CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **00**:1–31
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

BSP Cost and Scalability Analysis for MapReduce Operations

Hermes Senger[1]* Veronica Gil-Costa[2], Luciana Arantes[3], Cesar A. C. Marcondes[1], Mauricio Marín[4], Líria M. Sato[5], and Fabrício A.B. da Silva[6]

[1] *Federal University of São Carlos (UFSCar), São Carlos-SP, Brazil*
[2] *UNSL-CONICET, San Luis, Argentina*
[3] *Pierre and Marie Curie University (UPMC), LIP6/CNRS/INRIA/REGAL, Paris - France*
[4] *CeBiB, DIINF, Universidad de Santiago, Chile*
[5] *University of São Paulo, São Paulo - SP, Brazil*
[6] *Oswaldo Cruz Foundation (FIOCRUZ), Rio de Janeiro, Brazil*

SUMMARY

Data abundance poses the need for powerful and easy to use tools that support processing large amounts of data. MapReduce has been increasingly adopted for over a decade by many companies, and more recently it has attracted the attention of an increasing number of researchers in several areas. One main advantage is that the complex details of parallel processing, such as complex network programming, task scheduling, data placement, and fault tolerance are hidden in a conceptually simple framework. MapReduce is supported by mature software technologies for deployment in data centers such as Hadoop. As MapReduce becomes popular for high performance applications, many questions arise concerning its performance and efficiency. In this paper we demonstrated formally lower bounds on the isoefficiency function for MapReduce applications, when these applications can be modeled as BSP jobs. We also demonstrate how communication and synchronization costs can be dominant for MapReduce computations, and discuss the conditions under which such scalability limits are valid. To our knowledge, this is the first study that demonstrates scalability bounds for MapReduce applications. We also discuss how some MapReduce implementations such as Hadoop can mitigate such costs to approach linear, or near-to-linear speedups. Copyright © 2015 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Mapreduce; Hadoop; Scalability;

1. INTRODUCTION

During the last decade, the amount of data produced, stored, and processed has significantly increased. In December 2012, IDC published a report [43] which estimates that from 2005 to 2020 the "digital universe" (that is, all the digital data created, replicated and consumed) will grow from 130 Exabytes to 40,000 Exabytes (i.e., 10^{18} bytes), doubling roughly every two years. The digital

*Correspondence to: Corresponding address: hermes@dc.ufscar.br.

universe is made up of Web related data such as user activity traces and user generated data, different types of documents uploaded to the Web, images and videos recorded on mobile phones uploaded to Web applications such as YouTube, digital movies for high-definition TVs, banking data swiped in ATMs, security footage at airports and major events such as the Olympic Games, subatomic particle collisions recorded by the Large Hadron Collider at CERN, transponders recording highway tolls, voice calls zipping through digital phone lines, message texts, genome datasets (including private individual data), and many others. It is worth noticing that most of such data will be unstructured content, which require a large effort to analyze.

Scientists and engineers in many areas are becoming increasingly reliant on the storage and processing of massive data collections, which has been dubbed as the Fourth Paradigm [42]. Data explosion is a phenomenon witnessed in many areas. For instance the Large Hadron Collider (LHC), one of the most powerful instruments ever built to investigate matter, deploys 150 million sensors, delivering data 40 million times per second. Only working with less than 0.001% of the sensor stream data, LHC produced 25 Petabytes each year before replication (as of 2012). This becomes nearly 200 PB after replication. Examples abound in areas such as environmental and climate research, astronomy, healthcare, brain studies, and many others [42].

An interesting application area is genomics, where computational methods of data analysis provide the means of formulating novel hypotheses [49]. The sequencing of genomes is a pioneering application of Big Data. A single human genome is composed of about 3 billion base pairs of DNA, and Next-generation sequencing technologies make complete sequencing of genomes at large scale feasible in terms of cost and time. Large-scale characterization of the human genome involves the generation and interpretation of an enormous volume of data, on an unprecedented scale, and one of the potential benefits is personalized medicine for cancer patients [13]. Indeed, it is worth emphasizing that currently there are dozens of Mapreduce applications available to process genomic data in the cloud [69].

On a more day-to-day side, we are both consumers and contributors to the ever increasing data produced from massive Web applications such as social networks and search engine services. Experts conjecture that in the last decade the Web has been continuously expanding so that its overall size is doubled every six to eight months, while there are large population countries where Internet access is still below 20%.

Data abundance poses the need for powerful and easy to use tools that support storage, transmission, and processing of large amounts of data. MapReduce is a widely accepted programming model and an associated implementation framework for parallel processing over large amounts of data. It was originally proposed by Google [17] as a result of the experience acquired after implementing a large number of special-purpose off line applications that process large amounts of raw data, such as crawled documents, Web request logs, and many others. Usually such computations manipulate huge input datasets which are partitioned and distributed across dozens to thousands of machines in order to complete their processing in a reasonable time.

Later Yahoo! developed a powerful MapReduce execution platform called Hadoop which has become widely used by practitioners. It is an open source implementation of MapReduce, supported and distributed by Apache [5, 82, 83]. Currently, more frameworks implement MapReduce model, such as the AppEngine-MapReduce, an open-source library for executing MapReduce-style computations on the Google App Engine platform [35], and Amazon Elastic MapReduce, a Web service that supports the execution of MapReduce applications on its EC2 infrastructure [24].

The reported practice and experience show us that the MapReduce model of computation enables effective parallel processing of huge data sets on unprecedented scale. For instance, in 2008 MapReduce was used by Google to process more than 20 petabytes of data per day [18]. In the same year, Facebook reported the use of a large Hadoop cluster processing 21 Petabytes of stored data and running over 60 thousand tasks over 2000 machines with 22400 processing cores (1200 eight-core machines and 800 16-cores machines) [29].

The model encapsulates the 90's principles believed to be the most suitable ones to help users who are not specialists in parallel computing to handle and process huge volume of data in a way that is simple to understand and implement. It presents a programming abstraction that contains a well-defined computation and communication pattern, and it is supported by mature software technologies for deployment in data centers such as Hadoop, which provides distributed file system facilities to handle the data and makes the usually involved details of data distribution and parallel programming transparent to the user. Hadoop also provides facilities for automatic job scheduling, data placement and replication, fault tolerance, and in general it hides system issues associated with the efficient execution of large applications on data centers.

Certainly the MapReduce model is not intended to be suitable for all kind of applications but it is able to embody a large number of practical applications. Typically these applications are featured by a data set of size N that can be split in P chunks of size N/P, where P is the number of participating processors, and most of the processing can be made locally on the N/P-sized pieces of data in parallel. Many experimental studies have shown that these applications are capable of achieving higher performance than a single processor realization of the computational task performed on the data. Given that the design goals of platforms like Hadoop are to provide a general purpose abstraction to the user, speedups with respect to the sequential program should not be expected to be optimal in the sense that the computational task executed on P processors should run P times faster than on a single processor. In many scenarios, the experiments with real data show quite the opposite.

In this context, given the particular ways in which the MapReduce model organizes computation and communication, we believe it is relevant to study how efficiently the model is able to scale up with the number of processors in terms of speedup as the problem size N and number of processors P scale up. To the best of our knowledge, a rigorous analysis of scalability has not been accomplished. Since MapReduce becomes popular for high performance applications, many questions arise concerning its performance and efficiency. For instance, how many machines can be effectively used by an application? What amount of the overall computation can be efficiently performed by one single application on a platform with P machines? Which factors impact its scalability? In the best case, which asymptotic behavior can be expected in terms of scalability for a given application/platform pair? Under which conditions such optimal behavior can be achieved? This paper aims at evaluating the scalability of MapReduce/Hadoop applications, in particular when these applications can be modeled as Bulk Synchronous Parallelism (BSP) [81] jobs.

BSP was adopted to model MapReduce computations due to its great simplicity, power, and portability to model a wide variety of parallel computations executing on shared-nothing multiprocessor hardware. Also, BSP model prescribes that the next superstep only starts after completion of the barrier synchronization at the end of the previous superstep. This synchronous behavior allows BSP to mimic MapReduce computations, whose reduce tasks cannot execute until the last output key-value pair is emitted by the map tasks.

The main contributions of this paper include:

- From the theoretical perspective, we provide a BSP-based cost model and scalability analysis with asymptotic bounds for the isoefficiency of single round MapReduce computations;
- We identify the conditions (i.e., setup for the application and platform) under which the maximum scalability can be achieved. This study provides insights for application designers to devise more efficient and scalable applications
- A simulation-based study on the scalability of MapReduce applications running on real frameworks like Hadoop. Our study provides useful capacity planning tool for practitioners and administrators to predict performance and scalability for applications under many different scenarios.

The remainder of this paper is organized as follows. In Section 2, an overview about existing related work is presented. A brief review on MapReduce, BSP, and scalability analysis is presented in Section 3. Section 4 presents the main assumptions which provide the basis for the scalability analysis presented in Sections 5 and 6. A BSP cost model for MapReduce computations is presented in Section 5. In Section 6, a demonstration on how communication and synchronization costs can be dominant for MapReduce computations based on BSP is presented. Section 7 presents experimental results on the scalability of Hadoop applications. In Section 8, the validity and scope of both theoretical and experimental results are discussed. Finally, the conclusion is provided in Section 9.

2. RELATED WORK

MapReduce is a powerful programming paradigm for processing large amounts of data which has increasingly been adopted for near one decade by many companies [18]. It was proposed by Google as a way to distribute computational tasks which are run over large datasets. Over the years, it has attracted the attention of an increasing number of researchers from several areas. One main advantage is that the complex details of parallel processing, and network programming, are hidden in a conceptually simple framework. MapReduce has been integrated as a core component in various projects towards novel, alternative data analysis systems [2, 11]. Some good surveys are presented by Doulkeridis et al. [23] and Sakr et al. [72].

Dittrich et al. present Hadoop++ [21], which includes additional functions designed to improve run time of tasks related to indexing and join processing. Later, the authors present HAIL [22] devised to improve Hadoop++ by reducing the index creation times using the replicas maintained in Hadoop by default for fault-tolerance. Nykiel et al. [66] present the MRShare framework, which transforms a batch of queries into a new batch that will be executed by merging jobs into groups and evaluating each group as a single query. Elghandour et al. [26] present ReStore which is an extension of Pig that re-uses intermediate results.

Jahani et al. [45] propose to use Manimal [9], which automatically analyzes the programming logic of MapReduce programs and applies appropriate optimizations. Herodotou et al. [41] present the What-If engine composed by a cost-based optimizer and a profiler used to collect detailed statistical information.

In [25], Ekanayake et al. present Twister, a lightweight MapReduce implementation with a set of extensions and architectural improvements that led to performance improvements over Hadoop. Among these improvements, Twister distinguishes between static and dynamic data in each iteration and supports long running (cacheable) tasks with static data that can be cached for performance. It also uses publish and subscribe messaging for the communication and data transfer that is more efficient than transferring data over the file system.

Regarding the imbalance in the workload of machines, Gufler et al. in [38] propose to estimate the cost of the tasks that are distributed to the reducers. Le et al. [57] presents a scheme that examines keys in a continuous fashion and assigns tasks to reducers with constrained version of the online minimum makespan problem [28]. Kolb et al. [51] propose and evaluate two load balancing approaches called BlockSplit and PairRange which are capable of dealing with skewed data distributions. The SkewTune system implemented by Kwon et al. [55] re-assign tasks, when a machine becomes idle, to fully utilize the machine in the cluster.

If the map function is slow, the whole running time will be affected. To overcome this problem, the works presented by Mohamed et al. [64, 65] propose MapReduce overlapping using MPI. These works present an adapted structure of the MapReduce programming model, where the map and the reduce functions run concurrently in parallel by exchanging partial intermediate data between them using MPI. In other words, the shuffling phase is merged with the mapping phase and reduce tasks do not wait until the map tasks finish their work. Experiments report good speedup.

There are various works dedicated to analyze the performance and efficiency of MapReduce. McCreadieIn et al. [62] implement and compare four text distributed indexing strategies and their scalability on the Terrier IR platform [68], with 30 processing machines. Experimental results show sublinear speedups. Similarly, the authors in [63] adapt the single-pass indexing [40] for MapReduce where posting lists are compressed to minimize the data that is transferred between map and reduce tasks. Results show that the performance of MapReduce scales close to linearly. Akritidis et al. [4] propose using MapReduce to compute metrics which evaluate the research activity of a scientist when data is huge and cannot be handled efficiently by a single workstation.

A performance study on Hadoop implementation of MapReduce is presented in [47]. The authors analyze how different factors such as I/O mode, indexing, data parsing, grouping scheme, etc., affects the performance of Hadoop. Experiments conducted on a 100-node cluster of Amazon EC2 showed that direct I/O outperforms streaming I/O by 10%. Additionally, by enabling an index, the selection task of MapReduce improves by a factor of 2.5 and the join task is improved by a factor of 10. The work in [19] evaluated the MongoDB, which is a NoSQL datastore, with Hadoop. Scalability and fault tolerance are the main metrics used for performance evaluation. The work in [39] analyze the scalability of MapReduce by executing the TeraSort benchmark on a 200-node cluster of Amazon EC2. Results show that superlinear speedup reaches a maximum value. However, these works present only experimental results and no cost analysis is given.

In previous work [77, 74, 75], we studied the scalability of Bag-of-Tasks applications executing on master-slave platforms with a centralized data repository to distribute input files to be processed, under several circumstances and different communication infrastructures. In a subsequent paper, we demonstrated that scalability bounds for the execution of Bag-of-Tasks applications on hierarchical platforms can be one order of magnitude [16].

2.1. Modeling MapReduce Applications

A rigorous analysis of the capabilities and limitations of MapReduce should start by putting it on a strong theoretical framework. One of the earliest theoretical models for parallel computing is PRAM [32]. Although PRAM allows to focus on a few essential characteristics of parallel computing, it assumes that the cost for communication operations is similar to elementary computations, and the number of processors is unlimited. Both are unrealistic assumptions for current parallel computers. A further and more appropriate model for parallel computing is Bulk Synchronous Parallelism (BSP) [81], which can be regarded as an abstract model for the hardware and software of parallel computing that is both architecture-independent and scalable [79].

The coarse-grained multicomputer (CGM) [20] model is an extension of BSP which assumes p processors, each one with $O(N/P)$ local memory and the size of the input N much larger than P. Then, it restricts the local memory access and the communication to $O(N/P)$ making it suitable for well-balanced applications.

A pioneer work to model MapReduce was proposed by Feldman et al. in [31]. In this work, authors define a subclass of applications named *massive, unordered and distributed algorithms* (MUD) for processing data streams and assume exponential computation cost due to the use of Stavitch's theorem. The model is too restrictive in the type of algorithms that can be designed because it is limited to one MapReduce round and input is assumed to come from separate streams. It also puts limits to memory size or I/O size for reducers.

In a further work, Karloff et al. proposed a model of computation for MapReduce [48]. The mode better captures some specifics of MapReduce (compared to Feldman's model), since it can have multiple rounds. However, they limit the number of processors of the architecture. Also, the size of input and output of reducers is limited to $O(N^{1-\epsilon})$ for some small constant $\epsilon > 0$, so that the memory of each reducer and the memory of each map are both limited to $O(\log N)$-words. Such limitations make the model more complicate and less generic.

In [34], Goodrich et al. describe some MapReduce algorithms by applying BSP and CRCW PRAM models. This work differs from the previous in the way communication cost is calculated. It proves that every BSP algorithm can be implemented as MapReduce using the reduce phase and setting the map function to the identity. In this case, the size of data received or sent by each reduce task is limited to an upper bound M on the I/O buffer size for all reducers to enforce parallelism.

One major drawbacks of MapReduce is that jobs must dump data to the distributed file system before they can be read by the next MapReduce job. To avoid this problem, Fegaras in [30] proposes to use the BSP model as an alternative to MapReduce.

The HAMA [1] is a general BSP framework running on top of Hadoop. It was developed as a framework for massive matrix and graph computations using MapReduce and BSP [76]. In addition, Google's Pregel [60] is a BSP implementation for graph analysis. An alternative to Pregel, Apache's Giraph [33] presents an open-source framework.

More recently, Pace [70] showed that BSP can simulate any MapReduce algorithm with similar asymptotic cost, as well as any BSP algorithm can be simulated (or implemented) as MapReduce framework, highlighting some difficulties in the later case when data should be communicated between rounds. In this work, no limitation is put on the framework. However, it does not consider cost of sort/merge in the intermediate phase.

Despite the popularity and wide adoption of MapReduce, and of near one decade of existence with a huge number of framework implementations and research works, the scalability limits of MapReduce still remains an open research topic to be addressed. To our knowledge, there is no rigorous analysis of its scalability limits such as presented in this paper. In fact, the prominence of MapReduce for parallel computing is a strong motivation to analyze its scalability. Therefore, this paper presents a relevant step towards modeling and analyzing its scalability limits. In order to put MapReduce on sound theoretical foundations of parallel computing, a well established framework such as BSP [81] was employed. BSP was chosen because is a simple and concise parallel computing model, and has a cost model that makes it simple to design, analyze and optimize massively parallel algorithms.

3. BACKGROUND

In this section we present a brief review about both MapReduce and the Bulk Synchronous Parallelism (BSP) model.

3.1. MapReduce Overview

MapReduce is a programming model and an associated implementation framework for processing and generating large datasets on potentially large clusters of computers. It was originally proposed by Google [17] as a result from the experience acquired after implementing a large number of special-purpose off line applications that process large amounts of raw data, such as crawled documents, Web request logs, and many others, and to compute various kinds of derived datasets, such as inverted indices, various representations of the graph structure of Web documents, summaries of Web pages crawled per host, the set of most frequent queries in a given day, etc. Usually such computations manipulate huge input datasets which are partitioned and distributed across hundreds or thousands of machines in order to complete the processing of data in a reasonable time. Despite its apparent complexity, MapReduce authors, J. Dean and S. Ghemat, observed that the majority of such applications are conceptually straightforward so that a new programming model and an implementation framework could be devised, aiming to facilitate the job of parallelizing and implementing new applications. The goal of it is, thus, to hide details of parallel execution from the user, who should focus on data processing strategies.

One of the main features of MapReduce model is simplicity. A MapReduce job is a unit of work to be executed. In summary, it consists of the input data, a map function, a reduce function, and configuration information. The execution of the job is then automatically parallelized on a large number of clusters of commodity or cloud environments machines. The basic unit of information is a $< key, value >$ pair and both the map and reduce functions receive as input a set of $< key, value >$ pairs. Dean and Ghemawat claim that many applications can be implemented in a natural way using this programming model [17]. In fact, considerable work have studied join processing

[67, 59, 84], processing of massive graphs [78, 80, 6], classification and regression [71, 56], and other algorithmic studies on MapReduce [12, 46, 50].

Basically, the data input is split into M units. Each data unit is processed by a *Map* task that invokes the *map* function. *Map* tasks are processed in parallel. An intermediate step, named *shuffle*, is automatically executed by the runtime system which groups together all intermediate values associated with the same intermediate key and passes them to the subsequent reduce function. In the shuffle phase, the intermediate keys are automatically sorted, partitioned into R pieces, and distributed to the R *Reduces* tasks that merge all the received data and then invoke the *reduce* function. Similarly to the *Map* tasks, *Reduces* tasks also execute in parallel. Both the initial input and the final output of a MapReduce application are stored in a fault tolerant distributed file system, such as HDFS [5] or Google File System [17]. However, intermediate keys are stored *Map* tasks' local disk.

A single MapReduce cycle is shown in Figure 1 with tree *Map* tasks and 2 *Reduce* tasks. Notice that a MapReduce application may be composed of one or more of such cycles.

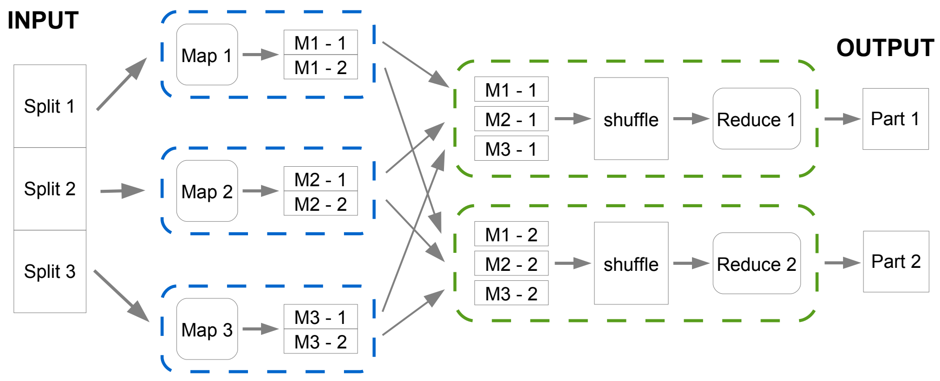

Figure 1. MapReduce cycle

MapReduce frameworks are usually based on a master-slave architecture. A job is submitted by a user to a master node which selects idle workers and assigns a *Map* or *Reduce* task to them. MapReduce automatically handles failures. If a machine crashes, MapReduce reruns its tasks on a second machine. Furthermore, if the task is performing poorly, MapReduce runs a speculative copy of it (also called a *backup task*) on another machine to finish the computation faster. On the other hand, master failures are not explicitly managed.

Currently, there are many frameworks that implement MapReduce model. For instance, AppEngine-MapReduce is an open-source library for executing MapReduce-style computations on the Google App Engine platform [35]. Amazon Elastic MapReduce (Amazon EMR) is a Web service that supports the execution of MapReduce applications on its EC2 infrastructure [24]. Also, it is possible to deploy local MapReduce clusters with the support of frameworks like Hadoop.

Hadoop Mapreduce [5], [83] is a wide adopted open-source implementation of MapReduce proposed by Apache which exploits Hadoop Distributed File System (HDFS). Under HDFS, each file in the system is stored as a collection of equal-sized data blocks and provides high data availability and reliability through data replication. The replication degree is specified by a replication factor (3 by default).

To control task execution, Hadoop Mapreduce has a single JobTracker (master process) that manages job status and performs task scheduling. On each worker machine, a process (TaskTracker) tracks the available execution slots. In order to overlap computation and I/O, several map and reduce tasks can concurrently run on each TaskTracker. Hence, whenever a TaskTracker detects an empty execution slot on its machine, it contacts the JobTracker for a new task assignment. If the latter concerns a *Map* task, the JobTracker, taking into consideration the placement of the splits, choose the task, whose input data (split) is closer to the TaskTracker machine in question. On the other hand, such a locality approach can not be applied to *Reduce* tasks.

3.2. The BSP model

Bulk Synchronous Parallelism (BSP) is a model proposed by Valiant [81] for the hardware and software of parallel computing that is both architecture-independent and scalable [79]. Under the Bulk Synchronous Parallelism (BSP), a computer can be specified by p processors, each being capable of performing one elementary operation or accessing a local memory in one time unit. Processors communicate by sending a data word to every other processor in g time units, and a barrier mechanism is capable of synchronizing all the processors in l time units. BSP computation is organized as a sequence of *supersteps*. During a superstep, the processors may perform computations on local data and/or send messages to other processors. Messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronization of the processors. The underlying communication library ensures that all messages are available at their destinations before starting the next superstep. The effect of the computer architecture is expressed by the parameters g and l, which are increasing functions of p. Such values, along with the processor's speed s (e.g. Mflops) can be empirically determinate for each parallel computer by executing benchmark programs at installation time.

The running time for one BSP superstep can be expressed as

$$T_{sstep} = w + g \cdot h + l, \tag{1}$$

where w is the maximum number of operations performed by any single processor within the superstep, h is the maximum number of messages sent or received by any processor.

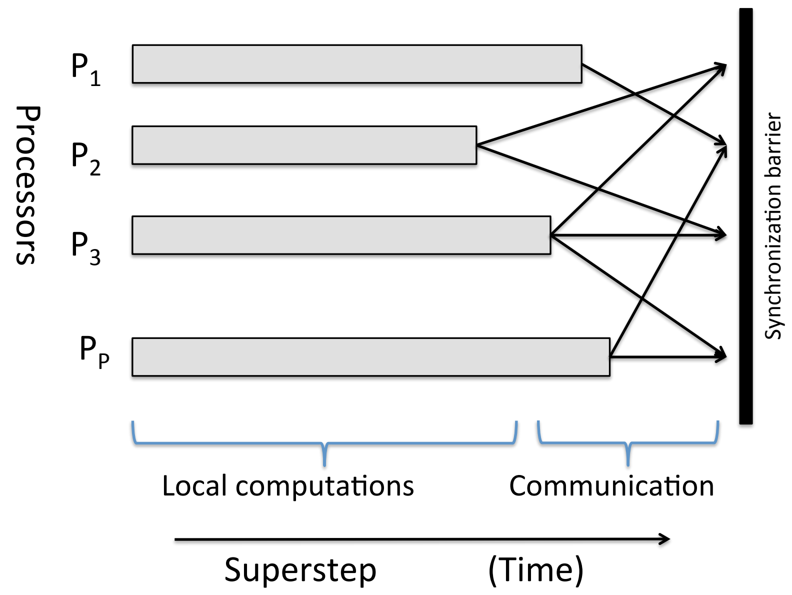

Figure 2. BSP model.

3.3. *Speedup and Isoefficiency*

First, we review the concept and formulation of the isoefficiency metric, as defined in [36]. One limiting factor for the scalability of parallel systems is the overhead. Let T_S be the execution time for the best known sequential algorithm to solve a problem, while T_P is the time spent for a parallel execution to solve the problem on a parallel computer with P processors. Then, the total time collectively spent by all the processing elements is $P.T_P$ time units. T_S time units are spent with useful work, and the remainder is overhead. Thus, total overhead of a parallel system can be expressed by an *overhead function*

$$T_0 = P.T_P - T_S. \tag{2}$$

Similarly to [36], we assume that each basic computation step of an algorithm takes one time unit to execute. Other constant-based costs, such as the cost to transmit one word or to set up a message for transmission, can be normalized with respect to the basic computation time. In this case we can assume that the workload W is $\Theta(T_S)$, and parallel runtime can be defined as

$$T_P = \frac{T_S + T_0(W, P)}{P}, \tag{3}$$

and the notion of scalability can be measured by the speedup (S), defined as:

$$S = \frac{W}{T_P} = \frac{P.T_S}{W + T_0(W, P)}. \tag{4}$$

Once speedup has been defined, we can define scalability as "the system ability to increase speedup as the number of processors increase" [53]. Efficiency (E) can then be denoted as:

$$E = \frac{S}{P} = \frac{W}{W + T_0(W,P)} = \frac{1}{1 + {T_0(W,P)}/{W}}. \tag{5}$$

For scalable parallel systems, efficiency can be maintained at some desired level (between 0 and 1) if the ratio ${T_0(W,P)}/{W}$ can be maintained at some fixed value as the number P of processors is increased. Depending upon the parallel system and on the algorithm, W must grow exponentially to keep E constant. It means that this system needs enormous workloads to scale. On the other hand, if the growing rate of W needs to grow less or equal than $\mathcal{O}(P)$, then the system is highly scalable. With this idea in mind, Kumar and Rao proposed the isoefficiency concept [53]. The basic idea is to fix the efficiency and measure how much work must be increased to keep the efficiency unchanged as the number of machines scale up. Then, isolating W from Equation 5, we obtain the following expression for the *isoefficiency function*:

$$f(P) = KT_0(W,P) = \frac{E}{1-E}T_0(W,P). \tag{6}$$

where $K = {E}/{(1-E)}$ is a constant value dependent on the desired efficiency level. Thus, the isoefficiency function $f(P)$ relates the number of machines P to the amount of work needed to maintain the efficiency. It shows the growth rate of W necessary to maintain the desired efficiency value as P increases.

4. ASSUMPTIONS ABOUT THE PLATFORM AND APPLICATION MODELS

In this section we present the main assumptions regarding the platform and application models, which provide the basis for the scalability analysis presented in Sections 5 and 6. We should point out that such assumptions are not necessarily valid for any Hadoop application (e.g., the application presented in Section 7).

In Section 5 and Section 6, our objective is to investigate the scalability asymptotic bounds for computations based on pure MapReduce model, as originally proposed by Dean and Ghemawat (2004). Thus, in order to devise a simple and tractable model which captures only the essential characteristics of MapReduce model, we abstracted from many optimization and implementation details which are usually realized in frameworks such as Hadoop. Furthermore, assumptions such as homogeneous processors, perfect load balancing, and no disk overhead are considered in order to show under which conditions the best possible scalability could be achieved for MapReduce computations.

4.1. Platform Model

For the theoretical analysis presented in this paper, we assume a platform composed of $P = \{p_1, \ldots, p_p\}$ homogeneous processors. Each processor performs at least a map and a reduce task. The input collection is composed of $N = n_1, \ldots, n_n$ data items which are evenly (and previously) distributed among the processors and stored on their respective local disks.

Regarding I/O processing, it is worth emphasizing that, in the best possible case, local I/O writing may be deferred until the disk is available [73] due to I/O caches. Therefore, under specific conditions it is possible to assume that local I/O processing time is negligible. In [47], it was shown that local HDFS read operations can have a similar behaviour through the implementation of *direct I/O*. We should also mention that block replication in HDFS can be deferred as well. Indeed, in [47], the authors concluded that the I/O mode (direct I/O vs. streaming I/O for remote reads) is not a major factor in the performance evaluation of MapReduce/Hadoop. In this paper, we consider that I/O processing time both after *Map* (local) and *Reduce* (Distributed File System) steps are not relevant for the scalability analysis presented in the following sections. The rationale for this come from some optimizations which are commonly implemented in map reduce frameworks such as Hadoop:

- First, computation is co-located with input data, i.e., map tasks are scheduled to execute in the same nodes were their input data is located. MapReduce assumes that each node has its own local disk, which can be used to store input data. In the best possible case all map tasks are local, and there are no delays in task execution due to input data transfer. Notice that input data sets are uploaded and spread over the local disks of the processing nodes (and then replicated) by the HDFS.
- Disk latency may be significantly reduced when sequential read/write access pattern is dominant. Sequential disc operation may be as much as 150,000 times faster than random access to disks in modern computers [44], which makes input data reading before map phase and output after reduce phase very efficient, even for less computational intensive map tasks [58].
- For well tuned applications, it is possible to minimize the cost of writing map outputs to the disk at the end of the Map phase. Map outputs can be written to a in-memory circular buffer to be transmitted to reduce tasks. When this buffer fills up, data is spilt [†] to the disk. Furthermore, disk access cost can be reduced by compressing the spilt data [‡]. In the best possible case, for well tuned applications, the disk writing cost at the end of map tasks can be neglected.

4.2. Application Model

We also define a set of assumptions related to MapReduce applications for the theoretical analysis that follows. We consider that number of map tasks is greater than the number of processors $(M > P)$, otherwise there will be idle processors when the number of processors increase. We also assume that the number of data items is significantly greater than the number of maps $(N \gg M)$, so that each map task executes a significant amount of work. We assume that each output data produced by one map is sent only once, for one reduce task (i.e., there is no replication of map outputs to more than one reducer). In other words, we assume that the *replication rate* of MapReduce algorithms considered in the theoretical analysis is 1. The replication rate of a MapReduce algorithm gives the average number of reducers each input is sent to. The replication rate is highly impacting on the amount of data sent during the intermediate phase, and on the application scalability [3].

[†] Disk writings in this case are named *spill* in Hadoop related literature
[‡] Data compression in this case can be enabled by the parameter `mapped` thus preventing from disk writing [83]

5. ESTIMATING COST FOR MAPREDUCE OPERATIONS BASED ON BSP

One purpose of this study is to assess the scalability limits of MapReduce applications when those applications can be modeled as BSP jobs. Such limits can be expressed by the lower bounds on the isoefficiency function under specific circumstances. First, we assume a MapReduce application composed of one single round of a map step and a reduce step executing on P homogeneous and dedicated processors that communicate through an homogeneous interconnection network (e.g., similar to computers in a datacenter site). In the following, we propose a definition of Scalable MapReduce Computation (SMC).

Definition 5.1. Scalable MapReduce Computation (SMC): The input data to be processed by a SMC is composed by N data items previously copied on local disks of P homogeneous and dedicated processors. For the execution of the map phase, all P processors receive the same amount of data items to execute and every data item takes the same time to process (*i.e.* no data skewness), so that the load is perfectly balanced among the processors. Also, we assume that the map operation performed on each data item (*i.e.* on each key-value pair) is memoryless and only depends on this data item (*i.e* no memory is maintained along several invocation of the map function. Likewise, for the execution of the reduce phase all the P processors receive the same amount of work to execute so that load is also perfectly balanced within this phase. Under such assumptions, the duration of the map phase is minimum, as well as the duration of the reduce phase. As stated in subsection 4.2, the replication rate of a SMC is 1.

Given this definition, we can find a lower bound on the isoefficiency function for MapReduce applications as follows.

Lemma 5.1. *Given a communication infrastructure that takes time T_{Comm} to send data from map to reduce tasks and takes time T_{Sync} to synchronize the end of the map stage and the beginning of the reduce stage in a MapReduce round, the lower bound on the isoefficiency function for a MapReduce application running on this platform with P processors is $\Omega(P + T_{Comm} + T_{Sync})$.* [§]

Proof
First, we simulate a SMC as a BSP computation composed of two supersteps. The first superstep simulates the map phase, and the second simulates the reduce phase, as depicted in Figure 3. A similar modeling is proposed in [70]. In the first superstep, each processor p_i performs the map task with $\Theta({}^{N}/{}_{P})$ local data, then sorts the results with a cost of $\mathcal{O}(\frac{N}{P}\log\frac{N}{P})$ and sends $\mathcal{O}(P)$ messages of size $\mathcal{O}(\frac{N}{P^2})$ to every processor p_j as input for the reduce tasks. We should point out that the sort (resp., merge) of the shuffle phase is considered to take part at the map (resp., reduce) superstep.

In the second superstep, each processor receives $\mathcal{O}(P(N/P^2))$ data, merges the results in time $\mathcal{O}(\frac{N}{P})$, and performs the reduce task in time $\mathcal{O}(\frac{N}{P})$. Afterwards, a disk operation can be required.

The problem size must grow at least as $\Omega(P)$, otherwise eventually there will be idle processors as P increases. Let T_S be the runtime of the best-known sequential algorithm, which is a function of N, the number of input data items to be processed, and let T_P be the parallel runtime for a SMC running on P processors. As each input data item requires at least one computing operation, we have

[§]The communication time (T_{Comm}) and the synchronization time (T_{Sync}) will be discussed in details in Section 6

```
SStep:1
       P1              P2              ...    Pp
    M(N/P)          M(N/P)            ...   M(N/P)  --> map operation
     sort            sort             ...    sort   --> cost:N/P logN/P
   send(p1,N/P^2)  send(p1,N/P^2)  ... send(p1,N/P^2)
   send(p2,N/P^2)  send(p2,N/P^2)  ... send(p2,N/P^2)
     ...             ...                  ...
   send(pp,N/P^2)  send(pp,N/P^2)  ... send(pp,N/P^2)
-------------------------------------  Synchronization
SStep:2
   recv(p1,N/P^2)  recv(p1,N/P^2)  ... recv(p1,N/P^2)
   recv(p2,N/P^2)  recv(p2,N/P^2)  ... recv(p2,N/P^2)
    ...             ...                  ...
   recv(pp,N/P^2)  recv(pp,N/P^2)  ... recv(pp,N/P^2)
   merge           merge           ... merge    --> cost: N/P
   R(N/P)          R(N/P)          ... R(N/P)
   disk(D)         disk(D)         ... disk(D)  --> this operation
                                                    can be neglected
```

Figure 3. One MapReduce round represented as two BSP supersteps: columns illustrate operations performed by each processor.

$T_S = \Omega(N)$. Remind that the amount of data items (N) need to grow at least as $\Theta(P)$, otherwise there will be idle processors as P increases and the efficiency cannot be maintained. Thus, we can assume that $T_S = \Theta(P)$. Thus, by replacing these cost values in Equation 2, the overhead function can be expressed as:

$$\begin{aligned} T_0 &= P.T_P - T_S \\ &= P\Big(\Theta\Big(\frac{N}{P} + \frac{N}{P}\log\frac{N}{P} + T_{Comm} + \frac{N}{P} + T_{Sync}\Big)\Big) - \Theta(P) \\ &= \Theta\Big(P + P.T_{Comm} + P.T_{Sync}\Big) - \Theta(P) \\ &= \Theta\Big(P.T_{Comm} + P.T_{Sync}\Big) \end{aligned} \tag{7}$$

and the isoefficiency function for MapReduce applications under such conditions can be defined as

$$\begin{aligned} f(P) &= KT_0(W, p) \\ &= \Theta(P.T_{Comm} + P.T_{Sync}). \end{aligned}$$

Hence, the overhead T_0 for SMC programs with both map and reduce stages is highly dependent on both the communication and synchronization costs. In this scenario, the application scales only if T_0 can be maintained constant as P increases. Otherwise the overhead will increase above $\Theta(P)$ and the isoefficiency function should grow fast enough to prevent the ratio T_0/W from increasing as P increases. In other words, the overhead T_0 should not grow faster than P, otherwise the application will not scale.

□

Although the cost for disk operation is mentioned in this model, it will be ignored from this point on in our analysis, following the discussion presented in subsection 4.1. Hence, when no disk access is required in the second super step, parallel runtime is:

$$T_P = T_{Map} + T_{Sort} + T_{Comm} + T_{Sync} + T_{Merge} + T_{Reduce} + T_{Sync}. \tag{8}$$

The above cost expression will be used in our further analysis.

5.1. Bounds for reduceless SMC

Some MapReduce applications do not have reduce tasks, so that map outputs are written directly to disk (one file per map task) [58]. Such applications exist and they are one possible variation from the "canonical" MapReduce processing flow, for embarrassingly parallel problems, such as independently processing a large number of images of a large text collection. Such a pattern may be very common.

Theorem 5.2. *The lower bound on the isoefficiency function for a reduceless SMC program to process N data items running on a platform with P processors is $\Omega(P)$.*

Proof
Consider a SMC program that does not have reduce tasks, and whose map output is written directly to disk (one file per map task). In this case, the overall cost expressed by equation 8 can be reduced to $T_P = T_{Map} + T_{Sort}$. The problem size N must be significantly greater than P (i.e., $N \gg P$), otherwise eventually there will be idle processors as P increases. Thus, we assume $N = \Theta(P)$.

Let T_S be the runtime of the best-known sequential algorithm, which is a function of N the number of input data items to be processed. As each input data item requires a constant number of computing operations (remember that the map function is stateless and only depends on one input data item), we have $T_S = \Theta(P)$. Thus, substituting these cost values in Equation 2, the overhead function can be expressed as:

$$\begin{aligned}
T_0 &= P.T_P - T_S \\
&= P\Big(\Theta\Big(\frac{N}{P} + \frac{N}{P}\log\frac{N}{P}\Big)\Big) - \Theta(P) \\
&= \Theta\Big(N + N\log\frac{N}{P}\Big) - \Theta(N) \\
&= \Theta(P + P - P) \\
&= \Theta(P).
\end{aligned}$$

And the isoefficiency function for MapReduce applications under such conditions can be defined as

$$\begin{aligned}
f(P) &= KT_0(W, p) \\
&= K.P \\
&= \Theta(P)
\end{aligned}$$

Therefore, the problem size W for reduceless SMC programs should be increased as $\Omega(P)$ for two reasons. First, we need to maintain the ratio T_0/W as a non-increasing value when P increases. Second, notice that the workload W must grow as $\Omega(P)$ in order to keep the overhead T_0 constant, otherwise there will be idle processors when P increases. Under such circumstances MapReduce applications are highly scalable. In other words, linear speedups can be achieved. □

6. THE INFLUENCE OF COMMUNICATION AND SYNCHRONIZATION COSTS

In the previous section, two general lower bounds for MapReduce computations implemented as BSP were demonstrated, without any assumption regarding the underlying network architecture where the application executes. In contrast, in this section, a similar analysis wil be performed, but assuming an underlying packet switched network where devices are organized in some form of tree topology (e.g. fat-tree), that is frequently deployed in typical datacenters [7, 8, 37].

For a more detailed analysis on communication costs mentioned in Section 5 (in the proof of Lemma 5.1), we highlight the following points:

i For the isoefficiency analysis we assume that the number of data items N must be significantly larger than P (i.e., $N \gg P$). However, it suffices to assume that N is $\Theta(P)$ and thus the total amount of output data sent by each individual processor does not increase when P increases. Thus, the amount of data sent by each processor does not grow as P increases.

ii Remember that every processor sends $O(P)$ messages of size $O({}^{N}\!/_{P^2})$. As a consequence, the size of output messages sent by each processor decreases fast as P increases. The rate at which messages sizes decrease depend on the assumption concerning the growth rate of the input data (N). If we assume that the problem size (N) is fixed (as for most day-to-day situation), then message size (${}^{N}\!/_{P^2}$) will tend to $O(1/P^2)$. In the second scenario, we assume that N is $\Theta(P)$ (as for the isoefficiency analysis) and thus the total amount of map output data sent by each processor (${}^{N}\!/_{P}$) remains constant as P increases. In this scenario, map output messages size decreases proportional to $1/P$. In both scenarios there will be consequences on the communication cost for large values of P, and the consequence is that the size of individual messages will decrease, eventually becoming smaller than the maximum packet payload size (MPPS). As current networking technologies generate packets with fixed size headers, the reduction in the payload size may impact the communication cost.

iii The total number of output messages sent by all processors at the end of map phase grows as $\Theta(P^2)$. Remember that for every new processor added to the platform, $\Theta(P)$ new messages must be sent to the other processors (in all-to-all pattern). Since the total amount of data transmitted during the intermediate phase can rise at rate $\Theta(P)$, the network can eventually become congested.

iv The total amount of data items sent during the intermediate phase over the network increases as $O(P)$. For each new processor added to the system, $O(N/P)$ data items are added to the application workload (since $N \gg P$ should be maintained).

Hence, in this section we analyze the impact of the above points on the performance of MapReduce computations, and its scalability.

6.1. The Communication Cost

In [36], a simple architecture-independent model is proposed for estimating the communication cost for parallel computers. In this model, the cost for transferring a m-word message is given by

$$T_{Comm} = t_s + t_w \cdot m, \tag{9}$$

where t_s is the *startup time*, that is the time required to handle a message at the sending and receiving nodes (adding header, trailer, and error correction information); t_w is the *per word transfer time*, given by $1/b$, where b is the channel bandwidth expressed in words per second. It is possible to design algorithms with this simple cost model and port it to any parallel computer, including those whose networking topology implements a hypercube, mesh or a fat tree (which is typical for clusters and data centers where MapReduce applications are executed).

Typically, MapReduce applications execute in datacenter composed of a set of machines interconnected by high throughput switches organized in a tree topology (e.g., fat-tree) [37]. In such a scenario, communication latency can be considered as proportional to a constant number of hops, since the tree height can be kept constant for many hundreds of thousand machines. However, the time for each processor to send its output data to other processors at the end of map phase should be studied in more details.

First, remember that data is transmitted in packets composed by one fixed size header, and one variable size payload. The time to transmit packets over the network is influenced by several delay types, each type impacting the transmission time with different intensity [54]. For instance, *queuing delay* (d_{queue}) may be negligible if the network is uncongested, while it may raise to hundreds of milliseconds or even seconds depending upon the number of intermediate nodes (e.g. switches, routers) and on their utilization (or congestion) level.

The *processing delay* (d_{proc}) is the time taken by networking devices (e.g., a switch or router) to examine the packet's header and determine where to direct the packet, plus the time to check for bit-level errors in the packet that occurred in transmitting the packet's bits from the upstream node to router. Processing delays in high-speed networking devices are typically in the order of microseconds or less for each packet. Thus, it is not significant for our analysis.

The *propagation delay* (d_{prop}) is given by d/s, where d is the length of the physical link, and s is the propagation speed of the signal on the physical medium, which is within the range $2 \cdot 10^8$ meters/sec to $3 \cdot 10^8$ meters/sec. For a typical computer cluster composed of machines within the same datacenter (i.e., a few meters), (d_{prop}) is smaller than one microsecond. Thus, we assume that (d_{prop}) is negligible.

Finally, we consider the *transmission delay* (d_{trans}), which is composed of a few components that deserve more attention in this scenario. Consider that a packet transmission can be divided in two main steps. First, the packet should be assembled within the NIC memory (buffer), before its transmission. This step includes the assembly of a number of bits that compose all the fixed size fields of a packet (e.g., header, trailer), as well as the assembly of a variable size payload. The second step is the effective data transmission, i.e., the coding of all bits that compose the packet as signals that are transmitted by the physical medium at some specific rate (i.e., at 1Gbps, 10 Gbps). Thus, we can model T_{Comm} as

$$T_{Comm} = num\_packets * (H + D) + \frac{num\_packets * packet\_size}{b}, \tag{10}$$

where H is the time to assemble the packet header in the NIC memory buffer, D is the time to assemble the packet payload in the NIC memory buffer, and b is network bandwidth. In fact, packets are transmitted by the NIC in a first come first served policy, with some overlapping between the packet assembly and packet transmission steps. Thus, T_{Comm} grows asymptotically as $\mathcal{O}(Max\{num\_packets * (H + D), {}^{(num\_packets*packet\_size)}/_{b}\})$.

When the network operates at rate of Gigabits per second and large data volumes should be transmitted, the impact of both packet assembly and packet transmission can become significant to d_{proc}. The analysis of the former component ($num\_packets * (H + D)$) may depend on many variables. For instance, it depends on the speed that processors can assemble the bits that compose packet headers and packet payloads. Also, it depends on the speed as the network I/O system can copy packet bits into the NIC memory, which depends on how the software layers (e.g. device drivers, the operating system, hypervisors) proceed to make this copy.

Before stepping into the different communication scenarios of interest, we need to estimate the communication time of the shuffle phase T_{Comm}. First, remember that every processor receives $\Theta({}^{N}/_{P})$ input data items in the map phase. Assuming that the size of its output is proportional to the input size, we may consider the output data size also proportional to $\Theta({}^{N}/_{P})$. Assuming that the output data of a SMC is balanced among all the P processors, then each processor will transmit P messages of size $\Theta({}^{N}/_{P^2})$.

When the size of the messages (that is $\mathcal{O}({}^{N}/_{P^2})$) is greater than the maximum packet payload size of the network MPPS (i.e., when the messages are large enough to fill at least one or more packets), the total number of packets transmitted by each processor is $P \cdot (\lceil {}^{N/P^2}/_{MPPS} \rceil)$, and T_{Comm} can be estimated by

$$T_{Comm} = \Theta\Big(Max\{P \cdot \Big(\lceil \frac{\frac{N}{P^2}}{MPPS} \rceil \cdot (H + D)\Big), \frac{N}{P} \cdot \frac{1}{b}\}\Big). \tag{11}$$

In summary, Equation 11 states that the communication cost T_{Comm} will be dominated by the maximum between the cost for packet assembly, and the cost for packets transmission. See [61] for further details on current state of the technology for high throughput packets transmission and these costs. In the next section we will analyze in depth some specific circumstances regarding communication cost.

6.2. Estimating the Synchronization Time (T_{Sync})

The synchronization time T_{Sync} is influenced by the communication latency of the platform and by the barrier algorithm. Latency depends on the topology diameter, i.e., the maximum number of hops between any pairs of nodes. We assume that machines are placed within the same physical datacenter facility, so that computers are typically interconnected by network switches and routers organized in some form of tree topology. This operation requires all-to-all communication which can be optimistically implemented in time $O(\log P)$ by organizing processors in a tree or hypercube topology [36].

It is worth noticing that the focus of our analysis is the MapReduce model, instead of any of its implementation frameworks. In the MapReduce model, the reduce computation cannot start until the map tasks emit all of their output data. Under such conditions we consider that the synchronization cost T_{Sync} is $O(\log P)$. However, MapReduce implementations such as Hadoop implement optimizations between the map and reduce phase that can significantly mitigate this synchronization cost. These optimizations will be addressed in future work.

6.3. Scalability Bounds for MapReduce Applications when the network is not congested

In this section we define some lower bounds on the isoefficiency function for MapReduce applications under some specific scenarios. We assume that data messages are broken into packets and transmitted within a datacenter network.

Theorem 6.1. *The lower bound on the isoefficiency function for a SMC to process N data items running on a platform with P processors is $\mathcal{O}(P \cdot \log P)$ when the network is not congested and the size of messages in the intermediate phase is larger than the maximum packet payload size of the network (MPPS).*

Proof

Consider a SMC program with both map and reduce tasks running on P homogeneous and dedicated processors and whose overall computational cost may be expressed as in equation 8. Under such circumstances, problem size N must grow at least as $\Theta(P)$, otherwise eventually there will be idle processors as P increases. It follows that $T_{Map} = \Theta(\frac{N}{P}) = \Theta(1)$. In other words, under ideal conditions the execution time and output size of the map tasks running on each processor can be maintained constant as more processors are added to the platform. Assuming that all processors execute reduce tasks and SMC has perfect load balancing during the map phase and during the reduce phase, then T_{Sort}, T_{Merge}, and T_{Reduce} are all $\Theta(\frac{N}{P}) = \Theta(\frac{c \cdot P}{P}) = \Theta(1)$, i.e. such costs can be maintained constant when P increases.

Because the size ($^{N}/_{P^2}$) of individual output messages sent by the processors is greater than the maximum packet size ($MPPS$), and the total output size of each processor ($^{N}/_{P}$) remains constant as P increases, then the first component of Equation 12 (i.e., $P \cdot \Big(\lceil \frac{\frac{N}{P^2}}{MPPS} \rceil \cdot (H + D)\Big)$) will reduce to a constant value. In fact, since both the output size ($\frac{N}{P}$) and network bandwidth (b) are constant, the second term will also reduce to constant. Thus, we can estimate T_{Comm} as:

$$T_{Comm} = \Theta\Big(Max\{P \cdot \Big(\lceil \frac{\frac{N}{P^2}}{MPPS} \rceil \cdot (H + D)\Big), \frac{N}{P} \cdot \frac{1}{b}\}\Big) \tag{12}$$

$$= \Theta\Big(Max\{c_1, c_2 \cdot \frac{1}{b}\}\Big) \tag{13}$$

$$= \Theta(1). \tag{14}$$

Then, substituting T_{Sync} (estimated in Section 6.2) and T_{Comm} (Equation 12) in Equation 8, we have $T_P = \Theta(\log P)$ and the overhead can be estimated as follows:

$$\begin{aligned} T_0 &= P.T_P - T_S \\ &= P \cdot (\Theta(\log P)) - \Theta(P) \\ &= \Theta(P \cdot \log P) - \Theta(P) \\ &= \Theta(P \cdot \log P). \end{aligned} \tag{15}$$

Hence, even when the network in not congested, the SMC application is perfectly balanced, and the output messages from the map phase are greater than the maximum packet size ($MPPS$), the isoefficiency function $f(P)$ is $\Omega(P \cdot \log P)$. In other words, even in the best possible scenario the workload should grow above the number of processors to pay the overhead and to maintain the ratio T_0/W at some fixed value. In this scenario, MapReduce applications present poor scalability dominated by the synchronization cost. It is worth noticing that this result is valid for synchronous MapReduce applications. □

6.4. *Cost-optimality of MapReduce programs*

A parallel system can be considered *cost optimal* if its cost for solving a problem has the same asymptotic growth (in terms of Θ) as a function of the problem size as the fastest-known sequential algorithm on a single processing element [53]. In other words, a parallel system can be considered cost-optimal if and only if

$$pT_P = \Theta(W). \tag{16}$$

Then, substituting T_P for the right-hand side of Equation 3, we get the following

$$\begin{aligned} W + T_0(W, p) &= \Theta(W) \\ T_0(W, p) &= O(W). \end{aligned} \tag{17}$$

Equation 17 suggests that MapReduce applications can be cost-optimal if and only if its overhead cost does not exceed the problem size. This can be accomplished only if the communication and synchronization costs (which are the most expensive costs that compose the overhead function in Equation 7) is $\Theta(1)$. Under these conditions, both Equations 16 and 17 will be satisfied, and MapRecuce applications can be cost-optimal. As we discuss in further sections, MapReduce implementations (such as Hadoop) are very successful in mitigating both the communication and synchronization costs as we discuss further.

6.5. *Scalability Bounds for MapReduce Applications when messages are fragmented in small packets*

As mentioned in the beginning of this section, the messages sent by the map tasks tend to decrease when the number of processors P increases. There are two scenarios of interest to be discussed. In the first scenario we consider that workload grows as $\Theta(P)$ (as for the isoefficiency analysis). In this case the size of output messages decrease proportional to $O(1/P)$. The second scenario is even worse. When we consider that workload size N is fixed, the messages size decrease proportional to $O(1/P^2)$. In both scenarios, the map output messages will eventually become smaller than the maximum packet payload size (MSST) as P increases.

Theorem 6.2. *The lower bound on the isoefficiency function for a SMC to process N data items running on a platform with P processors is $\mathcal{O}(P^3)$ when the network is not congested and the size of messages in the intermediate phase is smaller than the maximum packet payload size of the network (MPPS).*

Proof
Remember that the total amount of output data sent by each processor is fixed $\Theta(\frac{N}{P})$. As the number of processors P increases, the size of the individual output messages will reduce, eventually becoming smaller than MPPS. Because SMC applications has perfect load balancing in the reduce phase, every processor executes at least one reduce task that needs to receive at least one message from every P processors. Under such conditions, the total number of messages (and packets) transmitted in the system will grow as $\Omega(P^2)$. Since every packet has a minimum fixed cost H, then we can estimate T_{Comm} as follows:

$$\begin{aligned} T_{Comm} &= O\Big(Max\{P \cdot \Big(P \cdot (H+D)\Big), \frac{N}{P} \cdot \frac{1}{b}\}\Big) && (18) \\ &= O\Big(Max\{P^2 \cdot H, c_2 \cdot \frac{1}{b}\}\Big) && (19) \\ &= O(P^2). && (20) \end{aligned}$$

As parallel runtime T_P is given by Equation 18, then by substituting the cost values in Equation 7 the overhead function can be expressed as:

$$\begin{aligned} T_0 &= P.T_P - T_S && (21) \\ &= P \cdot (\Theta(P^2) + \Theta \log P) - \Theta(P) \\ &= \Theta(P^3) + \Theta(P \cdot \log P) - \Theta(P) \\ &= \Theta(P^3). \end{aligned}$$

Therefore, when processors send map output messages smaller than the maximum packet size ($MPPS$), the isoefficiency function $f(P)$ is $O(P^3)$. In other words, MapReduce applications are not scalable under this circumstance.

□

6.6. Scalability Bounds when the Network Reaches Congestion

In the previous section, we assumed that the network is uncongested during the intermediate phase. However, when ratio at which output data is produced (by all processors) in the map phase exceeds the network bandwidth (b), then the communication time (T_{Comm}) can rise, loosing scalability.

Theorem 6.3. *The lower bound on the isoefficiency function for a SMC to process N data items running on a platform with P processors is $\mathcal{O}(P^2)$ when the network reaches congestion.*

Proof
Because the size (N/P^2) individual output messages sent by the processors is greater than the maximum packet size ($MPPS$), and the total output size of each processor (N/P) remains constant as P increases, the first component of Equation 22 (i.e., $P \cdot \left(\lceil \frac{\frac{N}{P^2}}{MPPS} \rceil \cdot (H+D)\right)$) can be reduced to a constant value. Until this point, the proof is similar to theorem 6.1. However, the analysis of the second component is not the same. The output size ($\frac{N}{P}$) remains constant. However, when the total amount of map output data transmitted by all the processors exceeds the transmission capacity of the network, then the effective bandwidth per processor will decrease. In the worst case, the effective bandwidth available for each processor may decrease as $\mathcal{O}(\frac{b}{P})$. Hence, we can estimate T_{Comm} as:

$$T_{Comm} = \mathcal{O}\Big(Max\{P \cdot \left(\lceil \frac{\frac{N}{P^2}}{MPPS} \rceil \cdot (H+D)\right), \frac{N}{P} \cdot \frac{1}{\frac{b}{P}}\}\Big) \tag{22}$$

$$= \mathcal{O}\Big(Max\{P \cdot c_1, c_2 \cdot \frac{1}{\frac{b}{P}}\}\Big) \tag{23}$$

$$= \mathcal{O}(P). \tag{24}$$

As mentioned in subsection 6.2, the synchronization time is negligible. Consequently, parallel execution time T_P can grow $\mathcal{O}(P)$ as the number of processors P increase, and we can, thus, estimate the isoefficiency function. Notice that the problem size N must be significantly greater than P (i.e., $N \gg P$), otherwise eventually there will be idle processors as P increases. As each input data item requires at least one computing operation, we have $T_S = \Omega(N)$. Remind that the amount of data items (N) need to grow at least as $\Theta(P)$, otherwise there will be idle processors as P increases and the efficiency cannot be maintained. Therefore, we can assume that $T_S = \Theta(P)$. Finally, as parallel runtime T_P is given by Equation 8, by substituting these cost values in Equation 2, the overhead function can be expressed as:

$$\begin{aligned} T_0 &= P.T_P - T_S \\ &= P \cdot (\mathcal{O}(P) + \mathcal{O}(\log P)) - \Theta(P) \\ &= \mathcal{O}(P^2) + \mathcal{O}(P \cdot \log P) - \Theta(P) \\ &= \mathcal{O}(P^2). \end{aligned}$$

Hence, when the map phase produces output data with ratio higher than the network capacity to transmit, isoefficiency function $f(P)$ is $\mathcal{O}(P^2)$. In other words, under such circumstances, the amount of work needed in order to pay the overhead and to maintain the ratio T_0/W at some fixed value may increase at rates at most $\mathcal{O}(P)$. MapReduce applications are not scalable under such conditions. □

7. EXPERIMENTAL RESULTS

In previous sections we demonstrated some lower bounds on the isoefficiency of MapReduce applications under specific circumstances and that communication and synchronization prevent application that implement both the map and reduce phases from scaling linearly. However, MapReduce implementation such as Hadoop are prodigious in mitigating such costs by overlapping computation and communication tasks, by compressing data sent over the network, and other optimizations. In this section we focus on the scalability analysis of MapReduce applications using an experimental approach that better captures such optimizations with better accuracy and realism.

The first experiment was conducted on a local 32 node cluster. Each node has 2 AMD Opteron 246 processors with 8 GB of memory, 4 250 GB SATA disks, and 1Gbps network interfaces. The network is implemented by Gigabit Ethernet switches. The servers run Linux CentOS and Hadoop 1.0.4. We executed experiments with the Terrier information retrieval system (http://terrier.org/) for calculating inverted indexes. We chose inverted indexing application because: (i) it is relevant (e.g. to implement Web search engines); (ii) requires high scalability when operating at Web scale; (iii) Terrier is a mature and open source information retrieval system that implements efficient inverted indexing algorithms whose performance has been tested and reported [62].

A subset of the ClueWeb 09 corpus has been chosen as the dataset for indexing. ClueWeb09_English_1, a set of 50 million documents written in English which is an integral part of the corpus ClueWeb09 [¶]. This subset was used in several tracks of the conference TREC 2009 [14], being known as ClueWeb09 'B'. The set ClueWeb09 complete, composed of a billion of Web pages in 10 languages, was collected by a group from Carnegie Mellon University in early 2009.

The application was executed on the cluster with up to 32 nodes, and results are shown in Table I. As we can observe, with 32 nodes, the map phase achieved speedup of 26.62, the reduce phase achieved speedup of 18.26, and the general makespan achieved speedup of 16.02. This result, including the sublinear speedup, is consistent to others experimental results found in the literature, such as in [62]. For this experiment, the application was composed of 192 map tasks that received input files around 178 MB compressed in zip format. Each file is read and inflated in memory resulting in near 1 GB of input data per task. Compressing the input files is a good practice employed in large scale deployments to improve performance. Map output data was also compressed for transmission, resulting in chunks of 12.7 MB on average (i.e., the ratio between the output data size and input data size is 0.01183) [62]. Compression of reduce output data can reduce costs for writing output to disk. There are several optimizations implemented by Hadoop, such as overlapping between computation and communication. The original MapReduce execution model states that the reduce tasks cannot execute until the last output key-value pair is emitted by the map tasks. However, it is possible to transmit key-value pairs to the nodes that will execute the reduce tasks in advance, as soon as each map task complete its execution, thus overlapping the map phase with the transmission of output data. Such implementation details impact the application scalability.

Empirical methods will hardly provide amounts of computational resources for the study of scalability limits. To circumvent this limitation, we combine simulation and empirical methods to evaluate the scalability of MapReduce applications on a meaningful number of machines. We

[¶]See http://boston.lti.cs.cmu.edu/Data/clueweb09/.

Table I. Execution times of the map phase, reduce phase, and total makespan on a cluster with 1 to 32 nodes.The values represent the average makespan for ten runs.

# of nodes	Map phase (s)	Reduce phase (s)	Total makespan
01	17,305	18,173	19,088
04	4,582	4,739	5,188
08	2,398	2,644	2,850
12	1,626	1,881	2,088
16	1,170	1,540	1,744
20	953	1,327	1,535
24	790	1,173	1,364
28	766	1,013	1,218
32	650	995	1,192

Table II. Accuracy of the Execution times of the map phase, reduce phase, and total makespan on a cluster with 1 to 32 nodes.

# of nodes	Makespan for real execution (s)	Makespan simulated (s)	Error (%)
01	19,088	19,248	0.83
04	5,188	5,127	0.17
08	2,850	2,871	0.73
12	2,088	2,095	0.33
16	1,744	1,739	0.28
20	1,535	1,528	0.45
24	1,364	1,373	0.07
28	1,218	1,207	0.91
32	1,192	1,188	0.33

used MRSG (MapReduce over SimGrid)[52]. As the name suggests, MRSG is a framework that simulates MapReduce applications on top of SimGrid, a validated and widely used tool for the accurate simulation of large scale distributed systems [10]. Before running experiments, it is necessary to calibrate the simulator to reproduce the behavior of the real system. The calibration process comprises the setup of the architecture resources (number and capacity of machines, network links and topology) and application parameters, such as the number and amount of work of map and reduce tasks, the size of input data files, the amount of output data (produced by the map tasks) sent over the network, among others. The calibration process is iterative and finishes when the simulator can mimic the real system with acceptable accuracy. Table II shows the accuracy achieved in the calibration process.

After the calibration, we simulated the inverted indexing of 100 TB of input data (i.e., 100,000 map tasks) increasing the cluster size up to 10,000 machines. The execution times are shown in Table II and in Figure 4. As depicted in Figure 4, the map phase achieved near to linear speedup, while the reduce phase could not achieve speedups above 300. The bottleneck in the reduce stage lead to poor scalability of the overall application.

The bottleneck is due to the lack of parallelism in the reduce phase. The application simulated on top of MRSG implements only 26 reduce tasks (for terms starting with the 26 letters of the alphabet). This bottleneck could easily be removed by increasing the number of reduces to 676, simulating a partitioner that looks to the first two characters of each term to decide which reduce task it should be sent to. The result is illustrated in Figure 5, and a new bottleneck limited the speedup around 600.

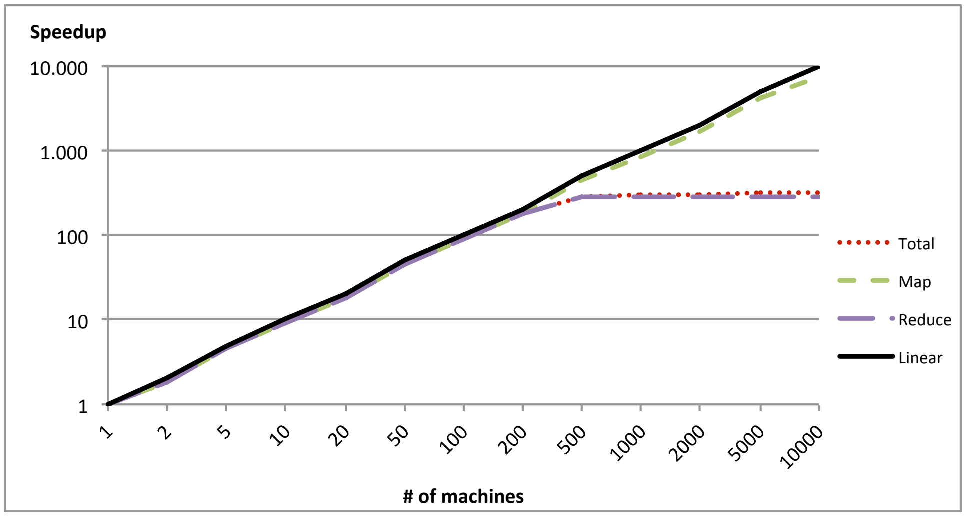

Figure 4. Simulation Results

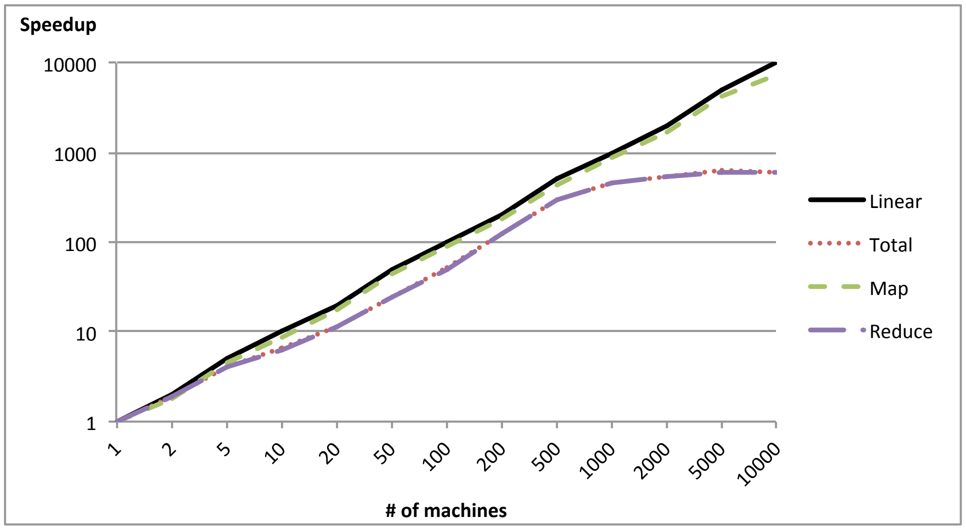

Figure 5. Speedup for the application with increased parallelism in the reduce phase.

Another bottleneck is not related to the degree of parallelism in the reduce phase but it is due to network congestion. As the number of reduces increased, the capacity to process data in the reduce phase was increased. However, the network did not provide enough performance to deliver the data transmitted in the intermediate phase. In the next experiment we changed the Ethernet switch for an Infiniband switch with bandwidth of 300 Gbps, and network latency of 100 nanoseconds. The result is depicted in Figure 6.

In summaryn in this section we have shown how simulation tools can be used for evaluating scalability of MapReduce applications in more realistic scenarios. First, we experimented with a real and well optimized application. To circumvent our limitation of physical resources, we calibrated a simulator to carry out experiments with thousands of machines with reasonable confidence about its accuracy. Simulation experiments confirmed that (i) under specific circumstances, MapReduce applications can achieve linear, or near-to-linear speedups, even for thousands of machines; (ii) even

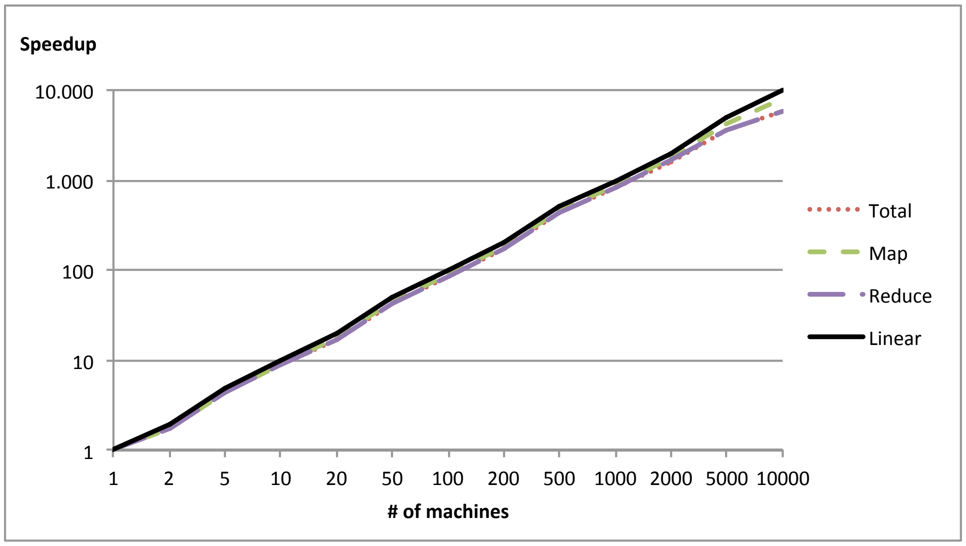

Figure 6. Speedup results for the application executing on a cluster with Infiniband.

for very optimized applications, network congestion is a major drawback for large scale MapReduce computations, and (iii) although theoretical results show that synchronization cost can dominate the execution of the application, Hadoop implements optimizations that significantly mitigate these costs.

8. COMMENTS ON THEORETICAL AND EXPERIMENTAL RESULTS

The BSP cost analysis presented in Sections 5 and 6 demonstrated that, in the best case, the lower bound on the isoefficiency function for MapReduce computations is $\Omega(P \log P)$. Those results have been obtained under the assumptions discussed in Section 4 implying that, under such conditions, speedup stays far below linear. On the other hand, the experiments presented in the previous section suggest it is possible to achieve speedups near to linear. This apparent discrepancy can be explained as follows.

BSP was adopted to model MapReduce computations because it is regarded as an abstract model for the hardware and software of parallel computing and it is both architecture-independent and scalable. It offers great simplicity, power, and portability for modeling a broad variety of parallel computations executing on shared-nothing multiprocessor hardware. Furthermore, BSP prescribes that the next superstep only commences after completion of the barrier synchronization in the end of the previous superstep. Messages sent in the previous superstep are available at the destination only at the start of the next superstep. Such a synchronous behavior allows BSP to mimic MapReduce computations as originally defined in [17], when reduce tasks cannot execute until the last output key-value pair is emitted by the map tasks.

However, MapReduce implementation frameworks may deploy many optimization strategies for performance improvement. For instance, some strategies anticipate the transmission of output data emitted by the map tasks (as presented in [83], pg. 208-2011, and in [27, 15]). Those strategies aim to mitigate performance losses due to the transmission of large data volumes over the network.

They can also anticipate reduce tasks readiness, thus allowing some overlapping between the map and reduce stages and reducing the barrier cost between these two stages.

Executing mappers and reducers together during two consecutive synchronizations of processors (BSP superstep) is intended to increase slackness, to improve load balancing, and/or to increase processor utilization. In such a situation, the BSP cost model used is still capable of supporting the conclusions of the paper. We observe two cases: (1) If during a given superstep in each processor i one mapper M_i and one reducer R_i are executed so that one is executed after the other, then in asymptotic terms this is equivalent to perform two supersteps, one for executing M_i and the another for executing R_i. Such a case becomes equivalent to the case studied in the paper. Namely, the total asymptotic cost remains the same since it is $O(\max M_i + \max R_i)$ with maximum taken considering all processors i. (2) If the execution of M_i and R_i are in some way overlapped, then the cost of the superstep is modeled as $\max\{M_i, R_i\}$ which does not affect the asymptotic trend for the scalability predicted in the paper because in this case the most costly asymptotic trend will still dominate the total cost $O(\max(\max\{M_i, R_i\}))$ for all processors i.

Finally, the asymptotic results presented in Sections 5 and 6 could be eventually refined with some hidden constant factors to provide more accurate estimates for the involved costs. On the theoretical side, our study provides important estimates on the computational cost of oblivious MapReduce computations, when most information about the particular application are unknown. On the other hand, the simulation approach presented in Section 7 is useful for evaluating several realistic *what-if* scenarios when most information about the application and platform is known.

9. CONCLUSION

MapReduce is a parallel computing framework proposed to execute computations that process large amounts of data on thousands of machines. The framework, and its implementations (e.g. Hadoop) support several techniques and optimizations that support the execution of highly scalable applications. In this paper we formally demonstrated lower bounds on the isoefficiency function for single round MapReduce applications, whenever these applications can be modelled as BSP jobs. BSP was chosen as modelling paradigm because it is a well established model for parallel computations. In this work, we also showed and proved how communication and synchronization costs can be dominant for MapReduce computations. To our knowledge, this is the first study that demonstrates scalability bounds for MapReduce applications. These results are useful for further asymptotic analysis of MapReduce application executions.

On the other hand, MapReduce implementations such as Hadoop can mitigate such costs to approach linear, or near-to-linear speedups. In some scenarios, the modeling of MapReduce applications as BSP jobs overestimate synchronization costs, and the analysis proposed in this paper does not hold. We also pointed out some conditions which can prevent MapReduce applications from scaling. The scalability analysis of MapReduce applications under those specific circumstances will be the subject of future work.

ACKNOWLEDGMENT

All the authors thank CAPES and STIC-AmSud project 13-STIC-11 (RESPOND). Mauricio Marín thanks to Basal funds FB0001, Conicyt, Chile; and Hermes Senger also thanks CNPQ, and FAPESP for their support.

REFERENCES

1. Hama:. http://incubator.apache.org/hama/.
2. Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, 2009.
3. Foto N Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D Ullman. Vision paper: Towards an understanding of the limits of map-reduce computation. *arXiv preprint arXiv:1204.1754*, 2012.
4. Leonidas Akritidis and Panayiotis Bozanis. Computing scientometrics in large-scale academic search engines with mapreduce. In *Proceedings of the 13th International Conference on Web Information Systems Engineering*, pages 609–623, 2012.
5. Apache. In *Hadoop website*. Avalable at: `http://hadoop.apache.org/`, 2010.
6. Ana Paula Appel1 and Estevam Rafael Hruschka Junior. Centaurs - a component based framework to mine large graphs. In *Proceedings of the Brazilian Symposium on Databases - SBBD*, Belo Horizonte, MG, Brazil, 2010. Brazilian Computer Society.
7. L. A. Barroso and U. Hoelzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan and Claypool Publishers, 2009.
8. Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.
9. Michael J. Cafarella and Christopher Ré. Manimal: Relational optimization for data-intensive programs. In *Procceedings of the 13th International Workshop on the Web and Databases*, pages 10:1–10:6, 2010.
10. Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014.
11. Ronnie Chaiken, Bob Jenkins, Perake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.
12. Flavio Chierichetti, Ravi Kumar, and Andrew Tomkins. Max-cover in map-reduce. In *Proceedings of the 19th International Conference on World Wide Web*, pages 231–240, 2010.
13. Lynda Chin, Jannik N Andersen, and P Andrew Futreal. Cancer genomics: from discovery science to personalized medicine. *Nature medicine*, 17(3):297–303, 2011.
14. Charles L Clarke, Nick Craswell, and Ian Soboroff. Overview of the trec 2009 web track. Technical report, DTIC Document, 2009.
15. Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. Online aggregation and continuous query support in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1115–1118. ACM, 2010.
16. Fabricio AB da Silva and Hermes Senger. Scalability limits of bag-of-tasks applications running on hierarchical platforms. *Journal of Parallel and Distributed Computing*, 71(6):788–801, 2011.
17. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplifed data processing on large clusters. In *6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–13. USENIX, 2004.
18. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.
19. Elif Dede, Madhusudhan Govindaraju, Daniel Gunter, Richard Shane Canon, and Lavanya Ramakrishnan. Performance evaluation of a mongodb and hadoop platform for scientific data analysis. In *Proceedings of the 4th ACM Workshop on Scientific Cloud Computing*, pages 13–20, 2013.
20. Frank Dehne, Andreas Fabri, and Andrew Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, pages 298–307, 1993.

21. Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 3(1-2):515–529, 2010.
22. Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. Only aggressive elephants are fast elephants. *Proc. VLDB Endow.*, 5(11):1591–1602, July 2012.
23. Christos Doulkeridis and Kjetil Norvag. A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal*, 23(3):355–380, 2014.
24. EC2 Amazon. In *Amazon Elastic MapReduce*. Avalable at: `http://aws.amazon.com/elasticmapreduce/`, 2013.
25. Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.
26. Iman Elghandour and Ashraf Aboulnaga. Restore: Reusing results of mapreduce jobs in pig. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 701–704, 2012.
27. Marwa Elteir, Heshan Lin, and Wu-chun Feng. Enhancing mapreduce via asynchronous data processing. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 397–405. IEEE, 2010.
28. Matthias Englert, Heiko Röglin, and Berthold Vöcking. Worst case and probabilistic analysis of the 2-opt algorithm for the TSP. *Algorithmica*, 68(1):190–264, 2014.
29. Facebook. In *Facebook has the world's largest Hadoop cluster!* Avalable at: `http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html`, 2008.
30. L. Fegaras. Supporting bulk synchronous parallelism in map-reduce queries. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1068–1077, 2012.
31. Jon Feldman, S Muthukrishnan, Anastasios Sidiropoulos, Cliff Stein, and Zoya Svitkina. On distributing symmetric streaming computations. *ACM Transactions on Algorithms (TALG)*, 6(4):66, 2010.
32. Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978.
33. Apache Giraph. Giraph. http://incubator.apache.org/giraph/., 2015.
34. Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *Proceedings of the 22nd international conference on Algorithms and Computation*, pages 374–383. Springer-Verlag, 2011.
35. Google Appengine. In *Google App Engine API for running MapReduce jobs*. Avalable at: `http://code.google.com/p/appengine-mapreduce/`, 2013.
36. Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel computing*. Boston, MA: Addison-Wesley, 2003.
37. Albert Greenberg, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Towards a next generation data center architecture: scalability and commoditization. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 57–62. ACM, 2008.
38. Benjamin Gufler, Nikolaus Augsten, Angelika Reiser, and Alfons Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, pages 522–533, 2012.
39. Neil J. Gunther, Paul Puglia, and Kristofer Tomasette. Hadoop superlinear scalability. *Commun. ACM*, 58(4):46–55, 2015.
40. Steffen Heinz and Justin Zobel. Efficient single-pass index construction for text databases. *J. Am. Soc. Inf. Sci. Technol.*, 54(8):713–729, June 2003.
41. H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *VLDB Endowment*, 4, 2011.
42. Anthony JG Hey, Stewart Tansley, Kristin Michele Tolle, et al. The fourth paradigm: data-intensive scientific discovery. 2009.
43. IDC. In *The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East*. Avalable at: `http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf` 2012.
44. Adam Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, 2009.
45. Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic optimization for mapreduce programs. *Proc. VLDB Endow.*, 4(6):385–396, 2011.
46. Jeffrey Jestes, Ke Yi, and Feifei Li. Building wavelet histograms on large data in mapreduce. *Proc. VLDB Endow.*, 5(2):109–120, 2011.

47. Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: An in-depth study. *Proc. VLDB Endow.*, 3(1-2):472–483, 2010.
48. Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 938–948. Society for Industrial and Applied Mathematics, 2010.
49. Douglas B Kell and Stephen G Oliver. Here is the evidence, now what is the hypothesis? the complementary roles of inductive and hypothesis-driven science in the post-genomic era. *Bioessays*, 26(1):99–105, 2004.
50. Ioannis Kitsos, Kostas Magoutis, and Yannis Tzitzikas. Scalable entity-based summarization of web search results using mapreduce. *Distrib. Parallel Databases*, 32(3):405–446, 2014.
51. Lars Kolb, Andreas Thor, and Erhard Rahm. Load balancing for mapreduce-based entity resolution. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, pages 618–629, 2012.
52. Wagner Kolberg, Julio Anjos, Pedro de B Marcos, Alexandre KS Miyazaki, Claudio R Geyer, and Luciana B Arantes. Mrsg-a mapreduce simulator over simgrid. *Parallel Computing*, 2013.
53. Vipin Kumar and V Nageshwara Rao. Parallel depth first search. part ii. analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.
54. James Kurose and Keith Ross. Computer networks: A top down approach featuring the internet. *Peorson Addison Wesley*, 2013.
55. YongChul 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*, pages 25–36, 2012.
56. Nikolay Laptev, Kai Zeng, and Carlo Zaniolo. Early accurate results for advanced analytics on mapreduce. *Proc. VLDB Endow.*, 5(10):1028–1039, 2012.
57. Yanfang Le, Jiangchuan Liu, Funda Ergün, and Dan Wang. Online load balancing for mapreduce with skewed data input. In *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014*, pages 2004–2012, 2014.
58. Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.
59. Yuting Lin, Divyakant Agrawal, Chun Chen, Beng Chin Ooi, and Sai Wu. Llama: Leveraging columnar storage for scalable join processing in the mapreduce framework. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 961–972, 2011.
60. Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146, 2010.
61. Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). Seattle, WA: USENIX Association*, pages 459–473, 2014.
62. Richard McCreadie, Craig Macdonald, and Iadh Ounis. Mapreduce indexing strategies: Studying scalability and efficiency. *Information Processing & Management*, 48(5):873–888, 2012.
63. Richard M. C. McCreadie, Craig Macdonald, and Iadh Ounis. On single-pass indexing with mapreduce. In *Proceedings of the 32Nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 742–743, 2009.
64. Hisham Mohamed and Stéphane Marchand-Maillet. Mro-mpi: Mapreduce overlapping using mpi and an optimized data exchange policy. *Parallel Comput.*, 39(12):851–866, 2013.
65. Hisham Mohamed and Stéphane Marchand-Maillet. Distributed media indexing based on mpi and mapreduce. *Multimedia Tools Appl.*, 69(2):513–537, March 2014.
66. Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. Mrshare: Sharing across multiple queries in mapreduce. *Proc. VLDB Endow.*, 3(1-2):494–505, 2010.
67. Alper Okcan and Mirek Riedewald. Anti-combining for mapreduce. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 839–850, 2014.
68. I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and C. Lioma. Terrier: A High Performance and Scalable Information Retrieval Platform. In *Proceedings of ACM SIGIR'06 Workshop on Open Source Information Retrieval (OSIR 2006)*, 2006.
69. Aisling O'Driscoll, Jurate Daugelaite, and Roy D Sleator. 'big data', hadoop and cloud computing in genomics. *Journal of biomedical informatics*, 46(5):774–781, 2013.
70. Matthew Felice Pace. {BSP} vs mapreduce. *Procedia Computer Science*, 9(0):246 – 255, 2012. <ce:title>Proceedings of the International Conference on Computational Science, {ICCS} 2012.

71. Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. Planet: Massively parallel learning of tree ensembles with mapreduce. *Proc. VLDB Endow.*, 2(2):1426–1437, 2009.
72. Sherif Sakr, Anna Liu, and Ayman G. Fayoumi. The family of mapreduce and large-scale data processing systems. *ACM Comput. Surv.*, 46(1):11:1–11:44, 2013.
73. Rodrigo Schmidt and Fernando Pedone. Consistent main-memory database federations under deferred disk writes. In *Proceeding of the 24th IEEE Symposium on Reliable Distributed Systems*, 2005.
74. Hermes Senger and Fabrício Alves Barbosa da Silva. Bounds on the scalability of bag-of-tasks applications running on master-slave platforms. *Parallel Processing Letters*, 22(02), 2012.
75. Hermes Senger, Eduardo R Hruschka, Fabrício AB Silva, Liria M Sato, Calebe P Bianchini, and Marcelo D Esperidião. Inhambu: data mining using idle cycles in clusters of pcs. In *Network and Parallel Computing*, pages 213–220. Springer, 2004.
76. Sangwon Seo, E.J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 721–726, 2010.
77. Fabricio A. B. Silva and Hermes Senger. Improving scalability of bag-of-tasks applications running on master-slave platforms. *Parallel Computing*, 35(2):57 – 71, 2009.
78. Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, pages 607–614, 2011.
79. Alexander Tiskin. Bsp (bulk synchronous parallelism). In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 192–192. Springer US, 2011.
80. Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. Doulion: Counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 837–846, 2009.
81. Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
82. Jason Venner. *Build scalable, distributed applications in the cloud*. Apress, 2009.
83. Tom White. *Hadoop: the Definitive Guide. 3rd Edition*. O'Reily, 2012.
84. Xiaofei Zhang, Lei Chen, and Min Wang. Efficient multi-way theta-join processing using mapreduce. *Proc. VLDB Endow.*, 5(11):1184–1195, 2012.