Clause Sharing in Deterministic Parallel Maximum Satisfiability

Ruben Martins, Vasco Manquinho, and Inês Lynce

IST/INESC-ID, Technical University of Lisbon, Portugal
{ruben,vmm,ines}@sat.inesc-id.pt

Abstract. Multicore processors are becoming the dominant platform in modern days. As a result, parallel Maximum Satisfiability (MaxSAT) solvers have been developed to exploit this new architecture. Sharing learned clauses in parallel MaxSAT is expected to help to further prune the search space and boost the performance of a parallel solver. Yet, so far it has not been made clear which learned clauses should be shared among the different threads. This paper studies the impact of clause sharing heuristics. Evaluating these heuristics can be a hard task in parallel MaxSAT because existing solvers suffer from non-determinism behavior, i.e. several runs of the same solver can lead to different solutions. This is a clear downside for applications that require solving the same problem instance more than once, such as the evaluation of heuristics. For a fair evaluation, this paper presents the first deterministic parallel MaxSAT solver that ensures reproducibility of results. By using a deterministic solver one can independently evaluate the gains coming from the use of different heuristics rather than the non-determinism of the solver. Experimental results show that sharing learned clauses improves the overall performance of parallel MaxSAT solvers. Moreover, the performance of the new deterministic solver is comparable to the corresponding non-deterministic version.

1 Introduction

Maximum Satisfiability (MaxSAT) is an optimization version of Boolean Satisfiability (SAT) for which new algorithms have been proposed [8, 17, 1, 2, 13, 14]. These new algorithms for MaxSAT are based on iterative calls to a SAT solver and contrast with previous solvers based on the classical branch and bound approach [3, 16, 12, 15]. One of the new approaches is based on the ability of SAT solvers to produce certificates of unsatisfiability. In these algorithms, unsatisfiable subformulas are identified and are iteratively relaxed at each step of the MaxSAT algorithm [8, 17, 1]. This approach corresponds to a lower bound search on the optimal value of the MaxSAT solution. A complementary approach is to make an upper bound linear search on the MaxSAT solution [14]. In this case,

Proceedings of the 19th RCRA workshop on *Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion* (RCRA 2012).
In conjunction with AI*IA 2012, Rome, Italy, June 14–16, 2012.

after a solution is found, new constraints are added such that solutions with a higher or equal value are excluded. This approach had already been successfully used for other Boolean optimization problems [6, 7, 22].

Considering the existence of these complementary algorithms for MaxSAT and the fact that multicore processors are now commonly used, new parallel MaxSAT algorithms have been proposed to exploit this new architecture [19, 18]. These parallel solvers are based on a portfolio approach where some threads apply unsatisfiability-based MaxSAT algorithms, while other threads make a linear search on the value of the optimal solution. Therefore, these algorithms simultaneously search on the lower and upper bound values of the optimal solution. Searching in both directions and sharing learned clauses between these two orthogonal approaches makes the search more efficient. However, it is not clear which clauses should be shared among the different threads. The problem of determining if a shared clause will be useful in the future remains challenging, and in practice heuristics are used to select which learned clauses should be shared.

This paper includes and extends the paper already published at the Learning and Intelligent OptimizatioN Conference (LION'12) [20] where the main contributions are: (1) a new heuristic for clause sharing that freezes shared clauses until they are expected to be useful and (2) an empirical evaluation of static, dynamic and freezing heuristics for clause sharing. In this paper we further detail these procedures, as well as a new description of the deterministic parallel MaxSAT solver used to evaluate the different clause sharing heuristics. It should be noted that this is the first deterministic parallel MaxSAT solver. The determinism is a fundamental feature if one wants to integrate the solver in an application where the same problem instance must be solved more than once and the results must be fully reproducible.

The organization of the paper is as follows. First, the MaxSAT problem is defined and MaxSAT algorithmic approaches are briefly characterized. Next, section 3 describes different clause sharing heuristics that will be analyzed in the paper. Section 4 provides a description of our deterministic parallel MaxSAT solver and an experimental evaluation of the different clause sharing heuristics is presented in section 5. Finally, the paper concludes and suggests future work.

2 Preliminaries

A Boolean formula in conjunctive normal form (CNF) is defined as a conjunction ($\wedge$) of clauses, where a clause is a disjunction ($\vee$) of literals and a literal is a Boolean variable x or its negation $\bar{x}$. A Boolean variable may be assigned truth values true or false. A positive (negative) literal x ($\bar{x}$) is said to be satisfied if the respective variable is assigned value true (false). A positive (negative) literal x ($\bar{x}$) is said to be unsatisfied if the respective variable is assigned value false (true). A variable (and respective literals) not assigned is said to be unassigned. A clause is said to be satisfied if at least one of its literals is satisfied. A clause is said to be unsatisfied if all of its literals are unsatisfied. A clause is said to be unit if all literals but one are unsatisfied and the remaining literal is unassigned. Otherwise,

a clause is said to be unresolved. A formula is satisfied if all of its clauses are satisfied. The Boolean Satisfiability (SAT) problem is to decide whether there exists an assignment that makes the formula satisfied. Such assignment is called a solution.

The Maximum Satisfiability (MaxSAT) problem is an optimization version of the SAT problem which consists in finding an assignment that minimizes (maximizes) the number of unsatisfied (satisfied) clauses. In the remainder of the paper, it is assumed that MaxSAT is defined as a minimization problem. MaxSAT has several variants such as partial MaxSAT, weighted MaxSAT and weighted partial MaxSAT. In the partial MaxSAT problem, some clauses 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.

2.1 MaxSAT Algorithms

Recently, new algorithms for MaxSAT have been proposed [8, 17, 1, 13, 14]. As a result, state of the art MaxSAT algorithms are now able to solve many challenging problem instances. One approach for solving MaxSAT is to make a linear search on the objective value. In this case, a new relaxation variable is initially added to each soft clause and the resulting formula is solved by a SAT solver. Whenever a solution is found, a new constraint on the relaxation variables is added such that solutions with an higher or equal value are excluded. Usually, this new constraint is translated into a set of propositional clauses so that a SAT solver can handle the resulting formula [14]. Otherwise, a pseudo-Boolean solver must be used. In this case, the solver deals natively with generalizations of propositional clauses. The algorithm stops when the resulting formula becomes unsatisfied.

A different approach for MaxSAT relies on modern SAT solvers having the ability to produce certificates of unsatisfiability [8]. The algorithm starts by considering all hard and soft clauses in the formula. At each iteration, an unsatisfiable subformula is identified and relaxed. The relaxation procedure usually consists of adding a new relaxation variable to each soft clause in the unsatisfiable subformula. Moreover, a new constraint is added such that at most one of the new relaxation variables can be assigned to true. Again, if one wants to continue using a SAT solver, this new constraint must be encoded into a set of propositional clauses. The algorithm stops when the resulting formula is satisfiable. Furthermore, several variants of this approach have been proposed [17, 1, 2, 13].

The parallel MaxSAT solver PWBO [18] used in this paper is based on having several threads running a portfolio of two orthogonal algorithms that follow the approaches just described: (i) an unsatisfiability-based algorithm that searches on the lower bound of the optimal solution and (ii) a linear search algorithm

that searches on the upper bound. Therefore, this solver performs a parallel search on both sides of the optimal solution. Furthermore, each thread uses different procedures to handle the new constraints added at each iteration of the search [18], thus diversifying the exploration of the search space.

Notice that PWBO is not limited to the best performing algorithm in the portfolio, since threads can cooperate by exchanging information on the lower and upper bounds found during the search, as well as exchanging learned clauses that can prune the search on the other threads. As a result, the parallel MaxSAT algorithm is complete, but non-deterministic. Although the algorithm always returns an optimal solution, it is not guaranteed that any two runs of the algorithm return the exact same solution, i.e. the exact same set of assignments to the variables. Moreover, the time spent may also have large variations.

3 Clause Sharing Heuristics

Clause sharing heuristics can be divided into three categories: (1) static, (2) dynamic and (3) freezing. The static heuristics share clauses within a given cutoff, whereas the dynamic heuristics adjust this cutoff during the search. Alternatively, the freezing heuristics temporarily delay the incorporation of shared clauses until they are expected to be useful in the context of the importing thread.

3.1 Static

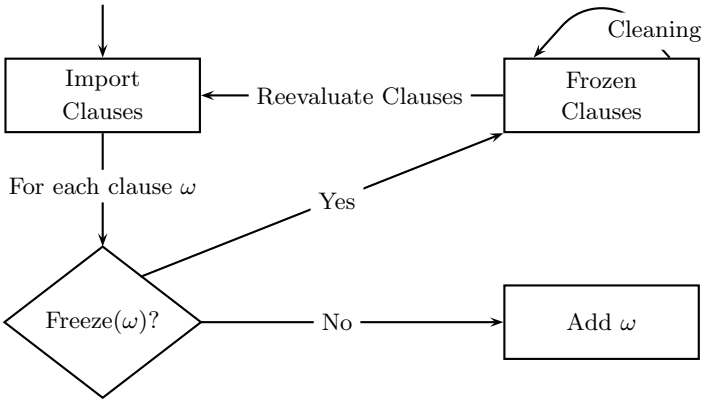
The static heuristics are the most used for clause sharing since they are simple but still efficient in practice. The following measures are used in these heuristics:

- *Size*: the clause size is given by the number of literals. Small clauses are expected to be more useful than larger clauses.
- *Literal Block Distance* (LBD) [5]: the literal block distance corresponds to the number of different decision levels involved in a clause. The decision level of a literal denotes the depth of the decision tree at which the corresponding variable was assigned a value. Clauses with small LBD are considered as more relevant.
- *Random*: randomly decide whether to share each learned clause with a given probability. This heuristic was designed to evaluate the other heuristics which are expected to be more effective than a random one.

3.2 Dynamic

It has been observed that the size of learned clauses tends to increase over time. Consequently, in parallel solving, any static limit may lead to halting the clause sharing process. Therefore, to continue sharing learned clauses it is necessary to dynamically increase the limit during search. Hamadi et al. [11] proposed the following dynamic heuristic. At every k conflicts (corresponding to a period

Fig. 1. Freezing procedure for sharing learned clauses



α) the throughput of shared clauses is evaluated between each pair of threads $(t_i \to t_j)$ according to the following heuristic:

$$\text{limit}^{\alpha+1}_{t_i \to t_j} = \begin{cases} \text{limit}^{\alpha}_{t_i \to t_j} + \text{quality}^{\alpha}_{t_i \to t_j} \times \frac{b}{\text{limit}^{\alpha}_{t_i \to t_j}} & \text{if sharing is small} \\ \text{limit}^{\alpha}_{t_i \to t_j} - (1 - \text{quality}^{\alpha}_{t_i \to t_j}) \times a \times \text{limit}^{\alpha}_{t_i \to t_j} & \text{if sharing is large} \end{cases},$$

where a and b are positive constants and the value of $\text{quality}^{\alpha}_{t_i \to t_j}$ corresponds to the quality of shared clauses that were exported from t_i and imported by t_j.

A shared clause is said to have *quality* [11] if at least half of its literals are active. A literal is *active* if its VSIDS heuristic [23] score is high, i.e. it is likely to be chosen as a decision variable in the near future. Hence, $\text{quality}^{\alpha}_{t_i \to t_j}$ gives the ratio between quality shared clauses and the total number of shared clauses in the period α. If the quality is high then the increase (decrease) in the size limit of shared clauses will be larger (smaller). The idea behind this heuristic is that the information recently received from a thread t_i is qualitatively linked to the information which could be received from the same thread t_i in the near future. In our experimental setting, we have selected $a = 0.125, b = 8$ and $\alpha = 3000$ conflicts. The throughput at each period is set to 750, i.e. if a thread t_j receives less than 750 shared learned clauses in the period α, it increases the limit of the size of shared clauses. Otherwise, this limit is decreased. These parameters are similar to the ones used by Hamadi et al. [11].

3.3 Freezing

There are possible drawbacks to importing clauses shared by other threads. One drawback is that the newly imported clauses do not become active in pruning the search space. Another possible drawback is that it might influence the exploration of the search space, such that the search becomes more closely related

with the exploration being performed in the thread from which the clauses originated. As a result, the diversificaton of the exploration of the search space is decreased by shifting the context of the current search in the importing thread.

Our motivation for the freezing heuristic is to only import shared clauses when they are expected to be useful in the near future. For that, the decision to import new learned clauses shared by other threads must take into consideration the current search context where these clauses are to be integrated. As a result, these new clauses should improve the efficiency of the search being carried out, without making a major change to the search context of the receiving thread.

Figure 1 illustrates the freezing procedure. Each shared clause ω is evaluated to determine if it will be frozen or imported. If ω is frozen then it will be reevaluated later. However, if ω is assigned the frozen state more than k times it is permanently deleted. When evaluating ω, our goal is to import clauses that are unsatisfied or that will become unit clauses in the near future. Next, the freezing heuristic is presented. According to the *status* of ω (satisfied, unsatisfied, unit or unresolved), it decides whether ω should be frozen:

- ω is *satisfied*: Let *level* denote the current decision level, $level_h(\omega)$ the highest decision level of the satisfied literals in ω, $unassignedLits(\omega)$ the number of unassigned literals in ω and $activeLits(\omega)$ the number of active literals in ω. If $(level - level_h(\omega) \leq c)$ and $(unassignedLits(\omega) - activeLits(\omega) \leq d)$ then ω is imported, otherwise it is frozen. A satisfied clause is expected to be useful in the near future if it is not necessary to backtrack significantly to make the clause unit. It is also important that the number of unassigned literals is small, otherwise the clause may not become unit in the near future. Active literals are also taken into consideration since they will be assigned in the near future.
- ω is *unsatisfied* or *unit*: ω is always imported;
- ω is *unresolved*: if $(unassignedLits(\omega) - activeLits(\omega) \leq d)$ then the clause is imported. Otherwise, it is frozen. Similarly to the satisfied case, if the number of unassigned literals is small then ω is likely to be unit in the near future.

In our experimental setting, we have selected $c = 31, d = 5$ and $k = 7$. In addition, the frozen clauses are reevaluated every 300 conflicts. These parameters were experimentally tuned. We note that freezing learned clauses has been recently proposed in the context of deletion strategies for learned clauses in SAT solving [4]. However, to the best of our knowledge, our solver is the only one that uses freezing shared clauses in a parallel solving context.

4 Deterministic Parallel MaxSAT

As described in section 2.1, current parallel MaxSAT solvers suffer from non-deterministic behavior. It is known that sharing learned clauses and exchanging information on the lower and upper bounds can prune the search space and boost the performance of the parallel solver. As a result, cooperation between

threads is essential for the performance of parallel solvers. However, this cooperation is also responsible for their non-deterministic behavior. To have a better understanding of the impact of the heuristics for clause sharing that have been proposed in section 3, it is necessary to use a deterministic version of the solver. The determinism allows for a fair evaluation of heuristics since it shows that the gains are coming from the use of different heuristics and not from the non-determinism of the solver.

In this section, we present the first deterministic parallel MaxSAT solver that ensures reproducibility of results. The deterministic solver uses synchronization points to exchange information between threads. Whenever a thread reaches a synchronization point it waits until the remaining threads reach the same point. Afterwards, when all threads reach the synchronization point, they share learned clauses and information regarding the lower and upper bounds. This synchronization guarantees the determinism of the cooperation between threads. A similar approach has been recently proposed for deterministic parallel SAT solving [10, 9]. Next, we describe how the SAT approach can be adapted for building a deterministic parallel MaxSAT solver.

4.1 Deterministic Solver

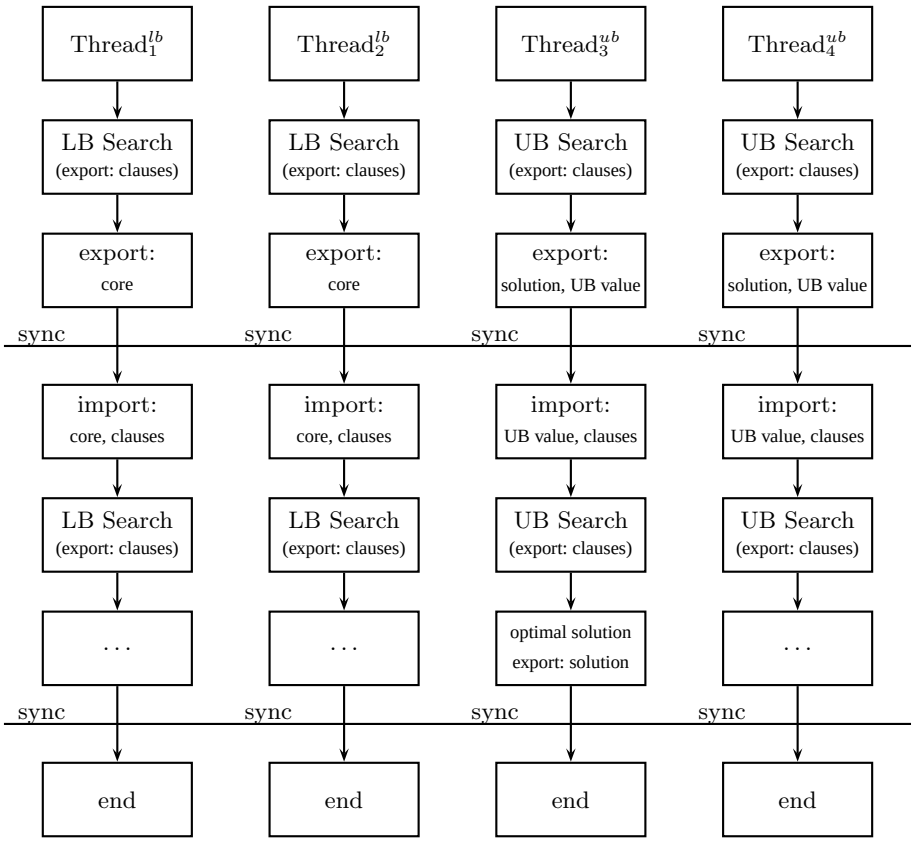
The portfolio version of the parallel MaxSAT solver PWBO [18] searches on the lower and upper bound values of the optimal solution. Half of the threads are used to search on the lower bound using an unsatisfiability-based algorithm (LB search), while the remaining threads search on the upper bound using a linear search algorithm (UB Search).

Our deterministic solver is built on top of PWBO (version 2.0). The goal of the deterministic solver is to be able to reproduce the same results on each problem instance by guaranteeing the following: (i) the solution reported by the solver is always the same and (ii) the search performed by each thread is also the same. The first requirement guarantees the determinism of the reported solution, whereas the second requirement guarantees the determinism of the solving time.

Figure 2 exemplifies an execution of the deterministic solver with 4 threads but it can be easily generalized for more threads. In this example, threads 1 and 2 search on the lower bound value of the optimal solution, while threads 3 and 4 search on the upper bound value of the optimal solution. Each thread begins by performing its search as in the non-deterministic solver [18]. Every time a clause is learned, it is exported to the remaining threads. However, in the deterministic solver, learned clauses are only incorporated in the solver at synchronization points. This contrasts to the non-deterministic version where learned clauses can be imported on-the-fly.

When a thread that is searching on the lower bound finds an unsatisfiable subformula (also know as a core) it stops its search and proceeds to the synchronization point. As can be seen in Figure 2, before entering the synchronization point each thread exports the core that was found during the last period. Note that if a core has not been found in the last period then nothing is exported. Each unsatisfiable core corresponds to an increase in the lower bound value and

Fig. 2. Execution of the deterministic solver based on syncronization points



is used in the unsatisfiability-based algorithm to iteratively relax the MaxSAT formula [17].

Consider k threads performing lower bound search. At a synchronization point, each of these k threads compares the cores that were found in the last period. Our goal is to import the core that corresponds to the largest increase in the lower bound value. If two threads found a core that corresponds to the same increase in the lower bound value, then the core with the smallest size is imported by all threads. If there are two cores that have the same size, then ties are broken considering the threads identifiers in increasing order. For example, consider in Figure 2 that thread t_1 and thread t_2 found a core with the same size. After the synchronization point, thread t_1 does not import the core from thread t_2. On the other hand, thread t_2 will discard the core that has been found in the last period and import the core exported by thread t_1.

Similarly to the non-deterministic version, all threads that are searching on the lower bound always have the same cores. This requires a synchronization point every time a core is found. For problem instances with a large number of unsatisfiable cores, this approach may result in high idle times since threads that are searching on the lower bound have to wait for all other threads to reach the synchronization point. On the other hand, since threads that are searching on the lower bound will always have the same cores, clause sharing may be more beneficial between these threads since they are always searching in equivalent formulas.

Threads that are searching on the upper bound value export their best solution and the corresponding upper bound value before entering the synchronization point. At a synchronization point, each thread imports the smallest upper bound value between all threads. As a result, all threads that are searching on the upper bound will have the same upper bound value after the synchronization point.

Learned clauses are also imported at synchronization points. Each thread imports the learned clauses that were exported by the remaining threads since the last synchronization point. Note that threads searching on the lower bound can also selectively import learned clauses from threads that are performing an upper bound search. The converse is also true [18]. In order to guarantee a deterministic behavior, learned clauses must be imported in the same order. Therefore, in our case, learned clauses are imported using an ascending order with respect to the threads identifiers.

In addition, we must also guarantee the determinism of the reported solution. For a given problem instance, the variable assignments of the optimal solution that the solver outputs must be always the same for all runs of the solver. Every time a new solution is exported, it is only recorded if its corresponding value is smaller than the best value found so far. If the new solution has the same value as the current best value, then the thread identifier is used to decide if the new solution is recorded or not. If the identifier of the exporting thread is smaller than the identifier of the thread where the previous solution was found, then the new solution is recorded. Otherwise, it is discarded. Finally, a thread stops when proves optimality. However, the remaining threads are only terminated when their next synchronization point is reached. This is done to guarantee the determinism of the reported solution, since new optimal solutions may be found. For example, in Figure 2 thread t_3 finds an optimal solution but it does not immediately terminates the solver. Instead, it waits until the remaining threads reach their next synchronization point to guarantee that no other optimal solutions have been found in the meantime.

4.2 Synchronization Points

The deterministic solver is based on synchronization points. However, to use synchronization points one must choose a deterministic measure. Hamadi et al. [10, 9] propose to use the number of conflicts as a measure for building the synchronization points. A simple strategy is to use a static number of conflicts to

Table 1. Comparison of the different heuristics for sharing learned clauses

	Heuristic	#Solved	Avg. #Clauses	Avg. Size	Time (s)	Speedup
	No sharing	137	–	–	32,188.57	1.00
Static	Random 30	134	10,140.22	128.21	27,394.46	1.18
Static	LBD 5	137	8,947.36	9.94	25,346.69	1.27
Static	Size 8	137	7,529.18	5.30	25,098.85	1.28
Static	Size 32	138	18,027.48	11.76	25,174.29	1.28
	Dynamic	138	13,296.28	7.33	24,218.84	1.33
	Freezing	140	16,228.53	11.01	21,611.21	1.49

determine when a thread should enter a synchronization point. For example, each thread has to perform k conflicts before entering the next synchronization point. Note that there are some issues to be considered when choosing the value for k. If k is small, then the number of synchronization points is high but learned clauses and information regarding the bounds is exchanged more frequently. On the other hand, if k is large, then the number of synchronization points is low but learned clauses and information regarding the bounds is exchanged less frequently. Therefore, there is a trade off between the idle time that results from having a large number of synchronization points and the frequency that information is exchanged between threads. In our experimental setting, we have set k to 100 conflicts.

5 Experimental Results

All experiments were run on the partial MaxSAT instances from the industrial category of the MaxSAT Evaluation 2011[1]. Instances that are easily solved have similar solving times with and without sharing learned clauses. Hence, if an instance takes less than 60 seconds to be solved it is not considered in this evaluation. Therefore, our results report only to a subset of 232 instances out of the 497 used in the MaxSAT Evaluation 2011. The results for the non-deterministic parallel solver were obtained by running the solver three times. Similarly to what is done when analyzing randomized solvers, the median time was taken into account. As a result, an instance must be solved by at least two of the three runs to be considered solved by the non-deterministic version. For the deterministic versions, each solver was run only once. All parallel solvers were run with 4 threads. The evaluation was performed on two AMD Opteron 6172 processors (2.1 GHz with 64 GB of RAM) running Fedora Core 13 with a timeout of 1,800 seconds (wall clock time). The tool RUNSOLVER [21] was used to control the execution of the solvers.

Table 1 shows the results of different sharing heuristics on a deterministic solver using 4 threads where two threads apply an unsatisfiability-based algorithm, while the other two threads use linear search on the value of the MaxSAT

[1] `http://www.maxsat.udl.cat/11/`

solution. Although there are threads running the same algorithmic approach, the search space exploration differs due to the way they handle the additional constraints added at each iteration of the solver [18]. For each heuristic considered, the number of solved instances, the average number of imported clauses by each thread, the average size of imported clauses, the solving time and the speedup are presented. Note that the solving time and the speedup in table 1 only refer to the 133 instances that were solved by all heuristics.

The first line in table 1 shows the data for the deterministic solver with no sharing of learned clauses between the several threads. For the static heuristics we started by trying a random heuristic that decides with probability 30% if it shares (or not) each learned clause. As a result, this leads to sharing clauses that may have a very large size. Notice that randomly sharing clauses can deteriorate the performance of the solver, since 3 instances less were solved. However, at the same time, for the instances that were solved, randomly sharing improved the performance of the solver when compared to not sharing any learned clauses.

Table 1 also shows the results for other static heuristics, namely using literal block distance with a maximum value of 5 (LBD 5), as well as sharing clauses with a size limit of 8 and 32 literals (Size 8 and Size 32). Other size limits were also evaluated with similar results. It was observed that if the limit is too small then the speedup is reduced since not many clauses are shared. On the other hand, if the limit is too large then the speedup is also reduced since many irrelevant clauses are shared. However, a size limit of 32 is comparable to a size limit of 8, since there are instances where learning larger clauses can be useful. In some cases, a static limit of 8 for clause sharing is too restrictive and does not allow the exchange of the clauses that are important for solving an instance. Nevertheless, notice that any of these heuristics always improve the performance of the solver in terms of time.

The dynamic heuristic outperforms the static heuristics but is outperformed by the freezing heuristic. In the dynamic heuristic, each thread starts by importing clauses with at most 8 literals. This cut-off is dynamically adjusted as described in section 3.2. Table 1 clearly shows the impact of this adjustment. The average number of imported clauses by each thread almost doubled from the static heuristic with size 8 to the dynamic heuristic that starts with a cut-off of size 8.

The freezing heuristic uses a static cut-off of size 32. However, it differs from the static heuristic of size 32 by delaying the incorporation of the received learned clauses until they are expected to be useful. Table 1 shows that the average number of imported clauses by each thread is smaller when using the freezing heuristic than when using the static heuristic of size 32. Notice that in the freezing heuristic some imported clauses may be deleted. If a clause is imported and in the following synchronization periods is not considered to be helpful for the solver, then this clause will be deleted.

To summarize, although sharing learned clauses does not improve the number of solved instances significantly, it does reduce the solving time considerably. The freezing heuristic clearly outperforms all other heuristics in terms of solving

Table 2. Comparison between the non-deterministic and deterministic solvers

Solver	#Solved	Time (s)	Avg. Idle CPU (%)	Speedup
Non-Deterministic	141	13,401.88	0	1.00
Deterministic	140	21,611.21	43.12	0.62

time and number of instances solved and provides a strong stimulus for further research.

The deterministic solver was built to perform a fair comparison between the different heuristics for clause sharing. Table 2 compares the deterministic solver using the freezing heuristic against the respective non-deterministic solver that uses the same heuristic. The comparison is done regarding the number of instances solved, the solving time, the average percentage of idle CPU time per problem instance and the speedup. The performance of the deterministic solver is comparable to the corresponding non-deterministic with respect to the number of instances solved. However, the deterministic solver is much slower than the non-deterministic solver. Table 2 shows that there is a correlation between the slowdown of the deterministic solver and the percentage of idle CPU time. For most of the time almost 2 out of 4 threads are idle. This may explain why the deterministic solver is 1.6× slower than the non-deterministic solver.

The high idle CPU time shown by the deterministic solver is mostly due to the synchronization mechanism. Different threads present different search behaviors and may reach a static synchronization point at different times. This problem is further accentuated in our solver since the size of the formula can differ substantially between threads. For example, threads that search on the upper bound of the optimal solution and use CNF encodings to encode the constraint on the upper bound value may have a formula that is several times larger than the formulas in other threads. As future work, we propose to improve our deterministic solver by implementing a dynamic period for each thread. A similar approach has been successfully used in parallel deterministic SAT solving [10, 9].

6 Conclusions

New parallel algorithms have been recently proposed for both SAT and MaxSAT solving. The main goal of these algorithms is to take advantage of multicore computer architectures by running several threads at the same time. Moreover, in a portfolio-based parallel solver, each thread runs a different algorithm on the same initial formula. However, instead of having a race of several algorithms for solving a given problem instance, collaborative procedures are usually integrated in the parallel solver. One of such procedures is to share learned clauses between the several threads, where each thread runs a different algorithm, thus allowing to prune the search space already explored in other threads.

In this paper different sharing heuristic procedures already proposed for parallel SAT solving are described and integrated in a MaxSAT parallel solver.

Moreover, a new heuristic based on the notion of freezing is proposed. This heuristic delays importing shared clauses by a given thread until it is considered relevant in the context of its own search. In order to proper evaluate the effectiveness of the several sharing heuristics, a deterministic parallel MaxSAT solver based in synchronization points was developed and details are given in this paper. This allows to have a parallel experimental setup such that the only variation is the sharing procedure.

Experimental results show that sharing learned clauses in a portfolio-based parallel MaxSAT solver does not increase significantly the number of solved instances. However, it does allow a considerable reduction of the solving time. Moreover, the new freezing heuristic outperforms all other heuristics both in solving time and number of solved instances. Finally, a preliminary analysis of the performance of our deterministic parallel MaxSAT solver against the non-deterministic version shows that the deterministic solver is much slower due to large idle times. It was observed that this is mainly due to the synchronization procedure. Nevertheless, this is the first deterministic parallel MaxSAT solver being proposed so far and the number of solved instances is similar to the non-deterministic version.

As future work one should consider aggregating several clause sharing heuristic criteria. Variations of the freezing heuristic can also be devised in order to take into consideration other information from the context of the search space being explored in the importing thread.

Finally, in order to observe the variance in the idle time, experimental results will be extended with additional runs of the deterministic solver. The large idle times in these preliminary results show that our deterministic parallel MaxSAT solver still has room for improvement. Currently, the synchronization points are statically determined, but a dynamic procedure such as the one already used in parallel SAT solving should produce better results. Moreover, specific features regarding MaxSAT solving should also be considered in order to produce a better synchronization of the threads.

Acknowledgements

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

References

1. C. Ansótegui, M. Bonet, and J. Levy. Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 427–440, 2009.
2. C. Ansótegui, M. Bonet, and J. Levy. A New Algorithm for Weighted Partial MaxSAT. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 3–8, 2010.

3. J. Argelich, C. M. Li, and F. Manyà. An improved exact solver for partial max-sat. In *Proceedings of the International Conference on Nonconvex Programming: Local and Global Approaches (NCP-2007)*, pages 230–231, 2007.
4. G. Audemard, J.-M. Lagniez, B. Mazure, and L. Sais. On Freezing and Reactivating Learnt Clauses. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 188–200, 2011.
5. G. Audemard and L. Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *International Joint Conferences on Artificial Intelligence*, pages 399–404, 2009.
6. P. Barth. A Davis-Putnam Enumeration Algorithm for Linear Pseudo-Boolean Optimization. Technical Report MPI-I-95-2-003, Max Plank Institute for Computer Science, 1995.
7. N. Eén and N. Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.
8. 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.
9. Y. Hamadi, S. Jabbour, C. Piette, and L. Sais. Deterministic Parallel DPLL. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):127–132, 2011.
10. Y. Hamadi, S. Jabbour, C. Piette, and L. Sais. Deterministic Parallel DPLL: System Description. In *Pragmatics of SAT Workshop*, 2011.
11. Y. Hamadi, S. Jabbour, and L. Sais. Control-Based Clause Sharing in Parallel SAT Solving. In *International Joint Conferences on Artificial Intelligence*, pages 499–504, 2009.
12. F. Heras, J. Larrosa, and A. Oliveras. MiniMaxSAT: An efficient weighted Max-SAT solver. *Journal of Artificial Intelligence Research*, 31:1–32, 2008.
13. F. Heras, A. Morgado, and J. Marques-Silva. Core-Guided Binary Search Algorithms for Maximum Satisfiability. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, pages 36–41, 2011.
14. M. Koshimura, T. Zhang, H. Fujita, and R. Hasegawa. QMaxSAT: A Partial Max-SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:95–100, 2012.
15. C. M. Li, F. Manyà, and J. Planes. New inference rules for Max-SAT. *Journal of Artificial Intelligence Research*, 30:321–359, 2007.
16. H. Lin and K. Su. Exploiting inference rules to compute lower bounds for MAX-SAT solving. In *International Joint Conferences on Artificial Intelligence*, pages 2334–2339, 2007.
17. V. Manquinho, J. Marques-Silva, and J. Planes. Algorithms for Weighted Boolean Optimization. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 495–508, 2009.
18. R. Martins, V. Manquinho, and I. Lynce. Exploiting Cardinality Encodings in Parallel Maximum Satisfiability. In *International Conference on Tools with Artificial Intelligence*, pages 313–320, 2011.
19. R. Martins, V. Manquinho, and I. Lynce. Parallel Search for Boolean Optimization. In *RCRA International Workshop on Experimental Evaluation of Algorithms for solving problems with combinatorial explosion*, 2011.
20. R. Martins, V. Manquinho, and I. Lynce. Clause Sharing in Parallel MaxSAT. In *Learning and Intelligent Optimization Conference*, 2012.
21. O. Roussel. Controlling a Solver Execution with the runsolver Tool. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):139–144, 2011.

22. H. Sheini and K. Sakallah. Pueblo: A Modern Pseudo-Boolean SAT Solver. In *Design, Automation, and Test in Europe (DATE) Conference*, pages 684–685, March 2005.
23. L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In *International Conference on Computer-Aided Design*, pages 279–285, 2001.