A Distributed Pseudo-Boolean Optimization Solver

Nuno Santos
nunocsantos@ist.utl.pt

Instituto Superior Técnico

Abstract. Parallel computing was the subject of great research in the last decades and is increasingly being chosen as a solution for developing applications that require high computational power, such as *Boolean Satisfiability* and *Pseudo-Boolean Optimization* problems. The research in SAT solvers obtained significant results in the last years, achieving significant reductions in execution times when solving the problem. Unfortunately, hard instancies of boolean satisfiability require large computational power and even efficient SAT solvers take huge execution times to obtain their solution. Therefore, SAT solvers adaptation to parallel computing systems began to be the subject of considerable research and there already exist several parallel versions of popular SAT solvers. Yet, this same field has not been well explored for the Pseudo-Boolean Optimization problem and therefore this project intends to contribute and encourage the research into parallel solutions to this problem. The goal of this project is to propose and implement a distributed Pseudo-Boolean Optimization Solver using MPI (*Message Passing Interface*), focused on an efficient search space partition, more specifically the partition of the optimization search space.

Contents

1 Introduction

The well known *Boolean Satisfiability* (SAT) and *Pseudo-Boolean Optimization* (PBO) problems gained a lot of attention in the last years due to their possible application in many domains, such as software and hardware verification. Since then, several algorithmic solutions have been proposed to solve both problems and many of them proved to be very efficient in solving several instances of the problems. Many SAT solutions (known as SAT solvers), for instance GRASP [24] and Chaff [22], contributed with several techniques to improve the SAT resolution. Due to the relevant results on SAT research, the PBO researchers embraced the SAT solvers efficiency and proposed efficient ways to solve PBO instances by extending SAT solvers to handle them.

Although SAT solvers are becoming more sophisticated to reduce execution time, using improvement techniques such as *clause learning*, *adaptive branching*, *non-chronological backtracks* and many more, the demand for more computation power led SAT researchers to explore solutions taking advantage of parallel computing systems. Parallel computing systems, like *clusters* and *grids*, allow the use of several resources in parallel and enable the partition of a problem for a concurrent resolution in such resources. Popular SAT solvers have been migrated to the parallel computing world using the well known *Task Farm* approach, which proved to be a successful solution to partition SAT solver instances in such environments. Unlike SAT solvers, the migration of PBO solvers to parallel computing systems has not been well explored. So the purpose of this project is to develop a distributed implementation of Minisat+ [13] with MPI (*Message Passing Interface*) technology, using the Task Farm approach adopted by several parallel SAT solver implementations.

The document is organized as follows: definitions of the two problems addressed in the project are presented in Section 2 and Section 3 describes important techniques responsible for the improvements in SAT solvers. Some alternatives to solve PBO problems and approaches to extend SAT solvers to handle PBO instances are presented in Section 4. In Section 5 are described parallel SAT solver strategies and techniques to partition the search space of a SAT instance. The approaches for the implementation proposed for the project are described in Section 6.

2 Satisfiability Problem

In this section are presented the definitions of two well-known problems that are the core of the context of this project, the *Boolean Satisfiability* and *Pseudo-Boolean Optimization* problems.

2.1 Boolean Satisfiability Problem

The *Boolean satisfiability problem* (also known as SAT) consists of determining if a *Boolean formula* is satisfiable or not. If exists at least one variable assignment that evaluates the formula to true, the formula is considered satisfiable, otherwise it is unsatisfiable. A variable assignment that makes a formula satisfiable is known as *model*.

Definition 1. *A Boolean variable is a symbol that might assume one of two values: true or false.*

Definition 2. *A Boolean formula represents a set of Boolean variables related by Boolean operators (and, or, not) and can be either true or false depending on the variable values.*

SAT was the first known example of a *NP-Complete* problem and by definition there is no known algorithm that solves all SAT instances efficiently, due its exponential worst case complexity [14]. SAT problem is particularly popular because of its easy application in plenty of other domains, such as electronic design automation (EDA) [20, 16] and Artificial Intelligence (AI) [7]. So there is a high demand and research with the aim of finding fast and efficient solutions for SAT.

2.2 Pseudo-Boolean Optimization Problem

Due to the complexity of some application domains, such as Digital Filters design [3, 17], there was the need to go beyond purely Boolean representation constraints and use a more complex representation form known as *Pseudo-Boolean constraints* (PB-constraints), a generalization of clauses where each variable can contain a weight associated.

Definition 3. *A literal represents a Boolean variable or its complement.*

Definition 4. *A clause is a disjunction of literals.*

A PB-constraint is an inequality $C_0p_0 + C_1p_1 + ... + C_{n-1}p_{n-1} \geq C_n$, where, for all i, p_i is a literal and C_i an integer coefficient. The inequality left-hand side will be abbreviated by LHS and the right-hand constant C_n refereed as RHS. A coefficient C_i is *activated* under a partial assignment if its corresponding literal p_i is assigned to true. If all constants C_i are 1, then the PB-constraint is equivalent to a standard SAT clause.

The *Pseudo-Boolean* (PB) problem, also known as *0-1 integer linear programming* (0-1 ILP), consists of determining if a PB-constraint is satisfiable or not. A PB-constraint is said to be *satisfied* under an assignment if the sum of its activated coefficients on the LHS exceeds or is equal to the RHS constant, otherwise it is *unsatisfied.*

PB problems often contain an *objective function*, a linear term that should be minimized or maximized under the given constraints. An *objective function* is a sum of weighted literals on the same form as a LHS. The *Pseudo-Boolean optimization* problem (PBO) is the task of finding a satisfying assignment to a set of PB-constraints that minimizes or maximizes a given objective function.

3 SAT Solvers

In yearly years, many effective algorithmic solution as been proposed for solving SAT instances. The most popular type of complete and efficient solvers of today are known as *conflict-driven* solvers.

Conflict-driven solvers are variations of the well known *Davis-Putman (DP) backtrack search* algorithm [10] combined with efficient improvement techniques, proposed in GRASP [24] and Chaff [22] algorithms. Techniques such as *clause learning*, *non-chronological backtracking*, *"two-watched-literals" unit propagation*, *adaptive branching* and *random restarts* are examples of the success in the research on SAT and became a basis on SAT solvers since then.

Popular conflict-driven solvers are Minisat [12], Sato [25], PicoSAT [8], Berkmin [15], Glucose [6] and zChaff (implementation of Chaff algorithm).

3.1 Davis-Putman backtrack search

Davis-Putman (DP) backtrack search algorithm [10](commonly named DPLL), a refinement of the earlier *Davis-Putman algorithm* [11], is a complete algorithm for determining the satisfiability of Boolean formulas and is the core of most actual SAT solvers.

The DPLL algorithm operates in problems where formulas are specified in CNF form, where a formula consists on a clause database.

Definition 5. *A formula is in the conjunctive normal form (CNF) if it is a conjunction of clauses.*

The advantage of CNF is that, for a formula be satisfied, each clause must be satisfied. A clause is satisfied if at least one of its literals is *true*. The DPLL performs a depth-first search on a binary decision tree where each node consists of a *decision*, i.e, an election of a variable from the clause database to assign a value. The algorithm has three main processes and its pseudo code is presented in Figure 1.

```
while(true){
    if(!decide())
        return satisfiable;
    while(!bcp())
        if(!resolveConflict())
            return not satisfiable;
}
```

Fig. 1. Pseudo code of DPLL.

The `decide()` process decides a variable in the clause database, among all unassigned ones, and assigns a value to it (*true* or $false$). Each *decision* represents a node on the decision tree and has a *decision level* associated. The *decision level* represents the depth of the node in the tree. The root (no assignments decided) has decision level 0, the first *decision* has decision level 1 and so on.

The `bcp()` process, which carries out *Boolean Constraint Propagation* (BCP), identifies new *implications* or *conflicts* produced by the current variable state. Implications are produced by applying the *unit clause rule*. When a clause of N literals has N-1 literals with value $false$ and one unassigned literal, the unassigned literal has to be obligatorily *true* for the clause be satisfied. Such clauses are known as *unit clauses*. Implications are then propagated and the process is repeated. Each implication generated has the same decision level of the *decision* that triggered it. The `bcp()` process ends when there is no more implications or a conflict is identified (an assignment turns the problem unsatisfiable).

In case a conflict is detected during the BCP process, the process `resolveConflit()` backtracks the search to the most recent *decision* that was not tried both values (*true* and $false$) and undoes all assignment performed until then. The remaining value is decided to proceed the search. If all previous *decisions* were tried both values, the problem is considered unsatisfiable.

If during the search, there is no conflicts detected and no more variables unassigned to decide, the problem is satisfiable.

3.2 Conflict Analysis

In order to reduce significantly the search space of the DPLL algorithm, the authors of GRASP [24] proposed an extension of the `resolveConflit()` process to handle conflict analysis techniques, such as *clause learning* and *non-chronological backtracks*.

In the occurrence of a conflict during BCP process, the conflict passes through a diagnose process where are discovered the assignments responsible for the appearance of the conflict. Discovered such assignments, a new clause denying such assignments is created and added to the clause database of the problem. These clauses are known as *conflict-induced clauses*. The addition of such clauses in the problem restricts conflicting and unnecessary paths in the search space. Another advantage of the diagnose process is the possibility of *non-chronological backtracks*. By analysis of the conflict is possible to know when there is no way of finding satisfying assignments until the search backtracks to a certain decision level. This level corresponds to the last level where assignments that caused the conflict were made and knowing that level, the search can jump immediately to it to save time and effort.

To better explain the generation of *conflict-induced clauses*, the following definitions are presented: $R(x)$ correspond to the "reason" of the assignment x and contains a set of antecedent assignments to the unit clause ω that lead to the implication of x. For a clause $\omega = \{x + y + \neg z\}$ and an assignment $\{y = 0, z = 1\}$, the "reason" of $x = 1$ is $R(x) = \{y = 0, z = 1\}$. Decision assignments have $R(x) = \emptyset$. $\delta(x)$ corresponds to the decision level of the assignment x. A conflicting assignment associated with a conflict c is denoted as $R_C(c)$ and the respective conflict-induced clause as $\omega_C(c)$. To facilitate the computation of $R_C(c)$, the "reason" of an assignment x is partitioned by decision levels and is given by:

$$\begin{aligned} \Lambda(x) &= \{y \in R(x) | \delta(y) < \delta(x)\} \\ \Sigma(x) &= \{y \in R(x) | \delta(y) = \delta(x)\} \end{aligned} \tag{1}$$

If a conflict c is detected, $R_C(c)$ is computed by the recursive definition:

$$R_C(x) = \begin{cases} x & \text{if } R(x) = \emptyset \\ \Lambda(x) \cup \left[\bigcup_{y \in \Sigma(x)} R_C(y)\right] & \text{otherwise} \end{cases} \tag{2}$$

The computation of a conflict-induced clause is simply the negation of the conflicted assignment. Computed a conflicted assignment $R_C(k) = \{x = 1, y = 0, z = 0, w = 1\}$, the conflict-induced clause produced is $\omega_C(k) = \{\neg x + y + z + \neg w\}$.

The backtrack level of a conflict is computed according to:

$$B(c) = max\,\{\delta(x) | x \in R_C(c)\} \tag{3}$$

When $B(c) = d - 1$, where d is the current decision level, the search process backtracks chronologically. When $B(c) < d - 1$ the process backtracks non-chronologically.

Conflict analysis introduces significant overhead and for small instances of SAT can lead to larger run times, but for large instances may contribute to a significant reduction in run times. Another drawback is the growth of the clause database during an execution. Because of conflict-induced clauses addition, the size of the clause database grows with the number of conflicts and in the worst case such growth can be exponential in the number of variables.

3.3 Adaptive Branching

One key aspect of SAT solvers is how variables are selected at each *decision* in the algorithm. Decisions affect the performance of an algorithm and using a good heuristic is crucial to the efficiency of a solver. Unfortunately there is no perfect heuristic and each heuristic performs better in different ranges of SAT instances than others. Over the years, many effective *branching heuristics* were proposed and detailed descriptions are made in [21] by Silva.

The simplest heuristic, commonly known as RAND, is to simply select the next decision randomly from among the unassigned variables. At the other extreme are the BOHM and MOM heuristics that involve the maximization of some moderately complex functions of the current variable state and the clause database.

Popular heuristics, somewhere in the middle of the spectrum, are the DLIS and DLCS heuristics. This two heuristics count the number of unresolved clauses in which a given variable x appears as a positive literal, C_P, and as a negative literal, C_N. The DLCS (*Dynamic Largest Combined Sum*) heuristic selects the variable with the largest sum $C_P + C_N$ and assigns the value *true* if $C_P \geq C_N$ or $false$ if $C_P < C_N$. The DLIS (*Dynamic Largest Individual Sum*) heuristic selects the variable with the largest individual sum (C_P or C_N) and assign *true* if a C_P value is the largest, or $false$ if a C_N value is the largest. Variations of DLIS and DLCS, referred to as RDLIS and RDLCS, consist in randomly selecting the value to be assigned to a given selected variable, instead of comparing C_P and C_N values.

Another efficient heuristic, proposed by the authors of Chaff [22], is VSIDS (*Variable State Independent Decaying Sum*). This heuristic associates a counter to each literal and every time that a clause is added, the counters of its literals are increased. The heuristic selects the variable associated with the literal that has the highest counter. Periodically, all counters are divided by a constant.

3.4 Two Watched Literals Unit Propagation

In practice, for most SAT problems, a major portion of the solvers is spent in the BCP process and so an efficient BCP engine is one of the keys to any SAT solver. Each time an assignment is deduced during a search, all *newly implied* clauses must be visited to generate possible new implications. A clause is *implied* if and only if all but one of its literals is assigned to $false$.

Chaff [22] authors proposed an efficient implementation technique to find newly implied clauses, avoiding constant visits to every clause that contains a literal which the current assignment sets to $false$. Instead of visiting a clause each time it is assigned a value to its literals, to every clause are picked two any literals not assigned to $false$, which can be watched at any given time, and a clause is visited only if one of those watched literals is assigned to $false$. It is guaranteed that until one of the two literals is assigned to $false$, there cannot be more than N-2 literals, in the clause, assigned to $false$ and so the clause cannot be implied. When a clause is visited but is not implied, means that at least one non-watched literal is not assigned to $false$. That literal is chosen to replace the watched literal assigned to $false$ and the property is maintained.

Two watched literals technique reduces significantly the time spent in the BCP process and at the time of backtracking, there is no need to modify watched literals in the clause database. Therefore, unassigning a variable can be done in constant time.

3.5 Restarts

A restart consists of clearing the state of all variables (including all decisions) during an execution of a solver and starting the search from the beginning. Any still-relevant conflict-induced clauses added to the clause database are still preserved after the restart, so that the solver does not repeat previous searches. The intention of restarts is to provide a chance to change early decisions in view of the previous problem state. If a search tend to a path where the probability of finding a satisfiable assignment is reduced and a lot of effort is need to leave it, a restart gives the possibility to follow a different path that may have a highest probability of success. Restarts period is extended with each restart so that the completeness of the algorithm is maintained.

4 Pseudo-Boolean Optimization Solvers

The demand and research for efficient solvers for the PBO problem are recent and falls into two categories of algorithms: *Generic ILP* solvers and *specialized 0-1 ILP* solver [4].

4.1 Generic ILP solvers

Generic ILP solvers are used to solve linear programming problems, in general, and since PB-constraints are a variant of linear programming problems, they were traditionally handled by this kind of solvers. Although *generic ILP* solvers fit naturally on PB problems, they tend to ignore the Boolean nature of 0-1 variables. One popular type of *generic ILP* solvers are *branch-and-bound* algorithms [19] and they have proven to be very effective for PB instances where is not hard to find a variable assignment that satisfies all constraints. Some of popular ILP solvers are CPLEX and SCIP [2].

4.2 Specialized 0-1 ILP solvers

Specialized 0-1 ILP solvers consist on an adaptation of conflict-driven SAT solvers to handle PB-constraints. There are two strategies to support PB-constraints: or the SAT solver is modified to handle PB-constraints directly (known as *native* support solvers) or each PB-constraint is translated into a set of equivalent CNF clauses, in a pre-processing step, to be processed by an unmodified SAT solver. The translation strategy has the advantage of using efficient SAT solver procedures as a black box, but suffers the possible exponential growth of CNF clauses, due to the translation nature. Native support solvers are more complex and imply a modification of the BCP process to handle PB-constraints in the way that they handle CNF clauses.

Popular solvers with native support are PBS [5] and Pueblo [23] (an extension of Minisat to handle PB-constraints). A popular non-native support solver is Minisat+ [13], a simple and well documented solver which uses Minisat as black box to handle translated constraints.

4.2.1 Processing of PB-constraints

Before start processing PB-constraints, it is advantageous to define a *normal form* for all constraints in the constraint database. Normalization of PB-constraints simplifies the implementation of a solver by giving fewer cases to handle and may reduce the number of constraints in the problem. Several techniques are used in normalization of constraints and each solver differs in the number of techniques implemented. The basic normalization technique for a solver performs the following steps:

– All constraints are changed to a common type ($\leq$ or $\geq$) by negating all constants, if necessary.
– Negative coefficients are eliminated by changing the associated literal x into $\neg x$ and updating the RHS.
– Multiple occurrences of the same variable are merged into a single term $C_i x$ or $C_i \neg x$.
– The coefficients are sorted into the order of increasing C_i values.
– Coefficients greater than the RHS are replaced with the RHS.
– The coefficients of the LHS are divided by their greatest common divisor ("gcd"). The RHS is replaced by $\lceil RHS/gcd \rceil$.

Assign values to variables in a PB-constraint is a more complex process than in pure CNF clauses. Each PB-constraint must contain a field representing the current LHS value, the sum of all activated C_i, and another field representing the maximum value possible for the LHS, the sum of all activated C_i and all C_i associated with unassigned variables. For simplicity, the fields are designated by $currLHS$ and $maxLHS$, respectively.

Assigning *true* to a variable v implies traversing all PB-constraints containing a literal of v. The $currLHS$ of every PB-constraint containing the positive literal of v is incremented by the coefficient associated with v in that constraint. On the other hand, the $maxLHS$ of every PB-constraint containing the negative literal of v is decremented by the coefficient value associated with that literal. Undoing an assignment of *true* to a variable v, entails the inverse process, the

$currLHS$ and $maxLHS$ of every affected PB-constraint are decremented and incremented, respectively. When the assigned value of a variable v is $false$, the process is similar, only changes the $currLHS$ and $maxLHS$ fields in the operations.

During the process of updating PB-constraint fields, new implications and conflicts can be found. Each PB-constraint type has associated a set of rules for detecting implications and conflicts. Given a PB-constraint of type "$\leq$", any literal x_i, whose coefficient is $C_i > (RHS - currLHS)$, is implied to $false$. On the other hand, given a PB-constraint of type "$\geq$", a literal x_i is implied to $true$ if its coefficient is $C_i > (maxLHS - RHS)$. Conflicts are detected when the $currLHS > RHS$ (for a PB-constraint of type "$\leq$") or the $maxLHS < RHS$ (for a PB-constraint of type "$\geq$").

PB-constraints of type "=" are divided into two constraints of type "$\leq$" and "$\geq$" that have the same LHS and RHS.

4.2.2 Optimization Process

As described in Section 2.2, each PBO problem has an objective function associated. By calling the SAT procedure recursively, it is possible to find an assignment that minimizes or maximizes a given objective function. Since the objective function is a linear term, it can be added to the problem as a constraint that limits the sum of its literals.

Assuming a problem with objective function $f(x)$, the first search of the solver into the set of constraints (without considering the objective function) will obtain an initial solution $f(x_0) = k$. In the second search is added the constraint $f(x) < k$ (in the case of minimization). If the problem is unsatisfiable, k is the optimal solution , otherwise the process is repeated with the new solution found (adding $f(x) < k_{new}$ as a new constraint). The process is repeated until an optimal k is found. In maximization, the optimization process is the same but constraints are added to the problem as $f(x) > k$.

4.2.3 Translation of PB-constraints

Translation of PB-constraints into CNF clauses is a technique used in non-native support specialized 0-1 ILP solvers. Consist on a process between the normalization of PB-constraints and the calls to the SAT solver. The translation process has two main steps: each constraint is converted into a single-output circuit and then each circuit is converted into CNF clauses by *Tseitin transformations*. A more detailed description of this conversions is presented in [1, 13].

The goal of translating PB-constraints into CNF clauses is not only to get a compact representation, but also preserve as many implications between literals of the PB-constraints as possible. This concept is known as *arc-consistency*. A translation maintains arc-consistency when all assignments that could be propagated on the original constraint can also be found on the SAT solver's unit propagation, operating on a translation of that constraint. Although arc-consistency is always desirable, maintain such propriety without an exponential growth in the translation is a very difficult task. So there is a trade-off between arc-consistency and the translation size.

There are several approaches to convert a PB-constraint into a single-output circuit. Each approach defines distinct properties on the translation of constraints, having its advantages and drawbacks, and until now there is no perfect conversion. Popular approaches are conversions into BDDs, networks of adders and networks of sorters [13]. BDDs guarantee arc-consistency in the translation, but suffer from exponential size translation in the worst case. On the other hand, networks of adders do not guarantee arc-consistency but result in a linear size translation. In the middle of the scope are the networks of sorters, not arc-consistent but closer to the goal, with an almost linear size translation.

5 Parallel Solvers

Due to the evolution and popularity of *parallel computing systems*, it became evident that the future of research on SAT solvers would fall into the world of parallel computing, like many other domains did. Since hard instances result in huge search spaces, having several resources to share the effort of the search became a very viable strategy. The demand of algorithms for SAT, taking advantage of *cluster* and *grid* architectures, is growing and the first strategies appear as a stimulation to the research in this area.

In this section are presented some popular techniques used in parallel SAT solvers of today and a popular paradigm to develop concurrent applications to parallel computing systems, called MPI.

5.1 MPI

The demand for more computational power lead the programmers to adopt *parallel computing systems* to run their programs. Super computers, clusters, grids and even actual personal computers (multi-core computers) are considered parallel computing systems, although with different characteristics. Unfortunately, developing programs for parallel systems is more complex than programming in a sequential system, since parallel systems involve multiple processing elements (*processes*) during a program execution. *Parallel programming systems* are used to handle the communication and synchronization between those processes and the MPI is one example of such systems.

MPI (*Message Passing Interface*) is a popular and widely used API that provides essential virtual topology, synchronization and communication functionality between a set of processes in parallel computing systems. MPI consists on a specific set of routines that are directly callable from a variety of languages (for example, C and C++) to integrate its features into any program. MPI was defined with the contribution of many commercial and open-source vendors with the aim of creating a parallel programming standard with high performance, scalability and portability. Since then, it has become the industry's de-facto standard to implement portable parallel programs for a wide variety of parallel computing systems.

5.2 Parallel SAT Solvers

A popular and very used parallel SAT solvers approach is the *Task Farm*, that uses the master/slave topology presented in the Figure 2. The objective of this approach is to divide the search space among all the slaves of the system so that different partitions of the search space can be computed simultaneously.

A *slave* (also known as *worker*) consists on an independent SAT solver that contains the entire clause database of a problem. The

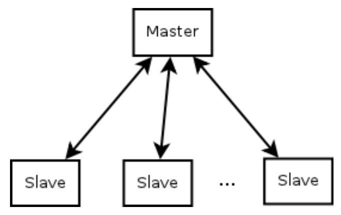

Fig. 2. Master/Slave topology

partition of the search space is managed by the *master* and the attribution of such partitions to the workers is performed by a process of task delivery. The master contains and manages a task repository which includes all tasks that must be sent to workers. A *task* consists on a set of initial assumptions (variable assignments) that correspond to a partition of the search space. Each task contains a different set of assumptions so that each worker computes distinct areas of the search space. Workers are responsible for computing the tasks received and responding to the master the obtained results (*sat* or *unsat*), according to the assumptions defined. A *sat* response may also include the model found, that satisfies the given problem, and an *unsat* response includes a set of conflicting assignments that originated the *unsat* result.

This approach is the core of several parallel SAT solvers such as PSato [26] (parallel version of Sato), GrADSAT [9] (parallel version of zChaff) and PMSat [18] (parallel version of Minisat). Although this solvers implement the same approach (differing in some upper level techniques), they differ in the parallel technologies used on their implementations. PMSat is highlighted since it was implemented using MPI, which will be used in this project.

5.2.1 Search Process

Using the variables of the problem, the master initially creates a task database by defining all necessary tasks (sets of assumptions) to compute the problem, this technique is descibed in the Section 5.2.2. After all tasks have been defined, the master initiates the search by sending one task, pulled from the task database, to each worker and waits for their responses.

Receiving an *unsat* response from a worker, the master uses the set of conflicting assignments received to remove unnecessary tasks from the task database (tasks proven to be useless due to the conflicting assignments) and to stop possible workers that are computing tasks proven to be *unsat*. When workers finish their tasks (or are ordered to stop one), a new task is pulled from the task database and assigned to them. The search process ends when the master receives a *sat* response from a worker or when the task database is empty and all workers computed their tasks resulting in *unsat* responses (the problem is *unsatisfiable*).

5.2.2 Assumptions Generation

Assigning values to a set of variables in a problem, redirects the search into a specific subspace of the search space. Therefore, by defining a specific group of assumptions, it is possible to partition the search space in such a way that all workers could compute cooperatively the entire search space. Defining such group of assumptions must guarantee two properties: the group must cover the entire search space and each set of assumptions in the group must represent a different subspace of the search space.

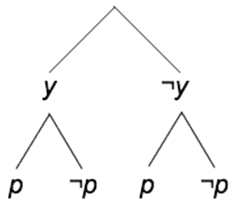

Fig. 3. Decision Tree of a problem with variables $\{y, p\}$

As an example, given a problem with two variables $\{y, p\}$ and the decision tree presented in the Figure 3, defining two assumptions with $y = true$ and $y = false$, respectively, will cover the entire search space and each workers could compute half the search space concurrently.

The example above is a very simple example, but generating assumptions is not a simple process. Assumptions must be carefully created and aiming the “right” variables to assume may improve the performance of a solver. The authors of PMSat [18] proposed some generation methods to improve the generation of assumptions. The methods proposed take into account the popularity of variables and literals (the number of occurrences on clauses) and use the most popular variables to start partitioning the search space.

5.2.3 Sharing Conflict-induced Clauses

As described in Section 3.2, conflict analysis techniques have several advantages on SAT solvers. Since a worker consists on an independent SAT solver, it can take advantage of conflict analysis techniques and generate conflict-induced clauses during their searches. In a parallel SAT solver, each worker can compute several tasks during an execution and each task may result in an addition of new clauses to the clause database of a worker. This gathering of conflict-induced clauses can be very advantageous since each task will have access to the “knowledge” of the previous ones.

One technique adopted by several parallel SAT solvers is the ability of sharing conflict-induced clauses among workers. Instead of a task has access only to the “knowledge” of their correspondent worker, it will have access to the global “knowledge” of the system. To archive such goal, each worker includes, in their responses to the master, a set of conflict-induced clauses generated during the task computation. The master spreads the clauses received by including them in the following attributions of tasks. The workers, after receiving a task, add the included conflict-induced clauses to their clause database.

The use of sharing conflict-induced clauses technique may help reducing the search space, avoiding unnecessary branches, but including such clauses in the communication flow implies an extra overhead in the execution. Some implementations restrict the size and the number of clauses, that are shared, as a solution to amortize the overhead imposed by this technique.

6 A Distributed Pseudo-Boolean Optimization Solver

In Section 5 were presented some results of the SAT research on parallel computing systems. The proposed *Task Farm* approach contributed for a successful migration of the SAT problem to the parallel computing world. Although great effort is spent developing improved parallel SAT solvers, there is no concrete results in the research on Pseudo-Boolean Optimization problem in the parallel computing field. So the aim of this project is to propose and implement a distributed PBO solver to contribute for PBO research.

6.1 Solver Approach

The good news for developing a parallel version of a PBO solver are that the characteristics of the *optimization process* in PBO favor its parallel implementation. Since the optimization process consists on iterative calls of the search procedure with different objective function restrictions, it is obvious that performing a concurrent search on the optimization search space will be a good strategy. Taking advantage of the well known *Task Farm* approach (described in Section 5.2), it is possible to propose a promising distributed implementation for PBO. The goal is to use independent Pseudo-Boolean solvers as workers and a master to manage and attribute tasks, where each task corresponds to a specific objective function restriction that must be explored. For simplicity, the solver will only handle the minimization process, since the maximization is the inverse process.

Given an objective function $f(x)$ with N variables, the size of the optimization search space to be explored (in the worst case) is 2^N. The optimization search space can be defined by an interval $\sigma = [k_{min}, k_0]$, where k_0 corresponds to the first solution found (when running the PB solver without considering the objective function) and k_{min} corresponds to the solution obtained when all positive C_i in the objective function are *inactivated* and all negative C_i are *activated*. Not all the values in σ are considered and values obtained only under a given assignment on the objective function are chosen to be explored.

Partitioning an interval σ in such a way that each worker computes significant subspaces, with different objective function restrictions, is not an easy task. The intuitive approach is to perform a binary search on the considered values of σ and restricting σ according to the results obtained from the necessary tasks. To each worker will be attributed a different value $k \in \sigma$ which will be used to produce the desired restriction contraint $f(x) < k$ for each task. The master manages and restricts the interval σ according to the responses received from workers. If a task, associated with a k value, results in an *unsat* response, means that there is no possible assignment which satisfies the given $f(x) < k$ restriction. By logic, if $f(x) < k$ is unsatisfiable, $\forall y \in \sigma, y < k \mid f(x) < y$ are also unsatisfiable. Using such property, the master substitutes the lower bound of σ with value k and aborts all workers computing restriction values lower then k. On the other hand, tasks resulting in *sat* responses lead the master to substitute the upper bound of σ with value k, aborting all workers computing restrictions higher then k. The process is repeated until only one value is covered by σ, which is considered the optimum solution.

During the optimization search, each worker will probably compute several values of σ and some changes to the clause database of a worker may be required between each task. The computation of following tasks with lower values does not imply any changes on the cause database, since the old objective function restriction does not affect the new one. On the other hand, in the computation of higher values, the old objective function restriction cancels the new search to be explored. To avoid such property, a removal of all contraints, created due the old objective function restriction, must be performed before adding and computing the new objective function restriction.

6.2 Solver Implementation

The proposed implementation will consists on a distributed version of Minisat+ [13] using the approach explained above. Minisat+ was chosen because of its clarity in the code and its well documentation, but we believe that this approach can be easily adapted in many other specialized 0-1 ILP solvers, since they usually use the same optimization process.

To handle the communication and synchronization between processes, it will be used the MPI because, as explained earlier, it is the industry's de-facto standard and has high performance and portability.

6.3 Challenges

After implementing the approach proposed in Section 6.1, some possible challenges may be taken into account to improve the implementation performance. Like in parallel SAT solvers, deciding the "right" tasks to be early computed can lead to a significant reduction in the execution time. So defining new heuristics for a better partition of the optimization search space (instead of the partition proposed) would be a good approach to improve the partition algorithm.

Learnt constraints sharing between workers is another challenge that can be explored. Sharing significant learnt contraints, which are not created due to the objective function, between all workers can result in a reduction on each task search space. However, this technique is very complex, since a selection for significant contraints must be performed at the end of each task, and imposes a significant overhead to the solver execution.

For a more fined grained partition of the problem, partitioning the search space in each task can be considered a good approach for future work. Attributing each task to be computed by a mesh of workers, which partition the search space among them, can lead to a better management of effort to the several processes of the system. Since Minisat+ can use any SAT solver as a worker, this challenge is facilitated because of its possibility of integrating an existing parallel SAT solver (such as PMSat) to operate as a worker.

7 Conclusion

The proposed PBO solver for this project aims to transport to the PBO world the efficient and well explored results in parallel SAT solvers research. Techniques such as Task Farm approach with master/slave topology, efficient heuristics for partition the search space and learnt clause sharing are examples that can be easily adopted to implement an efficient distributed PBO solver. Besides an efficient approach to maximize the performance of a search, the use of the MPI in the implementation, for communication and synchronization of processes, guarantees a high level of portability and performance on parallel computing systems, such as clusters or grids of computers.

Choosing Minisat+ [13] as the core for the solver proposed gives the possibility to integrate the most techniques described in this document. Since Minisat+ consists on a non-native support 0-1 ILP solver, it allows the integration of an efficient and already implemented parallel SAT solver, instead of using Minisat [12], to handle translated constraints concurrently. It is advantageous not only for its possibility of using efficient and well explored resources, like parallel conflict-driven SAT solvers described in this document, but also for the possibility of a more fine grained partition of the problem, partitioning not only the optimization search space but also the search space of each Task, which will imply a better management and usage of all resources in a parallel computing system.

References

1. Amir Aavani. Translating pseudo-boolean constraints into cnf. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT'11, pages 357–359, Berlin, Heidelberg, 2011. Springer-Verlag.
2. Tobias Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, July 2009.
3. Levent Aksoy, Ece Olcay Gunes, and Paulo Flores. Search algorithms for the multiple constant multiplications problem: Exact and approximate. *Microprocessors and Microsystems, Embedded Hardware Design (MICPRO)*, 34(5):151–162, March 2010.
4. Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Generic ilp versus specialized 0-1 ilp: an update. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, pages 450–457, New York, NY, USA, 2002. ACM.
5. Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Pbs: A backtrack search pseudo boolean solver. In *In Symposium on the theory and applications of satisfiability testing (SAT*, pages 346–353, 2002.
6. Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.
7. A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.
8. Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT), vol. 4, pages 75-97, Delft University*, 2008.
9. Wahid Chrabakh and Rich Wolski. Gradsat: A parallel sat solver for the grid, 2003.
10. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, July 1962.
11. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, July 1960.
12. Niklas Eén and Niklas Sörensson. An Extensible SAT-solver [ver 1.2].
13. Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.
14. Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.
15. E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat-solver. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '02, pages 142–, Washington, DC, USA, 2002. IEEE Computer Society.
16. Tracy Larrabee. *Efficient generation of test patterns using Boolean satisfiability.* PhD thesis, Stanford, CA, USA, 1990. UMI Order No. GAX90-24341.
17. Nuno Claudino Pereira Lopes, Levent Aksoy, Vasco Manquinho, and J. Monteiro. Optimally solving the mcm problem using pseudo-boolean satisability. Technical Report 43, INESC-ID, November 2010.
18. Luís Miguel Silveira Luís Gil, Paulo Flores. Pmsat: a parallel version of minisat. *Journal on Satisfiability, Boolean Modeling and Computation, Volume 6, pages 71-98*, November 2008.

19. Vasco M. Manquinho. Research Note On Using Cutting Planes in Pseudo-Boolean Optimization, 2005.
20. João P. Marques-Silva and Karem A. Sakallah. Boolean satisfiability in electronic design automation. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 675–680, New York, NY, USA, 2000. ACM.
21. João Marques-Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence*, EPIA '99, pages 62–74, London, UK, 1999. Springer-Verlag.
22. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.
23. Hossein M. Sheini and Karem A. Sakallah. Pueblo: A Modern Pseudo-Boolean SAT Solver. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 684–685, Washington, DC, USA, 2005. IEEE Computer Society.
24. João P. Marques Silva and Karem A. Sakallah. Grasp–a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.
25. Hantao Zhang. Sato: An efficient propositional prover. In *Proceedings of the 14th International Conference on Automated Deduction*, CADE-14, pages 272–275, London, UK, 1997. Springer-Verlag.
26. Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *J. Symb. Comput.*, 21:543–560, June 1996.