Distributed Solver for Maximum Satisfiability

Extended Abstract

Miguel Neves

INESC-ID/Instituto Superior Técnico
University of Lisbon, Portugal
miguel.neves@tecnico.ulisboa.pt

Maximum Satisfiability (MaxSAT) is an optimization version of *Boolean Satisfiability* (SAT). Many real world problems can be reduced to solving a MaxSAT instance, including problems of industrial interest. Therefore, MaxSAT is an important problem to solve and, in the recent years, many efficient algorithms have been developed, in order to tackle the challenge. Some of these algorithms are parallel, but no distributed algorithm has been developed so far.

This paper proposes two distributed algorithms for a cluster of processors under the user's control. These two algorithms were implemented and then experimented, studied and compared with one of the most effective sequential MaxSAT solvers that exist nowadays. The first algorithm is very similar to an existing successful parallel algorithm for MaxSAT, and the second one is an adaptation of an approach that proved to be very effective in sequential, parallel and distributed SAT solving.

Keywords: Boolean Satisfiability (SAT), Boolean Optimization, Maximum Satisfiability (MaxSAT), Distributed Search, Search Space Splitting, Guiding Paths with Lookahead.

1 Introduction

Maximum Satisfiability (MaxSAT) is an optimization problem that has been shown to be useful in many real-world problems of areas such as Computational Biology [20], Routing [21], Circuit Debugging [7], Software Debugging [5], among others. Much work has been done on algorithms for sequential MaxSAT during the last decade. Therefore, it has become harder to improve sequential algorithms. Besides, nowadays the computational power is increasing due to the predominance of multicore architectures instead of higher frequency CPUs. For this reason, recent work on MaxSAT solving has been done on the design of parallel solvers.

There is still room to improve the current state-of-the-art parallel MaxSAT algorithms, but the parallel approach has its drawbacks. All the cores of a typical computer access primary memory through the same BUS, and no two cores can use the BUS at the same time. Therefore, as the number of cores increases, so does contention on the access to primary memory, hindering the scalability of parallel algorithms. Machines with a BUS per core could be built, but this would further increase its cost. Moreover, as cores are added to a machine, it becomes much more costly.

Positive results have been obtained through recent work on parallel MaxSAT algorithms [11]. However, for the reasons mentioned previously, parallel algorithms hardly use more than 8 threads. Therefore, it is interesting to conduct research on distributed algorithms in order to further improve MaxSAT solving.

In section 2 we start by introducing some definitions and notations that will be used throughout the rest of this document. Then we follow up with the proposition of two distributed algorithms for MaxSAT in section **??**. Experimental results are presented in section 4. Finally, conclusions on the work developed are presented in section 5.

2 Preliminaries

In this section some definitions, notations and concepts that will be used throughout the rest of this document are introduced. The Boolean Satisfiability problem is described in section 2.1 and Maximum Satisfiability is described in section 2.2.

2.1 The Boolean Satisfiability (SAT) Problem

A *literal* is either a Boolean variable (*positive literal*) or the negation of a variable (*negative literal*). A *clause* is a disjunction of literals. A clause is called a *unit clause* if it has exactly one literal. Any propositional logic formula can be converted into an equivalent conjunction of clauses. A formula that is a conjunction of clauses is said to be in *conjunctive normal form* (CNF).

Given a propositional logic formula ϕ, an *assignment* is a partial mapping $\nu : X \to \{0, 1\}$ where X is the set of all variables in ϕ. Let x be a variable in X. The positive literal x is *satisfied* by ν if and only if $\nu(x) = 1$ and the negative literal $\neg x$ is *satisfied* by ν if and only if $\nu(x) = 0$; a clause is *satisfied* by ν if and only if at least one of its literals is *satisfied* by ν; a propositional logic formula in CNF is *satisfied* by ν if and only if all of its clauses are *satisfied* by ν. A propositional logic formula in CNF ϕ is *unsatisfiable* if and only if no assignment ν exists such that ν satisfies all the clauses in ϕ. An assignment is a *complete assignment* if $\nu(x)$ is defined for all the variables $x \in X$. An assignment is a *partial assignment* if it is not complete.

A SAT instance can be represented as a conjunction of clauses ϕ. A solution to the SAT instance ϕ is an assignment ν on the variables of ϕ that satisfies ϕ. Let $c_1, ..., c_n$ be clauses and $\phi = c_1 \wedge ... \wedge c_n$. The set of clauses $\{c_1, ..., c_n\}$ is an equivalent notation for ϕ. Throughout this document the set notation will be used for CNF formulas.

Given an unsatisfiable set of clauses ϕ, it may be required for the SAT solver to provide a more detailed explanation for the unsatisfiability of ϕ. A subset of clauses $\phi_C \subseteq \phi$ is an *unsatisfiable core* if ϕ_C is also unsatisfiable. ϕ_C is a *minimal unsatisfiable core* if all the subsets $\psi \subset \phi_C$ are satisfiable.

Conflict Driven Clause Learning (CDCL) SAT solvers [8] are among the most effective modern SAT solvers. These solvers apply a depth-first backtrack search where at each branching step a variable is chosen and a truth value is assigned to it. Each time a clause becomes unsatisfied (referred to as a conflict), CDCL SAT solvers backtrack up to a variable for which both truth values have not yet been tested and no clause is unsatisfied. Additionally, CDCL SAT solvers learn new clauses from conflicts through a process referred to as *conflict analysis* [10, 23]. It is widely known that learning new clauses that occur during the search can speed up the search process of Boolean solvers by pruning the search space. CDCL SAT solvers also apply a procedure called *unit propagation* in order to simplify the formula as the search progresses. *Unit propagation* generates new variable assignments from unit clauses.

Sometimes it is useful to invoke a SAT solver on the same instance multiple times but with different preset assignments to specific variables. This is achieved through the use of assumptions. Assumptions can be passed to a SAT solver when invoked. An assumption is a literal that only holds on that specific invocation of the SAT solver. Therefore, an assumption is very similar to a unit clause, but unlike the latter, assumptions are discarded after the search is finished. Assumptions should be used when one needs to enforce certain variable assignments only on a specific invocation to the SAT solver.

2.2 The Maximum Satisfiability (MaxSAT) Problem

A MaxSAT instance can be encoded in the same way as a SAT instance, but in the MaxSAT case, given a set of clauses ϕ, the goal is to determine an assignment ν over the variables in ϕ that maximizes the number of satisfied clauses in ϕ. Such an assignment is defined as an *optimum solution* (or *optimum assignment*). This version of MaxSAT is referred to as *unweighted MaxSAT*. We can also look at MaxSAT as a minimization problem instead of maximization. In this case, the goal is to minimize the number of unsatisfied clauses. Throughout the rest of this document it is assumed that MaxSAT is defined as a minimization problem.

Let X the set of all variables in ϕ, ν an assignment over ϕ and c a clause such that $c \in \phi$. We denote as $\nu(c)$ the value of clause c under assignment ν, in other words: $\nu(c) = 0$ if and only if ν unsatisfies c, $\nu(c) = 1$ otherwise.

Sometimes, in many real world applications, it is required that some of the clauses must be satisfied. Such a variation of MaxSAT is encoded with two sets of clauses, a set ϕ_H of *hard clauses* and a set ϕ_S of *soft clauses*, where the hard clauses are the ones that must be satisfied. This variation of MaxSAT is referred to as *partial MaxSAT* [2] and an optimum solution to such an instance is an assignment ν over the variables in ϕ_H and ϕ_S that satisfies all the clauses in ϕ_H and minimizes the number of unsatisfied clauses in ϕ_S.

With the addition of hard clauses to the MaxSAT problem, it becomes relevant to distinguish between valid and invalid assignments. It also becomes possible for a MaxSAT instance to be unsatisfiable. An assignment ν is *valid* for (ϕ_H, ϕ_S) if, and only if, ν satisfies all of the clauses in ϕ_H. Otherwise, ν is *invalid* for (ϕ_H, ϕ_S). (ϕ_H, ϕ_S) is *unsatisfiable* if, and only if, ϕ_H is unsatisfiable.

Several state-of-the-art MaxSAT solvers apply algorithms that iteratively call a SAT solver with a relaxed version of the problem instance. Such algorithms use constraints that restrict how many literals of a given set can be satisfied simultaneously. These constraints are called *cardinality constraints*. However, state-of-the-art SAT solvers do not offer native support for cardinality constraints. Therefore, it is necessary to encode those constraints into CNF.

Let $l_1, ..., l_N$ be literals and k a non negative integer. A *cardinality constraint* is a constraint with the following form:

$$\sum_{i=1}^{N} l_i \leq k. \tag{1}$$

Cardinality constraints as described above are also referred to as *at-most-k constraints*. Note that constraints of the form $\sum_{i=1}^{N} l_i \geq k$, referred to as *at-least-k constraints*, are equivalent to $\sum_{i=1}^{N} \neg l_i \leq N - k$. Given an at-most-k constraint C, we denote as $[C]_{CNF}$ a CNF encoding for C.

A CNF encoding for at-most-k constraints must be sound and complete. Given an at-most-k constraint C and a CNF encoding $[C]_{CNF}$ for that constraint, the set of all assignments that satisfy C must be exactly the same that satisfy $[C]_{CNF}$. A summary of encodings for cardinality constraints can be found in the literature [13, 14]. The definition of satisfaction for at-most-k constraints is the following. Given an integer k, an at-most-k constraint C and an assignment ν, we say that ν *satisfies* C if no more than k of the literals in C are satisfied by ν.

Most iterative MaxSAT algorithms [16] establish an upper/lower bound on the value of the optimum solution and continuously refine this bound until the optimum is found. This is done by relaxing the soft clauses with fresh relaxation variables, encoding an at-most-k constraint over those variables and invoking a SAT solver on the resulting formula. The right hand side of the at-most-k constraint is the current bound. If the SAT solver returns that the formula is satisfiable, then it is possible to unsatisfy k soft clauses or less. On the other hand, if the SAT solver returns unsatisfiable, then the optimum solution must unsatisfy more than k soft clauses.

3 Distributed MaxSAT

This section presents the two distributed algorithms that were studied and implemented. The first algorithm is an adaptation of the parallel Search Space Splitting approach, proposed by Martins et al. [11, 14]. The second algorithm is based on the guiding paths splitting strategy [22, 4], which was shown to improve the performance of sequential SAT solvers and to be successful in parallel and distributed SAT algorithms.

In both algorithms there are two main types of processes: a single mediator process and multiple slave processes. The mediator process is responsible for splitting the problem instance into tasks, assigning those tasks to the slave processes and handling communication. The slave processes' main purpose is to wait for a task from the mediator, process the received task and send the result back to the mediator.

3.1 Search Space Splitting Algorithm

Given n processes, the Search Space Splitting algorithm is composed by 1 mediator, 1 core guided process, 1 linear search process and $n - 3$ local linear search processes.

The mediator process is depicted in figure 1. Before assigning tasks to the remaining processes, the mediator initializes an sorted set that will store, through the execution of the algorithm, the best lower and upper bounds determined so far and the local bounds currently being searched by each local linear search process. The mediator starts by computing an initial upper bound, by invoking the SAT solver on the CNF formula ϕ_H. If ϕ_H is not satisfiable, then the algorithm terminates immediately returning that the instance is not satisfiable.

If ϕ_H is satisfiable, then the mediator computes the number of soft clauses satisfied by the model returned by the SAT solver, chooses that value as the initial upper bound and adds it to the set. The initial lower bound is 0 and the initial local bounds are computed as follows: given k local linear search processes, $p_1, ..., p_k$, and an initial upper bound μ, the initial local bound assigned to process p_i is given by $i \times \lfloor \frac{\mu}{k+1} \rfloor$. After adding these bounds to the bounds set, each of them is sent to the corresponding local linear search process, and $\mu - 1$ is sent to the linear search process.

Example 3.1. Let $\mu = 37$ be an initial upper bound. Given 5 local linear search processes, the initial bounds set is $B = \{0, 6, 12, 18, 24, 30, 37\}$.

Next, the mediator enters a loop, waiting for upper and lower bounds from the slave processes and assigning new bounds to them until the optimum has been found. If the mediator receives a lower bound λ from a process p, all values smaller than λ are removed from the bounds set, and λ is added to the set. Otherwise, if the mediator receives an upper bound μ, all values larger than μ are removed and μ is added to the set. Next, a new bound is computed and sent to process p.

Let $B = \{b_0, b_1, ..., b_k, b_{k+1}\}$ be the current bounds set. If p is the core guided search process, $b_1 + 1$ is sent to p. If p is the linear search process, $b_k - 1$ is sent to p instead. If p is a local linear search process, then the mediator chooses a pair (b_{m-1}, b_m) of contiguous values such that $b_m - b_{m-1} \geq b_j - b_{j-1}$ for all $1 < j \leq k$. A new local bound b is computed with the expression $\frac{b_m + b_{m-1}}{2}$. b is then added to B and sent to process p.

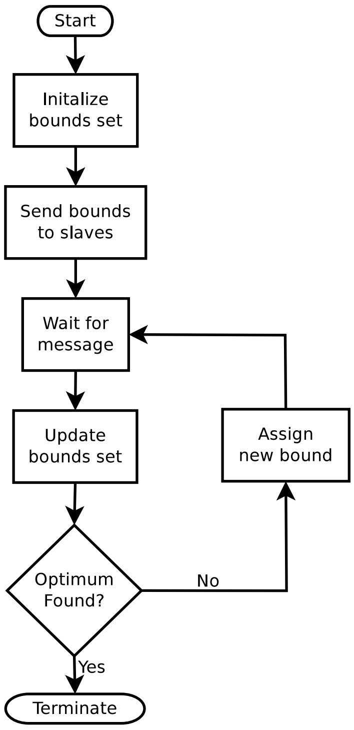
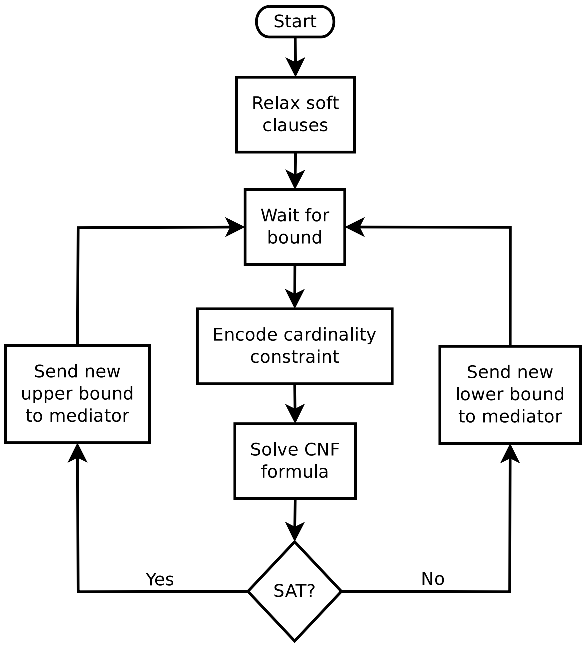


Figure 1: Search Space Splitting mediator process.



Start
Relax soft clauses
Wait for bound
Encode cardinality constraint
Send new upper bound to mediator
Send new lower bound to mediator
Solve CNF formula
Yes
SAT?
No

Figure 2: Search Space Splitting linear search process.

Example 3.2. Let $B = \{5, 12, 22, 27, 40\}$ be a bounds set. Suppose that a message from p_1 is received stating that 26 is an upper bound. The new bounds set will be $B_1 = \{5, 12, 17, 22, 26\}$ where 17 is p_1's new local bound. Suppose that, afterwards, a message from p_2 is received stating that 19 is a lower bound. The new bounds sets will be $B_2 = \{19, 22, 24, 26\}$ where 24 is p_2's new local bound.

Figure 2 depicts the behavior of the linear search and local linear search processes. These processes start by relaxing all the soft clauses with new relaxation variables. Then, each process enters an infinite loop, waiting for a bound b from the mediator and computing if b is an upper bound or a lower bound. For the linear search process, b will always be the current upper bound minus 1, but for the local linear search processes, b is a local bound chosen by the mediator as described above.

Let ϕ_H be the set of hard clauses, ϕ_S^r the set of relaxed soft clauses and R the set of relaxation variables. A linear search process p determines if b is an upper or a lower bound by invoking the SAT solver on the CNF formula $\phi_H \cup \phi_S^r \cup [\sum_{r \in R} r \leq b]_{CNF}$. After solving b, p notifies the mediator of the result. The mediator responds with a new bound to be solved $b^{'}$. If b was determined to be a lower bound, $b^{'}$ will be larger than b. On the other hand, if b is an upper bound, $b^{'}$ will be smaller than b.

If $b^{'} < b$, then it is possible to obtain an equivalent CNF encoding for $\sum_{r \in R} r \leq b^{'}$ from $[\sum_{r \in R} r \leq b]_{CNF}$ without removing clauses from the latter. This can be achieved merely by assigning truth values to certain auxiliary variables of the CNF encoding for $\sum_{r \in R} r \leq b$. Therefore, we only need to add the corresponding unit clauses to $[\sum_{r \in R} r \leq b]_{CNF}$ to obtain $[\sum_{r \in R} r \leq b^{'}]_{CNF}$. Martins et al. [12] take advantage of this property to improve the performance of linear search algorithms by avoiding the need to rebuild the SAT solver. This way, all clauses learned so far by the SAT solver are maintained between invocations.

However, the property above is not true if $b^{'} > b$. One could remove the clauses of $[\sum_{r \in R} r \leq b]_{CNF}$ from the SAT solver and then encode $\sum_{r \in R} r \leq b^{'}$, but this may invalidate a subset of the learned clauses stored in the SAT solver. Therefore, in the local linear search processes, when b is an upper bound the SAT solver is maintained and the new cardinality constraint encoding is obtained from the previous one, but when b is a lower bound the SAT solver is rebuilt.

The behavior of the core guided process is depicted in figure 3. That behavior is very similar to the sequential MaxSAT algorithm presented by Marques-Silva and Planes [9], the main difference being the additional exchange of messages with the mediator. The core guided process starts by adding relaxation variables to the soft clauses. Then it enters a cycle of invocations to the SAT solver with unsatisfiable CNF formulas until a satisfiable formula is found.

The cycle starts by invoking the SAT solver with the complements of the relaxation variables as assumptions. The assumptions are used to force the soft clauses to be satisfied. If it is not possible, then the algorithm retrieves a conflicting subset of the assumptions and relaxes the corresponding soft clauses.

The algorithm also maintains the best lower bound computed locally and the best lower bound that was

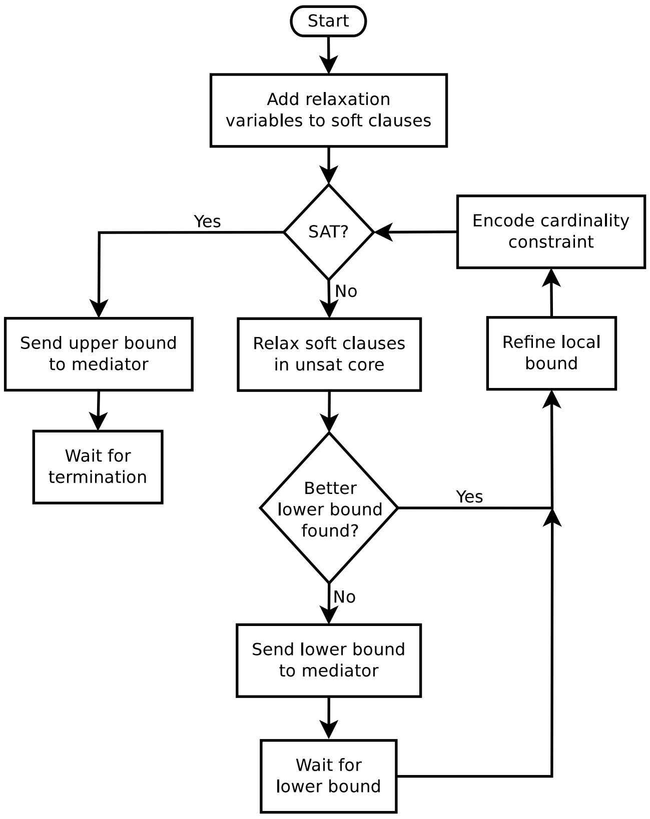

Start
Add relaxation variables to soft clauses
SAT?
Yes
No
Encode cardinality constraint
Send upper bound to mediator
Relax soft clauses in unsat core
Refine local bound
Wait for termination
Better lower bound found?
Yes
No
Send lower bound to mediator
Wait for lower bound

Figure 3: Search Space Splitting core guided process.

received from the mediator so far. The local lower bound is initialized as 0. If the local lower bound is better than the lower bound received from the mediator, then the local lower bound is sent to the mediator, and the core guided process waits for a new lower bound. The local lower bound is then refined, and a cardinality constraint is encoded into the CNF formula over the current set of relaxation variables and with the new local lower bound as right hand side.

Finally, the cycle restarts by invoking the SAT solver again, but the assumptions corresponding to the soft clauses that were relaxed are excluded. The cycle continues until either the SAT solver returns that the CNF formula is satisfiable or until the process is terminated by the mediator. In the first case, the core guided process sends the local bound to the mediator stating that it is an upper bound and awaits termination.

Example 3.3. Let ϕ_W be the current working formula and $\Lambda = \{\neg r_1, \neg r_2, \neg r_3, \neg r_4\}$ the current set of assumptions. Suppose that the SAT solver, after being invoked on ϕ_W under Λ, returns the conflict $\{\neg r_1, \neg r_4\}$. The new set of assumptions will be $\Lambda^{'} = \{\neg r_2, \neg r_3\}$.

An optimum is found when the lower bound λ and the upper bound μ stored in the mediator meet the condition $\lambda + 1 = \mu$. When this happens, the mediator aborts the execution of the remaining processes and terminates, returning μ as the optimum.

Two versions of this algorithm were implemented and tested. In one version, a local linear search process p rebuilds the SAT solver whenever it determines that a given local bound b is a lower bound. This is because the new local bound $b^{'}$ that will be assigned to p is guaranteed to be larger than b.

The second version makes use of assumptions and of the fact that the mediator computes an initial upper bound to only build the SAT solver once and never have to rebuild it again. This second version is referred to as the Fully Incremental Search Space Splitting algorithm.

Suppose that process p has an upper bound μ encoded in its current cardinality constraint. The mediator assigns a new local bound b to p. Instead of adding a set U of unit clauses to the SAT solver, since p does not know if b is an upper or a lower bound, p invokes the SAT solver with U as the set of assumptions. This updates the cardinality constraint without altering the SAT instance stored in the solver. If b is an upper bound, then p adds unit clauses to its SAT solver to update the cardinality constraint from μ to $b - 1$. If b is a lower bound, the SAT solver remains unchanged. It has been observed that always invoking the same SAT solver incrementally boosts the performance of sequential MaxSAT algorithms [12].

3.2 Guiding Paths with Lookahead Algorithm

A distributed algorithm for SAT based on guiding paths has already been proposed and evaluated with PaMiraXT [19]. Consider the search space of a SAT instance as a binary tree. Each tree node corresponds to a variable and each of its edges corresponds to an assignment to that variable (0 or 1). A path from the root node to a leaf node corresponds to a complete assignment. The SAT problem can be interpreted as, given a SAT instance, determine if a path exists in its search tree from the root to a leaf node that corresponds to a satisfying assignment. What the guiding paths approach does is to split the search tree into sub-trees and assign each of them to a distinct process. In partial MaxSAT, the goal is to find a path from the root to a leaf node corresponding to an assignment that satisfies all hard clauses and minimizes the number of unsatisfied soft clauses.

Heule et al. [4] have proposed recently an improved parallel SAT algorithm based on guiding paths that initially uses a *lookahead* solver to split the search tree in guiding paths. Lookahead solvers apply sophisticated reasoning at each branching step in order to guide the search more effectively. The algorithm described throughout the rest of this section is an extension of the previous approach to distributed MaxSAT. Given n processes, the following algorithm is composed by 1 mediator and $n-1$ guiding path solver processes.

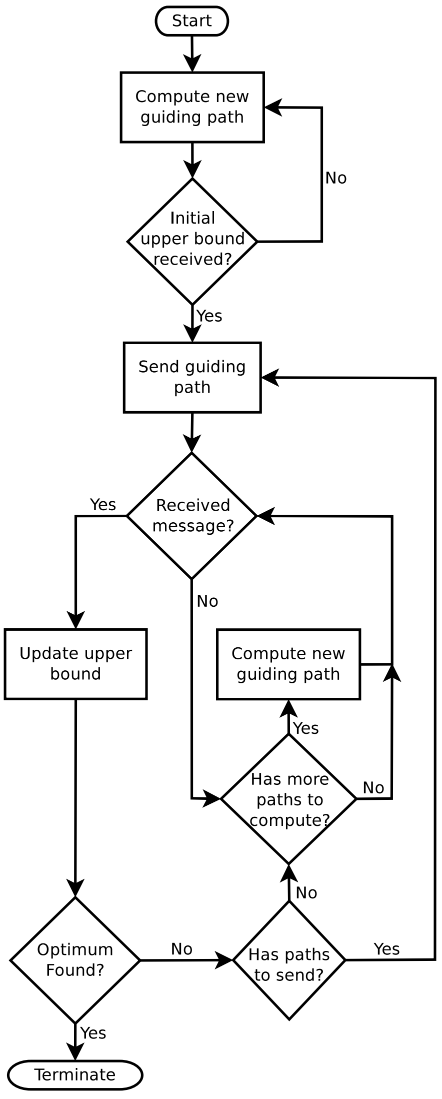


Figure 4: Guiding Paths with Lookahead mediator process.

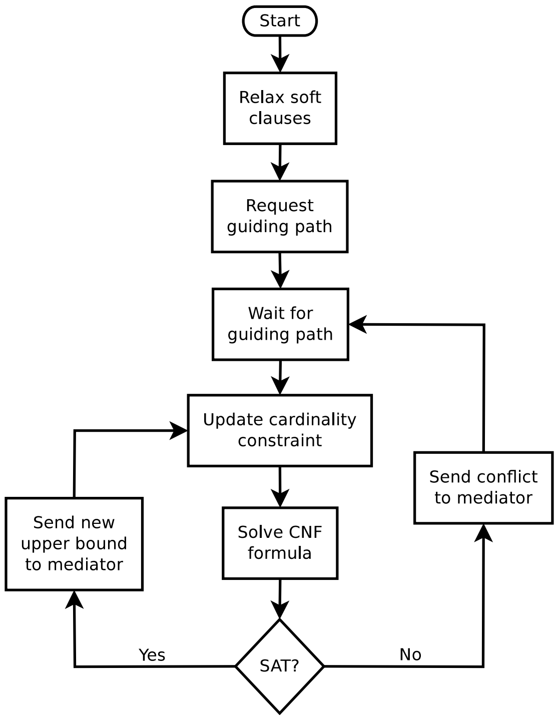


Figure 5: Guiding Paths with Lookahead guiding path solver process.

The behavior of the mediator process is depicted in figure 4. Initially, the mediator computes and stores guiding paths until an initial upper bound is received from one of the slave processes. In the Guiding Paths algorithm one of the slave processes is responsible for computing the initial upper bound, allowing the mediator to start computing guiding paths in parallel. When the initial upper bound is received, the mediator starts fulfilling the requests from the slave processes, sending them guiding paths to solve coupled with the best upper bound computed so far. The mediator continues computing more guiding paths while waiting for messages from the slaves.

Algorithm 1: Guiding Path generation algorithm [4]

```
Procedure GenerateGuidingPaths($\phi$, $C$, $D$, $I$, $\theta$)
    IncrementCutoff($\theta$)
    $(\phi, I) \leftarrow$ Propagate($\phi$, $D$, $I$)
    if $\phi$ is unsatisfied by $D \cup I$ or $|D| + \log_2 |\phi| > 25$ then
        DecrementCutoff($\theta$)
    end
    if $\phi$ is unsatisfied by $D \cup I$ then
        AnalyzeAndLearn($\phi$, $D$, $I$)
        return $C$
    end
    if $|D| \times |D \cup I| > \theta \times |$ Vars($\phi$) $|$ then
        return $C \cup \{D\}$
    end
    $x \leftarrow$ ChooseVariable($\phi$, $D$, $I$)
    $l \leftarrow$ ChoosePolarity($\phi$, $x$)
    $C \leftarrow$ GenerateGuidingPaths($\phi$, $C$, $D \cup \{l\}$, $I$, $\theta$)
    return GenerateGuidingPaths($\phi$, $C$, $D \cup \{\neg l\}$, $I$, $\theta$)
```

The pseudo-code for the guiding path generation procedure is presented in algorithm 1. This procedure receives as input a CNF formula ϕ, the set C of guiding paths computed so far by the procedure, the current partial assignment D, the set I of literals that are implied by the partial assignment D and a cutoff value θ. Formula ϕ used for guiding path generation in the MaxSAT algorithm is the set of hard clauses. It was observed experimentally that considering just the set of hard clauses in the guiding path generation procedure provided better results than considering both the hard clauses and the soft clauses. The functions in the procedure work as follows:

- *IncrementCutoff*(θ) increases the cutoff value θ destructively[1]. In practice, θ is incremented by 5%, like in the literature [4];
- *DecrementCutoff*(θ) decreases the cutoff value θ destructively. In practice, θ is decremented by 30%, like in the literature [4];
- *Propagate*(ϕ, D, I) applies unit propagation to formula ϕ. New literals are added to set I that represent assignments implied by the partial assignment D on ϕ;
- *AnalyzeAndLearn*(ϕ, D, I) applies conflict analysis [10, 23] to learn a new clause;
- *Vars*(ϕ) returns the set of all variables in ϕ;
- *ChooseVariable*(ϕ, D, I) chooses a new variable heuristically to be added to the partial assignment D. The addition of a new variable assignment to D is referred to as a *decision*. A rank is assigned to each variable not in $D \cup I$ and the one with the highest rank is chosen;
- *ChoosePolarity*(ϕ, x) chooses heuristically which truth value will be assigned first to variable x. Returns either x or $\neg x$.

The algorithm starts by incrementing the cutoff value θ (line 2). This is done to prevent θ from being reduced too much by the decrement rule in line 4 and generating too small guiding paths. The initial cutoff value is 1000 as specified by Heule et al. [4]. Next, unit propagation is applied to simplify ϕ and update set I (line 3). The algorithm then checks if ϕ is unsatisfied by the current partial assignment D and implied assignments in I (line 4). In this case, θ is decremented (line 5). θ is also decremented if $|D| + \log_2 |\phi| > 25$ (line 4). This rule takes into account the number of decisions and the size of the CNF formula in order to prevent the guiding path generation process of going too deep in the search tree and generating too many guiding paths that are also too large. Heule et al. [4] propose, for SAT, a value of 30 for the right hand side of the condition in line 4, but we observed experimentally that a value of 25 provided better results for MaxSAT.

If ϕ is unsatisfied, the procedure applies conflict analysis [10, 23] and learns a new clause (line 8), like in a CDCL SAT solver [8]. This may prevent the procedure from generating guiding paths that unsatisfy ϕ. If ϕ is not unsatisfied the algorithm checks if the cutoff has been triggered (line 11). If so, the current partial assignment D is returned as a guiding path. The cutoff condition takes into account the number of decisions and the total number of assignments, explicit and implied, in the current node of the search tree.

[1]Destructive modifications to a given parameter are globally visible. Therefore, destructive modifications are propagated upwards in the call chain of the procedure.

If the cutoff is not triggered, then an unassigned variable x is chosen heuristically to be added to D (line 14). Another heuristic is used to decide which truth value will be tested first (line 15). Algorithm 1 is then repeated for x and $\neg x$ (lines 16 and 17). Heule et al. [4] propose two different heuristics for ranking a variable, and both require an heuristic value to be computed for x and $\neg x$, denoted as $eval(x)$ and $eval(\neg x)$. The rank of a variable x is $eval(x) \times eval(\neg x)$ and ties are broken by $eval(x) + eval(\neg x)$.

Given a variable x, $eval_{var}(x)$ ($eval_{var}(\neg x)$) denotes the number of variables that are assigned by unit propagation after the assignment $x = 1$ ($x = 0$). The second heuristic, denoted as $eval_{cls}$, weights clauses depending on their length. $eval_{cls}(x)$ ($eval_{cls}(\neg x)$) is the sum of the weights of the clauses that are reduced[2] by the assignment $x = 1$ ($x = 0$) but are not satisfied. The clauses are weighted in a way such that a clause with length k has a weight five[3] times larger than a clause with length $k + 1$.

Example 3.4. Let $\phi = \{(x_1 \vee x_2 \vee x_3), (x_2 \vee \neg x_3), (\neg x_1 \vee x_2)\}$ be a CNF formula. In ϕ, $eval_{var}(\neg x_2) = 2$ and $eval_{cls}(\neg x_3) = 6$ (clauses with length 2 and 3 have weights 5 and 1 respectively if 3 is the maximum clause size).

A variation of the $eval_{cls}$ heuristic, which will be referred to as $eval_{wl}$, was implemented and tested. The only difference is, given a literal l, instead of considering all the clauses in ϕ, only the clauses "watching" [17] literal l are considered in the computation of $eval_{wl}(l)$. The clauses "watching" l are a subset of the clauses that contain l. In state-of-the-art SAT solvers, each clause "watches" only two of its literals at each instant. Suppose that a given clause c is watching literals l_1 and l_2 and that l_1 and l_2 are neither satisfied or unsatisfied. If l_1 becomes unsatisfied, then another non-unsatisfied literal $l_3 \in c$ is chosen to be watched instead of l_1. If l_3 is satisfied, then c is satisfied. If no such literal like l_3 exists, then the clause has become unit and the variable in l_2 is assigned accordingly. If the SAT algorithm backtracks, the watched literals remain unchanged. Watched literals are a lazy data structure that greatly improve the performance of SAT solvers. It was observed experimentally that the $eval_{wl}$ heuristic was more effective than the $eval_{var}$ and $eval_{cls}$ heuristics in this distributed MaxSAT algorithm.

After choosing a variable x, now a truth value must be chosen to be tested first. For SAT, Heule et al. [4] aim to improve the performance on satisfiable instances. Therefore, they choose to explore first the branch $x = 1$ if $eval(x) < eval(\neg x)$. However, in MaxSAT all guiding paths will inevitably result in one unsatisfiable call to the SAT solver. Besides, MaxSAT is an optimization problem, not a decision problem. We choose the direction based on the number of clauses that will be unsatisfied after assigning x. The branch to be explored first is the one that unsatisfies a smaller number of soft clauses. Ties are broken choosing the direction that satisfies more soft clauses. The rationale for this is that the branch that unsatisfies less soft clauses is more likely to reach an upper bound closer to the optimum, and since upper bounds are shared among slave processes, this may help skip unnecessary satisfiable calls to the SAT solver.

However, the polarity heuristic is only significant at the beginning of the search, when the mediator is generating guiding paths, because the mediator sorts guiding paths as they are generated. Therefore, in the long run, the sorting criterion dominates the polarity heuristic. Initially, the criterion used to sort the guiding paths was very similar to the polarity heuristic, sorting guiding paths by number of unsatisfied soft clauses. But it was observed experimentally that sorting guiding paths by the total of assigned variables (decision and implied), when the path is returned, lead to better results. The priority is given to the least restricting guiding path and ties are broken by choosing the one that was generated first.

Example 3.5. Suppose that algorithm 1 returns the guiding paths g_1, g_2, g_3 and g_4, exactly in this order. The dimensions of D when each path was returned were 10, 8, 11 and 7 respectively, and the dimensions of I were 100, 92, 97 and 103. The priority values for each of the paths are $p(g_1) = 10 + 100 = 110$, $p(g_2) = 8 + 92 = 100$, $p(g_3) = 11 + 97 = 108$ and $p(g_4) = 7 + 103 = 110$. $p(g_1)$ and $p(g_4)$ are the same, but g_1 was returned first. Therefore, the resulting sorted set of guiding paths will be $C = \{g_1, g_4, g_3, g_2\}$.

The behavior of the guiding path solver process is depicted in figure 5. The computation of the initial upper bound was not included in figure 5 for simplicity. These processes apply an adaptation of the Linear Search algorithm described in the literature [1]. A solver process starts by relaxing the soft clauses and requesting a guiding path from the mediator. Next, it enters a loop waiting for guiding paths to solve given by the mediator and applying linear search on those guiding paths.

A guiding path message includes the best upper bound μ found so far. Therefore, when a guiding path is received, the process updates the current cardinality constraint to $\mu - 1$. Having received a guiding path g, the solver process then invokes the SAT solver with g as the set of assumptions. If the SAT solver returns that the formula is satisfiable, the process sends the new upper bound to the mediator, refines the upper bound in the cardinality constraint and continues applying linear search. However, if the SAT solver returned that the formula is unsatisfiable, then a conflicting subset of g is retrieved from the SAT solver and sent to the mediator.

[2]A clause is reduced whenever one of its literals is assigned value 0.

[3]Same value used for SAT [4].

An optimum is found when all guiding paths have been solved or when an empty conflict is obtained. If a given guiding path g results in an empty conflict, then the reason for unsatisfiability does not depend on the assumptions. Therefore, the reason for unsatisfiability is either the cardinality constraint or a conflict in the hard clauses. If there is a conflict in the hard clauses, it is detected in the computation of the initial upper bound. On the other hand, if the reason is the cardinality constraint, then the constraint's right hand side is a lower bound and the optimum was found.

4 Experimental Results

In this section, both distributed algorithms presented in section 3 are evaluated. The distributed algorithms are compared with the sequential solver OpenWBO [15]. OpenWBO was configured to use Glucose 2.3 as its SAT solver, the same used in the distributed algorithms. OpenWBO implements two different algorithms: a Linear Search [1, 6] and a Core Guided [3] algorithm. The algorithms are compared in terms of number of solved instances and run time.

The algorithms were evaluated running on the instances of the industrial and crafted categories of the MaxSAT Evaluation of 2013[4]. Ideally, the algorithms should be run on each instance various times and it only should be considered solved if it was solved on more than half of the runs, but due to time constraints the results throughout this chapter were obtained with just one run per instance. The algorithms were run with a timeout of 1800 seconds (wall clock time) and with a memory limit of $4 \times N$ Gb where N is the number of processes. The execution of the algorithms was monitored using the *runsolver* application [18]. The tests were conducted on 4 machines, each with two Intel Xeon E5-2630V2 processors (2.6GHz, 6C/12T) with 64 Gb of RAM, running Ubuntu 13.10.

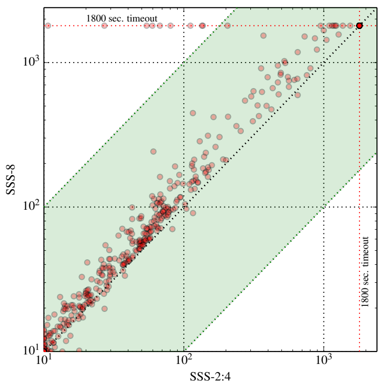


Figure 6: SSS-8 vs. SSS-2:4 for industrial instances.

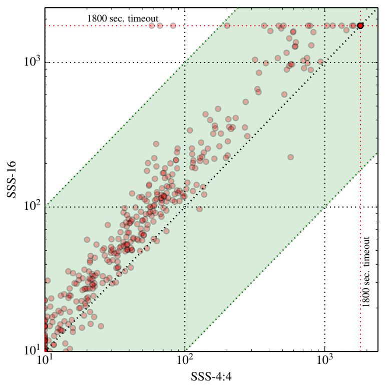


Figure 7: SSS-16 vs. SSS-4:4 for industrial instances.

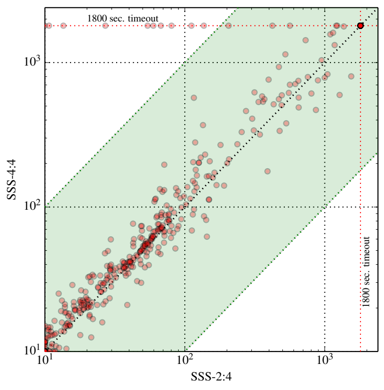


Figure 8: SSS-2:4 vs. SSS-4:4 for industrial instances.

Figure 6 is a scatter plot comparing the run times on the industrial category of the Search Space Splitting algorithm running with 8 processes on the same machine (SSS-8) with the run times of running the same algorithm with 8 processes, 2 per machine (SSS-2:4). Instances that are solved trivially by both solvers (in less than 10 seconds) were excluded from the plots.

It is clear that splitting the processes among multiple machines improves the performance of the algorithm compared to running all processes in the same machine. More 18 instances were solved by SSS-2:4 compared to SSS-8, and more 12 instances were solved by SSS-4:4 than by SSS-16. It was observed that some of these additional instances that were solved were not solved in a single machine due to the memory limit being exceeded. In fact, 10 of the 18 additional instances solved by SSS-2:4 were instances where SSS-8 exceeded the memory limit. However, this happens because *runsolver* ([18]) enforces the memory limit only on the processes in the host machine. *runsolver* has no means to monitor the amount of memory that is used by processes in other machines.

Figure 8 shows the run times of SSS-2:4 compared with SSS-4:4. SSS-2:4 slightly outperforms SSS-4:4 in terms of run times and SSS-2:4 solves 16 more instances than SSS-4:4, and 14 of those 16 are not solved by SSS-4:4 due to the memory limit being exceeded. In fact, the Search Space Splitting algorithm with 4 processes in the same machine (SSS-4) solves 3 more instances than SSS-4:4. It was observed that there were 4 instances that were solved by SSS-4 but not by SSS-4:4 due to the memory limit being exceeded by SSS-4:4.

This most likely happened because MPI assigned the processes with the largest initial upper bound to the host machine and, as stated above, runsolver only monitors the processes in the host machine. The larger the

[4]http://maxsat.ia.udl.cat/

right hand side of a cardinality constraint, the more memory will be necessary to encode that constraint into CNF. It was observed experimentally that changing MPI's scheduling policy to assign processes to machines differently caused one of the instances that were not solved by SSS-4:4, due to the memory limit, to become solved.

Figures 6, 7 and 8 show that increasing the number of machines improves the performance of the distributed algorithm, but increasing the number of processes per machine may have a hindering effect. The main reason for this is the contention on the access to primary memory. Computers usually have a single BUS connecting processors to primary memory. Therefore, if one processor is accessing primary memory and another one gets a miss on cache, the second one has to wait for the first to finish before accessing primary memory as well.

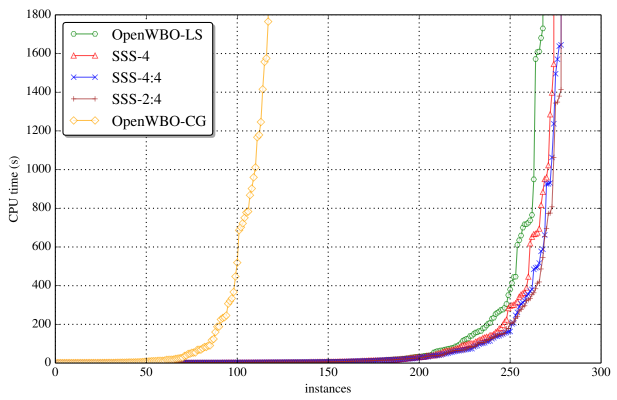


Figure 9: Cactus plot with the run times for the instances of the crafted category and the algorithms OpenWBO's core guided and linear search algorithms, and the Search Space Splitting algorithm with 4 processes in the same machine (SSS-4), 8 processes split among 4 machines (SSS-2:4) and 16 processes split among 4 machines (SSS-4:4).

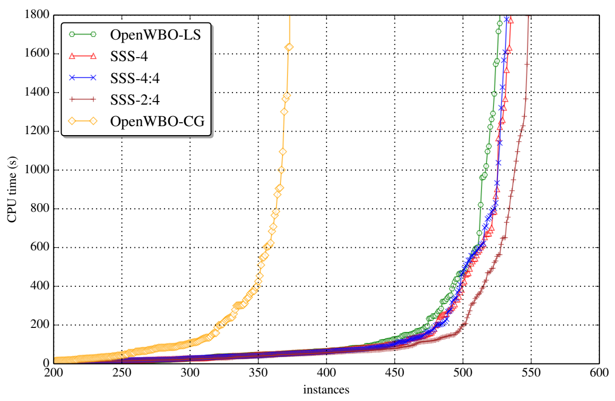


Figure 10: Cactus plot with the run times for the instances of the industrial category and the algorithms OpenWBO's core guided and linear search algorithms, and the Search Space Splitting algorithm with 4 processes in the same machine (SSS-4), 8 processes split among 4 machines (SSS-2:4) and 16 processes split among 4 machines (SSS-4:4).

Figures 9 and 10 show cactus plots with the run times of OpenWBO's core guided (OpenWBO-CG) and linear search (OpenWBO-LS) algorithms compared with the run times of SSS-4, SSS-2:4 and SSS-4:4.

The Search Space Splitting algorithm clearly outperforms OpenWBO in the crafted category. There is a slight increase in performance when executing the distributed algorithm with 8 processes compared to 4, solving 4 more instances. However, there was no gain when increasing the number of processes to 16, solving the same number of instances as SSS-2:4. A similar behavior can be observed in the industrial category, except that the performance of the Search Space Splitting algorithm actually starts degrading with 16 processes. However, most of the instances that were solved by SSS-2:4 and not by SSS-4:4 were due to the memory limit being exceeded by SSS-4:4, as mentioned previously.

The basic Search Space Splitting (SSS) algorithm was also compared with the fully incremental (FYSSS) version. We expected FYSSS to outperform SSS at least in the industrial instances, but the results of both algorithms were very similar for crafted instances and industrial instances. It was observed that incrementality [12] improved the individual performance of a local linear search process, but, in the long run, this can lead to the search space being explored differently. For example, in one version a local linear search process may end up solving local bounds that are harder than the bounds that it would solve in the other version.

Figures 11 and 12 show the run times of the Guiding Paths algorithm (GP-2:4) with 8 processes, 2 per machine, compared with OpenWBO-LS, for crafted and industrial instances respectively. We can clearly see that GP-2:4 is much more effective in the crafted category than in the industrial category. As stated in section 3.2, lookahead solvers for SAT are known to be not very effective in industrial instances. The guiding path generation generator (algorithm 1) works in the same way as a lookahead solver, the difference being that the generator partitions the search tree instead of looking for a solution. Therefore, it was expected that the Guiding Paths algorithm would be much more effective in crafted instances than in industrial instances.

GP-2:4 outperforms OpenWBO-LS in most crafted instances. In fact, we can see in figure 11 various instances that GP-2:4 managed to solve around 10 times faster than OpenWBO-LS. However, there are also a few instances on which GP-2:4 was around 10 times slower than OpenWBO-LS. This can happen if the guiding paths generated by algorithm 1 force the SAT solver to traverse branches of the search tree that are hard to solve and that would not be traversed if the search tree had not been partitioned. Therefore, a lot can be gained in performance by using a Guiding Paths with Lookahead algorithm, but it depends on which guiding

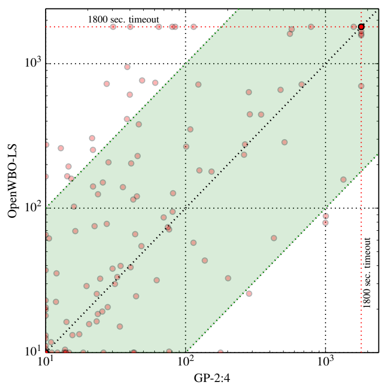


Figure 11: GP-2:4 vs. OpenWBO-LS for crafted instances.

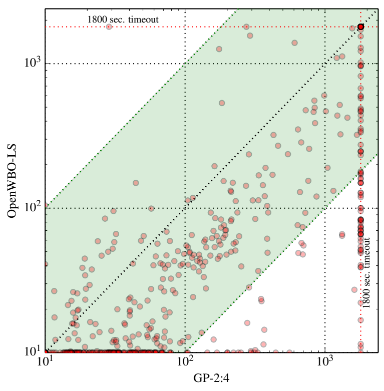


Figure 12: GP-2:4 vs OpenWBO-LS for industrial instances.

paths are being generated.

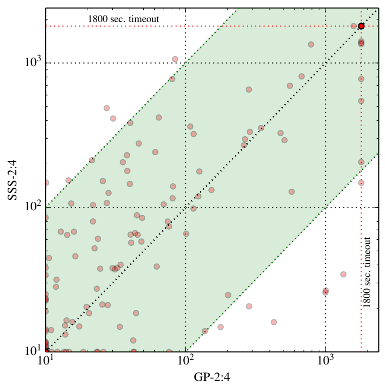


Figure 13: GP-2:4 vs. SSS-2:4 for crafted instances.

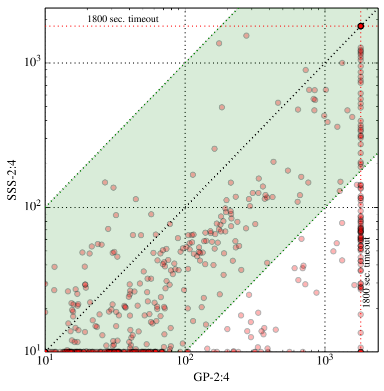


Figure 14: GP-2:4 vs. SSS-2:4 for industrial instances.

Figures 13 and 14 show the run times of GP-2:4 compared with the run times of SSS-2:4. The results are very similar to what was observed in figures 11 and 12. GP-2:4 is clearly outperformed by SSS-2:4 in the industrial category. However, GP-2:4 is slightly more effective in the crafted category than SSS-2:4, solving 1 more instance.

5 Conclusions

MaxSAT is a NP-hard problem with many applications in the real world. Several sequential algorithms have been proposed and improved in the recent years, making it possible to solve many instances of interest quite fast. Still, there exist instances that are very hard to solve in reasonable time, and so much work has been done on MaxSAT so far that it becomes harder to further improve sequential MaxSAT solving. For this reason, some parallel algorithms have been proposed and studied, but this kind of algorithms has very limited scalability due to BUS contention on access to primary memory. Also, a machine becomes much more costly as the number of processor cores increases.

Distributed algorithms are more likely to scale than parallel algorithms, as long as the amount of communication required between threads is not heavy, since threads are spread among computation nodes, each with its own independent primary memory. For this reason, we implemented two different distributed approaches for solving MaxSAT and studied how these approaches fare in comparison with one of the most effective sequential MaxSAT solvers that exist nowadays. We also studied how both distributed algorithms fare in comparison with each other.

This article introduced the first two distributed algorithms for MaxSAT. The first algorithm is based on splitting among processes the interval of possible objective values of the optimum solution and shrinking this interval until an optimum solution is found (section 3.1). One process searches on a lower bound, another on an upper bound and the others on local upper bounds. Two versions of this algorithm were implemented and tested. This approach was shown to be effective in practice at speeding up the search process, both in the crafted and industrial instances. Therefore, distributing algorithms can improve the performance of MaxSAT solvers.

The second algorithm partitions the search tree and assigns different sub-trees to distinct processes (section 3.2). This algorithm is based on an approach proposed for SAT solving that was shown to improve the performance of sequential, parallel and distributed SAT solvers. This algorithm was shown to be very effective in the crafted instances, but was not shown to be competitive in the industrial instances.

References

[1] X. An, T. Zhang, H. Fujita, and R. Hasegawa. QMaxSAT: A Partial Max-SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:95–100, 2012.

[2] B. Cha, K. Iwama, Y. Kambayashi, and S. Miyazaki. Local search algorithms for partial MaxSAT. In *AAAI Conference on Artificial Intelligence / IAAI Innovative Applications of Artificial Intelligence Conference*, pages 263–268, 1997.

[3] Z. Fu and S. Malik. On Solving the Partial MAX-SAT Problem. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 252–265, 2006.

[4] M. J. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. In *Hardware and Software: Verification and Testing*, pages 50–65. Springer, 2012.

[5] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *ACM SIGPLAN Notices*, volume 46, pages 437–446. ACM, 2011.

[6] D. Le Berre and A. Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[7] H. Mangassarian, A. Veneris, S. Safarpour, F. N. Najm, and M. S. Abadir. Maximum circuit activity estimation using pseudo-boolean satisfiability. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1538–1543. EDA Consortium, 2007.

[8] J. Marques-Silva, I. Lynce, and S. Malik. Conflict-Driven Clause Learning SAT Solvers. *SAT Handbook*, pages 131–154, 2009.

[9] J. Marques-Silva and J. Planes. On Using Unsatisfiability for Solving Maximum Satisfiability. *CoRR*, abs/0712.1097, 2007.

[10] J. Marques-Silva and K. Sakallah. GRASP: A New Search Algorithm for Satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, 1996.

[11] R. Martins. *Parallel Search for Maximum Satisfiability.* PhD thesis, Instituto Superior Técnico, 2013.

[12] R. Martins, S. Joshi, V. Manquinho, and I. Lynce. Incremental cardinality constraints for maxsat. In *Principles and Practice of Constraint Programming*, pages 531–548. Springer, 2014.

[13] R. Martins, V. Manquinho, and I. Lynce. Exploiting Cardinality Encodings in Parallel Maximum Satisfiability. In *Proc. International Conference on Tools with Artificial Intelligence*, pages 313–320. IEEE Computer Society Press, 2011.

[14] R. Martins, V. Manquinho, and I. Lynce. Parallel Search for Maximum Satisfiability. *AI Communications*, 25:75–95, 2012.

[15] R. Martins, V. Manquinho, and I. Lynce. Open-wbo: a modular maxsat solver. In *Theory and Applications of Satisfiability Testing–SAT 2014*, pages 438–445. Springer, 2014.

[16] A. Morgado, F. Heras, M. Liffiton, J. Planes, and J. Marques-Silva. Iterative and Core-Guided MaxSAT Solving: A Survey and Assessment. *Constraints*, 18:478–534, 2013.

[17] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[18] O. Roussel. Controlling a solver execution with the runsolver tool system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:139–144, 2011.

[19] T. Schubert, M. Lewis, and B. Becker. PaMiraXT: Parallel SAT Solving with Threads and Message Passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:203–222, 2009.

[20] D. M. Strickland, E. Barnes, and J. S. Sokol. Optimal Protein Structure Alignment Using Maximum Cliques. *Operations Research*, 53:389–402, 2005.

[21] H. Xu, R. A. Rutenbar, and K. Sakallah. sub-SAT: a formulation for relaxed Boolean satisfiability with applications in routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22:814–820, 2003.

[22] H. Zhang, M. P. Bonacina, and J. Hsiang. PSATO: a Distributed Propositional Prover and Its Application to Quasigroup Problems. *Journal of Symbolic Computation*, 21:543–560, 1996.

[23] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *International Conference on Computer-Aided Design*, pages 279–285, 2001.