AI Communications 25 (2012) 75–95
DOI 10.3233/AIC-2012-0517
IOS Press

Parallel search for maximum satisfiability

Ruben Martins *, Vasco Manquinho and Inês Lynce
IST/INESC-ID, Technical University of Lisbon, Lisbon, Portugal
E-mails: {ruben, vmm, ines}@sat.inesc-id.pt

Abstract. The predominance of multicore processors has increased the interest in developing parallel Boolean Satisfiability (SAT) solvers. As a result, more parallel SAT solvers are emerging. Even though parallel approaches are known to boost performance, parallel approaches developed for Boolean optimization are scarce. This paper proposes parallel search algorithms for Maximum Satisfiability (MaxSAT) and introduces PWBO, a new parallel solver for partial MaxSAT. Using two threads, an unsatisfiability-based algorithm is used to search on the lower bound of the optimal solution, while at the same time a linear search is performed on the upper bound of the optimal solution. Moreover, learned clauses are shared between threads during the search. For a larger number of threads two different strategies are proposed. The first strategy performs a portfolio approach by searching on the lower and upper bound values of the optimal solution using different encodings of cardinality constraints for each thread. The second strategy splits the search space considering different upper bound values of the optimal solution for each thread. Experimental results show that the techniques proposed in the paper enable PWBO to improve when increasing the number of threads.

Keywords: Parallel search, maximum satisfiability, cardinality constraints

1. Introduction

As a result of multicore processors having become the dominant platform, an increasing number of parallel Boolean Satisfiability (SAT) solvers have come to light in the recent past [9,21,24]. Portfolio approaches are the most common approach for parallel SAT solving, as they explore the sensitivity to parameter tuning of SAT solvers. As a result, each thread has a different combination of parameters that increases the diversification of the search [20,21].

The use of SAT is widespread with many practical applications and it is clear that the optimization version of SAT, i.e. Boolean optimization, can be applied to solve many real-world optimization problems. The competitive performance and robustness of Boolean optimization solvers is certainly required to achieve this goal. When compared with SAT instances, Boolean optimization instances tend to be more intricate as it is not sufficient to find an assignment that satisfies all the constraints, but rather an optimization function has to be taken into account. Hence, it comes as a natural step to develop parallel algorithms to Boolean optimization, following the recent success in the SAT field.

Although this reasoning comes as natural, there are only a few parallel implementations for solving Boolean Optimization. SAT4J PB RES//CP[1] implements a resolution based algorithm that competes with a cutting plane based algorithm to find a new upper bound or to prove optimality. When one of the algorithms finds a new upper bound, it terminates the search of the other algorithm and both algorithms restart their search within the new upper bound value. If one of the algorithms proves optimality, then the problem is solved and the search is stopped. Clause sharing is not allowed between these two algorithms.

Parallel algorithms have the advantage of allowing to implement orthogonal approaches that complement each other. That is the case in SAT4J PB RES//CP where cutting planes are run against resolution. Another alternative, which will be explored in this paper by our new parallel Maximum Satisfiability (MaxSAT) solver, is to run an algorithm that searches to increase the lower bound value against an algorithm that searches to decrease the upper bound value. For a larger number of threads, two different strategies are explored. The first strategy performs a portfolio approach by searching in

*Corresponding author: Ruben Martins, IST/INESC-ID, Technical University of Lisbon, Rua Alves Redol 9, 1000-029 Lisbon, Portugal. E-mail: ruben@sat.inesc-id.pt.

[1] http://www.satcompetition.org/PoS/presentations-pos/leberre.pdf.

the lower and upper bound values of the optimal solution, and using different encodings of cardinality constraints into clauses for each thread. The goal of using different encodings of cardinality constraints is to increase the diversification of the search. The second strategy splits the search space considering different upper bound values of the optimal solution. The parallel search on different upper bound values leads to constant updates on the lower and upper bound values, which result in reducing the search space. Moreover, learned clauses can be shared during search, which further boosts the performance of the parallel MaxSAT solver.

Preliminary versions of this work were published at the RCRA 2011 workshop [35] and at the ICTAI 2011 conference [34] proceedings. The first paper [35] addresses parallel search with two threads and for a larger number of threads the search space is split considering different upper bound values. The second paper [34] describes a portfolio approach using different encodings of cardinality constraints for each thread. This paper extends previous work in the following directions: (1) a dynamic heuristic is presented for selecting encodings of *at-most-k* cardinality constraints, (2) the split approach uses the dynamic heuristic to search in the different upper bound values of the optimal solution and (3) experimental results have been extended: (i) a detailed evaluation of our solver with two threads is now presented, (ii) the portfolio approach is compared with the split approach, and both approaches are run with 4 and 8 threads and (iii) the impact of clause sharing is further evaluated.

The paper is organized as follows. The next section introduces some background notions of MaxSAT and cardinality constraints. Additionally, algorithmic solutions for sequential MaxSAT solving are described. Section 3 briefly describes several encodings of cardinality constraints that will be used in this paper. Moreover, a dynamic heuristic for selecting encodings of *at-most-k* cardinality constraints is presented. Next, Section 4 describes the different versions of our parallel MaxSAT solver: (1) a parallel two-thread search algorithm, (2) a multithread algorithm based on a portfolio approach and (3) a multithread algorithm based on splitting the search space. Section 5 describes the clause sharing mechanism and the use of shared learned clauses during the search. Next, Section 6 presents experimental results that evaluate the dynamic heuristic, as well as the different encodings of cardinality constraints. Additionally, the different versions of our parallel solver are analyzed. Finally, Section 7 concludes the paper and suggests future work.

2. Maximum satisfiability

In this section the MaxSAT problem and its variants are presented and the main algorithmic solutions for solving MaxSAT are briefly described.

In a propositional formula, a literal l_i denotes either a variable x_i or its complement $\bar{x}_i$. If a literal $l_i = x_i$ and x_i is assigned value 1 or $l_i = \bar{x}_i$ and x_i is assigned value 0, then the literal is said to be *true*. Otherwise, the literal is said to be *false*. A propositional clause can be defined as a disjunction of literals and a CNF formula is a conjunction of propositional clauses. A clause is said to be unsatisfied if all of its literals are assigned value 0, and it is said to be satisfied if at least one of its literals is assigned value 1.

Given a CNF formula φ, the MaxSAT problem can be defined as finding an assignment to variables in φ such that it minimizes (maximizes) the number of unsatisfied (satisfied) clauses. MaxSAT has several variants such as partial MaxSAT, weighted MaxSAT and weighted partial MaxSAT. In the partial MaxSAT problem, some clauses in φ are declared as hard, while the rest are declared as soft. The objective in partial MaxSAT is to find an assignment to problem variables such that all hard clauses are satisfied, while minimizing the number of unsatisfied soft clauses. Finally, in the weighted versions of MaxSAT, soft clauses can have weights greater than 1 and the objective is to satisfy all hard clauses while minimizing the total weight of unsatisfied soft clauses.

A generalization of clauses are cardinality constraints. These constraints define that at least k literals must be assigned value 1. More formally, cardinality constrains define that a sum of n literals must be greater than or equal to a given value k, i.e. $\sum_{i=1}^{n} l_i \geqslant k$. However, in practice cardinality constraints are usually expressed as *at-most-k* constraints. Note that the previous *at-least-k* expression can be rewritten as $\sum_{i=1}^{n} \bar{l}_i \leqslant n - k$. In the remainder of the paper, cardinality constraints are defined as being *at-most-k* constraints. Although cardinality constraints do not occur in MaxSAT formulations, several algorithms for MaxSAT solving rely on these constraints. Next, MaxSAT algorithms that use cardinality constraints are briefly surveyed.

2.1. Algorithmic solutions

For solving the several MaxSAT variants, the most common approach is to use a classical branch and bound algorithm [4,14,26,27], where specific MaxSAT lower bound procedures are incorporated to prune the search space. Moreover, MaxSAT inference rules can also be applied during the search [10].

Another approach used in unweighted MaxSAT algorithms is to make a linear search on the number of unsatisfiable soft clauses [1,25]. In this case, for each soft clause ω_i, a new relaxation variable r_i is added to ω_i and all clauses are labeled as hard. Whenever a new solution is found, the value of the solution is determined by the number of relaxation variables assigned value 1. This corresponds to the number of unsatisfied soft clauses in the original formulation. If a solution of value k is found, a new cardinality constraint of the form $\sum r_i \leqslant k-1$ is added to the formula, in order to exclude solutions with a value greater than or equal to the best one found so far. The algorithm ends when the resulting formula becomes unsatisfiable.

In this algorithm, new cardinality constraints are iteratively added to the original formula. Hence, in order to continue using a SAT solver in the subsequent iterations, it is necessary to encode the cardinality constraints into clauses. Another option is to use a pseudo-Boolean (PB) satisfiability solver that is able to deal natively with cardinality constraints.

Notice that both the linear search algorithm and the branch and bound algorithm search on the upper bound of the optimal solution, i.e., at each moment of the search process they maintain a candidate solution that is iteratively improved until optimality is proved. These candidate solutions always have a value that is greater than or equal to the optimum.

Recently, a new generation of MaxSAT solvers have been developed based on the iterated use of a SAT solver to identify unsatisfiable sub-formulas [2,18,28]. Algorithm 1 presents the pseudo-code for the original Fu and Malik's proposal [18]. Consider that φ is the propositional working formula, where constraints are marked as either soft or hard. At each iteration, a SAT solver is used and its output is a pair (st, φ_C) where *st* denotes the resulting status of the solver (satisfiable or unsatisfiable) and φ_C contains the unsatisfiable sub-formula provided by the SAT solver (if φ is unsatisfiable). The sub-formula φ_C is a subset of the original clauses φ that is unsatisfiable. Therefore, φ_C explains the reason for the unsatisfiability of φ. A detailed explanation of how these sub-formulas are obtained can be found elsewhere [5,39]. When the solver returns unsatisfiable, a new relaxation variable is added to each soft constraint in φ_C. Moreover, φ is changed to encode that at most one of the new relaxation variables can be assigned value 1 (line 12) and the algorithm continues to the next iteration. Otherwise, if φ is satisfiable, then the SAT solver was able to find an assignment which is an optimal solution to the original MaxSAT or partial MaxSAT problem [18].

Algorithm 1. Unsatisfiability-based algorithm for MaxSAT and partial MaxSAT

```
FuMalikAlg($\varphi$)
1  while true do
2    $(st, \varphi_C) \leftarrow$ SAT($\varphi$)
3    if $st$ = UNSAT
4      then $V_R \leftarrow \emptyset$
5        for each $\omega \in \varphi_C \wedge$ soft($\omega$) do
6          $r$ is a new relaxation variable
7          $\omega_R \leftarrow \omega \cup \{r\}$
8          $\varphi \leftarrow \varphi \backslash \{\omega\} \cup \{\omega_R\}$
9          $V_R \leftarrow V_R \cup \{r\}$
10       if $V_R = \emptyset$
11         then return UNSAT
12         else $\varphi \leftarrow \varphi \cup$ CNF($\sum_{r \in V_R} r \leqslant 1$)
13     else return Satisfiable assignment to $\varphi$
```

Different algorithms have been proposed for MaxSAT and partial MaxSAT based on this approach. Additionally, different encodings for cardinality constraints have been proposed with better results [31,32]. Moreover, different strategies have been used regarding the total number of relaxation variables needed [2, 31]. The unsatisfiability-based approach has also been extended for weighted variants of MaxSAT [2,28].

Finally, notice that Algorithm 1 performs a lower bound search, i.e. at each step of the algorithm, the number of identified unsatisfiable sub-formulas provides a lower bound on the value of the optimal solution. This algorithm is complementary to the previously described upper bound approaches.

3. Encodings for cardinality constraints

In the previous section we have seen that cardinality constraints can arise when solving a MaxSAT formula. If the solver is not able to natively handle cardinality constraints, then it is necessary to translate cardinality constraints into clauses. Several encodings that translate cardinality constraints into clauses have been proposed. In this section, a brief description of several encodings that will be used in the remainder of the paper is provided. Moreover, a dynamic heuristic for selecting the encoding for cardinality constraints is also presented. Given a portfolio of encodings and a cardinality constraint, the dynamic heuristic tries to select the most adequate encoding for that constraint.

Our focus is on encodings for cardinality constraints (in the sequel referred as *cardinality encodings*) that allow unit propagation to maintain arc consistency in the resulting CNF encoding. Consider the following cardinality constraint $x_1 + \cdots + x_n \leqslant k$. If k variables are assigned value *true*, then unit propagation will enforce the value *false* on the remaining $n - k$ variables. However, if $k + 1$ variables are assigned value *true*, then a conflict arises since at most k variables can be assigned value *true*. An encoding that maintains arc consistency enables the SAT solver to infer the same information with the use of unit propagation on the resulting CNF encoding.

A special case of cardinality constraints are the *at-most-one* constraints. These constraints express that at most one out of n Boolean variables can be assigned value *true*. A large number of encodings have been proposed to handle *at-most-one* constraints. Therefore, a distinction is made between encodings that are used only for *at-most-one* constraints and encodings used for the general case of *at-most-k* constraints.

Table 1 shows the size of the evaluated encodings. Next, the encodings are briefly described. For further details on each encoding, the reader is pointed to the literature.

- *Pairwise* (also called naive): the most widely known encoding for the *at-most-one* constraint. For each pair of variables (x_i, x_j), add a binary clause $\bar{x}_i \vee \bar{x}_j$ that guarantees that only one of the two variables can be assigned value *true*. Even though this encoding adds a quadratic number of clauses, it does not require auxiliary variables.
- *Ladder* [3,19]: it uses $n - 1$ auxiliary variables to form a structure named ladder. Consider the chain of auxiliary variables $y_1, \ldots, y_{n+1}$. If y_i is *false* then all variables y_j with $j > i$ are also *false*. Each valid state in the ladder is associated with a variable of the cardinality constraint. Since each x_i is equivalent to a valid state in the ladder, this encoding guarantees that at most one variable x_i will be assigned value *true*.
- *Bitwise* [17,36]: this encoding introduces auxiliary variables $y_1, \ldots, y_{\log_2 n}$ that represent a bit string. It then associates a unique bit string with each variable x_i. The encoding guarantees that only one string may occur and therefore at most one variable x_i can be assigned value *true*. When n is not a power of 2, we can perform a small optimization by reducing the number of clauses from the encoding [17]. Note that if n is not a power of 2, then there are more strings than variables x_i. Hence, we can associate two strings to some of the variables x_i until the number of remaining strings is equal to the number of remaining variables x_i.
- *Commander* [23]: it starts by partitioning the set of variables x_i into groups of size 3. Next, for the variables of each group, an *at-most-one* constraint is encoded with the pairwise encoding. Finally, it associates a commander variable with each group and recursively encodes the *at-most-one* constraint over the commander variables with the method just described.
- *Product* [12]: this encoding decomposes cardinality constraint $x_1 + \cdots + x_n \leqslant 1$ into two constraints, $y_1 + \cdots + y_{p_1} \leqslant 1$ and $z_1 + \cdots + z_{p_2} \leqslant 1$, where $p_1 \times p_2 \geqslant n$. The idea is to associate each variable x_i with a coordinate (y_a, z_b). This procedure is applied recursively until the size of the constraint is smaller than 7. At that point, the pairwise encoding is used.
- *Sequential* [37]: it encodes a circuit that sequentially counts the number of variables x_i that are assigned value *true*. Each x_i is associated with k variables $s_{i,j}$ that are used as a counter. Assigning the value *false* to $s_{i,j}$ implies that at most j of the variables $x_1, \ldots, x_{i-1}$ can be assigned value *true*.
- *Totalizer* [8]: it consists of a totalizer and a comparator. The totalizer can be seen as a binary tree, where the leaves are the x_i variables. Each intermediate node is labeled with a number s and uses s auxiliary variables to represent the sum of the leaves of the corresponding subtree. The original encoding uses $\mathcal{O}(n^2)$ clauses. However, since we are using this encoding to encode *at-most-one* constraints, the optimization proposed by Büttner and Rintanen [11] that reduces the number of

Table 1

Cardinality constraint encodings

Encoding	#Clauses	#Variables	Type
Pairwise	$\mathcal{O}(n^2)$	0	*at-most-one*
Ladder	$\mathcal{O}(n)$	$\mathcal{O}(n)$	*at-most-one*
Bitwise	$\mathcal{O}(n \log_2 n)$	$\mathcal{O}(\log_2 n)$	*at-most-one*
Commander	$\mathcal{O}(n)$	$\mathcal{O}(n)$	*at-most-one*
Product	$\mathcal{O}(n)$	$\mathcal{O}(n)$	*at-most-one*
Sequential	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$	*at-most-k*
Totalizer	$\mathcal{O}(nk)$	$\mathcal{O}(n \log_2 n)$	*at-most-k*
Sorters	$\mathcal{O}(n \log_2^2 n)$	$\mathcal{O}(n \log_2^2 n)$	*at-most-k*

clauses to $\mathcal{O}(nk)$ is used. Instead of counting up to n, it is enough to count up to $k+1$, which can be used to reduce the number of variables used for each node.

- *Sorters* [15]: it is based on a sorting network, i.e. a circuit that receives n Boolean inputs $x_1, \dots, x_n$ and permutes them to obtain the sorted outputs $y_1, \dots, y_n$. Consider the cardinality constraint $x_1 + \cdots + x_n \leqslant k$. If after building the sorting network we assign *false* to the output y_{k+1}, then this guarantees that at most k variables x_i can have value *true*. Some improvements were introduced over the original sorting network encoding, namely, the use of half sorting networks [6] and adding redundant clauses over the outputs that amplify propagation [13]. Even though the size of the sorters encodings grows with n, unit propagation on the outputs $y_{k+1}, \dots, y_n$ will significantly reduce the size of the encoding. Therefore, if k is small, then the sorter encoding will be much smaller after unit propagation. Moreover, for the *at-most-one* constraints, the simplification of the sorting network through partial evaluation [13] is used and the size of the encoding is reduced to $\mathcal{O}(n)$ clauses and variables.

3.1. Dynamic heuristic

Different cardinality encodings lead to different performances. For the *at-most-k* cardinality constraint it has been empirically observed that several features may be used to build a dynamic heuristic for selecting the more adequate encoding for each cardinality constraint [34]. Consider the following portfolio of encodings: *Totalizer*, *Sorters* and the native representation of cardinality constraints, i.e. without encoding them into CNF. For a given MaxSAT formula with v variables and an *at-most-k* cardinality constraint with size n, the dynamic encoding heuristic behaves as follows:

(1) If (i) $0.25 \leqslant k/n \leqslant 0.75$, (ii) $n > 1024$ and (iii) $v/n < 0.75$:
   - then do not encode the cardinality constraint into clauses. In this case the native representation is used and we rely on the underlined pseudo-Boolean solver to handle it.

(2) Else if $n \times k < n \times \log_2^2 n$:
   - then encode the cardinality constraint into clauses using the *totalizer* encoding.

(3) Otherwise:
   - the cardinality constraint is translated into clauses using the *sorters* encoding.

The native representation is used when the ratio between the number of variables and the value of k is close to $n/2$. This is the worst case when using a CNF representation. However, this is only used when the number of variables in the cardinality constraint is larger than 1024. When n is small, encoding into clauses is still more effective than using a native representation, even when k is close to $n/2$. Note that when $k > n/2$ the *at-most-k* constraint can be rewritten as an *at-least-*$(n-k)$ constraint. Let $x_1 + \cdots + x_n \leqslant k$ be an *at-most-k* constraint. This constraint can be rewritten as $\bar{x}_1 + \cdots + \bar{x}_n \geqslant n-k$. The totalizer and sorters encodings allow the encoding of *at-least-*$(n-k)$ constraints. Therefore, if $k > n/2$, the *at-most-k* constraint is rewritten as an *at-least-*$(n-k)$ constraint and then encoded into clauses. Hence, the worst case when using a CNF representation is when k is close to $n/2$ since rewriting the constraint does not reduce the size of the encoding.

It was also observed that when the cardinality constraint contains the majority of the variables of the problem, encoding the cardinality constraint into clauses may lead to better results. In this case, the native representation has a low propagation since it consists of a constraint that has almost all the variables of the problem. The choice between the totalizer and sorters encodings is based on the size of the encoding. The encoding with smaller size is always chosen.

Note that the sequential and totalizer encodings have similar size complexities. However, it has been observed that for solving MaxSAT the totalizer encoding is, in general, more efficient than the sequential encoding [34]. Therefore, the sequential encoding is not considered in our portfolio of cardinality encodings.

4. Parallel search

Nowadays, extra computing power is not coming anymore from higher processor frequencies but rather from a growing number of cores and processors. Exploiting this new architecture is expected to allow MaxSAT solvers to become more effective, being able to solve more problem instances. Next, we describe how to take advantage of this new architecture by performing parallel search on the upper and lower bound values of the optimal solution.

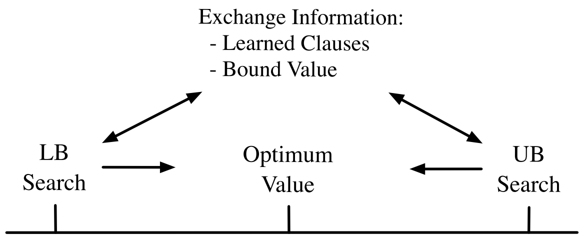

Fig. 1. Solver architecture for two threads.

4.1. Searching on the lower and upper bound values

Our parallel search is based on two orthogonal algorithms: (1) unsatisfiability-based algorithms that search on the lower bound of the optimal solution, i.e. that perform *lower bound search* and (2) linear search algorithms that search on the upper bound of the optimal solution, i.e. that perform *upper bound search*. Therefore, we propose to perform a parallel search on both the upper bound and the lower bound of the optimal solution.

For computer architectures with two cores, this approach consists in using one thread to perform lower bound search and another thread to perform upper bound search. Figure 1 shows the architecture for two threads. A parallel search with these two orthogonal strategies results in a performance as good as the best strategy for each problem instance. However, if both threads cooperate through clause sharing, it is possible to perform better than the best strategy. Additionally, both strategies can also cooperate in finding the optimum value. If during the search the lower bound value provided by the unsatisfiability-based algorithm and the upper bound value provided by the other thread become the same, it means that an optimal solution has been found. As a result, it is not necessary for any of the threads to continue the search to prove optimality since their combined information already proves it.

4.2. Using a portfolio approach

The previous section presented a parallel search solver for MaxSAT based on two orthogonal strategies. In the proposed approach, one thread is used for each strategy. For computer architectures with more than two cores, we can extend the previous idea by using several threads in the upper and lower bound search. However, if the algorithms that perform lower bound search are the same and the algorithms that perform upper bound search are also the same, then the gain from increasing the number of threads will be very small since all threads will be searching in a similar way. Therefore, it is important to increase the diversification of the search such that the search space is explored differently by each algorithm performing lower bound search and by each algorithm performing upper bound search. One solution is to exploit the variety of cardinality encodings by using a portfolio of algorithms using different encodings. Next, we present the parallel algorithms that are used for the lower and upper bound search.

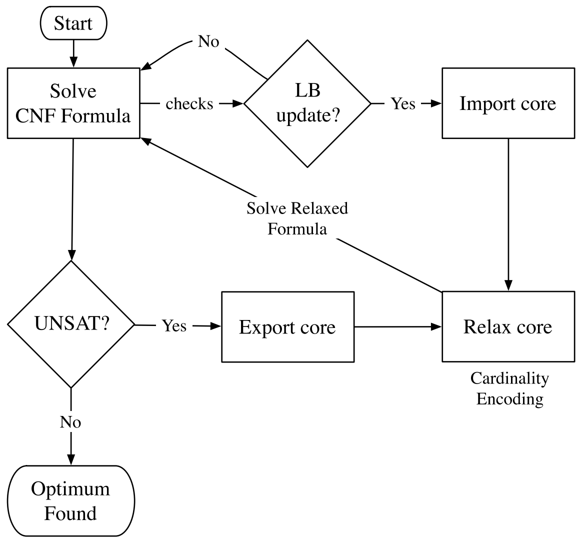

Fig. 2. Parallel unsatisfiability-based algorithms.

Figure 2 illustrates parallel unsatisfiability-based algorithms. These algorithms work by iteratively identifying unsatisfiable sub-formulas of the original formula. While solving the formula, the algorithm checks if another thread has found a better lower bound value, i.e. if it has found an unsatisfiable sub-formula. If this is the case, then it imports the unsatisfiable sub-formula and relaxes the unsatisfiable core by adding relaxations variables to the soft clauses as described in Algorithm 1 in Section 2. For each soft clause in the identified unsatisfiable sub-formula, a new relaxation variable is added such that when this variable is assigned value 1, the soft clause becomes satisfiable [18]. Moreover, a *cardinality constraint* is also added to the relaxed formula such that only one of the newly created relaxation variables can be assigned value 1. Next, the solver checks if the formula remains unsatisfiable.

If the algorithm is not informed that a better lower bound value was found, then it continues the search process until it finds an unsatisfiable sub-formula or a solution to the formula. If it finds an unsatisfiable sub-formula, it shares this formula with the remaining lower bound threads by exporting the unsatisfiable core to the other threads. Next, it relaxes the unsatisfied sub-formula as previously described and continues the search on the new formula. The procedure ends when the working formula becomes satisfiable and the solver returns a solution (i.e., the optimum value was found), or when the unsatisfiable sub-formula only contains hard clauses (i.e., the original problem instance is unsatisfiable) [18].

To increase the diversification of the search, unsatisfiability-based algorithms use different cardinality encodings in the relaxation step. These cardinality constraints are *at-most-one* constraints and any of the *at-most-one* encodings presented in the previous section can be used.

Note that there are a few details not shown in Fig. 2. In particular, only one thread exports an unsatisfied core for each lower bound value. Before exporting an unsatisfiable core, a thread checks if its lower bound value is the highest lower bound value among all threads. If this is the case, then it is safe to export the unsatisfiable core to the remaining threads. Otherwise, it discards its own unsatisfiable core and imports the unsatisfiable core that corresponds to the current lower bound value. This is done to guarantee that all threads use the same unsatisfiable cores. Moreover, when a thread relaxes an unsatisfiable core, it updates its lower bound value.

Figure 3 illustrates parallel linear search algorithms. Notice that the original MaxSAT formula φ_{MS} is modified by adding a new relaxation variable r to each soft clause ω from φ_{MS}, resulting in an equivalent formula

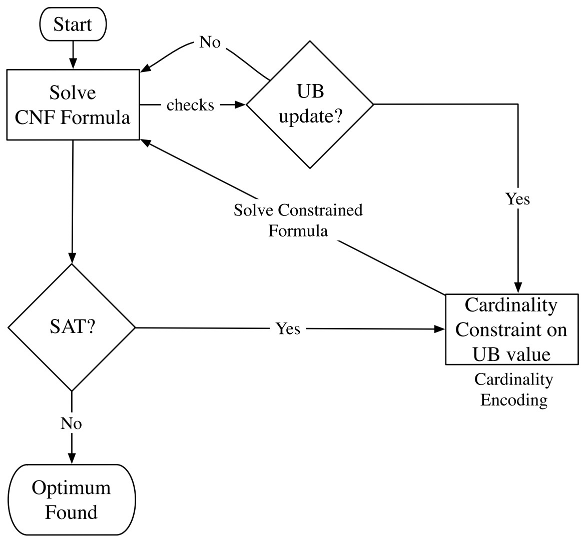

Fig. 3. Parallel linear search algorithms.

φ_{UB} where one wants to minimize the number of relaxation variables assigned value 1. In this approach, whenever a new solution is found for φ_{UB}, the upper bound value is updated and a new *cardinality constraint* on the relaxation variables is added such that all solutions with a greater or equal value are excluded. During search, each algorithm checks if there is a better upper bound value. If this is the case, it adds a cardinality constraint considering the new upper bound value. Afterwards, it restarts the search on the constrained formula.

To increase the diversification of the search, the linear search algorithms to be used should differ between themselves on the cardinality encoding that is used when a new cardinality constraint is added to the working formula. Consider that the current upper bound value is k. As a result of finding a new solution of value k', the cardinality constraint $x_1+\cdots+x_n \leqslant k$ becomes increasingly stronger by decreasing k to k'. For the *at-most-k* encodings presented in the previous section, it is only needed to encode the cardinality constraint into clauses when the first upper bound value is found. In the next iterations, one can set some specific literals in the encoding to *false* such that it restricts the cardinality constraint to the new upper bound value. This is denoted by *incremental strengthening* [6]. All learned clauses from previous iterations remain valid and can therefore be kept.

4.3. Splitting the search space

Another approach that can be done for computer architectures with more than two cores is to split the search space by searching on different local upper bound values. In this parallel search, if n cores are available, then one thread is used to search on the lower bound, another thread is used to search on the upper bound, and the remaining $n-2$ threads will search on different local upper bound values. The local upper bound values restrict the search space by enforcing a fixed upper bound value of the optimal solution. Since this fixed upper bound value is restricted to each thread, it is named as *local upper bound value*. The search performed by these threads is named as *local upper bound search*. The iterative search on different local upper bound values leads to constant updates on the lower and upper bound values that will reduce the search space. Next, an example of this approach is described. Afterwards, a more detailed description of the local upper bound search is presented.

Example 1. Consider a partial MaxSAT formula φ_{MS} as input. For the input formula, one can easily find initial lower and upper bounds. Suppose the initial lower and upper bound values are 0 and 11, respectively. Moreover, consider also that the optimal solution is 3 and our goal is to find it using four threads: t_0, t_1, t_2 and t_3. Thread t_0 applies an unsatisfiability-based algorithm (i.e., searches on the lower bound of the optimum solution). This thread starts with a lower bound of 0 and will iteratively increase the lower bound until the optimum value is found.

Thread t_1 searches on the upper bound of the optimum solution. Hence, thread t_1 starts its search with upper bound value of 11. Threads t_2 and t_3 search on different local upper bound values. For example, threads t_2 and t_3 can start their search with local upper bound values of 3 and 7, respectively.

Suppose that thread t_2 finishes its computation and finds that the formula is unsatisfiable for an upper bound of 3. This means that there is no solution with values 0, 1 and 2. Therefore, the lower bound value can be updated to 3. Thread t_2 is now free to search on a greater local upper bound value, for example, 5. In the meantime, thread t_3 finds a solution with value 6. Hence, the upper bound value can be updated to 6. Thread t_1 updates its upper bound value to 6 and thread t_3 is now free to search on a different local upper bound value, for example, 4. Afterwards, consider that thread t_1 finds a solution with value 3. Again, the upper bound value can be updated to 3. Since the lower bound value is the same as the upper bound value, the optimum has been found and the search terminates.

This parallel search incorporates three types of algorithms: unsatisfiability-based, linear search and local linear search. The unsatisfiability-based and linear search algorithms used by the threads that are searching in the lower and upper bound values of the optimal solution are the same as the ones presented in Figs 2 and 3 in the previous section. In what follows we will describe the algorithm for parallel local linear search algorithms that is used by the remaining threads to perform local upper bound search.

Figure 4 illustrates parallel local linear search algorithms. Similarly to the linear search algorithms, the original MaxSAT formula φ_{MS} is modified by adding a new relaxation variable r_i to each soft clause ω_i from φ_{MS}, resulting in an equivalent formulation φ_{UB} where the goal is to minimize the number of relaxation variables assigned value 1.

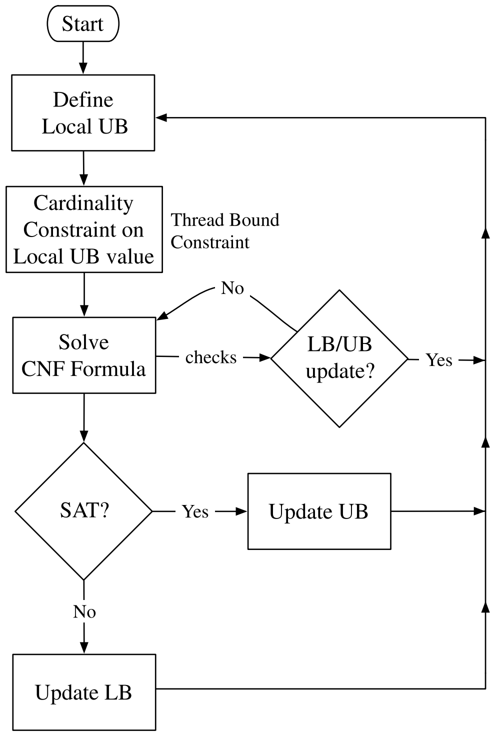

Fig. 4. Parallel local linear search algorithms.

These algorithms start by defining their local upper bound. Initially, we set the lower bound value to 0 and the upper bound value to the number of soft clauses, s, in φ_{MS} plus 1. Therefore, considering k local threads, $t_1, \ldots, t_k$, a thread t_j will have an initial local upper bound value b_j of $j \times \lfloor (s+1)/(k+1) \rfloor$.

Next, thread t_j adds a cardinality constraint of the form $\sum r_i \leqslant b_j - 1$ to exclude solutions with a value greater than or equal to b_j. Let this cardinality constraint be labeled the *thread bound constraint*. If a cardinality encoding is used, then all clauses that were created to encode the cardinality constraint will be labeled as thread bound constraints.

After adding a thread bound constraint, the algorithm starts the search. During the search, the algorithm checks if another thread has found a lower bound that is greater than the thread current local upper bound or an upper bound that is smaller than the thread current local upper bound. If one of these cases occurs, then the algorithm will terminate its search and a new local upper bound is defined. Next, the search restarts using the new upper local value.

If the algorithm is not informed that a better lower or upper bound value has been found, then it continues the search process until it finds a solution or proves that no solution exists for the current local upper bound value. If a solution is found, then the algorithm updates the upper bound value. Otherwise, if it proves that no solution exits, then the lower bound value is updated. In both cases, a new local upper bound value is set and the search restarts.

At the moment, parallel local linear search algorithms are not computing an unsatisfiable sub-formula when a new lower bound value is found. Therefore, no unsatisfiable cores are exported to the unsatisfiability-based algorithm that is searching on the lower bound value. Moreover, the thread searching on the lower bound does not update its lower bound value to the new lower bound value found by the parallel local linear search algorithms. As future work, we plan to compute the unsatisfiable sub-formulas from the parallel local linear algorithms and use them to speedup the search of the unsatisfiability-based algorithm.

There are a few details not shown in Fig. 4. Updates to the lower and upper bounds only take place when the new values improve the current ones. Additionally, when a thread is assigned a new local upper bound value after finding a solution or proving that a solution does not exist, this new local upper bound value covers the broadest range of yet untested bounds. More formally, the new local upper bounds are chosen as follows. Let $B = \langle b_0, b_1, \ldots, b_{k-1}, b_k\rangle$ be a sorted list where b_0 corresponds to the lower bound and b_k corresponds to the upper bound, while the remaining b_j are the non-aborted thread local upper bounds. Let $[b_{m-1}, b_m]$, where $1 \leqslant m \leqslant k$, define an interval such that for all $1 \leqslant j \leqslant k$ we have $b_m - b_{m-1} \geqslant b_j - b_{j-1}$. In this case, the new upper bound of the aborted thread is $\lfloor (b_m + b_{m-1})/2 \rfloor$. The sorted list B is updated with the new value and this process is repeated for each aborted thread.

Example 2. Consider the following scenario. A partial MaxSAT formula φ_{MS} is currently being solved by 4 threads. Thread t_0 is searching on the lower bound value of the optimal solution, and thread t_1 is searching on the upper bound value of the optimal solution. The current lower and upper bound values are 5 and 10, respectively. Thread t_2 is searching on a local upper bound with value 8 and thread t_3 is computing a new local upper bound. The sorted list B corresponds to $B = \langle 5, 8, 10\rangle$. Thread t_3 will now determine the largest interval between two consecutive values in B, i.e. $[5, 8]$. Therefore, the new upper local upper bound value of t_3 will be given by $\lfloor (8 + 5)/2 \rfloor = 6$.

5. Sharing learned clauses

Conflict-driven clause learning [33,38] is crucial for the efficiency of modern SAT solvers. After detecting a conflict, i.e. a sequence of assignments that make a clause unsatisfiable, a new clause is learned to prevent the same conflict from occurring again in the subsequent search. The new clause results from the analysis of the implication graph which represents the dependencies between assignments. A more detailed explanation can be found in the literature [33,38].

Clause learning is also essential to the efficiency of many modern MaxSAT solvers. In the context of parallel solving, it is expected that sharing learned clauses can help to further prune the search space and boost the performance of a parallel solver. Similarly to parallel SAT solving, only learned clauses that have less than a given number of literals are shared among all threads. In our parallel solver, we start by sharing learned clauses that have 8 or fewer literals. This cutoff is dynamically changed using the throughput and quality heuristic proposed by Hamadi et al. [22]. Additionally, all clauses with literal block distance 2 are also shared [7].

However, in our parallel solver not all learned clauses can be shared among all threads. This is due to the fact that the working formulas are different. As previously explained, unsatisfiability-based algorithms work directly with the input formula φ_{MS}, while algorithms that perform a linear search on the upper bound value use relaxation variables on the soft clauses, resulting in formula φ_{UB}. In order to define the conditions for safe clause sharing, we start by defining soft and hard learned clauses.

Definition 1 (Soft and hard learned clauses). If in the conflict analysis procedure used in the unsatisfiability-based algorithm, there is at least one soft clause used in the implication graph, then the generated learned clause is labeled as soft. On the other hand, if only hard clauses are used, then the generated learned clause is labeled as hard.

Since φ_{MS} contains both soft and hard clauses, it will have soft and hard learned clauses. On the other hand, φ_{UB} only has hard clauses, and as a result will only have hard learned clauses. Nevertheless, as mentioned previously, φ_{UB} contains additional relaxation variables that are not present in φ_{MS}. When using cardinality encodings, we also have to take into account the auxiliary variables used by those encodings. Therefore, each thread may contain variables not present in the other threads. Moreover, threads that perform local upper bound search contain thread bound constraints. These constraints cannot be shared among all threads, since they are only valid if the optimum value is smaller than the upper bound value of the thread. The same sharing rules must apply to conflict-driven learned clauses that depend on the thread bound constraint. Therefore, it is necessary to define what is a local constraint and in which conditions it can be shared with other threads.

Definition 2 (Local constraint). The thread bound constraint is labeled as a local constraint. Let ω be a conflict-driven learned clause and let φ_ω be the set of constraints used in the implication graph to learn ω. The new clause ω is defined as a local constraint if at least one constraint in φ_ω is a local constraint.

The safe sharing procedure between the different algorithms is as follows:

- Hard learned clauses from unsatisfiability-based algorithms that do not have auxiliary variables can be safely shared with the other threads.
- Soft learned clauses from unsatisfiability-based algorithms are not shared with the other threads. These clauses may not be valid for formulas φ_{UB} and cannot be shared with the algorithms that perform linear search on the upper bound.
  Notice that these clauses could eventually be shared with other threads that are using unsatisfiability-based algorithms. However, it would be necessary to establish an equivalence between the relaxation variables of the learned soft clause and the relaxation variables of the importing thread. Since variables are created for producing the encoding of cardinality constraints, the identification of the relaxation variables may differ between threads. Even though it would be possible to share soft learned clauses between unsatisfiability-based algorithms, it is currently not implemented in our parallel solver.
- Hard learned clauses generated when solving φ_{UB} can be shared with the other threads if the learned clause is not a local constraint and if it does not contain relaxation or auxiliary variables.
- Hard learned clauses that are local constraints generated when solving φ_{UB} cannot be safely shared with the lower bound threads. However, local constraints that do not contain auxiliary variables can be shared between upper bound threads. Sharing local constraints depends on the upper bound value of the thread. If an importing thread has an upper bound smaller than or equal to the upper bound of the exporting thread, then the import is safe. Otherwise, the import may be unsafe and the sharing is not done.

Finally, between iterations of the unsatisfiability-based algorithms, the working formulas φ_{MS} are also extended with additional relaxation variables. However, since these variables are added to soft clauses, if a conflict-based learned clause contains any relaxation variable, then it will necessarily be considered a soft clause. This is due to the fact that at least one soft clause would have been used in the learning procedure.

5.1. Integration of learned clauses

Whenever a learned clause is generated, if its size is smaller than the current cutoff and if it meets the safe sharing conditions, it is exported as a learned clause to the other threads. Later on, when a thread checks if there is a better lower or upper bound value, it also imports the learned clauses that were shared by other threads. Since importing clauses is done during the search, the learned clauses have to be integrated in the context of the current search space. Hence, for the integration of a shared clause ω we have to take into consideration the following cases:

- ω is a unit clause. A restart is forced and the corresponding literal is assigned.
- ω is unit in the current context. The SAT algorithm backtracks to the largest decision level of the assigned variables in ω. A decision level of a variable denotes the depth of the decision tree at which the variable is assigned a value. After backtracking, the unassigned literal is assigned and propagated.
- ω is unsatisfied in the current context. The SAT algorithm backtracks to the largest decision level of the variables in ω. Conflict analysis is performed to allow further backtracking. Moreover, during the conflict analysis procedure a new clause is learned.
- ω is satisfied in the current context. If exactly one literal in ω is satisfied and the remaining literals are falsified, and if the decision level of the satisfied literal is higher than the decision levels of all falsified literals, then the algorithm backtracks to the largest decision level among the falsified literals.

In the remaining cases the learned clause is simply added to the importing thread and no backtracking is needed. The integration procedure must be done in order to ensure the correctness of the solver. A similar procedure is done in the parallel SAT solver MANYSAT [21].

6. Experimental results

This section evaluates the different cardinality encodings in MaxSAT solving and their application to sequential and parallel MaxSAT algorithms. Moreover, several versions of PWBO based on the proposed algorithms are presented and evaluated. All experiments were run on the partial MaxSAT instances from the industrial category of the MaxSAT Evaluation 2011,[2] which correspond to a set of 497 instances. Note that this set of instances corresponds to the same 497 instances that enter the MaxSAT Evaluation 2010.[3] The evaluation was performed on a computer with two AMD Opteron 6172 processors (2.1 GHz with 64 GB of RAM) running Fedora Core 13 with a timeout of 1800 s (wall clock time).

For the parallel solvers, results were obtained by running each solver three times on each instance. Similarly to what is done when analyzing randomized solvers, the median time was taken into account. This means that an instance must be solved by at least two of the three runs to be considered solved.

6.1. Evaluation of the encodings of cardinality constraints

The different cardinality encodings were implemented in WBO [28,29]. In WBO [29] the search is initially done by a pseudo-Boolean solver that performs a search on the upper bound of the optimal solution. However, the use of the pseudo-Boolean solver is limited to 10% of the time limit given to solve the instance. If the pseudo-Boolean solver proves optimality within this time limit, the optimal solution has been found without having to search on the lower bound side. On the other hand, if the pseudo-Boolean solver was not able to prove optimality within the given time limit, an unsatisfiability-based algorithm is used to search on the lower bound side.

In the previous sections, we have seen that the cardinality constraint *at-most-one* is used in the lower bound search, whereas the cardinality constraint *at-most-k* is used in the upper bound search. To evaluate these two types of cardinality constraints, we have run WBO using only the lower bound search or the upper bound search.

6.1.1. Lower bound search

Table 2 shows the number of instances solved by the lower bound algorithm when using different cardinality encodings. Additionally, we also consider the native representation of cardinality constraints given by the pseudo-Boolean representation. The first column of Table 2 shows the set of benchmarks. The second column shows the number of instances per benchmark set. Columns 3–11 show the number of instances solved when using the different *at-most-one* encodings.

For the *at-most-one* cardinality encodings, the ladder encoding performed best overall. However, there is no clear winner for all the benchmarks. This shows that cardinality encodings can diversify the search, since each encoding enables solving different instances. When the number of variables in the *at-most-one* constraint is small (less than a few hundred), then it is better to use the pairwise encoding or a pseudo-Boolean representation. This occurs, for example, in the `pbo-mqc` benchmark set. For these benchmark instances, the optimum value is usually low (around 10) and the average size of each unsatisfiable sub-formula is small (a few hundred clauses). Recall that every time an unsatisfiable sub-formula is found, a new *at-most-one* constraint is added to the formula. In partial MaxSAT, the number of unsatisfiable sub-formulas (cores) will be the same as the optimum value. Hence, if the optimum value is small, then the number of iterations will also be small. Moreover, the number of variables in the *at-most-one* constraint is the same as the size of the unsatisfiable core. As a result, for small unsatisfiable cores, the number of variables in the *at-most-one* constraints will also be small. With the exception of the ladder encoding, encodings that use auxiliary variables do not perform well on the `pbo-mqc` benchmarks. It has been observed that changing the branching heuristic not to branch on auxiliary variables may lead to better results [30]. Hence, as future work we propose to study the impact of branching on auxiliary variables. On the other hand, if the number of variables in the *at-most-one* constraint is large (several thousands), then it is better to encode the constraint into CNF. This can be observed in the `bcp-fir` benchmark instances where the unsatisfiable sub-formulas found by our algorithm are usually larger. Therefore, it is necessary to encode larger cardinality constraints (with thousands of variables).

[2]http://www.maxsat.udl.cat/11/.

[3]http://www.maxsat.udl.cat/10/.

Table 2
Number of instances solved by lower bound search with different cardinality encodings

Benchmark	#I	*at-most-one* – Lower bound search								
		Pairwise	Ladder	Bitwise	Commander	Product	Sequential	Totalizer	Sorters	PB
bcp-fir	59	44	50	46	52	47	49	50	49	44
bcp-hipp-yRa1	55	21	22	21	21	22	21	23	20	20
bcp-msp	64	3	3	3	4	4	4	5	5	4
bcp-mtg	40	17	19	16	18	17	17	18	17	17
bcp-syn	74	34	35	35	35	35	34	34	34	34
CircuitTrace	4	0	1	1	1	1	1	1	1	0
Haplotype	6	5	5	5	5	5	5	5	5	5
pbo-mqc	168	46	44	35	37	36	38	39	36	47
pbo-routing	15	15	15	15	15	15	15	15	15	15
PROTEIN_INS	12	1	1	1	1	1	1	1	1	1
Total	497	186	195	178	189	183	185	191	183	187

Table 3
Number of instances solved by upper bound search with different cardinality encodings

Benchmark	#I	*at-most-k* – Upper bound search				
		Sequential	Totalizer	Sorters	PB	Dyn.
bcp-fir	59	51	53	51	10	53
bcp-hipp-yRa1	55	38	40	42	18	42
bcp-msp	64	26	26	26	12	26
bcp-mtg	40	40	40	40	26	40
bcp-syn	74	32	32	32	21	32
CircuitTrace	4	4	4	4	4	4
Haplotype	6	0	5	5	0	5
pbo-mqc	168	152	151	155	168	168
pbo-routing	15	15	15	15	13	15
PROTEIN_INS	12	2	2	2	1	2
Total	497	360	368	372	273	387

6.1.2. *Upper bound search*

Table 3 shows the number of instances solved by upper bound search with the different cardinality encodings and the dynamic encoding heuristic. Additionally to the cardinality encodings we also consider the pseudo-Boolean representation of cardinality constraints, i.e. without encoding them into CNF. For the *at-most-k* cardinality encodings, the sorter network performed better overall. Even though the diversity of cardinality encodings for *at-most-k* is smaller than for *at-most-one*, it suffices to show that different encodings may solve different instances.

Remember that the cardinality constraint is encoded only once, when the first solution is found. Hence, the size of the cardinality constraint depends on the number of variables and on the upper bound value. Encoding cardinality constraints into CNF is therefore more effective when the number of variables is high (thousands) and the upper bound value is small when compared with the number of variables. This occurs, for example, in the `bcp-fir` benchmark. On the other hand, when given a cardinality constraint of size n with upper bound value close to $n/2$, using a pseudo-Boolean representation can be more effective than encoding the cardinality constraint into CNF. This is the case of the `pbo-mqc` benchmark set.

When considering the dynamic encoding heuristic, we can see that this heuristic outperforms all other cardinality encodings. This shows the importance of selecting the most adequate encoding for each problem instance.

Table 4 shows the number of times that each encoding was selected by the dynamic encoding heuristic when it was able to solve an instance. As expected, the

Table 4
Number of times each encoding was used in the dynamic heuristic

#*I*	Dynamic heuristic		
	Totalizer	Sorter	PB
387	100	247	40

pseudo-Boolean representation was the least used. Indeed, it was only used in the `pbo-mqc` benchmark set. The totalizer encoding was the second most used, and its application was mostly in the `bcp-fir` and `bcp-syn` benchmark sets. Overall, the sorters encoding was the most used by our dynamic encoding heuristic, since it usually has a smaller size than the totalizer encoding.

Figure 5 shows a cactus plot with running times of solvers that search on the upper bound value of the optimal solution. The cactus plot shows the sorted run times for each solver. Each point in the plot corresponds to a problem instance, where the y-axis corresponds to the wall clock time required by the solver and the x-axis corresponds to the accumulative number of instances solved until that time. The solvers considered were the solvers with the different cardinality encodings presented in Table 3 and QMAXSAT 0.4 [1]. QMAXSAT 0.4 was the winner of the partial MaxSAT industrial category in the MaxSAT evaluation 2011. The pseudo-Boolean representation is much less effective than encoding the cardinality constraint into CNF. However, even between the different encodings we can see different running times. The sequential encoding is much less efficient than the totalizer and sorter encodings. Even though the sorter encoding is faster than the totalizer encoding, the performance of both encodings are comparable. Nevertheless, the dynamic encoding heuristic clearly outperforms all other encodings. Moreover, the dynamic encoding heuristic is able to outperform QMAXSAT 0.4, since it solves 387 instances whereas QMAXSAT 0.4 can only solve 378 instances. Therefore, the dynamic encoding heuristic is able to improve the current state of the art. In the remainder of the paper we will denote the solver based on the dynamic encoding heuristic as PWBO-T1, since it is our best performing solver with one thread.

6.2. Evaluation of the parallel solvers

This section evaluates the different versions of PWBO, a parallel solver implemented on top of WBO [28,29]. Using two threads, PWBO-T2 searches on the lower and upper bound values of the optimal solution. For more than two threads, PWBO-P(ortfolio) and PWBO-S(plit) are evaluated. The additional threads in PWBO-P search on the lower and upper bound values of the optimal solution with different cardinality encodings for each thread. On the other hand, in PWBO-S the additional threads perform a parallel search on different upper bound values of the optimal solution.

6.2.1. Two threads

PWBO-T2 uses two threads according to what is described in Section 4, thus having one thread searching on the lower bound value and another thread searching on the upper bound value. The following versions of PWBO-T2 have been evaluated:

- PWBO-T2-V1: uses the pseudo-Boolean representation for the lower and upper bound search. This version corresponds to the original PWBO with 2 threads [35].
- PWBO-T2-V2: uses the *ladder* encoding for the lower bound search and the *sorters* encoding for the upper bound search. These encodings were the ones that performed best for the lower and upper bound search, respectively.
- PWBO-T2-V3: uses the dynamic encoding heuristic that was proposed in this paper for the upper bound search. Similarly to the previous version, for the lower bound search it uses the *ladder* encoding.

Figure 6 shows a cactus plot with running times for the different versions of PWBO-T2. Additionally, it also shows the running times of PWBO-T1 to be compared with the two-threaded versions. PWBO-T2-V2 and PWBO-T2-V3 clearly improve PWBO-T2-V1. This is due to the cardinality encodings that are used in PWBO-T2-V2 and PWBO-T2-V3. Indeed, even PWBO-T1 clearly outperforms PWBO-T2-V1 which shows the importance of selecting the most adequate encoding. Moreover, our best version with two threads is PWBO-T2-V3 which uses the dynamic encoding heuristic in the upper bound search. This version clearly outperforms all other versions. By improving the efficiency of the upper bound algorithm we were able to improve the performance of our solver with two threads. For simplicity's sake, in the remainder of the paper we will denote PWBO-T2-V3 as PWBO-T2.

Since we are using two threads, the solution can be found by: the lower bound search, the upper bound search or the cooperation between the lower bound search and the upper bound search. If the lower bound value is the same as the upper bound value, then the optimal solution has been found by the information of both searches. Table 5 shows the num-

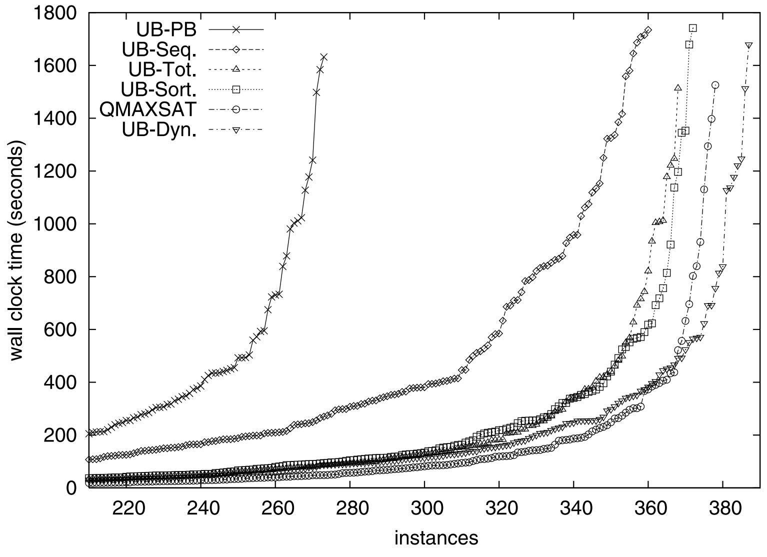

Fig. 5. Cactus plot with running times of upper bound solvers.

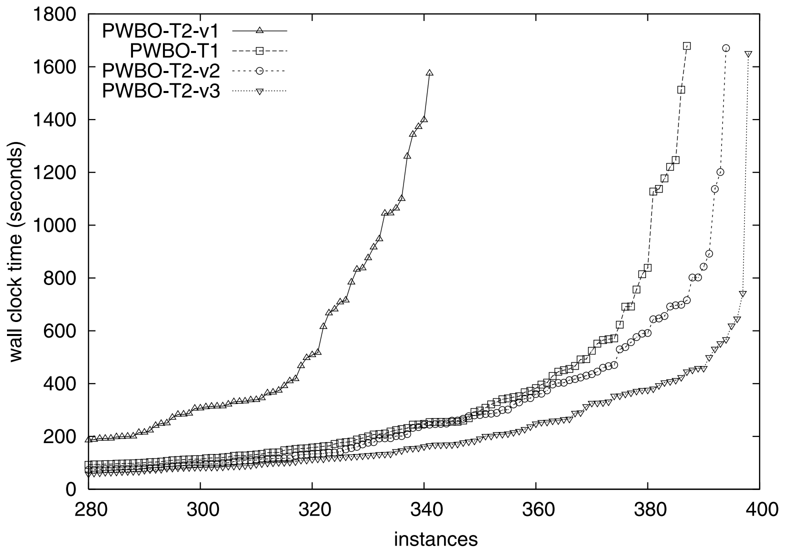

Fig. 6. Cactus plot with running times of solvers PWBO-T1 and the different versions of PWBO-T2.

ber of instances that were solved in each case. As expected, the upper bound search solves the most instances of the two-threaded version by solving 299 out of the 398 solved instances. However, the lower bound search contributes for the performance of PWBO-T2 by solving 65 instances. Moreover, 34 instances are solved by the combined information of the lower bound search and the upper bound search. For these instances the cooperation speeds up the solving process. Since the lower bound value was found to be the same as the upper bound value, it is not necessary for any of the threads to continue the search to prove optimality since their combined information already proves it. Additionally, Table 5 provides a strong stimulus to further improve our lower bound search, since a more efficient lower bound search may improve the overall performance of our parallel solver.

6.2.2. *Multithread based on portfolio*

PWBO-P(ortfolio) is based on a portfolio approach where each thread uses a different cardinality encoding [34]. To maintain a balance between lower and upper bound search, PWBO-P always uses the same num-

Table 5

Number of solved instances from PWBO-T2 divided by lower bound search, upper bound search and cooperation between searches

Benchmark	#I	PWBO-T2		
		Lower	Upper	Coop.
bcp-fir	56	25	19	12
bcp-hipp-yRa1	42	7	30	5
bcp-msp	26	0	23	3
bcp-mtg	40	0	40	0
bcp-syn	40	14	15	11
CircuitTrace	4	0	4	0
Haplotype	5	5	0	0
pbo-mqc	168	3	164	1
pbo-routing	15	11	2	2
PROTEIN_INS	2	0	2	0
Total	398	65	299	34

ber of threads for the upper bound and lower bound search. To build a portfolio of encodings for 4 and 8 threads we analyze Tables 2 and 3 and for each benchmark we try to maximize the number of solved instances by our portfolio of encodings for the *at-most-one* and *at-most-k* constraints Note that it may occur the situation where a cardinality encoding has an overall poor performance, but is the only one to be able to solve a given set of instances. In this case, it is interesting to incorporate such an encoding into a portfolio approach.

With 4 threads, PWBO-P-T4 uses the *Commander* and *Totalizer* encodings for the lower bound search and *Sorters* and pseudo-Boolean representation for the upper bound search. Although the ladder encoding performed better for the *at-most-one* constraint, it was mainly on solving the `bcp-mtg` and `pbo-mqc` benchmark sets. However, the performance of the upper bound search procedure on those instances is much better than the performance of the lower bound search. Therefore, the main gains of the ladder encoding are already covered by the *at-most-k* encodings for the upper bound search. A similar reasoning is applied to the pseudo-Boolean representation for the upper bound search. Even though this representation is less effective in general, it is the best performing encoding for solving the `pbo-mqc` benchmark set. Hence, this portfolio of cardinality encodings allows for a diversification of the search in all benchmarks.

With 8 threads, PWBO-P-T8 can use more encodings and therefore can further increase the diversification of the search. For the upper bound search, all four available encodings are used. For the lower bound search,

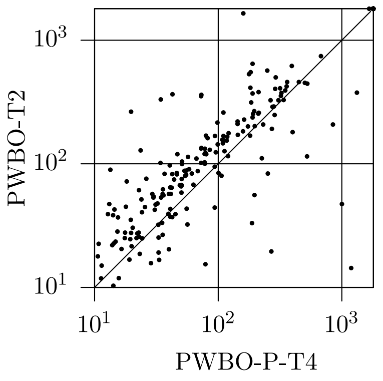

Fig. 7. Running times of PWBO-T2 and PWBO-P-T4.

we have selected the following encodings: *Commander*, *Totalizer*, *Ladder* and *Product*. The ladder encoding was now selected due to its overall robustness. On the other hand, even though the product encoding is less effective than other encodings, we have noticed that when it solves a given instance, it can be faster than when using other encodings. This has already been observed before [16]. Hence, for speedup reasons, we have decided to include the product encoding on our portfolio of cardinality encodings with 8 threads.

Figure 7 shows a scatter plot with running times of PWBO-T2 and PWBO-P-T4. The portfolio approach with 4 threads outperforms PWBO-T2 on most instances. However, there are some instances where PWBO-T2 performs better. PWBO-T4 does not use the *Totalizer* encoding whereas the PWBO-T2 uses a dynamic encoding heuristic that chooses that encoding for some instances. PWBO-T4 may be further improved if we consider a variation of the presented dynamic encoding heuristic. For the upper bound search, one thread could always use the *Sorters* encoding whereas the other thread could use a dynamic encoding heuristic that would select between the *Totalizer* encoding and the pseudo-Boolean representation.

Figure 8 compares PWBO-P-T4 with PWBO-P-T8. Notice that PWBO-P-T8 is able to solve more instances and with better running times than PWBO-P-T4 on most of the instances. This shows that even with 8 threads we are still able to increase the diversification of the search by adding different cardinality encodings.

6.2.3. Multithread based on splitting

PWBO-S(plit) is based on splitting the search space according to what was described in Section 4. One thread searches on the lower bound value of the optimal solution, another thread searches on the upper

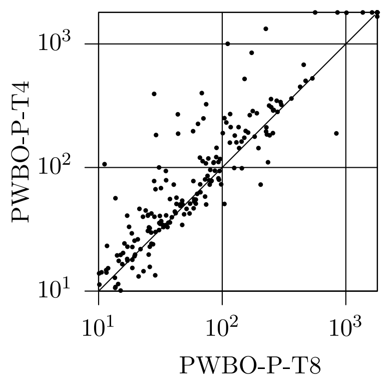

Fig. 8. Running times of PWBO-P-T4 and PWBO-P-T8.

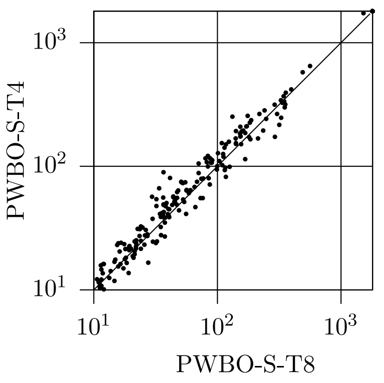

Fig. 10. Running times of PWBO-S-T4 and PWBO-S-T8.

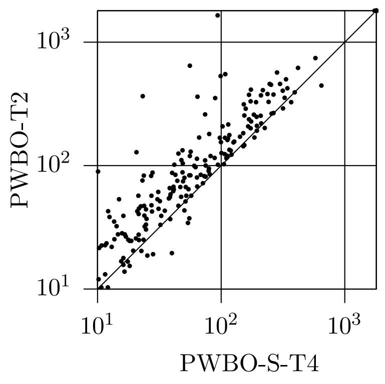

Fig. 9. Running times of PWBO-T2 and PWBO-S-T4.

bound value of the optimal solution, and the remaining threads search on different upper bound values. The first version, PWBO-S-V1 used the pseudo-Boolean representation for all threads [35]. This version under performed since it did not use cardinality encodings and therefore could only solve 345 instances with 4 threads and 347 instances with 8 threads.

PWBO-S-V2 uses the dynamic encoding heuristic proposed in Section 3 in all threads that are searching on the upper bound value of the optimal solution. As for the lower bound search, it uses the *ladder* encoding. Due to the cardinality encodings, PWBO-S-V2 increases its performance and is now able to solve 399 instances with 4 and 8 threads. Since PWBO-S-V2 is much more efficient than PWBO-S-V1, we will focus on PWBO-S-V2. For simplicity, in the remainder of the paper PWBO-S-V2 will be named as PWBO-S. For 4 and 8 threads we denote PWBO-S as PWBO-S-T4 and PWBO-S-T8, respectively.

Figure 9 shows a scatter plot comparing running times of PWBO-T2 and PWBO-S-T4. Each point in the plot corresponds to a problem instance, where the x-axis corresponds to the run time required by PWBO-T2 and the y-axis corresponds to the run time required by PWBO-S-T4. Instances that are trivially solved by both approaches (in less than 10 s) are not shown in the plot. The plot clearly shows that PWBO-S-T4 outperforms PWBO-T2, thus showing that the performance of the solver clearly improves with the increase of the number of threads from 2 to 4.

Figure 10 shows a scatter plot comparing running times of PWBO-S-T4 and PWBO-S-T8. Even though there is a slight improvement in time with the increase of the number of threads from 4 to 8, it is not as clear as before. With the split strategy, using more threads increases the number of threads that are searching on local upper bound values of the optimal solution. If the interval between the lower and upper bound values is small, then the threads that are searching on local upper bound values may be searching on similar values, thus performing redundant search. This may explain why the performance with 8 threads is not significantly better than the performance with 4 threads.

6.2.4. *Impact of clause sharing*

In Section 5 we have described the sharing mechanism of PWBO. It is expected that sharing learned clauses can help to further prune the search space and boost the performance of the parallel solver.

To evaluate the impact of sharing learned clauses we run the different versions of PWBO with and without clause sharing. Table 6 shows the speedup gain of sharing learned clauses. The speedup is determined by the ratio between the total solving time of the solver with and without clause sharing. Only instances that were solved by the solver with and without clause sharing are considered for the total solving time. Therefore, the speedup shows how many times the solver with clause sharing was faster than the solver without clause sharing. For example, PWBO-T2 shows a speedup of

Table 6

Speedup gain of sharing learned clauses

Solver	Speedup
PWBO-T2	1.36
PWBO-P-T4	1.39
PWBO-P-T8	1.48
PWBO-S-T4	1.40
PWBO-S-T8	1.42

1.36, i.e. it was 1.36× faster when learned clauses were shared.

Sharing learned clauses has a clear speedup on the solving times of the solvers. Moreover, increasing the number of threads increases the gains of sharing learned clauses. PWBO-P shows a clear improvement in speedup when using 8 threads compared to 4 threads. This may be explained by the diversification of the search provided by the cardinality encodings since learned clauses from different search spaces are exchanged. On the other hand, PWBO-S only shows a small improvement in speedup when using 8 threads compared to 4 threads. Since PWBO-S-T8 uses more threads for local bound search, if the interval between the lower bound value and the upper bound value is small, they will be searching on similar local upper bound values. Therefore, the diversification of the search will be small. This may explain why the gains of clause sharing from 4 to 8 threads are small when using the splitting approach.

The main improvement from clause sharing is in the speedup of the solver, since the number of solved instances does not increase significantly with clause sharing. For example, PWBO-S-T8 solves the same number of instances with and without clause sharing. The largest improvement can be seen in PWBO-P-T8 since it can solve more 4 instances with clause sharing.

A more detailed view of the impact of clause sharing can be seen in Figs 11 and 12. They provide scatter plots with the run times of PWBO-P-T8 and PWBO-S-T8 with and without sharing. It is clear that sharing learned clauses speedup the parallel solvers.

6.2.5. *Overview*

We have just presented the different versions of our parallel MaxSAT solver PWBO. Figure 13 shows a cactus plot with running times of the different PWBO versions. Since PWBO-T1 outperforms the best sequential solver (QMAXSAT 0.4) we did not compare PWBO versions against other sequential solvers. Furthermore, SAT4J MAXSAT [25] and SAT4J MAXSAT RES//CP were not evaluated since their performance is

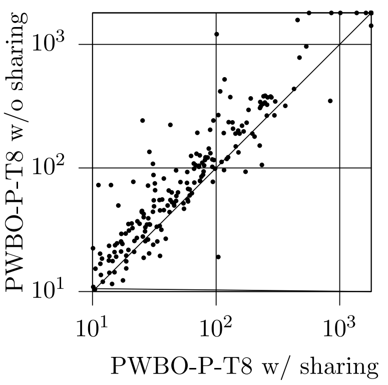

Fig. 11. Running times of PWBO-P-T8 with and without clause sharing.

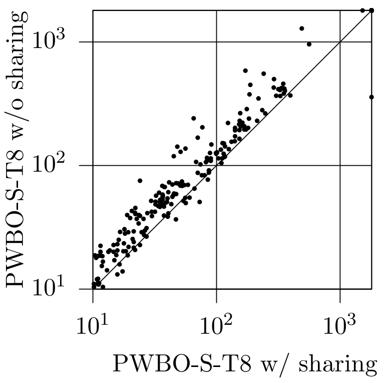

Fig. 12. Running times of PWBO-P-T8 with and without clause sharing.

not comparable to the remaining state-of-the-art partial MaxSAT solvers. For the 497 instances tested, SAT4J MAXSAT 2.2.3 and SAT4J MAXSAT RES//CP can only solve 277 and 290 instances, respectively.

The results are clear: all parallel solvers outperform the sequential solver. Moreover, we can see a clear improvement between PWBO-T2 and the parallel solvers with 4 and 8 threads. When evaluating a parallel solver, the wall clock time is always considered since it measures the real time that a solver used to solve the instances. From a user point of view, real time is clearly more important than CPU time. On the other hand, if we analyze the CPU resources, then parallel solvers with n threads are allowed to use n times more CPU time than sequential solvers. Figure 13 shows that when considering CPU time the most efficient solver is PWBO-T2, since it can solve 397 instances within 900 wall clock seconds, i.e. with a time limit of 1800 CPU seconds. Remember that with the same CPU resources PWBO-T1 is only able to solve

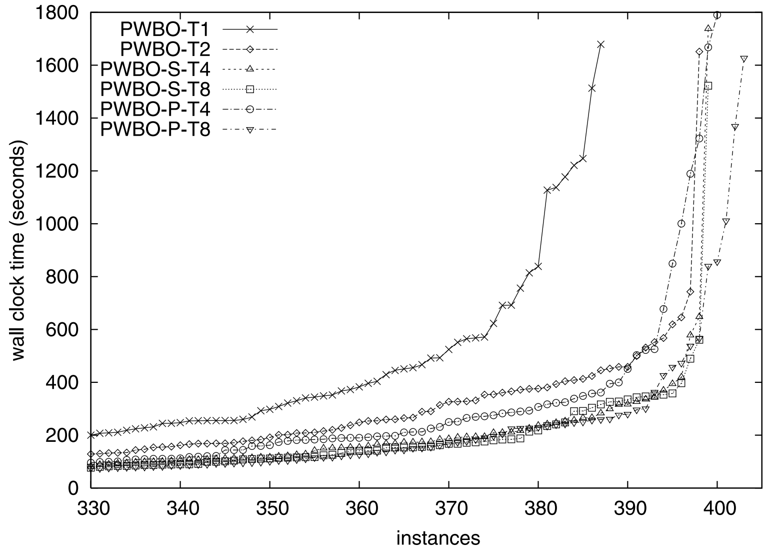

Fig. 13. Cactus plot with running times of the different PWBO versions.

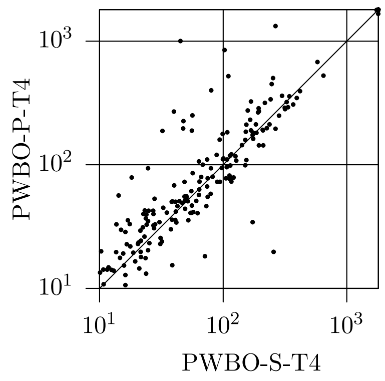

Fig. 14. Running times of PWBO-S-T4 and PWBO-P-T4.

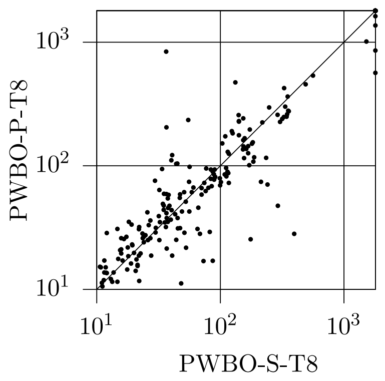

Fig. 15. Running times of PWBO-S-T8 and PWBO-P-T8.

387 instances. Moreover, even when considering parallel solvers with 4 threads, they can still outperform PWBO-T1 within 1800 CPU seconds. With 450 wall clock seconds, PWBO-S-T4 can solve 396 instances, whereas PWBO-P-T4 can solve 389 instances. On the other hand, when considering CPU time, our solvers with 8 threads are less efficient than PWBO-T1.

Figure 14 compares PWBO-P-T4 and PWBO-S-T4. PWBO-S-T4 solves one less instance than PWBO-P-T4. However, PWBO-S-T4 outperforms PWBO-P-T4 on the running time for most instances. Figure 15 compares PWBO-P-T8 and PWBO-S-T8. PWBO-P-T8 solves more 4 instances than PWBO-S-T4, even though Fig. 15 does not show a clear winner.

Table 7 shows an overview of the speedup on instances that were solved by all versions of PWBO. We choose to compare the speedup regarding PWBO-T1

Table 7
Speedup on the instances solved by all PWBO solvers

Solver	Time (s)	Speedup
PWBO-T1	40,415.57	1.00
PWBO-T2	29,038.06	1.39
PWBO-P-T4	23,389.74	1.73
PWBO-S-T4	17,996.43	2.25
PWBO-P-T8	16,886.84	2.39
PWBO-S-T8	16,613.28	2.43

and not the original version of WBO since PWBO-T1 is much more efficient than WBO. In fact, PWBO-T1 solves 387 instances whereas WBO solves only 317 instances. Moreover, PWBO-T1 outperforms the best sequential solver (QMAXSAT 0.4).

The speedup increases with the number of threads being 1.4× faster with 2 threads, 2.3× faster with

4 threads and 2.4× faster with 8 threads. PWBO-S-T4 solves one less instance than PWBO-P-T4 but has a larger speedup on the instances that were solved by both. However, when using 8 threads both PWBO-S-T8 and PWBO-P-T8 have similar speedups. This supports our previous findings that PWBO-P scales better than PWBO-S with an increasing number of threads. Indeed, PWBO-S-T8 is only 1.1× faster than PWBO-S-T4, whereas PWBO-P-T8 is 1.4× faster than PWBO-P-T4.

7. Conclusions and future work

This paper introduces PWBO, a new parallel solver for MaxSAT. This work was in part motivated by the recent success of parallel SAT solvers and also taking into account that parallel algorithms for Boolean optimization are scarce. Three versions of PWBO were proposed. The first version, PWBO-T2, uses two threads, one thread searching on the lower bound value of the optimal solution, and another thread searching on the upper bound value of the optimal solution. The second version, PWBO-P, is based on a portfolio approach using several threads to simultaneously search on the lower and upper bound values of the optimal solution. These threads differ between themselves in the encoding used for cardinality constraints, thus increasing the diversification of the search. The third version, PWBO-S, is based on a splitting approach searching on different values of the upper bound. The parallel search on the local upper bound values leads to updates on the lower and upper bound values that will reduce the search space.

This paper also examines a large number of cardinality encodings and evaluated their performance for solving the MaxSAT problem. Overall, the ladder encoding showed the best performance for the *at-most-one* cardinality constraints. As expected, when the number of variables is small it is better to use the pairwise encoding or pseudo-Boolean representation. On the other hand, when the number of variables in the cardinality constraint is large, it is better to encode the *at-most-one* cardinality constraint into CNF. For the *at-most-k* cardinality constraint, the sorter encoding showed the best performance. In general, it is better to translate the *at-most-k* cardinality constraint into CNF. However, in some cases, using the native pseudo-Boolean representation can be more effective. Therefore, a dynamic encoding heuristic that selects the most adequate encoding for each cardinality constraint is proposed in the paper. PWBO-T1 is the sequential version of our parallel solver and it performs upper bound search with the dynamic encoding heuristic. Experimental results show that the dynamic encoding heuristic outperforms all other encodings and that PWBO-T1 outperforms the best sequential solver (QMAXSAT 0.4) from the MaxSAT Evaluation 2011.

For two threads, experimental results show that most instances are solved by the upper bound search. Even though the lower bound search cooperates in solving the instances, it does not perform as well as the upper bound search. However, there are other unsatisfiability-based algorithms [2] that perform better than our implementation of the unsatisfiability-based algorithm for partial MaxSAT problems. As future work, we propose to improve our lower bound search algorithm, since a more efficient lower bound search may improve the overall performance of PWBO.

Experimental results also show that PWBO improves in performance with the increasing number of threads. PWBO-S-T4 outperforms PWBO-P-T4 on most instances. However, PWBO-S does not scale very well and its performance with 8 threads is only slightly better than with 4 threads. On the other hand, PWBO-P performance improves significantly when increasing the number of threads from 4 to 8. This shows that even with 8 threads, using different cardinality encodings still increases the diversity of the search.

Wall clock time is usually considered when evaluating a parallel solver. However, even if we consider CPU time, PWBO-T2, PWBO-S-T4 and PWBO-P-T4 still outperform the best sequential solver, PWBO-T1.

Experimental results with clause sharing show that sharing clauses does not have a strong impact on the number of solved instances. Nevertheless, the running times of PWBO are greatly improved when sharing learned clauses between threads.

As future work, we plan to build a hybrid version between splitting and portfolio. We could start with a splitting strategy and when the interval between the lower and upper bound values becomes small we can change to a portfolio approach. This may improve the efficiency of our parallel solver, namely when using 8 threads. Other future directions include the implementation of other *at-most-k* encodings, namely, cardinality networks [6] and pairwise cardinality networks [13]. These encodings are also based on sorters and are expected to further improve our portfolio of cardinality encodings.

Acknowledgements

This work was partially supported by FCT under research projects iExplain (PTDC/EIA-CCO/102077/2008), ParSAT (PTDC/EIA-EIA/103532/2008), ASPEN (PTDC/EIA-CCO/110921/2009) and INESC-ID multiannual funding through the PIDDAC program funds.

References

[1] X. An, M. Koshimura, H. Fujita and R. Hasegawa, QMaxSAT version 0.3 and 0.4, in: *Proc. International Workshop on First-Order Theorem Proving*, Bern, Switzerland, 2011.

[2] C. Ansótegui, M. Bonet and J. Levy, Solving (weighted) partial MaxSAT through satisfiability testing, in: *Proc. International Conference on Theory and Applications of Satisfiability Testing*, Springer-Verlag, Berlin, 2009, pp. 427–440.

[3] C. Ansótegui and F. Manyà, Mapping problems with finite-domain variables into problems with Boolean variables, in: *Proc. International Conference on Theory and Applications of Satisfiability Testing*, Springer-Verlag, Berlin, 2004, pp. 1–15.

[4] J. Argelich, C.M. Li and F. Manyà, An improved exact solver for partial max-sat, in: *Proc. International Conference on Non-convex Programming: Local and Global Approaches*, Rouen, France, 2007, pp. 230–231.

[5] R. Asín, R. Nieuwenhuis, A. Oliveras and E. Rodríguez-Carbonell, Practical algorithms for unsatisfiability proof and core generation in SAT solvers, *AI Commun.* **23**(2,3) (2010), 145–157.

[6] R. Asín, R. Nieuwenhuis, A. Oliveras and E. Rodríguez-Carbonell, Cardinality networks: a theoretical and empirical study, *Constraints* **16**(2) (2011), 195–221.

[7] G. Audemard and L. Simon, Predicting learnt clauses quality in modern SAT solvers, in: *Proc. International Joint Conference on Artificial Intelligence*, IJCAI/AAAI, Pasadena, CA, 2009, pp. 399–404.

[8] O. Bailleux and Y. Boufkhad, Efficient CNF encoding of Boolean cardinality constraints, in: *Proc. International Conference on Principles and Practice of Constraint Programming*, Springer-Verlag, Berlin, 2003, pp. 108–122.

[9] A. Biere, Lingeling, plingeling, PicoSAT and PrecoSAT at SAT race 2010, FMV Report Series, Technical Report 10/1, Johannes Kepler University, Linz, 2010.

[10] M.L. Bonet, J. Levy and F. Manyà, Resolution for Max-SAT, *Artif. Intell.* **171**(8,9) (2007), 606–618.

[11] M. Büttner and J. Rintanen, Satisfiability planning with constraints on the number of actions, in: *Proc. International Conference on Automated Planning and Scheduling*, Monterey, CA, 2005, pp. 292–299.

[12] J. Chen, A new SAT encoding of the at-most-one constraint, in: *Proc. International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, St Andrews, Scotland, 2010.

[13] M. Codish and M. Zazon-Ivry, Pairwise cardinality networks, in: *Proc. International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, Springer-Verlag, Berlin, 2010, pp. 154–172.

[14] S. Darras, G. Dequen, L. Devendevill and C.M. Li, On inconsistent clause-subsets for Max-SAT solving, in: *Proc. International Conference on Principles and Practice of Constraint Programming*, 2007, pp. 225–240.

[15] N. Eén and N. Sörensson, Translating pseudo-Boolean constraints into SAT, *J. Satisfiabil. Boolean Model. Comput.* **2** (2006), 1–26.

[16] A.M. Frisch and P.A. Giannaros, SAT encodings for the at-most-k constraint, in: *Proc. International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, St Andrews, Scotland, 2010.

[17] A.M. Frisch, T.J. Peugniez, A.J. Doggett and P. Nightingale, Solving non-Boolean satisfiability problems with stochastic local search: a comparison of encodings, *J. Automat. Reason.* **35**(1–3) (2005), 143–179.

[18] Z. Fu and S. Malik, On solving the partial Max-SAT problem, in: *Proc. International Conference on Theory and Applications of Satisfiability Testing*, Springer-Verlag, Berlin, 2006, pp. 252–265.

[19] I.P. Gent and P. Nightingale, A new encoding of all different into SAT, in: *Proc. International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, Toronto, Canada, 2004.

[20] L. Guo, Y. Hamadi, S. Jabbour and L. Sais, Diversification and intensification in parallel SAT solving, in: *Principles and Practice of Constraint Programming*, Springer-Verlag, Berlin, 2010, pp. 252–265.

[21] Y. Hamadi, S. Jabbour and L. Sais, ManySAT: a parallel SAT solver, *J. Satisfiabil. Boolean Model. Comput.* **6** (2009), 245–262.

[22] Y. Hamadi, S. Jabbour and L. Sais, Control-based clause sharing in parallel SAT solving, in: *Proc. International Joint Conference on Artificial Intelligence*, IJCAI/AAAI, Pasadena, CA, 2009, pp. 499–504.

[23] W. Klieber and G. Kwon, Efficient CNF encoding for selecting 1 from N objects, in: *Proc. International Workshop on Constraints in Formal Verification*, Bremen, Germany, 2007.

[24] S. Kottler and M. Kaufmann, SArTagnan – A parallel portfolio SAT solver with lockless physical clause sharing, in: *Pragmatics of SAT Workshop*, Ann Arbor, MI, 2011.

[25] D. Le Berre and A. Parrain, The Sat4j library, release 2.2 system description, *J. Satisfiabil. Boolean Model. Comput.* **7** (2010), 59–64.

[26] C.M. Li, F. Manyà and J. Planes, New inference rules for Max-SAT, *J. Artif. Intell. Res.* **30** (2007), 321–359.

[27] H. Lin and K. Su, Exploiting inference rules to compute lower bounds for Max-SAT solving, in: *Proc. International Joint Conference on Artificial Intelligence*, IJCAI/AAAI, Pasadena, CA, 2007, pp. 2334–2339.

[28] V. Manquinho, J. Marques-Silva and J. Planes, Algorithms for weighted Boolean optimization, in: *Proc. International Conference on Theory and Applications of Satisfiability Testing*, Springer-Verlag, Berlin, 2009, pp. 495–508.

[29] V. Manquinho, R. Martins and I. Lynce, Improving unsatisfiability-based algorithms for Boolean optimization, in: *Proc. International Conference on Theory and Applications of Satisfiability Testing*, Springer-Verlag, Berlin, 2010, pp. 181–193.

[30] J. Marques-Silva and I. Lynce, Towards robust CNF encodings of cardinality constraints, in: *Proc. International Conference on Principles and Practice of Constraint Programming*, Springer-Verlag, Berlin, 2007, pp. 483–497.

[31] J. Marques-Silva and V. Manquinho, Towards more effective unsatisfiability-based maximum satisfiability algorithms, in: *Proc. International Conference on Theory and Applications of Satisfiability Testing*, Springer-Verlag, Berlin, 2008, pp. 225–230.

[32] J. Marques-Silva and J. Planes, Algorithms for maximum satisfiability using unsatisfiable cores, in: *Design, Automation and Testing in Europe Conference*, ACM Press, New York, NY, 2008, pp. 408–413.

[33] J. Marques-Silva and K. Sakallah, GRASP: a new search algorithm for satisfiability, in: *Proc. International Conference on Computer-Aided Design*, IEEE Computer Society Press, Washington, DC, 1996, pp. 220–227.

[34] R. Martins, V. Manquinho and I. Lynce, Exploiting cardinality encodings in parallel maximum satisfiability, in: *Proc. International Conference on Tools with Artificial Intelligence*, IEEE Computer Society Press, Washington, DC, 2011, pp. 313–320.

[35] R. Martins, V. Manquinho and I. Lynce, Parallel search for Boolean optimization, in: *Proc. RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, Barcelona, Spain, 2011.

[36] S. Prestwich, Variable dependency in local search: prevention is better than cure, in: *Proc. International Conference on Theory and Applications of Satisfiability Testing*, Springer-Verlag, Berlin, 2007, pp. 107–120.

[37] C. Sinz, Towards an optimal CNF encoding of Boolean cardinality constraints, in: *Proc. International Conference on Principles and Practice of Constraint Programming*, Springer-Verlag, Berlin, 2005, pp. 827–831.

[38] L. Zhang, C.F. Madigan, M.W. Moskewicz and S. Malik, Efficient conflict driven learning in Boolean satisfiability solver, in: *Proc. International Conference on Computer-Aided Design*, IEEE Press, Piscataway, NJ, 2001, pp. 279–285.

[39] L. Zhang and S. Malik, Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications, in: *Design, Automation and Testing in Europe Conference*, IEEE Computer Society Press, Washington, DC, 2003, pp. 10880–10885.

Copyright of AI Communications is the property of IOS Press and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.