SArTagnan – A parallel portfolio SAT solver with lockless physical clause sharing*

Stephan Kottler and Michael Kaufmann

University of Tübingen, Germany

Abstract. Since multi–core architectures have become well–established the enquiry for parallel SAT solvers has drastically increased. Meanwhile, several successful SAT solvers have been presented that can be run in parallel mode. However, there are only a few solvers that use the shared memory architectures for physical clause sharing.
In this paper we present the parallel SAT solver SArTagnan that allows for sharing clauses between several threads logically and physically. Yet any thread is still able to keep its own set of clauses.
We show how physical clause sharing can be used to propagate one thread's improvements on the clause database to all solving threads. Despite the extensive sharing of data our solver does not require any operating system lock.

1 Introduction

The engineering of practicable Satisfiability (SAT) algorithms and the intensive optimisation of SAT solvers have made the SAT problem feasible for many computational real–world problems that can be transformed into SAT formulae. Since the improvement of the primal DPLL algorithm [11, 10] by the conflict learning procedure GRASP [28] many huge SAT instances can be solved in reasonable time. The design of efficient data structures and optimised implementations [30, 13] have been ground–breaking for the wide application of SAT solving.
In recent years parallel SAT solving has gained in importance to utilise the potential of multi–threaded architectures. However, the parallelisation of SAT solving is an interesting research area for a long time and many different approaches to parallelise SAT have been studied.
A search procedure may split the entire search space into different subareas that may or may not be disjunct. This approach seems to suggest itself in SAT solving where a subset of variables can be preassigned to different values for different parallel solving procedures. Preassigning variables forces any parallel process to search for a solution in a different part of the search space. This divide–and–conquer approach, which is often said to use a so–called *guiding path* [36], is widely–used in distributed parallel SAT solving. On multi–core architectures the guiding path approach can be realised by the application of dynamic work stealing. An inactive thread requests work from any active thread. The active thread divides its own guiding path into two paths one of which is given to the requesting thread. This idea is used by solvers as pMiniSAT [9], MiraXT [34] and ySAT [14].
On the contrary there is the parallel SAT solving approach that does not guide different solving processes in any way. Even when running the same algorithm in parallel

* This work is suported by the DFG grant SPP 1307 – Algorithm Engineering

the use of different heuristics [8, 18] and some random decisions will lead each process in different directions. Useful information that is globally valid for each solving process may be exchanged. In [7] the idea of exchanging learnt lemmata between parallel executions of the DPLL algorithm has been proposed. These days most parallel state–of–the–art SAT solvers apply this idea to a certain degree.

In the context of multi–threaded SAT solving sharing of learnt clauses is now widely applied. However, the term *sharing* may be ambiguous. Most solvers run an instance of the CDCL (conflict driven SAT solving with clause learning) algorithm in each parallel thread. A copy of any learnt clause that conforms to certain criteria is sent to (some of) the other parallel threads [18]. Hence, information is shared among parallel threads, but a clause itself is not shared physically. Each thread holds its own copy of each clause. To our knowledge the only solvers that share a unique clause database physically are MiraXT [34, 26] and ySAT [14], whereas the latter only shares the set of original clauses physically (see the respective articles for details).

In this paper we present the design and implementation of our parallel solver SArTagnan and we stress the most relevant differences between the version of the SAT Race 2010 and the current version for the SAT competition 2011. All solving threads are allowed to share clauses logically and physically. But the set of clauses in different threads is not required to be identical. Any solving thread can still decide whether it uses a shared clause of another thread and, whether it shares an own clause with other threads. Even though data is shared by all threads the solver does not use any operating system or OpenMP locks. All threads of the solver can be configured to apply different search strategies. Due to the physical sharing of clauses any solving thread can permanently improve the entire set of clauses. Moreover, all threads may benefit from the improvement that was made by one solving thread.

The paper is organised as follows: In Section 1.1 the basic CDCL procedure is sketched. Section 2 presents the most important aspects of clause sharing and the communication between threads. In Section 3 different search strategies of the solving threads are explained. The subsequent Section 4 shows some experimental results and gives an insight into configuration details. Section 5 finally concludes the work.

1.1 The CDCL Procedure

Algorithm 1 lists the basic CDCL procedure that has become predominant in industrial state–of–the–art SAT solving. For detailed information on SAT and SAT solving the reader is referred to [6].
As long as there are unassigned variables a branching choice is made and all implications are computed by the so-called *Boolean Constraint Propagation* (BCP). If an assignment to a variable $w \in \mathcal{V}$ is implied that contradicts its current state, a new clause L is created that expresses the conflict as a single condition. The solver jumps back to a previous partial assignment so that one literal in L becomes unassigned. From time to time the set of learnt clauses is reduced.

2 Parallel Solving

Working on shared memory architectures motivates for sharing clauses physically, so that shared information (i.e. each clause) exists only once in memory. This is

Algorithm 1: Sketch of the CDCL Approach

```
1  Require Formula F in CNF ;
2  Function CDCL(F)
3      A ← ∅ /* current partial assignment */
       V_U ← vars(F) /* unassigned variables */
4      while V_U ≠ ∅ do
5          l ← choose-next-decision(V_U) ;
6          A' ← BCP(l) ;
7          if A' in conflict with A then
8              L ← analyze-conflict(A, A') ;
9              if L = ∅ then return 'Unsatisfiable';
10             F ← F ∪ L; A ← backjump(L) ;
11         else A ← A ∪ A';
12         V_U ← V_U \ {vars(A)};
13         if Maximal number of learnt clauses reached then clean-set-of-clauses ();
14     return 'Satisfiable' /* A satisfies F */
```

motivated by the fact that the set of literals of a clause is basically static. Moreover, sharing clauses physically allows for a better exchange of additional information like subsumption or backward subsumption of clauses.
We first present the basic concepts used to exchange and share information during SAT solving. In Section 2.2 and 2.3 we then go into more detail.

2.1 Basic Concept for Multithreading SAT

We define the number of parallel threads to be t. And we refer to a particular thread by T_i where $i \in [0, t-1]$. During program execution each thread holds a unique user mask $M(T_i)$ that is defined to be 2^i. Each data object that is shared by several threads has a bit mask *usrs* that indicates the set of users of this object. The value of *usrs* can be formalised as $usrs = \Sigma_{T_i \in U_j} M(T_i)$ where U_j is the set of threads who links to the object O_j.
In general, the *usrs* field of an object O_j is always initialised by the creating thread before O_j is actually visible to the other sharing threads. After the creation of the object O_j a reference to O_j is given to all sharing threads $\in U_j$. As soon as any sharing thread of O_j wants to release the object it has to unsubscribe itself as a user of O_j. The last user is responsible for the destruction of O_j. The release operation is listed in Algorithm 2. Note that no thread can ever add itself as an user to an already created and shared object.

The function *bool exchangeIf(addr, assum, new)* is a typical atomic operation that replaces the content at the specified address *addr* by the value *new* but only if the current content is equal to *assum*. It returns true if the exchange operation was successful[1]. The application of user masks is actually very similar to the concept of semaphores that use a simple counter initialised to the number of users. However, there is a good reason for the user masks: For any shared object the set of its users can be determined easily. This turns out to be extremely useful for heuristics on data exchange of different threads.

[1] Using the GNU compiler: *bool __sync_bool_compare_and_swap(addr, assume, new_val)*

Algorithm 2: Release Object by Thread T_i

```
1 Require Reference of object O. Calling thread is T_i
2 Function releaseObject(O, T_i)
3     inv_msk ← ~M(T_i) /* inverted M(T_i)                    */
4     repeat
5         curr ← O.usrs /* copy bit mask                      */
6         rem ← curr & inv_msk /* remove M(T_i) */
7     until exchangeIf (O.usrs, curr, rem) ;
8     if rem = 0 then deallocate (O)
```

2.2 Physical Sharing of Clauses

The concept of user masks as described in the previous subchapter is used to realise sharing of clauses with more than two literals. Basically, in SAT solving each clause represents a static set of literals. However, most state–of–the–art SAT solvers implement the two watched literals scheme [30] in the way it was suggested in [35]: The two watched literals of a clause are always placed at the first two positions of the array of literals. Thus, the position of literals in a clause is permuted permanently. This idea can not be implemented when a clause is shared, since the two watching literals may differ in different solving threads. With the following observation the two watched literals concept can be implemented by a small extension.

Observation 1 *Whenever a clause C is addressed by the basic CDCL algorithm, at least one of the two watched literals of C is known.*

In the basic CDCL algorithm (see Algorithm 1) there are three main functions where clauses are actually touched:

- Boolean Constraint Propagation (BCP)
- Cleaning the set of clauses
- Conflict Analysis

During BCP the literals that became false by the current partial assignment A are examined. Their watcher lists are traversed to check for clauses that are unit or falsified by the assignment A. Hence, when traversing the watcher list of literal l all clauses that are accessed have l as one of its watching literals.
Cleaning the set of clauses can also be done by traversing the watcher lists of all literals. So the argument from above also applies.
In Conflict Analysis the implication graph [29] is traversed backwards. Any clause C that becomes unit by a partial assignment A during BCP causes the remaining literal l_c to be assigned to true. In doing so, C is stored as *reason* for the assignment of l_c. Moreover, l_c is one of the two watchers of C when C is kept as reason for the assignment. If C is traversed during Conflict Analysis it will only be accessed as reason for the assignment l_c. Thus, one watcher of C is known. With the observation above the following corollary can be stated.

Corollary 1 *The information on the two watchers l_{c1}, l_{c2} of a clause C can be saved by one value $C_w := l_{c1}$ XOR l_{c2}.*

Since in the process of accessing a clause C one watching literal l_{ci} is always known, the other watching literal is given by $l_{cj} = C_w$ XOR l_{ci}. This is similar to the concept of static graphs [31].
With Corollary 1 the set of literals of a clause can be shared among several parallel solving threads. By keeping the value C_w of a clause C locally for each thread, the set of literals of C needs only to be read but never to be written by any accessing thread. Thus, in SArTagnan any clause C with more than two literals consists of value C_w and a pointer to the shared set of literals L. To realise sharing and, in particular destruction, each set of shared literals has a user mask as described in Section 2.1.
Note that for single threaded SAT solvers or, for solvers where clauses are not physically shared, Corollary 1 can be applied to reduce the memory usage of each clause: The two watched literals of a clause C can be replaced by the value C_w without any loss of information. In our solvers SApperloT and MoUsSaka we use this technique to represent any clause C with $|C| > 2$ literals by $(|C| - 1)\cdot$ *sizeof(literal)* bytes.

One motivation of sharing clauses in parallel SAT solving is to allow for sharing additional important information on the state and the change of any clause among all threads. If the set of literals of a clause is reduced by simplification techniques like backward subsumption or on–the–fly clause improvement [19] it is desirable that any other thread can benefit from this information.
In SArTagnan we realise this by sending a new version of a clause to all threads that share this clause. As soon as a new version C_n is sent the previous version C_o of the clause is marked to be redundant by setting a particular bit flag in the clause. However, C_o is still valid and can still be used by any thread. Redundant clauses are released when a thread cleans its set of clauses. This requires that every thread is able to communicate with any other thread by sending and receiving messages. For the soundness of the solver two issues are crucial:

- Regard the order: New versions of clauses have to be sent before the previous version is marked to be redundant. Furthermore, after the release of redundant clauses a solver has to check for messages from all other threads.
- Messages between threads must never be lost. A message of any sender must be visible to all receivers immediately, so that a new version of a clause is guaranteed to be visible not later than the old version is marked redundant.

The realisation of the message system is presented in the next subsection.

2.3 Communication of Threads

In this subsection we present the implementation of lossless queues that are used for the communication between all threads. Moreover, receive and send operations are both non–blocking.
There is a known concept to realise non–blocking circular queues for concurrent programs: Sender and receiver use a shared array of fixed size n and the next writing and reading positions (*write*/*read*) are both visible to the sender and to the receiver of the queue. A write operation changes the value $write \leftarrow write + 1 \bmod n$. For read operations analogously. The queue is empty if the values of *write* and *read* are equal. If the queue contains $n-1$ elements a push operation to the queue will not be successful since this would empty the queue. However, this violates the soundness condition

of our solver. Based on the described concept, we present non–blocking queues that allow for a thread–safe extension of memory to ensure that write operations are always successful. The idea is related to the technique used in [20]. Note that concurrent data structures presented for programming languages that use garbage collection [21] can not always be adapted for manual memory management straightforwardly.

Linking Updates If an object is shared among different threads it is often the case that some of the object's data cannot be modified concurrently, since there is no way to perform the modification by an atomic operation. A straight solution to this problem is to provide a link to a new version within the object itself.

Algorithm 3: Link new version O_n as update of object O_o

```
1 Require Object O_o to be updated by a new version O_n
2 Function linkUpdate(O_o, O_n)
3   O_o.udt ← O_n /* copy reference to new object version        */
4   repeat
5     curr ← O.usrs /* copy bit mask                              */
6     udtd ← curr & udt_msk /* set indication for an update       */
7   until exchangeIf (O.usrs, curr, udtd)
```

A new version O_n of an object O_o is completely initialised before it is finally linked as a new version by Algorithm 3. To indicate that a new version for an object exists, a special user mask *udt_msk* is used ($udt_msk \neq M(T_i)$ for all threads). Note that the link $O.udt$ of an object is only used to link a new version of this object but not as an indication for an update. This ensures consistensy when an update is linked while another thread releases the same object concurrently. Otherwise, a thread t_i could be set as a user for a new version O_n of an object O_o, whereas t_i released object O_o at the time when the new version was linked. Hence, t_i would not know about object O_n, but would be registered as a user of O_n. Algorithm 4 shows the simple procedure to load an updated version.

Algorithm 4: Load Update of Object O_k by a thread T_i

```
1 Require Reference of object O
2 Return Next version of object O if availabe on time
3 Function loadUpdate(O, T_i)
    /* object O has pointer O.udt initialised with null. Only the
       owner of O may change this value.                          */
4   if ¬ hasUpdate (O) then return O ;          /* check udt_msk */
5   O_new ← O.udt/* copy reference                                */
6   releaseObject (O, T_i)
7   return O_new
```

We use this concept to make a queue extendable by its writing thread. Basically both, the reading and writing thread share the same data array for communication. The write operation is sketched in Algorithm 5. If there is enough space in the queue (check at line 9) writing can be performed immediately (lines 17,18). If data can not be written a new data array is created and linked as new version (lines 10-15). The new version of the array may allocate more space than the previous version (line 10).

Algorithm 5: Non–blocking push to queue

```
1  Class DataArray
2      udt; /* Pointer to a new version                                  */
3      read; /* next read operation in data                              */
4      write; /* next write operation in data                            */
5      size; data ... /* actual queue data                               */
6  /* A queue has 2 references of type DataArray: C, P consume/produce */
   Require Thread T_i pushes data D to queue
7  Function pushQueue(T_i, D)
8      next_w ← P.write + 1 mod P.size
9      if next_w = P.read then
10         N ← construct new DataArray
11         N.data[0] ← D;
12         N.read ← 0; N.write ← 1;
13         linkUpdate (P, N) /* Link new version                         */
14         releaseObject (P, T_i) /* Unregister for old DataArray        */
15         P ← N
16     else
17         P.data[P.write] ← D;
18         P.write ← next_w;
```

The reading thread only checks for new versions if all data from its actual data array has been read. The read operation is sketched in Algorithm 6. The crucial point to notice about Algorithm 6 is the double check in lines 4 and 6 whether the queue is empty. If the reading thread cannot pop any data from the queue (in line 4) it checks for an update of its data array C. If no update is available it returns in line 5. However, if an update is available it is not ensured that all data was fetched from its current version of C. It might be the case that the writing process W performed several push operations while the reading thread R was between lines 4 and 5 of Algorithm 6. If R reaches line 6 it is ensured that W will only operate on newer versions of the data array since it linked an update. Hence, in line 7 it can be ensured that no data was missed by the reader. The described course of events may appear unlikely, but it really happens in practice when one thread is paused by the scheduler.

The described queue can be extended to serve more than one reading process. In that case there is one reference of the data array $C_1 \dots C_k$ for each reading thread. Thus, it is sufficient to have one queue for each solving thread where it can write messages to all other threads concurrently. This implies that all messages of one sender will be read by all other threads. This is desired and necessary if non–optional clauses are sent but may be undesirable for optional (learnt) clauses.

Thus, every message contains an additional recommendation, which is basically a user mask where those users are marked, to whom the message should be interesting. For messages that contain newly learnt clauses, no user is marked, and for non–optional clauses every user is marked. However, if an optional (learnt) clause C_o is improved (e.g. reduced set of literals) a new clause C_n ($|C_n| < |C_o|$) is created and sent to all other threads. The recommendation is set to the user mask of C_o. C_n is marked as learnt if C_o was learnt. Any receiver will consider the recommendation of a message for the heuristics to decide whether a clause is imported or immediately released.

Algorithm 6: Non–blocking pop from queue

```
1  Require Thread T_i reads next data D from queue
2  Return true if new data was read, false otherwise
3  Function popQueue(T_i, D_out)
4      while C.read = C.write do
5          if ¬ hasUpdate (C) then return false ;          /* check udt_msk */
6          if C.read = C.write then
7              C ← loadUpdate (C, T_i)
8      D_out ← C.data[C.read]
9      C.read ← C.read + 1 mod C.size
10     return true
```

3 Portfolio Solving

The organisation of physical clause sharing and the lossless communication between threads allows for heterogeneous SAT solving. The ability to share the entire set of clauses of all threads allows for several simplification techniques. One advantage over parallel solving where each thread has its own copy of each clause is clearly that every thread may benefit from a simplification of the clause database. If, for instance, a thread reduces the set of literals of a clause by any simplification technique it can post this simplification immediately to all other threads. In general, progress made by one thread may be beneficial for several other threads. This motivates different solving approaches in different threads. To avoid rewriting of similar code for each approach, most functions and classes are parameterised using C++ templates.

3.1 Simplification Thread

One thread of SArTagnan is mainly dedicated to simplify the entire clause database. It imports most clauses that it receives from all other threads. It performs basic simplification techniques as subsumption and backward subsumption of clauses, and aims for eliminating variables as it is done by common preprocessors [12]. Moreover, it tries to detect blocked clauses [22]. Equal variables are detected by searching for strongly connected components in the graph of binary clauses [1].
Only this thread is allowed to decide on variable elimination, replacement of equal variables and deletion of blocked clauses. All three techniques are critical in terms of concurrent application. Granting these simplification techniques only to one thread is a safe way to guarantee the soundness of the parallel solver. If, for instance, two threads were allowed to perform variable elimination, one had to ensure that the concurrent eliminations of different variables are independent of each other. So that the elimination of a variable in one thread does not introduce any new clause that contains a variable which is concurrently eliminated by another thread.
Clauses that are removed as blocked clauses or by variable elimination are kept in an extra list η which can be read by any thread. If a thread finds an assignment which satisfies all clauses it can reimport the clauses of η to compute the complete model for the formula.
If a variable is eliminated or detected to be equal to another variable, new clauses are

constructed and sent to all other threads. In case of variable equality one variable r is chosen to be representative for the set of variables E_r that are equal to r. For each clause that contains a variable of E_r a new clause is created and send to all threads and the original clause is marked to be redundant. As soon as all replacements are performed a particular message is sent to the other threads.

One important task of the simplification thread is the detection of equal or subsuming clauses. The fact that any thread is allowed to create and send improved versions of a clause may introduce duplicate clauses. However, these duplicates will be detected and removed by normal subsumption checks. But to avoid unnecessarily many duplicates every solving thread obeys to the following rule: If a clause C_o can be improved the new version of the clause C_n is only sent to the other threads if C_o was not already marked to be redundant.

Guided search for Autarkies in the SArTagnan version 2010 An Autarky is a partial variable assignment that does not change the satisfiability state of a formula. However, it may change the set of models for satisfying formulae [25]. The simplification thread searches for Autarkies, whereas hints are given by other threads: When conflict analysis in the CDCL algorithm determines to backjump over several decision levels, then it was figured out that none of these decisions contributes to the conflict. At this point a CDCL thread sends a particular message to the simplification thread including the variable assignments that were jumped over. The simplification thread may check whether a subset of these assignments is autarkic to the entire formula. However, these checks are performed with little priority by the simplification thread.

Extensive asymmetric branching in the SArTagnan version 2011 By asymmetric branching[2] the conjunction of the negated literals of a clause $C = (l_1, l_2, \ldots, l_k)$ are propagated in order: $\overline{l_1} \wedge \ldots \wedge \overline{l_k}$. We refer to the partial assignment $\overline{l_1} \wedge \ldots \wedge \overline{l_i}$ by C^i $(l_1 \ldots l_i \in C)$. If C^i implies a value l_j $(l_j \in C; i < j)$ then clause C is redundant and can thus be deleted from the formula.
If C^i implies a value $\overline{l_j}$ $(l_j \in C; i < j)$, then l_j can be removed from C because of the following reason: if an assignment satisfies $(l_1 \ldots l_i)$ then C is also satisfied. If, on the other hand, an assignment contains $\overline{l_1} \wedge \ldots \wedge \overline{l_i}$ then C cannot be satisfied by l_j due to the implication of $\overline{l_j}$ and thus, the literal l_j will never be of any use to satisfy clause C.
The order in which the negated literals of a clause are propagated may produce highly different results. To overcome this drawback of asymmetric branching SArTagnan (and also SApperloT) examine alternative orders of propagation to minimise the number of literals that are kept in clause C. Different alternative orders of literals are not propagated again and again, but are examined by a deeper analysis of the implications graph, related to the idea of inverse arcs [2]. However, for the special application of asymmetric branching alternative predecessors in the implications graph can be stored by bitsets more efficiently, which allows for a fast analysis of different orders concurrently.

[2] Optionally used by several solvers as MiniSat, Precosat, SApperloT 2009 and CryptoMiniSat. Asymmetric branching is also referred to as vivification [32].

3.2 Decision Making with Reference Points

Decision Making with Reference Points (DMRP) is an alternative SAT solving approach that has been proposed by Goldberg [15, 16]. It spends more time on decision making than usual CDCL solving. The DMRP algorithm holds a complete assignment (a so–called reference point) to the variables and considers those clauses for decision making that are unsatisfied by the reference point. Therefore, it requires more computational effort. In [24] it was pointed out that clauses that are learnt during the DMRP algorithm are often more valuable than clauses that are learnt during CDCL. One thread solely applies DMRP solving as described in [24]. In the parallel context it implements a crucial modification at the initialisation of the reference point. At every restart each variable value in the reference point is chosen to be the value that is predominant for this variable in all other threads.

3.3 CDCL Threads

Most threads of SArTagnan apply conflict driven SAT Solving with clause learning (CDCL). The use of class and function templates allows for diverse configuration in each thread. If 8 or more threads are available, all but one CDCL thread use activities for variables for the decision heuristic, whereas one thread uses activity for literals as described in the original paper [30].
All CDCL threads can be configured to apply hyper binary resolution [4]. Binary dominators [5] are used to detect clauses for hyper binary resolution during BCP. Most threads apply on–the–fly clause improvement [19]. If a clause C can be improved (i.e. its set of literals can be reduced) a new clause is created and sent to the other threads. The message will be recommended to all users of C. All CDCL threads use different restart settings. Most threads use the Luby restart strategy [27] with different initial sizes. More details are listed in Section 4.

Extended Unit Propagation In modern SAT solvers Boolean Constraint Propagation is mostly equal to Unit Propagation. Basically, clauses that are unit (considering the current partial assignment) imply the corresponding value for the remaining variable. In [23] two approaches are analysed on how to extend Unit Propagation by considering clauses with more than one unassigned literal. The heuristic approach that either uses the pessimistic or optimistic sink tags technique is applied in almost all threads.

3.4 Handling incoming Messages

In Section 2 and Section 3.1 different types of messages were described to sent newly created clauses and simplification notifications. Overall there are the following types of messages:

- Unit and binary clauses
- Shared clauses with more than two literals
- Elimination of variables
- Replacement of equal variables

Any thread checks for new messages whenever its search process is at decision level zero. However, the receive procedure may be also called at higher decision levels when more than k conflicts happened without an application of the receive procedure. For

most threads k is equal to 256. Moreover, whenever the set of clauses has been cleaned, i.e. clauses that are marked redundant are released, all incoming messages are handled subsequently.

Unit and binary clauses are always imported and binary clauses are put in its own data structure. Messages with shared clauses contain a recommendation to whom the clause will be interesting. If a (learnt) clause C is not recommended to a receiving thread, it may still decide to import C if the following two criteria are satisfied:

– The LBD of C is smaller than $f \cdot \Lambda$, where Λ is the maximum LBD value of any learnt clause that survived the previous garbage collection of learnt clauses.
– Not more than p percent of literals resp. variables of C have an activity value that is smaller than $\frac{\Psi}{2}$, where Ψ is the current maximum activity value of all variables.

In [3] the LBD value of a clause is shown to be a successful criterion on how to predict the quality of a learnt clause. However, since the LBD value is related to a particular CDCL search, it does not have to be meaningful for a different search procedure. Our experiments have shown that using the second criterion as similarly presented in [17] causes a more stable behaviour of the solver. In difference to [17], we do not yet apply control–based adaption for the input criteria. The values f and p are static but different in each thread in the current solver version.

4 Evaluation

Our parallel SAT solver SArTagnan is implemented in C++ using the OpenMP library for parallelisation. This chapter presents some results on solver's performance and gives a more detailed insight into some configuration details.

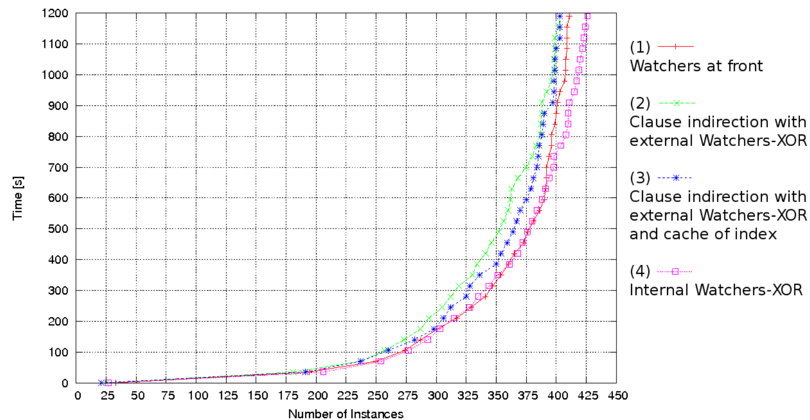

Fig. 1. Effects of different clause organisations for total runtime. Each curve represents one solver configuration. A plotted point x, y means that x instances can be solved within y seconds per instance. The tests have been performed on all industrial instances of the SAT competitions 2007 and 2009 and the SAT Races 2008 and 2010, in total 614 instances.

The data structure for clauses presented in Section 2.2 uses an additional indirection for clauses with more than two literals: A clause itself contains the XOR value C_w

and a link to the shared set of literals. Figure 1 shows the effect of the indirection to the shared set of literals on the solver's speed. To disregard effects of parallelisation and simplification techniques only one thread is used for this analysis that performs the basic CDCL algorithm with the same heuristic settings.
The first configuration implements the common watching scheme keeping the two watchers at the front of a clause [35, 13]. The second configuration wraps a clause into a data structure that contains the value C_w and a link to the set of literals (see 2.2). Configuration 3 also uses this idea but extends the wrapping data structure to cache the index *idx* of a literal in the clause. When a new watching literal has to be found during BCP the literals in a clause are processed in the order $[idx, \ldots, |C|-1, 0, \ldots, idx-1]$. Configuration 4 uses the XOR–idea to completely omit both watched literals and replace them by the value C_w, as it is implemented in our solvers SApperloT and MoUsSaka. As mentioned in Section 2 this reduces the memory to represent a clause C effectively to $(|C|-1)$ times the size of one literal. But configuration 4 is not designed to share clauses physically.
The drawback of indirecting clauses (2,3) compared to configuration 1 is clearly noticeable. However, caching one literal's index (3) is not significantly worse than applying the standard scheme in 1. This motivates for physical clause sharing in parallel SAT solving where the drawback may be compensated by the advantages of global simplification. Configuration 4 clearly outperforms the other configurations, and encourages the XOR–approach also for sequential solvers.

Determining proper settings and constants for sequential SAT solvers is a CPU-Time consuming task. Each configuration has to be evaluated using different random seeds. For parallel solvers this gets much worse. Running a parallel solver with the same configuration twice, may show different performance results. It may be crucial for the success of a solving thread to import a particular clause at a particular time. We have run several tests on the instances of the SAT Race 2008 using 8 cores for each instance. Figure 2 shows the configuration that performed best in several runs and was thus used for the SAT Race 2010.
The first three lines of table 2 show the application of activity values, the percentage of random decisions and the polarity mode (Prev. means phase saving as proposed in [33]). For the restart type the initial number of conflicts (for geometric also the growth factor) is shown. Line 5 states the type of extended Unit Propagation (see. 3.3, [23]). Lines 6 and 7 give the import criteria as explained in 3.4. In lines 8 and 9 the use of on–the–fly clause improvement [19] and hyper binary resolution [5] is indicated. The last three lines give the configuration for cleaning the set of clauses: the initial number of learnt clauses, the increment after each clean up and the percentage of clauses to keep.

An interesting observation that we made for several different configurations is the influence of the two threads that apply DMRP and, CDCL with activities of literals: In the best runs 95 of 100 instances were solved within a time limit of 1200 seconds. Both threads together only solved 9 of these instances. However, if both are replaced by CDCL threads using variable activity at most 88 instances could be solved.

5 Conclusion

In this work we have presented a design and implementation that allows for physical clause sharing in parallel SAT solving. Clause sharing and the communication between

Threads Main Task	1 Simplification	2 CDCL	3 CDCL	4 DMRP	5 CDCL	6 CDCL	7 CDCL	8 CDCL
Activity of: Decay	Vars 135/128	Vars 135/128	Vas 135/128	/	Vars 139/128	Lits 137/128	Vars 139/128	Vars 133/128
Rand. Decisions (%)	0.2	0.15	0.2	0.15	0.15	0.15	0.15	0.2
Polarity Mode	Prev	Prev	Prev	/	Prev	/	False	Prev
Restart type	Static 100	Luby 32	Luby 64	Luby 100	Luby 16	Geo. 100; 1.5	Geo. 100; 1.5	Geo. 100; 1.3
Extended UP	Opt.	Pes.	Opt.	No	Opt.	Opt.	No	Pes.
LBD import	3/2	3/2	3/2	3/2	5/4	3/2	3/2	7/6
Activity import	90	70	80	70	70	80	70	80
OTF Clause improve	yes	yes	yes	no	yes	yes	yes	yes
Hyper Bin. Res.	yes	yes	yes	no	yes	yes	yes	yes
Garbage Coll. Init	35000	50000	35000	50000	50000	50000	50000	50000
GC Increment	200	200	200	200	200	200	200	1
GC Survive Fact.	0.75	0.5	0.75	0.5	0.5	0.5	0.5	0.5
⇓ Differences in version 2011. Equal settings are omitted. ⇓								
Decay			139/128		135/128			
Rand. Decisions (%)		0.2	0.15		0.2			
Polarity Mode			False				Prev	
Restart type		Glucose 100	Geo 100; 1.5	Static 50	Glucose 200		Luby 16	Glucose 100
Extended UP							Opt.	
LBD import					7/6		5/4	7/6
Activity import			50		80			
Hyper Bin. Res.								no
Garbage Coll. Init		30000	50000			35000		
GC Increment		500		300			300	
GC Survive Fact.			0.5		0.75			

Fig. 2. Configuration with 8 threads for the SAT Race 2010 and SAT Competition 2011

threads is used to let all threads benefit from the application of simplification and clause minimisation techniques in any other thread. All communication and sharing of data is realised without the use of operating system locks.
In the SAT Race 2010 the first version of our parallel solver could already compete against state–of–the–art parallel SAT solvers. This motivates future research on how to further utilise physical clause sharing in parallel SAT solving to compensate its drawback of slowed unit propagation by enabling global simplification.
We have also presented a general idea on how to speed up unit propagation for sequential SAT solvers: Replacing the two watched literals of a clause by their XOR-value reduces the amount of memory allocated by each clause and, therefore improves the performance of unit propagation.

Acknowledgement: We thank the anonymous reviewers for helpful comments and valuable links to other related work indicating good directions for future research.

References

1. B. Aspvall, M. F. Plass, and R. E. Tarjan. A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas. *Inf. Proc. Lett.*, 8:121–123, 1979.
2. G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Sais. A generalized framework for conflict analysis. In *SAT*, pages 21–27, 2008.
3. G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In *International Joint Conference on Aritifical Intelligence IJCAI*, pages 399–404, 2009.
4. F. Bacchus. Enhancing Davis Putnam with Extended Binary Clause Reasoning. In *AAAI Conference on Artificial Intelligence*, pages 613–619, 2002.
5. A. Biere. Lazy hyper binary resolution. Algorithms and Applications for Next Generation SAT Solvers, Dagstuhl Seminar 09461, Dagstuhl, Germany, 2009.
6. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, Frontiers in Artificial Intelligence and Applications. IOS Press, 2009.
7. W. Blochinger, C. Sinz, and W. Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 29(7):969–994, 2003.
8. W. Blochinger, C. Sinz, and W. Küchlin. A universal parallel SAT checking kernel. In *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications PDPTA 03*, 2003.

9. G. Chu, A. Harwood, and P. J. Stuckey. Cache conscious data structures for boolean satisfiability solvers. *JSAT*, 6:99–120, 2009.
10. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
11. M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.
12. N. Eén and A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *SAT*, pages 61–75, 2005.
13. N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, 2003.
14. Y. Feldman, N. Dershowitz, and Z. Hanna. Parallel multithreaded satisfiability solver: Design and implementation. *Electr. Notes Theor. Comput. Sci.*, 128(3):75–90, 2005.
15. E. Goldberg. Determinization of resolution by an algorithm operating on complete assignments. In *SAT 2006*, 2006.
16. E. Goldberg. A decision-making procedure for resolution-based SAT-solvers. In *SAT 2008*, 2008.
17. Y. Hamadi, S. Jabbour, and L. Sais. Control-based clause sharing in parallel SAT solving. In *Joint conference on Artifical Intelligence*, pages 499–504, 2009.
18. Y. Hamadi and L. Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT*, 2009.
19. H. Han and F. Somenzi. On-the-fly clause improvement. In *SAT*, 2009.
20. D. Hendler, Y. Lev, M. Moir, and N. Shavit. A dynamic-sized nonblocking work stealing deque. *Distributed Computing*, 18(3):189–207, 2006.
21. M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.
22. M. Järvisalo, A. Biere, and M. Heule. Blocked Clause Elimination. In *TACAS*, pages 129–144, 2010.
23. M. Kaufmann and S. Kottler. Beyond Unit Propagation in SAT Solving. In *Symposium on Experimental Algorithms*, 2011.
24. S. Kottler. SAT Solving with Reference Points. In *SAT*, pages 143–157, 2010.
25. O. Kullmann. Investigations on autark assignments. *Discrete Applied Mathematics*, 107(1-3):99–137, 2000.
26. M. D. T. Lewis, T. Schubert, and B. Becker. Multithreaded SAT solving. In *12th Asia and South Pacific Design Automation Conference*, 2007.
27. M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. In *ISTCS*, pages 128–133, 1993.
28. J. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *EPIA '99: Proceedings of the 9th Portuguese Conference on Artificial Intelligence*, pages 62–74, London, UK, 1999. Springer-Verlag.
29. J. P. Marques-Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.*, 48(5):506–521, 1999.
30. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *DAC*, 2001.
31. S. Näher and O. Zlotowski. Design and implementation of efficient data types for static graphs. In *Algorithms – ESA*, pages 157–164. Springer, 2002.
32. C. Piette, Y. Hamadi, and L. Sas. Vivifying propositional clausal formulae. In *European Conference on Artificial Intelligence (ECAI'08)*, pages 525–529, 2008.
33. K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *SAT*, pages 294–299, 2007.
34. T. Schubert, M. D. T. Lewis, and B. Becker. PaMira - A Parallel SAT Solver with Knowledge Sharing. In *MTV*, pages 29–36, 2005.
35. A. Van Gelder. Generalizations of watched literals for backtracking search. In *Seventh Int'l Symposium on AI and Mathematics*, Ft. Lauderdale, FL, 2002.
36. H. Zhang, M. P. Bonacina, and J. Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.