PEDECIBA INFORMÁTICA

Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

TESIS DE MAESTRÍA EN INFORMÁTICA

Large Scale Optimization in Hadoop

Marcos Barreto
mbarreto@fing.edu.uy

January 2016

Academic Director:
Sergio Nesmachnow, Universidad de la República.

Large Scale Optimization in Hadoop
Barreto, Marcos
Tesis de Maestría en Informática
PEDECIBA
Instituto de Computación - Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay, Enero de 2016

Contents

1 Introduction

Optimization algorithms have long been discussed in the literature, and many are used as benchmarks to test different hardware infrastructures and/or algorithms performance. Nowadays, Large Scale Optimization (LSO) problems, i.e. problems involving very large search spaces and/or a considerable number of optimization variables, appear in many important real-world applications. Different areas of computer science and other disciplines have problems in which computational methods are applied, such as artificial intelligence, operations research, bioinformatics and electronic commerce. Prominent examples are tasks such as finding shortest or cheapest round trips in graphs, finding models of propositional formula or determining the 3D-structure of proteins.

For most combinatorial optimization problems, the space of potential solutions for a given problem instance is at least exponential in the size of that instance [53]. The efficiency of most state-of-the-art optimization algorithms deteriorates rapidly as the dimensionality of the search space increases, so different techniques must be used to have reasonable execution times.

Distributed computing comes to help researchers to face LSO problems by applying a cooperative approach that proposes splitting a big problem into many smaller (sub-)problems in order to speed up the search [49]. Parallel computing techniques, i.e. techniques used to implement an algorithm in a distributed environment, have long been used in the scientific community to implement distributed algorithms to solve LSO problems exploiting large clusters of heterogeneous computers [48, 89, 91, 63, 94, 9, 60, 46]. Nevertheless, parallel computing adds many difficulties that have to be taken into account when developing and executing a distributed application. Communication among different process of different computers in the network, synchronization of different process, failure tolerance and data replication are among the most common problems when implementing a distributed algorithm.

The last decade has been witness of the dramatical growth of the digital information generated and being able to analyze all these information is a powerful tool for any organization and also, a difficult task. Information is generated from vast sources: systems logs, web searches, online transactions, user actions like sending an image or a text message o when buying something online, etc. Currently, the information generated by each person keeps growing each day. Moreover, the advent of new technologies like mobile devices has contributed to the growth of data generated by users and therefore, any practical application must be able to scale up to datasets of interest. As a result, the community has paid a lot of interest in distributed computing for data analysis and other complex information processing problems [97]. Different frameworks have been created to analyze big amounts of information in a distributed environment.

Hadoop is the most known framework for analyzing big amounts of information in a distributed environment, providing fault tolerance, replication, communication between different process [92]. Hadoop has been designed to execute applications in clusters of commodity computers with little effort, by providing an infrastructure to run MapReduce jobs over a distributed file system [92].

MapReduce programming paradigm was designed to execute data intensive algorithms, and it has scarcely been applied for optimization algorithms [16]. MapReduce was created to easily implement algorithms to solve large scale problems in a distributed environment without having to address all the problems

of distributed algorithms. It comes to provide an easy to understand model for parallelizing an algorithm without the need learn parallel computing techniques, resulting in shorter developing times and easy to maintain distributed algorithms. MapReduce provides a powerful tool to anymore wanting to analyze huge volumes of information in a cluster of computers.

Solving a LSO problem with a sequential algorithm usually takes huge execution times and usually, the approach taken is to solve a small version of the problem to keep the algorithm in reasonable execution times. Learning parallel computing techniques and correctly implementing them is not an easy task and requires deep understanding in communication and synchronization of process in a distributed environment. Although Hadoop was designed to solve data intensive algorithms [92], it provides a promising environment and computing model to solve hard optimization problems by applying a cooperative approach that allows facing very large problem instances using distributed computing resources.

This work presents advances on applying MapReduce approach for designing a solver for LSO problems over Hadoop, as well as its benefits and limitations. MapReduce and Hadoop infrastructure could be used to easily solve a LSO problem in a distributed environment, without having to learn parallel computing techniques. The LSO problem used as an example in this work is the 3-SAT, a classical combinatorial problem used many times in the scientific community to test new algorithm approaches [43]. This work presents different algorithms to solve the 3-SAT on Hadoop and problems and limitations of implementing a LSO algorithm in Hadoop are presented. The main contribution of our research are: i) studying the advantages and challenges of developing LSO using MapReduce over Hadoop; ii) implementing three main MapReduce 3-SAT solver variants; iii) the experimental evaluation that shows that the collaborative approach is a promising option for LSO in the cloud and iv) Hadoop and MapReduce is indeed a promising solution for large scale optimization problems.

This work is structured as follows. An introduction to combinatorial problems and large scale optimization problems is presented in Section 2 as well as an introduction to the 3-SAT, its formulation, codification and importance. Different distributed computing techniques are introduced in Section 3, including a brief introduction to the MapReduce paradigm. Hadoop and its MapReduce execution environment is introduced in Section 4. Section 6 presents the devised algorithms to solve the 3-SAT using divide and conquer, cumulative learning techniques as well as a randomized approach. The the problems presented when solving a LSO problem in Hadoop using MapReduce paradigm and the experimental analysis is also reported. Finally, Section 7 presents the conclusions and formulates the main lines for future work.

2 Large Scale Optimization Problems

This section introduces optimization problems and its relevance. Different examples of large scale optimization problems are presented. Optimization problems are analyzed based on its computational complexity, which is also introduced in this section as well as the different classes of complexity available. Different search algorithms, which are used in the devised algorithms presented in this work, is presented in this section. Finally, the SAT problem is introduced, which is the optimization problem to be solved in this work and its codification, formulation and importance.

2.1 Combinatorial optimization problems

Combinatorial optimization problems are problems that consist in finding an optimal solution from a finite set of possible solutions [53]. Optimization problems are a family of problems inside combinatorial problems, where the solutions are additionally evaluated by an objective function and the objective is to find solutions with optimal objective functions values [53, 43]. Large Scale Optimization problems (LSO) are optimization problems involving very large search spaces and/or a considerable number of optimization variables. These kind of problems appear in many areas in which computational methods are used such as artificial intelligence, machine learning, bioinformatics and electronic commerce. Examples of Large Scale Optimization problems are finding the shortest path, graph coloring, determining the 3D-structure of proteins, scheduling tasks in a large cluster with priorities, resource allocation or hardware design and genome sequencing. These kind of problems typically involve searching for assignments given certain condition, grouping information, ordering or performing certain actions in an efficient manner given certain constraints.

Many combinatorial problems are decision problems where the solution for a given problem instance is specified by a set of logical conditions. The Propositional Satisfiability Problem (SAT), which is the problem solved in this work, is one of the classical combinatorial problems [17] which consists in finding a truth assignment value for a list of variables that make a set of clauses simultaneously true, each being a disjunctive set of literals. The SAT problem is further explained in section 2.4. Another classical combinatorial problem are the Graph Coloring Problems which, given a graph G and a number of colors, the problem consists in finding an assignment of colors to the vertices of the graph such that two vertices that are connected by an edge have never the same color.

A candidate solution satisfying the logical conditions are called feasible, valid or satisfiable. Decision problems can be extended or formulated as an optimization problem by defining an objective function. For example, an optimization problem for Graph Coloring Problems could have the objective function defined by the number of colors, minimizing the number of colors needed to assign colors to the vertices.

Solving combinatorial decision and optimization problems implies, given a problem instance, to search for solutions in the space of the problem instance. For this reason, these problems are also characterized as search problems [43]. Brute force algorithms are not suitable for solving combinatorial problems as, for a given problem instance, the search space is exponential in the size of the input and would take years to be solved. For example, a SAT instance with 100

variables has 2^{100} possible assignments and a brute force algorithm evaluates all possible assignments to the problem. If to evaluate 100 billon assignments takes 1 millisecond, then the execution time to evaluate the whole search space would be more than 4 billon years.

2.2 Computational Complexity

This section introduces the computational complexity classes, which categorize combinatorial and optimization problems based on their complexity. The complexity of an algorithm is defined in terms of the space and time requirements of computer power to solve such algorithm, for turing machines and other formal machine or programming models.

2.2.1 Complexity Classes

There are two complexity classes: P, the class of problems that can be solved by a deterministic machine in polynomial time, and NP, the class of problems that can be solved by a nondeterministic machine in polynomial time [33]. Nondeterministic machines are hypothetical machines which can be thought of as having the ability to make correct guesses for certain decisions. Every problem in P is also contained in NP, because deterministic calculations can be emulated on a nondeterministic machine.

The question whether $NP \subseteq P$, and consequently $P = NP$, is one of the most prominent open problems in computer science. Since many extremely application-relevant problems are in NP, but possibly not in P (i.e., no polynomial-time deterministic algorithm is known). For these computationally hard problems, the best algorithms known so far have exponential time complexity. Therefore, for growing problem size, the problem instances becomes quickly intractable and even improvements in state-of-the-art technology have little to no effect in keeping execution times in reasonable times.

Many hard problems from NP are closely related and can be translated into each other by an algorithm with polynomial deterministic time, these translations are called polynomial reductions [33]. A problem that is at least as hard as any other problem in *NP*, in the sense that can be polynomially reduced to it, is called *NP*-hard. Thus, *NP*-hard problems can be regarded as at least as hard as every problem in *NP*, but do not necessarily have to belong to the class *NP* as their complexity may actually be higher. *NP*-hard problems that are contained in *NP* are called *NP*-complete and are the hardest problems in *NP* [33].

Different combinatorial problems are *NP*-Hard. The SAT, as mentioned before, was the first algorithm to be proven to be *NP*-Hard [17]. Another example of a *NP*-Hard classical problem is the TSP [33]. The same holds for many special TSP cases, such as Euclidean TSPs and even TSPs in which all edge weights are either one or two. In all of these cases, the associated decision problem for optimal solution quality is *NP*-complete. However, there exist a number of polynomially solvable special cases of the TSP, such as fractal TSP instances that are generated by Lindenmayer Systems [65] or specially structured Euclidean instances where, for example, all vertices lie on a circle [10, 74]. Besides SAT and TSP, other well-known combinatorial problems are *NP*-hard or *NP*-complete, including the Graph Coloring Problem, the Knapsack Problem, as well as many scheduling and timetabling problems [33].

One fundamental result of complexity theory states that it suffices to find a polynomial time deterministic algorithm for one single NP-complete problem to prove that $NP = P$. This is a consequence of the fact that all NP-complete problems can be encoded into each other in polynomial time by a polynomial reduction. Today, most experts believe that $P \neq NP$. However, so far all efforts of finding a proof for this inequality have been unsuccessful, and there has been some speculation that the mathematical methods might be too weak to solve this fundamental problem.

Although many combinatorial problems are NP-hard, not every computational task that can be formulated as a combinatorial problem is difficult. A well-known example for a problem that seems to require searching an exponentially large space of candidate solutions is the Shortest Path Problem: given an edge-weighted graph G (where all edge weights are positive) and two vertices u,v in G, find the shortest route from u to v, that is, the path with minimal total edge weight. This problem is efficiently solved by Dijkstra's algorithm [26] which can find shortest paths in quadratic time. In many cases, efficient algorithms for solving combinatorial problems are based on a general method called dynamic programming [7].

Many practically relevant combinatorial problems, such as scheduling and planning problems, are NP-hard and therefore not exists an algorithm to solve them in polynomial execution time. However, being NP-complete or NP-hard does not mean that it is impossible for a problem to be solved efficiently. Practically, there are at least three ways of dealing with these problems:

- Find a special case of the problem with relaxed conditions or a subclass of the problem that can be solved efficiently.
- Use efficient approximation algorithms to find suboptimal solutions.
- Use stochastic approaches.

The first strategy means that, although a specific problem is of type *NP*-hard, it does not mean that the entire set of problem instances are of that type of complexity. Taking for example the satisfiability problem (SAT), which is *NP*-Hard, but there exists a polynomial time algorithm to solve the problem when having 2 literals by each clause [3]. The *NP*-Hardness results characterize the worst-case complexity of the problem and typical problem instances are easier to solve. For example, solving a problem instance of the SAT problem with many variables and clauses with only one solution is practically more difficult than to solve another problem instance which has hundreds of possible solutions. The same example applies to time complexity of concrete algorithms for combinatorial problems. A well-known example is the Simplex Algorithm for linear optimization, which has worst-case exponential time complexity, but has been empirically shown to achieve polynomial run-times in the average case [52].

In the case of a NP-hard optimization problem that cannot be narrowed down to an efficiently solvable subclass, one option is to accept suboptimal solutions. Formally, the degree of sub-optimality of a solution quality x is typically expressed in the form of the approximation ratio $r > 1$, defined by the Eq. 1, where $f : A \rightarrow S$ is the optimization function, $x*_{max}$ is the optimal solution for a maximization problem, $x*_{min}$ is the optimal solution for a minimization problem, for the given problem instance.

$$
\begin{aligned}
x*_{max} : f(x*) \geq f(x) \forall x \in S. \\
x*_{min} : f(x*) \leq f(x) \forall x \in S. \\
x \text{ is a suboptimal for } x *_{min} : \frac{x}{x*_{min}} > r > 1 \\
x \text{ is a suboptimal for } x *_{max} : \frac{x*_{max}}{x} > r > 1
\end{aligned} \tag{1}
$$

For a given optimization problem, an associated approximation problem can be defined. These problems, the objective is to find solutions with an approximation ratio bounded from above by a given constant r. Often, as r is increased, the computational complexity of these approximation problems decreases and they become practically solvable. In some cases, allowing a relatively small margin from the optimal solution quality renders the problem deterministically solvable in polynomial time. In other cases, the approximation problem remains *NP*-hard, while for practical problem instances, suboptimal solutions of acceptable quality can be found in reasonable time [43, 53]. Many examples of hard-to-solve problems are usually efficiently solved by accepting suboptimal solutions. Taking the TSP instance with arbitrary edge weights for example, there is no deterministic algorithm that is guaranteed to find solutions of quality within a constant factor of the optimum for a given problem instance in polynomial time [76]. Nevertheless, for instances satisfying the triangle inequality, the algorithm of Christofides guarantees a solution in a polynomial type with an approximation ratio of at most 1.5 [15]. Furthermore, in the case of Euclidean TSP instances, a polynomial time approximation scheme exists, where solutions can be found efficiently for arbitrary approximation ratios larger than one [2].

However, sometimes even reasonably efficient approximation methods cannot be devised. In these cases, one option is to use probabilistic rather than deterministic algorithms [43]. In many cases, probabilistic algorithms have been found to be considerably more efficient on *NP*-complete or *NP*-hard problems than the best deterministic alternative. In other cases, probabilistic methods and deterministic methods complement each other in the sense that for certain types of problem instances one or the other has been found to be superior [69, 43]. Moreover, many times deterministic and probabilistic approaches are both used to increase the efficiency of an algorithm.

2.3 Search Algorithms

The fundamental idea behind the search approach is to iteratively generate and evaluate possible solutions for a problem. In the case of combinatorial decision problems, the procedure means to decide whether it is an actual solution or not. Evaluating candidate solutions for a *NP*-Hard combinatorial problem is usually fast, in polynomial execution time. For example, evaluating a 3-SAT instance means to evaluate if a specific literal assignment makes all clauses in the formula true, which is fast to evaluate. Generally, the evaluation of candidate solutions depends on the given problem.

The fundamental differences between search algorithms are in the way in which candidate solutions are generated, which can have a very significant impact on the algorithm theoretical properties and practical performance. A search algorithm could be deterministic or stochastic, and constructive or perturbative. A deterministic search algorithm walks through the search space of a problem instance in a deterministic manner. This search guarantees that eventually either a (optimal) solution is found, or, if no solution exists, this fact is determined with certainty for a given search space. On the other hand, stochastic differentiate from deterministic search algorithms in that the decisions on first are performed randomly. In this case, if a stochastic search algorithm fails to find a solution, does not guarantee that the problem instance does not have a solution.

An example of a deterministic search algorithm compared to a stochastic approach is shown in Figure 1. The example presents the problem of finding a truth assignment for three literals x_1, x_2, x_3. The search space for this problem is all possible combinations of assignments for literals x_1, x_2, x_3. Both algorithms start at the initial state with the candidate solution $x_1, x_2, x_3 = \{0, 0, 0\}$ but the following steps are taken differently. Whereas the deterministic algorithm selects which literal to flip in each step in a specific manner, selecting x_3 first, then x_2 and then x_1. The stochastic approach randomly selects the literals to flip in each step. In the example shown, the stochastic algorithm selects x_1, then x_3 and in the third step selects x_1 again. Note that the stochastic algorithm repeats not only selected literals but also could repeat candidate solutions generated.

2.3.1 Constructive Search

The task of generating complete candidate solutions by iteratively extending partial candidate solutions is formulated as a search problem in which the goal is to obtain a *good* candidate solution. In the case of optimization problems, a good solution corresponds to the value of the objective function. Algorithms for solving this type of problem are called *constructive* search methods or *construction heuristics*.

An example of a constructive search algorithm for the 3-SAT with two literals is shown in Figure 2. The problem consists to find a truth assignment for literals x_1 and x_2. The constructive algorithm sets in each step a truth value for one literal extending the given partial candidate solution of previous step.

Constructive search algorithms can be used to generate candidate solutions for any combinatorial problem. For example, a constructive search algorithm could be used to generate candidate solutions for the TSP problem by, starting at a randomly chosen vertex in the graph, iteratively follow an edge with minimal weight connecting the current vertex to one of the vertices that has not

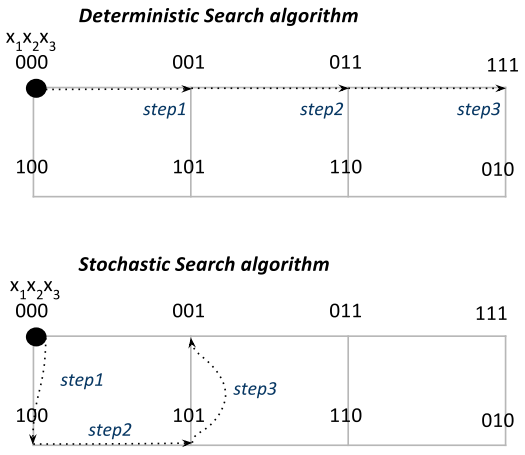

Figure 1: Search example comparing a stochastic approach vs a deterministic approach

step 1	$x_1=0$
step 2	$x_1=0, x_2=0$
step 3	$x_1=0, x_2=1$
step 4	$x_1=1, x_2=0$
step 5	$x_1=1, x_2=1$

Figure 2: Simple constructive search example for the 3-SAT.

yet been visited. This method generates a path that, by adding the starting vertex as a final element to the corresponding list, can be easily extended into a Hamiltonian cycle in the given graph. This simple construction heuristic for the TSP is called the Nearest Neighbor Heuristic. On its own, it typically does not generate candidate solutions with close-to-optimal objective function values but it is commonly and successfully used in combination with perturbative search methods.

2.3.2 Perturbative Search

Candidate solutions for instances of combinatorial problems are composed of solution components. Usually, candidate solutions can be changed into new candidate solutions by modifying one or more of the corresponding solution components. For example, in the SAT problem, each literal is a solution component and changing from one candidate solution to another can be to flip one literal truth value. This search approach is known as a *perturbative* search algorithm. For example, applying this technique to solve the SAT problem would

start with a complete truth assignment and then, in each step, generate other truth assignment changing the truth value of a number of variables in such assignment. An example of a simple perturbative search algorithm to solve a 3-SAT problem instance is shown in Figure 3, where the algorithm searches for assignments for literals x_1, x_2, x_3. The algorithm, in the first step sets a truth assignment for literals x_1, x_2, x_3 and then, in each iteration, a literal truth value is changed, creating a new candidate solution to be evaluated.

x_1	x_2	x_3	
0	0	0	*step 1*
0	0	1	*step 2*
0	1	0	*step 3*
0	1	1	*step 4*

...

Figure 3: Simple perturbative search example to solve a 3-SAT problem instance.

2.3.3 Stochastic Local Search

Local search algorithms, start at some location of the given search space and subsequently move from the present location to a neighboring location in the search space. Each location has only a relatively small number of neighbors and each of the moves is determined by a decision based on local knowledge only. As an example for a simple local search method for SAT, consider the following algorithm: given a propositional formula F over n propositional variables, randomly pick a variable assignment as a starting point. Then, in each step, check whether the current variable assignment satisfies F. If not, randomly select a variable, and change its truth value from true to false or vice versa. Terminate the search when a solution is found or after a specified number of search steps have been performed unsuccessfully. This algorithm is called Uninformed Random Walk.

Many widely known and high-performance local search algorithms make use of randomized choices in generating or selecting candidate solutions for a given combinatorial problem instance. These algorithms are called stochastic local search (SLS) algorithms, and they constitute one of the most successful and widely used approaches for solving hard combinatorial problems. SLS algorithms have been used for many years in the context of combinatorial optimization problems. Among the most prominent algorithms of this kind are the Lin-Kernighan Algorithm for the Travelling Salesman Problem [56] as well as general search methods such as Evolutionary Algorithms [5] and Simulated Annealing [51]. Stochastic local search algorithms can also be used to solve *NP*-complete problems such as the Graph Coloring Problem (GCP) [42, 13] or the Satisfiability Problem [79, 80].

Local search algorithms start by selecting an initial candidate solution, and then iteratively move from one candidate solution to a neighboring candidate

solution, where the decision on each search step is based on a limited amount of local information only. In stochastic local search algorithms, these decisions as well as the search initialization can be randomized. Furthermore, the search process may use additional memory, for example, for storing a limited number of recently visited candidate solutions.

2.3.4 Stochastic Local Search Example: Random Walk

The two simplest SLS strategies are Uninformed Random Picking and Uninformed Random Walk. Both do not use memory and are based on an initialization function that returns the uniform distribution over the entire search space. SLS algorithms based on this initialization function randomly select any element of the search space S with equal probability as a starting point for the search process.

For Uninformed Random Picking, a complete neighborhood relation is used (i.e., $N = S \times S$ for the search space S) and the step function maps each point in S to a uniform distribution over all elements in S. Uninformed Random Walk uses the same initialization function, but for a given arbitrary neighborhood relation $N \subseteq S \times S$, its step function returns the uniform distribution over the set of neighbors of the given candidate solution. A uniform, random selection from that neighborhood is implemented in each step. Both of these uninformed SLS strategies are quite ineffective, since they do not provide any mechanism for directing the search towards solutions. Nevertheless, in combination with more directed search strategies, both Uninformed Random Picking and variants of Uninformed Random Walk are used for overcoming premature stagnation in complex and much more effective SLS algorithms.

An example of a Random Walk is described in Figure 4. The random walk takes in each step a non-deterministic decision of moving in any of fourth directions: forward, left, backward or right. The walker performs each step based on the current location, moving to a neighboring candidate solution.

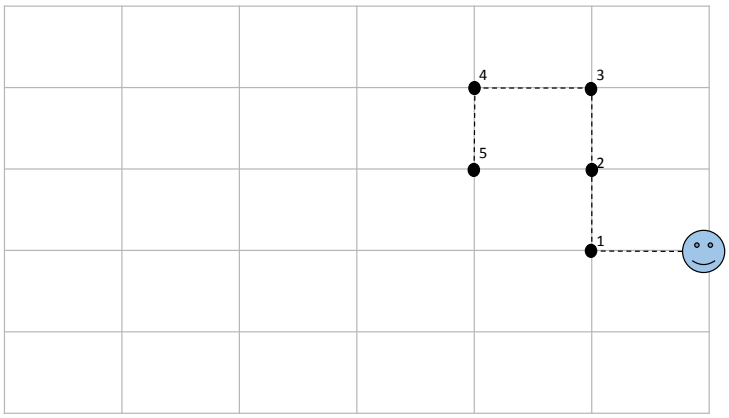

Figure 4: Random walk example

2.4 The SAT Problem

This section presents the k-SAT, the Propositional Satisfiability Problem (SAT) problem having at most k literals in each clause. The 3-SAT problem, which is a special case of the k-SAT, which is the problem solved in this work, is presented as well as its importance, formulation and codification. The SAT problem is the prototypical *NP*-complete problem and was the first problem for which *NP*-completeness was established [17].

2.4.1 The k-SAT Problem

The k-SAT is the propositional satisfiability problem restricted to clauses with k literals, and the 3-SAT problem is a special case for which $k = 3$. The k-SAT for $k > 3$ can always be mapped to an instance of a 3-SAT. The 3-SAT was the first problem to be demonstrated to be NP-Complete [17]. When $k = 2$ the problem is in the complexity class P, as it can be solved in polynomial time [3]. On the other hand, the k-SAT problem is NP-Complete when $k \geq 3$; in fact, it was the first problem proved to be NP-Complete and many NP-complete problems have been proven so, by reducing them to an instance of 3-SAT. Thus, if a polynomial time algorithm to solve the k-SAT for $k \geq 3$ is known, then every NP-complete problem can be solved in polynomial time. However, no such efficient algorithm to solve the 3-SAT is known. The k-SAT for $k > 3$ can be reduced to the 3-SAT by increasing the number of clauses.

The best algorithms known that solve the 3SAT all have exponential order. The stochastic algorithms [69] have been specially successful to solve hard-to-solve algorithms like the 3SAT, one of the is the Schöning algorithm [77]. The Schöning algorithm starts from a random state and performs a local search on a space of 3n elements. In case of not finding a solution, the algorithm randomly chooses another value and repeats the process.

In propositional logic, a literal is either a logical variable or its negation, and a Boolean expression in its conjunctive normal form (CNF) is a conjunction of a set of m clauses, each of whom is a disjunction of literals. Given a Boolean expression,t the Boolean Satisfiability Problem (SAT) consists of determining, if it exists, a truth assignment for the variables that makes a given boolean expression true. The mathematical formulation of the 3-SAT problem is as follows:

- Let there be a set of n literals $X \equiv \{x_1, \ldots, x_n\}$, where $x_r = \{0, 1\}$.
- Let there be a Boolean expression $\Phi = \bigwedge_{i=1}^{i=m} C_i$, formed by a set of m clauses $C = \{C_1, \ldots, C_m\}$, with where $C_i = \bigvee_{j=1}^{3} \tilde{x}_{r_{ij}}$ with $r_{ij} \in [1, n]$ and, either $\tilde{x}_{r_{ij}} = x_r$ or $\tilde{x}_{r_{ij}} = \neg x_r$. $C_i = \bigvee_{j=1}^{3} l_{ij}$, where l_{ij} is either a literal x_r or its negation $\neg x_r$.
- The 3-SAT problem consists in determining the set of values for the literals $X \equiv \{x_1, \ldots, x_n\}$ that makes Φ true.

The stochastic algorithms have been specially successful to solve hard-to-solve algorithms like the 3SAT, one of the is the Schöning algorithm. The

Schöning algorithm starts from a random state and performs a local search on a space of 3n elements. In case of not finding a solution, the algorithm randomly chooses another value and repeats the process. The problem considers a set of literals that can take the value 0 or 1 and a set of clauses where each clause is a disjunction of k literals or a negation of them. The problem consists on finding an assignment of literals such that all clauses are true, in that case the problem is said to be satisfiable [33].

2.4.2 3-SAT Codification

One of the most common codifications of the 3-SAT problem is in CNF form. A 3-CNF expression is a boolean expression such that each clause is a disjunction of exactly 3 literals. An example of a clause in CNF form is shown below. The clauses can be represented as integers. A positive integer represents the literal is presented not negated in the clause and a negative integer value represents that the literal is present negated in the clause. In the example shown, the clause $x_1 \vee \neg x_2 \vee x3$ is represented by the integers $1 - 23$.

$$\neg x_1 \vee x_2 \vee \neg x_3$$

The first line of the file has the number of clauses and literals in the problem as $p\ number\_of\_literals\ number\_of\_clauses$. A problem with 100 literals and 500 clauses would have *p 100 500* as the first line of the file.

2.4.3 Randomly Generated SAT Instances

Many empirical studies of SAT algorithms have made use of randomly generated CNF formula and have been proposed and studied in the literature [80]. In most cases, they are obtained by means of a random instance generator that samples SAT instances from an underlying probability distribution over CNF formula. The probabilistic generation process is typically controlled by various parameters, which mostly determine syntactic properties of the generated formula, such as the number of variables and clauses, in a deterministic or probabilistic way [64].

One of the earliest and most widely studied classes of randomly generated SAT instances is based on the random clause length model: Given a number of variables, n, and clauses, m, the clauses are constructed independently from each other by including each of the 2n literals with fixed probability p [31]. A variant of this model was used in Goldberg's empirical study on the average case time complexity of the Davis Putnam algorithm [37]. Theoretical and empirical results show that this family of instance distributions is mostly easy to solve on average using rather simple deterministic algorithms [18, 32]. As a consequence, the random clause length model is no longer widely used for evaluating the performance of SAT algorithms.

To date, the most prominent class of randomly generated SAT instances that is used extensively for evaluating the performance of SAT algorithms is based on the fixed clause length model and known as Uniform Random k-SAT [31, 64]. For a given number of variables n, a number of clauses m and a clause length k, Uniform Random k-SAT instances are obtained as follows. To generate a clause, k literals are chosen independently and uniformly at random from the set of $2 \times n$

possible literals (the n propositional variables and their negations). Clauses are not included into the problem instance if they contain multiple copies of the same literal, or if they are tautological, that is, they contain a variable and its negation. Using this mechanism, clauses are generated and added to the formula until it contains m clauses overall.

The 3-SAT problem instances used in this work were generated by the G3 algorithm [68], creating problem instances with few solutions. The G3 algorithm is based on creating a random solution for the 3-SAT problem and then generate m clauses related to such solution. Experimental results of the G3 algorithm shows that the algorithm generates unique solution problems for $m > 440$ with high probability [68]. The relation between the number of literals n and the number of clauses m use to generate hard-to-solve 3-SAT problem instances is defined by Eq 2. The relation is based on a theoretical and exhaustive experimental analysis of hard-to-solve 3-SAT problem instances [19, 93].

$$m \geq 4,26 \times n + 6,24 \tag{2}$$

2.5 Summary

This section introduced optimization and combinatorial problems. Large scale optimization problems were introduced in this section and few classical examples were mentioned. The k-SAT problem was introduced and the particular case when k=3 was presented, which is is the problem solved in this work. Finally, the 3-SAT mathematical formulation, codification and problem instance generation was presented.

3 Distributed Computing Techniques

Distributed computing techniques are usually used to solve LSO problems. Different models of communication between process as well as different approaches to implement a distributed algorithm and libraries available is presented in this section. Section 3.2 introduces the different parallel computing models used to implement a distributed algorithm. This section also introduces the MapReduce programming model in Section 3.3.1, which is the programming model used in this work to solve the LSO problem presented.

3.1 Parallel Computing paradigms

Distributed algorithms are used to improve an algorithm performance distributing its workload among different process. These process can run on the same physical node, exploiting multicore architectures, or distributed in a cluster of computers, horizontally scaling to hundreds of computers. There are two main parallel computing techniques: *parallel computing with shared memory* and *parallel computing with distributed memory* [59].

3.1.1 Parallel Computing with Shared memory

The shared memory paradigm is based on that different threads or processes that belong to the same parallel algorithm share a space of memory, through physical memory or belong to the same process, being able to exploit multicore architectures. The main advantages of this programming model is that it's relatively easy to develop and has low latency as the different processes or threads run or are executed in the same computer. The main disadvantage of this programming paradigm is that it cannot exploit distributed architectures such as grids or cluster of computers [12]. Figure 5 shows a graphical representation of the communication of four threads using shared memory.

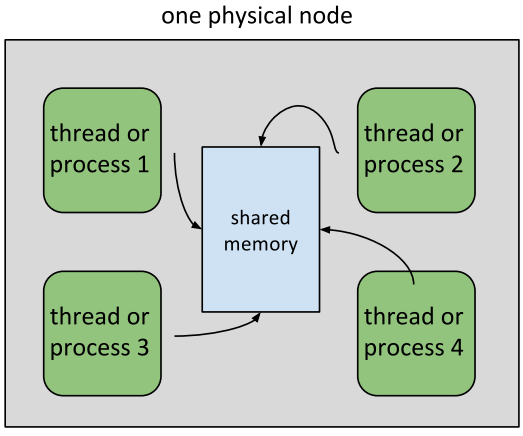

Figure 5: Parallel algorithm example using shared memory

Among the most known libraries to implement parallel algorithms using

shared memory is openMP [12], which is an API composed by compiler flags and a set of routines and environment variables that enable an easy and fast parallelization of a sequential algorithm.

3.1.2 Parallel Computing with Distributed Memory

The distributed memory model is based on that different process that work as a whole to solve one problem are executed in different computers and the communication among them is based on message communication. The main advantage of the distributed memory programming model is that it exploits distributed environments such as grids of cluster of computers. The main disadvantage of this programming model is the cost of communication. All the processes are executed on different physical computers so network communication cost have to be considered when designing the algorithm. If the distributed algorithm communicates too often, it can have a negative impact on the performance of the algorithm. An example of a distributed algorithm using the distributed memory paradigm is shown in Figure 6, where three nodes communicate between each other through the network.

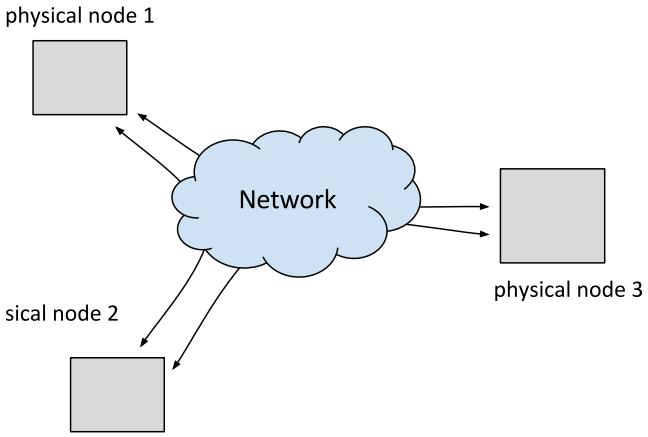

Figure 6: Parallel algorithm example using distributed memory

The most known library to implement distributed algorithms is the Message Passing Interface (MPI) [28]. MPI is the standard library for parallel programming under the paradigm of communication between processes using messages. The library was first developed by IBM, Intel, PVM and nCUBE whose objective was to design an efficient standard for parallel programming with distributed memory.

3.2 Models of communication between process

Parallel computing has been used in cluster for many years to run distributed algorithms in hundreds of nodes. The idea of a distributed algorithm is that different process cooperate between each other for a common goal. Depending

on the goal, there are three main models of communicating such process running on distributed computers.

3.2.1 Master-Slave

This is the simplest paradigm of communication between process and is the most common model used to implement a distributed algorithm [85]. In this paradigm, a problem is split into a set of subproblems and related process that solve such subproblems are created. There are two kinds of processes: the master and the slave. The master process is unique and controls the set of slaves, and the slaves process information. The master process is responsible for starting the slaves and for sending necessary information to process. The slave process, process information received from the master process. The communication is usually from the master to the slave. The slaves communicate with the master only to send execution results. The communication and control is centralized in the master and, although communication could exists among slaves, usually is none. An example of Master-Slave paradigm of communication is shown in Figure 7.

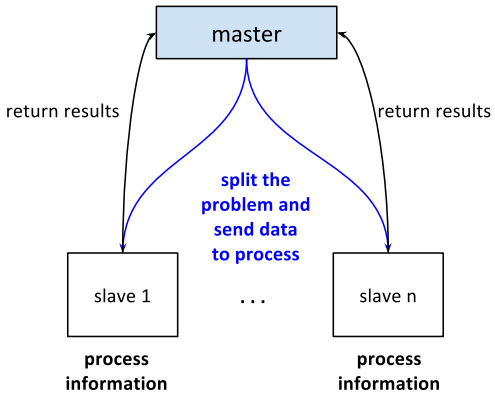

Figure 7: Example of process communication using master-slave model.

3.2.2 Client - Server

This paradigm of communication identifies two processes: the client process that request services to process of another type, the servers, that attend those requests [59]. The client/server model provides a mechanism of communication for distributed applications in which the process can work as client and server simultaneously for different services. The model client/server is the dominant model for distributed applications over the internet and is a model massively used in the TCP/IP protocol and web services. The main process, the server, is permanently active waiting for requests from clients and usually many clients consume services from only one server. An example of the client-server paradigm of communication is shown in Figure 8.

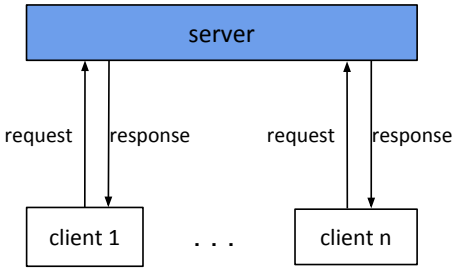

Figure 8: Example of process communication using client-server model.

3.2.3 Peer to peer

The peer-to-peer parallel computing model of communication is based on that each process has the same capabilities to establish communication. This model can be implemented as a client-server model in which each process is a server and a client at the same time. An example of the peer-to-peer paradigm of communication is shown in Figure 9.

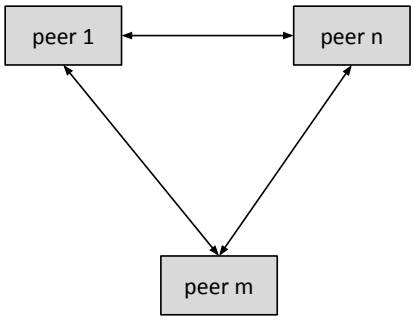

Figure 9: Example of process communication using peer-to-peer model.

3.3 Computational models for BigData & Graph Processing

3.3.1 MapReduce Programming Model

The MapReduce paradigm was first introduced by Google [23] as a mean to develop inverted indexes for web search. MapReduce was inspired in ideas from functional programming, distributed computing and data base systems. MapReduce jobs process the information in batch, allowing the processing of huge amounts of unstructured information in reasonable execution times, being this the main feature of a MapReduce job. MapReduce was designed to being able to implement a distributed algorithm to analyze huge amounts of information without having to solve the problems of a distributed algorithm over and

over [23].

The MapReduce paradigm is based on applying a *map* function that performs (in parallel) filtering, sorting, and/or computation and a *reduce* function that performs a summary operation based on the results of the map function. The MapReduce paradigm was first introduced by Google [23] as a mean to develop inverted indexes for web search to avoid being solving the same problem over and over again. MapReduce was inspired in ideas from functional programming, distributed computing and data base systems.

A MapReduce job has two main phases: the map phase and the reduce phase. The map phase of a MapReduce job is composed by the following steps: *record reader*, *mapper*, *combiner*, *partitioner*. The output of the map phase of a MapReduce job is a set or group of keys and values, named intermediate keys and values. The intermediate keys and values are a set of keys and values that are grouped by key, which will be sent to the reduce phase of the algorithm. The reduce phase of a MapReduce job is composed by the following steps: *shuffle*, *sort*, *reduce*, *output format*.

Traditional relational databases are often compared with the MapReduce paradigm as both are used for similar purposes: storing and reading information. The MapReduce paradigm and relational databases are usually complementary and not substitutes. A comparison between traditional databases and the MapReduce paradigm is presented in Table 1. The table shows the type of information to be saved in each type of structure by how the data is manipulated and used. The main difference between MapReduce and a traditional database is the volume of information managed and the structure of the information manipulated. MapReduce works best for big volumes of unstructured data. On the other hand, traditional databases work best with gigabytes of information, structured information with high integrity.

	Traditional RDBMS	MapReduce
Data size	gigabytes	petabytes
Access	interactive & batch	batch
Updates	read & write many times	write once, read many times
Structure	static schema	dynamic schema
Integrity	high	low
Scaling	nonlinear	linear

Table 1: Traditional RMDB vs MapReduce paradigm

MapReduce Design Patterns

The MapReduce design patterns are a general framework for solving problems with the MapReduce paradigm. The MapReduce patterns are divided in six families: *summarization patterns*, *filters patterns*, *data organization patterns*, *join patterns*, *meta patterns*, *IO patterns*.

Summarization Patterns The algorithms included in this pattern family are based on calculating summarization of information or counting specific information in large sets of data. Examples of the application of this pattern are:

counting the visits of a particular web page, counting the number of transactions in a specific country, calculate the summary of web visits by country, number of transactions by user, etc.

Filters Patterns This pattern family is based on finding a subset of a set of information, without modifying the original set of information. Examples of this kind of problems are: top-ten listing, the information generated by a particular user in a period of time. They are based on reducing information to a smaller set so that it can be deeply analyzed.

Data Organization Patterns This pattern family is based on the transformation of the information or the reorganization of it. Many times, the information manipulated by a MapReduce algorithm is the input to another data analysis tool or data warehouse. An example of application of this pattern is a job that process information, orders and sorts it and then saves it to a data warehouse.

Join Patterns This family is based on joining information from different sources or types of information. To implement a join in MapReduce two different sets of data are manipulated at the same time, typically with different structures. Joining information in MapReduce is network expensive, as the data is not filtered before the reduce phase so all the information has to travel to reducers to be processed.

Meta Patterns This family is based on patterns of patterns before mentioned, such as job chaining and job merging. An example of application of this pattern family is when a MapReduce output is the input for another MapReduce job.

IO Patterns This family is based on the customization of the IO of MapReduce jobs. The transformation of how Hadoop saves or reads the information. An example of application of this pattern family is when the output of a MapReduce job is saved to Hbase, or when the input is read from a source other than HDFS, which is the default source of information for MapReduce jobs in Hadoop.

WordCount example

The word count example is the most basic example of a MapReduce job, is the "Hello World" for MapReduce. Summarization patterns are used to implement the word count , which consists of finding the frequency (or number of appearances of different words) of the words in a very large set.

In the summarization patterns, the mappers return the key by which are summarizing and as value the sum. In the word count example, the key used by the mappers is the word, and the value is the number one (meaning that the word appeared once in the received line). The reducers receive as key the keys emitted by the mappers (a word), and as value a list of numbers meaning the number of appearances of that word in all the dataset. The reducer sums all

values received and returns as key the received word, and as value the calculated sum.

The word count example is shown in Figure 10, where two input files are received containing two lines each. In this example, four mappers are created, each one receiving one line. Each mapper emits the word as key and the number one as value, for each word received. Finally, the reducer receives the information emitted by the mappers grouped by word (the key used by the mapper) and sums the values, resulting in the final output of the problem. The code for the mapper is presented in Listing 1, and the code for the reducer is shown in Listing 2.

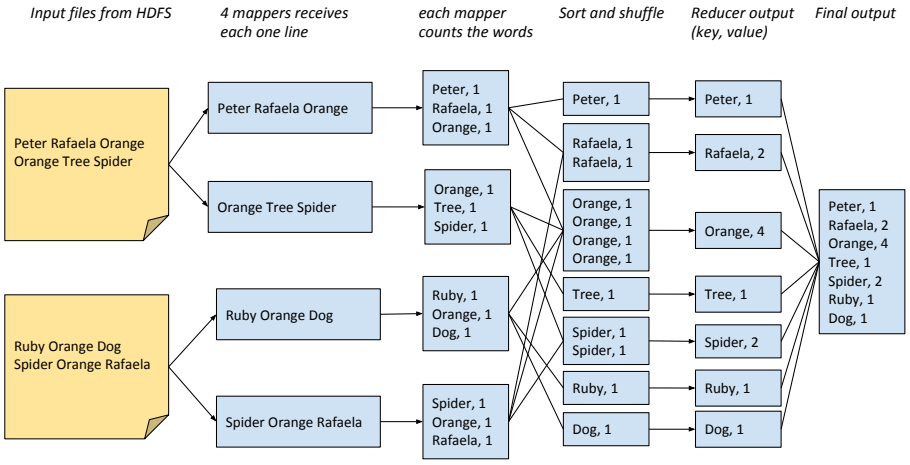

Figure 10: Wordcount example

```
public class WordCountMapper extends MapReduceBase implements
    Mapper<LongWritable, Text, Text, IntWritable> {
    //Initialize Hadoop types to use
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key,
    Text value, OutputCollector<Text, IntWritable> output,
    Reporter reporter) throws IOException {
        //taking one line at a time and tokenizing it
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);

        //iterating through all the words available in that line,
    create the key value pair forming the key value pair
       while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            output.collect(word, one);
        }
    }
}
```

Listing 1: Code for the mapper for the word count example

```
public class WordCountReducer extends MapReduceBase implements
Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key,
                       Iterator<IntWritable> values,
                       OutputCollector<Text,
                       IntWritable> output,
                       Reporter reporter) throws IOException {
       int sum = 0;
       /*iterates through all the values available, add them and
    return the received key as the key and sum of its values as
    value*/
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
 }
```

Listing 2: Code for the reducer for the word count example

3.3.2 Pregel Programming Model

As web and social media evolved, many companies are in the need to process and analyze large scale information. Many web and social media problems concern large graphs with more than a billion vertices [14]. For example, finding the shortest friendship path among two users of a social network or to find which users have more influence over the others in a specific social network like twitter. Despite MapReduce was first proposed to analyze large datasets in a cluster or grids, it is not suitable to solve graph algorithms. Different approaches can be taken to tackle graph problems in a large scale, being some of them [61]:

- Developing a custom distributed infrastructure requiring effort to be repeated for each new algorithm.
- Relying on existing distributed infrastructure like MapReduce leading to suboptimal solutions.
- Using single computer for solving graph algorithms, but that does not scale for big problems.
- Using existing parallel graph system that do not address important issues in distributed environments like fault tolerance.

Graph processing systems are suitable for social science problems where communication between nodes is needed and the model can be represented as a graph, where the nodes communicate with each other. Pregel [61] was first proposed to tackle large scale graph processing in a cluster of computers in an efficient manner, without having to address distributed algorithm problems. Pregel is a flexible, scalable and fault-tolerant computing model based on Valiant's Bulk Synchronous Parallel model [90]. Pregel programs are expressed as a sequence of iterations (or supersteps) where in each iteration the following actions can be done:

- A vertex can receive a message (or messages) sent in the previous iteration of the algorithm by another vertex.

- A vertex can send messages to other vertices (messages that will be received in the next iteration). Typically, one vertex can send a message along one of its outgoing edges to another vertex.
- A vertex can modify its state and the state of its outgoing edges (in a directed graph).
- Mutate graph topology.

Pregel model is an efficient, scalable and fault-tolerant framework to run graph algorithms on clusters of thousands of commodity computers, and its implied synchronicity makes reasoning about programs easier [61]. Pregel is a proprietary implementation from Google. There exists some open sources implementations that implement the Pregel paradigm such as Giraph [62], GraphX [95, 38] and GraphLab [57].

Giraph is an open source implementation of Pregel which uses Apache Hadoop MapReduce implementation to process graphs. Giraph is currently used at Facebook to analyze the social graph formed by users and their connections and has been successfully used to process a graph with over one trillion edges [14]. The key aspect of Giraph is that runs over Hadoop, which is now the most used framework for big data processing in the industry.

GraphX is a graph processing framework, which also implements the Pregel model but also many other graph algorithms. GraphX runs over Spark and provides a simple api for graph processing operations like triangle count, number of connected components, page rank, etc.

Pregel Example: Friends recommendations

This section presents a simple example of application of Pregel.The example problem consists in finding friends recommendations for each user. A friend recommendation (or potential friend) for a user A can be defined as the users that share a common friend with A. Consider Facebook for example, where the friendship relation is reciprocal, a user A is friend of the user B then, the user B is also friend of user A. Facebook social network can be represented as a graph where each vertex (node) is a user and nodes are connected by an edge if they are friends.

The Figure 11 shows how to use Pregel paradigm to calculate the friends recommendations for each user in a graph. The first step in a pregel process is usually an initialization process, sometimes called *superstep 0*. In the example shown, the set of friend recommendations is initialized as an empty set for all vertices. In each iteration of the pregel algorithm, each node sends to its neighbors the current list of recommended friends plus itself. So, in superstep 1, each node has its neighbors as the set of recommended friends (e.g.: the set of recommended friends for node a in superstep 1 is b, which is a's only neighbor). The process is repeated again but now the set of recommended friends is not empty, so each node receives from its neighbors a set containing all neighbor's neighbors. For example, node a receives from node b all b's neighbors, a,c,d. Finally, each node removes duplicates from the list and all neighbors (because they are already friends) and the algorithm ends.

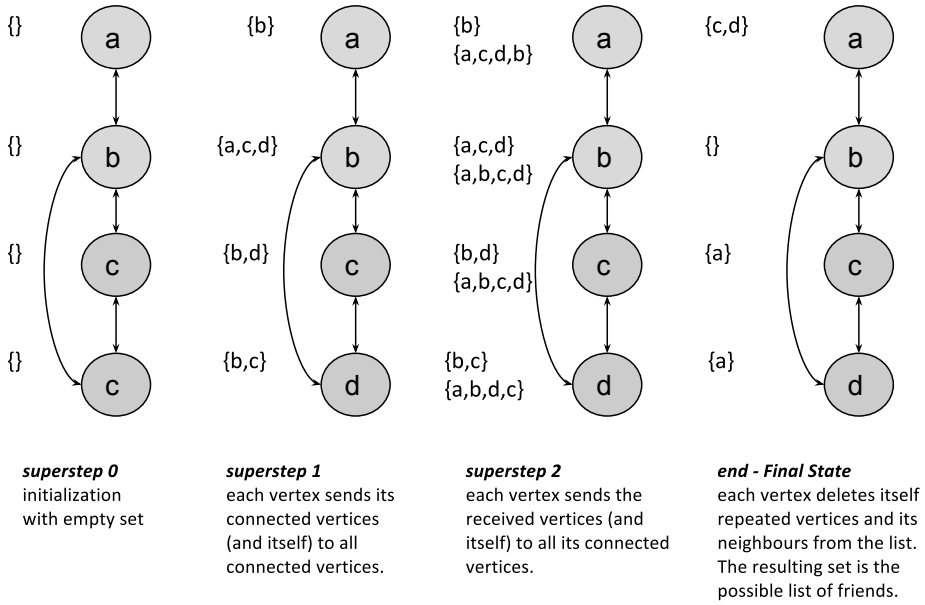

Figure 11: Example problem of using pregel to calculate friend recommendations for each user in a social network.

3.4 Performance metrics

There are different performance metrics used to measure the performance of a distributed algorithm. A metric of performance does not depend on the specific hardware being used and must be a function of the size of the problem to be solved. To evaluate the efficiency of an algorithm the number of steps required to solve the problem is considered.

Intuitively two main factors can be taken into account to evaluate an algorithm performance: the execution time and the usage of resources available. The execution time is commonly used as a metric of performance to evaluate the efficiency of an algorithm. The execution time depends on multiple factors such as complexity, size of the problem, number of tasks used (in case of a parallel algorithm) and of processing elements (hardware, networks, etc) [33].

Speedup

The speedup is a measure of the improvement in the performance of a parallel algorithm using many processing nodes, compared to the efficiency of the same algorithm using only one processing node. There are two types of speedup: *absolute speedup* and *algorithmic speedup.*

Absolute Speedup The absolute speedup compares the execution time of the best sequential algorithm that solves the problem (T_0) with the execution time of the parallel algorithm using N processors (T_N). The absolute speedup is defined by Eq 3

$$S\_absolute(N) = \frac{T_0}{T_N} \tag{3}$$

Algorithmic Speedup The algorithmic speedup compares the execution time of the same parallel algorithm executed in one processor (T_1) with the execution time of the parallel algorithm executed using N processors (T_N). The algorithmic speedup is defined by Eq 4.

$$S\_algorithmic(N) = \frac{T_1}{T_N} \tag{4}$$

The algorithmic speedup is most commonly used to evaluate the improvement of the performance of the parallel algorithms when using the executing time as a measure of performance. The absolute speedup is difficult to calculate because not always the best algorithm to solve a problem is known.

3.5 Summary

This section presented different distributed algorithm techniques and introduced the performance metrics used to measure the performance of the algorithms devised in this work.

The state of the art of many distributed algorithms to solve the 3-SAT are based on the techniques described above and many of them use the libraries before mentioned, like MPI. This work presents a novel idea to use MapReduce programming paradigm to implement a distributed algorithm to solve a LSO problem.

4 Hadoop & MapReduce

This section introduces the Hadoop framework, its structure and the Hadoop MapReduce job execution environment. A basic MapReduce example is presented as a basic example of how the MapReduce programming model works. The Hadoop distributed filesystem and the Hadoop database are also introduced in this section.

In the last decade, the information generated has grown exponentially. The capability of analyzing all these information available is a powerful tool to analyze human behavior. Digital information is doubled every two years and it is estimated to reach 44 trillion gigabytes by 2020, increasing in a factor of 10 from 2013 to 2020 [1]. All this digital information comes from different sources, such as geo-localization, multimedia information, web searches, logs, etc. The trend is that the information generated by each person grows each year. Nevertheless, digital information created by different systems is greater than the information created by users. System information is created by GPS logs, sensors, tracking information and many other systems.

Digital information is growing very fast and is composed by almost 80% of unstructured information and 20% of structured information, growing the unstructured information 15 times faster than the structured information [92].

Different frameworks have been proposed to analyze big volumes of unstructured information in a distributed environment. Hadoop is the most known framework for analyzing unstructured information in a distributed environment [92]. Hadoop has been designed to execute algorithms in clusters of commodity computers running MapReduce jobs, providing a distributed file system.

4.1 Apache Hadoop

Hadoop is currently the most used framework to analyze large volumes of information using the MapReduce programming model. Hadoop is a distributed system initially proposed as a framework to execute MapReduce tasks. Hadoop includes an open source implementation of the Google File System (GFS) [35], named Hadoop Distributed File System (HDFS) [81, 82]. Hadoop provides an abstraction level that allows these developers, having little knowledge about distributed computing programming, to easily implement distributed algorithms that could run on hundreds of commodity machines to analyze huge amounts of information in reasonable execution times. Hadoop was not designed for solving optimization problems or computer intensive problems. Nevertheless, the framework provides the infrastructure needed to run MapReduce jobs solving the classical problems of distributed algorithms like fault-tolerance, process communication and data replication transparently to the developer.

4.1.1 Hadoop Ecosystem

The Hadoop ecosystem is extended beyond the infrastructure to run MapReduce jobs or the HDFS. Many different projects or frameworks run over the Hadoop infrastructure. Some related projects of the Hadoop ecosystem are mentioned below. Some of these projects are included in the Apache Hadoop distribution and others (like Hbase) are currently full Apache projects.

- ***Common:*** Includes the common artifacts of Hadoop for distributed systems and IO (serialization, remote procedure call, persistence).
- ***Avro:*** Is a data serialization system, which provides rich data structures including compact and fast binary data formats. Also, remote procedure calls (RPC) is including with simple integration with dynamic languages [29].
- ***MapReduce:*** Hadoop includes the MapReduce runner for running distributed algorithms using the MapReduce paradigm.
- ***HDFS:*** [82] Is Hadoop distributed file system that provides high-performance access to data across Hadoop clusters of commodity machines. The file system is designed to be fault-tolerant providing replication and scalability. The Hadoop distributed file system is based on Google File System [35].
- ***Pig:*** Pig is an abstraction of MapReduce. Pig programs are developed using a pseudo SQL language which creates MapReduce tasks.
- ***Hive:*** Hive is a distributed data warehouse. Hive manages and maps HDFS information and enables a SQL-like language that is translated to MapReduce jobs [86].
- ***Hbase:*** [34] A column oriented distributed database. Hbase uses HDFS and supports MapReduce jobs. It is based on Google BigTable [11].
- ***ZooKeeper:*** Is a centralized service for maintaining configuration information, naming, providing distributed synchronization. It enables the communication and transfer of reduced scheduling information that help the developer to develop distributed algorithms [45].
- ***Sqoop:*** Sqoop is a tool to efficiently transfer information from RDBMS to the Hadoop ecosystem, mainly focused on HDFS and Hive [50].

4.2 Hadoop MapReduce implementation

The most known and widely used implementation of the MapReduce programming model is the Hadoop implementation. Hadoop was designed and developed to run MapReduce jobs in a distributed environment [92]. Although the MapReduce paradigm is easy to understand and describe, it is not always easy to express an algorithm in terms of map and reduce functions.

The MapReduce paradigm works by splitting the processing of data in two main steps, the map function and the reduce function. Usually, the input of the mapper is raw data saved in the HDFS. Text input format is the default format, which specifies the lines of the raw file as values for the mapper and the byte offset of the beginning of the line from the beginning of the file as key. A MapReduce job consists of the input data, the MapReduce program, and configuration information. Hadoop runs the job by dividing it into tasks. There are two types of tasks: map tasks and reduce tasks.

The main phases of a MapReduce job are briefly mentioned below where the user code is executed on the map and on the reduce phases. Nevertheless, the user can implement customs partitioners, record readers, input formats and combiners depending on the algorithms needs [92].

- *InputFormat*: The InputFormat specifies the input for the MapReduce job. The MapReduce framework relies on the InputFormat to: i) validate the input for the job; ii) split the input file into logical splits, each of which is assigned to an individual mapper; iii) provide the RecordReader an implementation to be used to translate input records from logical splits to data structures understandable by the mappers.

- *RecordReader*: The RecordReader translates the generated splits created by the InputFormat from the input files to key-value objects to be sent to the mappers. By default, each mapper receives a line of the input file at a time. The byte offset of the line in the input file is used as the key and the line as the value. New RecordReaders could be implemented to extend the default functionalities provided by Hadoop.

- *Map*: The mapper is where the developer maps the input retrieved from the RecordReader in the format (key-value) and returns another set of key-value pairs. The keys that the mappers write are the values to be used to group in the reduce phase.

- *Combiner*: A Combiner is a reducer that runs in the same physical computer as the mapper. The goal of the Combiner is to optimize the MapReduce algorithm by running a reduce operation in the same physical node as the mapper, thus reducing network traffic. The Combiner is run before the intermediate sets key-value are sent to the reducers.

- *Partitioner*: The Partitioner takes the set of intermediate key-value pairs that are outputs of the mappers and groups them by the keys.

- *Reduce*: Each reduce process receives a key, written by the mappers, and a list of related values. The reducer groups and applies a given function (implemented by the user) to the key-value pairs received as input. The reducer writes the output as key-value pairs to an output folder, specified in the MapReduce job. This is the last phase of a MapReduce algorithm and the Reducer has the responsibility of calculating the final result or to prepare the information for the next MapReduce job.

4.2.1 Job execution

A common Hadoop cluster includes the master node and multiple slave nodes. The master node is composed by the several processes: a *JobTracker*, a *TaskTracker*, a *namenode* and a *datanode*. A slave node is composed by a the following processes: a *datanode* and a *TaskTracker*. The JobTracker coordinates all the jobs run on the system by scheduling tasks to run on TaskTrackers. TaskTrackers run tasks and send progress reports to the JobTracker, which keeps a record of the overall progress of each job. If a task fails, the JobTracker can reschedule it to execute on a different TaskTracker. Both the namenode and the datanode belong to the HDFS cluster. These are daemon processes and they are used to manage the namespace and the distributed data and they are further explained in Section 4.3.1. A graphical representation of Hadoop processes is presented in Figure 12, including the MapReduce runner and the HDFS cluster.

Hadoop divides the input of a MapReduce job into pieces with a fixed size called input splits. Hadoop creates one map task for each split. Each split is

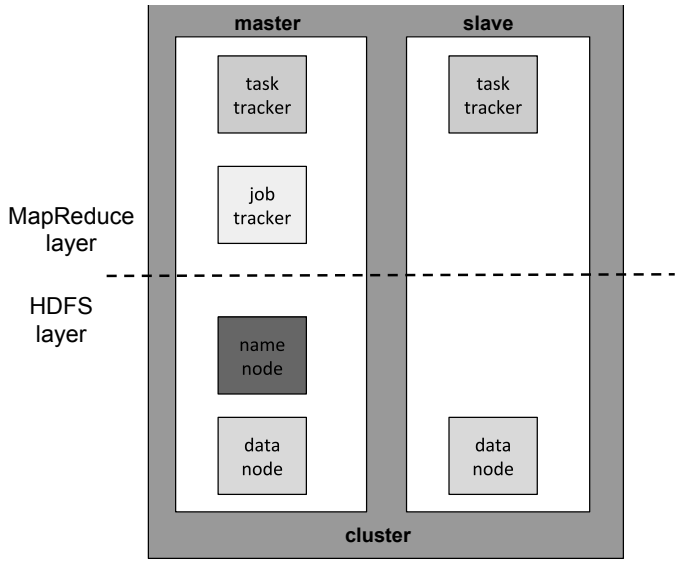

Figure 12: Hadoop diagram daemons

divided into records, and the map processes by running the user-defined map function for each record in the split, emitting key-value pairs. The default split size is of the size of a HDFS block (64 MB), although this parameter can be configured or specified by the InputSplit used.

Hadoop runs the map tasks on the node where the input data resides in HDFS. The output of the mapper task, named intermediate key-value pairs, is written directly to the local disk on the node where the mapper was executed. Usually, the input of a reduce tasks is the output from different mappers. Thus, the sorted map outputs have to be transferred across the network to the node where the reduce task is running. In this node, the results are merged and then passed to the user-defined reduce function.

The output of the reduce task is normally stored in HDFS using replicas. For each HDFS block of the reduce output, the first replica is stored on the local node, while the other replicas are stored on off-rack nodes. Thus, writing the reduce output does consume considerable network bandwidth. When multiple reducers are used, the output of the mappers are partitioned. Each mapper creates one partition for each reduce task, based on the key used as output in the mapper process. Many keys and their associated values exist in each partition, but the records for any given key are all in a single partition. Thus, each reducer will receive a single partition as input. A basic execution of a MapReduce task is shown in Figure 13, having three mapper tasks and a single reduce task.

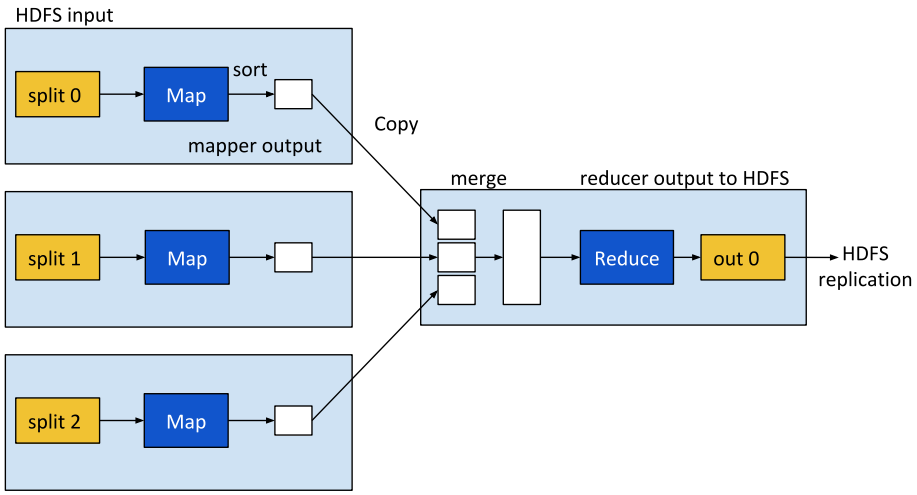

Figure 13: MapReduce job execution having only one reducer.

Figure 14 shows the execution flow of a MapReduce job executed in Hadoop. First, a MapReduce job is created in the client node; this job is executed inside a JVM. The JobClient submits a new job to the JobTracker, which centralizes all job executions and a new job id is returned. After that, in step three, the execution jar, the distributed cache information and required resources are copied to the nodes. The job is submitted to the JobTracker in step four and the job id is retrieved in step 5. The JobTracker initializes the job and retry the input for the job in step six. The TaskTrackers communicate with the JobTracker using messages to send information about availability and running capacity. The JobTracker assigns the job execution to a TaskTracker to run the map or reduce task. Before starting the map or reduce task, the TaskTracker retrieves the required resources to run the task, for example the executable jar file and other information. Finally, in case there is not a JVM already created, the TaskTracker creates a new JVM and launches the map or reduce task.

The JobTracker listens to TaskTracker control messages from the TaskTrackers, known as *heartbeat signals*. Heartbeat signals are sent from each TaskTracker to the JobTracker, who responds with specific commands to the TaskTracker. The heartbeat signals are messages containing synchronization information between the TaskTracker and the JobTracker, which is used to track tasks that are running in the Hadoop cluster. The data available in the heartbeat messages includes data about the worker managed by the TaskTracker; e.g. such as virtual memory and physical memory available, CPU information, etc, if the process has started or restarted, an identification for the first heartbeat of the tracking process or the first since a refresh; and a notification when the worker is ready to receive new tasks to execute.

TaskTrackers run a loop that periodically send heartbeat signals to the JobTracker, which in turn assigns a task from pending jobs to execute in the TaskTracker. TaskTrackers have a fixed number of slots for map tasks and for reduce tasks(for example, a TaskTracker may be able to run two map tasks and two

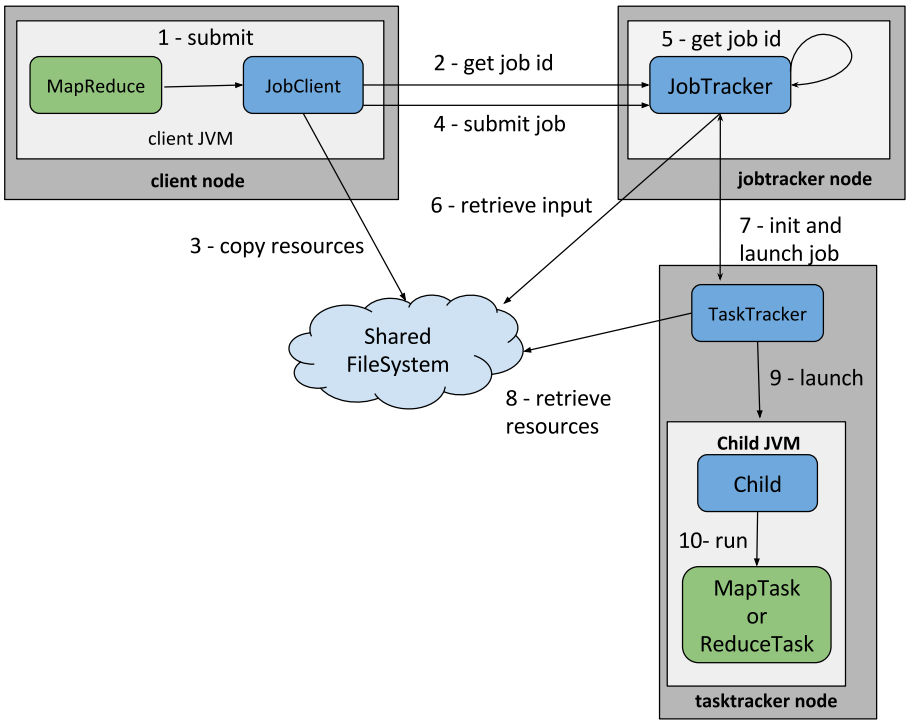

Figure 14: Example of a MapReduce job execution flow

reduce tasks simultaneously). Map tasks have higher priority than reduce tasks, so if the TaskTracker has at least one empty map task slot, the JobTracker will select a map task to execute; otherwise, it will select a reduce task. For a map task, the JobTracker takes into account the TaskTracker network location and picks a task whose input split is as close as possible to the TaskTracker, in order to improve performance. In the optimal case, the task may run on the same physical node where the data resides. Alternatively, the task may run on the same rack but not on the same node as the split.

When there are tasks to execute and a TaskTracker indicates that a worker is ready, the JobTracker creates the cleanup and setup tasks (one for each mapper). These tasks are special processes that allow the JobTracker and the TaskTracker to run the user programs without compromising the security and stability of Hadoop. When a TaskTracker has a task assigned, it executes by copying the the jar file of the job and all the data required for execution from the distributed cache to the local node. After that, an instance of TaskRunner is created to run the job. The TaskRunner launches a new JVM, which is reused in future executions to run each task so any problem related to the user program does not affect the TaskTracker.

When a task is running, it keeps track of the progress of completion of the task. For map tasks, this is the proportion of the input that has been processed. For reduce tasks, the system estimates the proportion of the reduce input processed. Progress is not always measurable, but it is important as it

used by Hadoop to know if a certain task is working. For example, a task writing output records is making progress, even though it cannot be expressed as a percentage of the total number that will be written, since the latter may not be known.

4.2.2 Task Failure

The most common way for a child task failure is when the user code in the map or the reduce task throws a runtime exception. If this happens, the child JVM reports the error back to its parent TaskTracker, before it exits. The TaskTracker marks the task execution attempt as failed, freeing up a slot to run another task. Another failure situation is the sudden exit of the child JVM. In this case, the TaskTracker notices that the process has exited and marks the task execution attempt as failed.

When the TaskTracker notices that is not receiving progress updates for a while from the executed task, it marks the task as failed. In this case, the child JVM process is automatically killed. This is the case when the tasks hang or are blocked.

When the JobTracker is notified by the TaskTracker that a task execution attempt that has, the task that was being executed by the TaskTracker is rescheduled by the JobTracker. The JobTracker tries to avoid rescheduling the task on a TaskTracker where it has previously failed. Furthermore, if a task fails a specific number of times (by default, configured to 4 times), the JobTracker does not retries the execution further. In this case, the JobTracker finishes the job. For some applications, it is undesirable to abort the job if a few tasks fail, as it may be possible to use the results of the job despite some failures. In this case, the maximum percentage of tasks that are allowed to fail without triggering job failure can be set for the job. The number of map tasks and reduce tasks that are allowed to fail is configured by two independent configuration properties.

4.2.3 TaskTracker Failure

The TaskTracker stops sending heartbeat signals to the JobTracker (or sends them infrequently) by two main reasons: i) TaskTracker crashes; ii) TaskTracker poor performance by running slow. The JobTracker will notice that a TaskTracker has stopped sending heartbeats if it has not received one for a specific time period (by default, configured to 10 minutes). In this case, the JobTracker removes the TaskTracker from its pool of TaskTrackers to schedule tasks on. The map tasks that were completed successfully on a failing TaskTracker are rerun, since their intermediate output is written to the local filesystem of the the failed TaskTracker and could not be accessed to by the reduce task. Any tasks in progress are also rescheduled.

A TaskTracker can be blacklisted by two reasons: i) an administrative user can blacklist a TaskTracker from the command line; ii) if the TaskTracker fails more than a specific number of times (by default, four times), then the JobTracker marks the TaskTracker as blacklisted. A blacklisted TaskTracker can be removed from the blacklist by: i) it is automatically removed from the blacklist after a day from being blacklisted; ii) from the command line by an administrative user; iii) if the TaskTracker is restarted

4.2.4 JobTracker Failure

The most serious failure situation of Hadoop is when the JobTracker fails. The JobTracker is a single point of failure and, currently, the Apache Hadoop distribution has no mechanism for dealing with failure of the JobTracker. In this case, the whole job fails. Usually, depending on the physical machine where the JobTracker runs, this failure situation has a low chance of occurring, since the chance of a particular machine failing is low.

Fault-tolerance for the JobTracker have been studied and implemented using checkpoints (but is not included in the main Apache Hadoop distribution). In the proposed schema, the JobTracker saves snapshots of the system state at certain times. If the JobTracker fails, the system automatically restarts the JobTracker using the last snapshot saved. Short execution time overheads have been reported after a JobTracker failure [55].

4.2.5 MapReduce Input Types and Input Formats

MapReduce has a simple model of data processing: inputs and outputs for the map and reduce functions are key-value pairs. Hadoop supports different types of input and output formats to be used as keys and values for mappers and reducers.

The map and reduce functions in Hadoop MapReduce have the following general form:

- map: $(K_1, V_1) \rightarrow \text{list}(K_2, V_2)$
- reduce: $(K_2, \text{list}(V_2)) \rightarrow \text{list}(K_3, V_3)$

In general, the map input key type K_1 and the value type V_1 are different to the map output types (K_2 and V_2) respectively. The reduce input K_3 and V_3 must have the same types as the map output ($K2{=}K_3$ and $V2{=}V_3$), although the reduce output types may be different (K_4 and V_4). The partition function applied to the mapper intermediate key and value types (K_2 and V_2 respectively), and it returns the partition index.

Hadoop provides a large list of types to be used by mappers and reducers that are suitable for MapReduce algorithms. The list of types that come with the default Apache Hadoop distribution is: BooleanWritable, ByteWritable, IntWritable, VIntWritable, FloatWritable, LongWritable, VLongWritable, DoubleWritable, NullWritable, Text, BytesWritable, MD5Hash, ObjectWritable, GenericWritable, ArrayWritable, TwoDArrayWritable, MapWritable, SortedMapWritable. Nevertheless, Hadoop provides the flexibility to implement custom types that are needed, in some special cases, to represent the data managed by a specific algorithm.

The InputFormat is used by Hadoop to read the data used as input by the MapReduce job. The InputFormat sends the information to the mappers specifying the types of keys and values. For instance, a TextInputFormat generates keys of type LongWritable and values of type Text. If not set explicitly, the default types for the intermediate keys and values is the same as the final output types, which are LongWritable and Text respectively by default. OutputFormats are used by Hadoop to write the final information, which is usually written

by the reducers. Hadoop provides an extensive list of InputFormats and OutputFormats to be used that are suitable for most MapReduce algorithms. A listo of InputFormats and OutputFormats is shown below:

- *FileInputFormat* - Reads all files from a specific directory and sends them to the mapper task. This is the base class for all input format implementations that use files as input. Two main items are defined in the FileInputFormat: which files are included as the input for a job and how the input splits are generated.

- *TextInputFormat* - This is the default input format. Each record is a line of input, the key is the byte offset in the file, and the value is the line read. The key is of type LongWritable and the value is of type Text.

- *NLineInputFormat* - Reads N lines of the file. The key is a LongWritable which specifies the byte offset in the file of the line received and the value is the line received. Each mapper will receive N lines of the input file.

- *KeyValueTextInputFormat* - Reads a file with the following structure: *key+separator+value*. The most known formats that follow this structure are: tabs separated values (tsv) and comma separated values (csv). The key is the value on the left of the separator, and the value is in the right side of the separator. The default separator is the tab and can be changed by setting the configuration parameter *key.value.separator.in.input.line*.

- *SequenceFileInputFormat* - This type of input format it applied to read binary information stored as key-value pairs. The sequence file in Hadoop is created with a defined key type and value type at the moment of its creation. The keys and values received by the mappers when using the SequenceFileInputFormat must be the types specified when the sequence file was created.

- *SequenceFileAsTextInputFormat* - This type of input format is a variation of the *SequenceFileInputFormat* that converts the keys and values of the sequence file (which are stored as binary information) to strings or *Text* objects.

- *SequenceFileAsBinaryInputFormat* - This type of input format is a variation of the *SequenceFileInputFormat* that converts the keys and values of the sequence file (which are stored as binary information) to binary objects or BytesWritable objects. This input format is used when the binary information has a specific custom type and the application knows how to interpret the underlying byte array.

- *TextOutputFormat* This is the default output format. This output format just writes records as lines of texts. The keys and values can be of any type and are separated by a separator (by default, a tab character). The separator of the TextOutputFormat can be changed by settings the configuration parameter *mapred.textoutputformat.separator*.

- *NullOutputFormat* This output format does not emit an output and suppress the corresponding output.

- *SequenceFileOutputFormat* - This output format writes sequence files based on the key-value pairs emitted by the MapReduce job. Usually, this output format is used when the output of a MapReduce task is used as input for another MapReduce task. This format is compact and compressed; it reduces the time needed to write and read the information, this providing an efficient method for data communication between tasks.
- *SequenceFileAsBinaryOutputFormat* - This output format writes keys and values in raw binary format.

The InputFormats and OutputFormats provided by Apache Hadoop are enough for most MapReduce applications. However, in some cases new InputFormats/OutputFormats are needed. Hadoop provides the flexibility to implement custom input/output formats. For example, consider the csv file in the Example 5, where the first field is a name and the next three values are coordinates in a map. In the example, a custom input format could be implemented that reads the file and outputs as the name as the key and a coordinate as the value.

$$\begin{aligned} ball, 3.5, 12.11, 11.3 \\ tree, 2.2, 54.3, 98.8 \\ rice, 1.1, 0.9, 5.4 \end{aligned} \tag{5}$$

4.2.6 Hadoop MapReduce Counters

Hadoop counters are a useful tool for gathering statistical information about a MapReduce job. Hadoop counters are used as diagnosis information if the MapReduce algorithm is not working, or to improve its performance. Instead of adding a log to a MapReduce program, it is better to add a counter which is easier to read and manipulate. Hadoop counters are global as the Hadoop aggregates them across all mappers and reducers to produce a total at the end of the job.

The default counters are grouped by the type of information they provide, defining three families: *MapReduce framework counters*, *filesystem counters*, *job counters*. *MapReduce framework counters* gather information about the data processed by the MapReduce job, like the number of records received by the mappers, the number of records received by combiners, the number of records skipped, the number of records received by the reducers, etc. The *filesystem counters* gather information about the filesystem such as the number of bytes read and written. The *job counters* gather information about the job execution, like the number of mappers launched, the number of failed reduce tasks, the number of data-local map tasks, etc.

Hadoop also allows the user to define custom counters, defined by a java enum. A job may define an arbitrary number of enums, each with an arbitrary number of fields. The group name of the counter is the name of the enum and the fields of the enum are the counter names.

4.2.7 Distributed Cache

The Hadoop Distributed Cache is a mechanism to distribute datasets in all nodes in a Hadoop cluster. For some MapReduce jobs, specific files (e.g: configuration files) are needed in all the nodes where map tasks and reduce tasks are executed. The Distributed Cache provides a service to ensure that these files are copied to every node and are accessible when map tasks and reduce tasks are executed.

The files to be copied by the distributed cache can be stored on any Hadoop readable filesystem. When a job is launched, Hadoop copies the files specified by the *-files* and *-archives* options to the filesystem of the JobTracker, which is usually the hadoop distributed filesystem. Then, before starting the task, the TaskTracker copies the files from the filesystem of the JobTracker to a local disk, so the task can access the files efficiently.

The TaskTracker maintains a reference count for the number of tasks using each file in the cache. Before a specific task starts execution, the reference count of the file added to the distributed cache, used by the MapReduce job, is incremented by one. Then, after the task finishes execution, the count is decreased by one. Only when the count reaches zero, the file is eligible for deletion from the distributed cache, because no tasks are using it. Files are deleted to make room for new files when the cache exceeds a configurable size (by default, 10GB).

The design of the distributed cache does not guarantee that subsequent tasks from the same job running on the same TaskTracker will find the file in the cache, but it is very likely that they will. Tasks from a MapReduce job are normally scheduled to run at the same time, so usually there are not enough other jobs executed at the same time to cause the original file to be deleted from the cache. For CPU intensive algorithms, such as the approach for large-scale optimization presented in this work, the file saved in the distributed cache is never deleted because only one job is executed at a time and the files saved in the distributed cache are not big enough to be deleted (not exceeding the maximum default value of 10GB).

4.3 Hadoop Distributed File System

Distributed filesystems are filesystems that manage storage across a network of computers. Usually, when a dataset outgrows the storage capacity of a single physical computer is necessary to perform a distribution of the information across many different computers.

The distributed file systems have the same challenges than designing distributed algorithms, including failure tolerance, synchronization and communication between nodes, data replication, etc. The Hadoop Distributed File System (HDFS) was originally designed to run MapReduce jobs in a distributed manner. The HDFS provides a filesystem for storing very large files with streaming data access patterns running on clusters of commodity hardware.

HDFS has the following characteristics:

- *Large Files*: HDFS was designed to store files of hundreds of gigabytes, terabytes or petabytes of data, used for storing and analyzing big datasets.
- *Streaming data access*: The streaming data access implies that instead of reading data as packets or chunks, data is read continuously with a

constant bitrate. The application starts reading data from the start of a file and keeps on reading it in a sequential manner without random seeks. The streaming capability enables to read a file while is still being written to HDFS. Usually the type of information saved to HDFS are logs or information that needs to be analyzed afterwards, but not updated.

- *Commodity hardware*: HDFS was designed to run on clusters of commodity computers to run MapReduce jobs. The chance of failure of a node in a large cluster of commodity computers is high. Thus, HDFS was designed to be fault tolerant without noticeable impact on the MapReduce job performance.

HDFS is suitable for solving large scale problems as it was designed to be used to run MapReduce jobs. Nevertheless, it does not work well in applications that require low-latency data access, work with many small files or there are many readers and writers. The following characteristics enumerate the scenarios where HDFS is usually not the best choice to be used to save data.

- *Low-latency data access*; HDFS is designed and optimized to read huge amounts of distributed information in expense of latency. Applications that do require to read the information fast, such as real time applications, not work well using HDFS.

  Regarding low-latency data access, Hbase [34] is a better choice than HDFS as it is optimized to read the information fast. Hbase was the choice selected for the algorithm presented in this work and will be further explained in Section 4.5.

- *Many small files*: HDFS keeps the filesystem information in system memory for a more performant access. Thus, the limit to the number of files in a filesystem depends on the amount of system memory in the master node keeping the filesystem information. Each file takes about 150 bytes thus, a million files would take at least 300MB of system memory just to hold the filesystem information. Although HDFS can handle problems with millions of files, the number of files that HDFS can handle efficiently depends on the capability of the hardware.

- *Multiple writers and file modifications*: HDFS does not provide support for multiple writers or modifications at an arbitrary offset in a file.

4.3.1 Scheduling and Synchronization

The HDFS cluster works using a master-slave communication paradigm. Two types of nodes are used: a namenode (the master) and a number of datanodes (the workers). The namenode manages the filesystem namespace, maintaining the filesystem tree and the metadata for all the files and directories in the tree. This information is persistently stored on the local disk. The namenode knows on which datanode all the blocks for a given file are located. However, the namenode does not store block locations persistently, since this information is reconstructed from datanodes when the system starts.

Datanodes store and retrieve blocks when they are requested by clients. The datanodes report to the namenode periodically the lists of blocks that they are

storing. Without the namenode, the filesystem cannot be used. If the computer running the namenode fails, all the files on the filesystem are lost, because there is no way of knowing how to reconstruct the files from the blocks on the datanodes.

Hadoop provides two mechanisms, which are described next, for failure tolerance for the namenode: the back up mechanism and the secondary namenode mechanism.

The first mechanism to provide fault tolerance to the namenode is to back up the files that make up the persistent state of the filesystem metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. The second mechanism is the traditional *primary-backup* schema, where a secondary namenode runs not acting as a real namenode. Instead, the main role of the secondary namenode is to periodically save the namespace state. Usually, the secondary namenode runs on a separate physical machine, because it has large CPU demands and requires as much memory as the namenode to save the namenode state. The secondary namenode keeps the required information to be used in case of a failure of the primary namenode. In the case of total failure of the primary namenode, the secondary namenode changes it state as active and starts running as the new primary namenode.

Files in HDFS are divided into block-sized pieces, which are stored as independent units. Having a block abstraction for a distributed filesystem brings several benefits. A file can be larger than any single disk in the network. HDFS can store the blocks from a file in different disks in the cluster. Using this approach, it would be possible to store a single file of the size of the sum of all the disks in the HDFS cluster.

4.4 Cluster configuration

In order to run MapReduce jobs on Hadoop, the cluster must first be configured by setting some configuration parameters to optimize the execution of MapReduce jobs, which will be described next. There are three possible cluster configurations: *non-distributed*, *pseudo-distributed* and *distributed*. By default, Hadoop is configured in a non-distributed setup and it runs as a single Java process. This setup is useful for debugging and testing MapReduce jobs, but not for executing real applications.

A minimal configuration is needed to setup a Hadoop cluster by setting some configuration parameters. Hadoop configurations are located in *etc/hadoop* inside the Hadoop installation directory. Hadoop has three main configuration files: *core-site.xml*, *hdfs-site.xml* and *mapred-site.xml*.

To run MapReduce, a JobTracker node needs to be designated, which on small clusters may be the same computer running the namenode. The configuration parameter *mapred.job.tracker* is used to set the location where the JobTracker will listen, setting the hostname or IP address and port of the JobTracker. During the execution of a MapReduce job, intermediate data and working files are written to temporary local files. This local temporary storage is configured by setting the *mapred.local.dir* property. Since the written data potentially includes a very large output of map tasks, using a disk with partitions that are large enough is usually advisable. The *mapred.local.dir* property takes a comma-separated list of directory names. The default filesystem is needed in a default cluster configuration configured by setting the *fs.default.name* param-

eter. The filesystem normally used in Hadoop is HDFS. Hadoop replicates the information to different nodes to provide failure tolerance and this value can be changed by setting the *dfs.replication* configuration parameter (by default, set to 3). The runtime framework used to run MapReduce jobs is changed by setting the *mapreduce.framework.name* configuration parameter, having *local*, *classic* or *yarn* as possible values. The *local* configuration is to run local MapReduce jobs, the *classic* is to run MapReduce jobs using the first version of the Hadoop MapReduce framework and the *yarn* configuration is to use the second version of the Hadoop MapReduce framework

The pseudo-distributed setup configures Hadoop to run each Hadoop daemon (e.g.: JobTracker, TaskTrackers, Datanode, Namenode) on its own Java process, but on the same node. A configuration example of a pseudo-distributed setup is shown in Listing 3, where the *core-site.xml*, *hdfs-site.xml*, *mapred-site.xml* and *yarn-site.xml* configuration files are set. The example shows a default configuration setting the *fs.default.name* parameter to specify the filesystem used (in the example, HDFS), the replication value to one. In the example, the JobTracker is also configured by setting the parameters *mapred.job.tracker* configuration parameter and the second version of the Hadoop MapReduce framework is used, setting the *mapreduce.framework.name* parameter to yarn. Also, in order to run in a cluster, Hadoop must have ssh access to all the configured nodes. In the pseudo-distributed setup, the only node available is localhost.

```
hadoop-2.x.x/etc/hadoop/core-site.xml

<configuration>
   <property>
         <name>fs.default.name</name>
          <value>hdfs://localhost:9000</value>
   </property>
 </configuration>


hadoop-2.x.x/etc/hadoop/hdfs-site.xml

<configuration>
  <property>
          <name>dfs.replication</name>
           <value>1</value>
  </property>
</configuration>

hadoop-2.x.x/etc/hadoop/mapred-site.xml

<configuration>
  <property>
      <name>mapred.job.tracker</name>
      <value>localhost:9001</value>
  </property>
   <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

Listing 3: Pseudo-distributed Hadoop configuration

The cluster configuration of Hadoop depends on the hardware available in the cluster and the nodes available. How to configure a Hadoop cluster is available online at the Hadoop Apache website (`https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/ClusterSetup.html`).

The *mapred.tasktracker.map.tasks.maximum* and *mapred.tasktracker.reduce.tasks.maximum* properties allow to set the maximum number of mappers and reducers that a TaskTracker will be able to execute. The TaskTracker launches a new JVM process every time a mapper task or reducer task is executed if no child JVM process was already launched. The *mapred.child.java.opts* property allows to set the maximum available memory for the JVM process.

Hadoop uses by default a buffer size of 4 KB for I/O operations. The *io.file.buffer.size* property, available in the *core-site.xml* configuration file, allows to set the buffer size to use. Usually, this parameter is modified increasing the buffer size, having performance benefits using a larger buffer size such as 64 KB or 128 KB. The HDFS block size (by default, 64 MB) is set by the *dfs.block.size* property, in the *hdfs-site.xml* configuration file, being 128 MB or 256 MB common choices. When increasing the HDFS block size gives more data to process to each mapper task and the namenode has less blocks to track, reducing the memory consumption.

4.5 Hbase

The Hadoop database (Hbase) is a column-oriented database, based on Google BigTable [11], built on top of HDFS. Hbase is a real-time database which provides read/write random-access to very large distributed datasets. Hbase does not compromise performance when working with distributed datasets. Hbase is designed to scale linearly by adding nodes to the cluster and is able to host very large sparsely populated tables on a cluster of commodity hardware.

In Hbase, tables are composed by rows and columns, and the intersection of rows and columns is named a cell. The cells in Hbase are versioned and, by default, the version is a timestamp auto-assigned by Hbase at the time of cell insertion. In Hbase, the keys of a row are byte arrays, so any standard value can serve as a row key, from strings to serialized data structures. The table rows are sorted by row key, and all table accesses are via the primary key of the table.

The columns in a row are grouped into column families. All column family members have a common prefix; for example, the columns *literals:invalid* and *literals:valid* are both members of the *literals* column family. The column family of a table must be specified in the schema definition for the table, but new column family members can be added on demand. For example, a new column *country:city* can be offered by a client as part of an update, as long as the column family *country* already exists on the targeted table. Physically, all column family members are stored contiguos on the filesystem.

The main differences between Hbase and relational databases (RMDB) tables is that tables in Hbase have versioned cells, rows are sorted, and columns can be added dynamically by the client without requiring that they are predefined in the database schema.

The tables in Hbase are automatically partitioned horizontally into regions. Regions are the units that get distributed over an Hbase cluster. In this way,

a table that is too big to be stored on one server can be stored in a cluster of computers, with each node in the cluster hosting a subset of the regions of a table. Each region comprises a subset of rows of a table. A region is denoted by the table it belongs to: its first row, inclusive, and the last row, exclusive. Initially, a table comprises a single region, but as the size of the table grows and after the region crosses a configurable size threshold, the region is split into two new regions of approximately equal size. Until this first split happens, all loading is performed by the single server hosting the original region. As the table grows, the number of regions of the table grows.

Hbase uses a master-slave communication paradigm to distribute the Hbase tables and to maintain the regions of the tables. Hbase is composed by a master node orchestrating a cluster of one or more region server slaves. Hbase uses Zookeeper to coordinate the communication between the master node and the region servers. The basic schema of communication is shown in Figure 15 with the master node and three region servers communicating using Zookeeper cluster for coordination and HDFS to save data. The Hbase master is in charge of initializing the Hbase cluster, for assigning regions to registered region servers and for providing failure tolerance in case of region server failures. The master node is lightly loaded to provide the best possible performance without having to store data and to maintain region servers. The region servers have zero or more regions and they respond to read/write requests from the clients. The region servers also manage region splits created for a table when the table size grows, informing the Hbase master node about new child regions to manage.

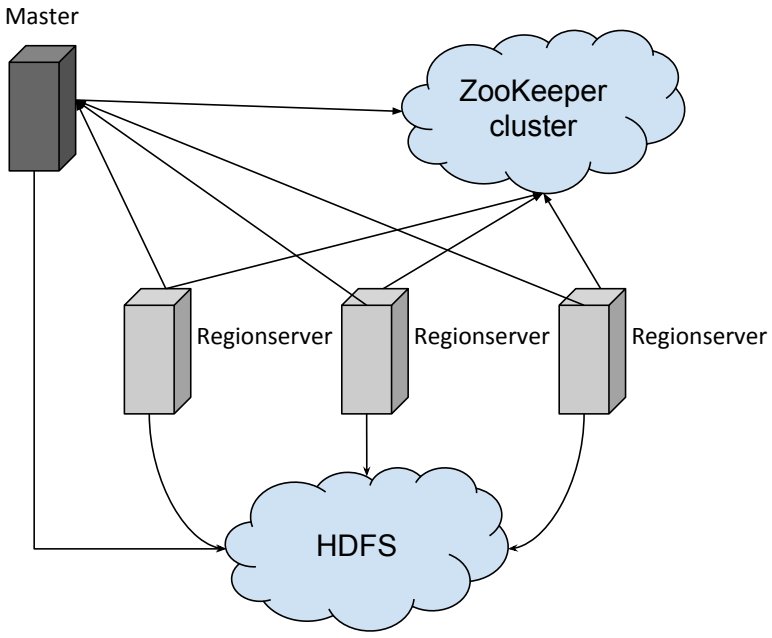

Figure 15: Hbase Distributed Architecture

4.5.1 Hbase Versus RDBMS

Hbase and other column-oriented databases are often compared to traditional relational databases management systems (RDBMS) [**?**]. Hbase and RDBMS significantly differ in their implementations and in the problems they were designed to solve. Nevertheless, despite their differences, they are potential solutions to the same problems.

On one hand, Hbase is a distributed, column-oriented data storage system that provides random reads and writes on top of HDFS. Hbase is designed from the start to handle huge amounts of distributed information to scale to billions of rows and millions of columns. Hbase has automatic partitioning, as the information grows and the table grows, the tables will be automatically split into regions and distributed in the Hadoop cluster [34]. The schemas of the tables in Hbase mirror the physical storage creating a system for efficient data structure serialization and retrieval. Hbase does not have a query language as expressive as the SQL query engine and the developer must use the storage and retrieval procedures in an appropriate way. The insert performance in Hbase is independent of the table size as rows are stored sequentially as well as the columns within each row. Hbase scales linearly when adding new nodes. Adding new nodes to the Hbase cluster has no performance penalty and the regions are automatically rebalanced and spread evenly. Hbase is designed to be fault tolerant to run on clusters of commodity hardware, it transparently address individual nodes downtimes by having data replication.

On the other hand, RDBMS are row-oriented and they have ACID (Atomicity, Consistency, Isolation, Durability) properties. RDMBS have fixed schemas which cannot be changed dynamically. In relational databases, the emphasis is on strong consistency, referential integrity, abstraction from the physical layer, and complex queries through the SQL language. The main issue with relational databases is when manipulating big volumes of information and when multiple clients read and write at the same time. Scaling a relational database usually involves loosening ACID restrictions, removing indexes and foreign keys and properties that made relational databases a good choice for small to medium applications. When scaling up a relational database and removing the properties mentioned before, the RDBMS behaves like Hbase in how information is stored, distributed and queried. Usually, for big datasets, all important information is denormalized, no integrity is checked and secondary indexes are inefficient thus all queries become primary key lookups.

The Table 2 summarizes the differences between Hbase and RDBMS presented. Hbase is used in this work as a centralized database for saving found solutions and invalid assignments, further explained in Section 6. For the problem solved in this work that uses the centralized database for saving small amounts of information requiring fast read/write access, both RDBMS and Hbase are suitable solutions. The motivation for using Hbase instead of a traditional RDBMS is because is tightly integrated with HDFS and MapReduce and it is easier and faster to integrate.

RDBMS	Hbase
Focus on ACID properties	Focus on performance and scalability for big volumes of unstructured information
Fixed database schema	Dynamic database schema
Fixed partitioning and needs to be administered by the user	Automatic and dynamic partitioning of tables
Row oriented	Column oriented
Performance of the queries depends on the size of tables	Performance of the queries is independent of the size of the tables
Uses the default filesystem	Uses HDFS filesystem and stores the information mirroring the underlying filesystem for performant access
Supports an expressive query language (SQL)	Does not support a query language and the storage and retrieval has to be performed in the appropriate way

Table 2: Summary of RMDB and Hbase comparison

4.6 Summary

This section described the Apache Hadoop MapReduce implementation and the MapReduce job execution on Hadoop. The Hadoop distributed filesystem was introduced including its main characteristics. The cluster configuration was introduced including the main configuration parameters that can be changed to improve performance of Hadoop job execution. Finally, the Hadoop Database was introduced and the main differences between Hbase and RDBMS were presented.

5 Related Work

This section presents a review of the state of the art of 3-SAT solvers. First, the state-of-the-art exact solvers are presented, which use different approaches to solve the 3-SAT problem. After that, different strategies using parallel programming techniques to improve algorithm performance are described. Finally, several heuristics and stochastic searches to solve the 3-SAT are introduced.

5.1 3-SAT exact solvers

The state-of-the-art exact solver for 3-SAT is the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [21], a backtracking search algorithm introduced in 1962 to solve the problem. A schema of the DPPL algorithm is provided in Algorithm 1.

Algorithm 1 DPLL Algorithm

1: **Input**: A set of clauses θ, a set X_{input} of literals
2: **Output**: A truth value, a set X_{output} of literals
3: **if** X_{input} is satisfiable for clauses θ **then**
4: return true, X_{input};
5: **end if**
6: **if** all literals are set in X_{input} **then**
7: return false, X_{input};
8: **end if**
9: **for** every unit clause l in θ **do**
10: θ = unit-propagation(X_{input} + l, θ);
11: **end for**
12: **for** every literal l that occurs pure in θ **do**
13: θ = pure-literal-elimination (X_{input} + l, θ);
14: **end for**
15: l = choose_literal(θ)
16: **return** DPLL(θ, X_{input} + l) or DPLL(θ, X_{input} + $\neg l$)

DPPL applies a divide and conquer approach, splitting the problem into simpler sub-problems to be solved recursively. The input of the DPPL algorithm is a set of clauses θ, in CNF form, and the output is true if exists a solution to the problem (and the literal assignment that makes all clauses true) or false. The DPPL algorithm starts by checking if the input literal assignment X_{input} makes the received set of clauses true (lines 3-5). If the X_{input} is satisfiable, then the algorithm ends. After that, the algorithm simplifies the formula by applying the *unit propagation* [21] algorithm (lines 9-11). The DPPL algorithm further simplifies the formula by applying the *pure literal elimination* [21] algorithm (lines 12-14). The *unit-propagation* and *pure-literal-elimination* algorithms are used to avoid exploring naive parts of the search space. Finally, the algorithm chooses a literal to fix and recursively applies the DPPL to the received literal assignment plus the chosen literal (lines 15-16). In case that satisfiability is not fulfilled, the algorithm recursively checks using the opposite truth value for the considered literal. The recursion ends when: i) all literals are set in the input assignment and is a solution to the problem, then the formula is

unsatisfiable (lines 6-8) or ii) all variables are assigned or all clauses are satisfied, then the formula it is satisfiable. The DPLL algorithm is effective for solving small problems but inefficient when solving large problem instances, because the unsatisfiability of a given formula only is detected after performing an exhaustive search of the solution space.

The *unit-propagation* is an algorithm to simplify the 3-SAT formula by propagating a value of a literal that appears single in a clause. The unit propagation algorithm is described in Algorithm 2. The algorithm checks all clauses in the formula if there are unit clauses (e.g.: clauses that have only one literal). For every unit clause, the algorithm sets the literal l that appears single in the clause to true, or false if the literal appears negated in the clause (line 2). Then, for every other clause that contains the literal l (line 3), the clause is removed from the formula because the literal l is set to true(lines 4-6) and the clause will be always true. If the clause contains the literal l negated (lines 7-9), then the clause is simplified by removing the literal from the clause.

Algorithm 2 DPLL: Unit Propagation

```
1:  for each unit clause cl (with single literal l) do
2:    set l to true
3:    for every clause c different from cl do
4:      if c contains l then
5:        delete clause c
6:      end if
7:      if c contains -l then
8:        remove l from c
9:      end if
10:   end for
11: end for
```

After applying the unit propagation algorithm, an equivalent set of clauses is obtained. Figure 16 shows an example the application of the unit propagation algorithm.

1: Set of clauses: a v b, -a v c, -c v d, a
2: Simplify by propagating the unit clause a
3: a v b: is removed
4: -a v c: a is deleted resulting in c
5: -c v d: not changed
6: a : is removed
7: resulting set: c, -c v d

Figure 16: Example of application of the unit-propagation algorithm

The pure elimination algorithm eliminates from the formula the literals that appears only true or only false in all clauses. The algorithm is presented in Algorithm 3.

An example of application of the Pure Literal Elimination algorithm is presented in Figure 17.

Algorithm 3 DPLL: Pure literal elimination

1: **if** x only appears true in all clauses (-x does not appear) **then**
2: add x=true to the assignment
3: all clauses containing literal x are satisfied
4: **end if**
5: **if** y only appears in its negative form **then**
6: add x=false to the assignment
7: all clauses containing literal -x are satisfied
8: **end if**

1: Set of clauses: a v b v c, a v c v -d, -c v d, -b v -d
2: Simplify by setting a to true
3: a v b v c is removed
4: a v c v -d is removed
5: -c v d is not changed
6: -b v -d is not changed
7: Formula is simpified to: -c v d, -b v -d

Figure 17: Example of application of the pure-literal-elimination algorithm

Another state-of-the-art exact solver is the Conflict Analysis Clause Learning (CDCL) [83] algorithm, in which many 3-SAT solvers are based on [8, 54]. The CDCL algorithm was inspired on DPLL solvers, but can learn new clauses by applying conflict analysis algorithms and backtracking non-chronological. The CDCL algorithm also applies conflict analysis algorithms to learn new clauses from the formula. The conflict analysis algorithm generates new clauses based on the conflicting literals of clauses. If two clauses A and B have the same literals but differ only on the truth value of one of them, then a new clause C can be created with the non-conflicting literals. This algorithm is known as the resolution logic, and an example of application is shown in Figure 18.

A v B v C
A v B v -C
→ A v B

Figure 18: Example of application of the resolution logic algorithm

5.2 3-SAT Heuristics and Metaheuristics

Local search methods have long been used for solving computationally hard optimization problems [43]. Local search algorithms search for solutions in the search space of candidate solutions until an optimal solution is found or the algorithm fails to find a solution given a specific time bound. A particular type of local search methods are the stochastic local search (SLS) methods, where randomness is introduced in the search algorithm to improve the algorithm

performance [43]. Stochastic local search algorithms have been proposed to solve the 3-SAT, being the GSAT [36] and the WalkSAT [43] two of the most popular. Both techniques start by assigning a random value to each variable. If the assignment satisfies all clauses, then the solution is found. Otherwise, a variable is flipped and the procedure is repeated until all clauses are satisfied.

WalkSAT and GSAT heuristics to solve the 3-SAT differ in the selection of the variable to flip: GSAT makes the change which minimizes the number of unsatisfied clauses in the new assignment, or picks a variable at random; on the other hand, WalkSAT selects a variable (from an unsatisfied clause) that will result in the fewest previously satisfied clauses becoming unsatisfied, with some probability of picking one of the variables at random. When selecting an optimal variable, WalkSAT performs fewer calculations than GSAT (because it considers fewer possibilities). When picking at random, WalkSAT has at least a chance of $1/n$ (being n the number of variables of the problem) in the clause of fixing a currently incorrect assignment. GSAT and WalkSAT are not just "heuristics". Both algorithms may restart with a new random assignment if no solution has been found, implementing a diversification strategy to avoid getting stuck in a local search in the search space.

Regarding metaheuristics methods to solve the 3-SAT, the state-of-the-art is Schöning algorithm (or variations of it), introduced in 1999 [77]. Schöning is a randomized search algorithm that runs in $O(1.33^n)$ for n literals and succeeds with high probability to decide a 3-SAT formula. The probability of not finding a satisfying assignment using Schöning algorithm after performing $t = k \times (4/3)^n$ restarts is at most e^{-k} (i.e the error of not finding a solution after $t = 30 \times (4/3)^n$ restarts for $k = 30$ is e^{-30}). A schema of Schöning algorithm is provided in Algorithm 4. The Schöning algorithm starts by selecting a candidate solution at random (line 1). If the selected candidate solution is solution to the problem, the algorithm ends (lines 4-6). If the candidate solution is not satisfiable, a literal of the first false clause is selected at random and its truth value flipped (lines 8-10). This process is repeated $3 \times n$ times or until a solution is found. If a solution is not found after $3 \times n$ iterations, the algorithm is restarted selecting another candidate solution, performing a diversification strategy (line 1).

Algorithm 4 Schöning Algorithm

```
1:  pick an initial assignment x ∈ X uniformly at random
2:  for 3n times do
3:    if a satisfies all clauses of F then
4:      if F is satisfied then
5:        end
6:      end if
7:    else
8:      pick any unsatisfied clause C (uniformly at random)
9:      pick a literal l in C (uniformly at random)
10:     flip the value of l
11:   end if
12: end for
```

Several other metaheuristics algorithms have been proposed for 3-SAT, combining ideas of heuristics and exact solvers. The PPSZ algorithm (Paturi, Pudlak, Saks, and Zane, 1998 and 2005) [70, 71] is a randomized search algorithm

that runs in $O(1.3631^n)$ for n literals and succeeds with high probability to decide 3-SAT. A schema of the PPSZ algorithm is provided in Algorithm 5. The PPSZ algorithm first pre-process the formula, creating a new formula with more clauses increasing the probability of finding unit clauses (clauses that contain only one literal) in the formula (line 1). After that, the PPSZ algorithm forces the assignment for the literals that appear in unit clauses (lines 4-5). For literals that do not appear in unit clauses, set the literal at random (line 7). The inner loop in Algorithm 5 executes in polynomial time and has exponentially small probability of finding a satisfying assignment (lines 3-9). The randomized PPSZ algorithm with independent repetitions executes in $O(1.3633^n)$ time. The current best algorithm to solve the 3-SAT is the PPSZ-based algorithm presented by Hertli [41]. The previous article proves that the PPSZ algorithm solves every 3-SAT instance in $O(1.3071^n))$ time.

Algorithm 5 PPSZ Algorithm

```
1:  pre-process formula F            [assign literals that appear in unit clauses]
2:  for t times do
3:    for loop through variables in random order do
4:      if x or −x belongs to unit clause then
5:        force assignment           [true for x and false for −x]
6:      else
7:        guess assignment at random from {0,1}
8:      end if
9:    end for
10: end for
```

5.3 Parallel algorithms to solve the 3-SAT

This section introduces parallel programming techniques applied to solve the 3-SAT problem. First, the challenges when solving the 3-SAT in a distributed environment are introduced. After that, the different strategies used for tackling the 3-SAT in a distributed environment are presented. Finally, this section describes different algorithms that use parallel programming techniques to solve the 3-SAT.

5.3.1 Challenges

Many approaches have been proposed to solve the 3-SAT using parallel programming techniques. Some of the main challenges when designing a parallel 3-SAT solver are described by Hamadi [40]. The main aspects described by Hamadi are mentioned below.

Dynamic resource allocation: Adding more computational resources not always increases the algorithm performance. The first main challenge when designing a parallel SAT solver is to find the optimal set of computational resources to use to solve a specific problem instance, optimizing the use of resources and reducing the communication overhead. One of the main challenges is to design a dynamic resource allocation algorithm that can predict the optimal amount of resources needed to solve the problem instance efficiently.

Decomposition strategies: There are two main decomposition strategies used: *space decomposition* and *instance decomposition*. The first splits the search space in subspaces which are processed by different solvers. In the *instance decomposition*, the problem instance is decomposed such that non of the computing elements knows the whole problem instance. This kind of decomposition is particularly useful for big problem instances. The second main challenge when designing a parallel SAT solver is to design a dynamic decomposition technique that is efficiently computable and results in efficient decompositions.

Preprocessing techniques: Pre-processing techniques are commonly used in the state-of-the-art heuristics such as the DPLL or CDCL algorithms, which are proven techniques to reduce the overall execution time of the algorithm. The third challenge when designing a parallel SAT solver is to design efficient pre-processing algorithms that, with knowledge of the type of decomposition being used, simplify the problem instance such that the overall performance is increased. Moreover, for very large formulas, it may be infeasible to preprocess the whole problem instance before solving the problem. It would be worth to design parallel pre-processing algorithms to be executed before the SAT solver.

Strategies for knowledge sharing: The fourth challenge when designing a parallel SAT solver is to share learnt clauses between workers of the parallel algorithm. For big problem instances, the number of clauses to share can grow impacting the algorithm performance. One challenge when designing a parallel algorithm is to implement a dynamic limit for the number of clauses to share, sharing the clauses that provide more value to the algorithm reducing the communication cost associated with sending and receiving clauses.

Designing and implementation of new algorithms: Many algorithm are based on currently existing algorithms such as the DPLL or CDCL algorithms, such as the algorithm presented in this work. One of the main challenges when designing a parallel SAT solver is to create new algorithms and data structures.

5.3.2 Strategies

Two main strategies have been proposed in the related literature to solve very-large 3-SAT instances: *divide and conquer* (D&C) and the *portfolio* strategies.

The D&C approach divides the problem into small subproblems, successively allocating to sequential (DPLL/CDCL) SAT solvers and have been extensively used in many SAT solvers [24, 44, 91, 48, 60]. The problem can be split using one of the two decomposition strategies: the domain decomposition strategy or the instance decomposition strategy.

On the one hand, the domain decomposition strategy decomposes the search space in subspaces, assigning each solver a different subspace. The domain decomposition is usually implemented by sending each solver some fixed literals with different values, which guarantees non-overlapping search spaces for each solver. An example of a D&C approach using a domain decomposition strategy is shown in Figure 19, where the first solver receives x_1, x_2, the second solver receives $-x_1$ and the third solver receives $x_1, -x_2$. The assignment generates three non-overlapping search spaces. On the other hand, the instance decomposition strategy splits the problem instance in subproblems. This approach is

useful for big problem instances, where each solver can process a smaller part of the problem instance by, for example, processing only a reduced amount of literals.

The decomposition strategies usually do not generate balanced subproblems by its hardness. Generating balanced subproblems by its hardness before execution of the solver is a hard to solve problem [47]. Because the subproblems are usually not balanced, some solvers may end its execution before others, so cooperation between solvers is achieved applying of a load balancing strategy that dynamically transfers subspaces to idle workers. Another cooperation between solvers is achieved through the exchange of learnt clauses or invalid literal assignments.

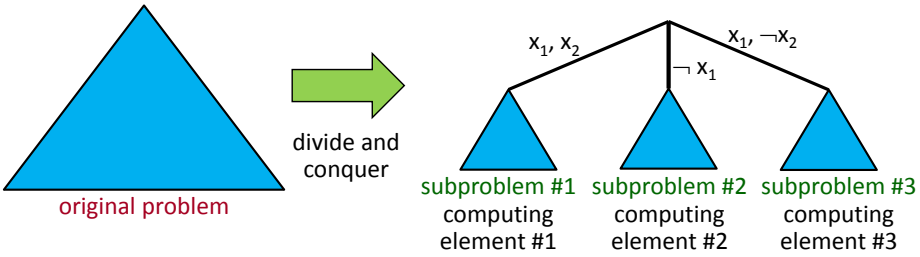

Figure 19: Divide and Conquer approach for parallel 3-SAT solvers

The portfolio approach uses several sequential SAT solvers, which compete and cooperate on solving the same formula. The sequential solvers can be completely different solvers but are usually based on the DPLL or CDCL algorithms [8, 54]. The solvers usually are differentiated between each other by the configuration parameters used: the search algorithm, the literal selection algorithm or its knowledge sharing technique. Since each solver works on the whole formula, there is no load balancing overhead. Cooperation between solvers is achieved through the exchange of learnt clauses which is usually limited to avoid communication overhead when solving big problem instances [40]. Portfolio solvers became prominent in the related literature since 2008. A basic scheme of the portfolio approach is shown in Figure 20.

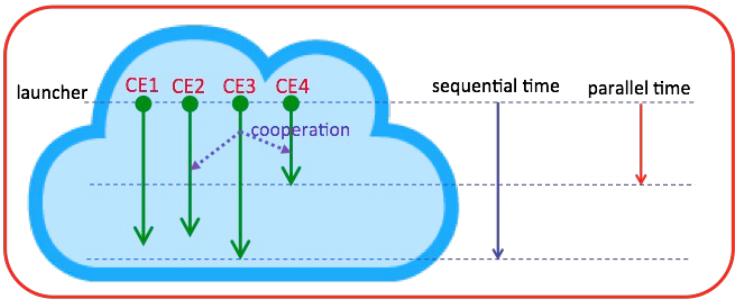

Figure 20: Portfolio approaches for parallel 3-SAT solvers

The ManySAT algorithm (2009) [39] is a portfolio-based algorithm to solve

the 3-SAT using DPLL based solvers. The ManySAT solver runs different versions of the DPLL algorithm, having each different restart policies, different seeds and different heuristics for literal selection. The ManySAT algorithm learns and shares invalid clauses and assignments using an extension of an implication graph [4]. The algorithm generates four variants of the DPLL algorithm based on variation of five parameters: i) the restart policy; ii) the heuristic for literal assignments; iii) the heuristic for setting the truth value for literals; iv) the learning technique and v) the clause sharing technique. The ManySAT parallel algorithm was the winner of the 2008 SAT-Race contest (`http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/index.html`).

Another portfolio-based parallel SAT solver is the Plingeling (2010) [8] algorithm. The Plingeling algorithm uses a master-slave paradigm of communication, where the worker threads run the same SAT solver with different configurations. The master process sends to the slaves information to process and unit clauses learnt from other slaves. The slaves communicate only with the master process sending and receiving unit clauses. The worker solvers differ between each other in the seed selected, the pre-processing algorithm used and in the heuristic used for variable selection.

A similar approach is taken in the SArTagnan (2010) [54] algorithm, which runs different algorithms and different search strategies using a multithread algorithm. The algorithm shares clauses but just specific ones as the algorithms running in the different threads are different and some clauses may not be valid in other algorithms. Most threads use CDCL with the VSIDS heuristic [66] for variables [27], Luby restarts [58] and phase-saving [72]. However, other threads use different restart policies and different configuration parameters for the VSID heuristic.

Other parallel algorithms to solve the 3-SAT use incomplete/local search solvers, such as our parallel spatial quantum algorithm [60]. The proposed algorithm combines a parallel Schöning algorithm applied to a subspace of the search space performing a local search and a quantum search in the search space. Another example is the GPU4SAT algorithm [25], that considers the number of satisfied clauses as a quality function, and uses a local search to evaluate neighborhoods, looking for the best literal to flip.

5.3.3 Parallel 3-SAT Solvers in the cloud

When proposing a parallel algorithm to solve the 3-SAT on a grid or cloud system, there are some specific issues related to the main characteristics of the infrastructure to take into account. Regarding cooperation, the communication to/from running processes is extremely time-expensive in a distributed computing infrastructure. In addition, the distributed platform may impose resource limitations, e.g. a predefined time limit for the solver execution.

In order to deal with the cooperation issue of a distributed infrastructure, maintaining a *master database* of learnt clauses has been proposed [46]. The database is used for clause sharing only when a solver *starts*, then the solver imports a part D of the database permanently into the solver instance, i.e. it solves (φ & D) instead of φ, and when a solver *timeout* is detected, then (a subset of) the current learnt clauses is merged into the database (and simplify with unit propagation, etc.). This approach is known as *cumulative parallel learning with hard restarting solvers* (see Fig. 21).

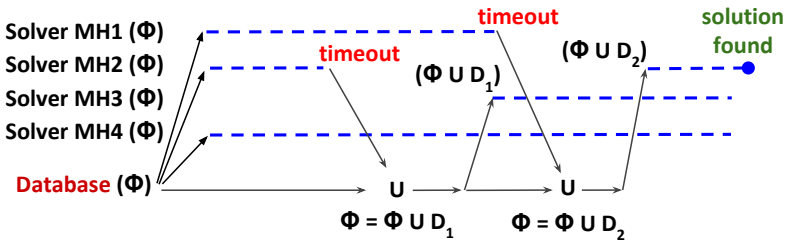

Figure 21: Problem Split in P_ij with collaboration using clause database.

Different architectures have been used to implement parallel 3-SAT solvers. The most common architecture for distributed computers are grids or cluster of heterogeneous computers, but hardware acceleration and graphical processing units have also been used to solve the 3-SAT in a distributed manner. Hardware accelerated techniques have long been in related literature to improve algorithm performance [75]. One of the most used hardware techniques used is the Field Programable Gate Arrays (FPGA) [87].

FPGA have been used to solve the SAT problem by exploiting parallelism in hardware [20], improving the performance of the unit propagation algorithm by hardware. The proposed design uses a D&C approach to split the search space in subproblems. The lookup and update of clauses is improved by hardware by saving the instance information in Block RAM (BRAM), which is a dedicated two-port memory containing several kilobits of RAM [75]. Moreover, the presented implementation supports dynamic insertion and deletion of learnt clauses. Usually, the hardware accelerated SAT solvers are based on variations of the DPLL algorithm [84] and are hybrid implementations of hardware and software.

Grids of commodity computers have long been used for solving the 3-SAT problem [78, 91, 94, 9, 73]. The GridSAT algorithm [94] creates a distributed implementation based on a variation of the DPLL algorithm using a conflict clause algorithm and clause sharing, named Chaff [67]. The GridSAT is focused on solving the 3SAT on a grid of heterogeneous commodity computers, optimizing the use of resources. The proposed algorithm, implements a scheduling technique that request resources on demand as needed so, for small problem instances, it runs a sequentially on one node. The GridSAT uses a D&C approach sending different search spaces for each solver and its performance has a speedup up to eight compared to the Chaff algorithm. Moreover, the GridSAT algorithm is able so solve problem instances that the Chaff algorithm fails to find.

The PaMiraXT [78] also uses a domain decomposition strategy using MPI to implement a parallel SAT solver. Another 3-SAT solvers use openMP and MPI to implement a distributed solver in the cloud [89]. Apart from the PaMiraXT algorithm that is designed to execute in cluster, an implementation to be executed on Desktop Grids [9, 73], where the nodes are volatile and heterogeneous in hardware and software. The Desktop Grid parallel SAT solver focuses on the scheduling and failure tolerance for a sudden leave of a node. In this context, this work presents a novel approach by using Hadoop as a distributed infrastructure, seemingly handling failure tolerance, resource managing

and communications between nodes.

The HordeSat [6] algorithm is a portfolio based algorithm in the cloud, executed in a cluster of commodity computers. The HordeSat algorithm uses a master-slave communication paradigm implemented using MPI [28]. Each solver runs a different algorithm in parallel and communicate to the master process to share learnt clauses. The presented results show a significant speedup for hard instances running on a cluster with up to 2048 cores. The HordeSat algorithm implements a clauses sharing algorithm and diversification strategy from the Plingeling algorithm [8]. Quantum mechanics has also been used to solve the SAT problem [60], implementing its simulation using parallel programming techniques executing in a cluster of computers. The presented approach uses a D&C approach using an instance decomposition strategy. The algorithm is an hybrid implementation with shared and distributed memory, using openMP and MPI.

Graphic processor have also been used to solve the 3-SAT problem in parallel [63], taking advantage of the parallel processing power of graphic cards using CUDA. The presented implementation uses a D&C approach, splitting the search space to run hundreds of thousands of lightweight threads in parallel. The main limitation of the presented work is the available on-chip memory, so the hardware does not provide enough memory to save learnt clauses and to share information between threads. Also, the hardware limitation does not allow to solve big formulas with thousands of clauses because of the limited memory resources.

5.4 Summary

This section described the related work, different algorithm approaches to solve the 3-SAT. Different heuristic and metaheuristics were presented as well as approaches to solve the 3-SAT in a distributed environment, using parallel programming techniques. Hadoop and MapReduce have never been used to solve the 3-SAT in a distributed environment. The approach presented in this thesis could benefit the development of new parallel algorithms to solve the 3-SAT without having to worry about the underlying problems of distributed algorithms. The algorithms designed to run on Hadoop can easily be executed on thousands of commodity computers, scaling horizontally by adding new nodes.

6 The MapReduce 3-SAT Solver

This section introduces the devised algorithms to solve the 3-SAT using the MapReduce programming model. First, design considerations for solving the 3-SAT algorithm using the MapReduce paradigm are described as well as the problem codification. After, different variations and optimizations of the introduced MapReduce algorithm for the 3-SAT are presented. Experimental analysis of the different variations of the algorithm are introduced in this section.

In this work, a candidate solution or a partial assignment is a subset of literals of the problem instance with some specific truth value assignment, which may or may not have all the literals fixed of the formula. A candidate solution is said to be valid if it does not make any clause false and it is invalid (or false), if it makes at least one clause false in the formula. In this context, a solution to the 3-SAT problem is valid candidate solution which has all the literals of the formula fixed.

6.1 3-SAT solver in Hadoop

The devised algorithm was created using an incremental approach, starting from a simple solver to a more complex one, solving in each step the problems faced when solving a LSO problem using the MapReduce programming paradigm. The purpose of this Master Thesis is to explore the capabilities of Hadoop and the MapReduce programming technique to solve large optimization problems. The main approaches taken to devise the 3-SAT MapReduce solvers are based on the portfolio with/without cooperation approach and divide & conquer approach.

For designing a 3-SAT algorithm using the portfolio approach, a master database approach (i.e using Hadoop Distributed File System or HBase) is needed for cooperation. Moreover, depending on the problem size, mostly one million literals could be allowed in a grid solver [48] and an empirical analysis of both the store criteria and number of solvers is needed.

The Divide & Conquer is a very promising approach for solving "very hard" 3-SAT instances. The strategy proposes dividing the workload among many nodes so, the partitioning of the problem is crucial. Obtaining good partitioning functions for the 3-SAT is usually challenging [48]. Even more, taking into account that the distributed algorithm will run following a MapReduce schema and many algorithms cannot be solved easily using MapReduce.

Algorithm using the MapReduce programming model are split among different mappers in different nodes based on the data to be processed. In this way, the problem split is easy, it is based on data. This work presents the problem of solving a computer intensive problem using Hadoop and MapReduce where the data is almost none, so the problem must be split differently from the usual approach taken when using Hadoop and MapReduce.

A domain decomposition technique was used to split the problem with a Master-Slave approach. The main MapReduce job is the master process, which splits the domain in subproblems, which are processed by mapper processes and reduce processes. The domain is decomposed by fixing d literals, so each mapper searches for solutions in different (exclusive) search spaces. In this case, each mapper evaluates candidate solutions in different spaces and two mappers

cannot find the same solution to the problem. For this to happen, all mappers (or reducers) must select the literals to fix in the same manner (and order).

In order to split the domain among different mappers, a file is written to HDFS specifying a subproblem in each line. The problem is divided by fixing d literals. For example, let the formula with 3 clauses defined in Eq. 6(a), the file written to HDFS with the problem split, using x_1, x_2 as literals to fix, is shown in Eq. 6(b).

$$\begin{array}{c} x_1 \vee x_2 \vee x_6 \\ \neg x_1 \vee x_4 \vee x_3 \\ \neg x_3 \vee x_1 \vee x_7 \end{array} \qquad \begin{array}{rr} 1 & 2 \\ -1 & 2 \\ 1 & -2 \\ -1 & -2 \end{array} \tag{6}$$

(a) 3-SAT Problem example (b) File written to HDFS

With the problem decomposition used, each mapper receives one line describing a subproblem of the 3-SAT problem. Different alternatives were taken for selecting the literals to fix. The first approach was to fix by order, fixing literal 1 first, literal 2 as the second and so on. Another approach taken was to fix literals based on frequency of appearance in the formula, selecting the literals that appear more frequently in the clauses first. For example, if literal 2 is the literal that appears in more clauses, then it will be fixed first, the second literal that appears more frequently in the formula is selected second and so on. An example of literal selection for problem decomposition is shown in Eq. 7 where the literals that appear more frequently in the formula are 1, 3, 2, 4, 6, 7, 5 with a frequency of 4, 4, 3, 2, 2, 2, 1 respectively.

$$\begin{array}{c} x_1 \vee x_2 \vee x_6 \\ \neg x_1 \vee x_4 \vee x_3 \\ \neg x_3 \vee x_1 \vee x_7 \\ \neg x_4 \vee \neg x_2 \vee x_6 \\ \neg x_7 \vee x_3 \vee x_5 \\ x_2 \vee x_3 \vee \neg x_1 \end{array} \tag{7}$$

3-SAT formula example

The main MapReduce job, described in Figure 22, receives as input a HDFS file which specifies the formula in CNF form. The initialization process consists of uploading the CNF formula received to to the distributed cache (described in Section 4.2.7) so its available to all mappers and reducers in different servers. The domain decomposition is also performed in the initialization process by setting r number of literals. The number of splits is controlled in the first devised algorithms by an input parameter and then, is optimized (and calculated in the MapReduce job) so memory consumption and performance is optimized for the algorithm.

After the initialization process, the MapReduce job is launched using as input format the *NLineInputFormat* with *N=1* so that, in the example shown, four mappers are created receiving one subproblem definition each. Depending on the devised algorithm, different search algorithms are used in the mapper to search for candidate solutions which will be described further below. Finally, the reduce phase receives the candidate solutions found by the map phase and checks if a solution is found. If a solution was found, it is saved to HDFS to a file with name *solution_{problem_instance_name}* (or to HBase in the randomized approach described in Section 6.7) and increments in one the solutions counter. If a solution was not found, valid candidate solutions received are emitted as

key in the output, which will be used as input for a new MapReduce job created afterwards.

The main MapReduce job, checks if a solution was found after each iteration checking the solutions counter. If a solution was found, then the algorithm ends. Otherwise, a new MapReduce job is created using the previous iteration output as input and a new folder is created in HDFS to save the next iteration output.

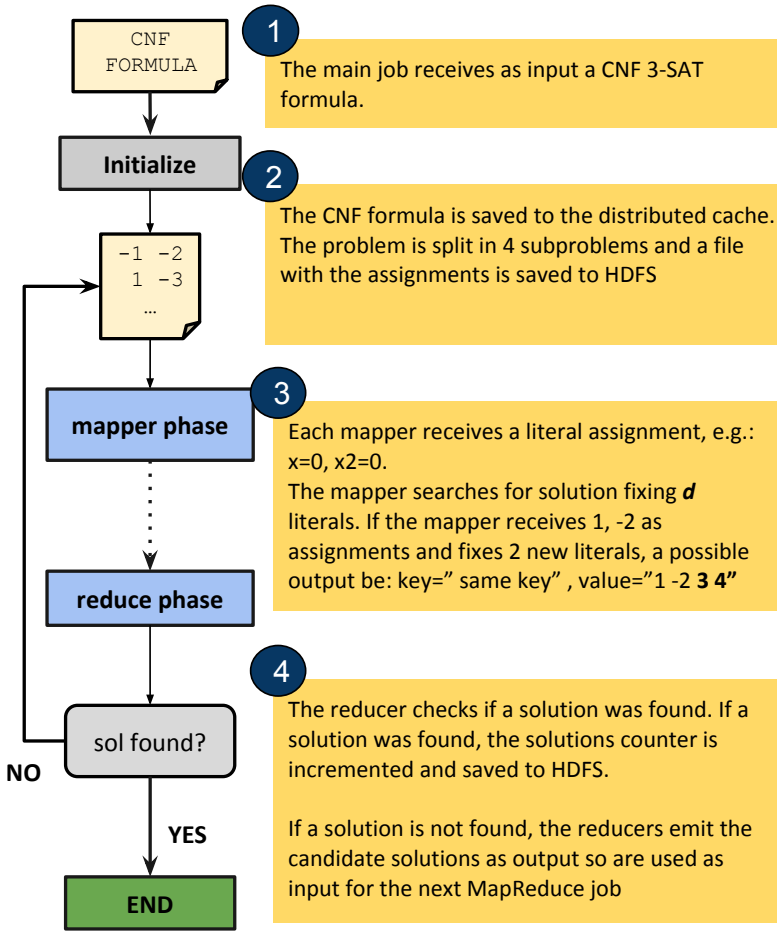

Figure 22: 3SAT MapReduce general structure
(1) The main job receives 3-SAT formula as a file in HDFS in CNF form and saves it to the distributed cache so it is available to all mappers; (2)The first devised algorithms split the problem in a fixed number of subspaces (based on an input parameter) but this approach is later changed to a more optimal value, based on the number of mappers; (3) each mapper receives a candidate solution (fixed literals). The mapper emits as key the same key received so that the number of reducers is the same as the number of mappers (unless a mapper fails to find valid candidate solutions, in that case no candidate solution is emitted and that mapper does not emit nothing); (4) the reducers in the first devised algorithms are dummy and only prepare the data for the next iteration of the algorithm if no solution was found. This is not performant as Hadoop anyway creates JVM for such executions and shuffles and groups all mapper output. An approach taken later is to exploit the processing power of the reduce phase. Another possible approach could be to just eliminate the reducer phase of the algorithm.

Hadoop framework has two special functions for MapReduce, in the map and reduce phase where initialization and cleanup can be performed. The setup and cleanup method are executed once by each mapper and reducer process at the start and at the end of the process respectively. The setup method is used in different proposed algorithms to retrieve the formula from HDFS (or from HBase) or to retrieve solutions found in the collaborative approaches presented. The cleanup method is not used in the presented algorithms.

In each iteration, d literals are fixed, generating 2^d possible candidate solutions. If the number of mappers is not limited, the algorithm could generate 2^d mappers (using a NLineInputFormat with one line per mapper), resulting in OutOfMemory errors (because of too many mappers are created, each one demanding around 500 kb). Moreover, creating one mapper for each possible assignment is not efficient, because Hadoop creates a new Java Virtual Machine process for each task, producing a significant overhead in both CPU and memory.

The main MapReduce job, decomposes the problem using a constructive search approach (by fixing r initial literals) and a DFS algorithm is then used inside the Mappers. A better approach for a deterministic 3-SAT solver would be a Depth First Search approach (DFS) to split the problem among Mappers. Although MapReduce has been proposed to solve large scale graphs [16], it is inappropriate for DFS algorithms as the mappers cannot communicate between each other and a MapReduce job has a specific structure that must be followed. The mappers cannot communicate between each other and output a list of (key,value) pairs, which will be the received, grouped by key, by the reducers.

When a solution is found in a mapper, the standard MapReduce algorithm cannot finish until all mappers finish processing their data. This is a big inconvenient because, for large problems, it could mean the unnecessary processing of thousands of possibilities (after a solution is found).

The main performance challenge when solving a LSO problem with MapReduce are related to improving the computational efficiency (given the design limitations of the framework), and controlling the memory consumption. The main limitations of the framework and paradigm are:

- Mappers cannot communicate among each other.
- Mappers and reducers always work in (key,value) pairs, so everything must be translated to that mapping structure.
- The number of mappers is controlled by the framework and the input format. Special considerations must be taken to correctly manage the number of mappers the framework creates.
- Iterations in Hadoop are expensive.
- Peer to peer distributed computing techniques cannot be used as clients (in this case, mappers and reducers) cannot communicate.
- Intermediate keys and values, created by the mappers, must be thought carefully as problems in balance of intermediate keys could significantly impact in algorithm's performance.

6.2 The Noncooperative Domain Decomposition

The NonCooperative Domain Decomposition (NDD) approach uses a perturbative search algorithm by fixing d literals in each iteration. An example of the search algorithm is described in Figure 23. The algorithm receives as an input parameter literals already fixed. In the Figure, literals 1 and 2 are already fixed with true. After, a new candidate solution is created given the received literals. In the example, the first candidate solution created is 1 2 3 4 5, where all literals from 1 to 5 are set to true. Then, the candidate solution is evaluated to check if is valid or not. If the candidate solution is valid, then it is emitted using as output the same key received as input and as value the candidate solution found. Afterwards, a literal's truth value is flipped creating a new candidate solution (in the example, the second candidate solution created is 1 2 -3 4 5) and the process is repeated again until all candidate solutions for that search space are evaluated. The reducer process in the NDD algorithm just checks if each received candidate solution is a solution to the problem. Figure 23 shows the Noncooperative Domain Decomposition approach presented and Algorithm 6 shows the mapper process.

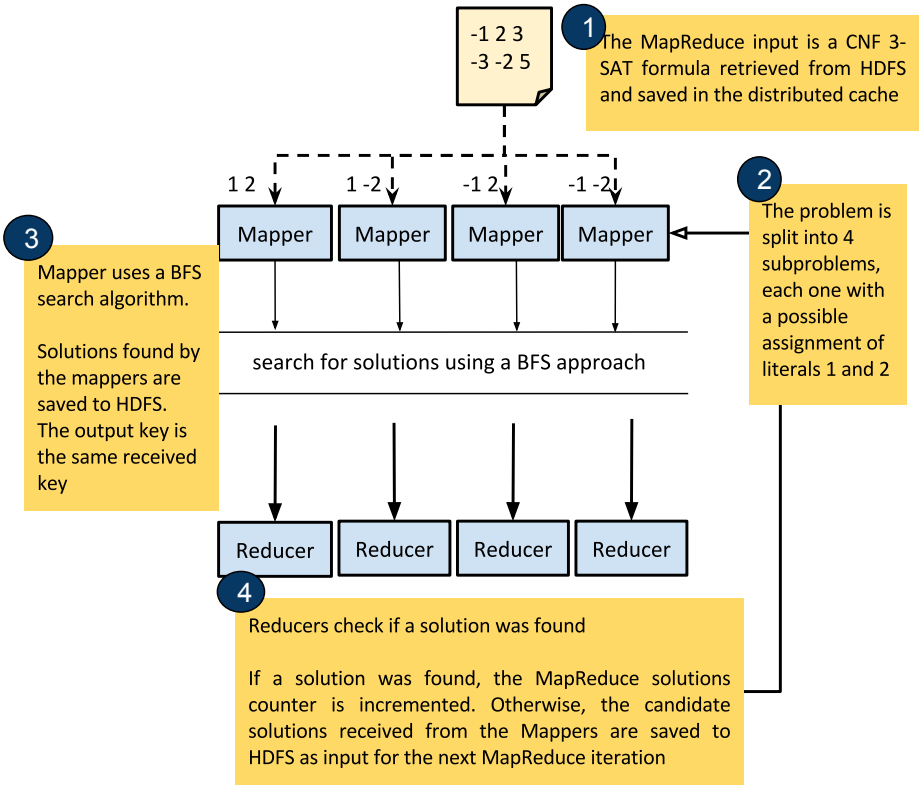

Figure 23: 3SAT MapReduce - Noncooperative Domain Decomposition (NDD)

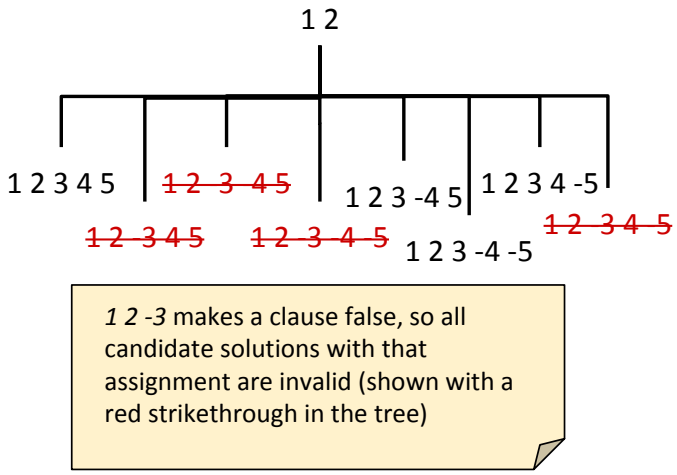

Figure 24: Example of the 3-SAT Perturbative Search
The algorithm starts with literals 1 and 2 already fixed. In each leave of the search tree tests if a new candidate solution can be solution to the problem (e.g: 1 2 -3 4 5). In this example, 1 2 -3 makes a clause false and so, all combinations with that assignment (e.g.: 1 2 -3 4 5, 1 2 -3 -4 -5) are invalid candidate solutions.

Algorithm 6 3-SAT Mapper

```
1:  setup method:
2:  problem formula ← from HDFS
3:  mapper method:
4:  (x_i...x_r) ← select d number of literals to set
5:  while not all assignments tested do
6:    X = set literals (x_i...x_r)
7:    if X is candidate solution then
8:      if all literals set then
9:        save solution to HDFS
10:       increment solutions counter
11:     else
12:       emit: key= input key, value = X
13:     end if
14:   end if
15: end while
```

The key emitted by each mapper is the same key received as input, thus the number of reducers is the same as the number of mappers. The only of drawback of this approach is that the algorithm have sever local imbalance in the reducers (e.g.: one reducer process could receive many values for a key and other reducer could receive few of them). Nevertheless, in this first approach, the reducers only prepare the information for the next iteration, setting the received values as output of the MapReduce algorithm.

Using this noncooperative domain decomposition approach, the Mapper al-

Algorithm 7 3-SAT MapperReduce job

```
1: job ← createJob() [split problem in 2 × max_mappers subproblems] {initial job}
2: job.run_job                                   {creates max_mappers mappers}
3: while solution not found and not finished do
4:   if solution found then
5:     finish algorithm (satisfiable: TRUE, solution stored in HDFS)
6:   else
7:     if all literals have been fixed then
8:       finish algorithm (satisfiable: FALSE, problem without solution)
9:     else
10:      input_new ← job.output                  {get previous job results}
11:      subproblems_number ← job.get_number_of_subproblems()
12:      job_new ← createJob()                   {create new MapReduce job}
13:      set_number_of_mappers(new_job, subproblems_number)
14:      setJobInputandOutput(job_new, input_new, output_new)
15:      job_new.run_job
16:    end if
17:  end if
18: end while
```

gorithm is very inefficient. No pruning can be performed on the search space. For example, if each mapper fixes 20 literals, then each mapper will have to test 2^{20} possible assignments. This approach do not scale for solving large 3-SAT problems, and even when dealing with small problems the algorithm does not solve the 3-SAT in reasonable execution time. In the noncooperative domain decomposition approach, the mappers are all independent and do not collaborate, having a negative impact the execution time of the algorithm. If one Mapper finds a solution, the MapReduce algorithm do not end until all MapReduce executions (including all the mappers and reducers).

The algorithm was not efficient in memory consumption, reaching memory limits on a Intel i7 2.2 GHz 16GB RAM for even small problems with more than 30 literals. The main job splits the problem into 4 subproblems and suppose each mapper tries to fix *15* literals. Suppose all candidate solutions are valid, thus each mapper generates $2^{15} = 32768$ assignments and after the first iteration $4 \times 32.768 = 131.072$ subproblems are generated. In this example, the second iteration of the algorithm would generate 131.072 mappers (and reducers) which might cause an OutOfMemory error.

Literal configurations assignments that generates invalid candidate solutions is not used to reduce the search tree. The perturbative search algorithm does not prune branches of the tree based on literal configurations. Figure 24 shows an example of the perturbative search algorithm where the literal configuration *1 2 -3* makes a clause false but it is not used to prune.

6.3 The Noncooperative Deep First Search with Partial Assignments

The Noncooperative Deep First Search with Partial Assignments (NDFSwithPA), described in Figure 26, is based on the NDD algorithm but improving the mapper's search algorithm used and the mapper output is changed to reduce the whole MapReduce memory consumption by reducing the number of intermediate output keys and values. The mappers use a Deep First Search (DFS) with pruning algorithm instead of using a perturbative search algorithm, as shown in the example of Figure 25. Each time a candidate solution is invalid (e.g: cannot generate a valid solution to the problem), that whole branch is pruned avoiding searches in branches where there are no solutions.

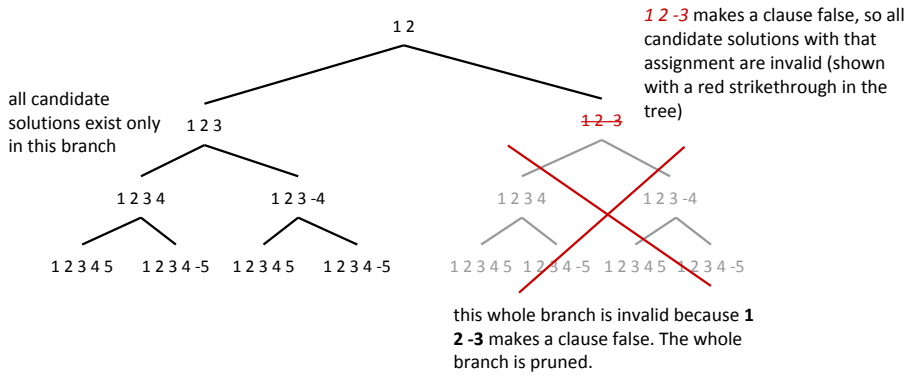

Figure 25: 3SAT MapReduce - mapper Deep First Search (DFS) search algorithm example having literals 1 and 2 already set.

The NDFSwithPA algorithm tackles the OutOfMemory issues by reducing the number of candidate solutions the mapper process emits. In each iteration of the whole MapReduce algorithm, the mapper phase fixes d new literals and outputs all the previously fixed literals plus the new fixed literals. The approach taken in the NDFSwithPA algorithm is to emit only the new literals fixed, grouping with the previously fixed literals so that the total number of new candidate solutions each MapReduce job iteration emits is at least 2^d. For example, suppose the literals 1 and 2 are already fixed and the problem was split in 4 subproblems generating 4 mapper process. The first mapper will receive 1 and 2, the second 1 and -2, the third -1 and 2 and the fourth mapper will receive as input -1 and -2. If each mapper tries to fix another 2 literals, suppose the literals 3 and 4 and all candidate solutions composed by literals 1, 2, 3 and 4 are valid candidate solutions, then the NDD algorithm's mapper phase will output all possible combinations of literals 1, 2, 3 and 4. On the other hand, the NDFSwithPA algorithm will output only the combinations of literals 3 and 4 and save as path in HBase all combinations of literals 1 and 2 as keys with all combinations of literals 3 and 4 as values. The NDFSwithPA algorithm, reconstructs previous assignments (paths) in the setup method of mappers and reducers, which is called once by each mapper and reducer so that new fixed literals are evaluated with all previous assignments to check if a solution is found.

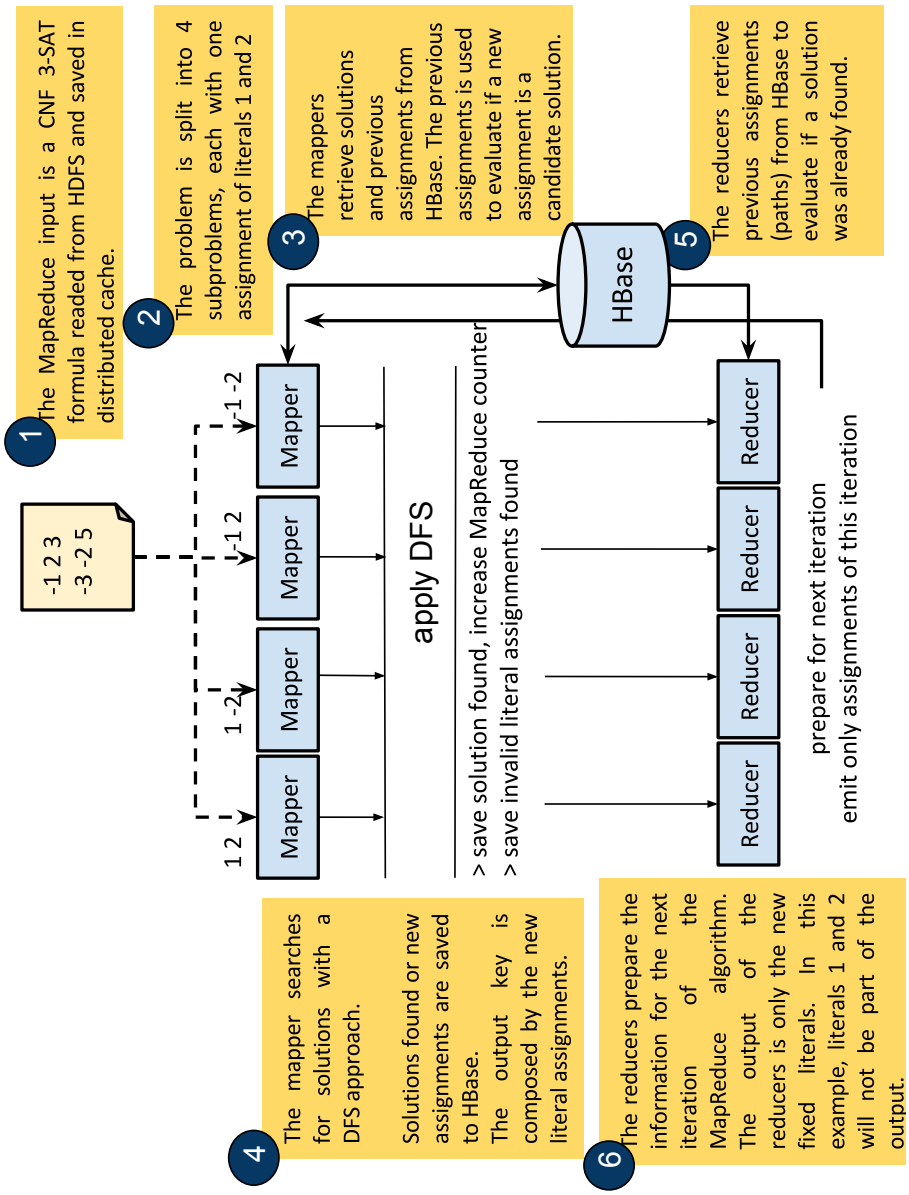

Figure 26: 3SAT MapReduce - Cooperative with Partial Assignments (CPA)
(2) The algorithm decomposition is the same as the NDD approach; (3) All the mappers retrieve previous assignments based on the received literal assignments. For example, if a mapper receives as input 3 4, then it will use that as the key to search for assignments of previous iterations. This information is used to evaluate if a candidate solution is valid; (4) The mappers use as key the new assignments as output. For example, suppose -3 and 4 are new assignments found and 1 2 are the received assignments, then the output of that mapper will be key=3 4, value=1 2; (5) In order to correctly evaluate if a received assignment is solution, the reducers also retrieve previous assignments from HBase. This is done in the setup method, which is called only once before calling the reduce method; (6) If a solution is found, the reducers increment the solutions counter and end the algorithm. If a solution is not found, then only the received values (new assignments) are written to HDFS.

The execution time of the mapper is considerably reduced by using a DFS search algorithm, compared with the NDD algorithm. For a problem instance with 30 literals and 134 clauses, the NDFSwithPA algorithm has a speedup up to 50%, as shown in the Table 3.

literals	*clauses*	**execution time (seconds)**	
		NDD	**NDFSwithPA**
20	150	30	28
25	90	43	29
30	134	120	63

Table 3: Execution times of NDD and NDFSwithPA algorithms

The NDFSwithPA algorithm uses HBase to centralize the fixed literals the mapper processes fixes in each iteration. The HBase table schema has one table with one column family called *path*. The row used in the table is the received literal assignment and the value are the new fixed literals. An example of how the literals assignments are saved in Hbase after two iterations of the MapReduce algorithm is shown in Figure 27.

The number of intermediate keys of the NDFSwithPA algorithm is greatly exponentially reduced when compared with the NDD algorithm. Suppose d literals are fixed in each iteration and after two iterations, all candidate solutions are valid. The NDD algorithm generates up to $2^d \times 2^d$ possible candidate solutions, thus $2^d \times 2^d$ mapper process and reducer process are created. On the other hand, the NDFSwithPA generates 2^d intermediate keys, significantly reducing the number of process created.

Even though the NDFSwithPA significantly reduces the number of intermediate assignments, for medium problems (more than 30 literals and 150 clauses) an OutOfMemory error is still easily reached in a computer with intel i7 2.2 GHz processor with 16 GB of RAM. Moreover, the overhead of saving the assignments to Hbase and retrieving them from the database in each iteration, in every mapper and reducer is considerable.

Another problem with the devised algorithm is that it needs to query many times Hbase for each mapper and reducer process consuming considerable mount of time. For example, considering the previous example, the mapper input for the third iteration would be different assignments for literals 5 and 6. The mapper process that receives the fixed literals $-5-6$ queries the database and retrieves [3 4, -3 -4]. After, for each retrieved assignment (3 and 4, -3 and -4), the database is queried again to retrieve the remaining previous assignments. In this example, using (3, 4) as key the literal assignments (1,2) is retrieved and using (-3, -4) as key, the literal assignments [(1 2), (-1 2), (-1 -2)] is retrieved. Finally, for each retrieved assignment (1,2), (-1 2), (-1, -2) HBase is queried again to check if a full candidate solution can be reconstructed. Hbase in this last step returns no data so mappers and reducers know they have all information to reconstruct the full candidate solution. In this example, each mapper and reducer on the third iteration has to query HBase six times.

The NDFSwithPA algorithm is yet not collaborative; if a Mapper finds a solution to the problem, the algorithm still has to execute all Mappers and assignments until the end.

Figure 27: 3SAT MapReduce - NDFSwithPA example of invalid literals
After two iterations, the following candidate solutions could be recreated using the information saved on Hbase: [-5 -6 3 4 1 2], [-5 -6 3 4 1 2] , [-5 -6 -3 -4 1 2], [-5 -6 -3 -4 -1 2], [-5 -6 -3 -4 -1 -2], [-5 6 3 -4 1 2] , [-5 6 3 -4 1 -2] , [-5 6 3 -4 -1 -2], [-5 6 -3 -4 1 2], [-5 6 -3 -4 -1 2], [-5 6 -3 -4 -1 -2], [5 6 -3 -4 1 2], [5 6 -3 -4 -1 2], [5 6 -3 -4 -1 -2]

6.4 The Noncooperative Deep First Search with Recalculated Mappers

The Noncooperative Deep First Search with Recalculated Mappers (NDFSwithRM) is a variation of the NDFSwithPA algorithm reverts the NDFSwithPA approach of emitting as output only the literals set in each iteration. Instead, it outputs all the assignments as the NDD algorithm but recalculating in each iteration the number of mappers to be created based on the candidate solutions found in the previous iteration of the MapReduce algorithm. Instead of reducing the number of assignments in order to reduce the number of mappers, this approach fixes the number of mappers by changing the number of lines (i.e., problems) each Mapper receives.

The number of mappers to be created by Hadoop depends mainly on the *InputFormat* and the input files. The maximum number of mappers ($max\_mappers$) is specified as an input parameter of the MapReduce job and it is used to split the problem in $2 \times max\_mappers$ subproblems, fixing r literals, specified by Eq. 8.

$$r = \frac{\log(2 \times max\_mappers)}{\log(2)} \tag{8}$$

In order to have $max\_mappers$ processes, the number of lines y each mapper receives in each iteration is calculated by Eq. 9. The *NLineInputFormat* input format is used using y as the number of lines each mapper receives.

$$y = \frac{number\_of\_lines\_input\_file}{max\_mappers} \tag{9}$$

The NDFSwithRM algorithm selects the literals to fix based on the frequency of appearance of the literals in the formula, improving the pruning of branches. Selecting literals that appear more frequently in different clauses, the probability of selecting an invalid candidate solution and thus, pruning that branch is higher.

The NDFSwithRM mapper process DFS search algorithm is implemented in a recursive fashion. If a literal assignment both for true and false makes at least one clause false, then the search for that mapper is over and the branch is pruned. When the mapper succeeds in fixing d literals, then a pair key-value is written, using as key the mapper key received as input (byte offset) and the value is the concatenation of the received fixed literals string with the new fixed ones.

The algorithm is yet not collaborative in the mappers when finding a solution or when sharing learning invalid literal assignments. The reducer process (Algorithm 9) checks if a solution has been found. In that case, the solutions counter is incremented and the solution is saved to HDFS. Otherwise, in case not all literals are already set, it emits the received candidate solutions so they are used as input for the next MapReduce iteration. The number of reducers is the same as the number of mappers, controlled in Hadoop by the function *job.setNumReduceTasks(int)*.

Algorithm 8 3-SAT *NDFSwithRM* mapper (INPUT: *depth* to use in the DFS, mapper *key*, mapper *value*, new_fixed_literals)

```
1: problem definition ← value
2: if depth == 0 then
3:    emit key = input_key, value = problem definition + new_fixed_literals
4: else
5:    [get next literal to fix, in order, no repetition]
6:    lit = get_literal_to_fix(value + new_fixed_literals)
7:    if satisfiable(lit + value + new_fixed_literal) then
8:       recursive call with: depth−1, value, lit + new_fixed_literals
9:    end if
10:   {check if the literal in false is a possible assignment}
11:   if satisfiable(-lit + value + new_fixed_literals) then
12:      recursive call with: depth−1, value, -lit + new_fixed_literals
13:   end if
14: end if
```

Algorithm 9 3-SAT *NDFSwithRM* reducer (INPUT: subproblems found by mappers as values)

```
1: for all values received do
2:    if all literals are set then
3:       if solution found then
4:          save solution to file
5:          increment counter of solutions found
6:       end if
7:    else
8:       emit null, value
9:    end if
10: end for
```

6.5 The Cumulative Learning Technique

In order to improve the algorithm performance, a cumulative learning technique is applied, using a master database to store learnt clauses and solutions found during the search. The master database is implemented using HBase, with two column families: *invalid_literals* and *solutions_found*.

In each Mapper, the setup method retrieves from HBase solutions already found in the search or a set of all invalid literals assignments that make at least one clause false. In the mapper search, each time an assignment makes a clause false, it is saved as an *invalid assignment* to HBase. This assignment could be used by another Mapper in the next iteration. The set of invalid literals is used to avoid unnecessary checks in order to speedup the mapper. When a solution is found, it is send to HBase. This way, the solution will be available to all other Mappers and Reducers to speed up the search.

Two cooperation methods were devised: *collaborative with invalid literals* (CID), using the master database for both invalid literals and solutions found, and *collaborative without invalid literals* (CnoID), using the master database only for solutions found in the search.

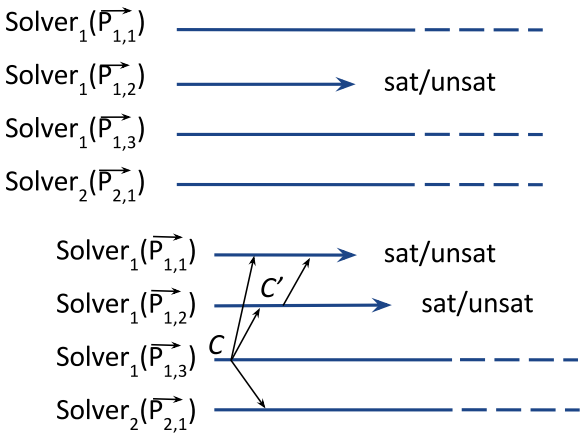

Figure 28: Basic cooperation schema where each solver shares information with other solvers.

Algorithm 10 3-SAT Mapper setup method

```
1: setup method:
2: solution = search solution in HBase
3: if solution not found then
4:    invalid_literals = get invalid literals from HBase
5: end if
6: mapper method:
7: if solution found then
8:    end mapper code
9: else
10:   run previous Mapper code
11: end if
```

The structure of the CID and CnoID algorithms is shown in Figure 29

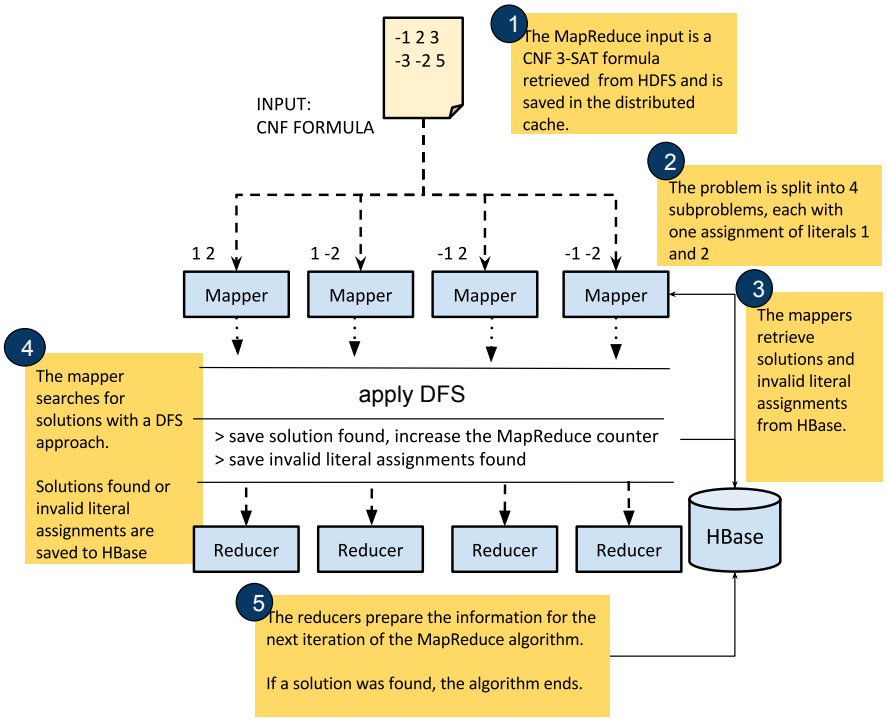

Figure 29: Basic structure of CID and CnoID algorithms

(3) The CnoID algorithm does not use or retrieve invalid literal assignments. The CID algorithm, retrieves found invalid literal assignments from HBase in the setup method, which is called once for each mapper when the mapper is first created. (4) The CnoID algorithm, does not send invalid literal assignments found in the execution. The invalid literal assignments are not used in this variant of the algorithm. The CID algorithm saves invalid literal assignments found in the execution in the cleanup method, which is called once when the mapper ends.

6.6 Experimental Analysis

This section presents the experimental results for the proposed algorithms. The analysis is focused on studying the efficacy of the combined D&C and cumulative learning approaches for solving LSO problems over Hadoop, which was not developed for solving this kind of problems and is focused on data intensive problems.

The experimental analysis was performed in an AMD Opteron 6272 (24 cores at 2.09GHz, 72GB RAM), using HBase with its default configuration and 24 mapper processes. The presented algorithms where executed first on a single node i7 (4 cores at 2.2 GHz) with 16 GB of RAM to test the devised algorithm, with Hadoop 2.2 with a pseudo-distributed configuration of Hadoop [30]. HBase was used with its default configuration for a semi-distributed environment. After, the algorithm was executed in the Cluster Fing [22] with the pseudo-distributed configuration on an AMD Opteron 6272 (24 cores at 2.09GHz, 72GB RAM). Five independent executions were executed, using 15 literals as d=*depth* with 24 mappers when executed on the AMD Opteron node. Three variations of the Map-Reduce algorithm are presented:

- The MapReduce 3-SAT solver (MR 3-SAT)
- The Mapreduce 3-SAT solver using cumulative technique saving invalid literal configurations and found solutions (CID)
- The MapReduce 3-SAT solver using cumulative technique only saving found solutions (CnoID).

Table 4 reports the average execution times in seconds, computed 5 executions of each algorithm.

literals	*clauses*	**execution time (seconds)**		
		MR 3-SAT	**CID**	**CnoID**
20	150	13	6	9
25	90	16	25	15
30	134	30	46	16
32	142	29	53	20
34	150	43	73	42
40	176	559	675	227
42	185	485	746	152
45	197	868	1023	365
50	280	1996	7260	1506

Table 4: Execution times of MR 3-SAT, CID and CnoID algorithms

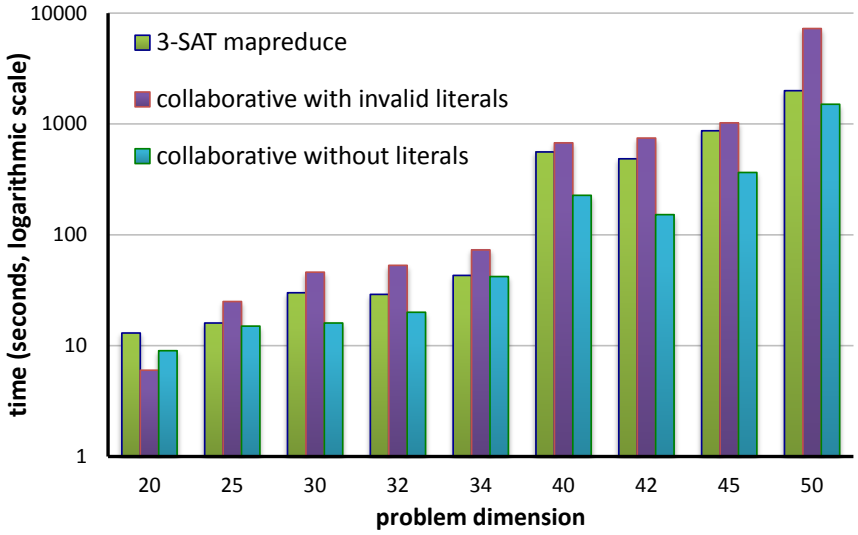

Figure 30: Execution time analysis.

The results show that small problem instances (less than 40 literals) are solved in reduced running times.

The execution time of increases for problem instances having more than 34 literals, because the algorithm splits the problem fixing 6 literals according to

$\frac{log(2 \times max\_mappers)}{\log(2)}$ and then, in each iteration, 15 literals are fixed thus, at least 2 iterations of the MapReduce algorithms are needed to find a solution. Iterative algorithms are not efficient in Hadoop because Hadoop job initialization process is expensive, starting new tasks for job clients and possibly, creating new java virtual machines for new executions. The information needed to execute a new MapReduce job is not kept in the main system memory, so it is usually retrieved from the hard drive.

In each iteration, the search space of each mapper has 2^{15} elements, generating at most 2^{15} new possible assignments. After each iteration, each mapper has exponentially more search spaces to process from the previous iteration, exponentially increasing the execution times. The experimental results show this behavior: the execution time for 40 literals, which needs only one additional iteration to find a solution than the smaller problem instances, is significantly higher.

The proposed algorithms differ from each other in the stop condition used and how they evaluate if a literal assignment makes one clause false. The key aspect that speeds up the CnoID algorithm is the stop condition used, reporting a performance improvement by stopping the algorithm as soon as a solution is found. The MR 3-SAT algorithm finishes only when all mappers have been processed, slowing down the whole algorithm.

The CID algorithm has poor performance compared to the CnoID algorithm. The difference in execution times between CID and CnoID grows with the size of the problem, as more possible invalid literal assignments are found and sent/retrieved to/from HBase, thus increasing the time needed to retrieve all the information from the centralized database which is higher than simply evaluating each possible candidate solution against the read formula.

Empirical analysis demonstrates that maintaining an invalid literal set in a centralized repository has a huge impact on the performance when working with many literals. The execution time of evaluating a particular assignment in a formula is very low compared with the network cost of sending the literals to HBase. Nevertheless, the proposed algorithm used the invalid literal set just to speedup the evaluations of assignments. A smarter use of the invalid literal set could exploit that information to reduce the mapper search space. Some possible strategies to reduce the communication costs of the CID algorithm are:

1. Sending invalid literals configurations in the cleanup method of the mapper (which is executed only once).

2. Using the invalid literal assignments found in the map phase to prune search spaces in the reduce phase.

3. Use invalid literal configurations to fix literals and, doing so, pruning branches of the search space.

All these experimental results give important hints about the main performance issues of developing the deterministic Map-Reduce algorithm to solve a LSO problem. Figure 31 shows a graphical comparison between the three deterministic algorithms devised, showing the performance slowdown of the CID algorithm by using HBase as a repository of invalid literals.

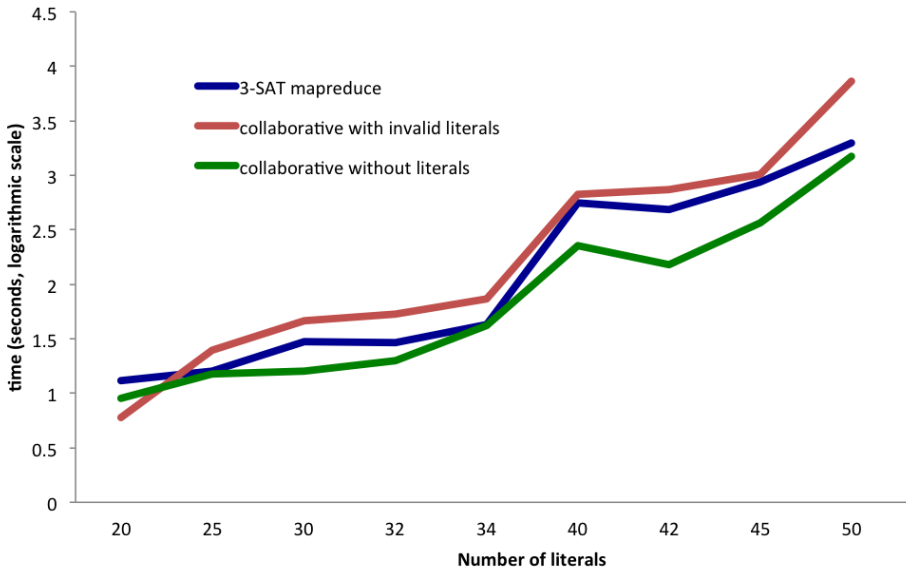

Figure 31: Graphical comparison of the execution time between MR, CID and CnoID algorithms

6.7 The Randomized MapReduce 3-SAT

The Randomized MapReduce 3-SAT (RandMR) is described in this section. The devised algorithm takes advantage of the processing power in mappers and reducers and by using a hybrid implementation with well known deterministic algorithm (Davis–Putnam–Logemann–Loveland (DPLL), described in Algorithm 1) and non-deterministic algorithm (Schöning algorithm, described in Algorithm 4) to solve the 3-SAT.

The motivation of an hybrid implementation is that, with some probability, a solution can be found in any iteration of the algorithm. A comparison example of the deterministic and randomized search algorithms are shown in Figure 32.

The problem starts by splitting the problem in $2 \times number\_of\_mappers$ problems. The Unit Propagation Pure Literal Elimination (Algorithm 1) algorithm is employed in the reducer, to modify the formula based on fixed literals. For each partial solution a new formula is generated, which is saved in Hbase.

Both, mappers and reducers, retrieve the formula associated to the partial assignment received. If no formula is found in Hbase, then the default formula it is used which was saved initially by the main MapReduce job. Each time a mapper or reducer finds a solution or a new formula is generated by the UPPLE algorithm, it is save to Hbase instead of HDFS. The Hadoop Distributed File System is only used to save the intermediate keys and values of hadoop between each iteration and between mappers and reducers. The use of Hbase for storing all data related to the algorithm is an improvement over the use of HDFS, as the information read and written is small and must be accessed fast in a write many read many fashion.

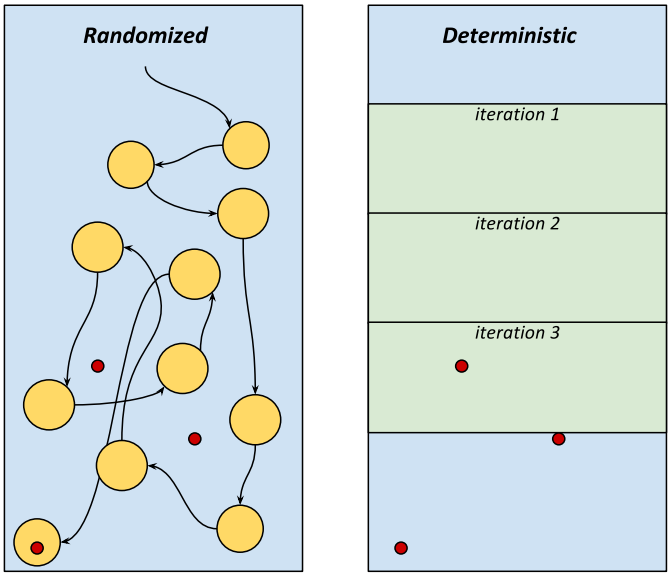

Figure 32: 3SAT randomized mapper process
(1) The red dot denote solutions to the problem. (2) The yellow circles are local searches of the randomized algorithm. (3) The green areas are search spaces of the deterministic algorithm in each iteration. In this example, the randomized algorithm succeeds to find a solution to the problem in the first iteration. On the other hand, the deterministic algorithm finds a solution in the third iteration (a different solution from the one that the randomized algorithm finds).

6.8 Randomized 3-SAT Mapper

The improvements implemented to speedup the Mapper algorithm are described in this section. The mapper base structure is the same as the mapper algorithm described in Algorithm 8 but adding the Schöning's algorithm (Algorithm 4) in the subspace of solution space, given the fixed literals result of the DFS algorithm. A description of the mapper algorithm is presented in Figure 11.

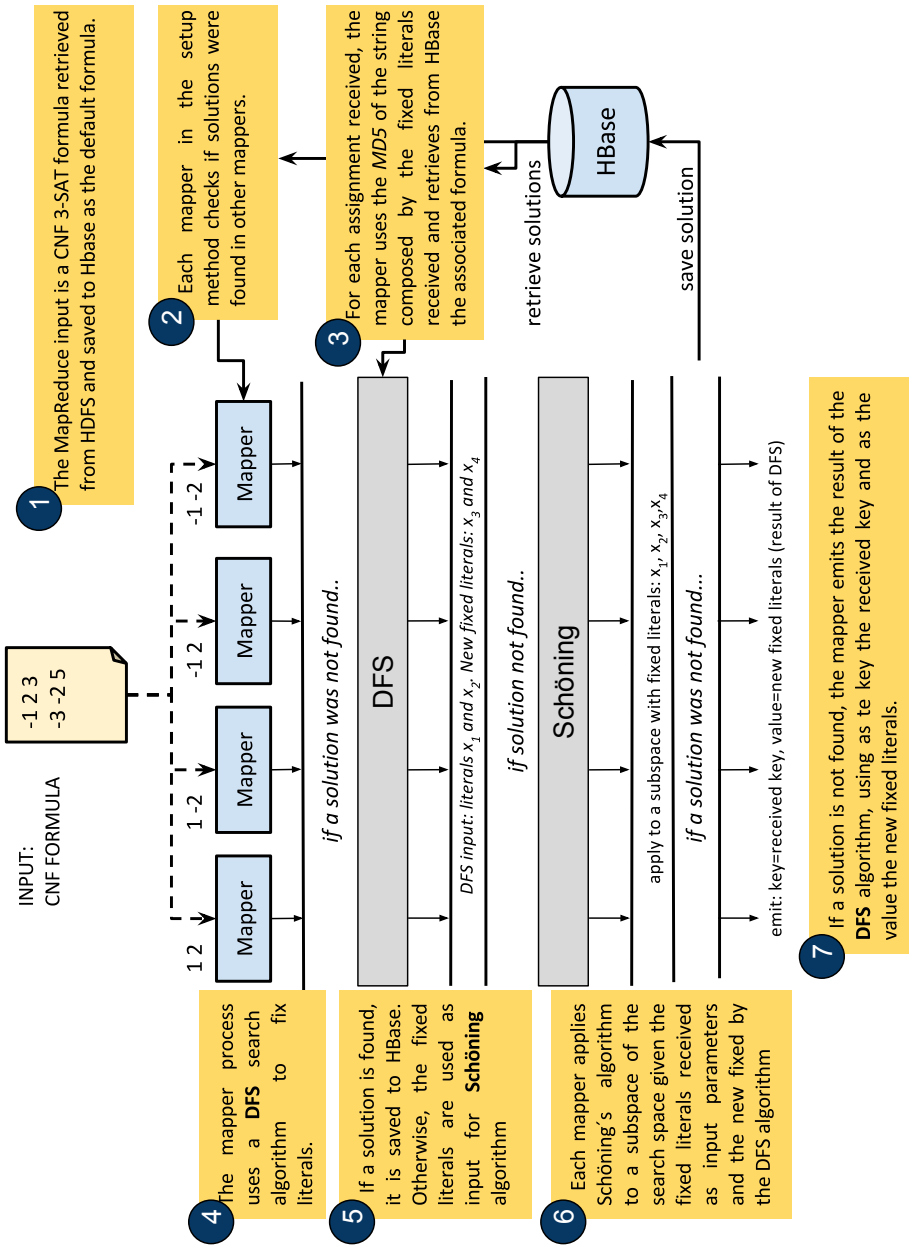

Figure 33: 3SAT randomized mapper process
(1) The main MapReduce job, saves the retrieved formula from HDFS to HBase to its available to all mappers and reducers; (2) The setup method is called once for each mapper and reducer. Each mapper and reduce task checks if a solution was not already found; (3) For each candidate solution received by the mapper (e.g: list of fixed literals), the MD5 of the string composed for such literals is used to retrieve the corresponding formula. For example, suppose mapper 1 receives fixed literals 1 and -2, the MD5("1 2") is used as key to retrieve the formula from Hbase (because each time the UPPLE algorithm is applied, the formula changes based on current assignments). The default formula is returned if no formula is found in Hbase for such MD5; (4) The mapper uses a DFS search algorithm to fix d new literals and, in case of failing to find a solution, the result of this algorithm will be emitted to the reducers; (6) Schöning's algorithm is applied to a subspace of the search space, using the received fixed literals and the new fixed literals result of the DFS algorithm. If a solution is found, then it is saved to HBase.

A graphical representation of the search algorithm used in the mapper process is presented in Figure 34. Each mapper searches for solutions in its own subspace (gray and blue areas). The DFS algorithm is applied inside each mapper subspace given the literals already fixed received as input parameters of the mapper. After, if the DFS algorithm does not fix all literals, the Schöning algorithm is executed inside the mapper search space given the literals fixed received as input parameter and the literals fixed by the DFS algorithm (in green in the figure). The Schöning algorithm is not deterministic and each time it is executed, performs a local search (described in yellow in the figure) and then jumps to another search space (inside the mapper search space, described in yellow in the figure) to search for solutions. Every execution of the Schöning algorithm is different and can succeed in finding a solution at any time with some probability.

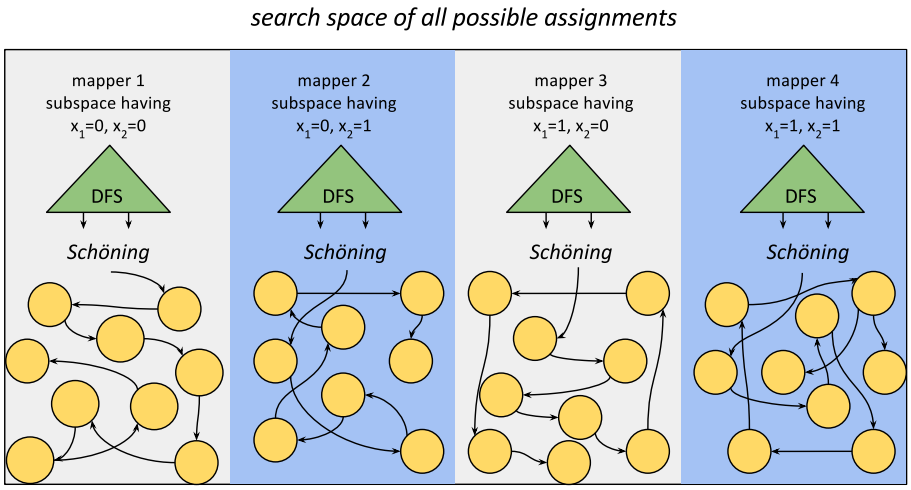

Figure 34: Example of the search approach used in the mappers for the randomized solver.

The mapper starts by applying the DFS algorithm (described in section 6.4). If all literals were not set, for each possible partial solution, the Schöning Algorithm (shown in Algorithm 4) is executed, using the partial assignment as input. The Schöning's algorithm is only applied to a subspace of the entire solution space taking the literals set by the DFS algorithm as input parameter.

If the application of the Schöning algorithm fails to find a solution to the problem, the Mapper outputs as key the key originally received by the mapper and as value the literals set by the DFS algorithm. The mapper algorithm is described in Algorithm 11.

6.9 Randomized 3-SAT Reducer

This section describes the reducer algorithm for the randomized algorithm. The number of reducers is the same as the number of mappers and, apart

Algorithm 11 Randomized 3-SAT Mapper (INPUT: key= LongWritable, value=Text)

```
1: if solution not found then
2:    formula = retrieve formula from Hbase (MD5(value))
3:    possible_solutions = apply DFS algorithm(value, formula)
4:    for all s ← possible_solutions do
5:       if s is solution then
6:          save_solution_to_hbase(s)
7:       else
8:          schonning_result = apply schonning algorithm (s)
9:          if schonning_result is solution then
10:            save solution to Hbase (schonning_result)
11:         else
12:            emit: key=received key, value=s
13:         end if
14:      end if
15:   end for
16: end if
```

from the previous reducer algorithm that just prepared the information for the next MapReduce iteration, this algorithm searches for a solutions using a DFS Algorithm (described in Section 6.3), the Unit Propagation and Pure Literal Elimination algorithm (Algorithm 2 and Algorithm 3) and Schönning algorithm (Algorithm 4).

A description of the randomized version of the reducer process is shown in Algorithm 12. The reducer process, in the setup method, checks Hbase if a solution was already found. If a solution was already found, the algorithm ends. Otherwise, the randomized reducer algorithm starts starts by retrieving the 3-SAT formula for each assignment received, using the MD5 of the candidate solution.

The formula associated for the partial assignment received is retrieved from Hbase. If no formula is found, the formula received as an input parameter of the MapReduce job (and saved to Hbase in the initialization process) is used.

As a result of the UPPLE, a new assignment and new formula is generated. If all literals are not set in the new assignment, the DFS algorithm (Subsection 6.3) are applied to the new assignment and new formula generated by the UPPLE algorithm.

If a solution is not found after applying the DFS algorithm, then Schönning's algorithm is applied to a subspace of the solution space, using as input assignment the result of the DFS algorithm and the formula as a result of the UPPLE algorithm.

If the Reducer fails to find a solution, the assignment returned by the DFS algorithm is saved to HDFS to prepare the input for the next MapReduce iteration, its MD5 is used to save the formula returned by the UPPLE algorithm to Hbase. The randomized reducer algorithm is described in Figure 35.

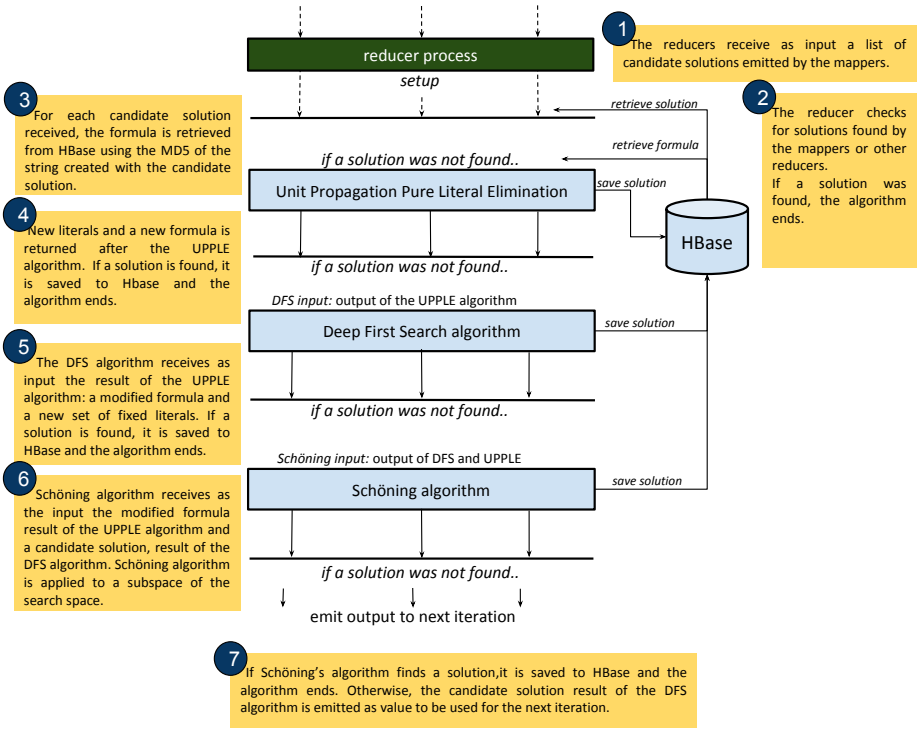

Figure 35: The 3SAT reducer process of the randomized algorithm

(3) If no formula is found for the candidate solution received, then the default formula is used (received as input parameter of the MapReduce job and saved to HBase in the initialization process); (4) The UPPLE algorithm fixes new literals and returns a reducer formula along with a new candidate solution; (6) Schöning's algorithm is applied to a subspace of the search space where the literals present in the received candidate solution are already fixed. It uses the formula result of the UPPLE algorithm to evaluate if a new candidate solution is solution to the 3-SAT problem; (7) If the reducer algorithm fails to find a solution, the reducer saves the new formula result of the UPPLE algorithm using the MD5 of the candidate solution result of the DFS algorithm, which is emitted as output for the next iteration of the MapReduce algorithm.

Algorithm 12 Randomized 3-SAT Reducer Job (INPUT: key=LongWritable, possible_solutions: List[Text])

```
1: if solution not found then
2:   for all s ← possible_solutions do {iterate over all possible solutions}
3:     formula = retrieve_formula_from_HBASE(s)
4:     if  all literals are set in s then
5:       if s is solution then
6:         save solution to Hbase(s)
7:       end if
8:     else
9:       {Apply Unit Propagation and Pure Literal Elimination (UPPLE)}
10:      new_formula, new_assignment = apply_UPPLE(s)
11:      if all literals are set in new_assignment then
12:        if new_assignment is solution then
13:          save new_assignment to HBASE
14:        end if
15:      else
16:        new_assignments = apply DFS algorithm (new_assignment)
17:        {iterate over all new possible solutions}
18:        for all s1 ← new_assignmetns do
19:          if all literals are set in s1 then
20:            save solution s1 to HBASE
21:          else
22:            sch1 = apply Schoning algorithm (s1)
23:            if sch1 is solution then
24:              save solution sch1 to HBASE
25:            else
26:              {emit empty key and possibleSolution as value}
27:              {save new formula after UPPLE for new assignment}
28:              emit: key=NullWritable , value= s1
29:              save new_formula to HBASE with key MD5(s1)
30:            end if
31:          end if
32:        end for
33:      end if
34:    end if
35:  end for
36: end if
```

6.10 Experimental Analysis

Experimental executions where made first on a sandbox environment with a single node having an intel i7 processor (4 cores at 2.2 GHz) with 16 GB of RAM using a pseudo-distributed Hadoop configuration. All reported executions were made on a AMD Opteron 6272 (24 cores at 2.09GHz, 72GB RAM). Because the devised algorithm is not deterministic (so execution times vary between different executions depending on the probability of the algorithm to find a solution), 30 executions of each problem instance were made. The experimental results are shown in Table 5. As described in Figure 34, the Schöning algorithm

is a random algorithm and each execution of the algorithm is different, jumping from one local search to another in the search space. For that reason, a statistic execution was made, running the algorithm 30 times for each problem instance and then reporting the average of the execution time. The standard deviation of the RandMR algorithm is also reported in seconds.

literals	*clauses*	**execution time (seconds)**	
		CnoID	**RandMR**
20	150	9	0.001 ± 0.001
25	90	15	0.002 ± 0.001
30	134	16	0.119 ± 0.020
32	142	20	0.095 ± 0.020
34	150	42	0.378 ± 0.030
40	176	227	8.479 ± 0.700
42	185	152	5.670 ± 0.500
45	197	365	15.780 ± 1.120
50	280	1506	152.048 ± 7.540
100	200	-	8921.049 ± 353.820

Table 5: Execution times of CnoID and randomized MapReduce algorithm (RandMR) as well as the standard deviation of the RandMR algorithm.

The best deterministic algorithm devised (CnoID) is compared with the new randomized algorithm. For the problem instance with 100 literals and 200 clauses, the CnoID algorithm was not executed because of the high execution times reached for the problem instance having 50 literals and 280 clauses, so there is no reported execution time for that problem instance.

The most notably result of the experimental results is shown in Figure 36. The performance improvement of taking more advantage of the processing power of mappers and reducers provides a speedup up to 10× comparing with the best devised deterministic algorithm. The devised algorithm is fully scalable to any number of computers in a cluster or grid to solve any large scale problem, taking full advantage of Hadoop infrastructure.

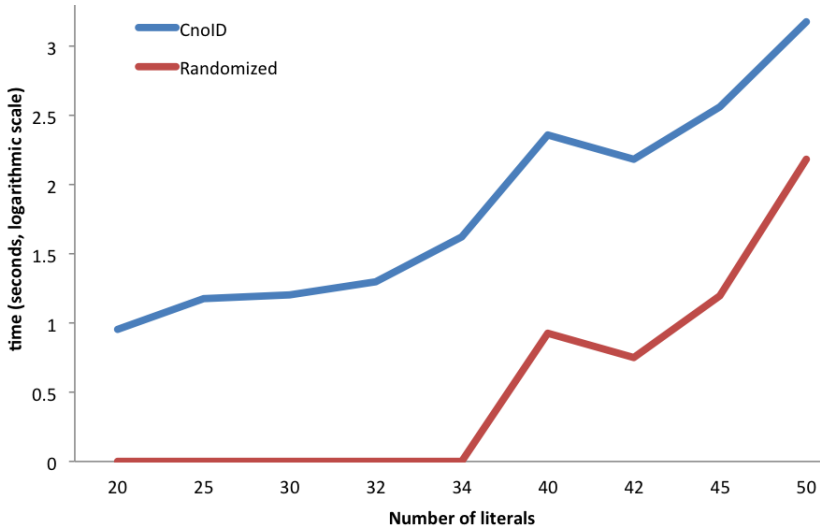

Figure 36: Graphical evolution execution time randomized vs CnoID.

The key difference between the CnoID algorithm and the RandMR is that the randomized algorithm exploits better the map and reduce phase of the MapReduce algorithm, improving the search algorithm used in such phases. In the case of the CnoID algorithm, the reduce phase was not exploited waisting execution time just preparing the information for next MapReduce iteration. In Hadoop, the reinitialization process is expensive as hadoop has to reinitialize everything again, create new JVM's, create a new job which checks for task trackers available, check the data in HDFS needed and start again. The CnoID algorithm, always finishes in the same iteration for a specific problem instance and the RandMR algorithm can succeed to find a solution in less iterations with some probability but never with more iterations.

Taking as an example the problem instance with 40 literals, suppose initially the problem is split into 16 subproblems (fixing initially 4 literals) and in each iteration *d=15* literals are fixed. The CnoID needs at least 3 iterations to find a solution (if any), in each iteration at most 15 literals are fixed. On the other hand, the RandMR algorithm could succeed in finding a solution in any iteration but no more than three iterations are needed. The Schöning algorithm can find a solution in any iteration in any time with some probability. Moreover, if the formula is not hard to solve and the literals fixed produce that the UPPLE greatly simplifies the problem, a solution can also be found by a deterministic manner. Also, the RandMR fixed at least *d=15* literals in each iteration in a deterministic manner, as the CnoID algorithm does, so for this toy example, at most three iterations are needed to find a solution to the problem (if any).

The RandMR algorithm uses different search techniques to search for solutions (DFS, UPPLE and Schöning) in the reducers and (DFS and Schöning) in the mappers, significantly increasing the computations in both, mappers and reducers, thus reducing execution times.

The performance improvement of a randomized algorithm is of an order of magnitude, achieving 10x of speedup comparing with the previous deterministic algorithm. For small problem instances (less than 34 literals), the randomized

algorithm succeeds to find a solution in less than one second so in logarithmic scale, the value is normalized to zero. The speedup of the randomized algorithm is up to 10x, being an order of magnitude faster than the deterministic approach.

Although with very slow probability, the RandMR algorithm has one small problem. The UPPLE algorithm used in the reducers in the RandMR algorithm changes the formula, reducing it by fixing literals that appear unique in a clause or removing from clauses the literals that are already fixed, and propagating its value to other clauses. The result of the UPPLE algorithm is a new formula and a new set of fixed literals. This new formula is saved to HBase and the string created by the MD5 of the candidate solution is used as key. For different candidate solutions found in different mappers, the MD5 of the strings created from those candidate solutions could eventually collide. As different mappers search for solutions in different search spaces and so the same candidate solution is never tested in two mappers, the algorithm assumes that the collisions do not happen. Nevertheless, the algorithm performs a final check agains the original formula received as input parameter when a solution is found, just to be sure of the correctness of the algorithm.

The algorithm can be further improved by improving the mapper algorithm by adding the UPPLE algorithm and/or Schöning algorithm, as the reducer does. The RandMR algorithm was designed to be extensible in the mappers and reducers to any combination of algorithms can be applied in both phases. The depth used when fixing the literals in all the proposed algorithms must be fine tuned and trained to find the best suitable depth to use in mappers and reducers. A small value of d could end up wasting execution time in many iteration of the MapReduce algorithm. On the other hand, a too big value for d could imply that mappers and reducers could end up tacking too much time applying the DFS algorithm, but, would requiere less iterations to find a solution and the UPPLE algorithm would have more information to further simplify the formula. Many things can be modified and tested to speedup the algorithm.

7 Conclusions and future work

This section presents the conclusions of the work of solving a large scale optimization problem using Hadoop and MapReduce. It also briefly details the main lines of future work for improving the current results.

7.1 Conclusions

This thesis tackled the 3-SAT problem, a well known Large Scale Optimization problem, using a novel approach by implementing a distributed algorithm using the MapReduce paradigm over Hadoop.

In the last decade, the dramatical growth of the digital information generated has encouraged the development of different frameworks able to analyze all these information as efficiently as possible. Hadoop is the most known framework for analyzing big volumes of unstructured information in a distributed environment, providing failure tolerance, data replication and process communication. Hadoop was designed to execute algorithms in a distributed infrastructure with little effort by running MapReduce algorithms over a distributed file system.

Despite that Hadoop was not designed to solve LSO problems nor CPU intensive algorithms, it provides a promising infrastructure to run distributed algorithms without having to solve the inherent problems of distributed computing like data replication, failure tolerance and communication between process. In order to implement a MapReduce algorithm to solve the 3-SAT problem, a D&C approach was used using a domain decomposition strategy. The D&C approach was complemented with cumulative learning approaches and stochastic local searches. An iterative approach was used to design and develop the MapReduce algorithm, starting from a simple algorithm to a more complex one. In each iteration, the algorithm was improved focusing in improving the use of computational resources (reducing memory consumption and improving the use of CPU time in searching for solutions) and reducing the execution time. The proposed algorithms had to be specifically designed to Hadoop, taking into account the particular aspects of the underlying infrastructure. Tackling some specific features of the MapReduce paradigm and the Hadoop implementation, such as how Hadoop creates the mappers/reducers and/or how the map phase is executed can lead to execution time improvements to overcome poor algorithm performance.

Mapreduce and Hadoop abstract the developer of the common distributed computing problems, such as communication among processes, failure tolerance, etc, but they add other issues related to the design of the algorithm and constraints for a MapReduce job: how to split the problem, how to implement the stop condition or the kind of search algorithm to use, etc. The MapReduce paradigm does not considers the communication between processes (mappers and reducers). The proposed algorithms achieved the communication between mappers and reducers by sharing information in a centralized database. The mappers have to save/read information from/to the database (e.g.: invalid clauses found, solutions found, etc). The mappers cannot be notified when new information is available so they have to query the database from time to time to retrieve new information, if any. Another challenge when implementing a LSO algorithm using MapReduce is the implementation of the stop condition. In the first proposed algorithm, the stop condition used was the default of any

MapReduce algorithm: it ended after one iteration of the MapReduce phase, checking in the main job if a solution was found. The improvement of the implementation of the stop condition dramatically improved the algorithm, by saving saved solutions to the centralized database so mappers and reducers would avoid execution if a solution was already found. The MapReduce algorithm does not considers the case that a mapper execution is conditioned by the result of another mapper execution. Moreover, it also does not considers the case that the reduce phase can be conditioned by the result of the execution of a particular mapper process. In both cases, improving how the mappers and reducers communicate and the stop condition implemented lead to improvements in the execution time of the LSO algorithm.

The first algorithms focused on the development of a strait-forward approach, to evaluate if Hadoop infrastructure was able to execute CPU intensive algorithms and to face the problems of the underlying infrastructure as early as possible. The first problems tackled were the excessive memory consumption of the MapReduce algorithm by reducing the number of mappers generated in each iteration of the algorithm. The cumulative learning approach without using invalid literals, implemented in the CnoID algorithm was able to improve the performance of a simple D&C MapReduce algorithm. This strategy is implemented following the way Hadoop executes a MapReduce job and using HBase as a collaborative database of solutions. On the other hand, using a centralized database of invalid literals was ineffective to speed up the search.

The randomized algorithm significantly improved the performance of the MapReduce algorithm, reaching up to 10x of speedup comparing execution times. The randomized version of the algorithm significantly improved the reducer algorithm. In first versions of the algorithm, the reducer was a dummy algorithm but in the randomized version, the reducer algorithm also searched for solutions using a hybrid approach (as well as the mapper in the randomized version of the algorithm). The randomized algorithm was able to find solutions to the problem in less iterations than the deterministic approaches, by having a non-deterministic algorithm that can potentially find a solution in any iteration. Because the devised algorithm is a hybrid implementation (deterministic and non-deterministic approaches), it still succeeds in finding when a problem instance does not have a valid solution. The randomized algorithm better exploited the processes created by Hadoop, by searching for solutions in both, the mappers process and the reducers process.

Using HBase as a centralized source helps the synchronization and cooperation among mappers and reducers and its fully distributed. In the randomized version, the solutions and the formula were also saved to HBase. Hbase is optimized for many small saves that are read many times so was a natural selection to save the modified formulas in the randomized approach. The randomized algorithm uses the centralized database not only to save solutions found and to retrieve them to implement the stop condition to the algorithm, but also is used to save the modified formulas found by the UPPLE algorithm in each reducer process. The key used to save the modified formula is the MD5 of the set of literals already fixed in the reducer process. Despite that the MD5 of two assignments could eventually collide, it is with a very small probability. The centralized database can be further exploited by saving conflict clause analysis to further improve the algorithm.

7.2 Future work

The randomized algorithm as designed to scale horizontally without any limitation to be executed on a large Hadoop cluster. A future line of work is to test the devised algorithm in a very large cluster with hundreds of commodity machines, solving a really large problem instance with at least 600 literals. In this context, the only extra work that needs to be done is to configure Hadoop and Hbase to be executed using the cluster configuration.

Apart from executing the devised algorithm in a large cluster, the algorithm can be further improved, better exploiting the MapReduce algorithm. A future line of work is to fine tune the devised algorithm by improving both mappers and reducers. The UPPLE algorithm can be used in the mappers and a better clause sharing algorithm can be used. Conflict clause algorithms have not been used in this work and, as described in the related work, can significantly improve the algorithm performance. Moreover, the heuristic used to fix the literals can be further improved by better using the formula information to select the best literal to fix in each step.

The algorithm presented in this thesis uses a D&C approach to split the problem in subproblems and each mapper searches for solutions in a subspace of the search space. Another line of future work is to implement a portfolio approach using MapReduce, where each mapper runs a different configuration of different algorithms to solve the 3-SAT. In the context of a portfolio approach using MapReduce, cooperation is more limited because, as explained before, it is achieved by using a centralized database and each process needs to explicitly query the database to retrieve new information, if any.

The final line of research relates to the execution of the randomized version of the algorithm in Spark [96]. Spark is a big data analysis framework, which is more efficient for iterative, in-memory computing than Hadoop. In the last few years, Spark has taken much interest in the academic and in the industry for analyzing big volumes of information. Spark is known to be at most 10x faster in iterative algorithms than Hadoop, and is currently used by a wide range of companies and gaining a lot of adoption, known as the replacement of Hadoop [88]. Because LSO problems have never been solved using Hadoop nor Spark, it could be interesting to evaluate the difference in execution time and resource consumption of both alternatives when solving a cpu intensive algorithm.

References

[1] How large is the digital universe? how fast is it growing? [Accessed online in October 2015 at `http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm`].

[2] S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. *Journal of the ACM*, 45(1):70–122, Jan. 1998.

[3] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inf. Process. Lett.*, 8, 1979.

[4] G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Sais. A generalized framework for conflict analysis. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing*, 2008.

[5] T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, 1996.

[6] T. Balyo, P. Sanders, and C. Sinz. Hordesat: A massively parallel portfolio SAT solver. In *Theory and Applications of Satisfiability Testing*, 2015.

[7] D. P. Bertsekas. Dynamic programming and suboptimal control: A survey from ADP to MPC. In *CDC Proceedings*, 2005.

[8] A. Biere. Lingeling, plingeling, picoSAT and precoSAT. Technical report.

[9] S. . Blochinger. Parallel SAT solving on peer-to-peer desktop grids. *Journal of Grid Computing*, 8, 2012.

[10] R. E. Burkard, E. Çela, P. M. Pardalos, and L. S. Pitsoulis. The quadratic assignment problem, 1998.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, 2006.

[12] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[13] M. Chiarandini, I. Dumitrescu, and T. Stützle. Stochastic local search algorithms for the graph colouring problem. In *Approximation Algorithms and Metaheuristics; Computer and Information Science Series*, 2005.

[14] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8, 2015.

[15] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[16] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engineering*, 11, 2009.

[17] S. Cook. The complexity of theorem proving procedures. In *Proc. 3rd Annual ACM Symposium on Theory of Computing*, 1971.

[18] S. A. Cook and D. G. Mitchell. Finding hard instances of the satisfiability problem: A survey. In *Satisfiability Problem: Theory and Applications : DIMACS Workshop*, 1996.

[19] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *In Proceedings of the Eleventh National Conference on Artificial Intelligence*, 1993.

[20] J. D. Davis, Z. Tan, F. Yu, and L. Zhang. Designing an Efficient Hardware Implication accelerator for SAT solving. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2008.

[21] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5, 1962.

[22] F. de Ingeniería Udelar. Cluster fing. [Accessed online in July 2014 at `https://www.fing.edu.uy/cluster/`].

[23] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51, 2008.

[24] H. Deleau, J. Christophe, and M. Krajecki. GPU4SAT: solving the SAT problem on GPU. In *9th International Workshop on State-of-the-Art in Scientific and Parallel Computing*, 2008.

[25] H. Deleau, C. Jaillet, M. Krajecki, and C. Syscom. GPU4SAT: solving the SAT problem on GPU.

[26] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[27] N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, 2003.

[28] M. Forum. MPI library standard 2.1.

[29] A. Foundation. Apache avro. [Accessed Online in October 2015 at `http://avro.apache.org/`].

[30] A. Foundation. Hadoop pseudo-distributed installation. [Accessed online in July 2014 at `http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/SingleCluster.html#Pseudo-Distributed_Operation`].

[31] J. Franco and M. Paull. Probabilistic analysis of the davis putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, 5, 1983.

[32] J. Franco and R. Swaminathan. Average case results for satisfiability algorithms under the random-clause-width model. *Annals of Mathematics and Artificial Intelligence*, 20, 1997.

[33] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[34] L. George. *HBase: The Definitive Guide*. O'Reilly Media, first edition, 2011.

[35] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS- Operating Systems Review*, 37, 2003.

[36] M. L. Ginsberg and D. A. McAllester. GSAT and dynamic backtracking. In *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming*, 1994.

[37] A. Golberg. Average case complexity of the satisfiability problem. *Proc. Fourth Workshop on Automated Deduction*, pages 1–6, 1979.

[38] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[39] Y. Hamadi and L. Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 6, 2009.

[40] Y. Hamadi and C. M. Wintersteiger. Seven challenges in parallel SAT solving. *AI Magazine*, 34, 2013.

[41] T. Hertli. 3-SAT faster and simpler: Unique-SAT bounds for PPSZ hold in general. In *Proceedings of the IEEE 52nd Annual Symposium on Foundations of Computer Science*, 2011.

[42] A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39:345–351, 1987.

[43] H. Hoos and T. Sttzle. *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[44] J. Huang and A. Darwiche. A structure-based variable ordering heuristic for SAT. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, 2003.

[45] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, 2010.

[46] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä. Incorporating clause learning in grid-based randomized SAT solving, 2009.

[47] A. E. J. Hyvärinen, T. A. Junttila, and I. Niemelä. Partitioning SAT instances for distributed solving. In *Proc. of LPAR*, 2010.

[48] J. . N. Hyvärinen. Partitioning SAT instances for distributed solving. In *Proc. of 17th LPAR*, 2010.

[49] I.Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[50] A. Jain. *Instant Apache Sqoop*. Packt Publishing, 2013.

[51] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[52] V. Klee, G. Minty, and W. U. S. D. O. MATHEMATICS. *How good is the Simplex Algorithm*. Defense Technical Information Center, 1970.

[53] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer Publishing Company, Incorporated, 4th edition, 2007.

[54] S. Kottler and M. Kaufmann. SArTagnan - A parallel portfolio SAT solver with lockless physical clause sharing. In *Pragmatics of SAT*, 2011.

[55] N. Kuromatsu, M. Okita, and K. Hagihara. Evolving fault-tolerance in hadoop with robust auto-recovering jobtracker. *Bulletin of Networking, Computing, Systems, and Software*, 2, 2013.

[56] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[57] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5, 2012.

[58] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

[59] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[60] B. M., A. G., and N. S. A parallel spatial quantum search algorithm applied to the 3-SAT problem. In *In Proceedings of XII Argentine Symposium on Artificial Intelligence*, 2011.

[61] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[62] C. Martella, R. Shaposhnik, and D. Logothetis. *Giraph in Action*. Manning Publications Company, 2015.

[63] S. W. Meyer, Schönfeld. 3-SAT on CUDA: Towards a massively parallel SAT solver. In *Proc. High Performance Computing and Simulation Conference*, 2010.

[64] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992.

[65] P. Moscato and M. G. Norman. A memetic approach for the traveling salesman problem implementation of a computational ecology for combinatorial optimization on message-passing systems. In *In Proceedings of the International Conference on Parallel Computing and Transputer Applications*, 1992.

[66] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, 2001.

[67] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, 2001.

[68] M. Motoki and R. Uehara. Unique solution instance generation for the 3-satisfiability (3sat) problem.

[69] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[70] R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, 1998.

[71] R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. *Journal of the ACM*, 52, 2005.

[72] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing*, 2007.

[73] Z. Posypkin, Semenov. Using BOINC desktop grid to solve large scale SAT problems. 13.

[74] G. Reinelt. *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer-Verlag, Berlin, Heidelberg, 1994.

[75] H. F.-W. Sadrozinski and J. Wu. *Applications of Field-Programmable Gate Arrays in Scientific Research*. Taylor & Francis, Inc., Bristol, PA, USA, 1st edition, 2010.

[76] S. Sahni and T. Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23(3):555–565, 1976.

[77] T. Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *40^{th} Annual Symposium on Foundations of Computer Science*, 1999.

[78] B. Schubert, Lewis. Pamiraxt: Parallel SAT solving with threads and message passing. *Journal on satisfiability, boolean modeling and computation*, 6.

[79] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1)*, 1994.

[80] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992.

[81] J. Shafer, S. Rixner, and A. L. Cox. The Hadoop distributed filesystem: Balancing portability and performance. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software*, 2010.

[82] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[83] J. a. P. M. Silva and K. A. Sakallah. GRASP—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, 1996.

[84] I. Skliarova and A. d. B. Ferrari. Reconfigurable Hardware SAT Solvers: A Survey of Systems. *IEEE Trans. Comput.*, 53, 2004.

[85] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, New York, NY, USA, 2nd edition, 2001.

[86] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2, 2009.

[87] S. M. Trimberger. *Field-Programmable Gate Array Technology*. Kluwer Academic Publishers, Norwell, MA, USA, 1994.

[88] Typesafe. Apache spark - preparing for the next wave of reactive big data, 2015. [Accessed online in January 2016 at `https://dzone.com/articles/apache-spark-survey-typesafe-0`].

[89] S. . Vagner. Parallel resolution of the satisfiability problem (SAT) with OpenMP and MPI. In *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, 2005.

[90] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[91] K. Vander-Swalmen, Dequen. A collaborative approach for multi-threaded SAT solving. *International Journal of Parallel Programming*, 37.

[92] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[93] C. Williams and T. Hogg. Exploiting the deep structure of constraint problems. *Artificial Intelligence*, 70:73–117, 1994.

[94] C. . Wolski. GridSAT: A chaff-based distributed SAT solver for the grid. In *In Proceedings of the ACM/IEEE Conference on Supercomputing*, 2003.

[95] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, 2013.

[96] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[97] P. Zikopoulos and C. Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne, 2011.