clean way to make a generic CORBA bridge component since it must always provide or use a CORBA interface that copies the bridged C++ interface. Thus, all the bridges component are distinct and have to be written by hand. Moreover, there are always a

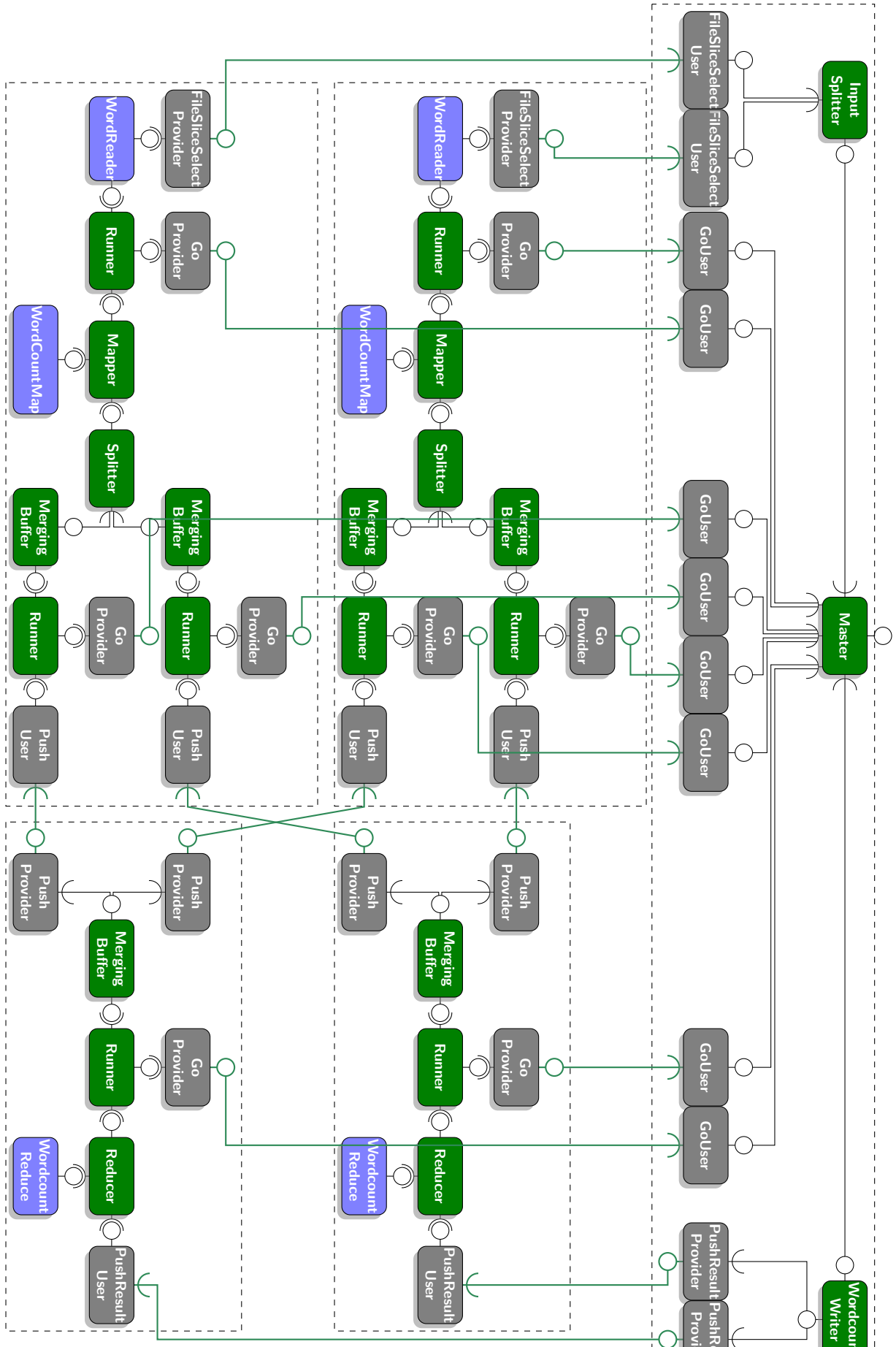


Figure 4.12: HoMR assembly for 2 processes.

use and *provide* side. The naming scheme used here of those CORBA bridges is to start with the name of the interface bridged and end with *Provider* or *User* depending if the CORBA interface is used or provided.

In this assembly, the `Master` component is responsible for scheduling and orchestration of the whole job. It configures the `InputSplitter` component with the name of the input file, starts all the *map*, *shuffle* and *reduce* through `Go` interfaces.

The `Runner` components appear several times in the assembly and its only job is, upon a call to its `Go` method, to pull data from a source and push them to a destination. This component is used to feed the data to the `Mapper` component, to perform the data transfer that make up the *shuffle* phase, and to run feed the data (and hence actually run) the `Reducer` component.

The way the *shuffle* phase is handled is that the `Splitter` component choses the process that will perform the reduction for a given key, thus effectively splitting the data. This part is called the *partitioner* in Google's MapReduce. The output of the `Splitter` is connected to several `MergingBuffer` which merge all the $\langle key, value \rangle$ pairs into a $\langle key, listofvalues \rangle$ pair for an identical key. Some implementations call this the grouping operation. When the `Splitter` has decided that some data should be reduced on another node, they will first accumulate in a local `MergingBuffer` until the scheduler launches the `Runner` so it can transfer its data to the corresponding `MergingBuffer` on the target process. This second `MergingBuffer` aggregates all the data from all the processes that will be processed in that process.

The goal of the `Mapper` and `Reducer` components is to iteratively call the user components `WordCountMap` and `WordCountReduce` on every key-value pair to be processed. That way, the user components only process one pair at time.

The `WordReader` reads blocks of data and produces words as output. Since the MapReduce jobs would likely not need to split data in words, this is part of the job-specific components. The `WordCountMap` performs the actual *map* computation. It transforms a word into a pair $\langle word, 1 \rangle$. The `WordCountReduce` component performs the actual word counting and transforms a pair $\langle word, [1, 1, ...] \rangle$ into $\langle word, count \rangle$. Finally, the `WordcountWriter` component is also user-defined in HoMR to allow to write the result data in a useful format.

As a side note, this assembly does not have a combiner to perform a partial reduction on the *mapper* side, but this could be easily done by adding an intermediate component between the output of the `MergingBuffer` and the `Runner`.

4.3.1.2 Modifications

There are two main kinds of algorithms presented in Chapter 7. The first kind only controls which transfer to start. The second kind also controls how much bandwidth this transfer should take.

Matching the processes In the previous architecture, the `Master` component is responsible for starting the all the transfers. Since this task will require some non-trivial computations and several algorithms will be tried, this task should be split out of the `Master` component. This is done by creating a `TransferScheduler` interface that is used by the modified `Master` component (called `MasterS`, `S` for *scheduler*). This interface allows the `Master` to notify the transfer scheduler when a computation ended and to run the scheduler to start the transfers when possible. The idea is to start the transfer scheduler as soon as the *map* computations are started and inform it whenever a *map* computation ended. On a side node, two components `RMBasic` and `ModelBasic` (RM for Resource Model) have been added to ease the handling of the data regarding respectively the platform and the state the computation is in (e.g. is a node in its *map*, *shuffle* or *reduce* phase).

Another issue with the previous assembly is that the `Master` component has a *multiple use* connection between with the `Runner`s that handle the *shuffle* phase, a *multiple use* for the components starting the *map* computations and a third *multiple use* for the components starting the *reduce* computation. In L^{2}C, in case of *multiple use* there is no way to tell apart the components the reference belong to. It is therefore not possible to identify which `Runner` to start when a *map* computation terminates nor which *reduce* computation to start when a transfer terminates. To overcome this limitation, a proxy component is added between the `Master` or transfer scheduler, and the components starting the computations or the transfers. The proxy components in front of the *mapper* and *reducer* processes exposes a `GoId` interface that provides a `go` method and an `id` method. The `go` method, when called, calls the `go` method of the *mapper* or *reducer*. The code of the component implementing this interface has been already shown in Listing 4.9 and Listing 4.10. The transfers of the *shuffle* phase are handled similarly. The interface is called `GoId2` and exposes two methods `id1` and `id2` returning the id of the processes this transfer would happen between.

Bandwidth regulation The aforementioned design modifications would be enough to run all the discrete transfer schedulers that either run a transfer or let it wait. However, some schedulers also throttle the bandwidth of every transfer in order to avoid contention. This leads to two modifications in the previous assembly: one to actually throttle the bandwidth allocated to a transfer, and one to communicate the limit bandwidth from the scheduler to the process. This is done by implementing a `RunnerRegulated` component which pushes data at a maximal fixed rate. This component exposes a `GoRegulatedBandwidth` that merges a `Go` interface and a `RegulatedBandwidth` interface in order to avoid the burden to the user component to match those components together as presented earlier. However, since this `RunnerRegulated` component replaces the former `Runner`, the master component needs to know which between which process it will run the transfer and thus need a `GoRegulatedBandwidthProxyId2` component as earlier.

Generating fake data In order to control the amount of data generated for the experiments with the transfer schedulers, the components `WordReader` that reads the data from the file system need to be replaced by a `WordGenerator`. This component is configured with two properties that indicate the number of unique words and the total number of words that will be generated. In this situation, the components `InputSplitter` that partition the input data into chunk is no longer used. So in order to keep the modification minimal, especially regarding the interface with the `Master` or `MasterS`, a `FakeInputSplitter` is devised. It does not actually partition the data, it just start the word generator instead.

4.3.2 HoMR in HLCM

4.3.2.1 Overview of the Architecture

HoMR, as written in L^{2}C, is not easy to use and require an external program that would generate the LAD for the given number of *mapper* and *reducer* and is not easy modify and maintain. So it has been written in HLA by Christian Perez to use the features of HLCM like the composites, looping constructs and genericity. The implementation presented here is the one without the modifications presented in Section 4.3.1.2. Since it only uses C++ and CORBA *use – provide* connectors (with *multiple use* variations), the connectors here are drawn with the formalism used for L^{2}C components.

Per design of HLCM, the whole MapReduce application is a single component which has a single open connection that provides a `Go` interface. This interface is what will be run to

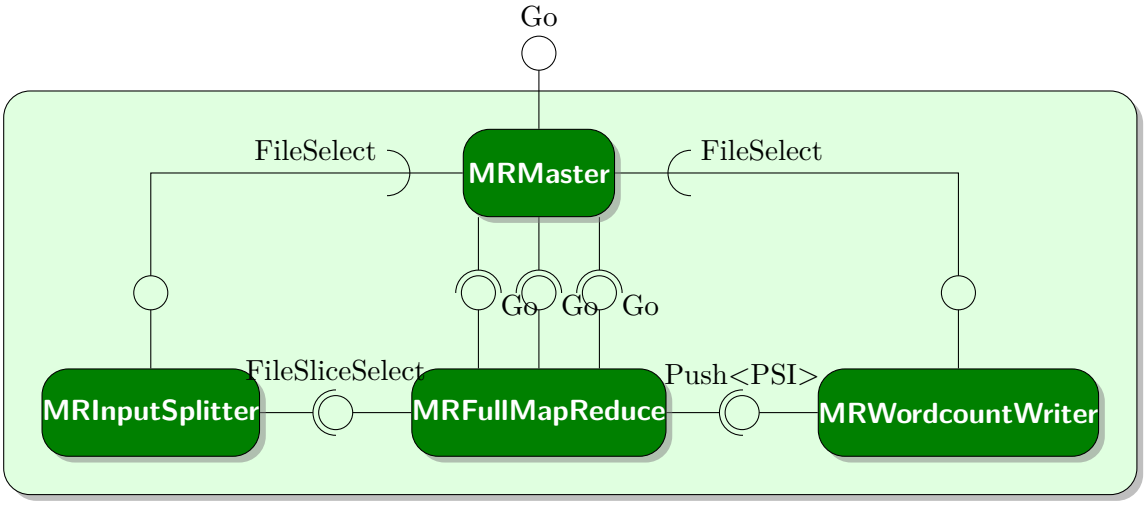


Figure 4.13: Initial global design of HoMR in HLCM.

start the application once deployed. As show in Figure 4.13, this component is itself composed of 4 subcomponents that instantiate all the components that are unique in the assembly. Namely, this is the master component and the I/O components. The MapReduce core component `MRFullMapReduce` implements the actual *map*, *shuffle* and *reduce* phases. The components `MRMaster`, `MRInputSplitter` and `MRWordcountWriter` are directly implemented by the primitive components `Master`, `InputSplitter` and `WordcountWriter` respectively. Their roles are the same as before and handle the coordination, data reading/splitting and data writing. Those components are placed on the master node, except for the component `MRFullMapReduce` which is distributed on all the compute nodes.

`MRFullMapReduce` has 5 connections, one to receive the input data from the splitter, one to send the data to the writer, and 3 multiple provide to start the *mappers*, *shuffle* and *reducers*. Figure 4.14 shows the assembly that is generated by this component for 2 *mappers* and 2 *reducers*. Globally, the components are located on some processes here represented as dashed boxes. All the connections that have to go out of a box are drawn in thick dark green lines meaning that it's a CORBA interface that is used. The `lReadMap` components are composed of 3 subcomponents, one that splits a chunk of input data into words, a runner that takes the words and feeds them into the actual *map* computation, and the *map* component that produces the intermediate key-value pairs as output of this component.

The component `MRSplitter` plays the role of the partitioner in Hadoop. It choses which reducer will process every key of intermediate data. The default implementation is based on a hash function.

There are several instances of the `MRMergingBuffer` component. Its role is to merge the key-value pairs with the same key into a key-list of values pair. This component is found just before sending the data on the network for the *shuffle* so that all the values for a give key are send at once. It is also found just after the data has been received from the *mappers* so that several lists of values for the same key are merged together. Between them is only a `MRRunner` whose only role is to pull the data from one side and push them on the other side.

The `lReduce` component has nothing more than the actual *reducer* computation and a runner to pull the data from the merging buffer and push them to the *reducer* component.

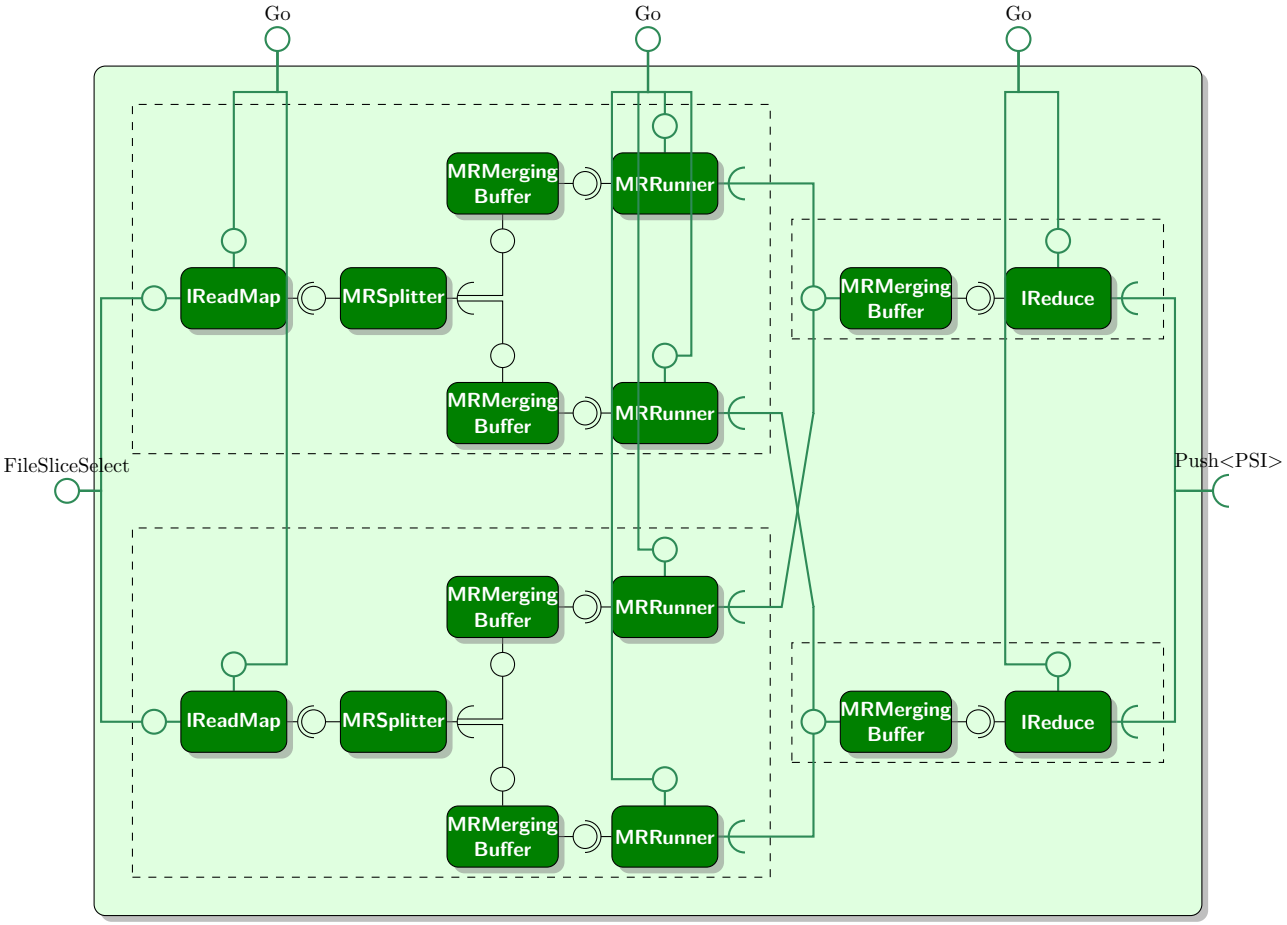


Figure 4.14: Initial design of component `MRFullMapReduce` of HoMR.

4.3.2.2 Modifications

This design has several issues, all of them are about the difficulty of reusing the MapReduce skeleton as-it-is. The first issue is that it is not easy to adapt the assembly to perform another computation than a word count since the word count components are hard coded deep into the assembly. The second issue is that some work of this thesis (presented in Chapter 7) need to swap the scheduler for another one. Most of those require the `GoProxyId` and `GoProxyId2` components to be used as stated previously.

The components may be grouped in 3 categories. Those changed to perform another computation. Those changed for performance reason or to adapt the application to the platform. And those that are part of the framework but need to be modified for this thesis.

The simplest thing to do is probably to pass components as arguments of the MapReduce skeleton to provide a *map* and a *reduce* component that will be integrated in the assembly. In addition to just the *map* and *reduce*, the word splitting of text is also very specific to the word count job. In the previous assembly, these components corresponds approximately to `lReadMap` and `lReduce`. The difference is that the component that splits a chunk of data into words should not read the block for the file system directly so that the job-specific operations (splitting into words) would be distinct from the IO operations (reading from the file system).

The components responsible for reading and writing the data from / to the storage system are specific to the platform and thus should not be part of the MapReduce skeleton itself. In

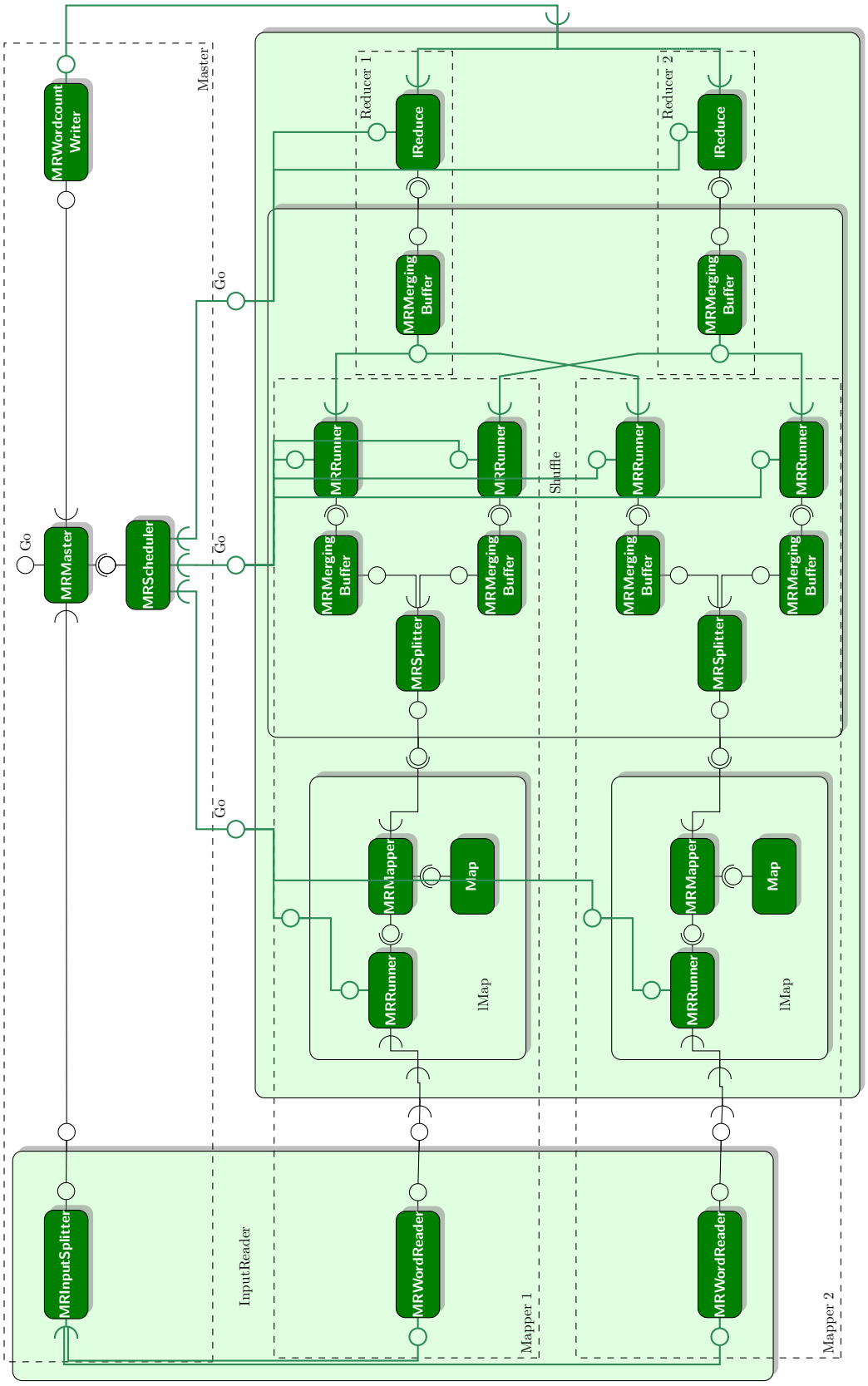


Figure 4.15: Modified design of component HoMR in HLCM.

the previous design, the actual data reading was done by a subcomponent of `lReadMap`. This should be done by an `InputReader` component. The `MRWordcountWriter` for writing can be kept as-is. It is specific to the type of data to be written but it is not a problem since it will be an argument of the global MapReduce component. The left part of Figure 4.15 shows modified design of the data input.

Still for performance, but also for this thesis, the scheduler has to be split out of the `Master` component. Although a composite aggregating the scheduler and the master without scheduler could be defined, this does not seems necessary. Some scheduling algorithms require another transfer runner that would be able to control the actual throughput of the network transfer. To simplify this, a shuffle component can be defined to that would span from the *splitter* that partition the intermediate data up to the merging buffer on the *reducer* side as shown in the center of Figure 4.15. The actual component for the splitter should be an argument of the `Shuffle` component because it may have some performance implications. This component would have $m \times r$ connections to receive the data, $m \times r$ connections to set the wanted throughput and start the transfers, and r connections for the *reducers* to read. Note that this component span over several processes without encompassing them totally, but this is not a problem.

4.4 Discussion

This new design of MapReduce with HLCM components extensively uses the genericity and always requires several arguments. To ease its use, either the support for default values could be added in HLCM, or some composite could be defined with less arguments and fix the value of the others. For instance a component `MapReduceNfsCorba<M, R, Map, Reduce>` could be defined that would make the input and output read and write from / to the shared file system and that uses CORBA as protocol for the *shuffle* phase. Also, in the proposed design, it may be surprising that the `Shuffle` component will be instantiated once and must have $m \times r$ connections while the other components like `Map` and `Reduce` have two connections and are instantiated M or R times. However, this design allow to fully encapsulate the protocol used for the *shuffle* phase.

HLCM is a research prototype, both in the model and in the implementation. As such, it has rough edges and is barely usable without being taught by an expert as the syntax can be surprising and the error message hard to understand.

During the development of the MapReduce framework with HLCM, some limitations of the model and implementation have been found and fixed. For instance, the root component to be instantiated that is given on the command line can now have generic arguments and the command line syntax allow to state what is the entry point of the application. The placement of the components into the process is still experimental as it is not yet well understood as how to express the placement constraints in a generic but not cumbersome way. Also, as a non-trivial usage of HLCM, the compiler has shown some performance issues. The time to generate the LAD file for the MapReduce application grows as a polynomial of degree at least 3 with the number of *mappers* and *reducers*. It is estimated that an assembly with 40 *mappers* and 40 *reducers* would take a full month to generate. The fix for this issue is a work in progress as the time of writing. It is thus faster to generate the recurring pattern with another program once the L^2C assembly is defined. Also, some small features are not implemented in HLCM like the command line argument handling or the arithmetics expressions for the properties of the components. Thus in the experiments presented in Chapter 7, the LAD file had to be modified.

4.5 Conclusion

Thanks to a high level component model, a new MapReduce framework has been designed. This framework makes it easy to change the number of *mappers* and *reducers* as well as the *map* and *reduce* operations. Additionally, with the help of a careful parametrization of the components, the commonly modified components are easily customized while still preserving the whole application structure into one component. In Chapter 7, several transfer schedulers are actually put in place of the default one, and some of them require to handle the bandwidth of the transfers in addition to starting them on demand. This kind of modification is made really easy with the MapReduce application designed here and would be a lot of work if an existing MapReduce framework had to be modified.

Chapter 5

Modeling MapReduce

The contributions in Chapters 6 to 7 try to optimize some parts of the execution of a MapReduce application. In order to do so and to study seriously the problems, a model of the system is needed. There are two categories of models that are needed: the platform models that describe the nodes and the network behavior, and the application models that describe the steps of a MapReduce job. Since the following chapters are performance-oriented, the models have to allow to compute the duration of the execution. But they must also keep simple enough to allow for a solution to be easily computed.

This chapter first review in Section 5.1 the different modeling approach that are known to be used and explain why they do not completely fit the mentioned requirement. Section 5.2 explains the global model that is used in the remaining of this document. This model is completed when needed with some additional properties in the given chapters.

5.1 Related Work

Infrastructure Models Despite the fact that the cloud tries to hide the details of the hardware infrastructure, the performance may depend on it. It thus need to be modeled. A cloud platform is usually modeled as a 2 or 3 levels tree network [69, 70]. At the lowest level, the nodes are grouped into racks with a switch for each one. Then the racks are connected together with another switch to make a cluster. The clusters are connected together forming another level. Even though the network tree may look like a fat-tree [13, 71], it is common that the switch to switch links are undersized regarding the actual number of nodes they connect. This practice is called oversubscription. Those models may, however, need to be simplified to solve a given problem.

Network Models On a more specific point, some researchers [72] modeled the effect of contention in an all-to-all operation. Those models are represented by affine functions. In this thesis, contention will not have to be modeled since these models will be used in Chapters 6 to 7 to actually avoid contention.

Coarse MapReduce Models Based on those cloud models, it is possible to build other models to try to predict the duration of applications in the cloud. Some work [73] take the oversubscription into account that may create contention. Others [74] model applications processing jobs and ignore the internal network as they assume the usage of high-throughput low-latency network. Those models are very coarse grain and do not take into account the specificities of MapReduce.

Detailed MapReduce Models On the opposite side, some work model the execution of a MapReduce job in too much details to be easily usable. A research report [75] provide extensive details in modeling the execution of a job in Hadoop MapReduce. It includes 5 steps on the *mapper* side: *read*, *map*, *collect*, *spill* and *merge*. The amount of intermediate data produce is proportional to the amount of input data. On the *reducer* side there are 4 steps *shuffle*, *merge*, *reduce*, *write*. Most of those 9 phases are proportional to the amount of data. Moreover, some parts of the models are quite Hadoop-specific and could be simplified. It could also be noted that the network model used doesn't take contention or latency into account and thus need no assumption on the network topology.

Simulation-Based Modeling MRPerf [76] and Starfish [77] with its *What-If* engine [78] both relies on a simulation to predict the time taken by a MapReduce job. The models implemented in the simulator are very fine grain. They notably take into account the scheduling policy and the data location which are usually unpractical to manipulate as a model.

Specific Models In ARIA [79] an unusual approach is taken to model a MapReduce when the *map*, *shuffle* and *reduce* phases are split into several waves of tasks. Its goal is to allocate the resource to meet a soft deadline as a Service-Level Objective. For each phase, the minimum, maximum, and average time are used as they may have a different impact on the total duration of the job. It also make a special case of the first wave of *shuffle* tasks. The models here also use the average input size of the *map* tasks and two constants that model the ratio of data volume between the input and output of the *map* phase, and between the input and output of the *reduce* phase.

Another approach [80] to model the duration time of the tasks of a MapReduce job is to use a stochastic model to take into account of the variability of the duration time. It can thus produce a probability function of the duration of the *map* and one for the *reduce* tasks. This approach is augmented with the date of arrival of the workers and the duration distribution of the *shuffle* phase. This approach is most useful to simulate the execution of a MapReduce job as long as nothing changes the distribution function of the duration of the tasks. This makes the models hard to use when it comes experiment with new scheduling algorithms.

Some work [81] models the execution of a MapReduce job into the cloud taking into account the first upload from the permanent storage to the cloud storage. The *shuffle* phase seems to be left out, or, at least, not detailed enough. The models use a discrete time interval during which the computation and / or the data transfers occurs. The models used are all linear or affine functions. The goal is to be able to use linear programming to determine an optimal scheduling strategy on several cloud services. However, no follow-up publication actually doing it could be found.

History-Based Models SkewTune [45] and HAT (History-based Auto-Tuning MapReduce) [47] estimate remaining time of the tasks to find stragglers. To this end, they uses historical information to estimate the progress of a task to find the slowest tasks and handle them separately as they probably would slow down the whole jobs execution.

Another interesting approach that uses historical information is to use statistics tools [82]. This work does not actually explicitly models the execution of a MapReduce job. Instead, it uses a statistical framework [83] based on KCCA (Kernel Canonical Correlation Analysis) [84] to estimate the time required to execute a give MapReduce job based on previous runs. To do this, it takes into account two sets of parameters. The job configuration and data characteristics on one side, and several performance metrics on the other side. The algorithm is trained with some measures and then tries to predict the performance metrics based on the job configuration

and data characteristics. This can also be used as a workload generator. The first downside of this method is that it require several runs before giving out accurate results. The second one is that this method is complex to implement and manipulate.

Reliability Models On a side note, none of those models take into account the dependability issue which may occur in large scale platforms or on low cost hardware. Some work explore this issue on the hardware [85] and virtual machine level [86]. The reliability has also been taken into account into broader cloud models [87, 88].

5.2 Defined Models

5.2.1 Platform Models

The considered target platform is a cluster connected by a single switch, forming a star-shaped communication network. Every link connecting a node to the switch has a capacity of C bytes per second and the switch have a bandwidth of σ bytes per second. The network links are assumed to be full-duplex[1]. σ is supposed to be an integer multiple of the link bandwidth. Thus $\sigma = l \times C$. This restriction is only important for some algorithms presented in Chapters 6 and 7. Moreover, the overall bandwidth is assumed to be limited. Meaning that if there are n nodes connected to the switch, then $l \leq n$. Above l concurrent transfers, the communications will suffer from contention, thus degrading the performance.

5.2.2 Application Models

A MapReduce application is represented here by the number of *mapper* processes m, and the number of *reducer* processes r. There is a total volume of data V to process. A *mapper* process i receives a total amount of data α_i, and since all the data has to be processed once and only once, $V = \sum \alpha_i$. The amount of data produced by a *mapper* process is assumed to be linear with the amount of input data with a γ factor. Thus, we call ζ_i the amount of data produced by the *mapper* i, $\zeta_i = \gamma \times \alpha_i$. Moreover, every *mapper* have to send its intermediate data produced to all *reducers* depending on the key. $\zeta_{i,j}$ is the amount of data the *mapper* i have to send to the *reducer* j. Thus, $\zeta_i = \sum \zeta_{i,j}$.

It is also assumed that the intermediate data generated by a given *mapper* process cannot start being sent to the *reducers* before the end of the computation. And a *mapper* i finishes its computation S_i seconds after the first mapper has finished its computation. For the sake of simplicity, it is assumed that the *mappers* are ordered by the date of termination of the computation. Thus, $S_i < S_{i+1}$ for $1 \leq i < m$ and $S_1 = 0$. Figure 5.1 shows the Gantt chart of a possible execution of a MapReduce application following this model. The computation time is green, the transfer time is grey and in red is the idle time.

5.2.3 Performance Models

Given the computation structure of MapReduce, the data is split into chunks of similar size before processing the *map* phase. Thus the processing time of a chunk of data should only depend on the computational power of the node. Thus, it is assumed that each compute node i processes data at a rate of A_i bytes per second. It also assumed that the produced amount of data is linear in the amount of input data.

[1] A full-duplex network link can send and receive a frame at the same time.

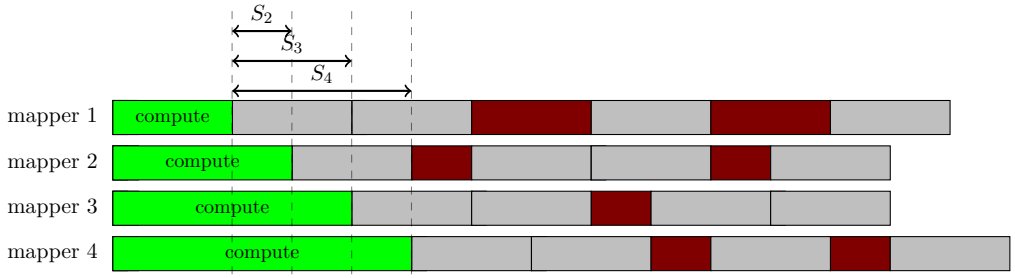


Figure 5.1: Gantt chart of a possible execution following the application model.

As the focus is on the throughput, the data are supposed to be large enough and so that the time to transfer a chunk of data is linear with the size of the chunk. More formally, this means: $time = \frac{datasize}{bandwidth}$.

Thus, the model ignores any latency as well as any mechanism of the network stack that could make the actual bandwidth lower than expected for a short amount of time, such as the TCP slow-start. This network model also ignores any acknowledgment mechanism of the underlying network protocols that can consume some bandwidth and any interaction between CPU usage and bandwidth usage. Therefore, in order to make these assumptions realistic, we choose to map one *mapper* or *reducer* process per physical node when this matters.

5.3 Conclusion

This chapter reviewed the state-of-the-art for modeling a MapReduce application and its performance. A certain degree of precision of the models is needed in order to allow a scheduling algorithm to take decisions minimize the makespan. This makes the non-MapReduce specific models not well suited for this. On the opposite, the models need to be simple enough to allow the computation of some parameters given the others. Thus excluding the stochastic, history-based, and simulation-based models that are hard to manipulate in a formal way.

In order to circumvent these limitations of the existing models, some are proposed in this chapter. Given that the performance model are mostly linear, this allows the formulae to be solved for some parameters, or some metric to be optimized. Moreover, those models still capture the main components of a MapReduce job as they appear in most of the related work.

The defined platform model is indeed very simplistic and does not capture the reality of the current cloud infrastructures. However, being able to deal with a simple platform model is a necessary step before tackling more complex cases. Every time this model is used, an insight is given as to how extend the work to a multi-level platform model.

Chapter 6

Global Performance Optimization

This chapter try to find a solution that would optimize the execution time of the whole MapReduce job at once. For this, a problem description is proposed in Section 6.1 followed by a summary of the existing work around this problem in Section 6.2. Then the platform and performance models presented in Section 6.3 extend, for the purpose of this contribution, the models presented in Section 5.2.1, Section 5.2.2, and Section 5.2.3. The approach of Berlińska and Drozdowski on this problem is then reviewed in Section 6.4 before proposing two improvements in Section 6.5 and Section 6.6. These improvements are evaluated against the work of Berlińska and Drozdowski in Section 6.7 both in terms of quality of the computed schedule and in terms of computation duration. Finally, this chapter is concluded in Section 6.8.

6.1 Problem Description

MapReduce is designed to run on all kind of hardware, from low-end to high-end ones. And actually, Google stated they used it mostly on clusters of commodity hardware. Commodity network hardware has a higher chance of showing contention. However, in a MapReduce application, there are several things that can have an impact on the computing time. Especially, how the data are partitioned among the nodes and the way the transfers are handled during the *shuffle* phase.

Therefore, this chapter proposes to study the proposition of Berlińska and Drozdowski to find a data partitioning and transfer schedule under bandwidth constraints that minimizes the makespan[1], and then improve their solution by relaxing some constraints.

6.2 Related Work

The problem of optimizing a MapReduce job as a whole has rarely been studied. However, some work have focused on either partitioning the data among the *mappers* or reduce the amount of data to transfer between the *mappers* and *reducers*.

The problem of partitioning is linked to the problem of detecting and handling the stragglers, and thus it is the most studied. Meaning that all the solutions are dynamic and partition data on-the-fly, taking into account the actual processing speed of the nodes instead of a theoretical one.

LATE [43] (Longest Approximate Time to End) is not *per-se* a partitioning algorithm. It is a scheduler that refines default progress estimation of the *map* tasks in Hadoop in order to launch a speculative execution for all the tasks that are estimated to finish the farther in the

[1]The makespan is the elapsed time between the start and end of a job.

future. This work is refined by SAMR [89] (Self-Adaptive MapReduce) to take heterogeneity into account. Moreover, SAMR also classifies the nodes based on their history in order to run the backup tasks on nodes that should not be slow.

SkewTune [45] is an extension of Hadoop that also aims at avoiding the stragglers during the *map* phase. To this end, when some resources are available, it checks for the process with the further estimated time of termination whether the overhead for repartitioning is less than the expected time to complete a task. If so, its data are redistributed among the other nodes.

The FAIR [34] scheduler in Hadoop attempts to achieve fairness in a cluster. It does this by splitting the cluster into subclusters and dedicating them to a given pool (or group of users). However, if some nodes remain unused, they may be temporarily reassigned to another pool.

Delay Scheduling [35] is a method to improve locality and fairness in a cluster running Hadoop. It works by delaying the decision of which task should take a free *mapper* slot if no unprocessed task can run with the data locally available on that node. The further the data comes from (rack-local vs. inter-rack access), the greater the delay. This allow for other jobs to release a slot which may make all data access node-local.

The LEEN [36] (locality-aware and fairness-aware key partitioning) algorithm tries to balance the duration of the reduce tasks while minimizing the bandwidth usage during the *shuffle*. This algorithm relies on statistics about the frequency of occurrences of the intermediate keys to get to create balanced data partitions. This approach is complementary to that presented here.

Another complementary approach is the HPMR [48] algorithm. It proposes a *pre-shuffling* phase that lead to a reduced amount of transfered data as well as the overall number of transfers. To achieve this, it tries to predict in which partition the data go into after the *map* phase and tries to place this *map* task on the node that will run the *reduce* task for this partition.

Conversely, the Ussop [46] runtime, targeting heterogeneous computing grids, adapts the amount of data to be processed by a node with respect to its processing power. Moreover it tends to reduce the intermediate amount of intermediate data to transfer by running the *reduce* task on the node that hold most of the data to be reduced. This method can also be used together with our algorithms.

A MapReduce application can be seen as a set of divisible tasks since the data to be processed can be distributed indifferently on the *map* tasks. It is then possible to apply the results from the divisible load theory [90]. This is the approach followed by Berlińska and Drozdowski [4]. The authors assume a runtime environment in which the bandwidth of the network switch is less than the maximum bandwidth that could be used during the *shuffle* phase, thus inducing contention. From those models, they try to fix the data partitioning and the *shuffle* scheduling with a linear program. This approach is further detailed in Section 6.4.

6.3 Specific Models

The models used in this chapter are derived from those presented in Chapter 5, which are also similar to those used by Berlińska and Drozdowski in [4]. To avoid divisions in the linear program underlying their scheduling algorithm, Berlińska and Drozdowski expressed data processing and transfer rates in seconds per byte, even though the most common definitions express such rates in bytes per second. However, for better understanding, those values are converted into bytes per second in this document, although the resolution of the linear program may need these values to be inverted.

In this chapter, there are two additions with respect to the models presented in Chapter 5. First there is a constant delay between the *mappers* **start**, and not the *mappers* **end**. The delay between the beginning of a *mapper* and the next one is S Meaning that the *mapper* i

starts $S_i = S \times (i-1)$ seconds after the first *mapper*. Second, the amount of intermediate data produced by a *mapper* i is equally scattered among the *reducers*. Thus $\zeta_{i,j} = \frac{\zeta_i}{r}$.

Berlińska and Drozdowski also use a notation t_n to denote a given moment during the schedule. As their transfer scheduler works by intervals, the interval n span from t_n to t_{n+1}.

Name	Unit	Description
σ	B/s	Switch bandwidth.
C	B/s	Link bandwidth.
l		$l = \sigma/C$ Ratio between switch and link bandwidth.
m		Number of *mapper* processes.
r		Number of *reducer* processes.
S	sec.	Delay between two consecutive *mapper* starts.
α_i	B	Amount of data to be processes by *mapper* i.
γ		Ratio between the input and output amount of data of the *mappers*.
ζ_i	B	$\zeta_i = \gamma \times \alpha_i$ Amount of data produced by the *mapper* i.
V	B	$V = \sum \alpha_i$ Total amount of data to process.
$\zeta_{i,j}$	B	$\zeta_{i,j} = \zeta_i / r$ Amount of data to be sent from *mapper* i to *reducer* j.
t_n		n-th denoted time.

Table 6.1: Summary of the variables of the model.

The variables and notations and their meaning are reminded in Table 6.1.

6.4 Approach of Berlińska and Drozdowski

In their paper [4], Berlińska and Drozdowski proposed an optimization of both data partitioning and scheduling of communications. To control the network bandwidth sharing and to prevent contention, they decompose the shuffle phase into several steps. During each of these steps, only l concurrent data transfers are performed. Moreover, the order in which these data transfers are done is specified by Constraints (6.1) and (6.2) hereafter. In those equations, $start(i,j)$ stands for the start date of the transfer from *mapper* i to *reducer* j, and $end(i,j)$ stands for the end date of the same transfer.

$$start(i,j) > end(i,j-1) \qquad \forall i \in 1..m,\ \forall j \in 2..r \tag{6.1}$$

$$start(i,j) > end(i-1,j) \qquad \forall i \in 2..m,\ \forall j \in 1..r \tag{6.2}$$

Constraint (6.1) means that a transfer from a map task i to a reduce task j has to wait for the completion of the transfer from this map to the previous reduce task $(j-1)$ before starting, while Constraint (6.2) means that this same data transfer also has to wait for the reduce task j to have complete the transfer from the previous map task $(i-1)$. m and r respectively denote the total numbers of *map* and *reduce* tasks. Figure 6.1 show this ordering as a dependency graph. Each circle represents a transfer which is annotated with a name $i \to j$ that stands for *transfer from* mapper i *to* reducer j.

Those constraints are made to avoid contention on a link level. Indeed, if the same *mapper* had to perform two transfers at the same time, this would decrease the bandwidth usage of the network links on the *reducer* side as contention occurs. Symmetrically, if a *reducer* receives data from two *mappers* at the same time, this would impair the bandwidth usage on the *mapper* side. Even thought a configuration where every *mapper* sends to several *reducer* at the same time may be interesting to study, it would make the problem more complex.

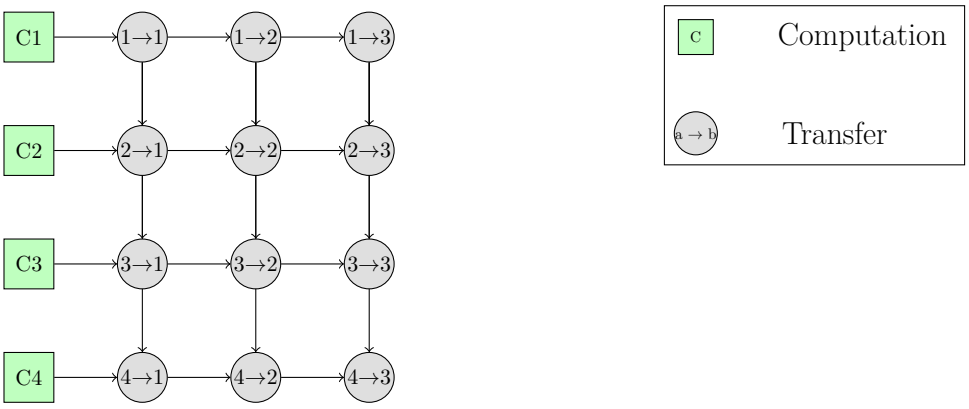


Figure 6.1: Dependency graph between data transfers between 4 mappers and 3 reducers

Table 6.2 illustrates this transfer ordering on a simple example that involves four map tasks and three reduce tasks. In this example, there is no contention on the network. The only constraints to respect are those expressed by Constraints (6.1) and (6.2). Six steps are needed while the number of concurrent data transfers is only limited by the number of reduce tasks.

Step 1	**Step 2**	**Step 3**	**Step 4**	**Step 5**	**Step 6**
$m_1 \to r_1$	$m_1 \to r_2$	$m_1 \to r_3$			
	$m_2 \to r_1$	$m_2 \to r_2$	$m_2 \to r_3$		
		$m_3 \to r_1$	$m_3 \to r_2$	$m_3 \to r_3$	
			$m_4 \to r_1$	$m_4 \to r_2$	$m_4 \to r_3$

Table 6.2: Data transfers ordering between four map tasks and three reduce tasks without contention on the network switch.

The algorithm proposed by Berlińska and Drozdowski has been designed for configurations in which the network switch becomes a performance bottleneck. The network has capacity of l concurrent transfers. Although Constraints (6.1) and (6.2) guaranty the absence of contention on the links, contention may still occur on the switch. Thus, the solution proposed by Berlińska and Drozdowski is to allow the transfers to start in a round robin fashion. During every step, the *mappers* $i..(i+l)$ perform their transfer. As soon as they all finish, *mappers* $(i+1)..(i+1+l)$ perform their transfers, and so on.

In order to model this, Berlińska and Drozdowski introduced a function itv that computes for every transfer $i \to j$ the interval number in which it is scheduled. This function is defined in Equation (6.3) that takes as argument the id of the *mapper* and *reducer* involved in the transfer for which to compute the interval number. This equation also depends on the number of *mapper* m, and the maximal number of concurrent transfers l. Note that $itv(m, r)$ corresponds to the last transfer to be performed.

$$itv(i,j) = \left(\left\lceil \frac{j}{l} \right\rceil - 1\right) m + i + (j-1) \mod l \qquad \forall i \in 1..m, \forall j \in 1..r \tag{6.3}$$

Table 6.3 details the communication phase in a setting similar to that of Table 6.2, but with $l = 2$. This means that at most two data transfers can be done concurrently. With this additional constraint, eight steps are now needed to transfer all the data produced by the map tasks to the reduce tasks.

Equation (6.4) defines vti as the reciprocal function of itv. For a given step s, it returns the set of *mapper* that perform a data transfer during this step.

Step 1	**Step 2**	**Step 3**	**Step 4**	**Step 5**	**Step 6**	**Step 7**	**Step 8**
$m_1 \to r_1$	$m_1 \to r_2$			$m_1 \to r_3$			
	$m_2 \to r_1$	$m_2 \to r_2$			$m_2 \to r_3$		
		$m_3 \to r_1$	$m_3 \to r_2$			$m_3 \to r_3$	
			$m_4 \to r_1$	$m_4 \to r_2$			$m_4 \to r_3$

Table 6.3: Data transfer ordering between four map tasks and three reduce tasks with a limit of two concurrent transfers at a given time step.

$$vti(s) = \{a | itv(a, b) = s, b \in 1..r\} \tag{6.4}$$

Defining the order and in which step data transfers between map and reduce tasks have to be performed is not enough to build a good schedule of the shuffle phase. The initial partition of the data set among the *mappers* also has an impact on the shuffle as it defines its start time and the sizes of the exchanged messages. Berlińska and Drozdowski aims at minimizing the overall completion time of the shuffle phase, i.e. the end of the last data transfer. This is achieved using to the following linear program.

$$\begin{array}{l} \text{MINIMIZE } t_{itv(m,r)+1}, \\ \text{UNDER CONSTRAINTS} \\ \begin{cases} \forall i \in 1..m,\ iS + \dfrac{\alpha_i}{A_i} = t_i \\ \forall i \in 1..itv(m,r), \forall k \in vti(i),\ \dfrac{\alpha_k \gamma}{rC} \leq t_{i+1} - t_i \\ \displaystyle\sum_{i=1}^{m} \alpha_i = V \end{cases} \end{array} \tag{6.5}$$

The first constraint ensures that a *mapper* cannot start to send data to a *reducer* before the completion of the computations it has to execute. The inequality in the second constraint indicates that the duration of an interval has to be long enough to allow the completion of all the transfers scheduled in this step. Finally, the sum in the last constraint ensures that the whole data set is processed by the *mappers*. The outputs of this linear program are the amount of data α_i to be processed by each *mapper*, and the date t_k bound of the intervals.

Figure 6.2 shows an example of schedule produced by the algorithm proposed by Berlińska and Drozdowski for four *mapper* processes, four *reducer* processes, and a limit of two concurrent data transfers. We can see that, due to the start-up time S, the map tasks are launched sequentially. The first constraint of the linear program leads to schedules in which the first transfer issued by a map task ends when the next map tasks completes its computations.

The sequential startup time S is shown in white, the computation time of the *mappers* is shown in green, the data transfers to every *reducer* i are shown in gray, and the idle times are shown in red. The light red is the idle time induced by the limit to l concurrent transfers, while the dark red is the idle time induced by the *per-wave* property of the scheduling. The reduce tasks are not shown there because they do not influence the schedule produced.

This algorithm has two major drawbacks that are addressed in the next two sections. First, the use of a costly linear program to determine the partitioning of the data in a way to optimize the length of the shuffle phase becomes intractable as the size of the problem, i.e. the number of map and reduce tasks, grows. Its complexity is that of the simplex algorithm with $\mathcal{O}(mr)$ constraints. It is exponential in the worst case, but polynomial in many cases. For certain configurations, this linear program may fail to find a solution at all. Secondly, the scheduling

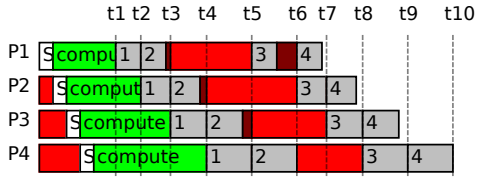


Figure 6.2: Schedule produced by the seminal algorithm proposed by Berlińska and Drozdowski.

of data transfers in steps leads to an important amount of idle times on the nodes, as shown in Figure 6.2.

6.5 Partitioning: From Linear Program to Linear System

One property that can be observed from the results of the schedule produced by the algorithm of Berlińska and Drozdowski is that the first transfer issued by the *mapper* i ends when the *mapper* $i+1$ completes its assigned computations. This is a consequence of the order constraints given by Constraints (6.1) and (6.2). It can be reformulated as follows.

$$iS + \frac{\alpha_i}{A_i} + \frac{\alpha_i \gamma}{rC} = (i+1)S + \frac{\alpha_{i+1}}{A_{i+1}} \tag{6.6}$$

It has been observed that this holds as long as α_i is less than α_{i+1}. On a homogeneous cluster, where all the nodes have the same processing power, (i.e. all the A_i have the same value), Berlińska and Drozdowski have observed $\alpha_i < \alpha_{i+1}$ to be equivalent to:

$$Srm < \frac{\gamma V}{C} \tag{6.7}$$

Indeed, when $A_i = A$ and $\alpha_i = \alpha = \frac{V}{m}$, then Equation (6.6) imply that $\frac{\alpha\gamma}{rC} = S$, which is equivalent to $Srm = \frac{\gamma V}{C}$. Starting from this, Berlińska and Drozdowski have observed that when $Srm < \frac{\gamma V}{C}$ then $\alpha_i < \alpha_{i+1}$ and conversely.

However, the startup time S (in the order of a few seconds) is often less than the time to transfer all the data (in the order of several terabytes). Thus the condition $\alpha_i < \alpha_{i+1}$ is commonly filled, which means that Equation (6.6) would also hold. Under these conditions, the partitioning can be computed by the following linear system which would compute the same values for α_i as the linear program 6.5 provided that $\alpha_i < \alpha_{i+1}$.

$$iS + \frac{\alpha_i}{A_i} + \frac{\alpha_i \gamma}{rC} = (i+1)S + \frac{\alpha_{i+1}}{A_{i+1}} \qquad \forall i = 1..m-1 \tag{6.8}$$

$$\sum_{i=1}^{m} \alpha_i = V \tag{6.9}$$

The complexity of this linear system is in $\mathcal{O}(m)$, that is less than the resolution of a linear program. Moreover, except for roundoff errors, the resolution of the linear system leads to the same partitions of the computations, i.e. determines the same values of α_i. This is illustrated in Figure 6.3 in which the partition of the data between the *mappers* is determined by the proposed linear system, while the effective scheduling of the data transfers is done using the static strategy proposed by Berlińska and Drozdowski. Note that determining the partition also sets the start time of the first four steps of the shuffle phase t_1 to t_4.

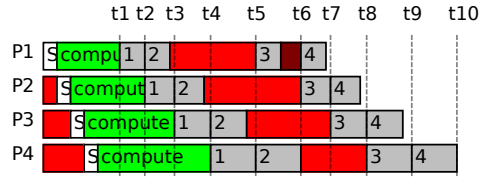


Figure 6.3: Schedule produced by the linear system.

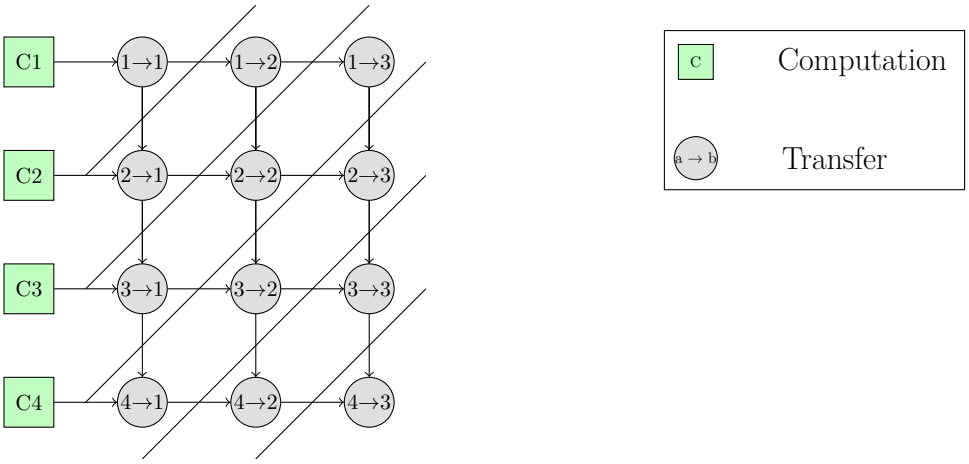


Figure 6.4: Dependency graph between the transfers with diagonals.

6.6 Data transfer Scheduling: From Static to Dynamic

In this section we propose to schedule data transfers dynamically instead of building a static schedule in several steps as done by Berlińska and Drozdowski. Before introducing this new scheduling method, recall that network contention can occur not only on the switch of limited capacity, but also on the links that connect the compute nodes to this switch. That's why the algorithm proposed here has been designed to respect the constraints expressed by Constraints (6.1) and (6.2) similarly to Berlińska and Drozdowski. These constraints prevent any compute node to perform more than one transfer at a time either on the sender or receiver side, and thus avoid contention on the links.

These constraints only force a partial order on the transfers. A scheduler still have to chose which transfers to start when several transfers are ready. The intuition would say it is better to keep a maximum number of active transfers at the same time in order to maximize the network usage and minimize the completion time. However, as suggested in Figure 6.1, Constraints (6.1) and (6.2) force a delay of one transfer between the sender transfers. That's why the heuristic proposed here is to keep the active transfers on a diagonal as shown on Figure 6.4.

More formally, a *reverse priority* ρ is assigned to every transfer $i \to j$ such that $\rho_{i\to j} = i + j$. The lower the value of ρ, the higher the priority. However this priority only comes into play when the dependencies of a transfer are met.

Algorithm 1 present the proposed strategy. In this algorithm, *node.state* hold the representation of the current activity of the node, and *node.target* hold the identifier of the *reduce* to which *node* is transferring or will perform its next transfer. When the transfer from a *mapper* i to a *reducer* j ends, then for each idle *mapper* i' and its next target *reducer* j', the priority $p_{i'} = i' + j'$ is computed. Then the nodes that minimize $p_{i'}$ is selected. These nodes are considered to be the *most late* and their transfers have to start as soon as possible. This promotes

Algorithm 1 Transfer scheduling algorithm

```
1: procedure REQUEST_TRANSFER(node)
2:     if the network link of the target reducer is busy or the limit of the switch has been reached then
3:         node.state ← IDLE
4:     else
5:         node.state ← TRANSFER
6:         START_TRANSFER(node, node.target)
7:     end if
8: end procedure
9: procedure ON_COMPUTE_END(node)
10:    REQUEST_TRANSFER(node)
11: end procedure
12: function NODE_TO_WAKE
13:    for all N node in IDLE state do
14:        if target reducer's network link is busy then
15:            continue with next node
16:        end if
17:        p[N] ← number of N + number of N.cible
18:    end for
19:    if p is empty then
20:        return undefined value
21:    else
22:        return N for which p[N] est is the lowest
23:    end if
24: end function
25: procedure ON_TRANSFER_END(node)
26:    n ← NODE_TO_WAKE
27:    if n is not undefined then
28:        REQUEST_TRANSFER(n)
29:    end if
30:    if node hasn't done every transfers then
31:        node.target ← next node
32:        REQUEST_TRANSFER(node)
33:    else
34:        node.state ← TERMINATED
35:    end if
36: end procedure
```

maximization of bandwidth usage and allow not to break Constraint (6.2) without stating it explicitly in the algorithm.

Procedure ON_COMPUTE_END is called as soon as a *map* task finishes to process its data. It calls the procedure REQUEST_TRANSFER that will start the requested transfer if that does not violate the bandwidth constraints. The procedure ON_TRANSFER_END is called when a transfer ends. It start by launching the transfer with higher priority if it exists. Then it starts the next transfer of the node that has just terminated if that is possible.

This algorithm enforce the constraints on the network usage while using it at its maximum at any time. Indeed, if the bandwidth of the switch was already fully used, then, the only call to REQUEST_TRANSFER that will actually start a transfer is the one on Line 28. Furthermore, if the order defined by Constraints (6.1) and (6.2) prevented a transfer to start, then the switch is not fully used and the termination of one transfer may start 2 new transfers at most. That's what the calls to REQUEST_TRANSFER on Line 28 and Line 32 do.

6.7 Comparative Evaluation

This section evaluates our algorithm by comparing it to the one proposed by Berlińska and Drozdowski. For that, a small simulator has been written in Perl that implements both algorithms. The decision has been made not to use an existing simulator because it was easier and faster to implement the models presented in Chapter 5 and Section 6.3, and to implement several scheduling policies in a custom tool rather than to struggle with the existing tools to only keep a minimal part.

This simulator is event-driven. It has a queue of event which is initially filled with the events of computation termination. And every time an action is taken, new future events may be generated. For instance, when a transfer finishes, a new transfer may start, in which case, its termination date is computed, and the future event is added to the list of future events. The resolution of the linear program is implemented with `lp_solve` while the resolution of the linear system is directly implemented in Perl.

These algorithms are compared in two experiments. The first experiment presented in Section 6.7.1 sets the number of concurrent transfers allowed by the network switch and varies the number of nodes. The second experiment presented in Section 6.7.2 sets the number of nodes and varies the number of concurrent transfers supported by the switch.

Table 6.4 lists the parameters related to the platforms and applications that are used in both following experiments. In these experiments, the platforms are considered homogeneous with a processing power $A = A_i, \forall i$. It is also assumed that the total amount of intermediate data produced is equal to the amount of data processed, hence $\gamma = 1$.

Common parameters			
A	250 MB per second	C	125 MB per second
S	1 second	V	10 To
	Experiment 1		**Experiment 2**
l	50 concurrent transfers		from 50 to 300 concurrent transfers
m and r	from 50 to 300 nodes		300 nodes

Table 6.4: Experiments parameters summary.

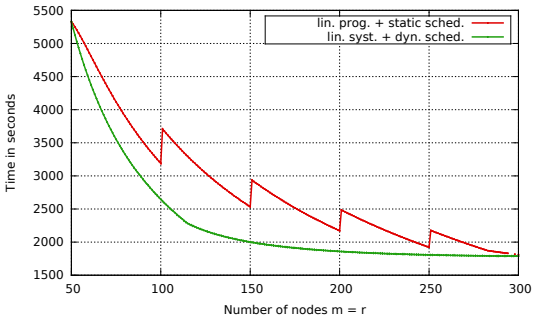


Figure 6.5: Termination date of the last transfer w.r.t m and r.

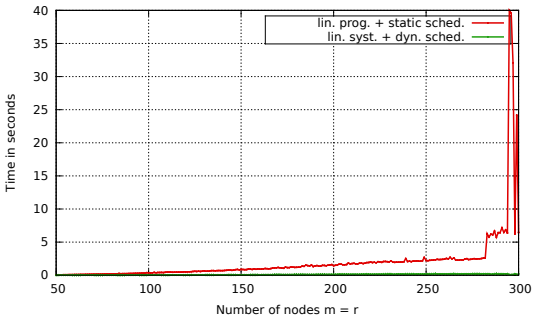


Figure 6.6: Partitioning duration w.r.t m and r.

6.7.1 Varying the Number of Nodes

In the first experiment, the maximum number of concurrent transfers that do not induce contention on the switch l, is fixed, and the number of *mappers* is increased gradually. The number of *mapper* is equal to the number of *reducers* i.e. $m = r$. Figure 6.5 shows the execution time in seconds since the launch of the application up to the termination of the last transfer with respect to the number of *mapper* processes. Equation (6.7) applied with the chosen parameters say that above 282 nodes, the condition $\alpha_i < \alpha_{i+1}$ no longer hold. The execution time for less than 50 nodes is not shown because in this case the bandwidth of the switch is not fully used and both approaches result in the same makespan.

It can be seen that the approach of Berlińska and Drozdowski results in a completion time that is globally decreasing by intervals. These discontinuities are due to phase-based algorithms. Indeed, when the number of transfers to be performed for every node is a multiple of l, then the bandwidth will be fully used at some point in every phase. On the other hand, if there is at least one more node, then one more phase is necessary, and this one will under-utilize the resources. There is no such phenomenon with our algorithm that does not force phase-based execution while still guaranteeing the congestion avoidance of the switch.

Moreover, the linear program and the phase-based scheduler from Berlińska and Drozdowski always produces longer schedules than the linear system with our dynamic scheduler. The maximal gain has been obtained for 151 nodes and is around 47%.

Figure 6.6 shows the time for resolution of the linear program and the time of the linear system with respect to the configuration presented in Table 6.4. These measurements have been conducted on a computer with an *Intel core i5* processor at 2.40GHz and 3Go RAM. The resolution of the linear program is done by `lp_solve`. The resolution of the linear system is hard-coded into the simulator.

On this figure it can be sees that the resolution time of the linear program increase linearly until 282 nodes, which means as long as $\alpha_i < \alpha_{i+1}$. Above this limit, the solver takes a lot more time and does not always finds a solution. On this figure, every resolution that takes more than 10 seconds ended with a failure. The proposed linear system never required more than a few tenth of seconds to find a partitioning that produce an efficient scheduling of the data transfers.

We tried to push the limits of this experiment until $r = m = 1000$. However, the failure rate was above 90% when the number of processes is above 300, which makes the results insignificant. Nevertheless, the linear system and its dynamic scheduler keep their asymptotic behavior.

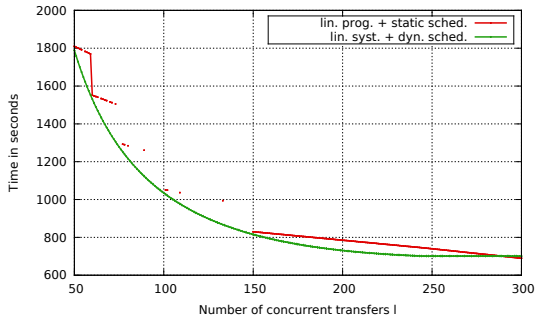


Figure 6.7: Termination date of the last transfer w.r.t l.

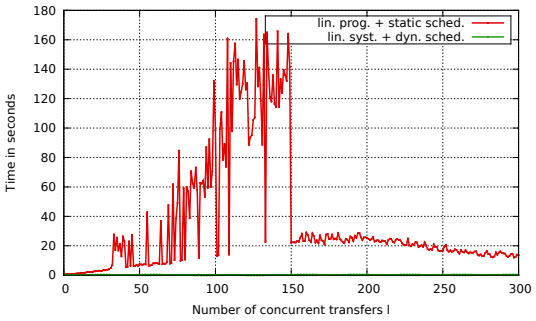


Figure 6.8: Partitioning duration w.r.t l.

6.7.2 Varying the Number of Concurrent Transfer Limit

In the second experiment, the number of nodes is fixed, and the number of concurrent transfers that do not induce congestion is increased. Figures 6.7 and 6.8 show respectively the execution time and the resolution time of the linear program and the linear system.

It can be seen that the execution time induced by the linear program decreases by segments. These discontinuity are due to the phase-based algorithm that can – sometime – eliminate a phase when l increase. Our linear system with its dynamic scheduler results in continuously decreasing makespan. Figure 6.8 shows that the time to resolve the linear program is still larger and unstable. In a lot of cases exceeding several minutes, the resolution is even impossible. On the other hand, the resolution time of the linear system is negligible.

6.8 Conclusion

These algorithms focus on optimizing the *shuffle* phase of a MapReduce application. For this, a linear system with a dynamic transfer scheduler has been proposed. When compared to the approach of Berlińska and Drozdowski based on a linear program and a static phase-based scheduler, our approach shows that it produces shorter schedules, takes less time to compute in addition to being more stable and with a better scalability.

However, this approach is only tested on simulator and not on a real hardware. And even if the performance of `lp_solve` can be questioned with respect to the other linear program solvers available. It is hardly conceivable that one of them can be faster than a solving a linear system since solving a linear program is strictly harder than solving a linear system. Moreover, there is no guaranty of performance comparison to an optimal solution or to a lower bound. These issues are addressed in the next chapter when studying several alternative algorithms for scheduling the shuffle phase alone.

Chapter 7

Shuffle Phase Optimization

7.1 Introduction

Among the 3 phases of MapReduce, the *reduce* phase usually takes a negligible amount of time compared to the two others. Most works [34, 35, 43, 45, 89] that try to optimize MapReduce have mainly focused their efforts on the *map* phase, improving the data-computation locality, computation balance between node, or fault tolerance. But, despite being an important phase, the *shuffle* has been largely forgotten.

In order to optimize the performance / cost ratio, most MapReduce platforms run on moderately high-end commodity hardware. With today's technologies (as of 2013), common hard disk drives can achieve a throughput of more than 170 MB/s on a 7200 rpm HDD. RAID configurations and SSD drives can lead to much higher throughput. The CPU can also achieve a high data throughput. For instance, a simple `md5sum` reading from a shell pipe can achieve a throughput of 372 Mo/s. As most *map* tasks involve a much simpler computation than a md5, it can be assumed that the CPU is not the bottleneck of a MapReduce application.

However, the network throughput is usually bound to 1 Gbps (or 125 MB/s) on commodity hardware. From this, the time taken by the *map* phase is expected to be more or less or equivalent to that of the *shuffle* phase if the *map* operation is optimized enough and generates an amount of data of the same order of magnitude as its input data.

Network bandwidth can become a scarce resource on some platforms. Indeed, most non high-end network equipments cannot guaranty that the overall sustained throughput would be equal to the sum of the throughput of all its connected ports. Some previous work [72] showed that when contention occurs in a LAN, the overall throughput drops because of the delay needed by TCP to detect and retransmit the lost packets. Moreover, the well-known Cisco System corporation also sells, for instance, 10 Gb Ethernet switches that do not provide a backplane bandwidth equal to the sum of the bandwidth of all the ports. Thus it is asserted that, in general, the overall bandwidth is limited.

7.2 Problem Description

In a MapReduce application, it is quite common that the *mappers* do not process the same amount of data and that the *map* processes do not terminate at the same time. Thus, sharing the bandwidth equally among the mappers (as the network stack would do by default) may lead to a suboptimal bandwidth usage due to the mappers that finished later and those with more intermediate data. How should the *shuffle* phase be managed to make it as quick as possible?

This chapter proposes and compares several algorithms to optimize the *shuffle* phase. Section 7.3 review some works that try to optimize the *shuffle* phase in MapReduce. The models

presented in Chapter 5 are analyzed to produce a lower bound on the duration of the *shuffle*. Then, 6 algorithms are presented in Section 7.4.2. In Section 7.4.3 an algorithm to regulate the bandwidth is presented, this part is actually used by 3 of the algorithms. Section 7.6 evaluate the accuracy of the platform simulation, the regulation algorithms presented before and all the 6 algorithms under various conditions. And finally Section 7.7 conclude this work and propose some ideas for further research.

7.3 Related Work

The previous work that investigated this problem have already been presented in Section 6.2 and are only briefly reminded here.

The LEEN [36] algorithm and HPMR [48] both try to balance the reduction partition in order to avoid having one *reducer* that takes significantly longer than the others. As a result, the duration of the transfers of the *shuffle* phase are also balanced. The approach taken is however different. LEEN tweaks the partition function while HPMR acts on the input of the *map* phase. Conversely, the Ussop runtime [46] reduces the amount of data sent during the *shuffle* phase by processing the reduction on the node that has already most of the data locally.

And finally, Berlińska and Drozdowski modeled a MapReduce application using the divisible load theory [90]. This leads to a linear program where the input data partitioning and some parameters of a static transfer schedule for the *shuffle* phase can be optimized to reduce the total time.

7.4 Shuffle Optimization

7.4.1 Model Analysis

Variable	Unit	Description
σ	B/s	Switch bandwidth.
C	B/s	Link bandwidth.
l		$l = \sigma/C$ Ratio between switch and link bandwidth.
m		Number of *mapper* processes.
r		Number of *reducer* processes.
S_i	s	Delay between the start of the first and the i-th *mapper*.
ζ_i	B	Amount of data produced by *mapper* i.
V	B	$V = \sum \zeta_i$ Total amount of data to transfer.
$\zeta_{i,j}$	B	$\zeta_i = \sum \zeta_{i,j}$ The amount of data to be sent from *mapper* i to *reducer* j.

Table 7.1: Summary of the variables of the model.

From the models given in Chapter 5 some properties can be derived which may prove useful regarding the optimization of the duration of the *shuffle* phase. Namely, a sufficient condition of optimality for a transfer scheduling algorithm can be defined, as well as a lower bound for the duration of the *shuffle* phase. As a reminder, Table 7.1 summarizes the notations and variables of the models used. Please pay attention to the variable V that here refers to the total volume of data to transfer and not to the amount of data to process as it was the case in Chapters 5 and 6.

7.4.1.1 Sufficient Conditions of Optimality and Lower Bound

As a metric, what we try to optimize is the time between the start of the first transfer and the end of the last transfer. Indeed, in the general case, a *reduce* task cannot start before all the data are available. Thus, all the *reduce* tasks starts almost at the same time, which is the end of the *shuffle* phase.

Sufficient condition From the chosen models in Chapter 5, we can derive a few properties of an optimal algorithm. It would be trivial to prove that an algorithm that uses all the available bandwidth from the beginning to the end of the *shuffle* phase would be optimal, which means that all the transfers would have to end at the same time. This is not a necessary condition for an algorithm to be optimal since, in some cases, these requirement cannot be met. But this is sufficient to make an algorithm optimal. See Figure 7.1 for a representation of the bandwidth usage through time of the case that fulfill the sufficient condition.

Lower bound This sufficient condition allows to compute a lower bound of the *shuffle* duration. This lower bound is divided in two parts. The first part t_1 is when the overall bandwidth is limited by the number of nodes transferring with a bandwidth C. The second part t_2 is the period when the bandwidth is actually limited by the switch to σ octets per second.

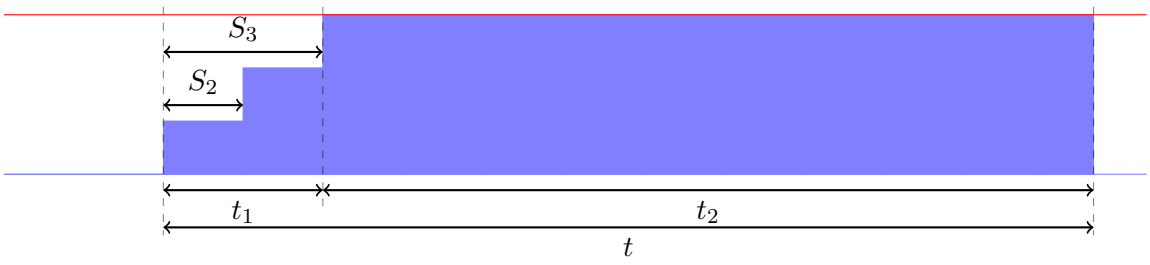


Figure 7.1: Lower bound calculation based on the bandwidth usage.

$$
\begin{aligned}
t &= t_1 + t_2 \\
t_1 &= S_l \\
t_2 &= \frac{V - C\sum_{i=1}^{l-1}(S_l - S_i)}{\sigma}
\end{aligned}
$$

As a special case when $S_{i+1} = S_i + \Delta_S \ \forall i \in [1..l-1]$, t_2 can be simplified further. This case is the one we had in the experiments presented in Section 7.6.4 and Section 7.6.6 and the following formula is the one used as lower bound for comparison.

$$
\begin{aligned}
t_1 &= \Delta_S(l-1) \\
t_2 &= \frac{V - C\sum_{i=1}^{l-1} i \times \Delta_S}{\sigma} \\
&= \frac{V - C \times \Delta_S \sum_{i=1}^{l-1} i}{\sigma} \\
&= \frac{V}{\sigma} - \frac{C \times \Delta_S \times l(l-1)}{2\sigma} \\
&= \frac{V}{\sigma} - \frac{C \times \Delta_S \times l(l-1)}{2 \times l \times C} \\
&= \frac{V}{\sigma} - \frac{\Delta_S(l-1)}{2}
\end{aligned}
$$

Then the expression of t simplifies also.

$$
\begin{aligned}
t &= t_1 + t_2 \\
&= \Delta_S(l-1) + \frac{V}{\sigma} - \frac{\Delta_S(l-1)}{2} \\
&= \frac{V}{\sigma} + \frac{\Delta_S(l-1)}{2}
\end{aligned}
$$

This lower bound can be interpreted as the optimal time to transfer all the data through the switch plus a term that represents the non-full usage of the bandwidth during the start of the *shuffle*. It should be noted that this lower bound do not depend on the individual intermediate data size ζ_i, it only depends on the total amount of intermediate data and on the network characteristics.

7.4.2 Algorithms

In order to try to maximize bandwidth usage during the *shuffle* phase, several algorithms are evaluated. The first one is the simplest algorithm we could imagine, and probably the one implemented in every framework. It just starts every transfer as soon as the intermediate data are available. Then two discrete algorithms that are either based on a partial order of the transfers or on two ordered lists are presented. And eventually, three algorithms based on bandwidth regulation are presented. They only differ on the way the bandwidth is allocated to every transfer.

7.4.2.1 Start-ASAP Algorithm

As reference, the simplest algorithm considered is that which consists in starting every transfer as soon as the intermediate data are available. It thus relies on the operating system to share the bandwidth between every transfer from a single node. On average, the system shares the bandwidth equally among the transfers. It also relies on the switch to share the bandwidth fairly among the nodes it interconnects in case of contention.

7.4.2.2 Order-Based Algorithms

The second algorithm is the one proposed in Section 6.6. This algorithm was designed to work with the partitioning algorithm presented in Section 6.5, but this part is left out here and the transfer scheduling algorithm only is evaluated. As a reminder, this algorithm orders the transfers in a kind of grid and only start a transfer when the other transfers it depends on are finished.

7.4.2.3 List-Based Algorithm

In order to overcome the limitations of the order-based algorithm, it has been decided to break the dependencies defined by Constraints (6.1) and (6.2) between transfers, and allow to reorder them. However, constraints remain that there must never be two transfers at the same time from a single *mapper* (7.1) or toward a single *reducer* (7.2) as this would create some contention and degrade the performance.

$$start(i,j) \geq end(i,j') \vee end(i,j) \leq end(i,j') \; \forall i \in [1..m], j, j' \in [1..r], j \neq j' \tag{7.1}$$

$$start(i,j) \geq end(i',j) \vee end(i,j) \leq end(i',j) \; \forall i, i' \in [1..m], j \in [1..r], i \neq i' \tag{7.2}$$

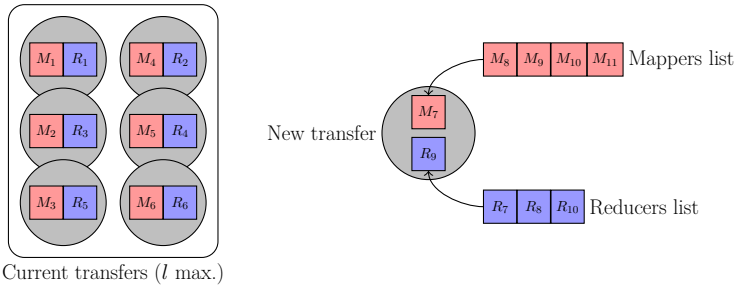


Figure 7.2: Overview of the list-based algorithm.

Algorithm In order to fulfill Constraints (7.1) and (7.2) this algorithm handles the *reducers* in a list from which a *reducer* will be taken from and associated to a *mapper* to form a couple that represents a transfer. Figure 7.2 show an overview of the way this algorithm work. When a *mapper* is ready to run a transfer, the algorithm iterates through the list and take the first *reducer* that the current *mapper* has not transfered its data to, yet. Once found, the reducer is removed from the list. This *reducer* will be put back to the end of the list when the transfer is finished, allowing another *mapper* to transfer its data to this *reducer*. Thus, this list of *reducers* will be kept ordered by the time their last transfer finished.

It may happen that for a given *mapper* there is no *reducer* it has not already transfered its data. In that case, another *mapper* may be waiting for a transfer, and it is given a chance to start a transfer with the same method. Thus the *mappers* are handled in another list. When there are less than l transfers running at the same time, the first element of the *mappers* list is taken and the algorithms search for a *reducer* to run a transfer to from this mapper as previously described. If no *reducer* can be found, the next *mapper* in the mapper list is taken, and so on. When a *mapper* has been found, it is removed from the list. And when a *mapper* finished its transfer, it is put back in the list if it has not finished all its transfers. But, it is inserted in the list so that the list is ordered by the number of transfers left to be started for every *mapper*. The more a *mapper* has transfers to run, the earliest it is in the list.

More formally, that means that there are two lists named ml and rl. ml is empty in the beginning and will contain only the id of the *mappers* that finished their *map* computation and still have some intermediate data to transfer. ml is assumed to be automatically ordered by the number of remaining transfers, like a priority queue. rl contains the id of every *reducer* at the beginning, and is only ordered by the fact that the *reducer* id will be always be enqueued at the end. Algorithm 2 defines a function TRANSFER_CHOICE that returns a pair of *mapper* and *reducer* representing the transfer to be started. This procedure makes explicit the way a transfer is chosen among the remaining transfers.

Algorithm 2 List-based transfer scheduling algorithm.

```
1: function REDUCER_CHOICE(i)
2:     for all j ∈ rl do
3:         if i has some data to transfer to j then
4:             rl ← rl \ {j}
5:             return j
6:         end if
7:     end for
8:     return ∅
9: end function
10: function TRANSFER_CHOICE
11:     for all i ∈ ml do
12:         j ←REDUCER_CHOICE(i)
13:         if j ≠ ∅ then
14:             ml ← ml \ {i}
15:             return (i, j)
16:         end if
17:     end for
18:     return ∅
19: end function
20: procedure START_TRANSFERS
21:     t ←TRANSFER_CHOICE
22:     while t ≠ ∅ and the number of concurrent transfers ≥ l do
23:         start transfer t
24:         t ←TRANSFER_CHOICE
25:     end while
26: end procedure
27: when mapper p finishes its computation
28:     ml ← ml ∪ {p}
29:     START_TRANSFERS
30: end when
31: when Transfer p → q finish
32:     if p still have some data to transfer then
33:         ml ← ml ∪ {p}
34:     end if
35:     rl ← rl ∪ {q}
36:     START_TRANSFERS
37: end when
```

Limitations Although we expect this algorithm to perform better than the order-based algorithm, some corner cases may still remain. Indeed, it may happen a *mapper* finishes a transfer and no other transfer can start because all the *reducers* that have data to receive are already managing another transfer. Thus leading to a suboptimal usage of the bandwidth. We can expect that this situation is more common when l is almost as large as r. Indeed, when l is close to r, most *reducers* will be involved in a transfer at any given time, thus reducing the possible choice for a target *reducer*.

7.4.2.4 Per-Process Bandwidth Regulation

Unlike the previous discrete algorithms, we also propose three algorithms based on the bandwidth regulation of every data transfer. The idea behind those comes from the sufficient condition of an optimal algorithm that either uses all the bandwidth of the switch or all the bandwidth it can use on the links use and that makes all the transfers finish at the exact same time.

For this, we assume that for a given data transfer, a given bandwidth can be maintained. We also assume that this bandwidth can be modified dynamically. The way we achieved this is explained in Section 7.4.3. We first propose an algorithm based on a per-process bandwidth regulation.

This first algorithm divides the switch bandwidth σ among the *mapper* processes. The amount of bandwidth allocated to a given mapper is computed so that its transfer will finish at the same time as the others.

Model addition For this algorithm, the models described in Chapter 5 and reminded in Table 7.1 need to be completed. Every *mapper* is assumed to have the same amount of data to transfer to every *reducer* process.

$$\zeta_{i,j} = \frac{\zeta_i}{r} \; \forall j \in [1..r] \tag{7.3}$$

$ready(t)$ is the set of *mapper* processes that have finished their computations and have not yet finished to transfer their intermediate data at a date t. $\beta_{i,j}(t)$ is the bandwidth allocated to the transfer from *mapper* i to *reducer* j at a date t, and $\beta_i(t) = \sum \beta_{i,j}(t)$ the bandwidth allocated to a given *mapper* process i. And because of assertion (7.3), we also have $\beta_{i,j}(t) = \beta_i(t)/r$. Is also define $\zeta_i(t)$ the amount of intermediate data a ready *mapper* i still have to transfer to the reducers at a date t.

Algorithm $\beta_i(t)$ can be computed by solving the following system.

$$\frac{\zeta_i(t)}{\beta_i(t)} = T \qquad \forall i \in ready(t) \tag{7.4}$$

$$\sum_{i \in ready(t)} \beta_i(t) = \sigma \tag{7.5}$$

The system (7.4) – (7.5) can be reformulated as follows.

$$\frac{\zeta_i(t)}{\beta_i(t)} = \frac{\zeta_j(t)}{\beta_j(t)} \qquad \forall i \in ready(t) \wedge \text{ any } j \in ready(t) \wedge i \neq j$$

$$\sum_{i \in ready(t)} \beta_i(t) = \sigma$$

And then as:

$$\zeta_i(t)\beta_j(t) = \zeta_j(t)\beta_i(t) \qquad \forall i \in ready(t) \wedge \text{ any } j \in ready(t) \wedge i \neq j \tag{7.6}$$

$$\sum_{i \in ready(t)} \beta_i(t) = \sigma \tag{7.7}$$

The later is clearly a linear system that can be solved in linear time with the number of *mappers* $\mathcal{O}(m)$. However, a trivial solution exists to this system. Let's write $V(t) = \sum_{i \in ready(t)} \zeta_i(t)$ as a shorthand. Then a solution is:

$$\beta_i(t) = \sigma \frac{\zeta_i(t)}{V(t)} \; \forall i \in ready(t)$$

The verification of this solution is quite simple. The solution for $\beta_i(t)$ can be replaced in Equation (7.4) and it can be seen that the value doesn't depend on i.

$$\begin{aligned} \frac{\zeta_i(t)}{\beta_i(t)} &= \frac{\zeta_i(t)}{\sigma \frac{\zeta_i(t)}{V(t)}} \\ &= \frac{\zeta_i(t)V(t)}{\sigma\zeta_i(t)} \\ &= \frac{V(t)}{\sigma} \end{aligned}$$

Similarly, the value for $\beta_i(t)$ can be replaced in the left hand side of Equation (7.5) and it can be seen that it is equal to σ as expected.

$$\begin{aligned} \sum_{i \in ready(t)} \beta_i(t) &= \sum_{i \in ready(t)} \sigma \frac{\zeta_i(t)}{V(t)} \\ &= \frac{\sigma}{V(t)} \sum_{i \in ready(t)} \zeta_i(t) \\ &= \frac{\sigma}{V(t)} V(t) \\ &= \sigma \end{aligned}$$

Solving the linear system (7.6) – (7.7) may lead to a process bandwidth being greater than the link bandwidth $\beta_i(t) > C$, which could not be supported by the hardware. It is then possible to truncate the bandwidth to C and redistribute the remaining bandwidth among the remaining ready *mapper* processes. This is more precisely describe in Algorithm 3.

In this algorithm, E, E', τ, and γ_i are only used as temporary storage. $ready(t)$ and σ are the input, and $\beta_i(t)$ is the output. As previously said, every iteration of this algorithm runs in time $\mathcal{O}(m)$. This can be seen in Algorithm 3 on Lines 4 to 7. And since the set E holds a maximum of m elements in the beginning and is reduced by at least one element on every iteration, that means that a maximum of m iterations may be needed. Thus this algorithm have an overall worst-case complexity of $\mathcal{O}(m^2)$. It should be noted that when the bandwidth of a *mapper* process i is reduced to C, it means that this process will not be able to complete its transfers at the same time of the others. In this case, this algorithm may not be optimal. A sufficient condition for this algorithm to be optimal can thus be determined.

Algorithm 3 Per process bandwidth distribution.

$E \leftarrow ready(t)$
$\tau \leftarrow \sigma$
repeat
  $\gamma_i \leftarrow \tau \frac{\zeta_i(t)}{V(t)} \; \forall i \in E$
  $E' \leftarrow \{i | i \in E \wedge \gamma_i > C\}$
  $\beta_i(t) \leftarrow C \; \forall i \in E'$
  $E \leftarrow E \setminus E'$
  $\tau \leftarrow \tau - C \, |E'|$
until $E' = \emptyset$
$\beta_i(t) \leftarrow \gamma_i \; \forall i \in E$

Sufficient condition of optimality. From the sufficient condition of optimality presented in Section 7.4.1.1, this algorithm is optimal as long as it can make all the transfers end at the same time. It will make the transfers end at the same time from the point there is enough contention for the bandwidth of all the *mapper* process to be less than C. A necessary condition for this to happen is that every transfer is long enough to not terminate before the last transfer starts, which means $\frac{\zeta_i}{C} > S_m - S_i$. And that at the date $t = S_m$ the remaining intermediate data must be distributed so that no process bandwidth is greater than C.

$$\sigma \frac{\zeta_i(S_m)}{V(S_m)} < C \; \forall i \in [1..m]$$

Limitations As this algorithm assume that a given *mapper* process has the same amount of intermediate data to transfer to every *reducer* (7.3), the bandwidth computed for every process is divided evenly among the transfers $\beta_{i,j} = \beta_i / r$. However, in reality, assertion (7.3) will rarely be met, and despite the bandwidth control, the transfers from the same *mapper* may not all progress at the same speed, thus introducing more imbalance and degrading the performance.

7.4.2.5 Per-Transfer Bandwidth Regulation

The second algorithm is very similar to the previous one. The main difference is that it tries to address the limitation of setting the same bandwidth to every transfer from a given process by computing a bandwidth for every transfer, thus removing Assertion (7.3).

Model addition The model and notation here are mostly the same as the previous algorithm. $ready(t)$ is the set of couple (i, j), i being a *mapper* which finished its computation, and j being a *reducer*. $\zeta_{i,j}(t)$ is the amount of intermediate data the ready *mapper* i has to transfer to the *reducer* j at a date t. And $\beta_{i,j}(t)$ the bandwidth allocated to the transfer from the ready *mapper* i to the *reducer* j at a date t.

Algorithm The bandwidth of transfers $\beta_{i,j}(t)$ can be computed by solving the following system.

$$\frac{\zeta_{i,j}(t)}{\beta_{i,j}(t)} = T \qquad \forall (i,j) \in ready(t) \tag{7.8}$$

$$\sum_{(i,j) \in ready(t)} \beta_{i,j}(t) = \sigma \tag{7.9}$$

The system (7.8) – (7.9) can be reformulated as follows.

$$\zeta_{i,j}(t)\beta_{k,l}(t) = \zeta_{k,l}(t)\beta_{i,j}(t) \quad \forall (i,j) \in ready(t) \wedge \text{ any } (k,l) \in ready(t) \wedge (i,j) \neq (k,l)$$
$$\sum_{(i,j)\in ready(t)} \beta_{i,j}(t) = \sigma$$

This linear system can be solved in time $\mathcal{O}(m \times r)$. The trivial solution to the linear system of the previous algorithm can be adapted to this one. Let's redefine a shorthand $V(t) = \sum_{(i,j)\in ready(t)} \zeta_{i,j}(t)$. Then a solution is:

$$\beta_{i,j}(t) = \sigma \frac{\zeta_{i,j}(t)}{V(t)} \; \forall (i,j) \in ready(t)$$

This direct solution may, again lead to some bandwidth being greater than the link bandwidth. It can be truncated in a similar way as previously, as shown is Algorithm 4.

Algorithm 4 Bandwidth calculation of the per-transfer bandwidth regulation algorithm.

```
1: E ← ready(t)
2: τ ← σ
3: repeat
4:     γ_{i,j} ← τ ζ_{i,j}(t)/V(t) ∀(i,j) ∈ E
5:     E' ← {(i,j) | (i,j) ∈ E ∧ γ_{i,j} > C}
6:     β_{i,j}(t) ← C ∀(i,j) ∈ E'
7:     E ← E \ E'
8:     τ ← τ − C |E'|
9: until E' = ∅
10: β_{i,j}(t) ← γ_{i,j} ∀i ∈ E
```

Since E may contain every couple of *mapper* and *reducer*, its maximal size is $m \times r$. Thus, every iteration runs in time $\mathcal{O}(m \times r)$ because of Lines 4 to 7. And because at least one couple is removed from E at every iteration, this whole algorithm has a time complexity $\mathcal{O}(m^2r^2)$ in the worst case.

Limitations This algorithm has two main limitations. The first one is that, although it takes into account the capacity of the network links, it does not take into account that several transfers occur at the same time from a given process or toward a given process. Thus, even if every transfer bandwidth $\beta_{i,j}(t)$ is less than the capacity of the link C, the sum of the bandwidth of all the transfers of one process may be greater than C. A consequence of this, is that the bandwidth of the switch σ may not be reached because the bandwidth throttled by the network interface cannot be allocated to the other transfers. The second limitation is the scalability of this algorithm. If $m = r$, the complexity of this algorithm is $\mathcal{O}(m^4)$ which is not acceptable.

7.4.2.6 Two Phases Per-Transfer Regulation

A mix of both above regulation-based algorithms should be able to overcome most of the limitations. The idea of this third algorithm is to compute the bandwidth with the per-process bandwidth regulation algorithm, and then distribute it according to the amount of intermediate data of every transfer, instead of allocating it evenly. This algorithm is expected to never allocate too much bandwidth to a given *mapper* process and make all the transfers of all the *mappers* finish at the same time.

Model addition Similarly to the previous algorithms $ready(t)$ is the set of *mapper* processes that have finished their computation but not the transfer of their intermediate data. $\beta_{i,j}(t)$ is the bandwidth allocated to the transfer from *mapper* i to *reducer* j at a date t, and $\beta_i(t) = \sum \beta_{i,j}(t)$ the bandwidth allocated to a given *mapper* process i. $\zeta_i(t)$ is also the amount of intermediate data a ready mapper i still have to transfer to the reducers at date t. And $\zeta_{i,j}(t)$ the amount of data still to be transfered from *mapper* i to *reducer* j.

Algorithm The first phase of this algorithm is exactly Algorithm 3 that computes values for $\beta_i(t)$. The second phase applies a very similar algorithm for every process in order to distribute the bandwidth among the transfers. Algorithm 5 shows whole algorithm. Lines 1 to 10 come from the per-process bandwidth regulation algorithm, it computes $\beta_i(t)$ from $\zeta_i(t)$, $V(t)$, σ and $ready(t)$. The second phase spans across Lines 11 to 13. It takes values for $\beta_i(t)$ as input and produce values for $\beta_{i,j}(t)$ as output.

Algorithm 5 Bandwidth calculation of the two-phases bandwidth regulation algorithm.

1: $E \leftarrow ready(t)$
2: $\tau \leftarrow \sigma$
3: **repeat**
4: $\quad \gamma_i \leftarrow \tau \frac{\zeta_i(t)}{V(t)} \; \forall i \in E$
5: $\quad E' \leftarrow \{i | i \in E \wedge \gamma_i > C\}$
6: $\quad \beta_i(t) \leftarrow C \; \forall i \in E'$
7: $\quad E \leftarrow E \setminus E'$
8: $\quad \tau \leftarrow \tau - C \, |E'|$
9: **until** $E' = \emptyset$
10: $\beta_i(t) \leftarrow \gamma_i \; \forall i \in E$
11: **for** $i \in ready(t)$ **do**
12: $\quad \beta_{i,j}(t) \leftarrow \beta_i(t) \frac{\zeta_{i,j}(t)}{\zeta_i(t)} \; \forall j \in E$
13: **end for**

The worst-case complexity of the first phase is $\mathcal{O}(m^2)$ just like the per-process regulation algorithm. The complexity of the second phase is $\mathcal{O}(m \times r)$ because the loop iterates m times, and every iteration has $\mathcal{O}(r)$ operations to perform. The complexity of the full two-phases algorithm is then $\mathcal{O}(m^2 + m \times r)$. The second phase can be distributed and every *mapper* can distribute its allocated bandwidth $\beta_i(t)$ on its own. Thus reducing the worst-case complexity of the whole algorithm to $\mathcal{O}(m^2 + r)$.

Limitations Although this algorithm prevents any contention on the switch or on the private links of the *mappers*, contention may happen on the *reducers* side. Indeed, nothing prevents several *mappers* from sending some data to the same *reducer* with a bit rate sum greater than the bandwidth of its private link.

7.4.3 Point-to-Point Bandwidth Regulation

The regulation-based algorithms assume that it is possible to regulate the bandwidth of every transfer and to change the target bandwidth at any moment. However, this is not an immediate task since, in the end, only packets can be sent over the network. Our regulation algorithms raise two main challenges regarding this issue. The first one is about actually limiting the average bandwidth to a given value. The second challenge is about dynamically modifying the bandwidth at any time and any given number of times while still guarantying that the

average bandwidth is the one requested. Some work exists on the topic [91, 92, 93], however, most of them focus mainly on only maintaining an average bandwidth. Moreover, it was pretty straightforward to implement in our own framework.

7.4.3.1 Limiting the Average Bandwidth

The first step is to limit the average bandwidth. The algorithm we used for this is quite simple. The amount of data that has been sent is accumulated, and after a chunk of data have been sent, the execution is suspended for a certain duration.

The sleep duration is computed so that the average bandwidth from the beginning of the transfer to that date is exactly the average bandwidth wanted. So if t_0 is the date the whole transfer started, d is the total amount of data sent, and β is the wanted bandwidth, then, the execution will be suspended until the date $t_1 = t_0 + \frac{d}{\beta}$. Algorithm 6 shows a pseudo code for this algorithm. This algorithm is independent from the size of the chunk of data. However, as it is shown in Section 7.6.2.2 on a real computer, the size of the chunk may have an impact on the performance.

Algorithm 6 Simple regulation of a point to point bandwidth.

```
1: t_0 ← now()
2: d ← 0
3: for chunk ∈ data do
4:     send(chunk)
5:     d ← d + size(chunk)
6:     sleep_until(t_0 + d/β)
7: end for
```

As long as every *send* happens at a bandwidth greater than β, it is a direct consequence that this algorithm will guaranty the average bandwidth (from t_0 to the end) to be β. However, it can also proved that between two iterations, the average bandwidth is also β. Indeed, let's name t_i the date at the end of the i-th iteration and d_i the value of d at the date t_i. Then an expression for the date t_i can be written and the time taken by one iteration can be computed.

$$
\begin{aligned}
t_i &= t_0 + \frac{d_i}{\beta} \\
t_{i+1} - t_i &= \left(t_0 + \frac{d_{i+1}}{\beta}\right) - \left(t_0 + \frac{d_i}{\beta}\right) \\
t_{i+1} - t_i &= \frac{d_{i+1}}{\beta} - \frac{d_i}{\beta} \\
t_{i+1} - t_i &= \frac{d_{i+1} - d_i}{\beta}
\end{aligned}
$$

Thus, the time taken by the $i+1$-th iteration is exactly the time that would be needed to send the $i+1$-th chunk of data at a bandwidth of β octet per second.

7.4.3.2 Run Time Setting of the Bandwidth

The second step is to make it work with β being variable through the time, making it a function of the time $\beta(t)$. The easiest solution of atomically resetting t_0 to $now()$ and d to 0 could produce undesirable effects. If it is chosen to not interrupt the *sleep_unitl*, then the average bandwidth would not be equal the average of $\beta(t)$ because Algorithm 6 may sleep too long if the target bandwidth has been suddenly raised. Conversely, interrupting the *sleep_until* call

and continuing the execution would always produce another send to happen. Which can may compromise the regulation if the target bandwidth is changed often.

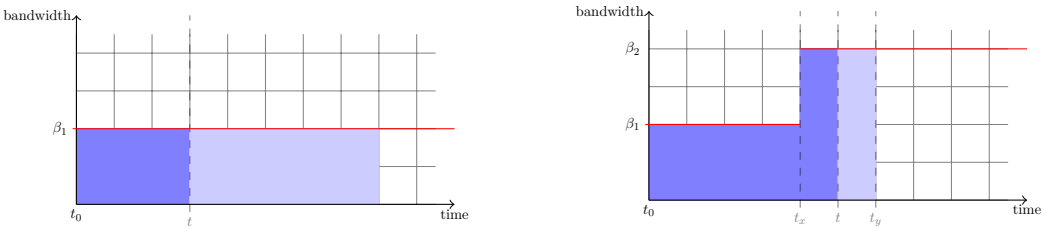
Another approach is to lie to Algorithm 6 by setting a fake value for t_0 named t'_0. A value that would lead it to make the average real bandwidth to be equal to the average value of $\beta(t)$ while still guarantying that when β is not modified, the target bandwidth is maintained.

Let's consider a scenario where from t_0 to t_x the target bandwidth is β_1 and from t_x to t_y the target bandwidth is β_2. d_x bytes are transfered between t_0 and t_x. And d_y bytes are transfered between t_0 and t_y. At t_x the value of t_0 change to be t'_0, hence $t_y = t'_0 + \frac{d_y}{\beta_2}$. Then, solve for t'_0 the equation between the average real bandwidth and the average of the wanted bandwidth.

$$
\begin{aligned}
\frac{d_y}{t_y - t_0} &= \frac{\beta_1 (t_x - t_0) + \beta_2 (t_y - t_x)}{t_y - t_0} \\
d_y &= \beta_1 (t_x - t_0) + \beta_2 (t_y - t_x) \\
d_y &= \beta_1 (t_x - t_0) + \beta_2 \left(t'_0 + \frac{d_y}{\beta_2} - t_x \right) \\
d_y &= \beta_1 (t_x - t_0) + \beta_2 t'_0 + d_y - \beta_2 t_x \\
t'_0 &= \frac{\beta_1 (t_x - t_0) - \beta_2 t_x}{-\beta_2} \\
t'_0 &= t_x - (t_x - t_0) \frac{\beta_1}{\beta_2}
\end{aligned}
$$

It is really interesting to remark that this value of t'_0 does not depend on any d_i nor on t_y. This means that this new value for t_0 can be computed at the date t_x when the target bandwidth is actually changed. Then, the *sleep* of Algorithm 6 can be interrupted and restarted with the new values for β and t_0.

This result can actually be understood intuitively by picturing a graph of the wanted bandwidth through time. Figure 7.3 shows how evolves during the time t, the past and foreseen bandwidth usage. In dark blue is the past bandwidth usage as pictured by the model, and in light blue is how the bandwidth should be used if nothing changes in the future. In Figure 7.3(a) the bandwidth limit is set at β_1, while, in Figure 7.3(b) the bandwidth limit is set to β_2 at the date t_x. The total blue area (proportional to the amount of data) is the same in both cases.



(a) Before target bandwidth change.

(b) After target bandwidth change.

Figure 7.3: Schema of bandwidth usage past and foreseen.

When computing a fake value for t_0 what really happens is that a *fake shape* is with the same surface as before but, with a height (a bandwidth) equal to the new bandwidth. Figure 7.4 shows an example of what could happen. This allows to remove any reference to the previous bandwidth as well. Given this, it can easily be seen that this technique will work for more than one bandwidth update.

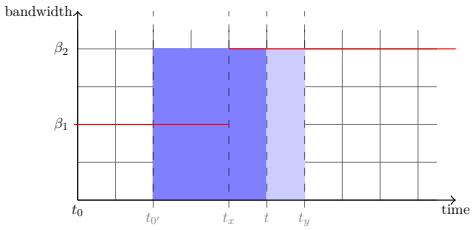


Figure 7.4: Schema of the calculated $t_{0'}$.

7.4.3.3 Precision Improvement

As the three regulation-based algorithms rely on a continuous data transfer while the data are actually sent by chucks. Providing them with the amount of data really transfered may lead to wrong bandwidth calculation. Indeed, reporting that a chunk of data has been fully transfered while a *sleep* is currently smoothing the consumed bandwidth, break the assumption that the transfer is continuous. This can lead the regulation-based algorithm to compute a wrong bandwidth.

That is why the remaining data size reported is interpolated as follow. With t_x being the date when the remaining size is asked, and r the amount of data actually reported.

$$r = \zeta_{i,j} - \beta_{i,j}\,(t_x - t_0)$$

7.5 Implementation Details

All those algorithms are implemented in *HoMR*. *HoMR* is our HOme-made MapReduce. It is written in C++ under the scope of the ANR project MapReduce. It is built from software components, based on the L2C low-level component model. The low-level component assembly is generated from a high-level component model HLCM [19]. This allows to easily swap any component implementation with another variant. This is used to test several *shuffle* schedulers and to replace the word reader with a word generator.

The transfer scheduler components are implemented in an even-driver way. Every time a computation or a transfer finishes, a method of the transfer scheduler is called. In every such event the three regulation-based schedulers recompute the new bandwidth allocated to the processes or to the transfers. In a perfect world, there is no need to recompute the bandwidth allocation out of these events. However, it does not cost anything more than an $\mathcal{O}(m \times r)$ packets exchange to recompute the bandwidth allocation when no even have been received during a few seconds. In our regulation-based transfer schedulers, the bandwidth allocation is recomputed after 5 seconds of idle.

The bandwidth regulator presented in Section 7.4.3 uses 4 threads to send the data. This helps improve the maximal bandwidth that can be reached. This number of thread has been determined by running a few tests by hand, increasing the number of threads until it no longer improve the maximal bandwidth. The same behavior has been observed with *iperf* which show a maximal throughput for 4 client threads.

7.6 Experiments

In order to test these algorithms, a few experiments have been performed on the Grid'5000 experimental testbed, first to ensure the environment behave as expected, and second to compare the 6 algorithms presented in this document.

7.6.1 Platform Setup

The model assumes that the platform is a switched star-shaped network with a limited bandwidth on the switch. And the whole point of the algorithms is to control the bandwidth used during the *shuffle* phase. Thus, in order to evaluate these algorithms in the case of several switch bandwidth limit configurations, a switch is simulated by the mean of a node dedicated to routing packets and all other nodes configured to route packets through this node. However, as all the nodes are physically connected to a real switch, the *ICMP redirect* mechanism had to be disabled to ensure that every packet sent over the network really go through the node designated as the router.

This may not be an optimal simulation of a switched network since the routing mechanism implies a *store-and-forward* method of forwarding the packets, instead of a *cut-through* as most switches do. However, we believe that this does not have a big effect on the measured throughput. This allow to easily control the overall bandwidth available on that routing node.

As all the packets have to go through the network interface of the router node twice, a fast network is needed in order to simulate a switch with a throughput greater than 1 Gbps. Thus, *InfiniBand 40G* interfaces are used with an *IP over InfiniBand* driver for the ease of use. The bandwidth of the router is controlled with the Linux `tc` tool. The performance behavior and limit of this setup has been tested and the results are shown in Section 7.6.2.1. As the network is based on fast network interface controllers (NIC), the bandwidth of the private links is also limited to 1 Gbps with `tc`. The `tc` rules used to limit the bandwidth are the same on the router and on the compute nodes. They are based on the Hierarchical Token Bucket (HTB) method to limit the outgoing bandwidth and on the default algorithm to limit the incoming bandwidth. The operating system of the nodes is Debian wheezy with Linux 2.6.32 as a kernel.

The hardware used is the Edel cluster on Grid'5000. Every node on this cluster has two quad-core CPUs Intel Xeon E5520 @2.27 GHz. Every node is equipped with 24 GB of memory, 1 Gigabit Ethernet and 1 InfiniBand 40G cards. On this hardware, a latency of 0.170 ms has been measured on average on a direct point to point ping using the Gigabit Ethernet NIC. A latency if 0.315 ms is also measured using the above-mentioned routed network setting on the InfiniBand NIC.

7.6.2 Preliminary Tests

7.6.2.1 `tc` Regulation on Router

In order to test whether the bandwidth can be limited correctly on the router, a setup with 2 nodes plus a router node is used. Only the router node has a limited bandwidth, and `iperf` is run on the two other nodes to measure the actual bandwidth.

The bandwidth limit is set with `tc` from 100 Mbps up to 12 Gbps by steps of 100 Mbps. And the actual bandwidth is measured with `iperf` with 4 parallel clients threads on the client side. Every measure is run 5 times to estimate the variability.

Figure 7.5 shows the results of this experiment. Globally, it can be seen that the measured bandwidth follows an almost linear trend which corresponds to roughly 90% to 95% of the target bandwidth. This trend continues until the maximal bandwidth the system can support is reached. Also, the measures are quite stable as the difference between the maximal and minimal measured bandwidth never exceeds 0.28 Gbps or 8% of the average bandwidth.

However, some steps are clearly distinguishable around 5 Gbps and from 6.5 to 8 Gbps. During these steps, increasing the bandwidth limit with `tc` does not increase the actual bandwidth. As the result is surprising, it has been re-executed on another cluster with *InfiniBand 20G* network adapters, and we see that the same result happen. We have no real explanation for that. The experiment has also been tried with a Linux 3.2.0 with a Debian Wheezy, the

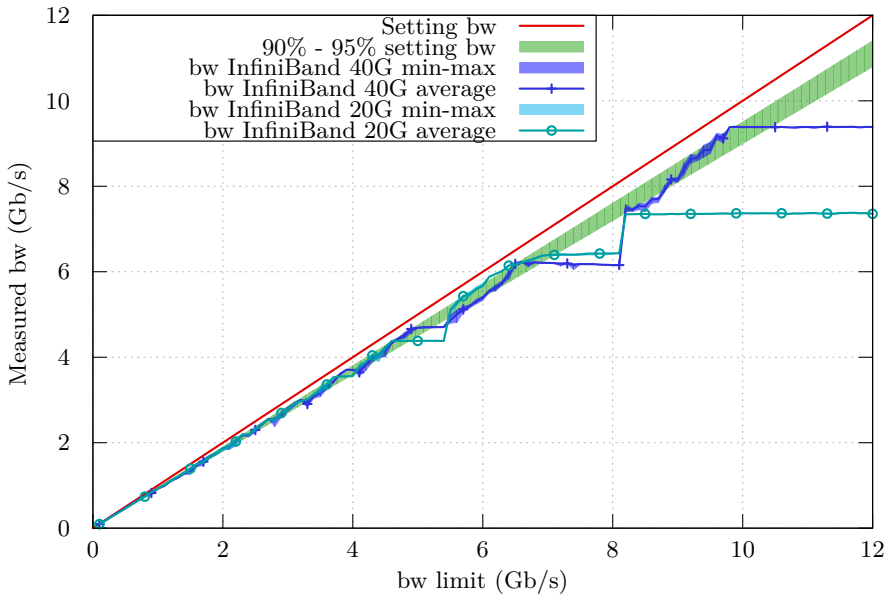


Figure 7.5: Bandwidth limitation on Linux + `tc` on InfiniBand.

results (not shown here) are completely different and show a greater variability. Thus, we think that this is a performance bug in Linux.

In order to still get the wanted bandwidth, the data from Figure 7.5 are used to find a setting bandwidth that would result in an actual bandwidth being close to the target one. For instance, to get an actual bandwidth of 5 Gbps, the bandwidth limit set with `tc` is 5.994 Gbps. Since for the real experiments the bandwidth limit will vary by 1 Gbps steps, the value for only 10 target bandwidth have to be found. Figure 7.6 shows the bandwidth measured with `iperf` when using to the corrected setting bandwidth. It can be noted that the average actual bandwidth is remarkably accurate with respect to the setting bandwidth. The only deviations that can be noted happen for a target bandwidth of 7, 8 and 10 Gbps. Those points corresponds to the biggest steps in Figure 7.5 and to the maximal reachable bandwidth. For those points, the drift could not be completely compensated. Those outliers show a bandwidth still greater than 90% of the target bandwidth. Except from those 3 points, all others show an actual bandwidth between 96% and 103% of the bandwidth wanted.

7.6.2.2 Bandwidth Regulation

The second base block on which these algorithms rely on is the ability to regulate the bandwidth at the application level. The method used for this is described in Section 7.4.3.

To check whether this performs correctly, an experiment is set up with only two nodes interconnected by an *InfiniBand 40G* network. Then the size of the messages is varied from 4 bytes to 64 MB and the target bandwidth from 1 KB/s to 1 GB/s and the overall average bandwidth is measured. Each measure is repeated 10 times.

Figure 7.7 shows a 3D plot of the results of this experiment. It shows the actual bandwidth with respect to the message size and to the desired bandwidth. Some points are missing in the result because sending a large amount of data at a very low bandwidth would require too much time. The colors represent the percentage of variability. Figure 7.8 represents the same information under the form of a contour plot.

The black plan on Figure 7.7 shows that when the required bandwidth is small enough and

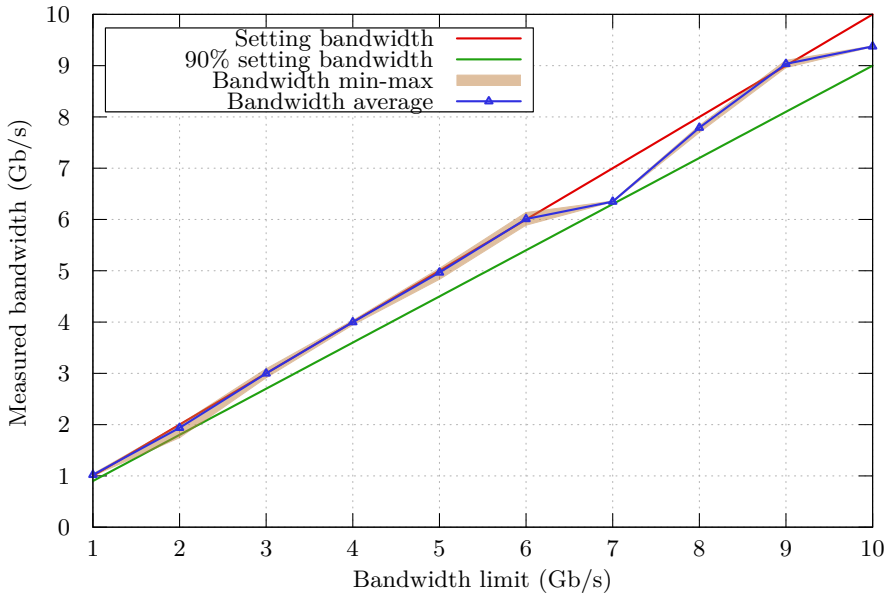


Figure 7.6: Bandwidth test of Linux + `tc` on InfiniBand with correction.

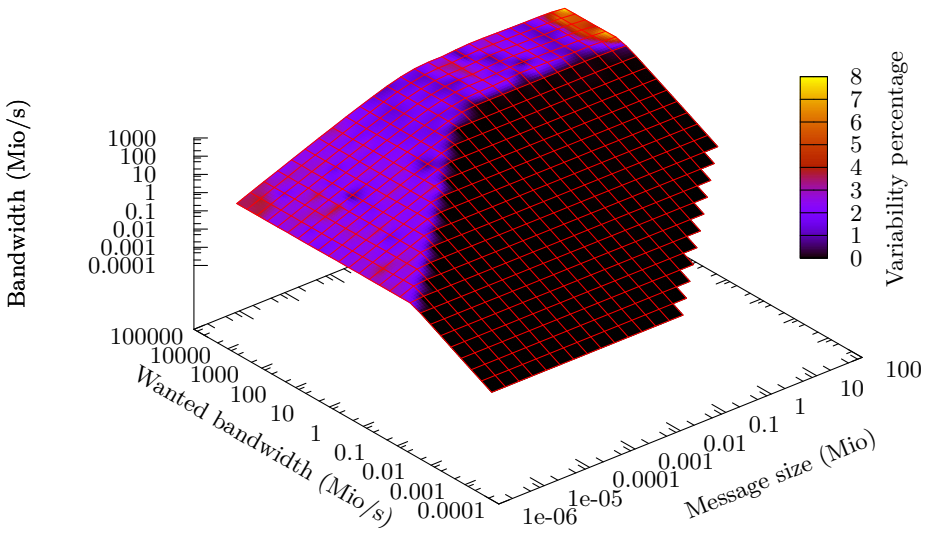


Figure 7.7: Bandwidth regulation test, 3D plot.

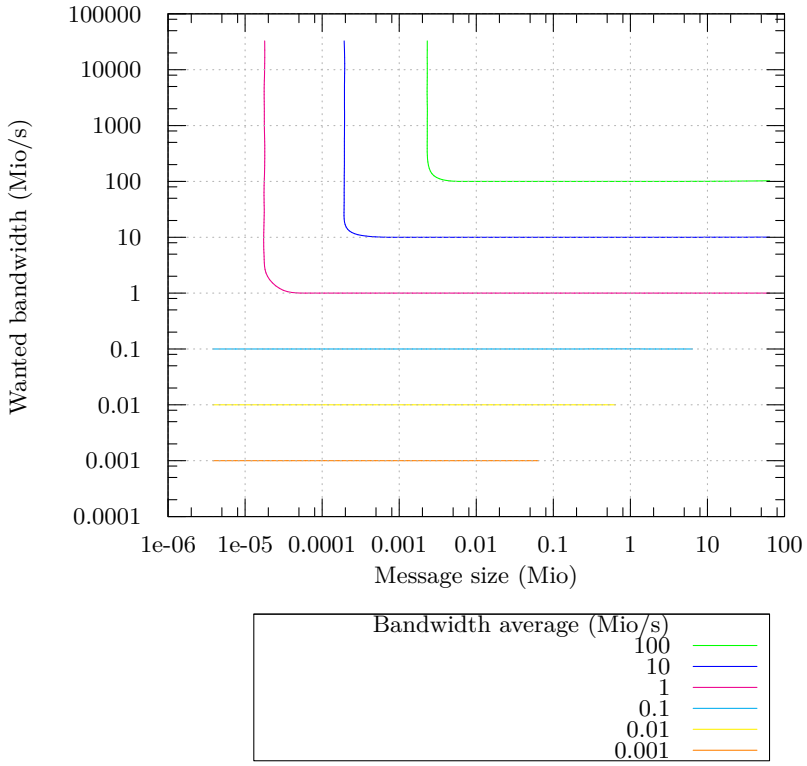


Figure 7.8: Bandwidth regulation test, contour plot.

the block size is big enough, the bandwidth regulation system is able to maintain the desired bandwidth with a pretty good accuracy. However, when the message size is too small, the system becomes CPU bound and cannot send enough data to reach the desired bandwidth. And because this case is CPU bound, a variability of 0.5% to 1% can be seen. Finally, when the messages are large enough and the required bandwidth is high enough, the system reaches the limit of the network interface and become IO bound. Thus our bandwidth regulation component works as intended.

7.6.3 Synchronous Transfer Start

The first experiment with our algorithms is simple and all other experiments are only variations of this one. The job that is run is a word count. However, for the sake of simplicity and control, the data are not read from a file, they are generated by a component *WordGenerator*. This allows to control the amount of intermediate data produced. In order to control the time at which the *map* computations end, an artificial synchronization barrier is added. This allows for an evaluation of the behavior of the *shuffle* phase.

For this first experiment all the *map* computations finish at the exact same time and every *mapper* have the same amount of intermediate data. Every *mapper* process generates 2.56 GB of intermediate data. The same amount of data has to be sent to every *reducer*. The router's bandwidth is then varied from 1 Gb/s to 10 Gb/s and the time taken from the start of the first transfer to the end of the last transfer is measured. This duration is then compared to the lower

bound. This experiment is run with 10 *mappers* and 10 *reducers*. Every configuration is run 5 times.

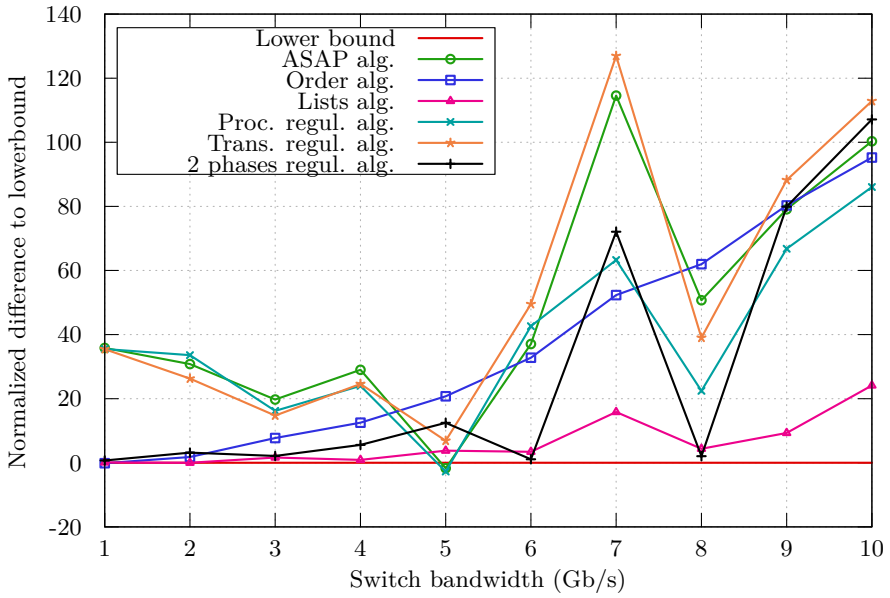


Figure 7.9: Median time taken by all the 6 algorithms under various bandwidth restrictions with same amount of data and synchronous start of transfers.

Figure 7.9 shows the result of this experiment in terms of percentage to the lower bound. As every measure has been made 5 times, the median time is represented on this figure. Figure 7.10 shows the variability of the measures in candle sticks.

The result for the discrete algorithms (*Order alg.* and *Lists alg.* on the figure) is close to what was expected. The result for the order-based algorithm increases on Figure 7.9, not because it takes more time as the bandwidth of the switch increase, but because it does not decrease as fast as the lower bound. The time needed before the maximal number of concurrent transfers can run simultaneously become less and less negligible as the overall allowed number of concurrent transfers increase. The lists-based algorithm show a behavior close to the optimal. The only performance degradation occurs for a switch bandwidth of 7 Gb/s and 10 Gb/s. Those configuration, as of Figure 7.6 are known not to offer the actual bandwidth wanted.

The per-process regulation algorithm, per-transfer regulation algorithm, and the reference algorithm behave similarly for the same reasons. They create contention at some point, thus losing some packets creating a latency. The per-process regulation algorithm is not supposed to generate any contention for this experiment. The cause of this is unknown. However, the instantaneous egress bandwidth of the switch (not shown here) shows that those 3 algorithms can reach the bandwidth actually set with `tc` by running some transfers from several nodes at the same time. While the `tc` skew correction has been tested only for several transfers from the same node with `iperf`.

The 2 phases regulation algorithm shows an good behavior for a switch bandwidth less or equal to 8 Gb/s. Above that limit it creates contention and exhibits a behavior as bad as the reference algorithm. Also, for 7 Gb/s, this algorithm produces a peak of bad performance. This can be interpreted as a high sensitivity to the setting of the switch bandwidth. If the switch bandwidth is overestimated, the 2 phases algorithm creates contention which leads to a very bad performance. While if it is underestimated, the switch would never be used at its maximal capacity.

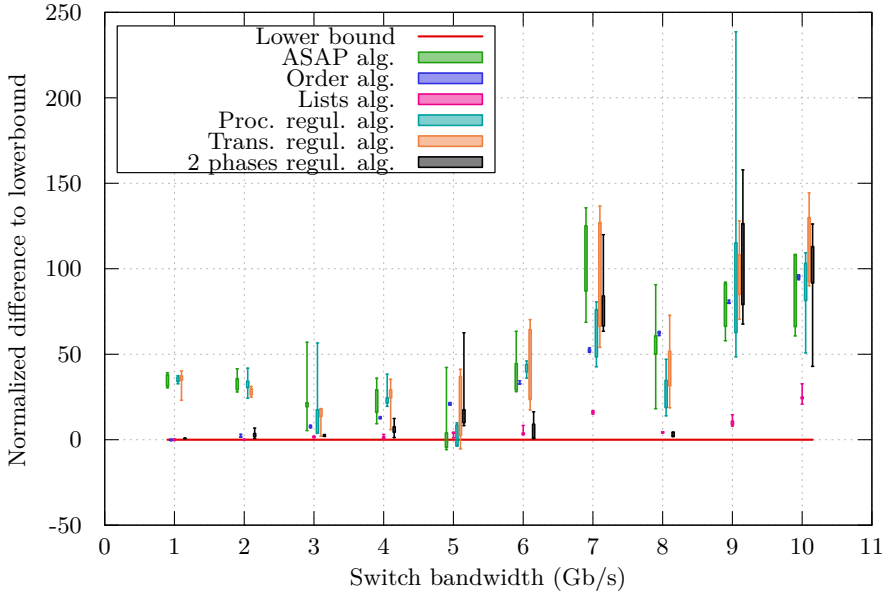


Figure 7.10: Variability of the time taken by all the 6 algorithms under various bandwidth restrictions with same amount of data and synchronous start of transfers.

7.6.4 1 Second Steps Between Computation End

The second experiment is very similar to the previous one. Only a one-second delay between the end of every *map* computation has been added, thus creating a slight imbalance among the *mapper* process.

Figure 7.11 shows the result of this experiments in terms of percentage to the lower bound.

The discrete algorithms show a similar behavior as the previous experiment. However, the order-based algorithm shows a slightly better behavior: its distance to the lower bound increases slower. This is due to the time taken to gain parallelism among the transfers that is partially compensated by the 1 second steps between the computation end. The reference algorithm also shows a better performance. This is due to the fact that in the beginning and in the end, not all the *mappers* are transferring data, thus there is less contention and less performance degradation. The global behavior of the list-based and two-phases algorithms remain the same. However the two-phases algorithm appear to be super-optimal by up to 5% for some configurations. The cause is not very clear. It is supposed to be caused by `tc` that is not very accurate to limit the bandwidth from several sources.

7.6.5 Synchronous Transfer Start with Heterogeneous Amount of Data

For the third experiment, all the *mappers* finish their computation at the same time, but all the *mapper* do not have the same amount of intermediate data. The amount of intermediate data for each *mapper* is 2.56 GiB plus $(i-1) \times 64$ MiB, i being the id of the *mapper* process, ranging from 1 to m. As previously, we expect the smarter algorithms to perform better.

The results of this experiments presented in Figure 7.13 show similar results as the previous experiment. Variability is shown in Figure 7.14.

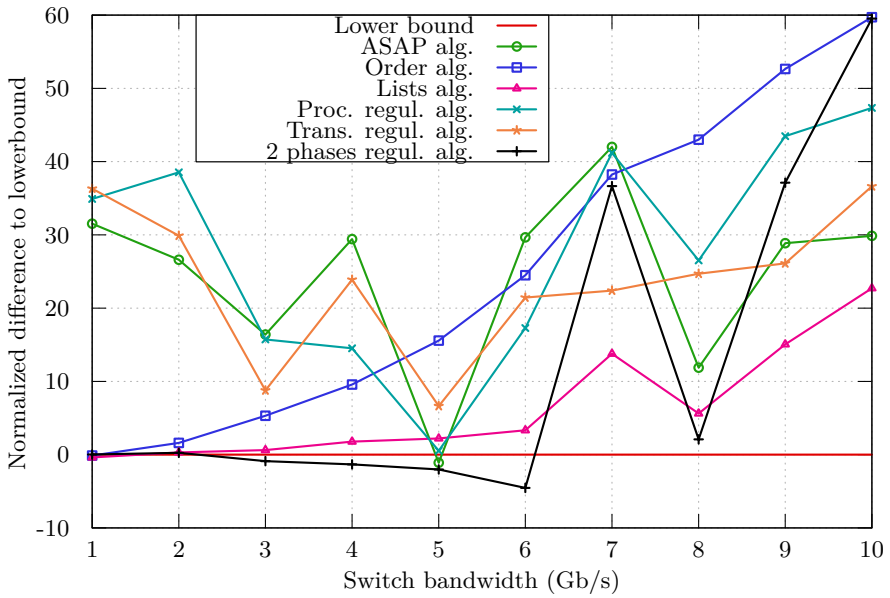


Figure 7.11: Median time taken by all the 6 algorithms under various bandwidth restrictions with same amount of data and 1 second step between transfer start.

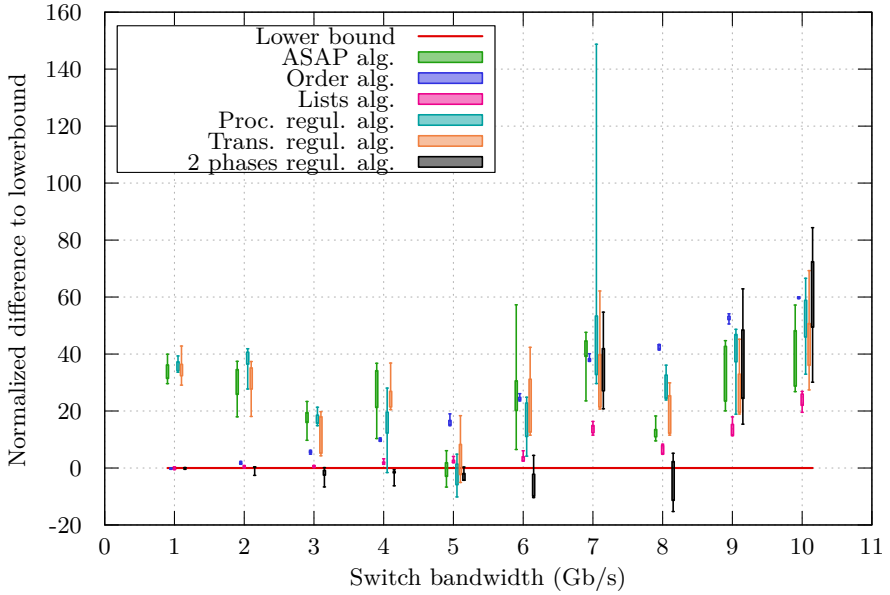


Figure 7.12: Variability of the time taken by all the 6 algorithms under various bandwidth restrictions with same amount of data and 1 second step between transfer start.

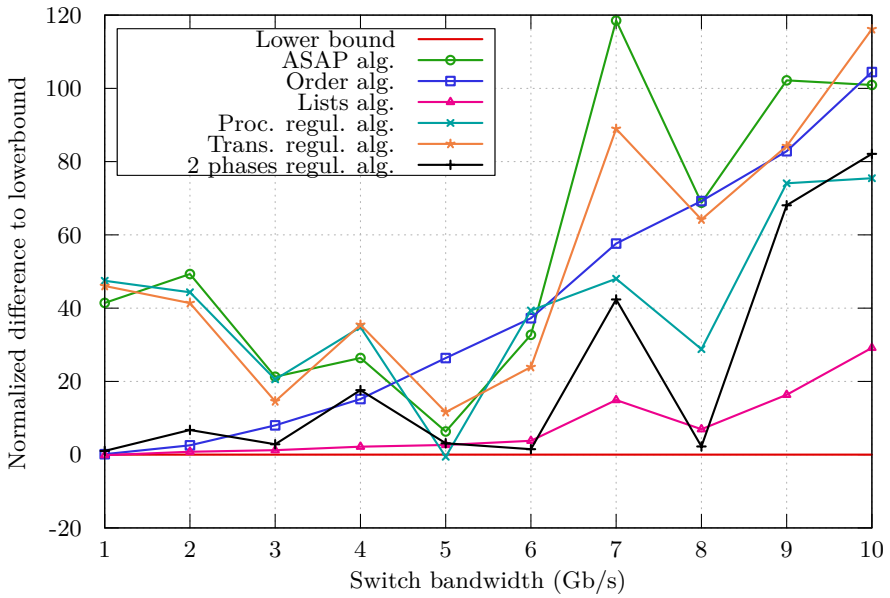


Figure 7.13: Median time taken by all the 6 algorithms under various bandwidth restrictions with various amount of data and synchronous start of transfers.

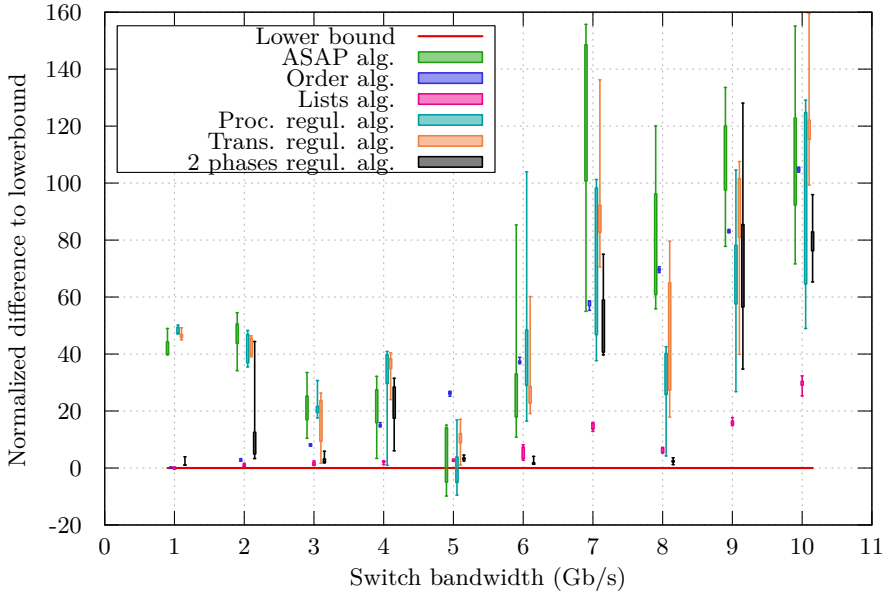


Figure 7.14: Variability of the time taken by all the 6 algorithms under various bandwidth restrictions with various amount of data and synchronous start of transfers.

7.6.6 1 Second Step Between Transfer Start with Heterogeneous Amount of Data

And finally, the fourth experiment combines the delay before starting the transfers and the imbalance of the amount of intermediate data.

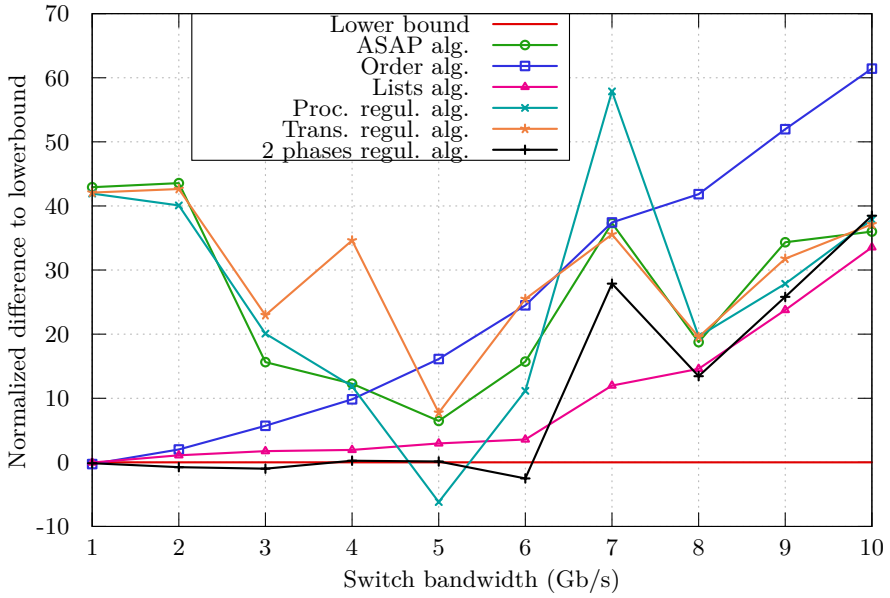


Figure 7.15: Median time taken by the 6 algorithms under various bandwidth restrictions, with an imbalanced amount of intermediate data and non-synchronous transfer start.

The results of this experiments presented in Figure 7.15 in terms of difference to lower bound normalized to the lower bound itself. The variability of the measures is presented in Figure 7.16 in the form of whiskers boxes.

On the results it can be seen that the discrete algorithms show a quite smooth behavior, the list-based algorithm being always better than the order-based algorithm. Both show a quite low variability, except one measure for the order-based algorithm that show an outlier for a router bandwidth of 5 Gb/s. During this measure, the bandwidth on the router node has dropped to 0 unexpectedly during 10 seconds. The order-based algorithm show a performance behavior that gets further and further from the lower bound, as we would expect because of slow start up and slow *stop down* as explained in Section 7.4.2.2. The same global behavior is observed for the lists-based algorithm, this time it is the heuristic for sorting the *mappers* by priority that cannot catch up the imbalanced transfer progression brought by the 1 second delay of the transfer start and the 64 MiB steps of the amount of intermediate data.

The three algorithms ASAP, per-process regulation, and per-transfer regulation show a poor performance for a small router bandwidth. Indeed, those algorithms generates some contention either on the router or on the private links. This contention leads to packet loss and retransmission after a timeout. These algorithms also show a performance that is equivalent to that of the 2 phases regulation and lists-based, for a router bandwidth large enough. The per-process regulation algorithm show a super-optimal performance for a router bandwidth of 5 Gb/s. This is actually due to an overshoot of the bandwidth set with `tc`. Indeed, while the transfers from a single node to another does not exceed the wanted bandwidth on the router as seen on Figure 7.6, it appears that when several nodes transfer some data at the same time, the total bandwidth may exceed a bit the target bandwidth. Globally, the 3 algorithms ASAP, per-

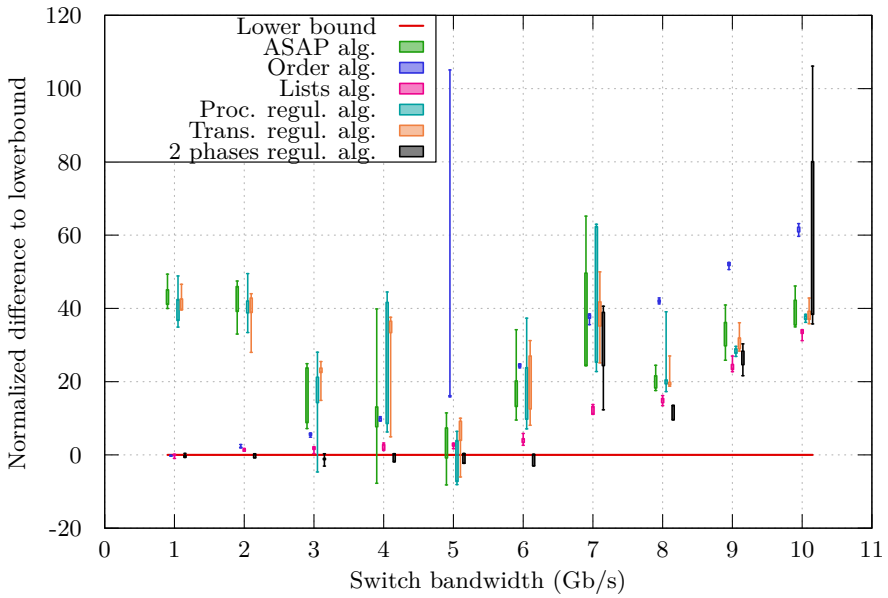


Figure 7.16: Variability of the time taken by the 6 algorithms under various bandwidth restrictions, with an imbalanced amount of intermediate data and non-synchronous transfer start.

process regulation, and per-transfer regulation show a bowl-shaped performance curve centered around 5 Gb/s. The left part seems to be due to the decrease of the contention ratio, leading to an increase of the performance. And the right part seems to be due to the imbalance among the amount of intermediate data to transfer that makes the increase of the bandwidth of the router have only a slight impact in the actual performance.

The 2 phases regulation algorithm exhibits an optimal behavior for a router bandwidth less or equal to 6 Gb/s. It may even be slightly super-optimal for the same reason exposed before. However, as this algorithm regulates the bandwidth so that it never exceeds the bandwidth of the switch (here simulated by a router) the overshoot can never be very large. As before, the 2 phases regulation algorithm also shows a peak at 7 Gb/s due to the fact that the bandwidth of the router could not be set to exactly 7 Gb/s. Thus some contention appears and degrades the performance as it happens to the ASAP algorithm for instance. This behavior can be interpreted as a high sensitivity to contention thus making the parameter σ of this algorithm (the switch bandwidth) critical for a good performance.

7.7 Conclusion and Future Works

The *shuffle* phase has been largely ignored by the academic work despite being a potentially important bottleneck. In this chapter we show here that although a no-op algorithm performs well under perfect balance and synchronous conditions without contention, smarter algorithms are proven to be more efficient in all other cases. Especially the list-based algorithm and the two phases algorithms. The second one may perform optimally in some cases, but is quite sensitive to the switch bandwidth parameter while the first one only needs to know how many concurrent transfers the switch can support. While those two algorithms periodically communicate with the centralized scheduler, the list-based algorithm induces an idle time between the transfers while the scheduler make a decision. This does not happen with the regulation algorithms since they just continue their transfer with the former bandwidth set while the scheduler computes the new one. The scalability of these algorithms still has to be tested.

However, we believe that the results could be better if the bandwidth of the router could be precisely limited with `tc`. Our attempt at mitigating the aberrations are a good start but it shows some limitation. A future direction could be to compare the behavior of Linux + `tc` with that of a real switch. This work assumed that a switched network with limited bandwidth could be simulated with a routed network. However, most current switches uses a *cut-through* method for forwarding the packets, while a routed network with a Linux system implies a *store-and-forward* method. It has not been proven that this difference does not have a sensible influence on the performance. With more time, we could make some experiments on a real switch. We could also try to map the *mappers* and *reducers* processes on the same nodes and check whether or not the same behavior is observed. Some parameters have to be known by the regulation algorithms such as the link bandwidth and the switch bandwidth. It would be a great improvement for the usability if those parameters could be determined automatically. Or even better, if they could be adjusted at run time if the bandwidth has to be shared with other applications. The platform model currently assumes that the network topology is switched star-shaped with an equivalent bandwidth on every private link. Extending both the models and algorithms to other network topologies and with less restriction is also interesting. Some of these algorithms could be extended to start the data transfer of the intermediate data before the computation is finished. Although extending the algorithms seems easy, their performance and the influence of the uncertainty about the data not produced yet bring new challenges.

Chapter 8

Conclusions and Perspectives

The advent of Big Data raises new challenges that need to be addressed in order to be able to continue to build knowledge out of the collected data. Moreover, with Big Data comes volume and velocity of the data production, which both make the usage of a distributed platform almost a requirement.

However, distributed platforms are not so easy to use by themselves and some software are needed to manage the complexity of the infrastructure as well as to ease the management of the data and the computations to perform on this platform. MapReduce can manage the computation and is actually widely even thought there is still room for performance improvement.

8.1 Conclusion

In this thesis several improvements for MapReduce have been proposed. The first one is a component-based MapReduce framework which makes use of the genericity of a high-level component model to parametrize the components and make it easily adaptable. The final design of HoMR is non-trivial because the borders of the process do not always match with the border of the components, but this is, in the end, more logical and easier to modify.

The second contribution of this work is the MapReduce models that have been developed. These models are linear which make them analytically tractable. This property has been proved useful and the models have been proved accurate enough to allow the scheduling algorithms to provide good results. This shows that a performance model for MapReduce need not be complicated to be accurate.

The third contribution is a partitioning and scheduling algorithm that tries to reach a global optimum. Despite there is no longer a guaranty of optimality, the combination of those algorithms have shown a gain up to 47% in schedule length, and a negligible time to compute when compared to a previous work on this topic.

And last but not least, this thesis proposed several algorithms to schedule the transfers of the *shuffle* phase of a MapReduce application. Among the five proposed algorithms, two show very good performance. The two-phase regulation algorithm shows under some condition a behavior equal to the lower bound, but is very sensitive to inaccurate estimations of the switch bandwidth and to contention on the reducers network links. The list-based scheduling algorithm is always a bit further from the lower bound than the two-phase algorithm, but is less sensitive to bad estimations of the available bandwidth. This shows that the network contention during the *shuffle* phase may hinder the performance while a careful scheduling of the transfers may show an improvement of up to 40%.

8.2 Perspective

Although the MapReduce models presented in Chapter 5 seem sensible, they are inspired by the existing models and lead to good results in practice, they have not been validated directly against a real life MapReduce job. This would increase the confidence we can have in these models and show the limits of their applicability.

Notwithstanding the successful performance optimization, there is still room for improvement. Contrary to what the results may suggest, there is no obvious reason for a trade-off to be mandatory between performance and reliability of the performance of the transfer scheduling algorithms. Since the peaks of bad performance are probably partially due to contention on the *reducer* side links, maybe the two-phase algorithm could be changed to work on the *reducer* side throughput instead of the *mapper* side throughput. This could help since there are less *reducers* than *mappers*, but contention on the *mapper* side could still occur. So another algorithm could be tried based on a linear program that would fix the actual transfer throughput to minimize the duration of the whole *shuffle* phase while guarantying that neither the switch bandwidth nor the links bandwidth would be exceeded.

Since the performance issues of the transfer schedulers are also due to an inaccurate estimation of the switch bandwidth, estimating it on the fly and dynamically would be of great help. This would also help on real platforms that may be shared with other applications and other users.

The network contention on the switch has been simulated with the tool `tc`. There is no guaranty that the behavior of Linux is similar to that of a real switch reaching its maximal throughput. Linux offers many options to control the behavior of the network stack, the values used for the experiments in this document may not be the best ones.

Most work of this thesis are specific to a star shaped network topology. This is a real fundamental limitation of the applicability to the real world of the work presented here. The platform model and performance model can be easily adapted to a hierarchical network topology. The transfer scheduling algorithms are not trivial to adapt, here are a few ideas. The list-based algorithm may be adapted to a tree-shaped network by counting the number of transfers using each switch-switch link and each switch as well. So that when a couple *mapper* – *reducer* is selected, the algorithm checks whether the network resources have enough room for this transfer to happen. If so, it decreases the resource counters and launches the transfers, if not, it selects the next *mapper* – *reducer* couple and try again. Maybe simpler, the two-phase algorithm could be applied once for the top-level switch viewing the second-level switch as nodes having some transfers to do that must cross this switch. Then, the same algorithm can be applied for every second-level switch, taking into account the bandwidth used by the transfers that must go up one level of the tree. Alternatively, the regulation approach could probably be solved with a linear program as previously.

The *shuffle* phase is not the only part of a MapReduce job that is largely forgotten by academic work. There is a common assumption that the data processed with MapReduce are already stored on the node that process them. However, this is not always true. Most cloud providers provide a storage service distinct from the computing nodes with a fairly high bandwidth between those two type of nodes. So the beginning of a MapReduce job consists in transfering the data from the storage system to the computing nodes. The work done about scheduling the *shuffle* data transfers could therefore be reused. This could also be used to determine an optimal number of nodes. Indeed, the more compute nodes there are, the less time is needed to scatter the data and to process the data during the *map* phase. But the more compute nodes there are, the more levels of the network tree are needed, and the longer is the *shuffle* phase. So there is probably an optimal setting between the extremes.

Bibliography

[1] Dean Jeffrey and Ghemawat Sanjay. "MapReduce: Simplified Data Processing on large Clusters". In: *In OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Vol. 51. USENIX Association, 2004.

[2] Doug Laney. "3D Data Management: Controlling Data Volume, Velocity and Variety". In: *META Group Research Note* 6 (2001).

[3] Julien Bigot. "Du support générique d'opérateurs de composition dans les modèles de composants logiciels, application au calcul scientifique". PhD thesis. INSA de Rennes, Dec. 2010.

[4] Joanna Berlinska and Maciej Drozdowski. "Scheduling Divisible MapReduce Computations". In: *Journal of Parallel and Distributed Computing* 71.3 (Mar. 2010), pp. 450–459. DOI: 10.1016/j.jpdc.2010.12.004.

[5] Gabriel Antoniu, Julien Bigot, Christophe Blanchet, Luc Bougé, François Briant, Franck Cappello, Alexandru Costan, Frédéric Desprez, Gilles Fedak, Sylvain Gault, Kate Keahey, Bogdan Nicolae, Christian Pérez, Anthony Simonet, Frédéric Suter, Bing Tang, and Raphael Terreux. "Scalable Data Management for Map-Reduce-based Data-Intensive Applications: a View for Cloud and Hybrid Infrastructures". In: *International Journal of Cloud Computing (IJCC)* 2.2 (2013), pp. 150–170.

[6] Gabriel Antoniu, Julien Bigot, Christophe Blanchet, Luc Bougé, François Briant, Franck Cappello, Alexandru Costan, Frédéric Desprez, Gilles Fedak, Sylvain Gault, Kate Keahey, Bogdan Nicolae, Christian Pérez, Anthony Simonet, Frédéric Suter, Bing Tang, and Raphael Terreux. "Towards Scalable Data Management for Map-Reduce-based Data-Intensive Applications on Cloud and Hybrid Infrastructures". In: *1st International IBM Cloud Academy Conference - ICA CON 2012*. Research Triangle Park, North Carolina, États-Unis, 2012. URL: http://hal.inria.fr/hal-00684866.

[7] Sylvain Gault and Christian Perez. "Dynamic Scheduling of MapReduce Shuffle under Bandwidth Constraints". In: *Europar 2014 Workshop Proceedings, BigDataCloud*. Porto, Portugal, 2014.

[8] Sylvain Gault. "Ordonnancement Dynamique des Transferts dans MapReduce sous Contrainte de Bande Passante". In: *ComPAS'13 / RenPar'21 - 21es Rencontres francophones du Parallélisme*. 2013.

[9] Frédéric Desprez, Sylvain Gault, and Frédéric Suter. *Scheduling/Data Management Heuristics*. Tech. rep. Deliverable D3.1 of MapReduce ANR project. URL: http://hal.inria.fr/hal-00759546.

[10] Sylvain Gault and Frédéric Desprez. *Dynamic Scheduling of MapReduce Shuffle under Bandwidth Constraints*. Tech. rep. Inria/RR-8574.

[11] John Von Neumann. "First Draft of a Report on the EDVAC". In: *IEEE Annals of the History of Computing* 15.1 (1993).

[12] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (Aug. 2008), pp. 1–70.

[13] Charles E. Leiserson. "Fat-trees: Universal Networks for Hardware-Efficient Supercomputing". In: *IEEE Transactions on Computers* C-34.10 (1985), pp. 892–901.

[14] D.P. Anderson. "BOINC: A System for Public-Resource Computing and Storage". In: *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. Nov. 2004, pp. 4–10.

[15] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The Hadoop Distributed File System". In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (May 2010), pp. 1–10.

[16] Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, and Diana Moise. "BlobSeer: Next Generation Data Management for large Scale Infrastructures". In: *Journal of Parallel and Distributed Computing* 2 (2010), pp. 168–184.

[17] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Pearson Education, 2002.

[18] Julien Bigot, Zhengxiong Hou, Christian Pérez, and Vincent Pichon. "A Low Level Component Model Easing Performance Portability of HPC Applications". In: *Computing* 96.12 (2014), pp. 1115–1130. ISSN: 0010-485X.

[19] Julien Bigot and Christian Pérez. "High Performance Scientific Computing with Special Emphasis on Current Capabilities and Future Perspectives". In: vol. 20. Advances in Parallel Computing. IOS Press, 2011. Chap. On High Performance Composition Operators in Component Models, pp. 182–201.

[20] Message P Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. Knoxville, TN, USA, 1994.

[21] OMG. *The Common Object Request Broker: Architecture and Specification V3.3*. OMG Document formal/2012-11-12. Nov. 2012.

[22] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. "Basic Linear Algebra Subprograms for Fortran Usage". In: *ACM Transactions on Mathematical Software* 5.3 (Sept. 1979), pp. 308–323. ISSN: 0098-3500.

[23] Judit Planas, Rosa M Badia, Eduard Ayguadé, and Jesus Labarta. "Hierarchical Task-based Programming with StarSs". In: *International Journal of High Performance Computing Applications* 23.3 (2009), pp. 284–299.

[24] Dolbeau Romain, Bihan Stéphane, and François Bodin. "HMPP^{TM}: A Hybrid Multi-core Parallel Programming Environment". In: *First Workshop on General Purpose Processing on Graphics Processing Units*. 2007.

[25] OpenACC Working Group et al. *The OpenACC Application Programming Interface*. 2011.

[26] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, Gaël Utard, Ramamurthy Badrinath, and Louis Rilling. "Kerrighed: A Single System Image Cluster Operating System for High Performance Computing". In: *Euro-Par 2003 Parallel Processing*. Springer, 2003, pp. 1291–1294.

[27] Amnon Barak, Shai Guday, and Richard G Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer-Verlag New York, Inc., 1993.

[28] Garrick Staples. "TORQUE Resource Manager". In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06. New York, NY, USA: ACM, 2006. ISBN: 0-7695-2700-0.

[29] Christine Morin. "XtreemOS: A Grid Operating System Making your Computer Ready for Participating in Virtual Organizations". In: *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on*. IEEE. 2007, pp. 393–402.

[30] Peter Mell and Tim Grance. "The NIST Definition of Cloud Computing". In: (2011).

[31] Franck Cappello, Frédéric Desprez, Michel Dayde, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Nouredine Melab, Raymond Namyst, Pascale Primet, Olivier Richard, et al. "Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform". In: *6th IEEE/ACM International Workshop on Grid Computing-GRID 2005*. 2005.

[32] Katarzyna Keahey, Mauricio Tsugawa, Andrea Matsunaga, and José AB Fortes. "Sky Computing". In: *Internet Computing, IEEE* 13.5 (2009), pp. 43–51.

[33] Shadi Ibrahim, Hai Jin, Lu Lu, Bingsheng He, Gabriel Antoniu, and Song Wu. "Maestro: Replica-Aware Map Scheduling for MapReduce". In: *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. Ieee, May 2012, pp. 435–442. ISBN: 978-1-4673-1395-7. DOI: 10.1109/CCGrid.2012.122.

[34] Matei Zaharia, Dhruba Borthakur, J.S. Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. "Job Scheduling for Multi-User MapReduce Clusters". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55, Apr* (2009), pp. 2009–55.

[35] Matei Zaharia and Dhruba Borthakur. "Delay Scheduling: A simple Technique for Achieving Locality and Fairness in Cluster Scheduling". In: *Proceedings of the 5th European Conference on Computer Systems*. 2010. ISBN: 9781605585772.

[36] Shadi Ibrahim, Hai Jin, Lu Lu, Song Wu, Bingsheng He, and Li Qi. "LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud". In: *Proc. of the Second IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Indianapolis, IN: IEEE, Nov. 2010, pp. 17–24. ISBN: 978-1-4244-9405-7. DOI: 10.1109/CloudCom.2010.25.

[37] Jiahui Jin, Junzhou Luo, Aibo Song, Fang Dong, and Runqun Xiong. "BAR: An Efficient Data Locality Driven Task Scheduling Algorithm for Cloud Computing". In: *Proc. of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. Newport Beach, CA: IEEE, May 2011, pp. 295–304. DOI: 10.1109/CCGrid.2011.55.

[38] Cristina L. Abad, Yi Lu, and Roy H. Campbell. "DARE: Adaptive Data Replication for Efficient Cluster Scheduling". In: *IEEE International Conference on Cluster Computing*. 2011, pp. 159–168. ISBN: 9780769545165.

[39] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. "Quincy: Fair Scheduling for Distributed Computing Clusters". In: *Symposium on Operating Systems Principles*. 2009, pp. 261–276.

[40] Richard E. Ladner and Michael J. Fischer. "Parallel Prefix Computation". In: *Journal of the ACM* 27.4 (Oct. 1980), pp. 831–838. ISSN: 00045411.

[41] Ronald Rivest. "The MD5 Message-Digest Algorithm". In: *Request for Comments (RFC 1321)* (1992).

[42] Secure Hash Standard. "FIPS 180-1". In: *Secure Hash Standard, NIST, US Dept. of Commerce, Washington DC April* 9 (1995), p. 21.

[43] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. "Improving MapReduce Performance in Heterogeneous Environments." In: *OSDI*. 2008.

[44] Rohan Gandhi and Amit Sabne. *Finding Stragglers in Hadoop*. Tech. rep. 2011.

[45] YC Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. "Skewtune: Mitigating Skew in MapReduce Applications". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012, pp. 25–36. ISBN: 9781450312479.

[46] Yen-Liang Su, Po-Cheng Chen, Jyh-Biau Chang, and Ce-Kuen Shieh. "Variable-Sized Map and Locality-Aware Reduce on Public-Resource Grids". In: *FGCS* 27.6 (June 2011), pp. 843–849.

[47] Quan Chen, Minyi Guo, Qianni Deng, Long Zheng, Song Guo, and Yao Shen. "HAT: History-Based Auto-Tuning MapReduce in Heterogeneous Environments". In: *The Journal of Supercomputing* 64.3 (Sept. 2013), pp. 1038–1054.

[48] Sangwon Seo, Ingook Jang, Kyungchang Woo, Inkyo Kim, Jin-Soo Kim, and Seungryoul Maeng. "HPMR: Prefetching and Pre-shuffling in Shared MapReduce Computation Environment". In: *Proc. of the 2009 IEEE International Conference on Cluster Computing (Cluster)*. New Orleans, LA, Sept. 2009. ISBN: 9781424450121.

[49] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. "Twister: A Runtime for Iterative MapReduce". In: *IEEE International Symposium on High Performance Distributed Computing*. 2010, pp. 810–818.

[50] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. "Evaluating MapReduce for Multi-core and Multiprocessor Systems". In: *2007 IEEE 13th International Symposium on High Performance Computer Architecture* (2007), pp. 13–24.

[51] Justin Talbot, Richard M Yoo, and Christos Kozyrakis. "Phoenix++: Modular MapReduce for Shared-Memory Systems". In: *Proceedings of the Second International Workshop on MapReduce and its Applications*. 2011. ISBN: 9781450307000.

[52] Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, and Dhiraj Sehgal. "Hadoop Acceleration through Network Levitated Merge". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11* (2011), p. 1.

[53] Xiaoyi Lu, Nusrat S. Islam, Md. Wasi-Ur-Rahman, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K. Panda. "High-Performance Design of Hadoop RPC with RDMA over InfiniBand". In: *2013 42nd International Conference on Parallel Processing* (Oct. 2013), pp. 641–650.

[54] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. "High Performance RDMA-Based Design of HDFS over InfiniBand". In: *2012 International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov. 2012), pp. 1–12.

[55] Wenbin Fang, Bingsheng He, Qiong Luo, and N.K. Govindaraju. "Mars: Accelerating MapReduce with Graphics Processors". In: *IEEE Transactions on Parallel and Distributed Systems* 22.4 (2010), pp. 608–620. ISSN: 1045-9219.

[56] Gilles Fedak, Haiwu He, and Franck Cappello. "BitDew: A Programmable Environment for large-scale Data Management and Distribution". In: *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE Press, 2008, pp. 1–12. ISBN: 978-1-4244-2835-9.

[57] Bing Tang, Mircea Moca, Stéphane Chevalier, Haiwu He, and Gilles Fedak. "Towards MapReduce for Desktop Grid Computing". In: *Fifth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC'10)*. IEEE. Fukuoka, Japan, Nov. 2010, pp. 193–200.

[58] Bogdan Nicolae, Diana Moise, and Gabriel Antoniu. "BlobSeer: Bringing high Throughput under Heavy Concurrency to Hadoop Map-Reduce Applications". In: *Parallel & Distributed* (2010).

[59] Yingyi Bu, Bill Howe, Magdalena Balazinska, and M.D. Ernst. "HaLoop: Efficient Iterative Data Processing on large Clusters". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 285–296.

[60] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. "iMapReduce: A Distributed Computing Framework for Iterative Computation". In: *Journal of Grid Computing* 10.1 (Mar. 2012), pp. 47–68. ISSN: 1570-7873.

[61] Charles E Leiserson, Ronald L Rivest, Clifford Stein, and Thomas H Cormen. *Introduction to Algorithms*. MIT press, 2001.

[62] Eslam Elnikety, Tamer Elsayed, and Hany E. Ramadan. "iHadoop: Asynchronous Iterations for MapReduce". In: *2011 IEEE Third International Conference on Cloud Computing Technology and Science* (Nov. 2011), pp. 81–90.

[63] Tyson Condie, Neil Conway, Peter Alvaro, J.M. Hellerstein, Khaled Elmeleegy, and Russell Sears. "MapReduce Online". In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2010, pp. 21–21.

[64] Ahmed M. Aly, Asmaa Sallam, Bala M. Gnanasekaran, Long-Van Nguyen-Dinh, Walid G. Aref, Mourad Ouzzani, and Arif Ghafoor. "M3: Stream Processing on Main-Memory MapReduce". In: *2012 IEEE 28th International Conference on Data Engineering*. Ieee, Apr. 2012, pp. 1253–1256. ISBN: 978-0-7695-4747-3.

[65] Heshan Lin, X. 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*. ACM, 2010, pp. 95–106.

[66] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. "Pig Latin: A Not-So-Foreign Language for Data Processing". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008, pp. 1099–1110. ISBN: 9781605581026.

[67] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. "Hive - A Warehousing Solution Over a Map-Reduce Framework". In: *Proceedings of the VLDB Endowment*. Vol. 2. 2. 2009, pp. 1626–1629.

[68] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN: 0321245628.

[69] Luis André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 2013.

[70] Albert Greenberg, Srikanth Kandula, David A. Maltz, James R. Hamilton, Changhoon Kim, Parveen Patel, Navendu Jain, Parantap Lahiri, and Sudipta Sengupta. "VL2: A Scalable and Flexible Data Center Network". In: *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*. 2009, pp. 51–62.

[71] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. "A Scalable, Commodity Data Center Network Architecture". In: *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. ACM Press, 2008, pp. 63–74.

[72] Luiz Angelo Steffenel. "Modeling Network Contention Effects on All-to-All Operations". In: *2006 IEEE International Conference on Cluster Computing* (2006).

[73] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. "Towards Predictable Datacenter Networks". In: *Proceedings of the ACM SIGCOMM 2011 Conference*. New York, New York, USA, 2011, p. 242.

[74] Nikolay Grozev and Rajkumar Buyya. "Performance Modelling and Simulation of Three-Tier Applications in Cloud and Multi-Cloud Environments". In: *The Computer Journal* (2013).

[75] Herodotos Herodotou. *Hadoop Performance Models*. Tech. rep. 2011, pp. 1–16.

[76] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta. "Using Realistic Simulation for Performance Analysis of MapReduce Setups". In: *Large-scale system and application performance*. 2009.

[77] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. "Starfish: A Self-tuning System for Big Data Analytics". In: *Conference on Innovative Data Systems Research*. 2011, pp. 261–272.

[78] Herodotos Herodotou and Shivnath Babu. "Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs". In: *Proceedings of the very large Data Bases Endowment*. Vol. 4. 11. 2011.

[79] Abhishek Verma, Ludmila Cherkasova, and RH Campbell. "ARIA: Automatic Resource Inference and Allocation for MapReduce Environments". In: *8th ACM international conference on Autonomic computing*. 2011, pp. 235–244.

[80] Gunho Lee. *Resource Allocation and Scheduling in Heterogeneous Cloud Environments*. Tech. rep. 2012.

[81] Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. "Brief Announcement: Modelling MapReduce for Optimal Execution in the Cloud". In: *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. 4. ACM, 2010, pp. 408–409.

[82] Archana Ganapathi, Yanpei Chen, Armando Fox, Randy Katz, and David Patterson. "Statistics-Driven Workload Modeling for the Cloud". In: *IEEE 26th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 2010, pp. 87–92.

[83] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael Jordan, and David Patterson. "Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning". In: *2009 IEEE 25th International Conference on Data Engineering*. Mar. 2009, pp. 592–603.

[84] Francis R. Bach and Michael I. Jordan. "Kernel Independent Component Analysis". In: *The Journal of Machine Learning Research* 3 (2003), pp. 1–48.

[85] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. "Characterizing Cloud Computing Hardware Reliability". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. New York, New York, USA, 2010, pp. 193–204.

[86] Dong Seong Kim, Fumio Machida, and Kishor S. Trivedi. "Availability Modeling and Analysis of a Virtualized System". In: *International Symposium on Dependable Computing*. Nov. 2009, pp. 365–371.

[87] Haiyang Qian, Deep Medhi, and Kishor Trivedi. "A Hierarchical Model to Evaluate Quality of Experience of Online Services hosted by Cloud Computing". In: *International Symposium on Integrated Network Management*. 2011, pp. 105–112.

[88] Astrid Undheim, Ameen Chilwan, and Poul Heegaard. "Differentiated Availability in Cloud Computing SLAs". In: Sept. 2011, pp. 129–136.

[89] Quan Chen, Daqiang Zhang, Minyi Guo, Qianni Deng, and Song Guo. "SAMR: A Self-Adaptive MapReduce Scheduling Algorithm in Heterogeneous Environment". In: *2010 10th IEEE International Conference on Computer and Information Technology*. Ieee, June 2010, pp. 2736–2743. ISBN: 978-1-4244-7547-6. DOI: 10.1109/CIT.2010.458.

[90] Bharadwaj Veeravalli, Debasish Ghose, Venkataraman Mani, and Thomas Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, 1996, p. 292.

[91] M. Frank and P. Martini. "Practical Experiences with a Transport Layer Extension for End-to-End Bandwidth Regulation". In: *Local Computer Networks, 1997. Proceedings., 22nd Annual Conference on*. Nov. 1997, pp. 337–346.

[92] Ian F Akyildiz, Jörg Liebeherr, and Debapriya Sarkar. "Bandwidth Regulation of Real-Time Traffic Classes in Internetworks". In: *Computer Networks and ISDN Systems* 28.6 (1996), pp. 855–872.

[93] Matthias Frank and Peter Martini. "Performance Analysis of an End-to-End Bandwidth Regulation Scheme". In: *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 1998. Proceedings. Sixth International Symposium on*. IEEE. 1998, pp. 133–138.

Webography

[94] NVIDIA. *Tesla K40 and K80 GPU Accelerators for Servers*. URL: http://www.nvidia.com/object/tesla-servers.html.

[95] University of California. *BOINC*. URL: http://boinc.berkeley.edu/.

[96] Hans Meuer, Jack Dongarra, Erich Strohmaier, and Horst Simon. *TOP500 Supercomputer Sites*. URL: http://www.top500.org/.

[97] Hadoop. *Welcome to Apache™ Hadoop®!* URL: http://hadoop.apache.org/.

[98] OAR team. *OAR [start]*. URL: http://oar.imag.fr/.

[99] Microsoft blog editor. *Windows Azure General Availability*. URL: http://blogs.microsoft.com/blog/2010/02/01/windows-azure-general-availability/.

[100] Amazon Web Services Inc. *Amazon EMR*. URL: https://aws.amazon.com/fr/elasticmapreduce/.

[101] Amazon Web Services Inc. *AWS | Amazon Elastic Compute Cloud (EC2) - Scalable Cloud Hosting*. URL: https://aws.amazon.com/ec2/.

[102] Hadoop. *PoweredBy - Hadoop Wiki*. URL: https://wiki.apache.org/hadoop/PoweredBy.

[103] Apache Software Foundation. *Apache Hadoop NextGen MapReduce (YARN)*. URL: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[104] Owen O'Malley and Arun Murthy. *Hadoop Sorts a Petabyte in 16.25 Hours and a Terabyte in 62 Seconds*. URL: https://developer.yahoo.com/blogs/hadoop/hadoop-sorts-petabyte-16-25-hours-terabyte-62-422.html.

[105] Owen O'Malley and Arun Murthy. *Winning a 60 Second Dash with a Yellow Elephant*. URL: https://s.yimg.com/lq/i/ydn/blogs/hadoop/yahoo2009.pdf.

[106] Owen O'Malley and Arun Murthy. *Scaling Hadoop to 4000 Nodes at Yahoo!* URL: https://developer.yahoo.com/blogs/hadoop/scaling-hadoop-4000-nodes-yahoo-410.html.

[107] Arun Murthy. *The Next Generation of Apache Hadoop MapReduce*. URL: https://developer.yahoo.com/blogs/hadoop/next-generation-apache-hadoop-mapreduce-3061.html.

[108] Yahoo! Inc. *Hadoop Tutorial*. URL: https://developer.yahoo.com/hadoop/tutorial/module4.html#tolerence.

[109] Microsoft. *HDInsight | Cloud Hadoop*. URL: http://azure.microsoft.com/en-us/services/hdinsight/.