Scalable SAT Solving in the Cloud

Dominik Schreiber[0000−0002−4185−1851] and Peter Sanders[0000−0003−3330−9349]

Karlsruhe Institute of Technology
{dominik.schreiber,sanders}@kit.edu

Abstract. Previous efforts on making Satisfiability (SAT) solving fit for high performance computing (HPC) have led to super-linear speedups on particular formulae, but for most inputs cannot make efficient use of a large number of processors. Moreover, long latencies (minutes to days) of job scheduling make large-scale SAT solving on demand impractical for most applications. We address both issues with *Mallob*, a framework for job scheduling in the context of SAT solving which exploits *malleability*, i.e., the ability to add or remove processing power from a job during its computation. Mallob includes a massively parallel, distributed, and malleable SAT solving engine based on HordeSat with a more succinct and communication-efficient approach to clause sharing and numerous further improvements over its precursor. Experiments with up to 2560 cores show that Mallob outperforms an improved version of HordeSat and scales significantly better. Moreover, Mallob can solve many formulae in parallel while dynamically adapting the assigned resources, and jobs arriving in the system are usually initiated within a fraction of a second.

Keywords: Parallel SAT solving · Distributed computing · Malleable load balancing.

1 Introduction

Today's applications of SAT solving are manifold and include areas such as cryptography [26], formal software verification [23], and automated planning [30]. Application-specific SAT encoders generate formulae which represent the problem at hand stated in propositional logic. Oftentimes, multiple formulae which represent different aspects or horizons of the problem are generated [23,30]. The individual formulae range from trivial to extremely difficult, and their difficulty is usually not known beforehand. Up to a certain degree, today's high performance computing (HPC) can facilitate the resolution of difficult problems. In particular, we notice increased interest in performing SAT solving in on-demand HPC environments that are often referred to as *cloud* [15,29]. This is also reflected in the International SAT Competition 2020 featuring a cloud track for the first time [11]. However, prior achievements of super-linear speedups for particular application instances [4] must be set in relation with the total work which must be invested in every single formula to achieve such peak speedups. Furthermore, in most HPC systems, long latencies of job scheduling (ranging from minutes to days) hinder the quick resolution of a stream of jobs even if most

of the jobs are trivial. To address these issues, we believe that a SAT solver tasked with a formula of unknown difficulty should be allotted a flexible amount of computational resources based on the overall system load and further task-dependent parameters. In the context of scheduling and load balancing, this feature is called *malleability*: The ability of an algorithm to deal with a varying number of processing elements during its execution [10]. Malleable algorithms open up opportunities for highly dynamic load balancing techniques: The number of associated processing elements for each job can be adjusted continuously to warrant optimal and fair usage of available system resources [19].

In this work, we present a new framework for the scalable resolution of SAT jobs on demand. Our system named *Mallob* consists of two major contributions. First, we propose a decentralized approach to malleable job scheduling and load balancing in the context of SAT solving. Secondly, we present a distributed and malleable SAT solving engine based on the popular large-scale solver HordeSat [4]: Most notably, we introduce a succinct and communication-efficient clause exchange mechanism, adapt HordeSat's solver backend to handle malleability, and integrate a number of performance improvements. Experiments with up to 128 compute nodes (2560 cores) show that Mallob as a standalone SAT solver clearly outperforms an updated and improved version of HordeSat and scales significantly better. Moreover, Mallob can solve many formulae in parallel with minimal overhead and combines parallel job processing with a flexible degree of parallel SAT solving to make best use of the available resources. In most cases, it only takes a split second until an arriving job is initiated.

After describing important preliminaries and related work in Section 2, we present the malleable environment which hosts our solver engine in Section 3. Thereupon, in Section 4 we present the solver engine itself. We present the evaluation of our system in Section 5 and conclude our work in Section 6.

2 Related Work

Given a propositional formula F, the *SAT problem* is to find an assignment to all variables in F such that F is satisfied, or to report that no such assignment exists. For the sequential resolution of SAT problems, the most commonly used algorithm is *CDCL* [25], which is essentially a highly engineered heuristic depth-first search over the space of partial variable assignments. CDCL features advanced techniques such as non-chronological backtracking and restart mechanisms. Furthermore, when a logical conflict is encountered, the solver *learns* a clause which represents this conflict. The knowledge gained from this learning mechanism can help to speed up the subsequent search. Another branch of notable sequential SAT solving approaches is the family of *local search solvers* which perform stochastic local search over the space of variable assignments [18].

Parallel SAT solvers commonly use sequential SAT solvers as building blocks. One strategy which is often called the *portfolio approach* is to execute several solvers in parallel on the same formula [1,14]. Diversification strategies for an effective portfolio range from supplying different random seeds to the same

solver over reconfiguring the solver's parameters to employing wholly different SAT solvers. As an alternative to portfolio approaches, *search space partitioning approaches* subdivide the original formula into several sub-formulae and solve these in parallel [2,31]. An extreme case of this strategy is applied in parallel *Cube&Conquer* approaches where a large number of subproblems is generated and then distributed among all workers [15,17]. Regardless of the means of parallelization, an important feature of parallel solvers is to exchange learnt clauses among all workers and, notably, to find a good tradeoff between the sharing of useful information and the avoidance of unnecessary overhead [9].

The International SAT Competition 2020 [11] established a distinction between modestly parallel SAT solving and high-performance SAT solving by featuring both a parallel track and a cloud track. In the parallel track, a single 32-core node was employed for up to 5000 s per instance while the cloud track was evaluated on 100 8-core nodes for up to 1000 s per instance. These different modes of operation require different solver architectures: For modest parallelism in shared memory, high concurrency and memory consumption can become a considerable issue [20]. On a larger scale, concurrency can be less of an issue while good diversification and communication efficiency becomes critical. HordeSat [4] is a popular solver designed for massive parallelism which served as a baseline in the mentioned cloud track. It features a modular solver interface which allows to plug in and dynamically diversify different core solvers. Clause exchange is performed periodically via all-to-all collective operations. The HordeSat paradigm found adoption in a generic interface for parallel SAT solving [24].

Previously, a distributed system for SAT solving in the cloud was presented in [28,29]. It features a centralized scheduler which precomputes a schedule based on run time predictions and which employs sequential solvers without any communication among them: The authors noted that "*such solutions [for exchange of knowledge] are not necessarily suitable for distributed clouds in which the communication time could be important*" [29]. In contrast, we demonstrate that clause exchange is highly effective and introduce decentralized dynamic load balancing without any run time predictions. Another work related to ours is the distributed Cube&Conquer solver Paracooba [15] which can also resolve multiple jobs in parallel and also performs a kind of malleable load balancing. While Paracooba is designed for Cube&Conquer, we propose a malleable portfolio approach. In the cloud track of the SAT Competition 2020 [11], our system outperformed Paracooba and scored a clear first place.

3 Malleable Environment

We now outline the platform *Mallob* for the scheduling and load balancing of malleable jobs. Mallob is an acronym for **Mal**leable **Lo**ad **B**alancer as well as **M**ulti-tasking **A**gi**l**e **Lo**gic **B**lackbox. As a comprehensive presentation of Mallob in its entirety is too broad in scope for this publication, we present the design decisions and the features of Mallob that are necessary to understand our SAT

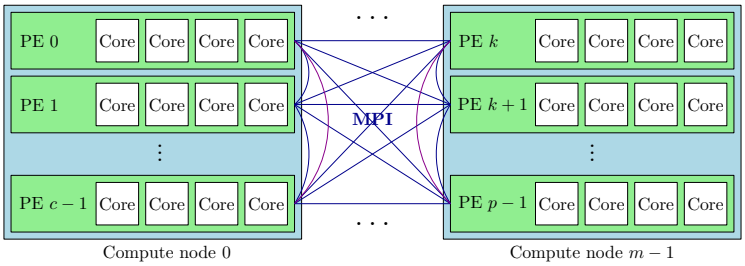


Fig. 1. System architecture used by Mallob

solving system and will describe the internal workings and theoretical properties of our scheduling and load balancing in a future publication.

We consider a homogeneous[1] distributed computing environment with m *compute nodes* (see Fig. 1). For the sake of generality, we do not assume any kind of shared (RAM or disk) memory between the nodes. As such, the only way for the nodes to exchange information is to send messages over some broadband interface. This is enabled by the Message Passing Interface (MPI) [13].

Each compute node contains several *cores*. We partition the cores on a node into c *groups* of t cores each running one *thread*.[2] Each group is implemented as a *process* and is also called *PE* (for *processing element*) in the following. Overall, our system contains a total of $p := c \cdot m$ PEs and $c \cdot m \cdot t$ parallel threads.

A number of *jobs* $1, \ldots, n$ arrive in the system at arbitrary times. A job is a particular problem statement, in our case given by a propositional logic formula in *Conjunctive Normal Form* (CNF). Every job j has a constant *priority* $\pi_j \in (0, 1)$ and a *demand of resources* $d_j \in \mathbb{N}$ which may vary over time. In the most simple setting, $d_j = p$ at all times. More generally, a job can express with d_j how many PEs it is able to employ in its current stage of computation. We expect the number of active jobs to be smaller than the number of workers, which allows us to restrict each PE to compute on at most one job at a time.

If a job j enters the system, a *request message* $r_0(j)$ performs a random walk through a sparse regular graph over all PEs until an idle PE $p_0(j)$, named the *root* of j, adopts the job. This root remains unchanged throughout the job's lifetime and represents j in collective load balancing computations. Such a balancing computation is triggered at most once within a certain period e (e.g., $e = 0.1s$) by (a) the arrival of a new job, (b) the completion of a job, and/or (c) the change of a job's demand. All such events are then broadcast globally with a single lightweight collective operation. The result of each balancing is a map $j \mapsto v_j$ which assigns to each job j a certain integer, the *volume* $v_j \geq 0$. v_j

[1] While we intend to generalize our system to heterogeneous environments in the future, this undertaking is out of scope for this publication.

[2] The cores may be distributed over several CPU chips (or sockets). Moreover, each core may be able to run several hardware threads. Our system can handle both additional levels of hierarchy by appropriately defining c and t.

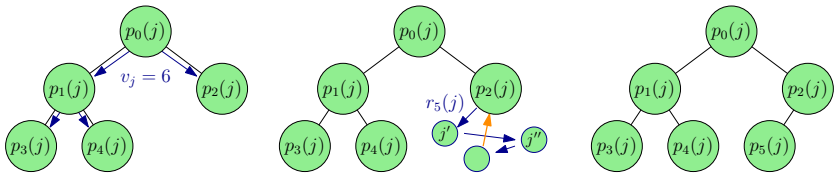


Fig. 2. Illustration of T_j growing from volume 5 to 6. Each circle is a PE.

is proportional to $d_j p_j / \sum_{j'} d_{j'} p_{j'}$ and determines the number of PEs which participate in the resolution of j until the next update of v_j.

The *job tree* T_j of job j is a binary tree of PEs that is rooted at $p_0(j)$. Its purpose is to enforce the volume assigned to j and to enable efficient job-internal communication. Each node $p_x(j)$ in T_j has a unique index $x \geq 0$. Node $p_x(j)$ may have child nodes $p_{2x+1}(j)$ (left child) and $p_{2x+2}(j)$ (right child). T_j is supposed to consist of exactly v_j nodes $p_0(j), \ldots, p_{v_j-1}(j)$ and adjusts accordingly whenever v_j updates: Beginning from $p_0(j)$ which computes a new value of v_j, a message containing v_j is sent through T_j as shown in Fig. 2. If this update arrives at a node $p_x(j)$ for which $x \geq v_j$, then the node will leave T_j and suspend its computation. Conversely, if $p_x(j)$ does not have a left (right) child node and if $2x + 1 < v_j$ ($2x+2 < v_j$), it will send out a request $r_{2x+1}(j)$ ($r_{2x+2}(j)$) for another idle PE to join T_j. These messages are first routed over any former children of $p_x(j)$ before they begin a random walk. As such, our node allocation strategy prioritizes PEs which may still host suspended job nodes of j. In order to make careful use of main memory, we allow each PE to host a small constant number of job nodes and let it discard the oldest job nodes if this limit is exceeded.

Mallob also features a special mode for the isolated resolution of a single job: After a binary tree broadcast of the job description, the i-th PE assumes the role of $p_i(j)$, and no further load balancing is required. As such, Mallob can be employed as a conventional distributed solver without any noticeable overhead compared to static distributed solver architectures such as HordeSat's.

4 The Mallob SAT Engine

We now present our massively parallel, distributed, and malleable SAT solving engine. We focus on (1) a succinct and communication-efficient clause exchange which supports malleability; (2) a rework of HordeSat's solver backend to support malleability; and (3) practical optimizations and performance improvements.

4.1 Succinct Clause Exchange

HordeSat uses synchronous communication in *rounds* to periodically perform an all-to-all clause exchange. The used collective operation is called an *all-gather*: Each PE i contributes a buffer b_i of fixed size β. The concatenation of all buffers, $B := b_1 \circ \ldots \circ b_p$, is then broadcast to each PE. This all-gather operation is

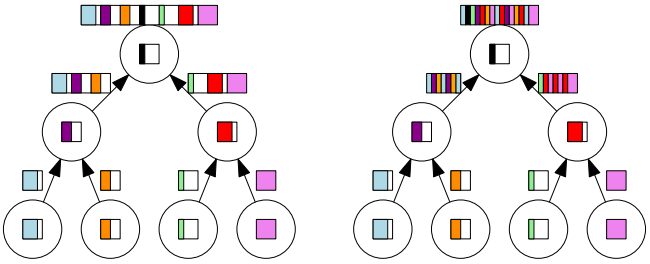


Fig. 3. Exemplary flow of information in the first half of HordeSat's all-gather operation (left) and in our aggregation within a job tree (right). Each circle is a PE; a buffer within a circle represents the PE's locally collected (*exported*) clauses.

included by default in all MPI implementations. Each b_i contains a list of learned clauses which were previously *exported* by the solvers of PE i. The clauses are serialized in a compact shape, sorted by their size in increasing order. After the all-gather, each solver *imports* clauses from B into its individual database.

We noticed that the above clause exchange mechanism has various shortcomings. First, whenever a PE does not fill b_i, B contains "holes" which carry no information (see Fig. 3). Secondly, B may contain duplicates: In particular in the beginning of SAT solving when a formula is simplified and basic propagations are done, this may lead to p almost identical buffers b_i. This effect is especially pronounced for unit clauses (see below). Thirdly, B grows proportionally to the number of involved PEs. For sufficiently large HordeSat configurations, this can constitute a bottleneck in terms of communication volume *and* local work.

In our system, we use job tree T_j (as described in Section 3) as the communication structure for the clause exchange of each job j. As such, we ensure that the PEs involved in a clause exchange are exactly the PEs that are currently associated with j. As soon as a fixed amount of time s has passed since the last broadcast of shared clauses (e.g., $s = 1$ second), each leaf $p_x(j)$ in T_j sends b_x to its parent. When an inner node $p_x(j)$ has received a buffer from each of its children, it exports its own clauses b_x and then performs a two- or three-way merge of the present buffers: All buffers are read simultaneously from left to right and aggregated into a single new buffer b'_x, similar to textbook k-way merge of sorted sequences [27, 5.7.1]. In addition, we use a hash set of seen clauses with hashing that is invariant to the order of literals [4] in order to recognize duplicates.

The size of b'_x is limited and any remaining unread information in the input buffers is discarded. As each b_i is sorted in increasing order by clause length, we aggregate some of the globally shortest clauses while we strictly limit the overall communication volume. Furthermore, we improve the density of useful information in B because each intermediate buffer is compact and contains no duplicate clauses. We limit the size of b'_x as follows: For each aggregation step, i.e., for each further level of T_j that is reached, we discount the maximum buffer size by a factor of α. Specifically, we compute the buffer size limit $l(u) := \lceil u \cdot \alpha^{\log_2(u)} \cdot \beta \rceil$ where u is the number of individual buffers b_i aggregated so far. This

limit can be steered by a user parameter $\alpha \in [\frac{1}{2}, 1]$, the *discount factor* at each buffer aggregation. We can see that $l(u)$ converges to β for $\alpha = \frac{1}{2}$ and grows indefinitely for $\alpha > \frac{1}{2}$ with respect to the number u of involved PEs. For $\alpha = 1$, $l(u)$ grows proportionally in u just like HordeSat's shared clause buffer.

HordeSat employs *clause filtering* to detect and discard redundant clauses which have already been imported or exported before. This technique is realized with an approximate membership query (AMQ) data structure. Each PE employs one *node filter* f_n and t *solver filters* (one for each solver thread). At clause export, each clause is registered in its solver filter and then tested against f_n. At clause import, each clause is tested against f_n and then against each solver filter. Unit clauses, however, are always admitted due to their high importance. This is problematic because particular unit clauses can be sent around many times and can waste a considerable amount of space in the buffers.

In our approach we omit f_n because its main use is to filter duplicate clauses which Mallob already detects during the aggregation of buffers. We complemented the solver filters with an additional filtering of unit clauses, using an exact set instead of an AMQ data structure. This way no false positives occur for unit clauses, and each such clause is shared once. We also implemented a probabilistic "restart" mechanism for clause filters: Every X seconds, half of all clauses (chosen randomly) in each clause filter are forgotten and therefore can be shared again. This allows solvers to eventually learn crucial clauses even if they join T_j after these clauses have already been shared for the first time.

4.2 Malleable Solver Backend

In the following we present the most relevant changes we made to HordeSat's solver backend to support malleability.

Malleable Diversification. As in HordeSat, our approach relies on three different sources of diversification: Employing different solver configurations, handing different random seeds to the solvers, and supplying each solver with different default polarities (*phases*) of variables. We diversify a particular solver S with a *diversification index* $x_S \geq 0$ and a *diversification seed* σ_S. We use x_S to determine a particular solver configuration and we use σ_S as a random seed and to select random variable phases. The i-th solver S ($0 \leq i < t$) employed by $p_k(j)$ is assigned $x_S := kt+i$. We obtain σ_S by combining x_S with the solver's thread ID (given by the operating system). As such, each instantiated solver is diversified differently even if a job node is rescheduled and a solver S' is instantiated for which some solver S with $x_S = x_{S'}$ already existed before.

Preemption of Solvers. In our malleable environment, it is essential that a PE's main thread can suspend, resume, and terminate each job node at will. We noticed that we cannot reliably notify a solver thread to stop or suspend its execution because it can get stuck in expensive preprocessing and inprocessing [6] for an extended period. Furthermore, it is impossible to forcefully abort a

thread without terminating or, otherwise, potentially corrupting its surrounding process. To still enable seamless preemption and termination, we enabled our solver engine to be launched in a separate process. While this involves some overhead, suspension and termination of a process is supported on the OS level in a safe and elegant manner through signals. For instance, a PE's main thread can terminate a job node by sending "SIGTERM" to the solver process, which then exits immediately regardless of the state of its solver threads.

4.3 Performance Improvements

We now present some further improvements of Mallob over HordeSat.

Solver Portfolio. HordeSat originally featured solver interfaces to Lingeling and Minisat. However, the clause import in HordeSat's Minisat interface treats shared clauses just like original, irredundant clauses and periodically interrupts each solver to add these clauses, what we believe to be detrimental to its performance. Therefore, for this work we focus on Lingeling as an efficient and reliable SAT solver with great diversification options and a dedicated clause import and export mechanism. We updated Lingeling from its 2014 version [5] to its 2018 version [7] with the side effect of rendering all core modules of our system Free Software. Similarly, instead of the 16 diversification options from the former Plingeling [5], we use 13 CDCL diversification options from the newer Plingeling [7]. Every fourteenth solver thread now uses local search solver YalSAT (included in the Lingeling interface), alternatingly with and without preprocessing.

Lock-free Clause Import. For each solver S within a PE, HordeSat's main thread copies all admitted clauses from clause sharing into a buffer B_S, increasing its size as necessary. The solver thread of S then imports the clauses in B_S one by one. As this implies concurrent access to B_S, a mutually exclusive lock is acquired by the solver thread before reading clauses and by the main thread before writing clauses. If the solver thread cannot acquire this lock, it gives up on importing a clause. We replaced B_S with a lock-free ring buffer[3] R_S and hence achieve a lock-free import of clauses. We also make more careful use of the available memory: The size of R_S is fixed and clauses are eventually discarded if a solver consumes no clauses for some time. We set $|R_S|$ to a low multiple of the maximum number of literals which may be shared in a single round.

Memory Usage. The memory consumption of parallel SAT solvers is a known issue [20]: As each solver commonly maintains its own clause database, memory requirements increase proportionally with the number of spawned solvers. As such, large formulae can cause out-of-memory errors. To counteract this issue, we introduce a simple but effective step of precaution: For a given threshold $\hat{s}$, if a given serialized formula description has size $s > \hat{s}$, then only $t' = \max\{1, \lfloor t \cdot \hat{s}/s \rfloor\}$

[3] https://github.com/rmind/ringbuf

threads will be spawned for each PE. The choice of $\hat{s}$ depends on the amount of available main memory per PE. Based on monitoring the memory usage for different large formulae within a run where 3.2 GB were available per solver, we use $\hat{s} := 10^8$. As t' only depends on s, the t' threads can be started immediately upon the arrival of a formula without the need for any further inspection.

5 Evaluation

We now turn to the evaluation of our work. After explaining our setup, we first evaluate the capabilities of our standalone SAT solver engine, denoted *Mallob-mono*. We then evaluate Mallob with malleable job scheduling.

We implemented Mallob in `C++17` and make use of OpenMPI [12]. Our software, all experimental data with supplementary material, and an interactive visualization of experiments can be found at `github.com/domschrei/mallob`.

We experimentally compare Mallob to HordeSat, both with its original portfolio and with the updated portfolio that Mallob uses. As HordeSat does not necessarily represent the state-of-the-art in distributed SAT solving [2], we refer to the SAT Competition 2020 [11] as well as the upcoming SAT Competition 2021 for state-of-the-art comparisons involving Mallob. We fixed a significant performance bug to make HordeSat more competitive: In its original code, Lingeling was not given a callback providing the elapsed time since program start. This caused each solver thread to fall back to frequent expensive system calls.

We ran most experiments on the ForHLR phase II, an HPC cluster with 1152 compute nodes with two 10-core Intel Xeon E5-2660 v3 processors and 64 GB of main memory (RAM) each, connected by an InfiniBand 4X EDR interconnection. In addition, we ran some experiments on SuperMUC-NG, a supercomputer which features 6336 compute nodes with a 24-core Xeon Platinum 8174 processor and 96 GB of DDR4 RAM each and an OmniPath network interconnection. We used the operating system Red Hat Enterprise Linux (RHEL) 7.x on ForHLR II and SUSE Linux Enterprise Server (SLES) 12.x on SuperMUC-NG.

We limited most runs to 300 seconds per instance. As such, the CPU time per instance at our largest configuration of 2560 cores is at 213 core hours (ch), similar in scale to the 222 ch per instance in the SAT Competition's cloud track. At the next smaller scale of 640 cores, 300 seconds translate to 53 ch which is similar in scale to the 44 ch per instance in the competition's parallel track.

5.1 Selection of Benchmarks

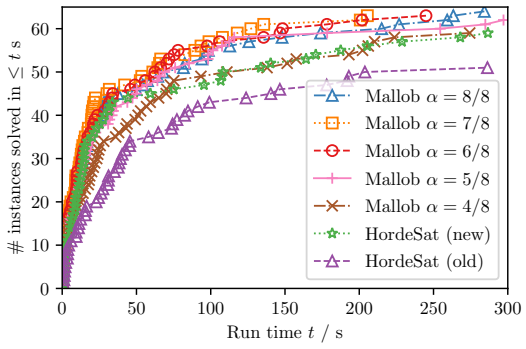
As the usage of HPC environments is costly in terms of money and energy, we aimed to run experiments responsibly and resource-efficiently while still ensuring statistical relevance and robustness of results. For this means we analyzed the 400 benchmarks of the SAT Competition 2020 with GBD [21] and partitioned them into 80 separate *families* (including families from past competitions). We sorted the instances of each family by the number of contained clauses and then randomly picked one SAT instance from the second (larger) half of each family's

sorted instance list. As such, we obtained a selection of 80 instances (35 satisfiable, 35 unsatisfiable, 10 "unknown"). We then compared the official rankings of the SAT Competition 2020 [11] with rankings resulting from our selection of benchmarks. In the cloud track, our selection of benchmarks reproduces the exact same ranking of solvers. In the parallel track, we computed a Kendall rank correlation coefficient [22] of $\tau = 0.82$ over all non-disqualified submissions: 41 pairs of solvers were ranked consistently while four pairs were ranked differently. In particular, the top three solvers were identical. Therefore, we believe that we found a reasonably diverse selection of benchmarks for our means. However, as the reduction of a test set generally increases the risk of overfitting, we treated better performing but more complicated configurations of our system with caution and only adopted them when we found the improvement to be significant.

5.2 Standalone SAT Solving Performance

We now discuss our experiments involving HordeSat and Mallob-mono. We performed our experiments on 128 nodes of ForHLR II with a total of 2560 physical cores. Consistent with the default configuration of HordeSat, we bind each MPI process to four physical cores. Consequently, we execute $20/4 = 5$ MPI processes on each node which results in up to $128 \cdot 5 = 640$ PEs with up to four solvers each. We included HordeSat both with its original solvers ("old") and our updated portfolio ("new"). We included Mallob with different discount factors α in a basic configuration that is as close as possible to HordeSat. HordeSat imposes an upper bound on the LBD or "glue" value [3] of clauses that are exported: Initially, a clause must be unit or have a maximum LBD score of 2 to be shared, and whenever a PE fills its clause buffer by less than 80% this limit is incremented. We also adopted this mechanism in Mallob. We turned off our clause filter half life mechanism (i.e., we set $X = \infty$) for all runs of Mallob-mono.

As Fig. 4 shows, the updated solvers improve HordeSat's performance considerably. Furthermore, the most naïve and untuned configuration of Mallob



Configuration	#	(+,	-)	PAR-2
HordeSat (old)	51	23	28	252.7
HordeSat	59	28	31	193.7
Mallob $\alpha = 4/8$	59	29	30	196.2
Mallob $\alpha = 5/8$	62	30	32	169.6
Mallob $\alpha = 6/8$	63	30	33	157.4
Mallob $\alpha = 7/8$	63	31	32	**154.0**
Mallob $\alpha = 8/8$	64	31	33	158.2

Fig. 4. Performance of HordeSat and "naïve" Mallob on 128 compute nodes. The table shows solved instances (SAT, UNSAT) and PAR-2 scores [16] (lower is better).

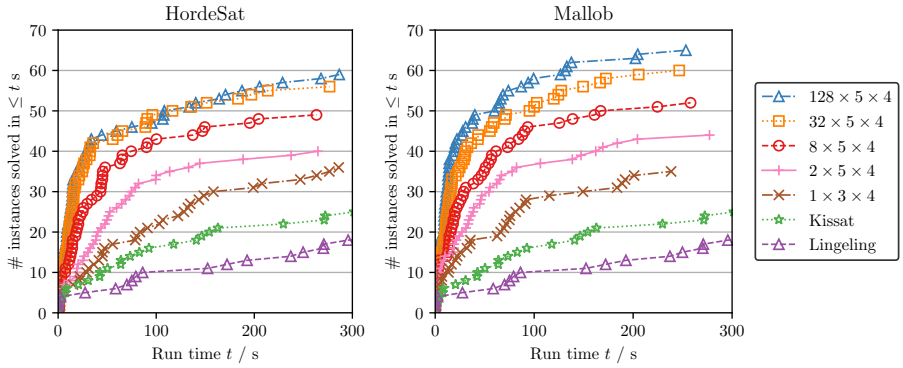


Fig. 5. Scaling behavior of HordeSat (with updated solvers) and Mallob ($\alpha = 7/8$, without any clause length or LBD limits) compared to two sequential solvers.

with $\alpha = 1$ outperforms HordeSat even if both systems make use of the exact same solvers. If $\alpha = 0.5$, only a very small clause buffer of less than 1500 integers is shared each round which proves to be highly detrimental to Mallob's performance and underlines the importance of clause sharing. The best overall performance is achieved with $\alpha = 7/8$ whereas $\alpha = 6/8$ is a close second.

We provide further experimental results for the parametrization of Mallob in the publication's supplementary material. Measured on 128 nodes, Mallob achieved best performance without HordeSat's LBD limit mechanism. We also tested a maximum clause length limit of 5 and 10 and found the results to be mostly inconclusive. As such, we continue with a very simple configuration of Mallob without any strict limits on clause lengths or LBD scores.

We now discuss the scalability of our solver. Fig. 5 provides an overview on the performance of both HordeSat and Mallob when executed on 12, 40, 160, 640, and 2560 cores. As sequential baselines we included Lingeling (in the 2018 version used by Mallob) as well as Kissat [8], the winner of the SAT Competition 2020's main track. Table 1 shows pairwise speedups. We used a time limit of $\tau_s = 50\,000\,\mathrm{s}$ for sequential solvers and $\tau_p = 300\,\mathrm{s}$ for parallel solvers. As in [4] we "generously" attribute a run time of τ_s to the sequential approach for each unsolved instance solved by the parallel approach. We computed the median speedup S_{med} and the total speedup S_{tot} (the sum of all sequential run times divided by the sum of all parallel run times). We also provide speedups emulating "weak scaling", i.e., only considering instances for which the sequential approach took at least as many seconds as the number of cores in the parallel approach.

While both parallel solvers show improved performance whenever the number of cores is quadrupled, HordeSat clearly lacks scalability beyond 32 nodes. As such, Mallob on only 32 nodes outperforms HordeSat on 128 nodes. Furthermore, the 128-node configuration of Mallob achieves a much more pronounced speedup over its 32-node configuration, although we do notice some degree of

Table 1. Parallel speedups for HordeSat (H) and Mallob (M). In the left half, "#" denotes the number of instances solved by the parallel approach and S_{med} (S_{tot}) denotes the median (total) speedup for these instances compared to Lingeling / Kissat. In the right half, only instances are considered for which the sequential solver took at least (num. cores of parallel solver) seconds to solve. Here, "#" denotes the number of considered instances for each combination.

	All instances					Hard instances				
	Lingeling		Kissat			Lingeling		Kissat		
Config.	#	S_{med}	S_{tot}	S_{med}	S_{tot}	#	S_{med}	S_{tot}	#	S_{med}	S_{tot}
H1×3×4	36	3.84	51.90	2.22	29.55	32	4.39	52.01	31	4.03	32.49
H2×5×4	40	12.00	95.80	5.06	64.44	35	12.27	96.83	33	9.11	69.63
H8×5×4	49	22.83	135.55	9.76	90.08	38	32.00	142.76	32	24.88	105.94
H32×5×4	56	42.12	203.66	15.25	112.14	34	97.61	231.77	19	114.86	208.68
H128×5×4	59	50.35	204.10	17.38	111.46	21	356.33	444.12	10	243.42	375.04
M1×3×4	35	4.83	58.15	3.62	64.66	31	5.37	58.24	30	5.29	66.08
M2×5×4	44	12.98	94.44	10.52	67.71	39	14.37	95.28	37	11.54	69.25
M8×5×4	52	28.38	154.62	12.06	89.61	41	34.29	162.23	34	23.43	106.85
M32×5×4	60	53.75	220.92	23.41	148.57	37	152.19	245.54	23	134.07	262.04
M128×5×4	65	81.60	308.48	25.97	175.58	25	363.32	447.97	12	363.32	483.11

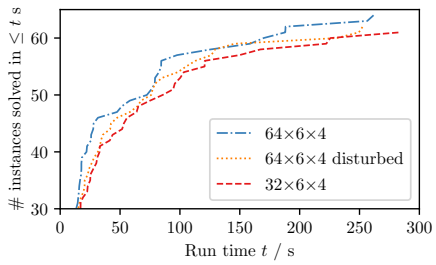
diminishing returns as well. This decline in efficiency motivates the next stage of our evaluations where Mallob resolves multiple jobs in parallel.

5.3 Malleable Job Scheduling

To evaluate Mallob in its scheduling mode, we appoint one PE as a designated "client" which introduces jobs to the system and receives results or timeout notifications. Furthermore, the randomized scheduling and load balancing paradigm of Mallob requires that a small ratio ε of PEs is reserved to remain idle. We cautiously chose $\varepsilon = 0.05$ but expect that lower values of ε can be viable. We limited each PE to keep a maximum of three job nodes (active or inactive).

In a first experiment, we test the basic malleability of our solving engine. We use 64 compute nodes of SuperMUC-NG with a total of 1536 cores and partition each node into six PEs à four cores, resulting in 384 PEs in total. As such, we obtain up to $\lfloor (1-\varepsilon)(p-1) \rfloor = \lfloor 0.95 \cdot 383 \rfloor = 363$ parallel active job nodes. We introduce a sequential chain of 80 jobs to the system. Periodically (once every 30 s), a "stranger" job arrives and resides in the system for a limited time (15 s) during which it occupies half of the available PEs. We run this experiment with and without a clause filter half life $X = 90$, chosen by preliminary tests, to evaluate its impact in such a malleable setting. As a comparison, we repeat the experiment on 64 and on 32 compute nodes without any disturbances.

Fig. 6 shows that the run with disturbances performed worse than the static (i.e., undisturbed) large run and better than the static small run, which is consistent with the available CPU resources in these runs. The periodic reduction of clause filters was not helpful but rather detrimental to Mallob's performance in this specific setting. Still, for the following experiments we continue with a (potentially suboptimal) value of $X = 90$ because we want to ensure from a design perspective that crucial clauses are eventually shared with the PEs which arrive



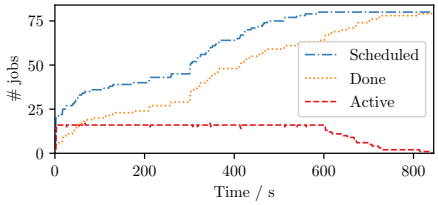
Configuration	#	(+,	-)	PAR-2
32×6×4 $X{=}\infty$	61	28	33	176.6
64×6×4 $X{=}90$ dstrb.	61	28	33	174.5
64×6×4 $X{=}\infty$ dstrb.	62	29	33	166.9
64×6×4 $X{=}\infty$	64	31	33	153.5

Fig. 6. Performance of Mallob with $X = \infty$ (note the range of the y-axis) with and without periodic disturbances. The table shows solved instances (SAT, UNSAT) and PAR-2 scores [16] (lower is better) and also includes a variant with $X = 90$.

late to a job. We intend to pursue more reliable and explicit clause re-sharing strategies for malleable SAT solving in the future.

In our next experiment, we let Mallob resolve several jobs at once to evaluate its load balancing. We use 128 compute nodes of ForHLR II and run four PEs à five threads on each compute node (because this fits best the two-socket hardware at hand). As such, we have 512 PEs and up to 485 parallel active job nodes with $\varepsilon = 0.05$. We limit the number of parallel jobs in the system to $J = 4$ (16, 64) which leads to about 121 (30, 7) PEs or 605 (150, 35) threads per job compared to the 640 (160, 40) threads of the closest tested configuration of Mallob-mono.

For 96% of all measurements we counted exactly 485 busy PEs (94.9% system load). The job scheduling times, measured from the introduction of the initial job request $r_0(j)$ to the initiation of the job description transfer to $p_0(j)$, ranged from 0.003 s to 0.781 s (average 0.061 s, median 0.006 s). Our scheduling and load balancing imposes very little overhead: With $J = 4$ (16, 64) we measured an average of 3.1% (3.0%, 3.0%) of active core time in the PEs' main threads which collectively perform the entire scheduling, load balancing, and communication.



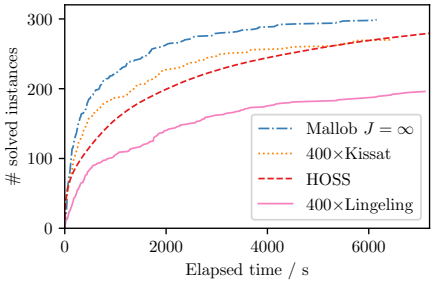
Approach	#	(+,	-)	PAR-2
Mallob $J = 4$	58	26	32	192.7
Mb-mono $m = 32$	**60**	**28**	32	**181.4**
Mallob $J = 16$	**54**	**24**	**30**	**232.7**
Mb-mono $m = 8$	52	23	29	240.1
Mallob $J = 64$	**49**	**21**	**28**	**279.0**
Mb-mono $m = 2$	44	19	25	299.8

Fig. 7. Experiment with a uniform number J of parallel jobs. Left: Number of active jobs and cumulative number of scheduled jobs and done (i.e., finished or cancelled) jobs with $J = 16$ (measured each second). Right: Solved instances and PAR-2 scores (lower is better) of Mallob with $J = 4, 16, 64$ and of comparable Mallob-mono runs.

We now compare Mallob with $J = 4$ (16, 64) with Mallob-mono on 640 (160, 40) cores. Fig. 7 (right) shows that the run with $J = 4$ performed worse, the run with $J = 16$ performed better and the run with $J = 64$ performed much better than its closest *mono* configuration: When few active jobs are left, additional PEs are available to accelerate the resolution of the remaining jobs. This effect is more pronounced the more jobs are being processed overall.

In a final experiment, we evaluate the performance and resource efficiency of Mallob and its scheduling in a more ambitious setting. We again use 128×4×5 cores of ForHLR II. We immediately introduce all 400 benchmark instances of the SAT Competition 2020 at system start and do not impose any time limit per job. As a comparison, we measured the performance of Mallob-mono on 128 nodes for each instance and computed a *hypothetical optimal sequential scheduler* (HOSS) which knows each job's run time in advance. To minimize average response times, the HOSS schedules the 400 runs of Mallob-mono sorted by their run time in ascending order. We also include two trivial but practical schedulers which process all jobs "embarrassingly parallel" by running 400 instances of Lingeling or Kissat at the same time.

Fig. 8 shows that the HOSS outperforms 400 Lingelings, but performs worse than 400 Kissats in terms of median response times. This underlines both the great performance of Kissat and the high resource efficiency of (state-of-the-art) sequential SAT solvers. However, Mallob with malleable scheduling outperforms any of the extremes as it combines parallel job processing with a flexible degree of parallel SAT solving: As more and more jobs finished over time, the average number of cores per job increased steadily from 7.2 to 24. Our system solved 299 instances within 4378 core hours (ch) while the HOSS solves 270 instances with the same resources and takes 7358 ch to solve the same number of instances. To put these measures in perspective [11], Mallob-mono in the SAT Competition 2020 spent 29449 ch for solving 299 instances (7005 ch for solved instances, 22444 ch for unsolved instances), more instances than any other solver. The win-



Configuration	R_{all} avg.	R_{all} med.	R_{slv} avg.	R_{slv} med.
Mallob $J = \infty$	2422.4	679.8	808.6	260.6
400×Kissat	2998.4	1362.5	975.5	355.5
HOSS	2774.7	2024.5	1396.4	937.3
400×Lingeling	4436.0	7200.0	1559.2	819.9

Fig. 8. Cumulative solved instances by different scheduling approaches on 128 compute nodes within two hours. The table shows average and median response times, calculated for all 400 instances (R_{all}) and for the solved instances per approach (R_{slv}). Each unsolved instance leads to a response time of 7200 s.

ning system of the parallel track solved 284 instances within 6548 ch (1392 ch for solved and 5156 ch for unsolved instances). In both cases we estimate the used hardware to be similar in per-core performance to the hardware we used.

To conclude, Mallob is able to find a flexible trade-off between the resource-efficiency of parallel job processing and the speedups obtained by parallel SAT solving based on the current system load. For real world applications, various mechanisms of Mallob can help to steer this degree of parallelism, such as limiting the maximum number J of concurrent jobs, setting individual job priorities, and limiting a job's maximum volume and its (wallclock or CPU) time budget.

6 Conclusion

In order to improve the scalability and resource efficiency of SAT solving in cloud environments, we introduced the *Mallob* framework for the scalable resolution of SAT jobs on demand. We presented a new approach to malleable job scheduling and a SAT solving engine based on HordeSat which features succinct clause sharing, a reworked solver backend supporting malleability, and various practical improvements. We showed that our standalone SAT solver outperforms an improved version of HordeSat and leads to better speedups. We observed that our job scheduling and load balancing imposes very little overhead and that Mallob's combination of parallel job processing and flexible parallel SAT solving is able to improve resource efficiency and response times in a cloud environment.

While we focused on Mallob's SAT solving capabilities in this work, for future work we intend to evaluate the general scheduling and load balancing properties of Mallob under more realistic job arrival rates and varying job priorities. Secondly, we intend to integrate further solver backends and explore better methods for the re-sharing of crucial clauses in order to improve Mallob's performance. Thirdly, we intend to advance Mallob by adding support for incremental SAT solving and for related applications such as automated planning [30].

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500). This work was performed on the supercomputer ForHLR funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research. The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (www.lrz.de). The authors wish to thank Tomáš Balyo and Markus Iser for fruitful discussions, the anonymous reviewers for their helpful feedback and suggestions, and Ekkehard Schreiber and Marvin Williams for kindly proofreading the manuscript.

References

1. Audemard, G., Hoessen, B., Jabbour, S., Piette, C.: Dolius: A distributed parallel SAT solving framework. In: Pragmatics of SAT. pp. 1–11. Citeseer (2014)
2. Audemard, G., Lagniez, J.M., Szczepanski, N., Tabary, S.: An adaptive parallel SAT solver. In: International Conference on Principles and Practice of Constraint Programming. pp. 30–48. Springer (2016)
3. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Twenty-first International Joint Conference on Artificial Intelligence. pp. 399–404 (2009)
4. Balyo, T., Sanders, P., Sinz, C.: HordeSat: A massively parallel portfolio SAT solver. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 156–172. Springer (2015)
5. Biere, A.: Yet another local search solver and Lingeling and friends entering the SAT competition 2014. Proceedings of SAT Competition p. 65 (2014)
6. Biere, A.: Splatz, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT competition 2016. Proceedings of SAT Competition pp. 44–45 (2016)
7. Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT competition 2018. Proceedings of SAT Competition pp. 14–15 (2018)
8. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. Proceedings of SAT Competition p. 50 (2020)
9. Ehlers, T., Nowotka, D., Sieweck, P.: Communication in massively-parallel SAT solving. In: 2014 IEEE 26th international conference on tools with artificial intelligence. pp. 709–716. IEEE (2014)
10. Feitelson, D.G., Rudolph, L.: Toward convergence in job schedulers for parallel supercomputers. In: Workshop on Job Scheduling Strategies for Parallel Processing. pp. 1–26. Springer (1996)
11. Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M.: SAT Competition 2020. Artificial Intelligence (2021), to appear
12. Graham, R.L., Shipman, G.M., Barrett, B.W., Castain, R.H., Bosilca, G., Lumsdaine, A.: Open MPI: A high-performance, heterogeneous MPI. In: 2006 IEEE International Conference on Cluster Computing. pp. 1–9. IEEE (2006)
13. Gropp, W., Gropp, W.D., Lusk, E., Skjellum, A., Lusk, E.: Using MPI: portable parallel programming with the message-passing interface, vol. 1. MIT press (1999)
14. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. Journal on Satisfiability, Boolean Modeling and Computation **6**(4), 245–262 (2010)
15. Heisinger, M., Fleury, M., Biere, A.: Distributed cube and conquer with Paracooba. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 114–122. Springer (2020)
16. Heule, M., Järvisalo, M., Suda, M.: SAT race 2019 (2019), `http://sat-race-2019.ciirc.cvut.cz/downloads/satrace19slides.pdf`, accessed: 2021-05-13.
17. Heule, M.J., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Haifa Verification Conference. pp. 50–65. Springer (2011)
18. Hoos, H.H., Stützle, T.: Local search algorithms for SAT: An empirical evaluation. Journal of Automated Reasoning **24**(4), 421–481 (2000)
19. Hungershofer, J.: On the combined scheduling of malleable and rigid jobs. In: 16th Symposium on Computer Architecture and High Performance Computing. pp. 206–213. IEEE (2004)

20. Iser, M., Balyo, T., Sinz, C.: Memory efficient parallel SAT solving with inprocessing. In: 2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI). pp. 64–70. IEEE (2019)
21. Iser, M., Sinz, C.: A problem meta-data library for research in SAT. Proceedings of Pragmatics of SAT **59**, 144–152 (2019)
22. Kendall, M.G.: Rank correlation methods. Griffin (1948)
23. Kleine Büning, M., Balyo, T., Sinz, C.: Using DimSpec for bounded and unbounded software model checking. In: International Conference on Formal Engineering Methods. pp. 19–35. Springer (2019)
24. Le Frioux, L., Baarir, S., Sopena, J., Kordon, F.: PaInleSS: a framework for parallel SAT solving. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 233–250. Springer (2017)
25. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of satisfiability, pp. 131–153. ios Press (2009)
26. Massacci, F., Marraro, L.: Logical cryptanalysis as a SAT problem. Journal of Automated Reasoning **24**(1), 165–203 (2000)
27. Mehlhorn, K., Sanders, P.: Algorithms and data structures: The basic toolbox. Springer Science & Business Media (2008)
28. Ngoko, Y., Cérin, C., Trystram, D.: Solving SAT in a distributed cloud: a portfolio approach. International Journal of Applied Mathematics and Computer Science **29**(2), 261–274 (2019)
29. Ngoko, Y., Trystram, D., Cérin, C.: A distributed cloud service for the resolution of SAT. In: 2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2). pp. 1–8. IEEE (2017)
30. Schreiber, D.: Lilotane: A lifted SAT-based approach to hierarchical planning. Journal of Artificial Intelligence Research **70**, 1117–1181 (2021)
31. Schubert, T., Lewis, M., Becker, B.: PaMiraXT: Parallel SAT solving with threads and message passing. Journal on Satisfiability, Boolean Modeling and Computation **6**(4), 203–222 (2010)