Modern Cooperative Parallel SAT Solving

Ahmed Irfan and Davide Lanti and Norbert Manthey

Knowledge Representation and Reasoning Group
Technische Universität Dresden, 01062 Dresden, Germany

Abstract. Nowadays, powerful parallel SAT solvers are based on an algorithm portfolio. The alternative approach, (iterative) search space partitioning, cannot keep up, although, according to the literature, iterative partitioning systems should scale better than portfolio solvers. In this paper we identify key problems in current parallel cooperative SAT solving approaches, most importantly communication, how to partition the search space, and how to utilize the sequential search engine. First, we improve on each problem separately. In a further step, we show that combining all the improvements leads to a state-of-the-art parallel SAT solver, which does not use the portfolio approach, but instead relies on iterative partitioning. The experimental evaluation of this system completely changes the picture about the performance of search space partitioning SAT solvers: on instances of a combined benchmark of recent SAT competitions, the presented approach can keep up with the winners of last years SAT competition. The combined improvements improve the existing cooperative solver SPLITTER by 24%: instead of 561 out of 880 instances, the new solver PCASSO can solve 696 instances.

1 Introduction

Many practical problems could be sped up by utilizing massively parallel hardware, for example by utilizing the GPGPU. Especially for problems with local calculations these execution units are beneficial (e.g. [1,2]). For more complex problems, for instance the satisfiability testing problem [3] whose complexity is $\mathcal{NP}$, only a few massively parallel solutions have been proposed – yet not suitable to cope with neither modern sequential CDCL solvers, nor with the high complexity of current application formulas [4]. The presented results consider only 3SAT with 100 variables – a size being easily solved by modern sequential SLS solvers. Still, there exists the interest in improving the performance of SAT solving, and parallel approaches have not yet been exhaustively explored.

With the goal of having a massively parallel SAT solver in mind, we go a step backwards from GPGPUs to multi-core CPUs. Nowadays, a usual CPU ships with four cores, and high performance computing CPUs already contain up to 16 cores, which should be exploited for solving SAT. For these architectures, many solutions to execute parallel solvers have been proposed. The latest generation of these solvers, namely *portfolio* solvers, has been introduced with MANYSAT [5], a solver that runs several configurations of a single solving engine in parallel, and which is based on a modern sequential solving engine. These *incarnations* of the solving engine all solve the same input instance, and additionally share clauses, that have been learned during their search process. Much research has been done on this type of solvers, for example investigating the properties of clause sharing [6,7], or whether additional information sharing could improve the performance of the parallel SAT solver [6,8]. However, there also exist parallel portfolio solvers that

simply combine many solvers with the aim to have a specialized solver for each category of instances, such that the overall portfolio provides a good performance [9].

An alternative approach to solve SAT in parallel is to *partition* the search space of the input formula, and then solve partitions in parallel. Recently, there has been less research on this field, especially for the multi core platform. Within the past five years, there have been only a few parallel search space splitting solvers: CUBEANDCONQUER [10,11], and SPLITTER [12][1]. A reason for this situation might be the following: In [13] it has been shown that when partitioning an input formula, and solving only the partitions in parallel but not considering the input formula itself, the overall run time of the parallel solver can be higher than when solving the input formula with a sequential solver. We refer to this splitting scheme as *plain partitioning*. When a solver incarnation proved the unsatisfiability of a partition, it simply continues by solving another partition, assuming that sufficiently many partitions have been provided. The first solver CUBEANDCONQUER follows this scheme. Another approach is to consider the input formula as well, and provide less partitions. Then, we solve the input formula and the partitions in parallel. As soon as we have another free solver incarnation, we assign a remaining partition to that solver. If no more partitions are present, we divide the search space of a partition into further partitions, where each of them is ready to be assigned to new solvers again. This solving scheme has been introduced for solving SAT with grids computing, and is referred to as *iterative partitioning* in the literature [13]. Most importantly, iterative partitioning has been shown to be more powerful in the presence of more parallel execution units [13]. In contrast to this statement, the solver SPLITTER that uses this scheme showed a poor performance in international competitions [14], even when adding an improved clause sharing [15]. Currently, the performance gap between portfolio solvers and search space splitting solvers is still huge, as the following table shows, both PLINGELING and PENELOPE solve more instances and are faster than SPLITTER. The data of the table has been created on a benchmark of application 880 instances originating from the SAT competition 2009, the SAT Challenge 2012 and the hard unselected instances of the SAT Challenge 2012. We executed the solvers on an 8 cores of an 16-core AMD Opteron 6274 with 2.2 GHz and 8 GB of memory. The time limit was set to 7200 seconds. In all the following tables we report the total number of solved instances (TOT), the number of solved satisfiable (SAT) and unsatisfiable instances (UNSAT). Furthermore, we give the average (Wall Time) and the median wall clock time (Median Time) for the whole benchmark, as well as the Par10 score, which takes the mean of the sum the wall time for all solved instances and ten times the timeout for each unsolved instance.

Solver	TOT	SAT	UNSAT	Wall Time	Median Time	PAR10
PENELOPE	704	304	400	305.649	89.39	14645
PLINGELING	672	296	376	663.526	442.28	17525
SPLITTER	561	292	269	450.126	366.42	26387

In this paper we analyze the decision choices made in SPLITTER, and try to improve the system, such that the results of the implementation are in line with the theoretical scalability results. As the above table indicates, this can be achieved mainly by improving the performance on unsatisfiable instances, where the gap to PENELOPE is 131 instances.

[1] The are other parallel SAT solvers that support search space splitting, for example CLASP or PMINISAT – however, no detailed results have been reported on these systems.

For this, we focused on the following aspects: (i) how to partition the input formula, (ii) how to share learned clauses among partitions, (iii) how to initialize solving engines, (iv) how to solve partitions, (v) how to perform extra communication among solving engines, and finally (vi) how to treat special situations that may arise in iterative partitioning. For each point we first analyze the decisions made in SPLITTER, and then try to identify weaknesses, propose improvements and finally evaluate the performance of the proposed modification on a large benchmark of application instances used or submitted to recent SAT competitions. We implemented all modifications into another system called PCASSO, which is an acronym of *Parallel, CooperAtive Sat SOlver*. When comparing SPLITTER with PCASSO, our evaluation shows a performance improvement of 24%. Furthermore, PCASSO now can keep up with PENELOPE, the best single engine parallel SAT solver of last years competition. The results of our work prove that search space partitioning SAT solvers are relevant, and supported by theoretical results [13], should be considered as promising candidates for solving SAT on future massively parallel computing architectures.

2 Preliminaries

We assume a fixed set V of Boolean variables, or briefly just *variables* or *atoms*. A *literal* is a variable x (*positive literal*) or a negated variable $\overline{x}$ (*negative literal*). We overload the overbar notation: the *complement* $\bar{l}$ of a positive (negative, resp.) literal l is the negative (positive, resp.) literal with the same variable as l. A *clause* is a finite set of literals, a *formula in conjunctive normal form* is a finite set of clauses. An *interpretation* is a mapping from the set V of all Boolean variables to the set $\{\top, \bot\}$ of truth values. In the following, we assume the reader to be familiar with propositional logic, and how propositional formulas are evaluated under interpretations. More details can be found in [16]. A clause that contains exactly a single literal is called a *unit clause*. If x is a Boolean variable and $C = x \vee C'$ as well as $D = \overline{x} \vee D'$ are clauses, then the clause $C' \vee D'$ is called the *resolvent of C and D upon x*. The SAT problem is to answer whether a given formula F is satisfiable. Since [3], SAT is known to be in the complexity class $\mathcal{NP}$.

3 Sequential SAT Solving

For showing the satisfiability or unsatisfiability of formulas that originate from applications, currently the *conflict driven clause learning* (CDCL) algorithm [17] shows the best performance. In principle, this algorithm follows the tree search of the well known DPLL algorithm [18], but it treats conflicts differently. Similar to DPLL, if the deduction of the solver, *unit propagation*, reached a fix point, a search step (*decision*) is performed, so that deduction can be applied again. However, if unit propagation reveals a *conflict*, which represents that the current sub-tree does not contain a solution, a *learned clause* is generated by resolution [17]. This learned clause is used to partially undo the current path and to continue search from this point by propagation. The search is stopped if either a satisfying interpretation has been found, or if the learned clause is the empty clause. In the latter case, the formula is found to be unsatisfiable.

Modern CDCL solvers are enhanced with advanced heuristics. First, the *decision heuristic* prefers variables, which have been used in the derivation of recent learned clauses [19], by maintaining an *activity* per variable. The polarity to branch on is determined by using the *phase-saving* scheme [20]. Furthermore, learned clauses are maintained, and after some time also removed again, where heuristics answer the follow-

ing questions: (i) when to remove learned clauses, and (ii) which learned clauses to remove. For (i) several schedules have been proposed (e.g. [21,22]). For (ii), learned clauses are augmented with measures, for example an *activity* [21], or the *literal block distance*(LBD) [22]. In addition, the search process is *restarted* resetting the current partial interpretation [23]. Again, different restart policies have been proposed [24,21,25,26,27]. More details about state-of-the-art SAT solving with the CDCL algorithm can be found in [28].

4 Parallel SAT Solving

Parallel SAT solvers can be divided into two families: *cooperative* approaches, where the search space is partitioned and each partition is assigned to a solver incarnation and *competitive* approaches, where instead each solver incarnation is fed with the same formula. Current state-of-the-art is represented by competitive approaches, also known as *portfolio* procedures, whereas cooperative approaches, also known as *search space partitioning* procedures, were quite popular in the past. Since we focus on search space partitioning procedures, we will discuss the portfolio approach only briefly. A more detailed survey on parallel SAT solving is given in [29,30].

4.1 Solving SAT with the Portfolio Approach

In the portfolio approach, several solvers solve the same formula in parallel. A nice property of this approach is that the parallel search can be terminated as soon as one solver incarnation found a solution. This property motivates combining special solvers for formulas from special categories to a portfolio solver, which then solves a given formula in the time required by the fastest sequential component solver. Among portfolio systems, different sub approaches arose, mainly differing in the way of which solving engines are used and how complex the communication among these engines is: portfolio solvers in recent competitions like PPFOLIO [9] and PFOLIOUZK [31] simply execute several powerful SAT solvers in parallel, even scheduled on a sequential machine.

A more sophisticated portfolio approach uses a single solving engine and executes multiple incarnations with different configurations in parallel (e.g. MANYSAT [5] or PENELOPE [32]). In this approach, learned clauses can be easily shared among the incarnations. Knowledge sharing enables the portfolio solver to solve a formula even faster than the best sequential incarnation, because the search of this best incarnation is enhanced with shared clauses that cut off search space. MANYSAT [5], winner of the *SAT Competition 2009* in parallel SAT solver track, shares clauses of up to size eight, and PENELOPE [7], runner-up of the *SAT Challenge 2012* in parallel SAT solver track, shares clauses of *LBD* value up to eight, and PLINGELING [8], winner of the parallel SAT solver track of the *SAT Race 2010* and *SAT Competition 2011*, shares only unit clauses and literal equivalences. Surprisingly, the portfolio solver PFOLIOUZK [31], winner of *SAT Challenge 2012* in the parallel SAT solver track, does not share any clause. Even more communication to these single engine portfolio solvers has been introduced with the PLINGELING system [8]. Its solving engine LINGELING applies simplification techniques during search, and thus knowledge about equivalent literals can be shared.

There exists much research on portfolio solvers, most prominently on sharing information among solving incarnations. First, a static sharing limit for the size of shared clauses has been introduced [5], which has been replaced by a dynamic quality-based filter [33]. Furthermore, not only the size of the clause, but also its LBD has been taken

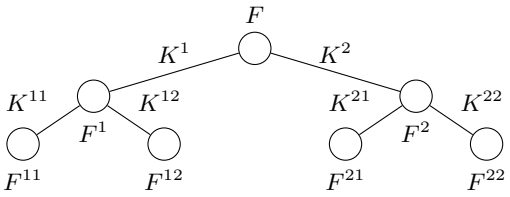

Fig. 1. The tree shows how a formula can be partitioned iteratively by using a partitioning function that creates two child formulae.

into account for sharing clauses [32]. Finally, not only learned clauses, but also active variables and other information has been shared [6,8]. A discussion about the soundness of clause sharing is presented in [34].

4.2 Solving SAT with the Search Space Partitioning Approach

4.2.1 Creating Partitions Partitions in cooperative SAT solvers are created through *partition functions*, where a partition function is a function ϕ such that, given a formula F and a natural number $n \in \mathbb{N}^+$, $\phi(F, n) := (F_1, \ldots, F_n)$, where $F \equiv F_1 \vee \ldots \vee F_n$ and each pair of partitions is disjoint: $i \neq j \in [1, n]$, $F_i \wedge F_j \models \bot$. Without loss of generality we assume that partitions $F_1, \ldots, F_n$ are always of the form $F \wedge K_1, \ldots, F \wedge K_n$, where $K_1, \ldots, K_n$ are sets of clauses, called *partitioning constraints*. By iteratively applying the partition function to a formula F, a *partition tree* like the one in *Figure* 1 is produced. Nodes in the partition tree are tagged with their *positions*: the root node F is tagged with the empty position ϵ; the i-th successor (from left to right) of a node F^p at position p is the node F^{pi}. Please notice that, as positions are strings, the standard *prefix* relation among strings ($<$) is defined for positions as well.

Partitions in SPLITTER and PCASSO are created through *scattering* [35]. The idea is to define each partitioning constraint as conjunctions of *cubes* [10], where a cube is a formula $Q := \{C_1, \ldots, C_k\}$ such that $|C_i| = 1$, for each $1 \leq i \leq k$ and $k \geq 1$. Observe that the negation of a cube $Q := \{\{l_1\}, \ldots, \{l_k\}\}$ is the clause $\{\overline{l_1}, \ldots, \overline{l_k}\}$. More precisely, given a formula F_0 and an integer n, n partitions $F_1, \ldots, F_n$ are created by using $n-1$ cubes $Q_1, \ldots, Q_{n-1}$ and applying them according to the following schema: $F_1 := F_0 \wedge Q_1$; $F_{m+1} := F_0 \wedge (\bigwedge_{i=1}^{m} \overline{Q_i}) \wedge Q_{m+1} (1 \leq m < n-1)$; and finally $F_n := F_0 \wedge \bigwedge_{i=1}^{n-1} \overline{Q_i}$.

4.2.2 Solving Partitions Hyvärinen [36] identifies two strategies for solving nodes in the partition tree: *plain partitioning* and *iterative partitioning*, where the latter is a hybrid that combines the competetive and cooperative approach. To describe the *node-state* of a node F^p at a certain point of execution we use a triple (F^p, s, r) where $s \in \{\top, \bot, ?\}$ ($\top$ indicates that an incarnation found a model for the node, whereas $\bot$ indicates that an incarnation proved unsatisfiability of F^p; finally, ? indicates that the node has not been solved yet) and $r \in \{\blacktriangleright, \blacksquare\}$ (indicating whether an incarnation is *running* on F^p or not, respectively). Given the notion of a node-state, we can easily differentiate between plain partitioning and iterative partitioning: a cooperative solver exploits the iterative partitioning strategy if two incarnations are allowed to run at the same time on nodes F^p, F^q such that $p \leq q$. Otherwise, the solver is said to be exploiting the plain partitioning strategy.

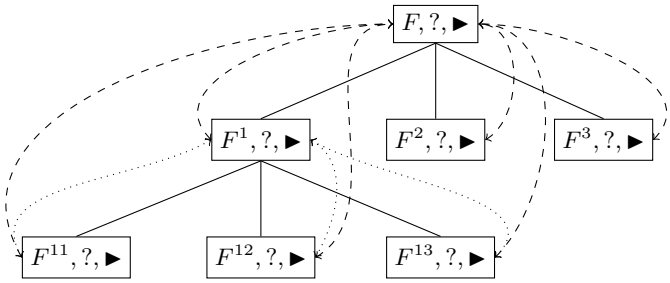

Fig. 2. Visualization of a partition tree with clause sharing and overlapped solving: The dashed lines show the possible ways flag-based sharing can send clauses. The dotted lines represent the possible additional communication when position based sharing is used. From this picture it can be seen that more clauses could be shared.

More informally, plain partitioning solves only the leaf nodes of the partition tree, whereas iterative partitioning processes all nodes in the search tree in a breadth first order.

In the authors' opinion these names are misleading since, for both strategies, the partitioning is fixed, and the difference relies instead in the order in which nodes are solved. We thus propose to rename iterative partitioning to *overlapped solving*, as it permits the parallel solver to look at overlapping partitions of the search space at the same time. Accordingly, plain partitioning will be called *non-overlapped solving*.

In each cooperative solver, in order to solve an unsatisfiable node F^p, either F^p has to be directly solved by some incarnation or each child node F^{pi} has to be solved. Hyvärinen shows in [13] that solving each of these children can be more expensive (in terms of required time) than solving the father node directly. A consequence of this fact is that overlapped solving is superior to non-overlapped solving; this claim is formally proved in [36].

4.2.3 Sharing Information Among Partitions A recent empirical study [37] shows that clause learning is the most important feature of modern SAT solvers. Most of the current portfolio solvers share learned clauses among the incarnations. This sharing of learned clauses shows improvements in portfolio solvers, because a learned clause can prune parts of a search space of the incarnation. An important question in sharing learned clauses is: which clauses are good for sharing? There is no general successful answer to this question, but different portfolio solvers use different heuristics (see Section 4.1).

In the overlapped solving, we cannot share each learned clause with every incarnation, because partitioning constraints can contribute to the learning of a clause, and so the clause cannot always be a logical consequence of partition formulas solved by other incarnations. The first approach for sharing clauses in the iterative partitioning is given in [38], this clause sharing approach is called *flag-based learned clause tagging*.

A learned clause is considered *unsafe* if it belongs to partitioning constraints, or it is obtained by a resolution derivation involving one or more unsafe clauses. A clause that is not unsafe is called *safe* clause, and only safe clauses are shared. Sharing in the overlapped solving has been further improved by *position-based clause tagging* [15]. The idea of safe and unsafe clause is extended to sub-partition trees: a learned clause is shared in a sub-partition tree if it is safe in that sub-partition tree. This information can be calculated by tagging each clause C^p with the position p of the sub-tree where C^p is valid. With position-based clause sharing an increased number of shared clauses and a better

performance when compared against non-sharing or flag-based have been reported [15]. For PCASSO, we therefore decided to exploit position-based clause sharing.

5 Analyzing and Improving Cooperative SAT Solving

In this section we will analyze design decisions of SPLITTER one by one – next, suggesting modifications to the procedure, and finally give an empirical evaluation. Therefore, we implemented the parallel search space splitting solver PCASSO, which is based on a MINISAT-style solving engine[2]. Evaluation is performed using an AMD Opteron 6274 with 2.2 GHz and 8 GB of memory. We evaluate the solvers on 8 cores. For each formula of the benchmark, we apply a wall clock timeout of 2 hours. The full benchmark to compare parallel solvers consists of 880 instances, originating from the application track of the SAT 2009 competition, as well as the SAT 2012 Challenge. Furthermore, we added the instances that have not been selected for the SAT 2012 Challenge. However, for the development of the single design improvements, we choose a subset of 118 instances where 66 could be solved by SPLITTER, 34 can be solved by PENELOPE only and the remaining instances cannot be solved by both systems. During the development phase (Section 5.1–5.3.6), we tested the performance of PCASSO only on these 118 instances. Furthermore, note that the results presented in the following paragraphs cannot be related to each other directly, since the implementation of the system grew while developing and applying the improvements. For the results presented in one table, all parameters of the solver are fixed and only a single other parameter has been modified, as we indicate in the tables[3].

5.1 Creating Partitions

In our experiments, we have observed that scattering creates partitions for a given node using cubes such that there are common variables among the cubes. We now define *tabu scattering* as an extension of scattering, by putting a restriction that a variable used in one cube, must not be used in the cubes for creating remaining partitions. Using tabu scattering, we diversify the search more.

Another observation is that scattering does not always create partitions that have equal difficulty in terms of solving time. Due to this difference, consider a scenario that the solver has some idle resources, so the solver creates partitions of some running unsolved node $(F^p, ?, \blacktriangleright)$ in the partition tree, but it may happen that $(F^p, ?, \blacktriangleright)$ is very close to find the result $\bot$ and thus the solver may waste resources on the newly created partitions. We propose a solution to decrease the chance of this scenario to happen, by sorting the child nodes in decreasing order of difficulty level; this way the solver will create partitions of more difficult node first than the less difficult nodes, avoiding the above scenario. We predict the difficulty level of a node by a simple heuristic that counts the number of propagated literals: the higher the number of top level units after propagation, the lower the estimated difficulty of the analyzed formula.

SPLITTER chooses the literals in the cubes by using VSIDS heuristic: it runs a solver for a certain number of conflicts (8196 conflicts) and picks the literals with highest VSIDS score and their saved polarity. Then, it creates the partition, adds the negated cube to the current formula and repeats the process until enough partitions are created. In PCASSO, we use lookahead techniques [39] for choosing the literals for creating partitions with

[2] The latest version uses GLUCOSE 2.2

[3] PCASSO is available at `http://tools.computational-logic.org`.

scattering, by choosing a variable with the maximum *mixdiff* score [39]. The score *mixdiff* of a variable is the product of the *diff* score of each polarity of the variable. We calculate the *diff* score of the polarity of a variable by applying lookahead, and use the following weighted sum: 0.3 times the number of propagated literals plus 0.7 times the number of newly created binary clauses. After choosing the variable with the maximum *mixdiff* score, we choose the polarity of the variable that has the lowest *diff* score for creating cubes. We also use the reasoning techniques: failed literals, necessary assignments, pure literals, and add learned clauses to the partition constraints. Techniques like constraint resolvent, double lookahead, and adaptive pre-selection heuristics are also used as proposed in the literature [39]. It is the first time that lookahead and scattering is combined for creating partitions. Previously lookahead has already been used in CUBEANDCONQUER for creating partitions, but without scattering [10,11].

	TOT	SAT	UNSAT	Wall Time	Median Time	PAR10
VSIDS	57	23	34	1319.43	7200	37857
LA	73	22	51	1135.43	1849.05	28160

The data clearly shows that combining look-ahead with scattering is superior to choosing variables by VSIDS, $37.7\,\%$‘ more instances can be solved, and the median run time drops significantly.

5.2 Solving Time Limit

SPLITTER uses a solving time limit for each node in the partition tree, proposed in [36] for grid environment. Here we propose a different strategy: we do not put any limit on the solving time for each node. To give the intuition about our idea, we first define the *ideal solving time limit:* a solving time limit is *ideal* if the given CNF formula is solved within that limit. Given an unsatisfiable CNF formula that we solve with overlapped solving, a void partition function (i.e. a partitioning that does not partition the problem in "simpler" problems) and a non-ideal solving time limit then overlapped solving slowly becomes similar to non-overlapped solving, because intermediate nodes are interrupted before they can be solved. By over-approximating the ideal limit, we overcome this problem.

NODE LIMIT	TOT	SAT	UNSAT	Wall Time	Median Time	PAR10
512000	73	22	51	1135.43	1849.05	28160
NONE	86	25	61	1177.48	1101.68	20346

When comparing solving with and without limits, it can be seen that the performance of the solver increases heavily, especially on unsatisfiable instances. Furthermore, the median run time drops, indicating that the limited solving also aborts incarnations just before they solve their node.

5.3 Diversification and Intensification

A search strategy in a modern SAT solver uses the following components: decision heuristic, polarity heuristic, restart policy, and learning scheme. *Diversification vs intensification* is a trade-off made by the search strategy. **Intensification** refers to search strategies with the goal to greedily improve the chances of finding a solution. **Diversification** strategies try to achieve a reasonable coverage of the search space. For further reading, we refer to [40].

5.3.1 Sharing VSIDS and Progress Saving We intensify the search by sharing information. First, we look into sharing VSIDS and progress saving information. Portfolio solvers do not share this information, because all incarnations start their search at the same time; but in case of our solver, we have a tree structure (partition tree) that we can exploit, and also the search of the nodes in the partition tree does not start at the same time. Thus, sharing heuristic information like VSIDS and progress saving from parent to child nodes, could help the child nodes. When PCASSO starts solving, the root node and the nodes at the partition tree level one start at almost the same time. The nodes at partition tree level greater than one are usually created after some time, so we initialize their search process with the VSIDS and progress saving information of their parent, because the child node searches in the sub-search space of its parent and whatever is learned by the parents search can help the solving child node as well.

	TOT	SAT	UNSAT	Wall Time	Median Time	PAR10
NONE	91	25	66	893.724	594.675	17163
ACTIVITY	92	26	66	867.278	508.19	16540
POLARITY	93	27	66	895.846	619.85	15960
BOTH	92	26	66	870.25	593.575	16542

When sharing information, the performance of the solver increases – however, not sharing the activity information seems to provide a slightly more powerful, but also slower..

5.3.2 Dynamic Sharing Learned clauses are shared between incarnations to intensify the search. Most parallel solvers use a static measure for sharing clauses. SPLITTER shares only binary clauses [15]. We propose a dynamic learned clause sharing scheme, that is based on LBD scores. A learned clause is eligible for sharing by an incarnation if the LBD score of this clause is lower than a parameter δ of the global LBD average of the incarnation. In PCASSO, we use $\delta = 0.5$.

SHARE LIMIT	TOT	SAT	UNSAT	Wall Time	Median Time	PAR10
DYNAMIC 0.5	95	27	68	765.479	467.98	14650
STATIC 6	94	26	68	896.848	506.575	15358

As indicated in related work already, the dynamic limit is superior with respect to the run time of the solver. However, in PCASSO the number of solved instances is not influenced much.

5.3.3 Different Restarts Portfolio solvers like MANYSAT and PENELOPE, use different restart policies for each incarnation, to diversify the search. Inspired by this idea, we diversify by using different restart policy parameters in PCASSO. As PCASSO uses GLUCOSE, the dynamic restart policy in GLUCOSE [41] can be modified to diversify the search of PCASSO as well. GLUCOSE maintains a global average of LBD scores. A restart is performed if the average LBD score of the last X learned clauses is greater than the global average times a magic constant K. First we classify the nodes in partition tree into three categories: (i) *root node:* the node at the root of the partition tree, (ii) *leaf node:* the nodes which do not have any child node, (iii) *middle node:* the node which is neither a root node nor a leaf node. According to these node categories, we apply different restart policies. The root node uses $X = 75$ and $K = 0.7$. Leaf nodes use $X = 50$ and $K = 0.8$.

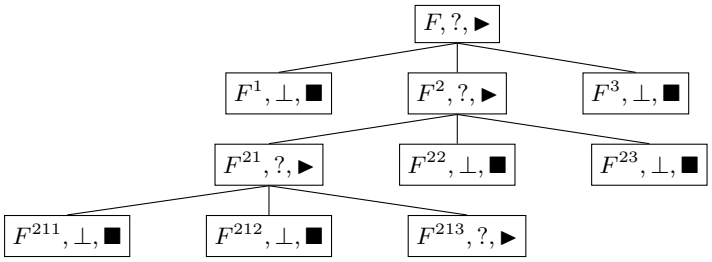

Fig. 3. The given snapshot shows the only child scenario for four computation units: for each node, only a single child is still unsolved, and all other nodes are evaluated to $\bot$.

Parent nodes use $X = 75$ and $K = 0.8$. We have selected the values of X and K based on experiments and the data provided in [41].

	TOT	SAT	UNSAT	Wall Time	Median Time	PAR10
SAME	89	27	62	792.393	673.405	18292
DIFFERENT	92	26	66	870.25	593.575	16542

As provided by the empirical data, diversifying restart strategies helps to improve the performance on unsatisfiable instances. Furthermore, the solving time improves.

5.3.4 Different Learnt Clauses Cleaning To diversify, PENELOPE uses different intervals between cleaning learned clauses for different incarnations. The purpose is that some incarnations keep learned clauses for a longer time than others. We introduce this idea in our solver, according to the node category as well. We give different cleaning intervals to the root node, middle nodes, and leaf nodes. Let Int_{root}, Int_{middle}, Int_{leaf} be the cleaning intervals of the nodes root, parent, and leaf, respectively. Then we have the following relationship: $Int_{root} \geqslant Int_{middle} \geqslant Int_{leaf}$. Note that a leaf node changes its cleaning policy dynamically when it becomes a middle node.

CLEAN INTERVAL	TOT	SAT	UNSAT	Wall Time	Median Time	PAR10
NONE	94	27	67	950.416	608.98	15401
DECREASE	93	24	69	1099.12	656.48	16120
INCREASE	92	26	66	870.25	593.575	16542

The default cleaning policy of GLUCOSE seems to be chosen well - however, when increasing the interval, the median run time can be improved slightly. Surprisingly, the number of solved unsatisfiable instances can increase when using the modifications, whereas the number of satisfiable instances decreases.

5.3.5 Only Child Scenario During our experiments we have observed, on some instances, that the height of the partition tree grows until it hits the number of parallel resources. This means that there is only one unsolved node at each partition level of the partition tree. On a smaller scale, there could be only one unsolved node at some partition level. For that reason, we call this scenario the *only child scenario*. Figure 3 shows an extreme case of only child scenario for a solver with four available resources. You can see that only one node is unsolved at each level of the partition tree, i.e. the nodes solving the partitions F, F^2, F^{21}, F^{213} are unsolved and running.

Consider that only child scenario happens at some level of the partition tree, then there are two cases: i) the parent node is looking into the search space which has been solved by one of its children already, ii) the parent node is looking into the same search space where its unsolved children are looking. In either case, we have the risk of doing redundant work. We propose a approach to get out of this scenario by reintroducing the solving limit in a node that has only one unsolved child (AVOID). To be on safe side, we do not apply this limit for the root node. The introduced limit grows with the level of the node (level $*$ 4096 conflicts). Since in the only child scenario all learned clauses can be shared among the two participating nodes, we can also EXPLOIT this situation, by enabling this sharing. In the extreme case, this configuration is very similar to portfolio solvers, since then all clauses can be shared without restrictions.

	TOT	SAT	UNSAT	Wall Time	Median Time	PAR10
NONE	93	27	66	979.23	726.09	16026
AVOID	92	26	66	870.25	593.575	16542
EXPLOIT	91	26	65	856.31	607.335	17134

Although ignoring the scenario yields the best results wrt. solved instances, the run time can be reduced significantly when either exploiting or avoiding the scenario, where avoiding the scenario yields the better speedup.

5.3.6 Conflict Driven Node Killing When clauses are tagged by position-based tagging [15] as described above, additional information can be obtained by performing a conflict analysis on solved unsatisfiable nodes. Consider a node $(F^p, \bot, \blacksquare)$, and let $\{\}^q$ be the empty clause labeled with position q, which was derived by the incarnation that solved F^p. Then, from the main theorem in [15], we conclude that $\{\}^q$ is a semantic consequence of the node of position q in the partition tree. Observe that q is a prefix p: $q \leq p$. Consequently, not only the node at position p can be marked as unsatisfiable, but also the node F^q as well as all its child nodes. As a result, more incarnations can terminated and start solving different partitions. We call this kind of technique *conflict driven node killing*. A similar approach is reported in [38] with *assumption-based* clause tagging, but the author did not report benefits from exploiting this technique. Instead, our tests show an improved performance in terms of the number of solved instances. However, similarly to [38], we observe an increase of the median time. A reason for this might be the fact that by stopping a node we prevent it from producing more shared clauses for its level and above levels.

	TOT	SAT	UNSAT	Wall Time	Median Time	PAR10
CKILL	92	26	66	870.25	593.575	16542
NOCKILL	89	24	65	795.058	498.435	18294

5.4 Evaluation

We evaluate PLINGELING, PENELOPE, SPLITTER and PCASSO on the full benchmark of 880 instances, allowing each solver to use eight cores and a time limit of two hours. The final configuration of PCASSO uses the following options: (i) tabu lookahead for creating partitions, (ii) no solving limit, (iii) sharing vsids+polarity, (iv) dynamic clause sharing 0.3, (v) different restarts, (vi) decrease clause cleaning interval w.r.t. level, (vii) exploiting the only child scenario by simulating portfolio, and finally (viii) conflict

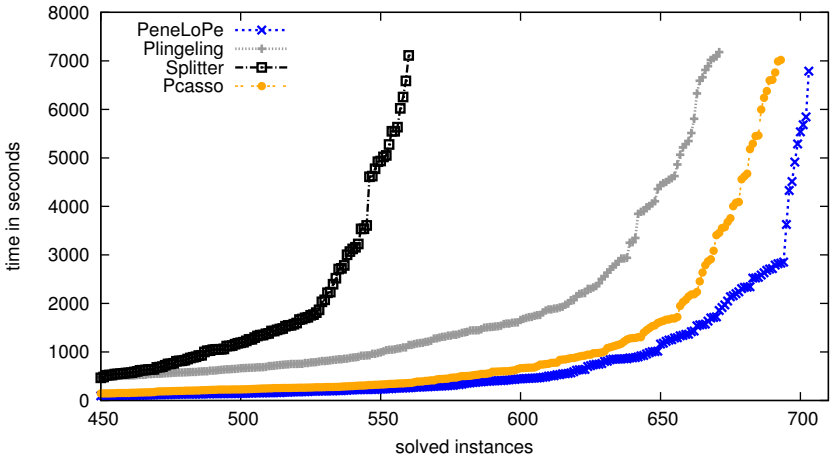

Fig. 4. Comparing the performance of parallel state-of-the-art SAT solvers

driven node killing. The result is presented in the cactus plot in Figure 4. With this data, we can validate our claim posted in the introduction: search space partitioning solvers have to be considered being state of the art! The new solver PCASSO (696 instances) outperforms PLINGELING (672 instances) on the benchmark, and is very close to PENELOPE (704 instances). Furthermore, the median run time on all instances of PCASSO (136.17 seconds) is close to PENELOPE (89.39 seconds). The contribution is not due to a single improvement, but rather due to the sum of all improvements. Adding only a single improvement of the above sections would not result in the presented performance of solving 135 more instances than the latest search space splitting solver SPLITTER.

6 Conclusion

We started with the search space splitting solver SPLITTER, which shows a poor performance especially on unsatisfiable instances compared to other parallel state-of-the-art SAT solvers, such as PLINGELING or PENELOPE. We then analyzed the design decisions of SPLITTER and proposed improvements, where some of them resulted in major improvements already. For example, we improved the partitioning by adding tabu information and combining look-ahead and scattering, increasing the number of solved instances by 37.7 %, or we removed the solving limit per partition, which improves the performance by another 17.8 %. By improved information sharing about learned clauses, or the variable phase, as well as diversifying the solver incarnations, or analyzing the unsat result for partitions, we could improve the system further. Finally, we pointed out special situations where iterative search space partitioning solvers come very close to portfolio solvers - a situation we call the *only child scenario*. We implemented the improved algorithms into the solver PCASSO, resulting in a state-of-the-art parallel SAT solver that does not rely on search space splitting. Depending on how the only child scenario is handled, this solver can also simulate portfolio systems.

By comparing the performance of PCASSO and PENELOPE, which solve a similar amount of instances, and combining this empirical result with the theoretical and empirical results of [13], we are convinced that search space partitioning is a good candidate for solving SAT on more parallel computing architectures. As future work, we suggest to

incorporate formula simplifications into the search engine, as well as using other recently proposed additions like clause freezing. However, applying modifications to search space splitting solvers will be more research intense than for portfolio systems, because any modification of the solver incarnations has to consider the partitioning constraints.

Acknowledgement We thank ZIH of TU Dresden for providing generous capacity of parallel computation resources.

References

1. Rossinelli, D., Bergdorf, M., Cottet, G.H., Koumoutsakos, P.: Gpu accelerated simulations of bluff body flows using vortex particle methods. J. Comput. Phys. **229**(9) (2010) 3316–3333
2. Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Buijssen, S.H.M., Grajewski, M., Turek, S.: Exploring weak scalability for FEM calculations on a GPU–enhanced cluster. Parallel Computing **33**(10–11) (2007) 685–699
3. Cook, S.A.: The complexity of theorem-proving procedures. In: Proc. of the third annual ACM symposium on Theory of computing. STOC '71 (1971) 151–158
4. Feier, M.C., Lemnaru, C., Potolea, R.: Solving np-complete problems on the cuda architecture using genetic algorithms. In: Proc. of the 2011 10th International Symposium on Parallel and Distributed Computing. ISPDC '11 (2011) 278–281
5. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. JSAT **6**(4) (2009) 245–262
6. Guo, L., Hamadi, Y., Jabbour, S., Sais, L.: Diversification and intensification in parallel SAT solving. In: Proc. of the 16th international Conference on Principles and Practice of Constraint Programming. CP'10 (2010) 252–265
7. Audemard, G., Hoessen, B., Jabbour, S., Lagniez, J.M., Piette, C.: Revisiting clause exchange in parallel SAT solving. In: Fifteenth International Conference on Theory and Applications of Satisfiability Testing(SAT'12). (2012) 200–213
8. Biere, A.: Lingeling and friends entering the sat challenge 2012. [42] 33–34
9. Roussel, O.: Description of ppfolio 2012. [42] 46
10. Heule, M.J., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Accepted for HVC 2011. (2012)
11. van der Tak, P., Heule, M.J.H., Biere, A.: Concurrent cube-and-conquer. In: Proc. of the 15th International Conference on Theory and Applications of Satisfiability Testing. SAT'12 (2012) 475–476
12. Hyvärinen, A.E.J., Manthey, N.: Splitter – a scalable parallel SAT solver based on iterative partitioning. [42] 62
13. Hyvärinen, A.E.J., Manthey, N.: Designing scalable parallel SAT solvers. In Cimatti, A., Sebastiani, R., eds.: SAT. Volume 7317 of LNCS., Springer (2012) 214–227
14. Järvisalo, M., Le Berre, D., Roussel, O., Simon, L.: The international SAT solver competitions. AI Magazine **33**(1) (2012) 89–92
15. Lanti, D., Manthey, N.: Sharing information in parallel search with search space partitioning. In: Proc. of the 7th International Conference on Learning and Intelligent Optimization (LION 7). LNCS, Springer (2013)
16. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (2009)
17. Marques-Silva, J.P., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Trans. Computers **48**(5) (1999) 506–521
18. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7) (1962) 394–397
19. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. Design Automation Conference (2001) 530–535
20. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Proc. of the 10th International Conference on Theory and Applications of Satisfiability Testing. SAT'07, Springer-Verlag (2007) 294–299

21. Eén, N., Sörensson, N.: An extensible sat-solver. In Giunchiglia, E., Tacchella, A., eds.: SAT. Volume 2919 of LNCS., Springer (2003) 502–518
22. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: Proc. of the 21st international jont conference on Artifical intelligence. IJCAI'09, Morgan Kaufmann Publishers Inc. (2009) 399–404
23. Gomes, C.P., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisf. and constraint satisfaction problems. J. Autom. Reason. **24**(1-2) (2000) 67–100
24. Biere, A.: Picosat essentials. JSAT **4**(2-4) (2008) 75–97
25. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Information Processing Letters **47** (1993) 173–180
26. Audemard, G., Simon, L.: Glucose 2.1: Aggressive, but reactive, clause database management, dynamic restarts (sys. descr.). In: Pragmatics of SAT 2012. (2012)
27. Audemard, G., Simon, L.: Refining restarts strategies for SAT and UNSAT. In: Proc. of the 18th International Conference on Principles and Practice of Constraint Programming. CP'12, Springer-Verlag (2012) 118–126
28. Marques-Silva, J.P., Lynce, I., Malik, S.: 4. [16] 131–153
29. Hölldobler, S., Manthey, N., Nguyen, V., Stecklina, J., Steinke, P.: A short overview on modern parallel SAT-solvers. In Wasito, I., Hasibuan, Z., Suhartanto, H., eds.: Proc. of the International Conference on Advanced Computer Science and Information Systems. (2001) 201–206 ISBN 978-979-1421-11-9.
30. Martins, R., Manquinho, V., Lynce, I.: An overview of parallel SAT solving. Constraints **17**(3) (2012) 304–347
31. Wotzlaw, A., van der Grinten, A., Speckenmeyer, E., Porschen, S.: pfolioUZK: Solver description (2012)
32. Audemard, G., Hoessen, B., Jabbour, S., Lagniez, J.M., Piette, C.: Penelope, a parallel clause-freezer solver. In: Proc. of SAT Challenge 2012: Solver and Benchmarks Descriptions. (2012) 43–44
33. Hamadi, Y., Jabbour, S., Sais, L.: Control-based clause sharing in parallel sat solving. In: Proc. of the 21st International Joint Conference on Artifical intelligence. IJCAI'09, Morgan Kaufmann Publishers Inc. (2009) 499–504
34. Manthey, N., Philipp, T., Wernhard, C.: Soundness of inprocessing in clause sharing SAT solvers. In: Proc. 16th International Conference on Theory and Applications of Satisfiability Testing. SAT'13 (2013, accepted.)
35. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: A distribution method for solving SAT in grids. In: Proc. of the 9th International Conference on Theory and Applications of Satisfiability Testing. SAT'06, Springer-Verlag (2006) 430–435
36. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Partitioning SAT instances for distributed solving. In: Proc. of the 17th International Conference on Logic for programming, artificial intelligence, and reasoning. LPAR'10 (2010) 372–386
37. Katebi, H., Sakallah, K.A., Marques-Silva, J.P.: Empirical study of the anatomy of modern sat solvers. In: Proc. of the 14th International Conference on Theory and Application of Satisfiability Testing. SAT'11, Springer-Verlag (2011) 343–356
38. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Grid-based SAT solving with iterative partitioning and clause learning. In: Proc. of the 17th International Conference on Principles and Practice of Constraint Programming. CP'11 (2011) 385–399
39. Heule, M.J.H., van Maaren, H.: 5. [16] 155–184
40. Hoos, H.H., Stützle, T.: Stochastic Local Search: Foundations & Applications. Elsevier / Morgan Kaufmann (2004)
41. Audemard, G., Simon, L.: Refining restarts strategies for sat and unsat. In: 18th International Conference on Principles and Practice of Constraint Programming (CP'12). (2012) 118–126
42. Balint, A., Belov, A., Diepold, D., Gerber, S., Järvisalo, M., Sinz, C., eds.: Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions. (Volume B-2012-2 of Department of Computer Science Series of Publications B.)