Hybrid algorithms for 3-SAT optimisation using MapReduce on clouds

Marcos Barreto and Sergio Nesmachnow*

Universidad de la República,
Herrera y Reissig 565,
Montevideo, 11300, Uruguay
Email: mbarreto@fing.edu.uy
Email: sergion@fing.edu.uy
*Corresponding author

Andrei Tchernykh

CICESE Research Center,
Carretera Ensenada-Tijuana 3918, Zona Playitas,
Ensenada, 22860, Mexico
Email: chernykh@cicese.mx

Abstract: This article presents the advances on applying a MapReduce approach for solving optimisation problems using Hadoop on cloud computing systems. The main advantages and limitations of the proposed strategy are presented and commented. A concrete case study is reported, analysing several algorithmic approaches to solve the 3-SAT, a well-known version of the Boolean satisfiability problem. Several variants of the MapReduce 3-SAT solver are designed and evaluated to demonstrate that the collaborative approach is a promising option for solving optimisation problems in the cloud.

Keywords: MapReduce; Hadoop; optimisation; cloud; 3-SAT.

Reference to this paper should be made as follows: Barreto, M., Nesmachnow, S. and Tchernykh, A. (xxxx) 'Hybrid algorithms for 3-SAT optimisation using MapReduce on clouds', *Int. J. Innovative Computing and Applications*, Vol. X, No. Y, pp.xxx–xxx.

Biographical notes: Marcos Barreto received his MSc in 2016, and he is an Engineer (2010) in Computer Science from Universidad de la República, Uruguay. Currently, he is a Software Engineer at Facebook with more than ten year working in the industry in different technologies from backend services, web technologies and in the last five years, in Android. Prior to Facebook, worked more than four years in Mercadolibre.com, leading different teams participating in the product design, both from its technical perspective and its interaction to the end user.

Sergio Nesmachnow received his PhD in 2010, MSc in 2004, and he is an Engineer (2000) in Computer Science from Universidad de la República, Uruguay. He is a Full Professor at Universidad de la República and researcher at ANII and PEDECIBA, Uruguay. His main research areas are scientific computing, high performance computing, and parallel metaheuristics. He has published over 200 papers in international journals and conference proceedings (H-index 15, more than 1000). He serves as the Editor-in-Chief of *International Journal of Metaheuristics* (Inderscience), Guest Editor in *Cluster Computing* (Springer) and *The Computer Journal* (Oxford), and reviewer and technical program committee member of many journals and conferences.

Andrei Tchernykh received his PhD in 1986 from Institute of Precision Mechanics and Computer Engineering of the Russian Academy of Sciences, Moscow. He is a Full Professor at CICESE Research Center, Ensenada, Baja California, Mexico. He leads many research projects and grants in different countries. He has published approximately 200 papers, and served as a TPC member and General Co-Chair of more than 240 professional peer reviewed conferences. He is active in grid and cloud research with a focus on resource optimisation, both, theoretical and experimental, uncertainty, scheduling, load balancing, multi-objective optimisation, heuristics and meta-heuristics, adaptive resource allocation, and energy-aware algorithms and security.

1 Introduction

Nowadays, optimisation problems with large search spaces and/or many variables appear in important real-world application areas, such as artificial intelligence, operations research, bioinformatics, electronic commerce, etc. Prominent examples are finding shortest or cheapest round trips in graphs (related to smart city problems), energy-aware planning and renewable energy management, finding models of propositional formulas, determining the 3D-structure of proteins, and many others (Satchell, 2009).

Most combinatorial optimisation problems are within the NP-hard problem class, and the space of potential solutions is exponential in the size of a given problem instance (Korte and Vygen, 2007). The efficiency of most state-of-the-art optimisation algorithms deteriorates rapidly as the dimension of the search space increases. In order to find solutions in reasonable execution times different programming techniques must be used, which could be implemented in practice for real-world problems.

Optimisation algorithms have long been discussed in the literature, and many of them have been used as benchmarks to test different hardware infrastructures and/or algorithms performance (Chong and Zak, 2013). A basic, yet powerful idea for solving combinatorial optimisation problems is to apply an *iterative search approach* to 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, and can be performed in polynomial execution time.

Distributed computing comes to help researchers to face complex problems by applying a cooperative approach that proposes splitting a big problem into many smaller subproblems in order to speed up the search (Foster, 1995).

Last decade has witnessed an extreme growth of digital information, generated from many different sources: systems logs, web searches, online transactions, user actions like sending images or text message, online shopping, etc. With the advent of new technologies, any practical application must be able to scale up for handling datasets of interest. Analysing all these information is a powerful tool for any organisation, but at the same time, it is a difficult task. As a result, the community has paid a lot of interest in distributed computing for data analysis and other complex information processing problems (Zikopoulos and Eaton, 2011).

The MapReduce programming paradigm was conceived to easily implement algorithms to solve large scale problems in distributed computing systems. It provides a simple parallelisation model, resulting in shorter development times and easy-to-maintain distributed algorithms. MapReduce was designed to execute data intensive algorithms, and it has scarcely been applied for optimisation (Cohen, 2009).

Different frameworks have been proposed to analyse big amounts of information in distributed computing systems (clusters, grid, and cloud). Hadoop is one of the most well-known frameworks for analysing big data in distributed systems, providing fault tolerance, replication, and communication between processes (White, 2009).

This article presents our advances on applying a MapReduce approach for solving optimisation problems using Hadoop. The main advantages and limitations of the proposed strategy are presented and commented. After that, a concrete case study is reported, analysing several algorithmic approaches to solve the 3-SAT, a well-known version of the Boolean satisfiability problem (Hoos and Stutzle, 2004). We design a deterministic algorithm for solving the 3-SAT problem using a divide-and-conquer approach, based on how Hadoop splits the tasks in mappers and reducers. Afterwards, we propose a cooperative approach to deal with the challenges of designing a MapReduce algorithm for optimisation. Finally, we design three variants of the MapReduce 3-SAT solver and evaluate them to demonstrate that the collaborative approach is a promising option for solving optimisation problems in the cloud.

The manuscript is structured as follows. Distributed computing techniques, MapReduce and Hadoop are introduced in Section 2. An description of combinatorial optimisation problems and 3-SAT is presented in Section 3. Section 4 presents the proposed algorithms to solve the 3-SAT using divide-and-conquer, cumulative learning techniques as well as a randomised approach.

2 Distributed computing, MapReduce and Hadoop

Distributed computing is an umbrella term that defines a model and a set of programming techniques for solving problems by using a set of connected computing elements (Attiya and Welch, 2004). Distributed processes executing on those computing elements communicate and coordinate their actions to cooperatively solve a common problem. In the last 20 years, distributed computing has been applied to efficiently solve complex problems with large computing demands (Foster, 1995).

2.1 MapReduce

MapReduce is a programming model for processing large data sets in a distributed environment like clusters, grids and cloud computing systems. 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 (Dean and Ghemawat, 2008).

A MapReduce job has two main phases: the *map* phase and the *reduce* phase. The map phase has four steps: *record reader*, *mapper*, *combiner*, and *partitioner*. The output of the map phase is a set of intermediate 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 has four steps: *shuffle*, *sort*, *reduce*, *output format*.

2.2 Hadoop

Hadoop is currently the most used framework to analyse large volumes of information using the MapReduce programming model. Hadoop is a distributed system initially proposed as a framework to execute MapReduce tasks. It also includes an open source distributed file system implementation, called Hadoop distributed file system (HDFS) (Shafer et al., 2010).

Hadoop provides an abstraction level that allows implementing distributed algorithms easily, even for developers with little knowledge about distributed computing. These algorithms can be executed on hundreds of commodity computing resources to analyse huge amounts of information in reasonable execution times. Hadoop includes a full *ecosystem* that provides extended capabilities for task management distributed programming, database interfaces, and other features.

Hadoop was not designed to solve optimisation or computing intensive problems. Nevertheless, the framework provides the infrastructure to run MapReduce jobs, without dealing with classical issues of distributed systems (faulttolerance, communications, data replication, etc.). In this article, we propose adapting the MapReduce/Hadoop paradigm to solve optimisation problems.

2.3 Hadoop MapReduce implementation

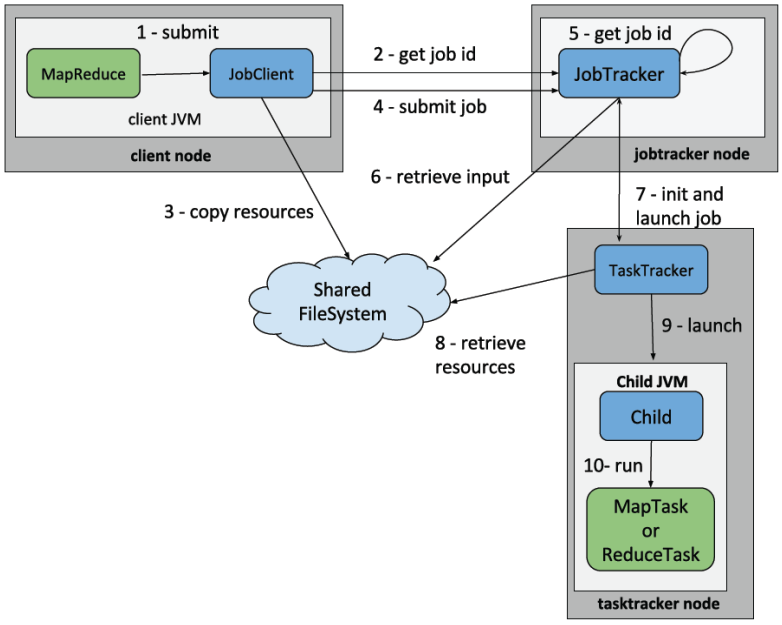
The most known and widely used implementation of the MapReduce programming model is the Hadoop implementation. Although the MapReduce paradigm is easy to understand and describe, it is not always easy to express an algorithm in terms of map and reduce functions.

In Hadoop, the input of the mapper is usually raw data saved in HDFS. The default format is *text input 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 map tasks and reduce tasks. The user can implement customs partitioners, record readers, input formats and combiners depending on the specific needs (White, 2009).

A Hadoop cluster includes a master node and multiple slave nodes. The master node has several processes: *JobTracker*, *TaskTracker*, *namenode*, and *datanode*. A slave node has datanode and TaskTracker processes. JobTracker coordinates all jobs 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.

Figure 1 shows the execution flow of a MapReduce job executed in Hadoop. First, a MapReduce job is created in the client node which is executed inside a Java Virtual Machine (JVM). The JobClient submits a new job to the JobTracker, which centralises all job executions and a new job identification is returned (step 2). After that, the execution file and distributed cache information needed for execution are copied to the nodes (step 3). Then, the job is submitted (step 4) and the JobTracker, using the job id (step 5) initialises the job and retries the input for the job (step 6). The JobTracker, assigns the job execution to a TaskTracker with availability to run the map or reduce task (step 7). The TaskTracker, retrieves the resources to run the task (step 8). Finally, the TaskTracker launches a new JVM (step 9) and executes the map or reduce task (step 9).

Figure 1 Execution flow of a mapreduce job in hadoop (see online version for colours)



3 Combinatorial optimisation and 3-SAT

This section introduces combinatorial optimisation and the 3-SAT problem, and reviews the main related works about algorithms to solve the 3-SAT.

3.1 Combinatorial optimisation

Combinatorial optimisation is a specific field of computing science that deals with finding a specific element (*solution*) within a finite set of elements (Korte and Vygen, 2007). The searched solution usually fulfils some specific properties or optimises (minimises/maximises) a given objective function. Combinatorial optimisation arises in many problems that model relevant real-world situations.

Many combinatorial problems are *decision* problems, where the solution for a given problem instance is specified by a set of logical conditions. The problem faced in this article, the propositional satisfiability problem (SAT), is a classical combinatorial decision problem. It 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. Several other combinatorial problems can be formulated as satisfiability problems (Cook, 1971). A candidate solution for SAT, satisfying the logical conditions, is called feasible or satisfable.

A fundamental difference between search/optimisation algorithms is on the way candidate solutions are generated, which can have a very significant impact on the theoretical properties and practical performance of the algorithm. A search algorithm can 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. On the other hand, in stochastic algorithms some decisions are performed randomly. In this case, if the algorithm fails to find a solution, there is no guarantee that the problem instance does not have a solution.

3.2 The 3-SAT problem

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, 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 propositional satisfiability problem restricted to clauses with k literals is k-SAT. For $k = 2$ the problem can be solved in polynomial time (Aspvall et al., 1979). The 3-SAT problem is a special case of k-SAT for which $k = 3$, i.e., each clause is limited to at most three literals. The k-SAT for $k > 3$ can be reduced to an instance of a 3-SAT.

Despite the simple formulation of the problem, the best known 3-SAT solver runs in exponential time. In fact, 3-SAT was the first problem demonstrated to be NP-Complete (Cook, 1971) and many other NP-complete polynomial time to an instance of those problems.

Thus, if a polynomial time algorithm to solve the 3-SAT 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 up to date.

The mathematical formulation of the 3-SAT problem is as follows. Consider the following elements:

- a set of *Boolean literals* $X = \{x_1,\ldots,x_n\}$, $x_r = \{0, 1\}$.
- A set of *clauses* $C = \{C_1,\ldots,C_m$". Each clause C_i is the conjunction of three elements, $C_i = \bigvee_{j=1}^{3} l_{ij}$, where l_{ij} is either a literal x_r or its negation $\neg x_r$, $r = 1,\ldots,n$
- A *Boolean expression* $\Phi = \bigwedge_{i=1}^{i=m} C_i$, formed by a set of m clauses from C.

The 3-SAT problem consists in determining a *truth assignment*, i.e., a set of values for the literals $\{x_1,\ldots,x_n\}$, that makes the Boolean expression Φ true, or guaranteeing that no such assignment exists.

The best known algorithms to solve the 3-SAT have exponential complexity. The main algorithmic proposals to solve the 3-SAT are reviewed in the next subsection.

3.3 Related work

This subsection reviews the related work on literature about deterministic and stochastic algorithms, both in sequential and parallel flavours, for solving the 3-SAT problem.

3.3.1 Exact 3-SAT solvers

The state-of-the-art exact solver for 3-SAT is Davis-Putnam-Logemann-Loveland (DPLL), a backtracking search algorithm introduced in Davis et al. (1962). DPPL starts by assigning a truth value to a literal, then simplifies the formula taking into account the previous truth assignment, and recursively checks satisfiability. If satisfiability is not fulfilled, the algorithm recursively checks using the opposite truth value for the considered literal. DPPL applies a divide-and-conquer approach, splitting the problem into simpler sub-problems to be solved recursively. The algorithm uses two rules at each step: unit propagation and pure literal elimination in order to avoid exploring naive parts of the search space. The recursion ends when all variables are assigned or all clauses are satisfied, and then the formula is satisfiable, or when one clause becomes empty, and then, the formula is not satisfiable.

Other state-of-the-art exact solvers are based on DPLL, such as the conflict analysis clause learning (CDCL) algorithm (Silva and Sakallah, 1996). This method improves over the traditional DPLL because it includes specific logic to learn new clauses by applying conflict analysis and non-chronological backtracking. The learned clauses are then used to guide the search.

DPLL and other exact algorithms are effective for solving small instances of the 3-SAT problems. However, they are 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.

3.3.2 *Stochastic 3-SAT solvers*

Stochastic algorithms (Motwani and Raghavan, 1995) are successful alternatives to address hard-to-solve problems, like 3-SAT and many other decision and optimisation problems. Among stochastic methods, heuristics and metaheuristics (Nesmachnow, 2014) have been widely used to solve hard problems from the real world.

The state-of-the-art metaheuristics to solve 3-SAT are variations of Schöning algorithm. Schöning is a randomised search that starts from a random state and performs a local search on a space of $3n$ elements. If no solution is found, another state is chosen and the process is repeated. Schöning runs in $O(1.33^n)$ for n clauses and succeeds with high probability to decide a 3-SAT formula after performing $t = k \times (4/3)n$ restarts. The probability of not finding a satisfying assignment using Schöning is at most e^{-k} (Schöning, 1999).

Several other metaheuristics have been proposed for 3-SAT, combining Schöning with exact methods. The PPSZ algorithm (Paturi et al., 1998) is a randomised search that runs in $O(1.3631^n)$ for n clauses and succeeds with high probability to decide 3-SAT. The randomised PPSZ with independent repetitions (Paturi et al., 2005) executes in $O(1.3633^n)$. Up to now, the current best algorithm to solve the 3-SAT is the PPSZ-based algorithm by Hertli (2011), which solves the 3-SAT in $O(1.3071^n)$ time.

3.3.3 *Parallel 3-SAT solvers*

Two main strategies have been proposed in the literature to solve large 3-SAT instances using parallel computing: *divide and conquer* and *portfolio*.

The divide and conquer approach is based on splitting the search space into subspaces, which are successively allocated to sequential SAT solvers. Cooperation between solvers is achieved mainly by two means:

1 by the application of load balancing strategies that dynamically transfer unprocessed subspaces to idle workers

2 through the exchange of learned clauses, which represent knowledge discovered during the search.

This approach has been extensively used in many SAT solvers (Huang and Darwiche, 2003; Deleau et al., 2008; Vander-Swalmen et al., 2009; Hyvärinen et al., 2010; Barreto et al., 2011). A description of the divide and conquer approach is presented in Figure 2.

The portfolio approach uses several sequential SAT solvers that compete and cooperate solving the same formula.

As the solvers run independently and process the whole search space, there is no need for load balancing overheads.

Cooperation between solvers is achieved by exchanging learned clauses. Portfolio solvers became prominent in the related literature since 2008. Several well-known portfolio SAT solvers have been proposed in recent years, including ManySAT (Hamadi and Sais, 2009), Plingeling (Biere, 2010) and SArTagnan (Kottler and Kaufmann, 2011). A description of the portfolio approach is presented in Figure 3.

Figure 2 Divide-and-conquer approach for parallel 3-sat solvers (see online version for colours)

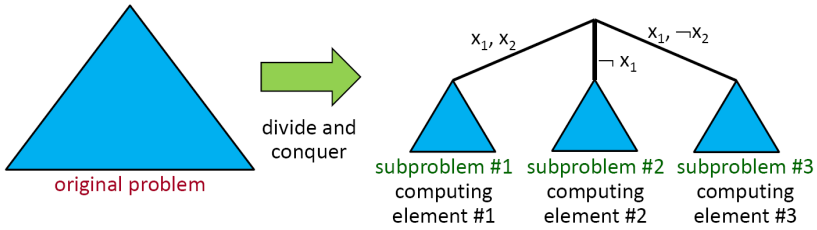
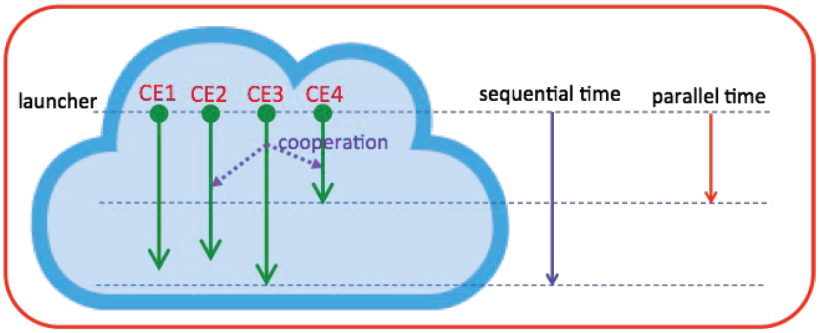


Figure 3 Portfolio approaches for parallel 3-sat solvers (see online version for colours)



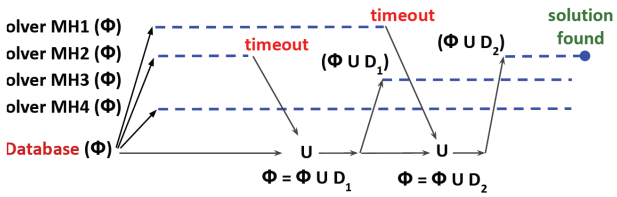
Other parallel algorithms for 3-SAT use incomplete local search solvers, such as a previous parallel quantum search algorithm combining a parallel Schöning, a local search, and a quantum search (Barreto et al., 2011). Another example is GPU4SAT (Deleau et al., 2008), that considers the number of satisfied clauses as a quality function, and uses a local search to evaluate neighbourhoods, looking for the best literal to flip.

3.3.4 Parallel 3-SAT solvers in grid and cloud

Specific issues of the infrastructure must be considered for 3-SAT parallel algorithms on a grid or cloud system. Regarding cooperation, the communication to/from running processes is extremely time-expensive in a distributed grid/cloud. In addition, the distributed platform may impose resource limitations, e.g., a predefined time limit for the solver execution.

Hyvärinen et al. (2009) proposed storing learned clauses on a *master database*, to deal with cooperation issues on a distributed platform. The database is used for clause sharing only when a solver starts: the solver imports a part *D* of the database permanently into its instance [i.e., it solves (Φ & D) instead of Φ], and when a solver *timeout* is detected, then (a subset of) the current learned clause is merged into the database. This approach is known as *cumulative parallel learning with hard restarting solvers* (see Figure 4).

Figure 4 Problem split with collaboration using clause database (see online version for colours)



Several works have proposed using grids and networks of commodity computers for solving the 3-SAT problem.

GridSAT (Chrabakh and Wolski, 2003) is a distributed 3-SAT solver based on a variation of the DPLL algorithm using conflict clause and clause sharing. It is implemented for a grid of heterogeneous commodity computers, optimising the use of resources by applying a scheduling technique that request resources on demand. GridSAT uses a divide and conquer approach sending different search spaces to several solvers, improving the performance and also solving problem instances that a baseline algorithm fails to find.

PaMiraXT (Schubert et al., 2009) applies a domain decomposition using the MPI library for parallel computing. Other 3-SAT solvers are conceived to be executed on volunteer grid platforms (Schulz and Blochinger, 2010; Posypkin et al., 2012), where the computing nodes are volatile and they are heterogeneous, both in hardware and software. These parallel implementations focus on scheduling and failure tolerance to support changes on the infrastructure (e.g., a node joins or leaves the volunteer platform).

HordeSat (Balyo et al., 2015) is a portfolio-based algorithm for solving 3-SAT in the cloud, executed in a cluster of commodity computers. HordeSat applies a master-slave paradigm for communication, implemented using MPI. Each solver runs a different algorithm in parallel and communicate with the master process to share learned clauses. Significant speedup improvements are reported for hard 3-SAT instances when running on a cluster with up to 2,048 cores.

The review of related works indicates that no previous proposals have been presented to develop a 3-SAT parallel solver using MapReduce and Hadoop. Our proposal is to develop an hybrid algorithm that combines a divide and conquer approach to split the problem in small subproblems and a cooperative portfolio approach applying MapReduce in cloud systems. This idea improves over previous distributed approaches for grid and cloud systems (Schulz and Blochinger, 2010; Posypkin et al., 2012), by including a novel approach using Hadoop to manage the distributed computing infrastructure, seemingly handling failure tolerance, resource managing and communications between nodes.

4 MapReduce 3-SAT solvers in Hadoop

The proposed solvers were designed following an incremental approach, from a simple solver to more complex ones. In each step, different improvements are included to deal with specific issues of 3-SAT by applying the MapReduce paradigm. This way, we explore the capabilities of Hadoop and MapReduce for solving optimisation problems.

4.1 3-SAT solver in Hadoop

The baseline 3-SAT solver applies a divide and conquer approach and portfolio with and without cooperation. HDFS is used to provide support for cooperation.

A domain decomposition technique is used to split the problem with a master-slave approach. The main MapReduce job is the master process, which splits the domain in subproblems, to be processed by mapper and reducers processes. The domain is decomposed by fixing *d* literals, so each mapper searches for solutions in different (exclusive) search spaces, and two mappers cannot find the same solution to the problem. For this, all mappers and reducers must select the literals to fix in the same manner and order.

The domain is split on different mappers by writing a file to HDFS, defining a subproblem in each line. The problem is divided by fixing *d* literals. For example, let the formula with three clauses defined in Figure 5(a), the file written to HDFS with the problem split, using x_1, x_2 as literals to fix, is shown in Figure 5(b). Using this problem decomposition, each mapper receives one line describing a 3-SAT subproblem.

Figure 5 Example of 3-sat problem instance and representation, (a) 3-sat problem instance (b) file written to HDFS

$$x_1 \vee x_2 \vee x_6$$
$$\neg x_1 \vee x_4 \vee x_3$$
$$\neg x_3 \vee x_1 \vee x_7$$

(a)

1	2
-1	2
1	-2
-1	-2

(b)

Two strategies were applied for selecting the literals to fix: by order (literal 1 to literal *n*), and based on the frequency of appearance (literals that appear more frequently in the clauses are selected first), focusing on the biggest subproblems first.

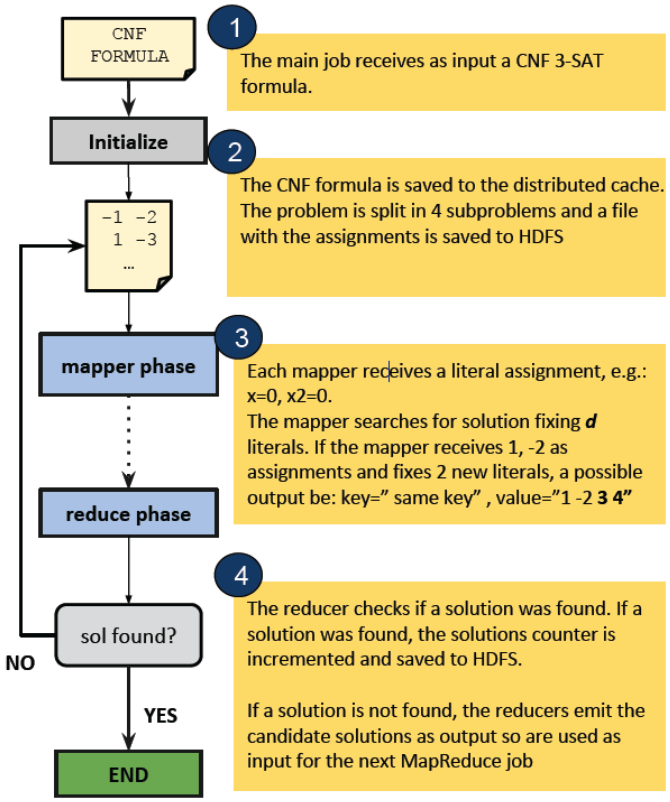
The main MapReduce job is described in the diagram in Figure 6. The job receives as input an HDFS file defining the formula in CNF form. The initialisation process uploads the CNF formula received as input to the distributed cache, so it will be available to all mappers and reducers in execution. Domain decomposition is also performed in the initialisation, by setting *r* literals. The number of splits is controlled in the first devised algorithms by an input parameter and after that it is optimised (and calculated in the MapReduce job) to improve memory consumption and performance.

After the initialisation process, the MapReduce job is launched using as input format the `NLineInputFormat` with $N = 1$ so that, in the example described above, four mappers are created, receiving one subproblem definition each. Depending on the algorithm, different methods are used in the mapper to search for candidate solutions. 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}` (in the randomised approach described in Section 4.6, the solution is written to HBase) 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.

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

The Hadoop framework includes two special functions for MapReduce (*setup* and *cleanup*), where initialisation and cleanup can be performed in both the map and in the reduce phase. The setup and cleanup methods are executed once by each mapper and reducer process at the start and at the end of the process respectively. We use the setup method in our different proposed algorithms to either 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 proposed algorithms.

Figure 6 General structure for the 3-sat mapreduce solver (see online version for colours)



In the proposed approach, the main MapReduce job splits the problem using a constructive strategy, by fixing d initial literals. Then, a depth first search (DFS) algorithm is used to search inside the mappers. Applying a DFS to split the problem among mappers seems to be a better strategy for a deterministic 3-SAT solver. MapReduce has been proposed to search on large scale graphs (Cohen, 2009), but the model is not appropriate for DFS algorithms, because the mappers cannot communicate between each other.

In a straightforward application of the proposed MapReduce algorithm to solve 3-SAT, d literals are fixed in each iteration, thus generating 2^d possible candidate solutions. If the number of mappers is not limited, the algorithm would generate 2^d mappers (using a `NLineInputFormat` with one line per mapper). This will certainly result in out-of-memory errors, because of the creation of too many mappers. Moreover, creating one mapper for each possible assignment is not an efficient option, because Hadoop creates a new Java Virtual Machine process for each task, thus producing a significant overhead in both CPU and memory. In the proposed algorithms, the number of mappers is limited by a given number specified as an input parameter, in order to avoid out-of-memory errors.

When a solution is found in a mapper, the standard MapReduce algorithm cannot finish execution until all mappers have processed the assigned data. This is a big inconvenient for search algorithms (especially for large problem instances), because it means that a lot of unnecessary processing (of thousands of possibilities) must be performed after a solution is found in a given mapper.

Some other performance challenges when solving optimisation problems with MapReduce must be taken into account. All of them are related to specific features and design limitations of the Hadoop framework and the MapReduce paradigm itself:

- Mappers and reducers work using (key,value) pairs, so all information must be translated to that 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.
- Mappers do not interact with each other. As a consequence, 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 managed carefully. Load balancing issues when handling intermediate keys could significantly impact in the performance of the resulting algorithm.

We propose a number of different algorithms and strategies to deal with these main issues. The proposed models are described next.

4.2 Non-cooperative domain decomposition

Non-cooperative domain decomposition (NDD) applies a domain decomposition and a perturbation search algorithm that works fixing d literals in each iteration, and exploring new candidate solutions with different combinations for the other literals (see a description in Algorithm 1). An example of the NDD search is described in Figure 7.

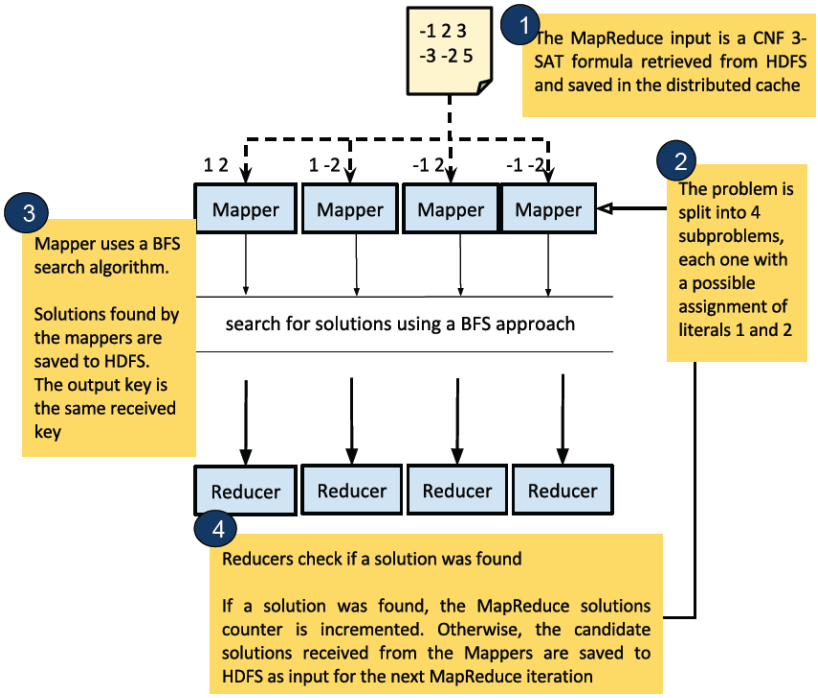
The NDD algorithm performs the domain decomposition, assigning different combination of truth values to mappers. Each mapper receives a literal assignment, e.g., $x_1 = 0$, $x_2 = 0$. Then, new candidate solutions are created by including the fixed values and different combinations for the other literals. Candidate solutions are evaluated: if a given assignment makes a clause false, all candidate solutions having that assignment are marked as invalid; otherwise, if the candidate solution is valid, the mapper emits a (key, value) pair, where the key is the same received as input and the value is the candidate solution found. This procedure continues until exploring all possible combinations for non-fixed literals that define that search subspace, following a breath first search (BFS) approach, and it is described in Algorithm 2.

Algorithm 1 3-SAT MapReduce job in NDD

```
1:  job ← createJob()                    ▷ initial job, split problem in
                                           2 × max_mappers subproblems
2:  job.run job                          ▷ create max_mappers mappers
3:  while solution not found and not finish do
4:    if solution found then
5:      finish (result: TRUE, store solution in HDFS)
6:    else
7:      if all literals have been fixed then
8:        finish (result: FALSE, no solution)
9:      else
10:       new input ← job.output   ▷ get previous job results
11:       s n ← job.get number of subproblems()
12:       new job ← createJob()   ▷ create new MapReduce job
13:       set number of mappers(new job, s n)
14:       setJobIO(new job, new input, new output)
15:       new job.run job
16:     end if
17:   end if
18: end while
```

The NDD reducer checks if each received candidate solution is a solution to the problem. Since the key emitted by each mapper is the same key received as input, the number of reducers is the same as the number of mappers. A drawback of this approach is that NDD suffers of severe load imbalance in the reducers (i.e., a reducer may receive many values for a key, and other reducer could receive few of them). Nevertheless, in NDD the reducers only prepare the information for the next iteration, setting the received values as output of the MapReduce algorithm.

Figure 7 3-SAT MapReduce – NDD approach (see online version for colours)



Algorithm 2 3-SAT mapper in NDD

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

Using this non-cooperative domain decomposition approach, the resulting mapper algorithm is very inefficient. Having no communications avoid performing any pruning on the search space when finding a literal assignment that generates invalid solutions, which is very useful to improve the efficiency. In consequence, the whole search space must be explored (e.g., if each mapper fixes 15 literals, then it will explore 2^{15} possible assignments). This approach do not scale for solving large 3-SAT problems, and even when dealing with small problems the algorithm demands significantly large execution times. Furthermore, as all mappers are independent and do not collaborate, in case one mapper finds a solution, the MapReduce algorithm do not end until all MapReduce executions are performed.

NDD is not efficient in memory consumption, reaching memory limits on a 16 GB RAM Intel server for even small problems with more than 30 literals. If the main job splits the problem into four subproblems and each mapper tries to fix 15 literals, each mapper generates 215 = 32,768 assignments and after the first iteration 4 × 32.768 = 131.072 subproblems are generated. The second iteration would generate 131.072 mappers, which easily causes out-of-memory errors.

4.3 Non-cooperative depth first search with partial assignments

Non-cooperative depth first search with partial assignments (NDFS-PA) is a first improvement over NDD that proposes improving the search in the mappers. In addition, NDFS-PA also proposes reducing the memory utilisation of the whole MapReduce process by reducing the number of intermediate output keys and values. NDFS-PA is described in Figure 8.

The mappers applies a DFS search with pruning, instead of a perturbation search as in NDD. Each time a candidate solution is invalid (e.g., it contains an invalid assignment, thus it cannot generate a valid solution for the problem), the whole branch is pruned, avoiding searching in branches where there are no solutions, as shown in the example of Figure 9.

Figure 8 3-SAT MapReduce – non-cooperative depth first search with partial assignments (see online version for colours)

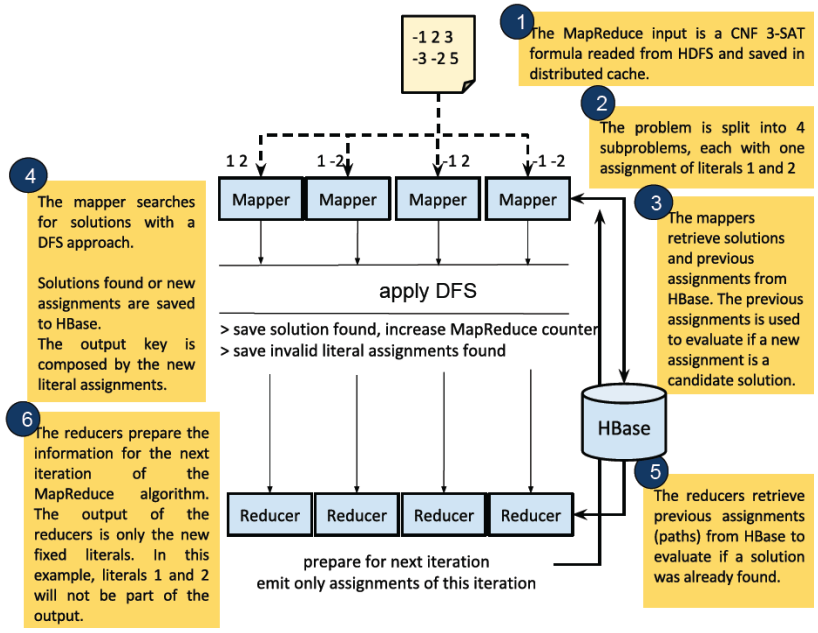


Figure 9 3-SAT MAPREDUCE – example of the DFS search in the mappers (literals 1 and 2 already set) (see online version for colours)

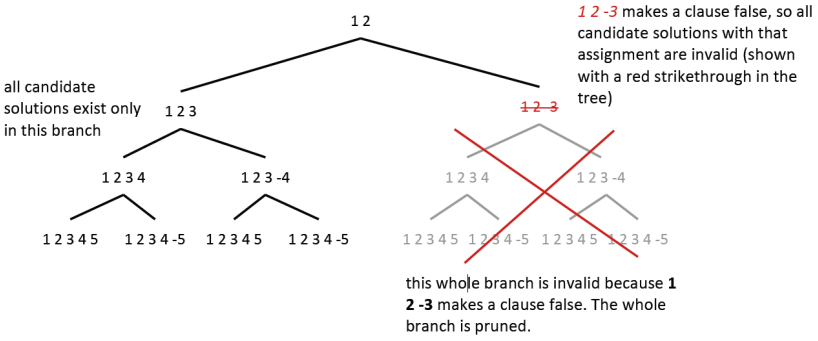
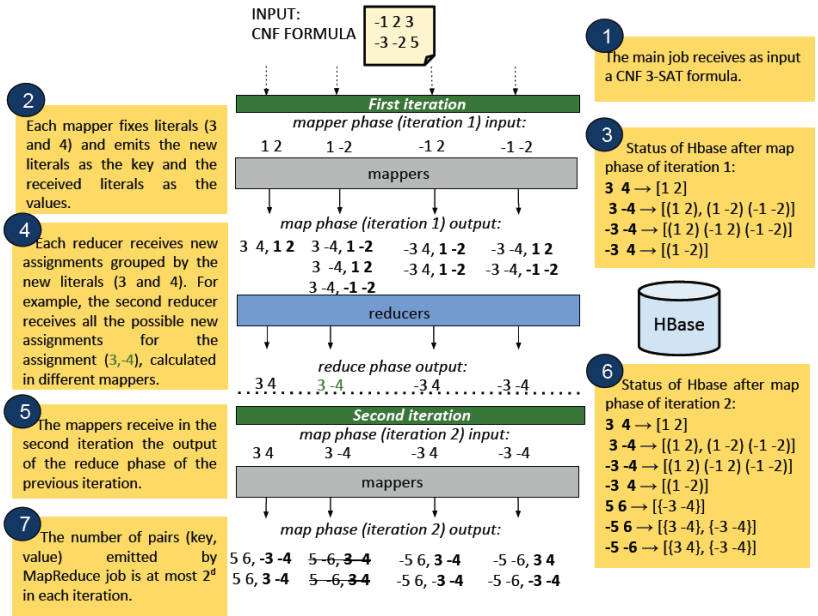


Figure 10 3SAT MapReduce – NDFS-PA example of invalid literals (see online version for colours)



NDFS-PA tackles out-of-memory issues by reducing the number of candidate solutions the mapper process emits. In each iteration of the MapReduce algorithm, the mapper fixes *d* new literals and outputs previously and new fixed literals. Using this grouping, the total number of new candidate solutions emitted in each iteration of the MapReduce job is at least 2^d.

NDFS-PA uses HBase (instead of HDFS) to centralise the literals that each mapper fixes in each iteration. The table schema defined in HBase has one table with one column family `path` and one row for the literal assignment received; the value is the list of the new fixed literals. Figure 10 shows an example of how the literals assignments are saved in HBase after two iterations of the MapReduce algorithm.

NDFS-PA exponentially reduces the number of intermediate keys when compared with NDD. Suppose *d* literals are fixed in each iteration and after two iterations, all candidate solutions are valid. The original NDD algorithm generates up to $2^d \times 2^d$ possible candidate solutions, creating $2^d \times 2^d$ processes. On the other hand, NDFS-PA only generates 2^d intermediate keys and $2^d \times 2^d$ processes.

Even though NDFS-PA significantly reduces the number of intermediate assignments, for medium problems (more than 30 literals and 150 clauses) an out-of-memory error is still easily reached in an Intel server with 16 GB of RAM. Moreover, NDFS-PA has a considerable overhead due to saving and retrieving the clause assignments to HBase in each iteration, in every mapper and reducer.

Regarding HBase queries, in the example on Figure 10 the mapper input for the third iteration has 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 that, the database is queried again to retrieve the remaining previous assignments: the literal assignment (1 2) is retrieved using (3, 4) as key and the literal assignments (1 2), (–1 2), (–1 –2) are retrieved using (–3, –4) as key. Finally, for each retrieved assignment, HBase is queried again to check if a full candidate solution can be rebuilt. In this last step, HBase returns no data, so mappers and reducers know they have all information to reconstruct the full candidate solution. In the example, each mapper and reducer on the third iteration has to query HBase six times.

The NDFS-PA 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.

4.4 Non-cooperative depth first search with recalculated mappers

Non-cooperative depth first search with recalculated mappers (NDFS-RM) is a variation of NDFS-PA that reverts the approach of emitting as output only the set of literals in each iteration. Instead, NDFS-RM outputs all the assignments, but recalculates the number of mappers to be created, based on the candidate solutions found in the previous iteration of the MapReduce algorithm. NDFS-RM does not limit the number of mappers by reducing the number of assignments, but fixes the number of mappers by changing the number of lines (i.e., problems) each mapper receives.

In Hadoop, the number of mappers created 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, used to split the problem in 2 × *max_mappers* subproblems, fixing *r* literals, according to $r = \frac{\log(2 \times max\_mappers)}{\log(2)}$.

In order to have *max_mappers* processes, the number of lines *y* each mapper receives in each iteration is calculated by $y = \frac{number\_of\_lines\_input\_file}{max\_mappers}$. Then, the `NLineInputFormat` input format is used taking into account that number of lines.

Another improvement included in NDFS-RM it that the literals to fix are selected based on its frequency of appearance in the formula. This strategy allows improving the pruning of branches, because selecting literals that appear more frequently in different clauses increase the probability of selecting an invalid candidate solution.

The mappers in NDFS-RM applies a recursive DFS search. The search finishes when a literal assignment for both for true and false values makes at least one clause false; in that case, the branch is pruned. When the mapper succeeds in fixing *d* literals, then a key-value pair is written, using as key the mapper key received as input (byte offset) and the value is the concatenation of the fixed literals received and the new fixed ones. The search is described in Algorithm 3.

Algorithm 3 3-SAT NDFS-RM mapper

input: *depth* to use in DFS, mapper *key*, mapper *value*, *new_fixed_lit*: new fixed literals

1: *prob. definition* ← *value*
2: **if** *depth* = = 0 **then**
3: *key* = *input key*,
4: *value* = *prob. definition* + *new_fixed_literals*
5: emit (*key*, *value*)
6: **else** ▷ [get next literal to fix, in order, no repetition]
7: *lit* = *get_literal_to_fix*(*value* + *new_fixed_lit*)
8: **if** satisfiable(*lit* + *value* + *new_fixed_lit*) **then**
9: NDFS-RM mapper(*depth* – 1,*value*,*lit* + *new_fixed_lit*)
10: **end if** ▷ check if literal is false is a possible assignment
11: **if** satisfiable(–lit + value + new fixed literals) **then**
12: NDFS-RM mapper(*depth* – 1,*value*,-*lit* + *new_fixed_lit*)
13: **end if**
14: **end if**

The reducer 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, the received candidate solutions are emitted to be used as input for the next MapReduce

iteration. The number of reducers is the same as the number of mappers, controlled in Hadoop using function *job.setNumReduceTasks(int)*. The reduction is described in Algorithm 4.

Algorithm 4 3-SAT NDFS-RM 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 found solutions
6:            end if
7:        else
8:            emit null, value
9:        end if
10:   end for
```

4.5 Cumulative learning technique

The next step in the incremental approach to design an effective 3-SAT solver using MapReduce in Hadoop was including collaboration between mappers. To implement such collaborative approach, we propose applying a cumulative learning technique, 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 a set of solutions already found in the search, or a set of all invalid literals assignments that make at least one clause false (see a description in Algorithm 5). 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 search. When a solution is found, it is send and stored in HBase. This way, that solution will be available to all other mappers and reducers to speed up their search.

Algorithm 5 Mapper setup method for collaborative learning

```
      setup method:
1:    solution ← search solution in HBase
2:    if solution not found then
3:        invalid literals ← get invalid literals from HBase
4:    end if
      mapper method:
5:    if solution found then
6:        end mapper code
7:    else
8:        run NDFS-RM mapper code
9:    end if
```

Two cooperative strategies were devised: collaborative with invalid literals (CID) and collaborative without invalid literals (CnoID). Both CID and CnoID share a common algorithmic structure, described in Figure 11. The main difference between them is that CID uses the master database for storing and sharing invalid literals and solutions found, while CnoID uses the master database only for storing and sharing solutions found in the search. In step 3, CnoID does not use or retrieve invalid literal assignments, while the setup method in CID (called once for each mapper when it is created), retrieves from HBase invalid literal assignments found in the search. In step 4, CnoID does not send invalid literal assignments found in the search, since they are not used in this variant of the algorithm. On the other hand, CID saves invalid literal assignments found in the execution in the cleanup method, called once when the mapper ends.

4.6 Randomised MapReduce 3-SAT algorithm

Our final approach to improve the search of the MapReduce 3-SAT solver consists in including randomisation in the search pattern, as proposed in previous 3-SAT algorithms (Schöning, 1999; Barreto et al., 2011).

The randomised MapReduce 3-SAT algorithm (Rand-MR) takes advantage of the available processing power to include a search in both mappers and reducers. In addition, Rand-MR hybridises that parallel search with a variant of the well known deterministic DPLL, and the non-deterministic Schöning algorithm. The motivation of designing this hybrid implementation is that, with some probability, a solution can be found in any iteration of the algorithm.

Rand-MR applies a domain decomposition of the search space, splitting the problem in $2 \times number\_of\_mappers$ subproblems. After that, reducers apply unit propagation pure literal elimination (UPPLE) (Davis et al., 1962) to simplify the formula based on fixed literals. For each partial solution a new formula is generated and saved in HBase.

Both mappers and reducers retrieve from HBase the formula associated to the partial assignment received, or they use a default formula, initially saved by the main MapReduce job. Each time a mapper or reducer finds a solution, or a new formula is generated by UPPLE, the corresponding information is saved to HBase. HDFS is only used to save intermediate keys and values used by the MapReduce engine in Hadoop; this information is shared between mappers, reducers, and processes in consecutive iterations of the algorithm. Using HBase for storing all data related to the algorithm represent a clear improvement over using HDFS: small pieces of information are read and written, thus a fast access must be provided in a write-many read-many pattern. The mapper and reducer in Rand-MR are described next.

Figure 11 Basic structure of cid and cnoid algorithms (see online version for colours)

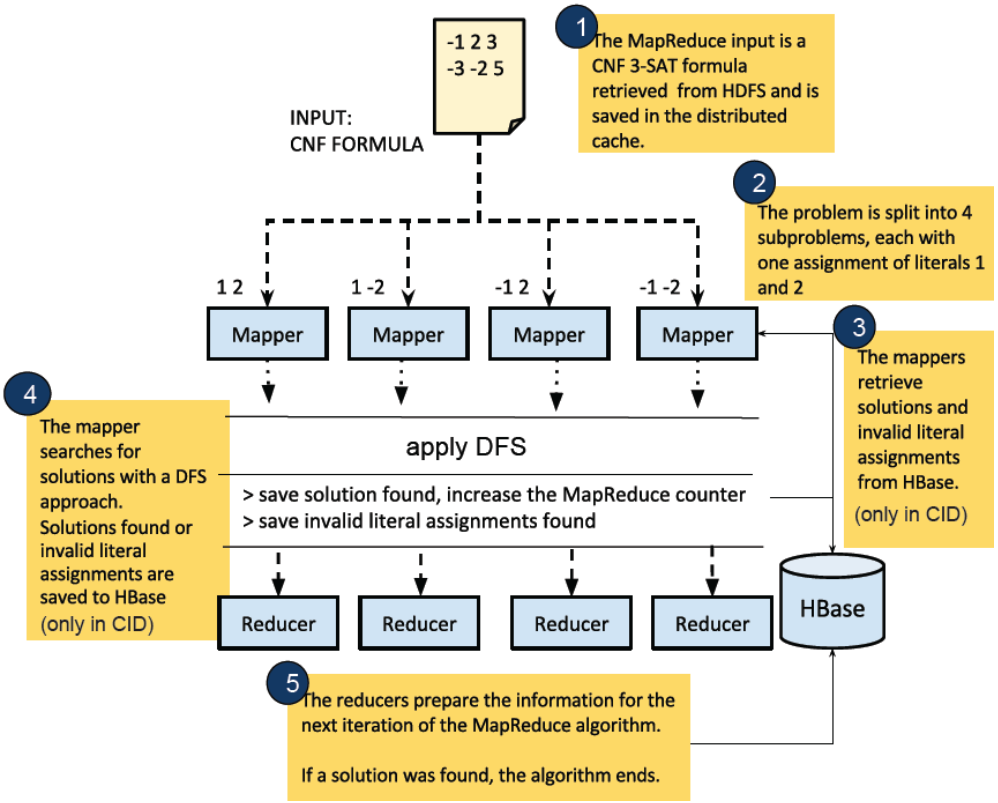


Figure 12 Diagram of the mapper process in Rand-MR (see online version for colours)

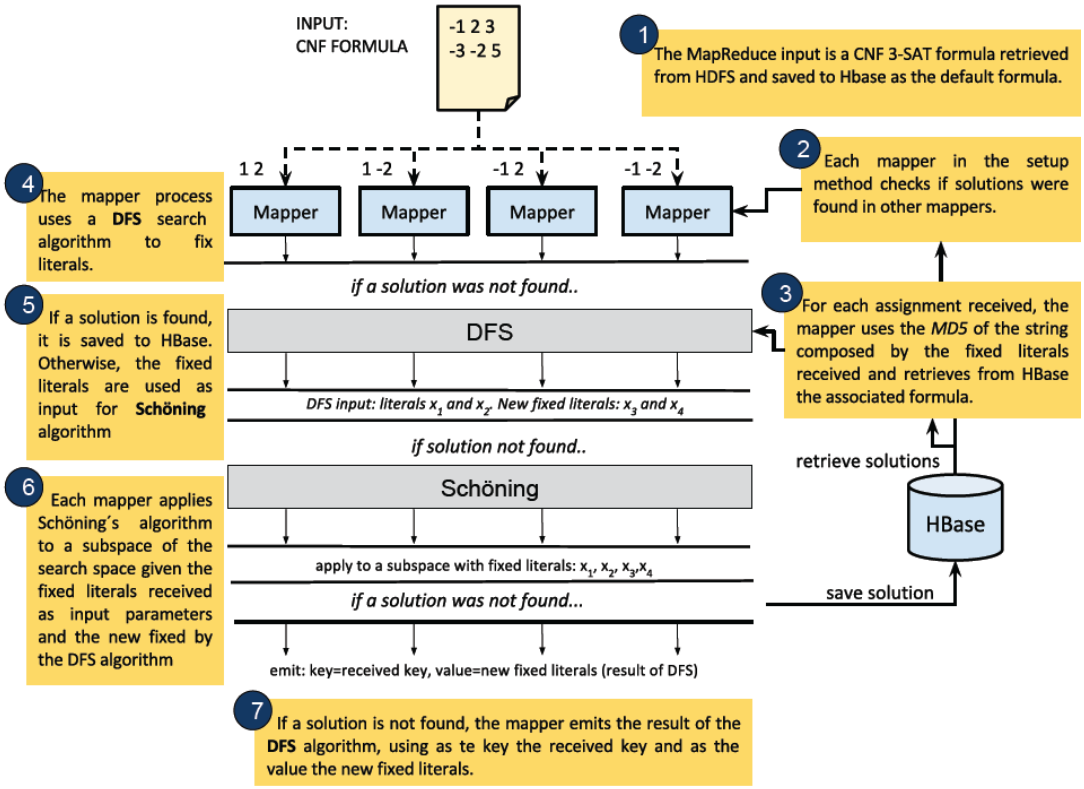


Figure 13 Example of the search approach used in the mappers in Rand-MR (see online version for colours)

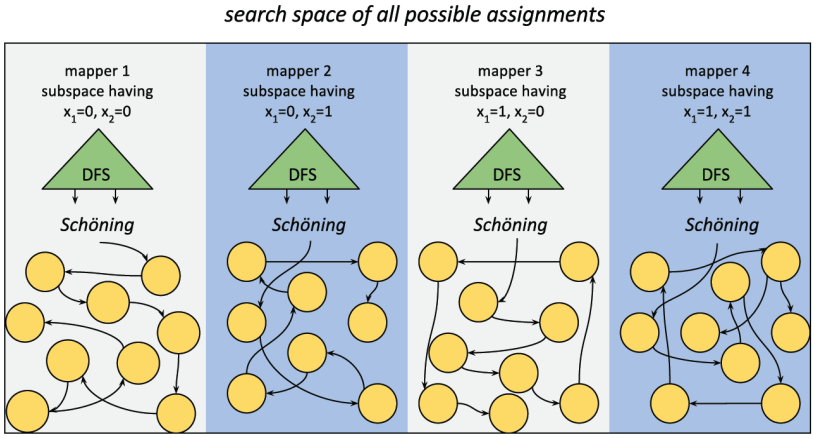
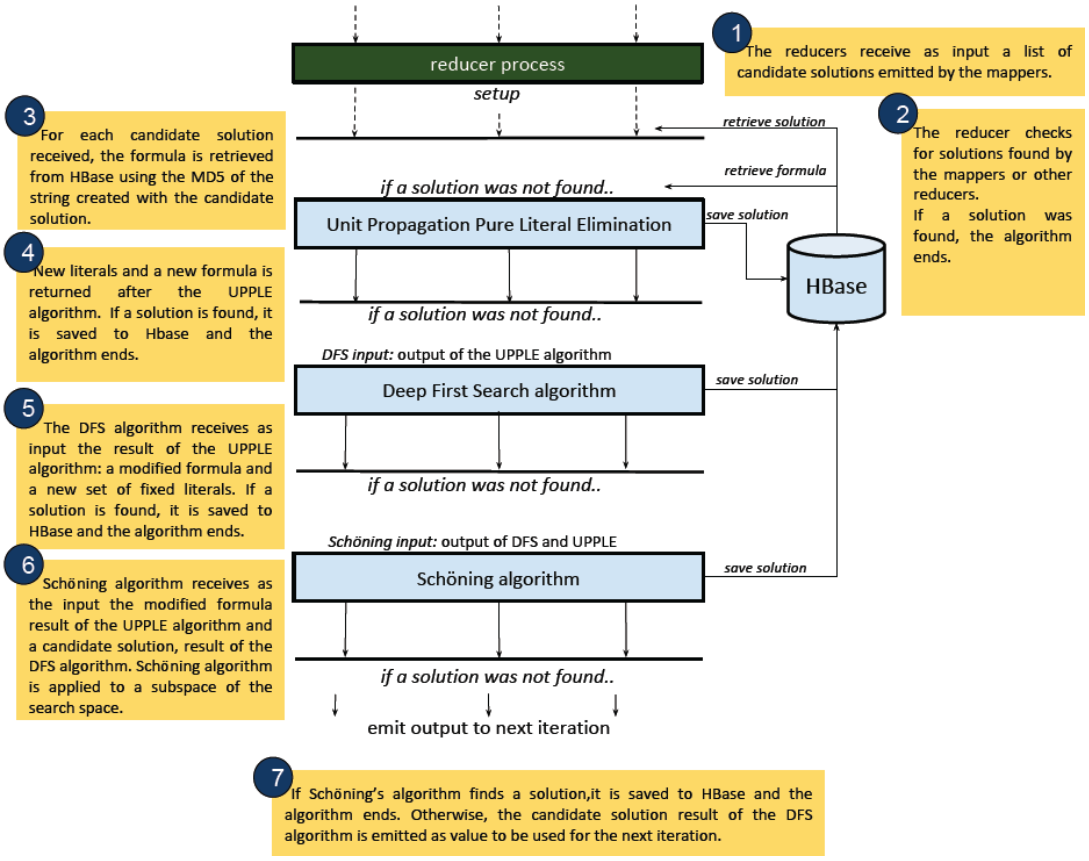


Figure 14 Diagram of the reducer process in Rand-MR (see online version for colours)



4.6.1 Rand-MR mapper for 3-SAT

The master follows the general structure described in Algorithm 3, but includes an important feature to improve the search: the Schöning algorithm is performed in selected subspaces of solution space, defined by the literals fixed in the DFS search. Figure 12 presents a description of the Rand-MR mapper, which applies six steps:

1 The main MapReduce job saves the retrieved formula from HDFS to HBase; this way, the formula will be available to all mappers and reducers.

2 The setup method is called once for each mapper and reducer; each process checks if a solution was already found in a previous iteration.

3 For each candidate solution received by the mapper, the Hadoop MD5 hash function is applied to the string composed by the fixed literals as a key to retrieve the corresponding formula. This step is needed because each time the UPPLE algorithm changes the formula based on the current assignments. If no formula is found in HBase for such MD5 hash, the default formula is returned.

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.

5 In case that a solution is found, it is saved to HBase; otherwise, the fixed literals are used as input of the Schöning's algorithm.

6 Schöning's algorithm is applied in a subspace of the search space, defined using the fixed literals received and the new fixed literals result of the DFS algorithm. If a solution is found, then it is saved to HBase.

The Rand-MR mapper is described in Algorithm 6 and Figure 13 presents a graphical representation of the search applied in its subspace (gray/blue areas).

The mapper first applies DFS for recalculated mappers in the subspace defined by the fixed literals received as input. After that, if the DFS search does not fix all literals, the Schöning algorithm is executed for each possible partial solution (i.e., the partial assignment is used as input). These partial solutions define a subspace of the entire solution space, taking into account the literals set by the DFS search.

Schöning is not deterministic. It performs a local search (represented in yellow in Figure 13) in each execution, exploring a neighborhood of the current solution and jumps to a different region of the solution space of each mapper to search for new solutions. Each Schöning execution explores different regions of the search space and can succeed in finding a solution at any time (with a given probability).

If Schöning fails to find a solution to the problem, the mapper outputs a pair (key, value), where key is the same key originally received by the mapper and the value are the literals set by the DFS algorithm.

4.6.2 Rand-MR reducer for 3-SAT

In Rand-MR, the reducer plays a more significant role than in previous variants. Each reducer prepares the information for the next MapReduce iteration (as in previous variants), but it also searches for solutions using DFS, UPPLE, and Schöning. This way, the hybrid search capabilities incorporated in Rand-MR are applied in two phases of the algorithm, and working over different solution spaces. The number of reducers is the same as the number of mappers, as in previous variants. Algorithm 7 presents the Rand-MR reducer and Figure 14 describes the search.

The reducer works according to the following steps:

1 Each reducer receives a list of candidate solutions emitted by the Rand-MR mappers.

2 In the setup method, the Rand-MR reducer checks Hbase to determine if a solution was already found. If a solution was already found, the algorithm ends.

3 Otherwise, the Rand-MR reducer retrieves from HBase the corresponding 3-SAT formula for each assignment received. The MD5 hash of the candidate solution is used. If no formula is found, the reducer uses the formula received as an input parameter by the MapReduce job, which was saved in Hbase in the initialisation.

Algorithm 6 Mapper process in Rand-MR

```
    input: (received key: LongWritable, value: Text)
1:  if solution not found then
2:      formula = retrieve formula from Hbase (MD5(value))
3:      possible solutions = apply DFS(value, formula)
4:      for all s ← possible solutions do
5:          if s is solution then
6:              save solution to HBbase(s)
7:          else
8:              schoning result = apply Schöning algorithm (s)
9:              if schoning result is solution then
10:                 save solution to Hbase (schoning result)
11:             else
12:                 emit: key=received key, value=s
13:             end if
14:         end if
15      end for
16: end if
```

4 UPPLE is applied, fixing new literals and generating a new assignment (candidate solution) and a new formula.

5 If not all literals are set in the new assignment produced by UPPLE, DFS is applied to the new assignment and to the new formula generated by the UPPLE algorithm.

6 If DFS finds a solution, it is saved in HBase and the algorithm ends. Otherwise, Schöning is applied in a subspace of the solution space where the literals in the received candidate solution are already fixed. Schöning uses the UPPLE result as the formula to evaluate if a new candidate solution solves the original 3-SAT problem.

7 In case the reducer fails to find a solution, the assignment returned by the DFS algorithm is saved in HDFS and emitted as output. In addition, the MD5 hash value is used to save the formula returned by the UPPLE algorithm to HBase, in order to prepare the information for the next iteration of the MapReduce algorithm.

5 Experimental analysis

This section presents the experimental analysis of the proposed MapReduce algorithms for 3-SAT.

Algorithm 7 Rand-MR reducer

```
    input: (key: LongWritable, possible sols: List[Text])
 1: if solution not found then
 2:   for all s ∈ possible solutions do
 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                ▷ Apply UPPLE
 9:       new formula, new assignment = apply UPPLE(s)
10:       if all literals are set in new assignment then
11:         if new assignment is solution then
12:           save new assignment to HBASE
13:         end if
14:       else              ▷ iterate over all new possible solutions
15          new assignments = apply DFS(new assignment)
16:         for all s1 ∈ new assignments do
17            if all literals are set in s1 then
18:             save solution s1 to HBASE
19:           else
20:             sch1 = apply Schöning algorithm (s1)
21:             if sch1 is solution then
22:               save solution sch1 to HBASE
23:             else                ▷ emit (empty key, possible sols)
24:               emit: key=NullWritable, value= s1
25:               save new formula to HBASE with key MD5(s1)
26:             end if
27:           end if
28:         end for
29:       end if
30:     end if
31:   end for
32: end if
```

5.1 Computational platform and software

All algorithms were implemented in Scala 2.10 (Odersky et al., 2008) using Hadoop 2.2 and HBase 0.98.22. The software developed is publicly available at https://github.com/marbarfa/mapreduce_sat/releases/tag/v3.0.

The experimental analysis was performed in AMD Opteron 6272 servers (24 cores at 2.09GHz, with 72GB RAM) from from Cluster FING (Nesmachnow, 2010) using HBase with its default semi-distributed configuration.

5.2 Methodology for the experimental evaluation

The main goal of the study was to analyse the computational efficiency of the proposed algorithms and their scalability when solving problem instances with increasing dimension.

The analysis studies the execution time of each version of the algorithm when solving different problem instances. For the deterministic algorithms (NDD, NDFS-PA, CID, and CnoID), ten executions were performed, to reduce the impact of possible variations regarding load balancing and the availability of the computational platform. On the other hand, taking into account that the Rand-MR algorithm is not deterministic, 30 independent executions were performed for each problem instance solved. Rand-MR is not deterministic because of the application of Schöning algorithm, so the execution times of Rand-MR vary depending on the probabilistic decisions taken in the Schöning search. In each case, we evaluate and report the average of the execution time, and the standard deviation computed on the 30 independent executions performed by each problem instance.

5.3 3-SAT instances

A set of standard 3-SAT problem instances, considering different (increasing) values for both the number of literals and the number of clauses, were solved.

Many empirical studies of SAT solvers have used randomly generated CNF formulas (Selman et al., 1992). In most cases, instances are built using a random generator that samples SAT instances using 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 (Mitchell et al., 1992).

Some of the most widely studied randomly generated SAT instances are 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 2^n literals with a fixed probability (Franco and Paull, 1983). Theoretical and empirical results show that this family of instances is easy to solve (on average), using rather simple deterministic algorithms (Cook and Mitchell, 1996; Franco and Swaminathan, 1997). As a consequence, the random clause length model is not useful for evaluating the performance of advanced SAT algorithms.

Up to date, instances most used for evaluating SAT solvers are based on the fixed clause length model (*uniform random k-SAT*) (Franco and Paull, 1983; Mitchell et al., 1992). For a given number of variables n, a number of clauses m and a clause length k, uniform random k-SAT instances are built as follows:

1 k literals are chosen independently and uniformly from the set of 2^n possible literals to generate a clause

2 clauses are not included into the problem instance if they contain multiple copies of the same literal, or if they are tautologies

3 clauses are generated and added to the formula until it contains m clauses overall.

The 3-SAT problem instances used in this work were built by the G3 algorithm (Motoki and Uehara, 1999). The relation between n and m to generate hard-to-solve 3-SAT instances is defined by $m \geq 4,\ 24 \times n + 6,\ 21$, based on theoretical and experimental analysis (Crawford and Auton, 1993; Williams and Hogg, 1994).

5.4 Execution time analysis

The main goal of the analysis is studying the efficiency of the proposed MapReduce approaches for 3-SAT, taking into account that Hadoop was not developed for solving this kind of optimisation problems, but for data intensive problems instead. More specifically, we focus on the combined divide and conquer and cumulative learning approaches and the randomised MapReduce 3-SAT algorithm.

5.4.1 Comparison of deterministic algorithms

First, we compare the non-cooperative MapReduce algorithm (NDD) with the cooperative strategies sharing invalid literal configurations (CID) and only saving solutions found in the search (CnoID). Table 1 reports the execution time (in seconds) demanded by NDD, CID, and CnoID when solving 3-SAT instances of increasing dimension, up to problem instances with 50 literals and 280 clauses. Results correspond to the average execution times, computed in ten executions performed for each algorithm, in order to avoid biased results due to non-determinism in the parallel execution.

Table 1 Execution times comparison of NDD, CID, and CnoID

Literals	*Clauses*	*Execution time (seconds)*		
		NDD	*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	1,023	365
50	280	1,996	7,260	1506

Results in Table 1 indicate that small problem instances (less than 40 literals) are solved in reduced execution times for all algorithms. CnoID is the fastest method, but the differences are small. Execution times increase significantly for problem instances with more than 34 literals, because all algorithms splits the problem fixing 6 literals, according to $\frac{\log(2 \times mex\_mappers)}{\log(2)}$. In each iteration, 15 literals are fixed. Thus, at least two MapReduce iterations are needed to find a solution. Iterative algorithms do not run efficiently in Hadoop, because the job initialisation process is expensive: new tasks for job clients are created and new Java virtual machines may also be created for new executions. The information needed to execute a new MapReduce job is not kept in the main system memory; it is usually retrieved from the hard drive, thus slowing the execution times.

The proposed algorithms differ in two aspects: the stop condition and how they evaluate if a literal assignment makes one clause false. The key feature that speeds up CnoID is stopping the search as soon as a solution is found. NDD finishes only when all mappers are processed, slowing down the whole algorithm. Execution time differences between CID and CnoID grow with the size of the problem, as more invalid literal assignments are found and stored/retrieved from HBase. This interaction increases the time needed to retrieve information from the centralised database, which is higher than evaluating each candidate solution.

Empirical analysis demonstrates that storing a set of invalid literals in a centralised 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 significantly smaller than the network cost of sending the literals to HBase. Nevertheless, the proposed algorithm use the invalid literal set just to speed up the evaluation. A smarter use of the invalid literal set could be based on exploiting the 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 (executed only once)

2 using the invalid literal assignments found in the map phase to prune the search space in the reduce phase

3 using the invalid literal configurations to fix literals and prune branches of the search space.

All these experimental results give important hints about the main performance issues of developing the deterministic MapReduce algorithm to solve the 3-SAT problem.

5.4.2 Evaluation of the randomised algorithm

The efficiency of the randomised algorithm (Rand-MR) was compared with the best deterministic algorithm (CnoID) over instances with up to 100 literals and 200 clauses.

Table 2 reports the execution time of Rand-MR and CnoID. The exact execution time for CnoID for the problem instance with 100 literals and 200 clauses is not reported, because the algorithm was not able to complete the

execution in two days. Considering the time needed for the 3-SAT instance with 50 literals and 280 clauses, we conjecture that CnoID would require a significantly larger execution time to solve 3-SAT instances with larger dimension.

Table 2 Execution time comparison of CnoID and Rand-MR

Literals	*Clauses*	*Execution time (seconds)*	
		CnoID	*Rand-MR*
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	1,506	152.048 ± 7.540
100	200	> 200,000	8,921.049 ± 353.820

Efficiency results show that Rand-MR uses the available processing power to perform an efficient search, using more mappers and reducers. The search is accelerated up to 10 × when comparing with the best deterministic MapReduce algorithm. Furthermore, Rand-MR is fully scalable and takes advantage of the Hadoop system, being able to use any number of computing resources (for example, available in cluster, grid, or cloud) to solve large scale problems.

The main difference between CnoID and Rand-MR is that the randomised method exploits both map and reduce phases, improving the search. CnoID, does not exploit the reduce phase, wasting execution time just preparing the information for the next MapReduce iteration. Hadoop reinitialisation is expensive because it involves creating and launching new Java virtual machines and a new job that will check for TaskTrackers available and data in HDFS. CnoID always demands a given number of iterations for a specific problem instance, while Rand-MR can succeed to find a solution in less iterations, but never using more iterations than CnoID.

Let us consider as an example a 3-SAT instance with 40 literals to analyse the behaviour of the algorithms. Suppose that initially the problem is split into 16 subproblems and $d = 15$ literals are fixed in each iteration. CnoID needs at least three iterations to find a solution (if any). On the other hand, Rand-MR could succeed in finding a solution in any iteration (using the Schöning algorithm). Moreover, a solution could also be computed deterministically if the formula is not hard to solve, for example when the fixed literals causes that UPPLE significantly simplifies the problem. Rand-MR fixes at least $d = 15$ literals in each iteration in a deterministic manner, as CnoID does, so for this toy example, at most three iterations are needed to find a solution to the problem (if any).

From the analysis, we conclude that the execution time can be improved in at least one order of magnitude applying a randomised MapReduce approach. The acceleration is more than 10× over the fastest deterministic MapReduce algorithm for problem instances up to 50 literals, and better for larger instances. For small problem instances (e.g., less than 34 literals), Rand-MR succeeds to find a solution in less than one second. When facing larger problem instances, the randomised approach significantly outperform CnoID in terms of computational efficiency, being able to solve the problem having 100 literals and 200 clauses in about two and a half hours, while CnoID cannot find a solution in two days.

There is only one issue regarding Rand-MR: the UPPLE algorithm used in the reducers modifies the original formula, reducing it by fixing literals that appear only once in a clause or removing from clauses those literals that are already fixed, and propagating its value to other clauses. UPPLE produces a new set of fixed literals and a new formula, which is saved to HBase using the MD5 of the candidate solution as the key. The MD5 for different candidate solutions found in different mappers 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 these collisions do not happen. Nevertheless, to deal with this issue we included in Rand-MR a final check with the original formula (the one received as input parameter) when a solution is found, just to be sure that the result is correct. Anyway, the collision situation only happens with a very low probability. Rand-MR found the correct solution in every execution performed in the experimental analysis reported in this section.

Rand-MR can be further improved by including UPPLE and/or the Schöning search in the mappers. Rand-MR 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 values for mappers and reducers. A small value of d could lead to waste time in many iterations of the MapReduce algorithm. On the other hand, large values of d could imply that mappers and reducers end up demanding too much time to apply the DFS, but fewer iterations might be required to find a solution and UPPLE would have more information to further simplify the formula. All these configurations should be deeply studied to analyse the trade-off between the search capabilities and the execution time of the proposed approach using MapReduce to solve the 3-SAT problem.

5.5 Complexity analysis

This subsection presents the time and space complexity analysis of the proposed algorithms.

The proposed MapReduce algorithms have three components: the main MapReduce job that configures the algorithm and then iterates until a solution is found, the mapper algorithm, and the reducer algorithm. The time

complexity of a MapReduce algorithm is $O(MapReduce) = \#iterations \times (O(mapper) + O(reducer))$.

- *NDD*. The main MapReduce job in NDD splits the problem in 2^r subproblems and performs $O(2^{n-d-r})$ iterations until all literals are set. The NDD mapper sets d literals, creates 2^d elements, and iterates over them. In each iteration, the algorithm checks if a candidate solution is valid with a time complexity of $O(m)$. Thus, the time complexity of the NDD mapper is $O(m.2^d)$ and its space complexity is $O(2^d)$, needed to store all candidate solutions generated by fixing d literals. Overall, the time complexity of NDD is $O(2^r + 2^{n-d-r} \times (m.2^d)) = O(2^r + m.2^{n-d-r+d}) = O(2r + m.2^{n-r})$ and its space complexity is $O(2^d)$.
- *NDFS-PA*. The NDFS-PA mapper improves over NDD, saving to HBase invalid literals found in previous iterations. In the worst case, the time complexity of the NDFS-PA mapper is $O(m2^d)$, the same as the NDD mapper, and its space complexity is also $O(2^d)$. Thus, the time complexity of NDD is $O(2^r + m.2^{n-r})$ and its space complexity is $O(2^d)$.
- *NDFS-RM*. NDFS-RM addresses memory errors, by recomputing the number of mappers to create in each iteration. The time complexity of NDFS-RM is the same as NDD, $O(2^r + m.2^{n-r})$ and its space complexity is $O(2^d)$.
- *CID/CnoID*. The cumulative learning technique allows sharing information among mappers to end the search as soon as a solution is found, but it does not reduce the time complexity of the mapper algorithm. The time complexity of CID/CnoID mapper is $O(m.2^d)$ and its space complexity is $O(2^d)$. The time complexity of both CID and CnoID is $O(2^r + m.2^{n-r})$ and its space complexity is $O(2^d)$.
- *Rand-MR*. The Rand-MR mapper introduces the Schöning algorithm which executes in $O(1.344^p)$ considering p literals. In Rand-MR, Schöning is executed on a subspace of the search space of $n–d–r$ literals. Overall, the time complexity of the Rand-MR mapper is $O(1.334^{n-d-r}.2^d)$ and its space complexity is $O(2^d)$. In the worst case, the Rand-MR reducer receives $O(2^d)$ candidate solutions. The Rand-MR reducer algorithm iterates over all $2d$ candidate solutions, first applying DFS, then UPPLE, and finally Schöning. DFS executes in $O(m + n)$. UPPLE iterates over all clauses, having a time complexity of $O(m)$. Schöning is executed on a search space of $n–d–d–r$ literals, so its time complexity is $O(1.344^{n-2d-r})$. Overall, the time complexity of the Rand-MR reducer is $O(2d \times (2^d \times (1.344^{n-2d-r})) = O(2d + 11.344^{n-2d-r})$. The time complexity of the main MapReduce job in Rand-MR is $O(2r + 2^{n-r-d})$. Overall, the time complexity of Rand-MR is $O(2^r + 2^{n-r-d} \times ((1.344^{n-d-r}2^d) + (2^{d+1} 1.344^{n-2d-r}))) = O(2^r + 2^{n-r-d} 1.344^{n-d-r} 2^d + 2^{n-r-d}2^{d+1} 1.344^{n-2d-r}) = O(2^r + 2^{n-r} 1.344^{n-d-r} + 2^{n-r} + 11.344^{n-2d-r})$.

6 Conclusions and future work

This work presents different variants of parallel algorithms to solve a classical constraint satisfaction problem (3-SAT) using the MapReduce model and the Hadoop framework. Up to our knowledge, this is an original contribution as no proposals applying MapReduce/Hadoop for optimisation have been published until now.

We review the state-of-the-art about problem solving using distributed approaches and comment on the main mechanisms for information sharing to speed up the resolution of combinatorial optimisation problems having large search spaces. The main features that can be applied on the MapReduce model and the implementation provided in the Hadoop framework are highlighted and discussed.

We follow an incremental approach to design and evaluate MapReduce algorithms for 3-SAT. The proposed algorithms combine several strategies for parallel/distributed problem solving, including divide and conquer, cumulative learning, and stochastic local searches. All algorithms are specifically designed for Hadoop, taking into account particular fetures of the MapReduce paradigm and the underlying software infrastructure (i.e., the way Hadoop creates the mapper and reducer processes, how the map phase is executed, etc.).

Mapreduce and Hadoop abstract the developer from common distributed computing issues, such as communications among processes, fault tolerance, etc. However, the computing paradigm and the framework add several other issues related to the design of the algorithm and limitations for a MapReduce job, including the model used to split the problem, how to implement the stop condition or the search used to explore solutions, etc. All these algorithmic features and their implementations can have a significant impact on performance. Furthermore, some of them can lead to important execution time improvements, which are useful to overcome situations of poor algorithm performance.

We implement four variants of the MapReduce 3-SAT solver: three deterministic approaches and a randomised algorithm. The deterministic methods include a standard MapReduce search applying domain decomposition without cooperation (NDD), and two collaborative approaches applying the cumulative learning technique to exchange useful information found in the search: using invalid literals (CID) and without using invalid literals (CnoID). The randomised algorithm (Rand-MR) combines several heuristics and a stochastic local search such as the one used in the Schöning algorithm.

The main results of the experimental evaluation indicate that the cumulative learning approach without using invalid literals improves over the performance of a simple divide and conquer MapReduce algorithm. The strategy using HBase as a collaborative database of solutions allows improving the execution times, but only when sharing small pieces of information. Using a central database to share solutions and all the information about invalid literals is ineffective to speed up the search, as it becomes more time

consuming when the dimension of the problem instances grows (and it has more information to share).

The randomised algorithm significantly improves the performance of the fastest deterministic approach (CnoID).

The execution time improvements of Rand-MR over CnoID is at least of one order of magnitude: the acceleration is 10× for problems with up to 50 literals and better for larger instances. Rand-MR succeeds to find a solution in less than one second for small problem instances and it also solves instances with 100 literals and 200 clauses, for which CnoID cannot find the solution(s) in more than two days.

Using HBase as a centralised repository helps the synchronisation and cooperation among mappers and reducers, providing a fully distributed method for sharing information. In Rand-MR solutions and formulas are saved to HBase, leading to significant performance improvements when using many small saves that are read many times.

We conclude that the proposed approach is a promising strategy for solving 3-SAT and other combinatorial optimisation problems. Although MapReduce and Hadoop were not conceived for optimisation, by including few modifications and tweaks researchers have a powerful platform to deal with complex problems.

The main lines for future work are related to improve the performance of the randomised algorithm and to extend the evaluation of the proposed approach. We plan to implement Rand-MR over Spark, a fast engine for big data processing that is more efficient for iterative, in-memory computing. In addition, we suggest several improvements to the mapper and reducer processes, including fine tunning and incorporating new search strategies in order to better exploit the MapReduce paradigm. Regarding the evaluation of Rand-MR, we propose using a larger distributed system (e.g., hundreds of processing elements) to solve larger problem instances with up to 600 literals. The proposed approach should also be tested to solve other combinatorial optimisation problems too.

References

Aspvall, B., Plass, M. and Tarjan, R. (1979) 'A linear-time algorithm for testing the truth of certain quantified Boolean formulas', *Information Processing Letters*, Vol. 8, No. 3, pp.121–123.

Attiya, H. and Welch, J. (2004) *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, John Wiley & Sons, Hightstown, NJ, USA.

Balyo, T., Sanders, P. and Sinz, C. (2015) 'HordeSat: a massively parallel portfolio SAT solver', in *Theory and Applications of Satisfiability Testing*, *Lecture Notes in Computer Science*, Vol. 9340, pp.56–172.

Barreto, M., Abal, G. and Nesmachnow, S, (2011) 'A parallel spatial quantum search algorithm applied to the 3-SAT problem', *Proc. of XII Argentine Symposium on Artificial Intelligence*, pp.1–12.

Biere, A. (2010) *Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010*, Technical Report 10(1), Institute for Formal Models and Verification, Johannes Kepler Universitat, Austria.

Chong, E. and Zak, S. (2013) *An Introduction to Optimization*, John Wiley & Sons, Hoboken, NJ, USA.

Chrabakh, W. and Wolski, R. (2003) 'Gridsat: a chaff-based distributed sat solver for the grid', *Proc. of the ACM/IEEE Supercomputing Conference*, pp.1–13.

Cohen, J. (2009) 'Graph twiddling in a MapReduce world', *Computing in Science and Engineering*, Vol. 11, No. 4, pp.29–41.

Cook, S. (1971) 'The complexity of theorem proving procedures', *Proc. of the 3rd Annual ACM Symposium on Theory of Computing*, pp.151–158.

Cook, S. and Mitchell, D. (1996) 'Finding hard instances of the satisfiability problem: a survey', *Satisfiability Problem: Theory and Applications*, *DIMACS Series in Discrete Mathematics*, Vol. 35, pp.1–17.

Crawford, J. and Auton, L. (1993) 'Experimental results on the crossover point in satisfiability problems', *Proc. of the 11th National Conf. on Artificial Intelligence*, pp.22–28.

Davis, M., Logemann, G. and Loveland, D. (1962) 'A machine program for theorem-proving', *Comm. of the ACM*, Vol. 5, No. 7, pp.394–397.

Dean, J. and Ghemawat, S. (2008) 'MapReduce: simplified data processing on large clusters', *Comm. of the ACM*, Vol. 51, No. 1, pp.107–113.

Deleau, H., Christophe, J. and Krajecki, M. (2008) 'GPU4SAT: solving the SAT problem on GPU', *9th Int. Workshop on State-of-the-Art in Scientific and Parallel Computing*, pp.1–4.

Foster, I. (1995) *Designing and Building Parallel Programs*. Addison-Wesley, Boston, MA, USA.

Franco, J. and Paull, M. (1983) 'Probabilistic analysis of the Davis Putnam procedure for solving the satisfiability problem', *Discrete Applied Mathematics*, Vol. 5, No. 1, pp.77–87.

Franco, J. and Swaminathan, R. (1997) 'Average case results for satisfiability algorithms under the random-clause-width model', *Annals of Mathematics and Artificial Intelligence*, Vol. 20, Nos. 1–4, pp.357–391.

Hamadi, Y. and Sais, L. (2009) 'ManySAT: a parallel SAT solver', *Journal on Satisfiability, Boolean Modeling and Computation*, Vol. 6, pp.245–262.

Hertli, T. (2011) '3-SAT faster and simpler: unique-SAT bounds for PPSZ hold in general', *Proc. of the IEEE 52nd Annual Symposium on Foundations of Computer Science*, pp.277–284.

Hoos, H. and Stutzle, T. (2004) *Stochastic Local Search: Foundations & Applications*, Morgan Kaufmann, San Francisco, USA.

Huang, J. and Darwiche, A. (2003) 'A structure-based variable ordering heuristic for SAT', *Proc. of the 18th Int. Joint Conf. on Artificial Intelligence*, pp.1–6.

Hyvärinen, A., Junttila, T. and Niemela, I. (2009) 'Incorporating clause learning in grid-based randomized SAT solving', *Journal on Satisfiability, Boolean Modeling and Computation*, Vol. 6, No. 4, pp.223–244.

Hyvärinen, A., Junttila, T. and Niemela, I. (2010) 'Partitioning SAT instances for distributed solving', *Proc. of 17th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, *Lecture Notes in Computer Science*, Vol. 6397, pp.372–386, Springer.

Korte, B. and Vygen, J. (2007) *Combinatorial Optimization: Theory and Algorithms*, Springer Publishing Company, Incorporated, Heidelberg, Germany.

Kottler, S. and Kaufmann, M. (2011) 'SArTagnan – a parallel portfolio SAT solver with lockless physical clause sharing', *Pragmatics of SAT*.

Mitchell, D., Selman, B. andLevesque, H. (1992) 'Hard and easy distributions of SAT problems', *Proc. of the 10th National Conf. on Artificial Intelligence*, pp.459–465.

Motoki, M. and Uehara, R. (1999) *Unique Solution Instance Generation for the 3-Satisfiability (3SAT) Problem*, Technical Report COMP98-54, Institute of Electronics, Information and Communication Engineers.

Motwani, R. and Raghavan, P. (1995) *Randomized Algorithms*, Cambridge University Press, New York, NY, USA.

Nesmachnow, S. (2010) 'Computacion cientıfica de alto desempeno en la Facultad de Ingenier´ıa, Universidad de la Rep´ublica', *Revista de la Asociaci´on de Ingenieros del Uruguay*, Vol. 61, No. 1, pp.12–15.

Nesmachnow, S. (2014) 'An overview of metaheuristics: accurate and efficient methods for optimisation', *Int. Journal of Metaheuristics*, Vol. 3, No. 4, pp.320–347.

Odersky, M., Spoon, L. and Venners, B. (2008) *Programming in Scala: A Comprehensive Step-by-step Guide*, Artima Incorporation, USA.

Paturi, R., Pudlak, P., Saks, M.E. and Zane, F. (1998) 'An improved exponential-time algorithm for *k*-SAT', *Proc. of the 39th Annual Symposium on Foundations of Computer Science*.

Paturi, R., Pudlak, P., Saks, M.E. and Zane, F. (2005) 'An improved exponential-time algorithm for *k*-SAT', *Journal of the ACM*, Vol. 52, No. 3, pp.337–364.

Posypkin, M., Semenov, A. and Zaikin, O. (2012) 'Using BOINC desktop grid to solve large scale SAT problems', *Computer Science Journal*, Vol. 13, No. 1, pp.25–34.

Satchell, S. (1999) *Optimizing Optimization: The Next Generation of Optimization Applications and Theory*, Academic Press.

Schöning, T. (1999) 'A probabilistic algorithm for *k*-SAT and constraint satisfaction problems', *40th Annual Symposium on Foundations of Computer Science*, pp.410–414.

Schubert, T., Lewis, M. and Becker, B. (2009) 'PaMiraXT: parallel SAT solving with threads and message passing', *Journal on Satisfiability, Boolean Modeling and Computation*, Vol. 6, No. 4, pp.203–222.

Schulz, S. and Blochinger, W. (2010)' Parallel SAT solving on peer-to-peer desktop grids', *Journal of Grid Computing*, Vol. 8, No. 3, pp.443–471.

Selman, B., Levesque, H. and Mitchell, D. (1992) 'A new method for solving hard satisfiability problems', *Proc. of the 10th National Conf. on Artificial Intelligence*, pp.440–446.

Shafer, J., Rixner, S. and Cox, A. (2010) 'The Hadoop distributed filesystem: balancing portability and performance', *IEEE Int. Symposium on Performance Analysis of Systems & Software*, pp.1–12.

Silva, J.M. and Sakallah, K. (1996) 'GRASP – a new search algorithm for satisfiability', *Proc. of the 1996 IEEE/ACM Int. Conf. on Computer-Aided Design*, pp.220–227.

Vander-Swalmen, P., Dequen, G. and Krajecki, M. (2009) 'A collaborative approach for multi-threaded SAT solving', *Int. Journal of Parallel Programming*, Vol. 37, No. 3, pp.324–342.

White, T. (2009) *Hadoop: The Definitive Guide*, O'Reilly Media, Inc.

Williams, C. and Hogg, T. (1994) 'Exploiting the deep structure of constraint problems', *Artificial Intelligence*, Vol. 70, Nos. 1–2, pp.73–117.

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