Constraints (2012) 17:304–347
DOI 10.1007/s10601-012-9121-3

SURVEY

An overview of parallel SAT solving

Ruben Martins · Vasco Manquinho · Inês Lynce

Published online: 11 April 2012
© Springer Science+Business Media, LLC 2012

Abstract Boolean satisfiability (SAT) solvers are currently very effective in practice. However, there are still many challenging problems for SAT solvers. Nowadays, extra computational power is no longer coming from higher processor frequencies. At the same time, multicore architectures are becoming predominant. Exploiting this new architecture is essential for the evolution of SAT solvers. Due to the increasing interest in parallel SAT solving, it is important to give an overview of what has been done so far. This paper presents an overview of parallel SAT solving and it is expected to be a valuable document for researchers in this field. This overview covers the main topics of parallel SAT solving, namely, different approaches and a variety of clause sharing strategies. Additionally, an evaluation of multicore SAT solvers is presented, showing the evolution of multicore SAT solvers over the last years.

Keywords Boolean satisfiability (SAT) · Parallel search

1 Introduction

In recent years, Boolean satisfiability (SAT) solvers have been successfully used in many practical applications, including planning, hardware and software verification, general theorem proving, and computational biology. The widespread use of SAT is the result of SAT solvers being so effective in practice. Even though real world problems tend to be large in size, SAT solvers are now able to solve many problems

R. Martins (✉) · V. Manquinho · I. Lynce
IST/INESC-ID, Technical University of Lisbon, Lisbon, Portugal
e-mail: ruben@sat.inesc-id.pt

V. Manquinho
e-mail: vmm@sat.inesc-id.pt

I. Lynce
e-mail: ines@sat.inesc-id.pt

with hundreds of thousands of variables and millions of clauses in a just a few minutes. However, there are still many problems that remain challenging for modern SAT solvers. For example, many industrial instances from the last competition remain unsolved. Even though SAT solvers are still improving, the gains given by small algorithmic adjustments are scarce.

Nowadays, multicore processors are becoming prevalent. For SAT solvers to improve, it is essential to exploit this new architecture. In the last years, there has been an increase in the research of parallel SAT solving, particularly for shared memory architectures. Due to the increasing popularity of this topic, it is important to give an overview of what has been done so far. This work closes that gap by presenting an overview of parallel SAT solving.

Similar work was presented by Singer [93]. However, at that time there was almost no research done in parallel SAT solving for shared memory architectures. Therefore, the focus of that review is mostly on parallel SAT solving for distributed architectures. Moreover, in the last years different approaches have been proposed and parallel SAT solving for shared memory architectures is now an active research topic. The overview presented here is expected to be a valuable document for researchers in parallel SAT solving, in particular for shared memory architectures.

This work is organized as follows: the next section introduces some background notions of Boolean satisfiability as well as the framework of sequential modern SAT solvers. Section 3 describes the main approaches to parallel SAT solving, namely, search space splitting and portfolios. Other alternative approaches are also described, such as collaborative solving, problem splitting and hybrid approaches. Section 4 presents various clause sharing selection strategies, as well as the integration procedure of shared clauses into each SAT solving algorithm. Section 5 presents other relevant features of parallel SAT solvers, such as the solver architecture and parallel preprocessing. Section 6 describes parallel approaches also proposed for related areas of SAT. Finally, Section 7 concludes the paper. Appendix A presents a chronological timeline of some of the most relevant parallel SAT solvers. Additionally, Appendix B presents an evaluation of modern parallel SAT solvers for shared memory architectures. This evaluation illustrates the evolution of multicore parallel SAT solvers and allows for a better understanding of their current performance.

2 Preliminaries

Propositional formulas are represented in Conjunctive Normal Form (CNF). A CNF formula is represented using n Boolean variables $x_1, x_2, \ldots, x_n$, which can be assigned truth values 0 (false) or 1 (true). A literal l is either a variable x_i (i.e., a positive literal) or its complement $\neg x_i$ (i.e., a negative literal). A clause ω is a disjunction of literals and a CNF formula φ is a conjunction of clauses.

Example 1 Consider the following CNF formula: $\varphi = \omega_1 \wedge \omega_2 \wedge \omega_3$ where $\omega_1 = (x_1 \vee x_2 \vee \neg x_3)$, $\omega_2 = (x_2 \vee x_3)$ and $\omega_3 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$. This formula has 3 variables and 3 clauses. Clause ω_2 has only positive literals whereas clause ω_3 has only negative literals.

A literal is *satisfied* if its truth value is 1 and *unsatisfied* if its truth value is 0. A literal with no truth value is said to be *unassigned*. A clause is said to be *satisfied* if at least one of its literals is satisfied, and it is said to be *unsatisfied* if all of its literals are unsatisfied. A clause with no literals, called an *empty* clause, is *unsatisfiable*. A clause is *unit* if all literals but one are unsatisfied, and the remaining literal is unassigned. If a clause is neither satisfied, unsatisfied or unit, it is said to be *unresolved*. A formula is *satisfied* if all its clauses are satisfied by a given assignment which is said to be a solution. A formula is *unsatisfied* if at least one of its clauses is unsatisfied. A formula is *unsatisfiable* if there is no assignment that makes the formula satisfied. Given the formula in Example 1, the truth assignment $\{x_1 = 1, x_2 = 1, x_3 = 0\}$ makes the formula satisfied and the truth assignment $\{x_1 = 1, x_2 = 1, x_3 = 1\}$ makes the formula unsatisfied (ω_3 is unsatisfied). The formula $x_1 \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is unsatisfiable.

The *Boolean Satisfiability (SAT) problem* consists of deciding whether there exists a truth assignment to the variables such that the formula is satisfied.

SAT solvers can be divided into two categories: *complete* and *incomplete*. Given a CNF formula, complete SAT solvers can either find a solution or prove that no solution exists. In contrast, incomplete SAT solvers can prove either satisfiability [89] or unsatisfiability [80]. Hence, incomplete solvers have applications in problems for which is known beforehand that only satisfiability or unsatisfiability is required to be proven. However, the majority of the industrial applications use a complete SAT solver since the solver is required to be able to prove satisfiability and unsatisfiability. For this reason, this paper focus on complete SAT solvers and their parallelization.

2.1 Modern complete SAT solvers

The first algorithm for solving SAT was based on resolution and was proposed by Davis and Putnam in [22]. The original algorithm suffers from the problem of memory explosion. Therefore, Davis et al. proposed a modified version [21] that used backtrack search to limit the memory required for the solver. This algorithm is often referred to as DPLL algorithm and corresponds to a depth-first backtrack search where at each branching step a variable is chosen and assigned a truth value. Following that, the logical consequences of each branching step are evaluated. Each time an unsatisfied clause (i.e. *conflict*) is identified, *backtracking* is executed. Backtracking corresponds to undoing branching steps until a branch is found where one of the values have not been tried. If for the first branching step both values have been considered, and backtracking undoes this first branching step, then the CNF formula can be declared unsatisfiable. This kind of backtracking is called *chronological backtracking*.

Nowadays, *Conflict-Driven Clause Learning* (CDCL) SAT solvers are very effective in practice. Their organization is still primarily inspired by DPLL solvers. Moreover, it has been shown that CDCL SAT solvers can also be seen as resolution engines [47, 77].

Algorithm 1 shows the standard organization of a CDCL SAT solver first proposed by Marques-Silva and Sakallah [67]. Since the introduction of CDCL SAT solvers, several techniques have been additionally proposed, namely search restarts [6, 36, 61] and implementation of clause deletion policies [35, 67]. Search restarts cause the algorithm to restart itself, but clauses already learnt are kept. Since

Algorithm 1 CDCL SAT solver (φ)

```
1:  dl ← 0; conflicts ← 0; answer ← UNKNOWN
2:  if (UnitPropagation(φ) = CONFLICT) then
3:    answer ← UNSATISFIABLE
4:  while (answer = UNKNOWN) do
5:    dl ← 0;
6:    while (conflicts < limit ∧ answer = UNKNOWN) do
7:      if (AllVariablesAssigned(φ)) then
8:        answer ← SATISFIABLE
9:      else
10:       dl ← dl + 1
11:       AssignBranchingVariable(φ)
12:       while (UnitPropagation(φ) = CONFLICT ∧ answer = UNKNOWN) do
13:         β ← ConflictAnalysis(φ)
14:         if (β < 0) then
15:           answer ← UNSATISFIABLE
16:         else
17:           Backtrack(φ,β)
18:           dl ← β
19:           conflicts ← conflicts + 1
20:   Restart(φ, limit)
21: return answer
```

search restarts are now a part of modern CDCL SAT solvers they are also shown in Algorithm 1. Clause deletion policies are used to decide which learned clauses can be deleted. Clause deletion allows the memory usage of the SAT solver to be kept under control. For a more detailed description of CDCL SAT solvers see the work done by Marques-Silva et al. [66].

In Algorithm 1, the following auxiliary functions are used:

- UnitPropagation consists of the iterated application of the unit clause rule. Given a unit clause, the *unit clause rule* [21, 22] may be applied: the unassigned literal has to be assigned value 1 for the clause to be satisfied. If an unsatisfied clause is identified, then conflict is returned.
- AllVariablesAssigned tests whether all variables have been assigned, in which case the algorithm terminates indicating that the CNF formula is satisfiable. This is done implicitly in modern SAT solvers by checking if there are no more variables to be assigned.
- AssignBranchingVariable consists of choosing a variable to assign and its respective value. Different heuristics have been explored in the past, including the VSIDS (Variable State Independent Decaying Sum) heuristic [70] which is considered to be one of the most effective. This heuristic associates an activity counter with each literal. Whenever a learned clause is created after a conflict, each literal that belongs to that clause has its activity increased by a predefined value. Periodically, all activity counters are divided by some experimentally

tuned number. When AssignBranchingVariable is called, the highest-value unassigned literal is chosen.

- ConflictAnalysis consists of analysing the most recent conflict and learning a new clause from the conflict. This is done by analysing the structure of unit propagation and deciding which literals to include in the learned clause [7, 67, 70, 111]. If the conflict does not depend on any variable assignment, backtrack cannot be performed and -1 is returned. Otherwise, it returns the decision level to which the solver can safely backtrack.
- Backtrack backtracks the search to the decision level computed by ConflictAnalysis. Backtracking can be *non-chronological*. While chronological backtracking always backtracks to the last decision level where one of the values has not been tried, non-chronological backtracking can perform larger backtracks to smaller decision levels. A decision level of a variable denotes the depth of the decision tree at which the variable is assigned a value. While backtracking, all variables at decision levels higher than the one computed by ConflictAnalysis become unassigned.
- Restart consists of restarting the search by unassigning all variables except the ones that have been assigned at decision level 0. Variables that are assigned at decision level 0 correspond to unit clauses and implications forced by unit clauses. After restarting, the cutoff for the next restart is set (*limit*). Rapid randomized restarts are used in order to eliminate the heavy-tail behaviour [6, 36, 61].

Algorithm 1 illustrates how modern SAT solvers work. The algorithm receives a propositional formula φ and applies unit propagation (line 2). If a conflict is found, then the formula is trivially unsatisfiable. Otherwise, the search process begins. The search is done until all variables are assigned a value or until unsatisfiability is proven. At each step, the solver increases the decision level (*dl*) (line 10). Next, it chooses a variable and its respective value (line 11) and applies unit propagation (line 12). If no conflict is found, the three steps shown in lines 10 to 12 are repeated. On the other hand, if a conflict is found, conflict analysis is performed in order to learn a clause that will prevent similar conflicts from occurring in the future, and to determine the decision level to which the solver can safely backtrack to (β in line 13). Notice that if β is less than zero, then the conflict does not depend on any variable assignment. As a result, backtracking cannot be performed and the formula is deemed unsatisfiable. Otherwise, backtracking is performed to decision level β and all assignments made at decision levels higher than β are undone (line 17). The decision level is updated to the current backtrack level (line 18) and the number of conflicts is increased by 1 (line 19). The solver returns to line 12 and repeats this process until unsatisfiability has been proven or no more conflicts are found. If the number of conflicts is greater than a given cutoff, the search is restarted (line 20). When restarting, all variables with decision level higher than 0 are unassigned, the cutoff for the next restart is increased and the decision level is set to 0. After each restart the search begins as usual and it is done until a solution is found or unsatisfiability is proven. For some applications, if unsatisfiability is proven then a more detailed answer may be needed. In this case, the SAT solver can be extended to obtain an unsatisfiable subset of the initial set of clauses. These subsets are denoted by *unsatisfiable cores* and are useful to explain the reason for unsatisfiability. A detailed explanation of how these unsatisfiable cores are obtained can be found in the literature [2, 113].

2.1.1 Lazy data structures

To increase the performance of the unit propagation procedure and to reduce the number of operations during backtracking, modern SAT solvers use *lazy* data structures [70, 108, 112].

Accurate data structures are able to know exactly the value of each literal in a clause, hence knowing whether the clause is satisfied, unsatisfied, unit or unresolved. However, in practice, the SAT algorithm only needs to perform additional operations when a clause is unsatisfied or unit. If a clause is unsatisfied then the SAT algorithm performs conflict analysis and backtracks. If a clause is unit then the SAT algorithm assigns the truth value to the unassigned literal and performs unit propagation. Since satisfied clauses do not require any action from the SAT algorithm, they can be *lazily* declared satisfied, i.e. it can be the case that a clause is already satisfied but for the SAT algorithm its status is still unresolved. Note that *lazily* declaring a clause to be satisfied does not affect the correctness of the algorithm.

Modern SAT solvers use the *Watched Literals* scheme [70], where for each clause two literals are always watched. Initially, any two literals can be watched in each clause since all literals are unassigned. When assigning literals, a clause is *visited* only when one of the two watched literals are assigned truth value 0. If this is the case then an unassigned or satisfied literal will be searched for, which leads to the following three cases:

1. *A satisfied literal is found*. If this literal is already watched, then nothing needs to be done. Otherwise, the satisfied literal that has been found becomes watched;
2. *An unassigned literal is found*. If this literal is not watched, then it becomes the new watched literal. In this case the clause remains unresolved. On the other hand, if the only unassigned literal is the other watched literal, then the clause is declared unit. The identification of unit clauses is done by checking the value of all literals in the clause;
3. *No satisfied or unassigned literals are found*. Both watched literals are assigned truth value 0 and a conflict has been found. The clause is declared unsatisfied since all of its literals are assigned truth value 0.

The watched literal scheme is optimized for backtracking considering that it is not necessary to perform any changes to the watched literals while backtracking since they will continue to point towards a satisfied or unassigned literal. From the time of the proposal of the watched literal scheme, some variations have been presented to optimize the use of lazy data structures. Literal sifting [62] can be applied to dynamically rearrange the list of literals by sorting them by non-decreasing decision level. With literal sifting the detection of a unit or unsatisfied clause can be done without checking the value of all literals in the clause. Another optimization is the special handling of the most common clauses: binary and ternary [62, 75, 84]. Both these clauses can be identified as unit, satisfied or unsatisfied in constant time. Since real-world problem instances are mostly composed by binary and ternary clauses, the average CPU time required to handle these clauses can be greatly reduced. Recently, two more optimizations have been presented [8, 98]. The first optimization is to move and keep the two watched literals at the beginning of the clause [8]. This can be done by exchanging literals instead of moving the references of the watched literals. When searching for a new watched literal, this optimization allows to detect, in constant time, whether the other watched literal is already satisfied. The second optimization

is to use blocking literals [98] and aims to prevent visiting clauses that are already satisfied. For each clause, a blocking literal is recorded. When searching for a new watched literal, the blocking literal is first checked. If the blocking literal is satisfied then the clause is detected to be satisfied without reading it from memory.

3 Parallel SAT solving

The demand for more computational power led to the creation of new computer architectures composed by multiple machines connected by a network, such as clusters and grids. In the last decade, parallel SAT solving has been the target of research since using these new architectures allows solving more problem instances [12, 14, 109]. Lately, with the predominance of multicore processors, the interest in parallel SAT solving has increased and more parallel SAT solvers are emerging. In this section, we present an overview of the main approaches for parallel SAT solving that are based on CDCL solvers. There are two main approaches in parallel SAT solving: *search space splitting* and *portfolios*.

3.1 Search space splitting

Search space splitting approaches are based on dividing the search space into disjoint subspaces to be explored in parallel. The most common form of search splitting makes use of guiding paths [109].

The guiding path consists of the list of variables that have been assigned until the current state of the search process. For each variable, the guiding path stores the truth value that has been assigned to that variable and a Boolean flag that shows if both values have been tried on that variable. For each variable, the guiding path records the corresponding assigned truth value and a Boolean flag that shows if both values have been tried. A variable where both values have been tried is said to be *closed*. On the other hand, if only one of the values has been tried, the variable is said to *open*. Open variables represent disjoint subspaces that are still not explored.

Figure 1 shows an example of dividing the search space through the use of guiding paths. On the left-hand side, it is shown the current search state of process 1. At decision level 1 the variable x_1 is assigned value 0 ($x_1 = 0@1$), at decision level 2 the variable x_2 is assigned value 1 ($x_2 = 1@2$) and at decision level 3 the variable x_3 is assigned value 0 ($x_3 = 0@3$). The guiding path of processes 1 describes this sequence of assignments and is given by $GP_1 = \langle(x_1, 0, closed), (x_2, 1, closed), (x_3, 0, open)\rangle$. In the guiding path, each variable has the truth value that was assigned to it (0 or 1) and a Boolean flag that shows whether there was an attempt to assign both values (0 and 1) to that variable. For example $(x_1, 0, closed)$ shows that x_1 is assigned value 0 and both values were already tried on this variable. On the other hand, $(x_3, 0, open)$ shows that x_3 is assigned value 0 but there was still no attempt to assign value 1 to this variable.

On the right-hand side, it is shown how the open variables in the guiding path can be used to create disjoint subspaces that can be searched in parallel. For example, process 2 can start searching on the subspace specified by the guiding path $GP_2 = \langle(x_1, 0, closed), (x_2, 1, closed), (x_3, 1, closed)\rangle$. Finally, to guarantee that no other process will work on the same guiding path as GP_2, the guiding path of

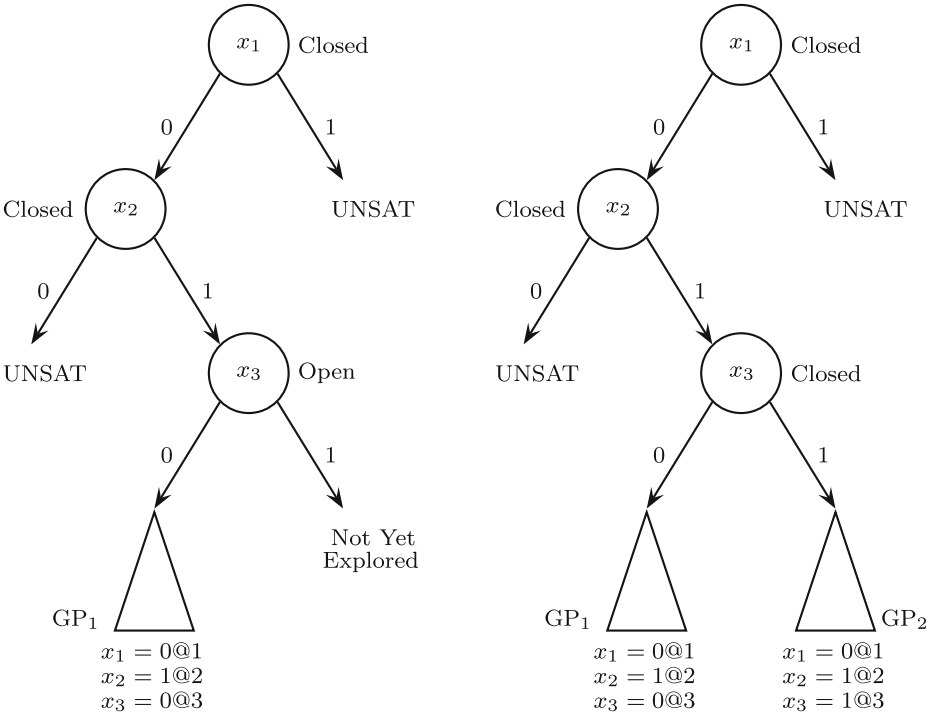

Fig. 1 Example of division of the search space through the use of guiding paths

process 1 gets modified by marking the previously open decision variable x_3 as *closed*: $GP'_1 = \langle (x_1, 0, \textit{closed}), (x_2, 1, \textit{closed}), (x_3, 0, \textit{closed}) \rangle$.

Even though guiding paths can be used to split the search space, there are some subspaces that are easier to prove (un)satisfiability than others. Since the time needed to prove (un)satisfiability on each subspace cannot be predicted, the work cannot be balanced prior to search. Therefore, dynamic work stealing procedures have to exist in order to balance the work between all processes. Without such procedure some processes may quickly become idle while others can take a long time to solve their subspace.

3.1.1 Dynamic work stealing

The most common architecture of a parallel SAT solver is based on the master-slave principle. This principle is a model for a communication protocol in which a master process controls one or more slave processes. In what follows is presented the dynamic work stealing that is implemented in most parallel SAT solvers that use the master-slave architecture [12, 19, 59, 85, 86].

The master process is responsible for maintaining a workload balance between all slave processes. If a slave process is idle, it sends a request for a new search space to the master process. The master process then selects the slave that has the shortest guiding path (with respect to the number of literals) to split its search space and provide an unexplored subspace of the search. The shortest guiding path is chosen

since it corresponds to a larger subspace of the search. Note that in practice there is no need to have a dedicated master process. Since the master process is idle during most of the time, a slave process may perform both the master and slave functions. However, for explanation purposes the remainder of the paper presents the master and slave as being independent processes.

Even though the shortest guiding path corresponds to the largest unexplored subspace of the search, it still often results in subspaces that are quickly proven to be unsatisfiable. Since the search space is often highly imbalanced, a process may require exploring several search spaces before it finds a subspace that requires a reasonably amount of work. If for each request the master has to interrupt the search of an active slave, then significant CPU time would be wasted. Also, it is possible that this process will be affected by the "ping-pong effect" [50]. Consider the case where all slaves are idle except one. The master will demand the active slave to split its current workload. However, if the subspace that is created is quickly proven to be unsatisfiable, it will return to a state where all slaves are again idle except one. This can happen several times in a row causing a high idle time and is known as the "ping-pong effect". To reduce the idle time, a central queue that contains unexplored guiding paths can be kept. Slaves with the shortest guiding path can periodically fill this queue, so that the master does not need to interrupt their search as often. When a slave is idle, it first checks if there are any guiding paths in this queue. If this is the case, then there is no need to ask for a new search space to the master. This reduces the idle time caused by work stealing and also reduces the probability of occurring the "ping-pong effect".

Figure 2 shows an example of the dynamic work stealing procedure. On the left-hand side, the search space is divided into 4 disjoint subspaces, S_1, S_2, S_3, S_4, and two processes, P_1, P_2, are currently searching the search space. The subspace S_1 was already proven to be unsatisfiable and currently P_1 is searching in the subspace S_2 and P_2 is searching in the subspace S_3. If there are only 2 processes, the subspace S_4 is not being searched by any process and therefore it is put into the work pool. Now,

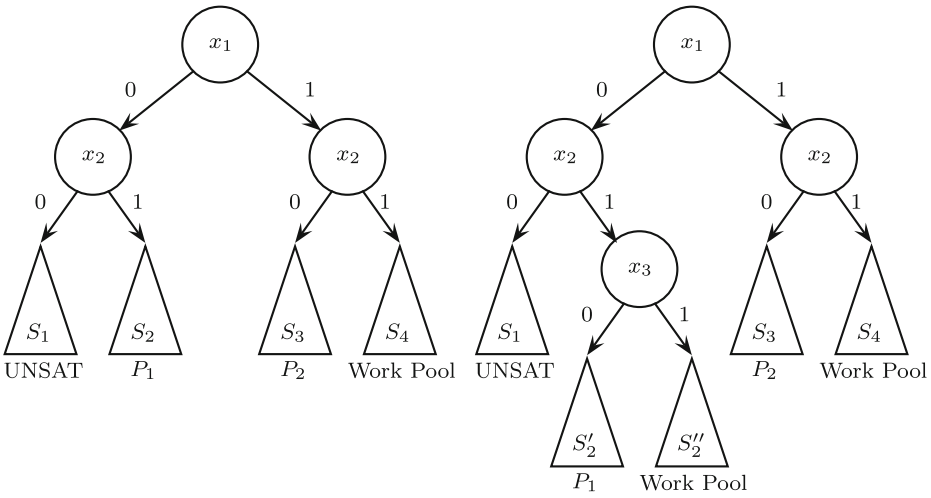

Fig. 2 Example of the dynamic work stealing procedure

consider that the master process has to maintain a work pool of size at least 2. Since there is only 1 guiding path in the work pool the master process will demand a slave process to split its subspace into two. On the right-hand side, process P_1 splits its subspace, S_2, into two disjoint subspaces, S'_2, S''_2. This is done by assigning both 0 and 1 values to a variable that did not occur in the guiding path of P_1. Afterwards, process P_1 continues the search in subspace S'_2, whereas the remaining subspace S''_2 is put on the work pool. When a process P_i proves that its current subspace is unsatisfiable, it retrieves a new subspace from the work pool. This procedure is repeated until a process proves that its subspace is satisfiable, in which case the formula is *satisfiable*, or until all subspaces have been proven to be unsatisfiable, in which case the formula is *unsatisfiable*.

3.1.2 Master-slave architecture for parallel SAT solvers

Until recently, the majority of the parallel SAT solvers used the master-slave architecture. Algorithm 2 shows the organization of the master process in the master-slave architecture, in which the following auxiliary functions are used:

- ProduceInitialGP consists of selecting the initial k partition variables and assigning values 0 and 1 to those variables in order to build 2^k guiding paths. Partition variables have their value fixed in the guiding path and are used to generate disjoint subspaces for parallel search. Each slave process will have a guiding path that corresponds to a subspace of the search. Since the number of generated guiding paths is usually larger than the number of slaves, the remaining guiding paths are put into the work pool and will be given to the slave processes upon request.
- LaunchSlaves launches the several slaves that will use a CDCL SAT solver to search in a given subspace of the search space.
- RequiredGP detects if the work pool is empty and a slave is requesting for a new guiding path.
- NoActiveProcesses checks if there are active processes. If the work pool is currently empty and there are no active processes, it means that all subspaces were proven to be unsatisfiable. Hence, the formula is unsatisfiable.
- GenerateGP orders a slave to split its current workload by splitting its search space into two. The slave will continue its search in one of the generated subspaces whereas the other is put into the work pool in the form of a guiding path.

Algorithm 2 illustrates how the master process works. It starts by initializing the shared variable *answer* (line 1) and building the initial set of guiding paths (line 2). Afterwards, it launches each slave with a guiding path so that all slaves search in disjoint subspaces of the search space (line 3). The slave algorithm is presented in Algorithm 3. While a solution is not found or all subspaces are not proven to be unsatisfiable, the master waits for a request from the slaves to create more work. If a slave is requesting work (line 5) it means that the work pool is empty. Therefore, the master must order an active slave process to create additional work. If there are no active slave processes, i.e. if all slaves are waiting for work, then it means that all subspaces were already proven to be unsatisfiable and therefore the formula is declared *unsatisfiable* (line 7). Otherwise, the master creates work by ordering an active slave process to divide its search space into two (line 9). The dynamic work

Algorithm 2 Master CDCL SAT solver (φ)

```
1: answer ← UNKNOWN
2: ProduceInitialGP()
3: LaunchSlaves()
4: while (answer == UNKNOWN) do
5:    if (RequiredGP()) then
6:       if (NoActiveProcesses()) then
7:          answer ← UNSATISFIABLE
8:       else
9:          GenerateGP()
10: return answer
```

stealing procedure guarantees that all slave processes have work, minimizing the idle time of those processes. If the current status of the formula (*answer*) is no longer unknown then the master stops its procedure and returns satisfiable (if a slave has found a solution) or unsatisfiable (if the formula does not have a solution) (line 10).

Algorithm 3 shows the organization of the slave process. The slave process follows the CDCL SAT solver algorithm presented in Algorithm 1 with additional auxiliary functions:

- GetGP retrieves a guiding path from the work pool.
- AssignBranchingVariableGP chooses as decision variable the variable that appears in the *dl* position of the guiding path. The corresponding truth value that appears in the guiding path is assigned to this variable. In practice, it is more efficient to assign all variables of the guiding path only once and not allow restarts to unassign any of these variables. For simplicity, this improvement is not included in Algorithm 3.
- ImportClauses imports learned clauses from other slaves. Sharing learned clauses can considerably speed up the search in parallel SAT solving [12, 43].
- ExportClauses exports learned clauses to be learned by other slaves.

Algorithm 3 illustrates how the slave process works. In line 1, the slave process gets a guiding path (GP_i) from the work pool. The literals in the guiding path are chosen as the first decision variables so that each slave searches in a disjoint subspace of the search space. When choosing and assigning a variable, if the decision level corresponds to a literal in the guiding path (line 13), that literal will be assigned and branched. If the forced assignment of the variable leads to a conflict, it means that the current subspace is unsatisfiable (line 15). In this case, the slave returns to line 1 and restarts the procedure by getting a new guiding path that corresponds to a new subspace of the search space. If the decision level does not correspond to a variable in the guiding path, the assignment and branching of the variables is done as usual. Local unsatisfiability is the most common case for slave processes. However, it is possible for a slave to prove global unsatisfiability. If the backtracking level (β) is less than zero (line 21), it means that the conflict does not depend on the assignments forced by the guiding path and therefore the formula is *unsatisfiable*.

Besides the use of guiding paths, other main differences regarding the sequential solver are importing and exporting learned clauses. During the search, each slave process imports learned clauses that have been sent by other slave processes (line

Algorithm 3 Slave CDCL SAT solver (φ)

```
1:  GetGP(φ, GP_i)
2:  dl ← 0; conflicts ← 0
3:  if (UnitPropagation(φ) == CONFLICT) then
4:    answer ← UNSATISFIABLE
5:  while (answer == UNKNOWN) do
6:    dl ← 0;
7:    while (conflicts < limit ∧ answer == UNKNOWN) do
8:      ImportClauses()
9:      if (AllVariablesAssigned(φ)) then
10:       answer ← SATISFIABLE
11:     else
12:       dl ← dl + 1
13:       if (dl ≤ |GP_i|) then
14:         if (AssignBranchingVariableGP(φ, GP_i, dl) == CONFLICT) then
15:           goto 1 ▷Current subspace is UNSATISFIABLE
16:       else
17:         AssignBranchingVariable(φ)
18:       while (UnitPropagation(φ) == CONFLICT ∧ answer == UNKNOWN)
          do
19:         β ← ConflictAnalysis(φ)
20:         if (β < 0) then
21:           answer ← UNSATISFIABLE
22:         else
23:           Backtrack(φ,β)
24:           dl ← β
25:           conflicts ← conflicts + 1
26:           ExportClauses(conflict)
27:   Restart(φ, limit)
```

8). While importing learned clauses, the slave process may need to backtrack to an earlier decision level. This is done to maintain the correctness of the solver. After each conflict, each slave exports the learned clause to the remaining slave processes (line 26). Since exporting all clauses would lead to an exponential blow-up, only a limited number of clauses are usually shared. Clause sharing can significantly boost the performance of parallel SAT solvers [12]. A detailed description of the sharing procedure [43, 85] as well as clause selection strategies [42, 85] are presented in Section 4.

The master-slave architecture together with a dynamic work stealing procedure allows for several slaves to do a continuous search in disjoint subspaces of the search space. The major challenges of this approach are in finding the best guiding paths and in load balancing. Since real world instances have many variables, it is hard to determine which variables should be selected to divide the search space. Moreover, since the search space is often highly imbalanced, the dynamic work stealing procedure may not prevent slaves from becoming idle during some time.

To improve the choice of the initial guiding paths, Plaza et al. [79] suggested to run the SAT algorithm during a certain amount of time prior to search in order to

retrieve information about the variables. The VSIDS heuristic can then be used to determine the most relevant set of variables to be used to divide the search space, by choosing the variables with the highest activity score as partition variables.

A generalization of this idea was proposed in [68] for a multicore architecture. Instead of running only one process to gather the VSIDS information, all the processes are run sequentially for a short number of conflicts. In order to increase the information that is gathered, each process searches a different initial search space through a different initialization of the VSIDS heuristic. This approach allows for a short period of *weak portfolio* since each process searches in a different part of the search space. The information of the VSIDS heuristic of all processes is then heuristically analyzed and the best candidates for the partition variables are chosen for building the initial guiding paths.

3.1.3 Other forms of search space splitting

Even though the use of guiding paths for search space splitting is dominant in parallel SAT solving, there exist a few other relevant forms of search space splitting. Two alternative ways of performing search space splitting is to make use of scattering functions [48] and XOR constraints [79].

3.1.4 Scattering functions

Hyvärinen et al. [48] introduced scattering functions. Given a formula φ, the scattering rule is used to divide φ into the following n disjoint subformulas that can be searched in parallel:

$$\varphi_i = \begin{cases} \varphi \wedge T_1 & , \text{if } i = 1 \\ \varphi \wedge \neg T_1 \wedge \ldots \wedge \neg T_{i-1} \wedge T_i & , \text{if } 1 < i < n \\ \varphi \wedge \neg T_1 \wedge \ldots \wedge \neg T_{n-1} & , \text{if } i = n \end{cases} \quad (1)$$

T_i is a conjunction of d_i literals, $l_1 \wedge \ldots \wedge l_{d_i}$, heuristically selected from the literals of φ. For balancing the subformulas, each d_i is determined by minimizing $\left|2^{-d_i} - (n - i + 1)^{-1}\right|$ [48].

The d_i literals are chosen as follows. Consider the following formula: $\gamma = \varphi \wedge \neg T_1 \wedge \ldots \wedge \neg T_{i-1}$. A SAT algorithm searches on γ until it reaches decision level $d_i + 1$. The $l_1, \ldots, l_{d_i}$ literals corresponds to the variables in decision levels $1, \ldots, d_i$.

Example 2 (Scattering function of a given formula φ)

$$\varphi_i = \begin{cases} \varphi \wedge x_1 \wedge \neg x_2 & , i = 1 \\ \varphi \wedge (\neg x_1 \vee x_2) \wedge \neg x_3 \wedge x_4 & , i = 2 \\ \varphi \wedge (\neg x_1 \vee x_2) \wedge (x_3 \vee \neg x_4) \wedge x_1 & , i = 3 \\ \varphi \wedge (\neg x_1 \vee x_2) \wedge (x_3 \vee \neg x_4) \wedge \neg x_1 & , i = 4 \end{cases}$$

Example 2 shows a possible scattering function that partitions the initial formula φ into 4 disjoint subformulas. For each i, the d_i literals are heuristically selected and d_i is determined by the minimization of the function mentioned above. For example, for $i = 1$ we get $d_1 = 2$ since this value minimizes the function $\left|2^{-d_1} - (4 - 1 + 1)^{-1}\right|$. The two literals are then selected by running the SAT algorithm over φ until it reaches the decision level 2. The chosen literals are the decision literals on levels 1 and 2,

which corresponds in this example to x_1 and $\neg x_2$. The remaining subformulas φ_i are obtained through the iteration of this procedure for the remaining values of i.

3.1.5 XOR partitioning

Plaza et al. [79] proposed the use of XOR constraints for partitioning the search space evenly by extending the work in solution-space reduction [101]. Adding the following XOR constraint to φ reduces the search space roughly in half:

$$\varphi_{\text{even}} = \varphi \wedge (x_1 \oplus x_2 \oplus \ldots \oplus x_j \oplus 1) \tag{2}$$

where j variables were randomly chosen from φ. The resulting formula is defined as φ_{even} denoting that an even number of variables must be assigned a truth value of 1. Similarly, φ_{odd} can be defined as:

$$\varphi_{\text{odd}} = \varphi \wedge (x_1 \oplus x_2 \oplus \ldots \oplus x_j \oplus 0) \tag{3}$$

$\{\varphi_{\text{even}}, \varphi_{\text{odd}}\}$ are disjoint subformulas when both formulas have the same set of variables. These partitions generate two subformulas that can be assigned to different processors. Moreover, it is possible to recursively divide the partitions by adding more XOR constraints. Since each XOR constraint probabilistically divides the search space into two, it implicitly tries to balance the workload between the different processors. However, in practice we cannot add large XOR constraints since they do not reduce the search space. Note that, a conflict is only detected when all x_j variables have been assigned.

It was observed that many real world instances have a small backdoor set of variables [105]. A backdoor set of a SAT instance is a set of variables that under some assignment results in a subformula that can be solved in polynomial time. For example, if after assigning these variables the remaining formula consists of only binary clauses which can be solved by a linear 2-SAT algorithm. Therefore, XOR constraints can be restricted to consider only the variables that belong to the backdoor set instead of considering all the variables of the formula. This significantly reduces the number of variables used in the XOR constraints. Since it is not always possible to compute the smallest backdoor set, the VSIDS heuristic can be used as an approximation. For a formula with n variables, Plaza et al. [79] choose a small number of variables ($\log(n)$) to produce the XOR constraints. To choose those variables, a SAT algorithm is run for a certain amount of time and the top $\log(n)$ variables given by the VSIDS heuristic are chosen to generate the XOR constraints.

3.2 Portfolios

Portfolios explore the parallelism given by different strategies on the same problem. Using complementary sequential SAT algorithms allows to perform as well as the best algorithm for that problem. Moreover, if the algorithms cooperate between themselves by exchanging learned clauses it is possible to outperform the best strategy for a given problem. The design of a portfolio parallel SAT solver is based on using different parameters for each SAT algorithm. For example, each SAT algorithm can have different restart strategies, decision heuristics, polarity strategies, and learning strategies.

ManySAT 1.0 was the first portfolio parallel SAT solver for a multicore architecture with 4 cores. Table 1 shows the different strategies implemented in ManySAT 1.0. For each core, the restart heuristic, the VSIDS heuristic, the polarity of variables and the learning scheme differ. The learning schemes used are the common Conflict Driven Clause Learning (CDCL) approach [67, 111] and an extension of CDCL by using the classical implication graph [3].

Since ManySAT 1.0, several techniques have been added to this solver in order to improve its performance. ManySAT 1.1 [42] uses dynamic heuristics that adjust the size and the quality of the shared clauses between any pair of SAT algorithms. The previous version used a static limit of size 8, i.e. all learned clauses that had 8 or less literals were shared among all SAT algorithms. However, the size of learned clauses increases over time, thus reducing the number of learned clauses that are shared among the different algorithms. Additionally, parallel SAT algorithms can search completely different viewpoints of the formula where clause sharing becomes pointless. The clause sharing heuristics used in ManySAT 1.1 are described in detail in Section 4.

ManySAT 1.5 [37] is based on ManySAT 1.0 and uses diversification and intensification strategies that allow it to outperform its original version. ManySAT 1.5 uses two masters and two slaves (one slave for each master). The masters use different algorithms to ensure diversification, whereas the slaves intensify the search done by the masters. The conflict sets of each master are given to their corresponding slave in order to intensify the search of the slave around the current search space of the master. Conflict sets are the sets of literals collected during the conflict analysis. When the master restarts, it sends to the slave an ordered sequence of conflict sets. The slave does not implement any restart strategy. However, when it obtains the sequence of conflict sets from the master, it restarts the search and enforces a sequence of decisions that resembles a guiding path. Since the sequence of conflict

Table 1 ManySAT 1.0: different strategies

	Restart	Heuristic	Polarity	Learning
Core 0	Geometric $x_1 = 100$ $x_i = 1.5 \times x_{i-1}$	VSIDS (3% rand.)	if $\#occ(l) > \#occ(\neg l)$ $l = true$ else $l = false$	CDCL (extended [3])
Core 1	Dynamic (fast) $\alpha = 1200$ $x_1 = 100, x_2 = 100$ $x_i = f(y_{i-1}, y_i), i > 2$ if $y_{i-1} < y_i$ $f(y_{i-1,y_i}) =$ $\frac{\alpha}{y_i} \times \left\lvert \cos(1 - \frac{y_{i-1}}{y_i}) \right\rvert$ else $f(y_{i-1,y_i}) =$ $\frac{\alpha}{y_i} \times \left\lvert \cos(1 - \frac{y_i}{y_{i-1}}) \right\rvert$	VSIDS (2% rand.)	Progress saving [76]	CDCL
Core 2	Arithmetic $x_1 = 16000$ $x_i = x_{i-1} + 16000$	VSIDS (2% rand.)	false	CDCL
Core 3	Luby 512 [61]	VSIDS (2% rand.)	Progress saving [76]	CDCL (extended [3])

sets may contain redundant literals (the same literal can occur several times in the sequence), the slave chooses as decision variables the next unassigned literal in the ordering. As a result, the slave intensifies the search of the master by exploring its search space in a different way. Since the master and the slave are searching on similar search spaces, clauses that have been learnt by these processes are expected to be more relevant for each other.

Even though parameter tuning is able to provide complementary algorithms, it may be the case that different sets of parameters perform similarly. If the number of processors is high, it may be difficult to find that many algorithms that provide orthogonal performance. Therefore, the main challenge for portfolio approaches is scalability, since it is difficult to ensure diversification through algorithms that complement each other. The framework used in `ManySAT 1.5` is already suited for scalability. Since intensification leads to a better quality in clause sharing, when using a large amount of cores one could restrict the clause sharing just between the master and its slave. Recently, Bordeaux et al. [15] have shown that portfolio approaches can scale in the number of instances solved while using 128 processors. To ensure diversification between the different processes, a hybrid approach between search space splitting and portfolio was used. Initially, the search space is divided into 8 disjoint subspaces by using three variables in the guiding path. Afterwards, a portfolio approach is used to solve each subspace. Even though portfolio approaches can scale in the number of solved instances, they do not scale in terms of solving time. There is a trade-off between solving more instances and using significantly more computational power. Therefore, using a large number of processors for parallel SAT solving may be only viable for solving very hard instances.

With the success of `ManySAT` and the predominance of portfolio parallel SAT solvers, new portfolio solvers have been built, namely `antom` [87], `SArTagnan` [54] and `Plingeling` [10]. Inspired by `ManySAT`, these solvers run several SAT algorithms that differ between themselves in the heuristics being used.

However, not all threads must necessarily use a CDCL SAT algorithm. For example, `SArTagnan` uses one thread to simplify the clause database by eliminating or replacing variables. The clause database contains the clauses from the original formula as well as clauses that have been learned during search. This thread also performs subsumption and backward subsumption checks [110]. Moreover, it also searches for autarkies [55, 102]. Another thread uses decision making reference points (DMRP) as proposed by Goldberg [33, 34] and similar to [53]. In DMRP, a complete assignment called a reference point is maintained. Periodically, a new reference point is computed that considers the current search space of all SAT algorithms. Each variable in the reference point has the truth value that is the predominant one between the different SAT algorithms. The clauses that are not satisfied by the current reference point are analyzed and new learned clauses may be discovered.

3.3 Other approaches

Even though search space splitting and portfolio approaches dominate parallel SAT solving, there are a few other relevant approaches, namely, collaborative solving, parallelization of unit propagation, problem splitting and hybrid approaches.

3.3.1 Collaborative solving

Collaborative solving maintains the sequential solving of the SAT algorithm while performing parallel inferences that will help the sequential solving. This was done previously by *nagging* in NAGSAT [30] and more recently through the notion of *rich and poor threads* in MTSS [23, 103].

Nagging [30] involves a master process and a set of slave processes, also known as naggers. The master performs a complete DPLL search for a given formula. When naggers are free, they request a guiding path from the master. The master briefly interrupts its own search to select a decision level at random (nagpoint) and gives to the nagger the corresponding guiding path until that decision level. To increase diversification of the search, the nagger reorders the unassigned variables and starts its own search. Even though the master and the nagger are performing search on the same space, it will be explored differently. Nagging can lead to three possible cases: (1) termination of the nagger, (2) pruning of the search of the master, and (3) solving the formula. If the master backtracks over the nagpoint it sends a termination signal to the nagger. On the other hand, if the nagger finds that the current subspace is unsatisfiable it prunes the search space of the master. Finally, if a nagger finds a solution to the formula, it reports this solution to the master.

Nagging does not require explicit load balancing, since idle naggers will always initiate requests to the master to keep themselves busy. Nagging was originally designed for grid computation and therefore the number of naggers could change during search without affecting the overall result, i.e. losing or gaining a nagger will never compromise the correctness or quality of the solution produced by the master. Nagging can also be seen as a dynamic weak portfolio approach since it uses different search strategies over dynamically created subspaces of the search space.

One of the main problems with guiding paths is that the search space can be highly unbalanced. To avoid this problem, Vander-Swalmen et al. [103] proposes the use of a guiding tree using *rich* and *poor* threads. The rich thread simulates the sequential solving of the SAT algorithm. Poor threads are tasks that provide additional information about the formula. Example of tasks associated with poor threads are: unit propagation, decision heuristic, clause learning, look-ahead, pre-processing, etc. If the poor threads do not contribute to the solving of the formula, the rich thread can be seen as a sequential SAT algorithm. Similarly to nagging, the poor threads manage their jobs autonomously, therefore inducing a natural workload balancing.

When the rich thread backtracks three cases can occur:

- *Continue*: If no information is found by the poor threads, then the rich thread continues its search as usual.
- *Prune*: If at least one poor thread has found some additional information about the formula, then the rich thread takes into account this information in order to prune its search.
- *Swap Roles*: If some additional information is under construction by a poor thread, then the rich thread swaps roles with the poor thread. This is done to prevent the rich thread from becoming idle.

MTSS is a collaborative solver that currently implements the following poor threads:

- *Open Guiding Subtree*: The poor threads assign an open variable that belongs to the guiding path of the rich thread. This results in an unexplored subspace that can be searched by the poor threads.
- *Develop Guiding Subtree*: The poor threads extend the guiding path of the rich thread by assigning a variable that does not belong to the guiding path. By branching in both phases of the variable, two disjoint subspaces are created that can be searched by the poor threads.

MTSS currently presents automatic parallelization [23], i.e. any sequential SAT solver can be automatically parallelized using MTSS. An external solver can be used in MTSS to perform the work done by the poor and rich threads. Additionally, learned clauses by the external solver can be shared through MTSS to the different processes.

Parallelization of unit propagation Other approaches consist of parallelization of procedures that are called many times by the solver, for example constraint propagation. However, it has been shown that the problem of achieving local consistency belongs to the class of inherently sequential problems called log-space complete for P (P-complete) [52].

Example 3 (Worst case of unit propagation) Consider the following CNF formula:

$$\varphi = (\neg x_1) \wedge (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3 \vee x_4) \ldots$$

Applying unit propagation to φ results in the following chain of successive (sequential) and unique implications:

$$x_1 = 0 \Rightarrow x_2 = 1 \Rightarrow x_3 = 1 \Rightarrow x_4 = 1 \Rightarrow \ldots$$

Hence, in the worst case, unit propagation is inherently sequential. Theoretical results are not encouraging and that may explain why unit propagation is usually not parallelized in parallel SAT solvers. However, recently there has been an attempt to parallelize unit propagation [64]. The clause database is divided into disjoint partitions and distributed between the different threads. Each thread runs unit propagation on its private clauses and then waits for the remaining threads to finish their own unit propagation. Implied variables that were found during propagation are then sent to the remaining threads. Next, each thread propagates the implied variables that were found by all threads. This procedure is done until a fix point is reached. Currently, no dynamic work stealing procedure is being used. Therefore, one of the challenges of this approach is to partition the clauses between the different threads in such a way that the work is evenly balanced. Experimental results show that parallelizing unit propagation only leads to small speedups [64].

Problem splitting Another approach for parallel search is to split the original formula into several subformulas that can be solved individually by each process [44, 94]. For example, Singer and Monnet [94] proposed the *Joining and model Checking* scheme (JaCk-SAT). The main goal of this method is to divide the variable set V (with $|V| = n$) into two subsets V_1 and V_2 of similar size. Let $\varphi = (V, C)$ be a

CNF formula where V and C are the variable and the clause sets, respectively. The decomposition step creates two subformulas $\varphi_1 = (V_1, C_1)$ and $\varphi_2 = (V_2, C_2)$ and a residual clause set C_3. Each clause in the subsets C_1 and C_2 is composed of variables belonging only to the subsets V_1 and V_2, respectively. If a clause has variables from V_1 and V_2 it will belong to the residual clause set C_3. The *joint variable set* is denoted by $V_{js} = V_1 \cap V_2$, whereas S, S_1 and S_2 represents the possible set of solutions of φ, φ_1 and φ_2, respectively.

Algorithm 4 shows the general idea of the `JaCk-SAT` algorithm. It starts by splitting the initial formula (φ) into two disjoint subformulas, φ_1 and φ_2 (line 1). The joint variable set (V_{js}) is composed of the variables of the two disjoint subformulas, whereas the residual clause set C_3 is composed of the clauses that have variables in φ_1 and φ_2. Next, it searches independently for all solutions of the disjoint subformulas (line 2). To speed up this procedure, the computation of all solutions is done in parallel. Following, the solutions S_1 and S_2 are joined into a global solution S (line 3). Finally, this global solution is checked for satisfiability with the residual clause set C_3 (line 4).

Algorithm 4 `JaCk-SAT`(P)

1: Decompose(φ, φ_1, φ_2, V_{js}, C_3)
2: SearchAllSol//(φ_1, φ_2, S_1, S_2)
3: Join(S_1, S_2, V_{js}, S)
4: CheckSat(S, C_3)

Although this is a novel approach it has some major drawbacks. First, the search for all solutions of a formula with $\frac{n}{2}$ variables compared to the search for one solution of a formula with n variables. Second, the *Join and Check* steps may be exponential in the number of variables.

Hybrid approaches Another idea is to use a hybrid approach between search space splitting and portfolio. Adaptive competition [11] follows this idea by initially using search space splitting, changing into a portfolio approach when a particularly hard region of the search space is found. A transition heuristic based on the number of conflicts is used to change between the two approaches. When the transition is initiated, the respective subspace is additionally treated using different search heuristics. Moreover, further search splitting of the subspace is disabled.

Search space splitting can lead to *bogus* or *oblique* splits. Bogus splits occur when the resulting subspaces are together significantly larger than the original search space. For instance, when the partition variables do not change the satisfiability of the subspaces. On the other hand, oblique splits occur when the resulting subspaces are small. To reduce the impact of bad splits, Schulz and Blochinger [88] propose a heuristic that tracks the quality of the split operations performed during the solving process. The notion of quality is based on balancedness of the runtime of the generated subspaces. In this approach, the solving process starts with search space splitting. When the quality of splits becomes low, further decomposition may be less beneficial. Therefore, further decomposition is disabled and the solving process changes to a portfolio approach.

Recently, Martins et al. [68] proposed `SAT4J//`, a hybrid solver for a multicore architecture that begins with an initial stage of search space spitting and switches

to a portfolio approach when load balancing becomes an issue or when a cutoff in the number of conflicts is reached. The transaction heuristic changes from search space splitting into a portfolio mode when one of the following conditions is reached:

1. If a given process is searching for more than k conflicts and has created work that occupies the majority of the work pool, it means that this process is currently searching in a hard subspace of the search tree and the work pool is being filled with guiding paths corresponding to this area. Since this can lead to load balancing issues, the parallel search is changed from search space splitting to a portfolio approach.
2. Alternatively, it could be the case that most processes are searching in hard subspaces of the search tree, i.e. the solving process of most subproblems is not making sufficient progress. In practice, a cutoff is used to prevent reaching hard subspaces. This happens when a given process is searching for more than z conflicts ($z \gg k$) and the previous condition was not verified. Even though it is not the case that a subspace of a process is dominating the others, the diversification of the search through the use of a portfolio approach leads to better results.

When changing from search space splitting to a portfolio mode, the existing guiding paths are merged. The merge operation builds a global guiding path with the literals that were common to all guiding paths, i.e. literals that were already found to be necessary. Clauses learnt previously by each process are also kept when changing to a portfolio mode.

Alternatively, Sripriya et al. [99] propose to combine DPLL with Stalmarck's algorithm [91]. While DPLL is based on a depth-first approach, Stalmarck's algorithm can be interpreted as a breath-first approach. This hybrid approach uses the DPLL algorithm as the core which is enhanced by the information provided by Stalmarck's algorithm. Since both algorithms run in parallel, the DPLL algorithm can continuously use the information provided by Stalmarck's algorithm to prune the search space.

4 Clause sharing

Sequential SAT algorithms learn a new clause every time a conflict occurs. Learned clauses prevent similar conflicts from occurring in the future. However, in a parallel SAT solver, each process generates its own conflict clauses. Sharing learned clauses between the different processes may be helpful since it can cut off parts of the search space that were yet to be explored. Whenever a process finds that a given subspace does not have a solution, it will generate a conflict clause. Sharing that conflict clause allows other processes to terminate their evaluation of that subspace, thus reducing their search space.

Since sharing all conflict clauses between processes can result in an exponential blow-up, usually parallel SAT solvers only share clauses that have less than a limited number of literals. For example, `PaSAT` [95] only shares clauses that have 5 or less literals. Additionally, other restrictions can also be considered. In `PaMira` [85],

several configurations for clause sharing were tested. The clause selection strategies were based on the following principles:

- Number of literals;
- Percentage of unassigments done by the backtracking procedure;
- Number of literals that belong to the guiding path.

From the evaluated configurations, we highlight the following two: (1) sharing learned clauses with size 15 or less and that contain at least one literal of the guiding path, and (2) sharing learned clauses with size 5 or less and that will make backtracking unassign at least 50% of all current assignments. The first strategy is the one that showed the best overall speedup. However, only instances that were solved by the sequential solver were considered when computing the overall speedup. In fact, the second strategy is the one that solved the most instances. This shows that fine tuning clause sharing is a demanding task and different strategies can both lead to different improvements, i.e. better speedup or more robustness.

`MiraXT` [59] is a multicore parallel SAT solver that uses a shared memory clause database. Each process has access to all learned clauses. This contrasts with other parallel solvers, since each process usually has its own clause database for their conflict clauses. Therefore, in other solvers, each process shares a limited number of clauses using some clause selection strategies (e.g. small learned clauses). However, most clause selection strategies have a serious drawback since each process current search is not taken into consideration when sharing learned clauses. Hence, useful clauses may not be sent to other processes whereas many irrelevant clauses may be sent. `MiraXT` is the only parallel solver that has a shared memory clause database.

Recently, Chu and Stuckey [19] extended the idea of using the guiding path to export learned clauses. A learned clause may be useful for some processes, while at the same time being irrelevant for others. Since each process has its own guiding path, each process may be searching on completely different areas of the search space. Hence, learned clauses from processes that are searching on similar search spaces may be more relevant than others. For example, suppose we have three processes, P_1, P_2 and P_3. Consider that P_1 and P_2 have many common literals in their guiding paths, whereas P_3 has a completely different guiding path. Hence, sharing learned clauses between P_1 and P_2 is expected to be more useful than sharing learned clauses between these processes and P_3. Given a guiding path, a clause effective length is denoted by the number of literals in the clause that do not belong to the guiding path. A clause effective length gives a better measure of the clause usefulness for that process. This idea is implemented in `PMiniSAT`, where short clauses in the context of another process guiding path are shared with that process. Note that these clauses are not necessarily short in general. As a result, processes working on similar parts of the search space are more likely to share more clauses with each other than with processes that are searching on different parts of the search space.

Clause size is not the only measure that can be used to share clauses. For example, clause sharing can be done based on clause activity [79] or by using the *Literal Block Distance* [4] of the learned clauses [54].

More recently, Hamadi et al. [42] observed that the size of learned clauses tends to increase over time. If a static limit is used to export learned clauses then the sharing of learned clauses may decrease during the search until it completely stops.

Therefore, if one wants to guarantee that learned clauses will always be shared, it is necessary to use a dynamic clause selection strategy. Hamadi et al. describe two clause selection strategies based on the throughput of shared clauses and on the throughput and quality of shared clauses. These strategies are based on the Additive Increase/Multiplicative Decrease (AIMD) feedback algorithm. The AIMD algorithm is successfully used in TCP congestion avoidance [49] to estimate the available communication bandwidth. If no package loss is observed, the algorithm slowly increases the communication rate. On the other hand, if a packet loss is found then the algorithm quickly decreases the communication rate. The evolution of the communication rate w is defined as follows (a and b are fined tuned parameters):

- $w = w - a \times w$, if loss is detected;
- $w = w + \frac{b}{w}$, otherwise.

In the context of clause sharing, the AIMD algorithm can control the number of shared clauses. When the number of exported learned clauses is small, the AIMD slowly increases the size of exchanged learned clauses in order to increase the throughput of shared clauses. On the other hand, in case of a large throughput, the AIMD algorithm can also help to quickly reduce the amount of shared clauses. In this case the AIMD algorithm reduces the size of exchanged learned clauses.

The first clause selection strategy proposed by Hamadi et al. [42] controls the throughput of shared clauses. Initially, each pair of processes ($P_i \rightarrow P_j$) exchanges learned clauses that have 8 or less literals. Periodically, each process P_j checks if the number of imported learned clauses from P_i is below or above a predefined value. If it is below (above) a predefined value then it uses the AIMD algorithm to increase (decrease) the throughput of exported learned clauses. This is done by increasing (decreasing) the size of learned clauses that are exported from P_i to P_j.

The second clause selection strategy [42] is an improvement to the previous strategy since it also considers the quality of imported learned clauses. The VSIDS heuristic is used to measure the quality of imported learned clauses. Variables with high activity represent those that are being used in the current search space. Hence, if an imported learned clause has many high activity variables it is expected to be more useful for the importing process, since it is related to its current search space. With this information, the AIMD algorithm can be adjusted to increase (decrease) the throughput between processes that are searching in similar (different) search spaces.

4.1 Integrating imported learned clauses

When importing learned clauses, these have to be integrated into the individual clause database of each process. The integration procedure has been previously described in `PaMira` [85] and `ManySAT` [43]. `PaMira` is a distributed parallel SAT solver that is based on search space splitting through the use of guiding paths, whereas `ManySAT` is a multicore parallel SAT solver based on a portfolio of SAT algorithms.

In `PaMira` [85], each imported learned clause ω is processed individually. Depending on the status of ω, the integration is done as follows:

1. ω is a unit clause. A restart is forced and the clause is satisfied by assigning the corresponding literal.

2. ω is unsatisfied in the current context. The process backtracks to the highest decision level of the variables in the clause. Conflict analysis is performed to allow further backtracking. Moreover, during the conflict analysis procedure a new clause is learned. If the imported learned clause is unsatisfied because of variable assignments belonging to the guiding path, the search process is stopped since the current search space is unsatisfiable. This is the most interesting case since it reduces the search space for the current process.
3. ω is unit in the current context. The process backtracks to the highest decision level of the assigned variables in the clause. After backtracking, the unassigned literal is assigned and propagated.

In the remaining cases no special operation are needed. After the integration of all imported learned clauses each process continues its search as usual.

`ManySAT` [43] describes the integration of imported learned clauses taking into consideration the watched literals scheme. If the imported learned clause is unsatisfied or unit, `ManySAT` proceeds similarly to `PaMira`. However, since in `ManySAT` each process does not have a guiding path, if the clause is unsatisfied the process will just perform conflict analysis and the corresponding backtrack. Additionally, `ManySAT` describes two cases where the imported learned clause ω has to be watched in order to be exploited in the future:

1. ω is satisfied in the current context. If exactly one literal in the clause 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 highest decision level among the falsified literals. If this case occurs, the clause should have been unit at an earlier decision level. In all cases, a satisfied literal is watched along with another literal.
2. ω is unresolved in the current context. The first two unassigned literals are watched.

The following example illustrates what happens to the watched literals in the case of an unsatisfied, unit, satisfied or unresolved imported learned clause.

Example 4 (Integration of learned clauses when considering the current search context) Consider a CNF formula φ and a partial assignment $\delta = \{x_1 = 1@1, x_3 = 1@1, x_2 = 0@2, x_5 = 1@2, x_4 = 0@3\}$. To make an imported learned clause ω_i exploitable in the future, ω_i must be watched accordingly to its status. Suppose that,

- $\omega_1 = (\neg x_1 \vee x_2 \vee x_4)$. The clause ω_1 is *unsatisfied* in the current context. The process backtracks to decision level 3 and literals x_2 and x_4 are watched.
- $\omega_2 = (\neg x_1 \vee x_2 \vee x_7)$. The clause ω_2 is *unit* in the current context. The process backtracks to decision level 2 and literals x_2 and x_7 are watched.
- $\omega_3 = (\neg x_1 \vee \neg x_2 \vee \neg x_4)$. The clause ω_3 is *satisfied* in the current context. A satisfied literal is watched, $\neg x_4$, along with another literal, for example $\neg x_2$.
- $\omega_4 = (\neg x_1 \vee \neg x_8 \vee x_6)$. The clause ω_4 is *unresolved* in the current context. Since any two unassigned literals can be watched, the first two unassigned literals are watched. In this case, these literals are $\neg x_8$ and x_6.

5 Other features of parallel SAT solvers

Parallel SAT solvers differ between themselves mainly in the parallel approach and in the clause sharing strategies. Even though these features are the most important for parallel SAT solvers, there are some other relevant features, namely *solver architecture* and *preprocessing*. The solver architecture is usually strongly connected with clause sharing since its implementation allows different kinds of clause sharing strategies. Preprocessing can substantially decrease the runtime of sequential SAT solvers. Hence, it should also be an important part of a parallel SAT solver.

5.1 Solver architecture

The solver architecture is strongly connected with the clause sharing mechanism. Clause sharing can significantly improve the performance of parallel SAT solving. However, it is usually also responsible for most of the communication. There is a trade-off between the advantage of clause sharing in pruning the search space and the communication overhead in sending learned clauses to all processes.

In a distributed environment, clause sharing can be supported through message passing, typically using the *Message Passing Interface* standard [96]. This is the case of several parallel SAT solvers [16, 50, 95]. If all learned clauses are shared, then there would be a significant overhead in communication. To handle this problem, only small learned clauses are usually exchanged. Additionally, clauses are not sent one by one but instead they are sent in groups of clauses. This reduces the communication overhead since less messages are exchanged between the processes. However, message passing is slow when compared to a shared memory system. In a shared memory environment, clauses can be shared directly through the shared memory [29, 43, 59]. Even though clause sharing can be done without the overhead of message passing, memory contention can have a detrimental effect on the overall performance of the parallel solver.

The most common architecture for parallel SAT solvers is to use a *Conflict Clause Buffer* to share clauses. Each process occasionally sends *well suited* learned clauses to the conflict clause buffer using some clause selection strategy, such as the clause length. At the same time, it also checks if any new learned clauses have been added to the conflict clause buffer by other processes. This architecture is described in Fig. 3. In search space splitting, a master process is usually in charge of controlling the available tasks and distributing work among all processes. At the same time, it also controls a shared clause database which has learned clauses that will be shared among all processes. In this architecture, each process maintains its own private clause database, containing all clauses of the original CNF formula, as well as learned clauses deduced by itself or copied from the *Conflict Clause Buffer*. Portfolio parallel solvers also use an architecture similar to the one described in Fig. 3, the main difference being that no available tasks need to be given to the different processes.

Contrary to distributed systems, in a shared memory architecture it is possible to have more alternatives since the clause database can be fully or partially shared among all processes. One alternative is to share the original CNF formula while storing all learned clauses in private databases for each process. In this architecture, if the original CNF formula is simplified, then all processes can benefit from those

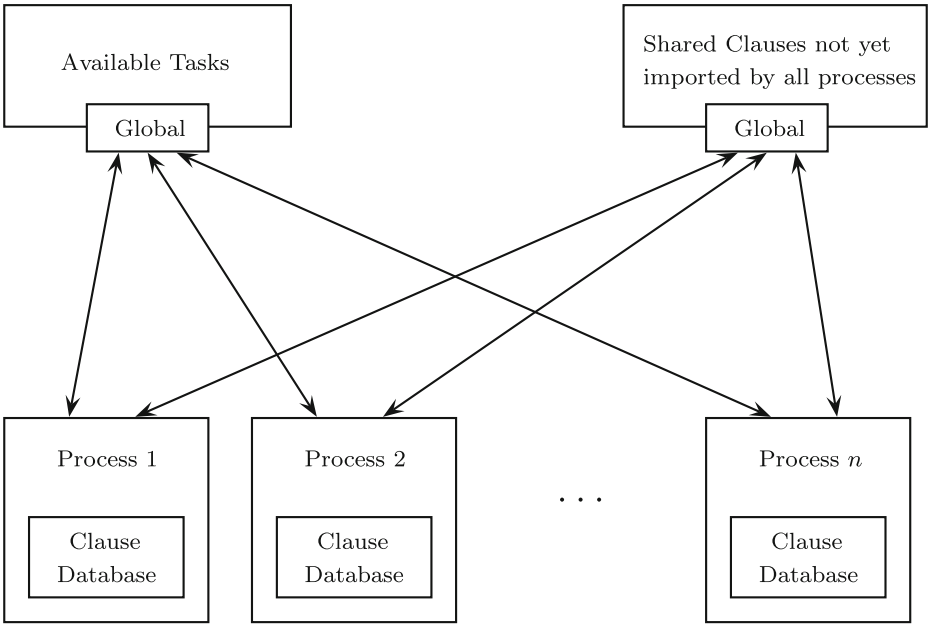

Fig. 3 High-level architecture of most parallel SAT solvers

simplifications. For example, if one process detects a clause for "on-the-fly improvement" [46], then all processes may profit from this immediately. This architecture has been implemented in the past in `ySAT` [29] and more recently in `SArTagnan` [54].

Alternatively, Lewis et al. [59] proposed a shared memory clause design that is implemented in `MiraXT`. Compared to the previous alternative, this approach goes one step further by sharing the original CNF formula and all learned clauses from all processes.

Even though all learned clauses are available, only some of them are selected to be used by each process. This design allows for each process to consider their current search space when deciding which learned clauses will be used. Additionally, this design also reduces the amount of memory used by the solver. On the other hand, sharing the entire database requires significant changes to the data structures.

5.1.1 Shared clause database

In a shared clause database, each clause only appears once in memory. After a clause is inserted into the clause database it becomes read-only. To control which clauses each process has already looked at, each process has a pointer to the last clause read by it. These pointers have to be regularly updated to include the new learned clauses that are regularly added to the shared clause database.

Since clauses are read-only, the unit propagation mechanism cannot be implemented using literal sifting. Hence, the watched literal scheme is implemented without literal sifting and with a few modifications to decrease cache misses. An additional data structure denoted by *Watched Literals Reference List* (WLRL) is used

by each process. This data structure handles the watched literals of each process. The WLRL stores the two watched literals for each clause, a pointer to the clause and up to two additional literals, denominated *Cache Literals* (CLs). When the watched literals need to be updated, the CLs are first checked as possible candidates. If the CLs cannot be used as new watched literals, then the process access the shared clause database to search for a new watched literal among the remaining literals of that clause. Clauses that have 4 or less literals are always stored inside the WLRL and therefore do not require access to the shared clause database. Since most industrial instances have many short clauses, this structure reduces the number of calls to the shared clause database.

The conflict analysis procedure of each process is also slightly changed. As usual, after the conflict analysis, the learned clause is added to the shared clause database. Next, a *clause addition procedure* is run. This procedure guarantees that all processes will have access to this new learned clause. Moreover, it also decides which clauses should be considered by each process. In `MiraXT`, each process adds to its WLRL the following learned clauses: (1) learned clauses that lead to a conflict, (2) learned clauses that force variable implications and (3) learned clauses that have 10 or less literals.

Clause deletion is also an important issue. To facilitate clause deletion in a multicore environment, each process has one Boolean flag associated with each clause. When a process removes its reference to a clause in the database, it sets the Boolean flag associated with that clause to true. After a process has removed all the references to the clauses it wants to delete, it requests for a physical removal of the clauses. A *master delete* can only be run if there exist clauses that are no longer referenced by any of the processes. If a clause is not referenced by any process, then it can be deleted from the shared clause database.

5.2 Preprocessing

Nowadays, it is acknowledged by the SAT community that using preprocessors prior to solving can substantially decrease the runtime of SAT solvers. This is due to the development of several preprocessors for which empirical evidence shows its importance in practice (e.g. [5, 25, 74]). On the other hand, parallel preprocessing is still taking its first steps. In fact, most parallel SAT solvers usually run a sequential preprocessor (e.g. [25]) prior to parallel solving.

However, additionally preprocessing can be used in parallel. Preprocessing based in unit propagation look ahead is implemented in `MiraXT` [59]. This preprocessing scheme can eliminate variables that have the same truth value in all possible solutions. This is done by assigning a variable both truth values and then using unit propagation. If both assignments lead to the implication of the same variable, then it means that such variable must have the implied truth value in all possible solutions. Since this solver is based on search space splitting, this preprocessing technique allows the elimination of variables that if selected as partition variables would lead to redundant search.

Interleaving search and preprocessing has shown to lead to good results in sequential SAT solving [9]. `Plingeling` [10] is a parallel version of the sequential

solver Lingeling [10] that uses this principle. In Plingeling each process uses different preprocessing techniques, which leads to different knowledge that can be shared among the different processes.

Alternatively, the preprocessing algorithm can be the target of parallelization. Plaza et al. [79] proposed a parallel preprocessor based on Recursive Learning (RL) [56]. This preprocessor tries to identify necessary assignments, i.e. assignments that must be contained in any truth assignment of the formula. Recursive learning can generate learned clauses that translate these necessary assignments. Even though the RL algorithm is exponential, it can be parallelized by splitting the formula into several disjoint subformulas that can then be given to different processors. Because of this behaviour, the speedup achieved by parallelizing RL is nearly linear in the number of processors involved. The level of recursion determines the complexity of the RL algorithm. For example, 1-RL finds clauses of size 1, i.e. literals that are necessary assignments. In general, k-RL finds learned clauses with k literals. Due to complexity issues the preprocessor proposed by Plaza et al. [79] only learns clauses associated with the necessary assignments of 1-RL and 2-RL.

6 Related work

The focus of this paper is the overview of complete methods for parallel SAT. Nevertheless, there has also been some work done in incomplete methods for parallel SAT [1, 73, 83]. In addition, parallel approaches have been the target of research in extensions of SAT, namely, in Quantified Boolean Formulas (QBF) [28, 58, 60] and in Satisfiability Modulo Theories (SMT) [51, 106]. Furthermore, there is also a considerable amount of work developed to solve the distributed Constraint Satisfaction Problem (CSP) [63, 107]. This section describes related work in these fields.

6.1 Incomplete methods

PGSAT [83] is a parallel version of sequential incomplete SAT solver GSAT [90]. In PGSAT the set of variables is randomly divided into n subsets that are distributed to different processors. At each iteration, if no global solution has been obtained, then each subset flips one variable using the GSAT heuristic. As a result, after an iteration, n variables have been flipped.

A different approach is to execute n independent copies of a sequential incomplete SAT solver. The multithreaded version of gNovelty+(v.2) [73] uses this approach by running n threads of gNovelty+(v.2) until at least one of them finds a solution or a given timeout is reached. This approach can be seen as a portfolio approach, since each algorithm starts from a different initial point. Recently, Arbelaez and Hamadi [1] proposed to improve this approach by sharing knowledge between the different threads. Each algorithm shares with the others the best configuration it has found so far with the respective cost (number of unsatisfied clauses). At each restart point, this knowledge can be exploited by each algorithm to carefully build a new starting point.

6.2 Quantified Boolean formulas

The quantified Boolean formula (QBF) validity problem is a generalization of the SAT problem. The QBF problem is to decide the satisfiability of a given propositional formula, in which the variables may be either universally or existentially quantified.

Proposed parallel QBF solvers [28, 58, 60] use search space splitting to divide the search space into disjoint subspaces. Each subspace is then searched in parallel by sequential QBF solvers.

`PQsolve` [28] uses a master-slave model. When a process is idle, it will request work from an active process and will become its slave. This dynamic work stealing allows slaves to become masters of other processes.

The communication between the different processes is done by the Message Passing Interface (MPI). Additionally, different scheduling methods are included in `PQsolve` to handle several problems that appear when searching in parallel, namely: (1) selection of subproblems, (2) reducing idle time and (3) reducing redundant work [27].

`QMiraXT` [60] is a QBF parallel solver designed for shared memory architectures based on the sequential SAT solver `MiraXT` [59]. There is no master process in `QMiraXT`, but instead a Master Control Object (MCO). Threads communicate between themselves using the MCO. The MCO is also responsible for splitting the search through the Single Quantification Level Scheduling (SQLS) algorithm [60] and distributing the work among the different processes. The SQLS algorithm allows for threads to split their search space. However, this can only be done if all splitting variables are on the same quantification level.

`PaQuBE` [58] is a parallel QBF solver based on MPI that uses the master-slave model. However, the master is only in charge of controlling the requests for new subproblems, ensuring that the entire search space is searched. The search space splitting procedure is done through the SQLS algorithm. The slave processes are in charge of sharing the knowledge between themselves. Knowledge sharing is usually restricted to learned clauses, but in `PaQuBE` the solution cubes are also shared among the different processes. A solution cube is a set of literals that satisfy all clauses.

Different sharing strategies have been proposed for `PaQuBE` [65]. Additionally, a solution cube compression is also proposed in order to reduce the overhead of the message passing mechanism. A bucket sort system was used in `PaQuBE` for knowledge sharing. For example, consider that three buckets are used for learned clauses and three buckets for solution cubes. If the clause size is used as the sharing criterion, then the first bucket will have all clauses with 5 or less literals. The second bucket will contain all clauses with 5 to 10 literals, and the third bucket will contain all clauses that have more than 10 literals. When the information is shared, the clauses are taken from the first, second and third buckets until a threshold is reached. Afterwards, the buckets are emptied and the process restarts. Slaves only add clauses or cubes that are helpful in their context, i.e. clauses that lead to conflicts or implications. Short clauses are always exchanged between processes since they may be useful in the near future. Using the bucket sort system guarantees that short clauses will always be chosen over larger clauses. However, when the number of small clauses is low, it will also allow the exchange of larger clauses that may be helpful for other processes.

6.3 Satisfiability modulo theories

Satisfiability modulo theories (SMT) is a generalization of SAT by adding other theories, such as, equality reasoning, bit-vectors, quantifiers, among others. A SMT solver is a tool for deciding the satisfiability of formulas in these theories.

The parallel version of `Z3` [106] uses a portfolio approach for deciding the satisfiability of SMT formulas. The portfolio is based on the same portfolio used by `ManySAT` [43] for the first 4 threads, i.e. diversification is based on heuristics specific to the SAT solver being used. Additional threads are configured by using different combinations of settings. Moreover, users may also define their own portfolio through the command line. Knowledge exchange is also implemented in the form of learned clauses. Similarly to `ManySAT 1.0`, clauses with 8 or less literals are shared among the different threads.

`Picoso` [51] is a parallel SMT solver for the first order theory of real numbers extended with transcendental functions. It uses a search space splitting approach that divides the search space into disjoint subspaces that can be explored in parallel. It follows a master-slave model, where the master is responsible for the dynamic load balancing procedure. Additionally, the master is also responsible for exchanging the learned clauses among all threads. Clause sharing is restricted to small clauses, with 6 or less literals.

6.4 Constraint satisfaction problems

A Constraint Satisfaction Problem (CSP) is defined by a tuple (V, D, C) where V is a set of variables, D a set of domains of possible values for the variables and C a set of constraints on the values of the variables. A solution for a CSP is an assignment to the variables such that all constraints are satisfied. This formalism has been extended for distributed problems. In the distributed constraint satisfaction problem (DisCSP), a problem is distributed between autonomous agents which are cooperating to compute a global solution. There is a considerable amount of work done in DisCSPs and a detailed overview of this field is beyond the scope of this paper.

Parallel approaches are not only restricted to DisCSPs. In the context of CSP, parallel approaches have also been applied to Quantified Constraint Satisfaction Problems [104] and to Distributed Constraint Optimization Problems [63, 69]. In what follows we will only describe some of the techniques used in DisCSPs.

6.4.1 Search space splitting

DisCSPs are usually solved using a search space splitting approach. The most common form of dividing the search space is with variable decomposition. Variable decomposition involves assigning each agent one or more variables with the constraints containing those variables.

A basic method for finding a global solution uses asynchronous backtracking (ABT) [24]. The agents are prioritized into a partial ordering graph such that any two agents are connected if there is at least one constraint between them. The solving process begins with agents finding solutions to their respective problems. The local solutions are then propagated to agents with lower priorities. This propagation of

local solutions between agents proceeds until an agent is unable to find a local solution. At that point, a nogood (conflict) is discovered and a new constraint is learnt from the conflict. Backtracking is then performed and the search continues. This process continues until either a solution is found or all the different combinations of local solutions have been tried and none of them satisfies all the constraints.

Since the proposal of ABT many other approaches have been proposed for solving DisCSPs [39, 40, 45, 63, 92, 107]. For example, asynchronous aggregation search (ASS) [92] extends ABT by allowing the instantiation of the same variable by several agents. Alternatively, the solution space of the top priority agent can be split into independent subspaces. An example of this approach is given by interleaved distributed intelligent backtracking (IDIBT) [39]. IDIBT uses a constructive approach to explore the space by an accurate book-keeping of the explored states. In IDIBT, nogoods are not learned during the search. Recently, Hamadi and Riingwelski proposed to boost distributed constraint satisfaction by using a portfolio of variable orderings [45]. Additionally, the different agents exchange knowledge between themselves.

Search space splitting demands the use of a dynamic work stealing procedure. Therefore, load balancing methods for DisCSPs have also been the target of research. Different heuristics for load balancing have been evaluated [81]. Experimental results showed that different heuristics are better suited for different problems. For example, if the problem is to find all the possible solutions, then distributing the work evenly among all agents leads to better results. On the other hand, in optimization problems it may be more important to quickly find solutions and therefore an unevenly division of the search space may lead to better results.

Alternatively, given a user-defined confidence regarding the number of solutions in each subspace, a confidence-based work stealing [18] can be applied. If this information is available, then confidence-based work stealing can lead to better results.

6.4.2 Portfolios

Contrary to parallel SAT, portfolio approaches are scarce for DisCSPs. Nevertheless, there have been successful attempts of using portfolios for solving DisCSPs. Clearwater et al. [20] use a portfolio approach to solve word puzzles. The problem is encoded as DisCSPs and each agent is given a copy of the problem. Agents communicate between themselves to speedup the search. Each agent cooperates by writing and reading "hints" from a shared blackboard. As the number of agents increases, the diversity of the search also increases which allows for super linear speedups. Due to the recent success of portfolios approaches for parallel SAT, it might be of interest to revive similar approaches for solving DisCSPs.

6.4.3 Parallel consistency and propagation

There has also been some amount of work done on distributed consistency algorithms for solving CSPs [38, 71]. The variables are partitioned among the different agents, and each agent maintains arc-consistency for its set of variables. Arc-consistency is done for each agent until a fix point is reached. If a domain deletion is found then this information is sent to the remaining agents. The distributed consistency algorithm reaches a fix point when every agent has processed every

domain deletion. The main challenge for this approach is to partition the variables such that the work is evenly distributed.

More recently, there has also been an attempt to combine search space splitting with parallel consistency for multicore architectures [82]. Two type of threads are used: consistency threads and search threads. Search threads divide the search space into disjoint subspaces. For each disjoint subspace, a set of consistency threads performs parallel consistency.

For more details on parallel constraint solving we point the reader to a recently published preliminary review [31].

7 Conclusions

In this overview we presented the main approaches to parallel SAT solving, namely, search space splitting and portfolios. Even though these approaches dominate parallel SAT solving, there are other interesting approaches worth mentioning. For instance, a hybrid solution between search space splitting and portfolio can lead to interesting results. Another interesting novel approach that could be further studied is collaborative solving through the use of rich and poor threads. Here, the sequential SAT algorithm (rich thread) is helped by poor threads that provide a partial or definite information about the formula.

Clause sharing in a parallel environment can considerably improve the performance of the parallel solver. Most parallel SAT solvers exchange clauses that are smaller than a given size. However, in the last years there have been some attempts to improve the quality and throughput of shared clauses. Moreover, exploiting the knowledge on the guiding path can increase the throughput of shared clauses. It is well known that learned clauses can be very large. However, if a learned clause has several literals that belong to the guiding path of a thread, it can then be considered as short and highly relevant in that thread context. Recently, it was observed that the size of learned clauses tends to increase over time. Hence, it is necessary to use dynamic heuristics that change the maximum size of exported learned clauses during runtime. Heuristics can also be used to compute the quality of learned clauses. Learned clauses that have a higher quality are more relevant and should be shared among all threads.

Most parallel SAT solvers follow a master-slave architecture where each process stores a private local clause database. The master is responsible for distributing the work among the processes as well as sharing the learned clauses given by each process. In a shared memory architecture is possible to share the clause database. The clause database sharing can be partial: for example, each process can store a private clause database for its learned clauses while sharing the clauses of the original CNF formula. Alternatively, the entire clause database can be shared, i.e. all the clauses that belong to the original CNF formula as well as the clauses learned by all processes. This design allows each process to consider its current search space when importing learned clauses.

Nowadays, it is acknowledged by the SAT community that using preprocessors prior to solving can substantially decrease the runtime of SAT solvers. Currently, most parallel solvers apply a sequential preprocessing to the formula prior to parallel solving. So far, the only parallel preprocessor that has been proposed is based on

recursive learning. Although the recursive learning algorithm has an exponential complexity, it can be easily parallelized by dividing the formula into n pieces for n different processes. Still, parallel preprocessing is almost unexplored and should be the target of research in the future.

In the last years the number of cores and processors has been continuously increasing. As a result, the SAT community is becoming more interested in parallel approaches. Since 2008, there has been a parallel track in the annual competition for parallel SAT solvers, which is designed for shared memory architectures. Currently, portfolio approaches are dominating this track. Even though the increase in the performance is not as significant as expected, modern parallel SAT solvers can now outperform state-of-the-art sequential solvers both in solving time and in the number of solved instances.

Parallel SAT solving is still taking its first steps and there is much room for improvement. With the increase in the number of cores, new approaches will most likely arise. Ensuring diversification with portfolios for a large number of cores can be difficult, and therefore new approaches or hybridization of already known approaches should appear. Another issue to be tackled is memory contention that is common in parallel SAT solvers and can deteriorate their performance. This is a critical issue and should be attended in the development of new parallel SAT solvers that want to scale for a large number of cores.

As shown in this paper, research in parallel SAT solving increased significantly in the last years and the tendency is to continue. It is expected that extra computational power does not come from higher processor frequencies, but by having additional cores able to execute processes is parallel. As multicore architectures become predominant, it is crucial to develop new frameworks and techniques that take advantage of this new reality in order for SAT solvers to continue to evolve. The overview presented in this paper covers the main topics related to parallel SAT solving, and is expected to be a valuable document for researchers in this field.

Acknowledgements This work was partially supported by FCT under research projects BSOLO (PTDC/EIA/76572/2006), iExplain (PTDC/EIA-CCO/102077/2008), ASPEN (PTDC/EIA-CCO/110921/2009) and INESC-ID multiannual funding through the PIDDAC program funds.

Appendix A: Timeline of parallel SAT solvers

Parallel computing has become an affordable reality, forcing a shift in the programming paradigm from sequential to concurrent applications, in particular those which demand significant computational power or large search spaces. Parallel SAT solving can be done on *distributed memory architectures* or *shared memory hardware*. Among the distributed parallel SAT solvers mentioned in the literature, we enumerate in chronological order the most noticeable ones:

- Solver by Böhm et al. [14]: It is based on search space splitting by dividing the formula into disjoint subformulas. Each subformula is then given to a different processor to be solved by a sequential SAT solver. Additionally, dynamic work stealing is used to maintain a workload balance between the different processors.
- `PSATO` [109]: This solver introduces the important notion of guiding paths, which is the most common form of search space splitting for parallel SAT solving.

Guiding paths split the search space into disjoint subspaces that can be explored in parallel.

- PSatz [50]: This solver uses a master-slave architecture, where the master is responsible for distributing the workload among the several slaves. The search space is divided using a different guiding path for each slave. Moreover, a dynamic workload balancing procedure is used to balance the workload between the different slaves.
- PaSAT [12, 95]: This solver is designed for distributed architectures. It presents clause sharing between the different processors following a master-slave model, where the master is also responsible for sharing the learned clauses recorded by each slave. Since adding all learned clauses to the formula can result in an exponential blow-up, only clauses that have less than a fixed number of literals are exchanged. In practice, clause sharing is implemented by *mobile-agents*.
- NAGSAT [30]: It implements *nagging*, which involves a master and a set of slaves called *naggers*. In NAGSAT, the master runs a standard SAT algorithm with a static variable ordering. When a nagger becomes idle, it requests a *nagpoint*, which corresponds to the current state of the master. Upon receiving a nagpoint, it applies a transformation (e.g. a change in the ordering of the remaining variables), and begins its own search on the corresponding subformula. While the search of the master and naggers is semantically equivalent, they will be searched differently.
- GridSAT [16]: This solver has been developed for grid computing. The master maintains a distributed learning clauses database and manages the workload by dividing the work among the available slaves. It differs from other parallel solvers by trying to keep the execution of the solver as sequential as possible.
- ZetaSAT [13]: It is targeted for parallel SAT solving on desktop grids. This solver uses a master-slave model and it is fault tolerant, i.e. it is capable of operating in volatile environments. Due to the volatile environment, clauses are not shared among the different slaves.
- PaMira [85]: It uses a master-slave architecture with a dynamic search space partitioning through guiding paths. Dynamic work stealing and several clause selecting strategies are implemented.
- Solver by Plaza et al. [78]: It proposes the use of the VSIDS heuristic to determine the partition variables. Its learning strategy is based on clause activity instead of the traditional clause size. Additionally, it implements a parallel preprocessing based on recursive learning.
- PMSat [32]: It is based on search space splitting and it uses a master-slave architecture. Several partition heuristics can be used to divide the search space into disjoint subspaces. Learned clauses are also shared between the different processes. PMSat runs on networks of computers through a Message Passing Interface (MPI) implementation and is based on MiniSAT [26].
- JaCk-SAT [94]: It presents an approach that does not divide the search space but decomposes the formula into simpler subformulas with less variables that can be solved independently by each process. For each subformula, all solutions are computed. Next, these solutions are joined and checked to find if a global solution exists.
- PaMiraXT [86]: It uses a master-slave architecture based on message passing. For the slaves, MiraXT is used which itself is a parallel solver for shared memory

architectures. Therefore, PaMiraXT is suitable for distributed systems that have multicore processors at each node.

- c-SAT [72]: A parallelization of MiniSAT 2.0 using message passing and a master-slave architecture. The master is responsible for receiving and exporting learned clauses between the different processors. Moreover, the master also deletes redundant learned clauses and dynamically balances the workload between the different slaves. Each slave performs search using different heuristics and random number seeds. Therefore, it combines search splitting with a portfolio of algorithms for each subspace of the search tree.
- Satciety [88]: It is the first parallel solver with clause exchange for Desktop Grids. This solver takes into consideration the resource volatility and the heterogeneity of the network. Moreover, the communication of the original formula from the master to all slaves is improved by encoding the original formula to binary CNF.

Recently, the SAT community has been looking at multicore approaches where the memory is shared among the different processes. The SAT-Race 2008 was the first evaluation with a special track for parallel SAT solvers. At that time, each parallel SAT solver was allowed to use up to four cores. More recently, the SAT-Race 2010 also had a track for paralell SAT solver, where each of them was allowed to use up to eight cores. Next, we present multithreaded SAT solvers.

- ySAT [29]: It shares the original clauses from the CNF formula, while each thread maintains its own local learned clause database. It follows a master-slave model, where the master is responsible for the dynamic work stealing procedure. It was observed that sharing original clauses from the CNF formula lead to a detrimental effect on the cache performance.
- MiraXT [59]: It uses a search space splitting approach and it differs from other parallel SAT solvers by having a shared clause database. This database contains the original clauses of the formula and the clauses that were learned by each thread during the search. Currently this is the only multicore solver that shares its entire clause database.
- PMiniSat [19]: It is a multithreaded version of the sequential solver MiniSAT 2.0 that uses a search space splitting approach based on guiding paths. Clause sharing is enhanced by taking into consideration the guiding paths of the different threads. It also improves unit propagation by using cache conscious data structures [17]. To reduce the idle time, a central queue of guiding paths is used. This queue has guiding paths that correspond to the threads that are running for the longest time. When a thread is idle, it checks if the central queue has guiding paths. If this is the case then it can start working on a new subspace without having to wait for other threads to respond.
- ManySAT [43]: ManySAT 1.0 was the winner of the parallel track of the SAT-Race 2008. ManySAT uses a portfolio of algorithms based on on MiniSAT 2.0. Moreover, each thread shares learned clauses. Since the first version of ManySAT, several improvements have been proposed. ManySAT 1.1 [42] uses dynamic heuristics that adjust the size and the quality of the shared clauses

between any pair of SAT algorithms. ManySAT 1.5 [37] uses a master-slave scheme within a portfolio approach. The masters perform an original search strategy ensuring diversification, whereas the slaves intensify the search done by the masters. All versions of ManySAT use the sequential preprocessor SatELite [25] prior to parallel solving. Recently, a deterministic version of ManySAT has also been proposed [41].

- MTSS [23, 103]: The MultiThreaded SAT Solver (MTSS) uses a collaborative approach with two types of threads: rich and poor. One rich thread is used to perform a sequential search. On the other hand, a set of poor threads try to simplify the search performed by the rich thread. An external solver can be used as a rich or poor thread. MTSS allows sharing learned clauses between external solvers. With just a few minor changes, a sequential SAT solver can be easily parallelized using the MTSS solver. Currently, MTSS uses MiniSAT 2.0 as an external solver for performing the rich and poor threads. Additionally, all learned clauses with 9 or less literals are shared among all threads.
- satake [100]: It uses a master-slave model where each slave has its own clause database and it is based on MiniSAT 1.13. Short clauses are shared among the different slaves. The size of learned clauses that may be exported is determined by a dynamic heuristic. Moreover, each slave has a different decision heuristic and a different random seed. This enables a hybrid approach between a weak portfolio and search space splitting.
- antom [87]: It runs several SAT algorithms that differ between themselves in terms of decision heuristic, restart strategy, conflict clause deletion, lazy hyper binary resolution and on-the-fly propagation lookahead. Clause sharing is additionally done between the different SAT algorithms.
- SArTagnan [54]: It runs a portfolio of SAT algorithms. Similarly to ySAT it shares the original clauses from the CNF formula, while maintaining a private learned clauses database for each thread. To take advantage of this architecture not all threads run a CDCL SAT algorithm. One thread is dedicated to the simplification of the clause database by eliminating or replacing variables. Another thread uses reference points for decision making which often allows the learning of valuable new clauses. Additionally, clause sharing is done using as sharing criterion the literal block distance value [4] of learned clauses.
- Plingeling [10]: It was the winner of the parallel track of the SAT-Race 2010. This solver is the multithreaded version of the state-of-the-art sequential SAT solver Lingeling. Each SAT algorithm differs in the initialization of the decision heuristic and in the effort allowed in different pre-processing algorithms. In Plingeling, clause sharing is restricted to unit clauses.
- SAT4J// [68]: It uses a hybrid approach between search space splitting and portfolio and it is based on SAT4J 2.1 [57]. It starts with an initial stage of search space splitting and switches to a portfolio approach when load balancing becomes an issue or when a cutoff on the number of conflicts is reached. The initial set of partition variables is chosen using a heuristic based on the VSIDS score of all threads which leads to a better selection of the partition variables. Clause sharing is currently not implemented in this parallel SAT solver. Prior to parallel solving, SatELite [25] is used as a sequential preprocessor.

Appendix B: Evaluation of parallel shared memory SAT solvers

This section evaluates the parallel SAT solvers that have been developed in the last years for multicore environments. The evaluated solvers can be divided into the following categories: *search space splitting*, *portfolio*, *collaborative solving* and *hybrid approaches*. In the search space splitting category, the parallel SAT solvers `MiraXT` [59] and `satake` [100] were evaluated. In the portfolio category, the following parallel SAT solvers were evaluated: `antom` [87], `ManySAT 1.0` [43], `ManySAT 1.1` [42], `ManySAT 1.5` [37], `Plingeling` [10] and `SArTagnan` [54]. `MTSS` [23, 103], a parallel SAT solver based on collaborative solving, and `SAT4J//` [68], a parallel SAT solver based on a hybrid approach between portfolio and search space spitting, were also evaluated.

Due to the large number of evaluated parallel SAT solvers, this evaluation was divided into two stages. In the first stage, we used 100 instances from the SAT Race 2008.[1] Even though the instances of the SAT Race 2008 were challenging at that time, nowadays state-of-the-art solvers are able to solve the majority of those instances. Therefore, these instances are used as a qualification stage to determine the best solvers that will be further tested in the evaluation stage.

In the second stage, the best 5 solvers of the first stage are considered for additional testing. In this stage we consider 200 instances: the previous 100 instances from the SAT Race 2008 and the 100 instances from the SAT Race 2010.[2] Note that the instances from the SAT Race 2010 are more challenging for state-of-the-art SAT solvers than the instances from the SAT Race 2008. Therefore, this stage allows a better understanding of the performance of parallel SAT solvers.

The evaluation was performed on an 8 core Intel Xeon CPU E5410 (2.33 GHz with 28 GB of RAM) running Fedora Core 13 with a timeout of 1,800 s (wall clock time). As runtimes are highly deviating for parallel SAT solvers, each parallel solver was run three times on each instance. The runtimes shown in this section are the median of the runs for each instance. Hence, an instance is considered solved if it can be solved in at least two of the three runs of a solver.

B.1 Qualification stage

Table 2 shows the number of solved instances by the different parallel SAT solvers. From the 100 instances of the SAT Race 2008, 48 are satisfiable (SAT) and 52 are unsatisfiable (UNSAT). In order to compare the performance of modern parallel SAT solvers with state-of-the-art sequential SAT solvers, `CryptoMiniSAT 2.7.1` [97] was also run with a timeout of 1,800 s for each problem instance (wall clock time). `CryptoMiniSAT` was the winner of the sequential track of the SAT Race 2010. The total time shown in Table 2 is the cumulative time (wall clock time) spent on solving all the instances. If an instance is not solved, 1,800 s were spent trying to solve it, hence this time is added to the total time.

Modern parallel SAT solvers are faster and can solve more instances than the best sequential solver `CryptoMiniSAT`. For example, `Plingeling` was able to

[1] Available from http://baldur.iti.uka.de/sat-race-2008/

[2] Available from http://baldur.iti.uka.de/sat-race-2010/

Table 2 Number of instances solved and total solving time for the different parallel SAT solvers

Solver	# SAT	# UNSAT	Total	Total time (s)
Sequential solver				
CryptoMiniSAT	44	48	92	36,625.11
Parallel solvers (4 threads)				
Plingeling	48	49	97	24,157.26
ManySAT 1.5	46	51	97	24,530.02
ManySAT 1.1	46	48	94	23,972.78
ManySAT 1.0	43	47	90	30,374.87
SArTagnan	42	47	89	42,064.91
MTSS	42	37	79	61,281.17
MiraXT	40	37	77	62,053.08
SAT4J//	35	34	69	82,234.61
Satake	36	33	69	87,149.22

solve 5 more instances than `CryptoMiniSAT` while using 33% less time. Table 2 also shows that the best parallel SAT solvers are based on a portfolio approach. There is a considerable gap between parallel SAT solvers based on portfolios and the remaining ones. For example, `MTSS` is only able to solve 79 instances, whereas the worst performing portfolio solver, `ManySAT 1.0`, is able to solve 90 instances. Portfolio approaches are currently dominating all the other approaches for shared memory architectures. However, this gap may also be explained by the fact that `MiraXT`, `MTSS`, `satake` and `SAT4J//` are based on sequential SAT solvers that are no longer state-of-the-art solvers.

The best 5 parallel solvers in the qualification stage, namely `Plingeling`, `ManySAT 1.5`, `ManySAT 1.1`, `ManySAT 1.0` and `SArTagnan`, are further evaluated in the next section.

B.2 Evaluation stage

Table 3 shows the number of solved instances by the different parallel SAT solvers in the evaluation stage. From the 200 instances, 73 are satisfiable (SAT), 119 are unsatisfiable (UNSAT) and 8 remain unknown. `CrytoMiniSAT 2.7.1` was also run in order to compare the best state-of-the-art sequential SAT solver with modern parallel SAT solvers. `Plingeling` can solve 3 more instances than `CryptoMiniSAT` while using less 16% of time (wall clock). This improvement is due to the performance of `Plingeling` on SAT instances, since it solves 5 more instances than `CryptoMiniSAT`. Even though this is a small improvement, it shows that modern parallel SAT solvers can outperform state-of-the-art sequential SAT solvers.

`ManySAT 1.1` can solve 4 more instances than `ManySAT 1.0` due to its dynamic heuristic for clause sharing. Improving the quality and throughput of shared clauses allows solving more instances, particularly SAT instances. On the other hand, `ManySAT 1.5` is the best solver on solving UNSAT instances. The master-slave scheme within this portfolio allows a better quality of shared clauses between the master and the slave, which further improves the performance of the solver, particularly on UNSAT instances.

Table 3 Number of instances solved and total solving time for parallel SAT solvers

Solver	# SAT	# UNSAT	Total	Total time (s)
Sequential solver				
CryptoMiniSAT	66	106	172	88,290.58
Parallel solvers (4 threads)				
Plingeling	71	104	175	73,844.56
ManySAT 1.5	66	108	174	81,720.11
ManySAT 1.1	67	104	171	79,648.41
ManySAT 1.0	64	103	167	87,384.97
SArTagnan	60	100	160	107,805.91

Table 4 shows the speedup of parallel SAT solvers when compared with their sequential versions. The speedup shows the ratio between sequential and parallel solvers for instances that were solved by both versions. The speedup is determined by the ratio between the total solving time of the sequential version and the total solving time (wall clock time) of the parallel version. Only instances that were solved by both the sequential and parallel versions are considered for the total solving time of each version. Therefore, the speedup shows how many times the parallel version was faster than the sequential version. For example, `Plingeling` had a total speedup of 1.60, i.e. it was 1.60× faster than the sequential version. In each line of Table 4 is presented the speedup for SAT and UNSAT instances and the total speedup. Between brackets is shown the number of instances that were considered for determining the speedup, i.e. the number of instances that were solved by both the sequential and parallel versions.

`Plingeling` only shares unit clauses and therefore presents the lowest overall speedup. Nevertheless, it has a speedup of 1.91 on solving SAT instances showing that using different preprocessing techniques, while sharing unit clauses, can lead to a reasonable speedup. `ManySAT 1.5` has the highest speedup on solving UNSAT instances, which supports the idea that its design is specially tuned for solving UNSAT instances. On the other hand, it seems to reduce the effectiveness of the solver on solving SAT instances. `ManySAT 1.1` also shows better speedups on solving UNSAT instances than its previous version. However, `ManySAT 1.0` has the best speedup on solving SAT instances. This suggests that dynamic heuristics for clause sharing increase the robustness of the solver by solving more instances but at the same time reduce the speedup achieved on solving SAT instances. `SArTagnan` was developed for 8 threads and should not be used with just one thread since it is not efficient. Therefore, the speedup when using 4 threads is significant.

Table 5 shows the average variation between the three runs of each parallel solver. This variation was determined through the relative variation coefficient that was used

Table 4 Speedup of parallel SAT solvers (4 threads) with respect to their sequential versions

Solver	SAT speedup	UNSAT speedup	Total speedup
Plingeling	1.91 (67)	1.41 (99)	1.60 (166)
ManySAT 1.5	1.11 (60)	2.75 (86)	1.73 (146)
ManySAT 1.1	1.44 (58)	2.69 (91)	2.27 (149)
ManySAT 1.0	4.45 (52)	1.97 (93)	2.71 (145)
SArTagnan	4.50 (32)	11.16 (51)	6.24 (83)

Table 5 Variation of the solving time between the three runs (4 threads)

Solver	SAT variation (%)	UNSAT variation (%)	Total variation (%)
Plingeling	16.10	4.43	8.49
ManySAT 1.5	23.27	4.14	10.93
ManySAT 1.1	21.66	5.41	11.10
ManySAT 1.0	25.90	6.15	13.08
SArTagnan	24.20	11.26	15.48

in the SAT Race 2008. The relative variation coefficient is given by: $100 - \sigma/(\mu \times \sqrt{n})$, where σ is the standard deviation, μ the average solving time and n the number of runs.

It was observed in the SAT Races 2008 and 2010 that the variation for parallel SAT solvers is considerable for SAT instances and usually reduced for UNSAT instances. This is due to sharing learned clauses, since in each run clauses are shared in different steps of the solving process, which may lead to completely different search processes. `SArTagnan` is the only solver that shows a high variation for UNSAT instances. This can be partially explained by the fact that some threads simplify the database and learn extra clauses, thus further increasing the differences between each run. `Plingeling` shows the smallest variation since it only shares unit clauses. `ManySAT 1.5` has the best performance on solving UNSAT instances and is also the solver that shows the smallest variation for this set of instances. `ManySAT 1.0` had the best speedup for solving SAT instances and is also the solver that presents the highest variation for this set of instances.

B.2.1 Scalability

With the current increase in the number of cores, scalability becomes an important issue. In the SAT Race 2010, each solver could use up to eight threads. In this section, we present the evaluation done with `Plingeling`, `ManySAT 1.5` and `SArTagnan` running eight threads. `Plingeling` and `ManySAT 1.5` were the best parallel solvers with four threads and `SArTagnan` was originally designed to run with eight threads.

Table 6 shows the number of solved instances with four and eight threads. `SArTagnan` shows a great improvement and is competitive with other modern parallel SAT solvers that use eight threads. Indeed, it was able to solve 171 instances with 8 threads whereas previously with 4 threads it could only solve 160 instances. This suggests that using threads to perform auxiliary work rather than using a CDCL SAT algorithm is only worth when the number of threads is large. On the other hand, `ManySAT 1.5` showed a detrimental effect of memory contention and solved less 1

Table 6 Comparison in the number of solved instances from 4 to 8 threads

Solver	4 threads			8 threads		
	# SAT	# UNSAT	Total	# SAT	# UNSAT	Total
Plingeling	71	104	175	70	107	177
ManySAT 1.5	66	108	174	68	105	173
SArTagnan	60	100	160	66	105	171

instance than before. `Plingeling` showed a small improvement when using eight threads, being able to solve 2 more instances than before. Nevertheless, it is not clear whether modern parallel SAT solvers will scale well for a larger number of cores. Even with 8 threads, the overall improvements were small and in some cases it was more beneficial to run the parallel solver with only 4 threads. Additionally, since all of these solvers duplicate their clause database for each thread, memory problems can occur when solving large instances with many threads.

References

1. Arbelaez, A., & Hamadi, Y. (2011). Improving parallel local search for SAT. In *Learning and intelligent optimization* (pp. 46–60).
2. Asín, R., Nieuwenhuis, R., Oliveras, A., & Rodríguez-Carbonell, E. (2010). Practical algorithms for unsatisfiability proof and core generation in SAT solvers. *AI Communications, 23*(2–3), 145–157.
3. Audemard, G., Bordeaux, L., Hamadi, Y., Jabbour, S., & Sais, L. (2008). A generalized framework for conflict analysis. In *International conference on theory and applications of satisfiability testing* (pp. 21–27).
4. Audemard, G., & Simon, L. (2009). Predicting learnt clauses quality in modern SAT solvers. In *International joint conference on artificial intelligence* (pp. 399–404).
5. Bacchus, F., & Winter, J. (2003). Effective preprocessing with hyper-resolution and equality reduction. In *International conference on theory and applications of satisfiability testing* (pp. 341–355).
6. Baptista, L., & Marques-Silva, J. (2000). Using randomization and learning to solve hard real-world instances of satisfiability. In *International conference on principles and practice of constraint programming* (pp. 489–494).
7. Bayardo, R. J., & Schrag, R. C. (1997). Using CSP look-back techniques to solve real-world SAT instances. In *AAAI conference on artificial intelligence* (pp. 203–208).
8. Biere, A. (2008). Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation, 4*(2–4), 75–97.
9. Biere, A. (2009). *PrecoSAT*. SAT Competition, Solver Description.
10. Biere, A. (2010). *Lingeling, plingeling, PicoSAT and PrecoSAT at SAT Race 2010*. SAT Race, Solver Description.
11. Blochinger, W. (2005). Towards robustness in parallel SAT solving. In *International conference on parallel computing* (pp. 301–308).
12. Blochinger, W., Sinz, C., & Küchlin, W. (2003). Parallel propositional satisfiability checking with distributed dynamic learning. *Journal of Parallel Computing, 29*(7), 969–994.
13. Blochinger, W., Westje, W., Kuchlin, W., & Wedeniwski, S. (2005). ZetaSAT—Boolean SATisfiability solving on desktop grids. In *International symposium on cluster computing and the grid* (pp. 1079–1086).
14. Böhm, M., & Speckenmeyer, E. (1996). A fast parallel SAT-solver—Efficient workload balancing. *Annals of Mathematics and Artificial Intelligence, 17*(3–4), 381–400.
15. Bordeaux, L., Hamadi, Y., & Samulowitz, H. (2009). Experiments with massively parallel constraint solving. In *International joint conference on artificial intelligence* (pp. 443–448).
16. Chrabakh, W., & Wolski, R. (2003). GridSAT: A Chaff-based distributed SAT solver for the grid. In *International supercomputing conference* (pp. 37–49).
17. Chu, G., Harwood, A., & Stuckey. P. J. (2009). Cache conscious data structures for Boolean satisfiability solvers. *Journal on Satisfiability, Boolean Modeling and Computation, 6*, 99–120.
18. Chu, G., Schulte, C., & Stuckey, P. J. (2009). Confidence-based work stealing in parallel constraint programming. In *International conference on principles and practice of constraint programming* (pp. 226–241).
19. Chu, G., & Stuckey, P. J. (2008). *PMiniSAT: A parallelization of MiniSAT 2.0*. SAT Race, Solver Description.
20. Clearwater, S. H., Huberman, B. A., & Hogg, T. (1991). Cooperative solution of constraint satisfaction problems. *Science, 254*(5035), 1181–1183.

21. Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM, 5*(7), 394–397.
22. Davis, M., & Putnam, H. (1960). A computing procedure for quantification theory. *Journal of the ACM, 7*(3), 201–215.
23. Dequen, G., Vander-Swalmen, P., & Krajecki, M. (2009). Toward easy parallel SAT solving. In *International conference on tools with artificial intelligence* (pp. 425–432).
24. Edmund, M. Y., Durfee, E. H., Ishida, T., & Kuwabara, K. (1992). Distributed constraint satisfaction for formalizing distributed problem solving. In *International conference on distributed computing systems* (pp. 614–621).
25. Eén, N., & Biere, A. (2005). Effective preprocessing in SAT through variable and clause elimination. In *International conference on theory and applications of satisfiability testing* (pp. 61–75).
26. Eén, N., & Sörensson, N. (2003). An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing* (pp. 502–518).
27. Feldman, R., Monien, B., Mysliwietz, P., & Vornberger, O. (1990). Distributed game tree search. In *Parallel algorithms for machine intelligence and pattern recognition* (pp. 66–101). Springer-Verlag.
28. Feldmann, R., Monien, B., & Schamberger, S. (2000). A distributed algorithm to evaluate quantified Boolean formulae. In *AAAI conference on artificial intelligence* (pp. 285–290).
29. Feldman, Y., Dershowitz, N., & Hanna, Z. (2005). Parallel multithreaded satisfiability solver: Design and implementation. *Electronic Notes in Theoretical Computer Science, 128*(3), 75–90.
30. Forman, S., & Segre, A. (2002). NAGSAT: A randomized, complete, parallel solver for 3-SAT. In *International conference on theory and applications of satisfiability testing* (pp. 236–243).
31. Gent, I. P., Jefferson, C., Miguel, I., Moore, N. C., Nightingale, P., Prosser, P., et al. (2011). A preliminary review of literature on parallel constraint solving. In *Workshop on parallel methods for constraint solving*.
32. Gil, L., Flores, P., & Silveira, L. M. (2008). PMSat: A parallel version of MiniSAT. *Journal on Satisfiability, Boolean Modeling and Computation, 6*, 71–98.
33. Goldberg, E. (2006). Determinization of resolution by an algorithm operating on complete assignments. In *International conference on theory and applications of satisfiability testing* (pp. 90–95).
34. Goldberg, E. (2008). A decision-making procedure for resolution-based SAT-solvers. In *International conference on theory and applications of satisfiability testing* (pp. 119–132).
35. Goldberg, E., & Novikov, Y. (2002). BerkMin: A fast and robust SAT-solver. In *Design, automation, and test in Europe* (pp. 142–149).
36. Gomes, C. P., Selman, B., & Kautz, H. (1998). Boosting combinatorial search through randomization. In *AAAI conference on artificial intelligence* (pp. 431–437).
37. Guo, L., Hamadi, Y., Jabbour, S., & Sais, L. (2010). Diversification and intensification in parallel SAT solving. In *International conference on principles and practice of constraint programming* (pp. 252–265).
38. Hamadi, Y. (1999). Optimal distributed arc-consistency. In *International conference on principles and practice of constraint programming* (pp. 219–233).
39. Hamadi, Y. (2002). Interleaved backtracking in distributed constraint networks. *International Journal on Artificial Intelligence Tools, 11*(2), 167–188.
40. Hamadi, Y. (2003). *Disolver: A distributed constraint solver*. Technical Report MSR-TR-2003-91, Microsoft Research.
41. Hamadi, Y., Jabbour, S., Piette, C., & Sais, L. (2011). Deterministic parallel DPLL. *Journal on Satisfiability, Boolean Modeling and Computation, 7*(4), 127–132.
42. Hamadi, Y., Jabbour, S., & Sais, L. (2009). Control-based clause sharing in parallel SAT solving. In *International joint conference on artificial intelligence* (pp. 499–504).
43. Hamadi, Y., Jabbour, S., & Sais, L. (2009). ManySAT: A parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation, 6*, 245–262.
44. Hamadi, Y., Marques-Silva, J., & Wintersteiger, C. M. (2011). Lazy decomposition for distributed decision procedures. In *International workshop on parallel and distributed methods in verification* (pp. 43–54).
45. Hamadi, Y., & Ringwelski, G. (2011). Boosting distributed constraint satisfaction. *Journal of Heuristics, 17*(3), 251–279.
46. Han, H., & Somenzi, F. (2009). On-the-fly clause improvement. In *International conference on theory and applications of satisfiability testing* (pp. 209–222).

47. Hertel, P., Bacchus, F., Pitassi, T., & Gelder, A. V. (2008). Clause learning can effectively p-simulate general propositional resolution. In *AAAI conference on artificial intelligence* (pp. 283–290).
48. Hyvärinen, A., Junttila, T., & Niemelä, I. (2006). A distribution method for solving SAT in grids. In *International conference on theory and applications of satisfiability testing* (pp. 430–435).
49. Jacobson, V. (1988). Congestion avoidance and control. In *Special interest group on data communication* (pp. 314–329).
50. Jurkowiak, B., Li, C. M., & Utard, G. (2001). Parallelizing satz using dynamic workload balancing. In *International conference on theory and applications of satisfiability testing* (pp. 205–211).
51. Kalinnik, N., Schubert, T., Ábrahám, E., Wimmer, R., & Becker, B. (2009). Picoso—A parallel interval constraint solver. In *International conference on parallel and distributed processing techniques and applications* (pp. 473–479).
52. Kasif, S. (1990). On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Journal of Artificial Intelligence, 45*(3), 275–286.
53. Kottler, S. (2010). SAT solving with reference points. In *international conference on theory and applications of satisfiability testing* (pp. 143–157).
54. Kottler, S., & Kaufmann, M. (2011). SArTagnan—A parallel portfolio SAT solver with lockless physical clause sharing. In *Pragmatics of SAT workshop*.
55. Kullmann, O. (2000). Investigations on autark assignments. *Discrete Applied Mathematics, 107*, 99–137.
56. Kunz, W., & Pradhan, D. K. (992). Recursive learning: An attractive alternative to the decision tree for test generation in digital circuits. In *International test conference* (pp. 816–825).
57. Le Berre, D., & Parrain, A. (2010). The sat4j library, release 2.2 system description. *Journal on Satisfiability Boolean Modeling and Computation, 7*, 59–64.
58. Lewis, M., Marin, P., Schubert, T., Narizzano, M., Becker, B., & Giunchiglia, E. (2009). PaQuBE: Distributed QBF solving with advanced knowledge sharing. In *International conference on theory and applications of satisfiability testing* (pp. 509–523).
59. Lewis, M., Schubert, T., & Becker, B. (2007). Multithreaded SAT solving. In *Asia and south pacific design automation conference* (pp. 926–931).
60. Lewis, M., Schubert, T., & Becker, B. (2009). QMiraXT—A multithreaded QBF solver. In *Methoden und beschreibungssprachen zur modellierung und verifikation von Schaltungen und systemen* (pp. 7–16).
61. Luby, M., Sinclair, A., & Zuckerman, D. (1993). Optimal speedup of Las Vegas algorithms. *Journal of Information Processing Letters, 47*(4), 173–180.
62. Lynce, I., & Marques-Silva, J. (2005). Efficient data structures for backtrack search SAT solvers. *Annals of Mathematics and Artificial Intelligence, 43*(1–4), 137–152.
63. Mailler, R., & Lesser, V. (2004). Solving distributed constraint optimization problems using cooperative mediation. In *International conference on autonomous agents and multiagent systems* (pp. 438–445).
64. Manthey, N. (2011). Parallel SAT solving—Using more cores. In *Pragmatics of SAT workshop*.
65. Marin, P., Lewis, M., Schubert, T., Narizzano, M., Becker, B., & Giunchiglia, E. (2009). Comparison of knowledge sharing strategies in a parallel QBF solver. In *Workshop on parallel satisfiability solving* (pp. 161–167).
66. Marques-Silva, J., Lynce, I., & Malik, S. (2009). Conflict-driven clause learning SAT solvers. In *Handbook of satisfiability* (Vol. 185, pp. 131–153). IOS Press.
67. Marques-Silva, J., & Sakallah, K. A. (1999). GRASP—A search algorithm for propositional satisfiability. *IEEE Transactions on Computers, 48*. 506–521.
68. Martins, R., Manquinho, V., & Lynce, I. (2010). Improving search space splitting for parallel SAT solving. In *International conference on tools with artificial intelligence* (pp. 336–343).
69. Modi, P. J., Shen, W. M., Tambe, M., & Yokoo, M. (2005). Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Journal of Artificial Intelligence, 16*(1–2), 149–180.
70. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Design automation conference* (pp. 530–535).
71. Nguyen, T., & Deville, Y. (1998). A distributed arc-consistency algorithm. *Science of Computer Programming, 30*, 227–250.
72. Ohmura, K., & Ueda, K. (2009). c-SAT: A parallel SAT solver for clusters. In *International conference on theory and applications of satisfiability testing* (pp. 524–537).
73. Pham, D. N., Gretton, C. (2009). *gNovelty(v. 2)*. SAT Competition, Solver Description.

74. Piette, C., Hamadi, Y., & Lakhdar, S. (2008). Vivifying propositional clausal formulae. In *European conference on artificial intelligence* (pp. 525–529).
75. Pilarski, S., & Hu, G. (2002). SAT with partial clauses and back-leaps. In *Design automation conference* (pp. 743–746).
76. Pipatsrisawat, K., & Darwiche, A. (2007). A lightweight component caching scheme for satisfiability solvers. In *International conference on theory and applications of satisfiability testing* (pp. 294–299).
77. Pipatsrisawat, K., & Darwiche, A. (2011). On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence, 175*, 512–525.
78. Plaza, S., Kountanis, I., Andraus, Z., Bertacco, V., & Mudge, T. (2006). Advances and insights into parallel SAT solving. In *International workshop on logic and synthesis* (pp. 188–194).
79. Plaza, S., Markov, I., & Bertacco, V. (2008). Low-latency SAT solving on multicore processors with priority scheduling and XOR partitioning. In *International workshop on logic and synthesis*.
80. Prestwich, S. D., & Lynce, I. (2006). Local search for unsatisfiability. In *International conference on theory and applications of satisfiability testing* (pp. 283–296).
81. Rolf, C. C., & Kuchcinski, K. (2008). Load-balancing methods for parallel and distributed constraint solving. In *Interational conference on cluster computing* (pp. 304–309).
82. Rolf, C. C., & Kuchcinski, K. (2010). Combining parallel search and parallel consistency in constraint programming. In *TRICS workshop at the international conference on principles and practice of constraint programming* (pp. 38–52).
83. Roll, A. (2001). Criticality and parallelism in GSAT. In *Workshop on theory and applications of satisfiability testing* (pp. 150–161).
84. Ryan, L. (2004). *Efficient algorithms for clause-learning SAT solvers*. Master's thesis, Simon Fraser University.
85. Schubert, T., Lewis, M., & Becker, B. (2005). PaMira—A parallel SAT solver with knowledge sharing. In *International workshop on microprocessor test and verification* (pp. 29–36).
86. Schubert, T., Lewis, M., & Becker, B. (2009). PaMiraXT: Parallel SAT solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation, 6*, 203–222.
87. Schubert, T., Lewis, M., & Becker, B. (2010). *Antom—Solver description*. SAT Race, Solver Description.
88. Schulz, S., & Blochinger, W. (2010). Parallel SAT solving on peer-to-peer desktop grids. *Journal of Grid Computing, 8*, 443–471.
89. Selman, B., Kautz, H. A., & Cohen, B. (1996). Local search strategies for satisfiability testing. In *Cliques, coloring, and satisfiability: Second DIMACS implementation challenge* (Vol. 26, pp. 521–532). American Mathematical Society.
90. Selman, B., Levesque, H., & Mitchell, D. (1992). A new method for solving hard satisfiability problems. In *AAAI conference on artificial intelligence* (pp. 440–446).
91. Sheeran, M., & Stalmarck, G. (2000). A tutorial on Stalmarck's proof procedure for propositional logic. *Formal Methods in System Design, 16*, 23–58.
92. Silaghi, M.-C., Sam-haroud, D., & Faltings, B. V. (2000). Asynchronous search with aggregations. In *AAAI conference on artificial intelligence* (pp. 917–922).
93. Singer, D. (2006). Parallel resolution of the satisfiability problem: A survey. In *Parallel combinatorial optimization* (pp. 123–147). Wiley.
94. Singer, D., & Monnet, A. (2008). JaCk-SAT: A new parallel scheme to solve the satisfiability problem (SAT) based on join-and-check. In *Parallel processing and applied mathematics* (pp. 249–258).
95. Sinz, C., Blochinger, W., & Küchlin, W. (2001). PaSAT—Parallel SAT-Checking with lemma exchange: Implementation and applications. *Electronic Notes in Discrete Mathematics, 9*, 205–216.
96. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., & Dongarra, J. (1998). *MPI-the complete reference*. MIT Press.
97. Soos, M. (2010). *CryptoMiniSat 2.5.0*. SAT Race, Solver Description.
98. Söreasson, N., & Eée, N. (2008). *MINISAT 2.1 and MINISAT++ 1.0—SAT race 2008 editions*. SAt Race, Solver Description.
99. Sripriya, G., Bundy, A., & Smaill, A. (2009). Concurrent-distributed programming techniques for SAT using DPLL-stalmarck. In *Workshop on parallel satisfiability solving* (pp. 168–175).
100. Tsuyuzaki, K., Ohmura, K., & Ueda, K. (2009). *Satake: Solver description*. SAT Competition, Solver Description.
101. Valiant, L., & Vazirani, V. (1985). NP is as easy as detecting unique solutions. In *Symposium on theory of computing* (pp. 458–463).

102. Van Gelder, A. (1999). Autarky pruning in propositional model elimination reduces failure redundancy. *Journal of Automated Reasoning, 23*, 137–193.
103. Vander-Swalmen, P., Dequen, G., & Krajecki, M. (2009). A collaborative approach for multi-threaded SAT solving. *International Journal of Parallel Programming, 37*(3), 324–342.
104. Vautard, J., Lallouet, A., & Hamadi, Y. (2010). A parallel solving algorithm for quantified constraints problems. In *International conference on tools with artificial intelligence* (pp. 271–274).
105. Williams, R., Gomes, C. P., & Selman, B. (2003). Backdoors to typical case complexity. In *International joint conference on artificial intelligence* (pp. 1173–1178).
106. Wintersteiger, C. M., Hamadi, Y., & de Moura, L. M. (2009). A concurrent portfolio approach to smt solving. In *Computer aided verification* (pp. 715–720).
107. Yokoo, M., Durfee, E. H., Ishida, T., & Kuwabara, K. (1998). The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering, 10*, 673–685.
108. Zhang, H. (1997). SATO: An efficient propositional prover. In *International conference on automated deduction* (pp. 272–275).
109. Zhang, H., Bonacina, M. P., Bonacina, M. P., & Hsiang, J. (1996). PSATO: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation, 21*, 543–560.
110. Zhang, L. (2005). On subsumption removal and on-the-fly CNF simplification. In *International conference on theory and applications of satisfiability testing* (pp. 482–489).
111. Zhang, L., Madigan, C., Moskewicz, M., & Malik, S. (2001). Efficient conflict driven learning in a Boolean satisfiability solver. In *International conference on computer-aided design* (pp. 279–285).
112. Zhang, L., & Malik, S. (2003). Cache performance of SAT solvers: A case study for efficient implementation of algorithms. In *International conference on theory and applications of satisfiability testing* (pp. 287–298).
113. Zhang, L., & Malik, S. (2003). Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, automation, and test in Europe* (pp. 10880–10885).