Student Research Project on "Parallelisation in SAT Preprocessing"

Linus Lutwin Feiten

Chair of Computer Architecture
University of Freiburg

Abstract. Taking advantage of parallel computation on multi-core processors to decrease the runtime of SAT preprocessing is possible but not trivial. In this work we particularily examined the parallelisation of the preprocessing steps *Subsumption* and *Self-Subsuming Resolution*. Sorting the clauses to be checked by clause length decreases the preprocessing runtime for very large SAT instances even further.

1 Introduction

The Boolean satisfiability problem (SAT) (see section 2.1) is of great importance in contemporary electronic design automation but also in AI planning and other fields where NP-complete problems need to be solved in reasonable time. During the last ten years, there has been a lot of progress in the field of so called SAT solver programs [5,6,10] and the continuing mission of the SAT community is to develope new SAT solvers which can solve the SAT problem faster and handle bigger SAT instances (i.e. Boolean formulas).

It has been shown that the preformance of SAT solvers can be improved drastically when certain preprocessing procedures are run on the SAT instances before the actual SAT algorithm starts [1,3,4,8]. In this work we are describing our approach to take advantage of modern multi-core processors' parallelisation capabilities in order to achieve a speed-up for some of the most time consuming preprocessing steps: *Subsumption* and *Self-Subsuming Resolution* (see section 2.2).

This article is structured as follows: Section 2 gives an introductory overview of the SAT problem and a detailed description of the preprocessing concepts used in our algorithm. In section 3 we discuss our ideas for parallelisation of these preprocessing concepts and describe our multi-threaded implementation. The benchmark results achieved with these parallel implementations compared to conventional sequential ones are presented in section 4. A discussion of these result concludes the work in section 5.

2 Preliminary considerations

2.1 SAT and CNF

In the following, familiarity with propositional logic formulas is assumed. Most of the time we will just use the word formula if we mean propositional logic formula. Furthermore, we will use $\wedge$ and $\vee$ for the propositional connectives *and* and *or* and we will use $\overline{x}$ for the negation of any propositional expression x. An example for such a formula over a set of variables $V = \{a, b, c\}$ could be $f = (a \wedge \overline{b}) \vee \overline{(\overline{a} \vee c)}$. An *assignment* for a formula is a relation $V \to \{true, false\}$ that assigns $true$ or $false$ to each variable $v \in V$. For such an assignment, which assigns $true$ or $false$ to every variable, the propositional expression $(x \wedge y)$ is $true$ iff x is $true$ and y is $true$; otherwise it is $false$. Similarly, $(x \vee y)$ is $false$ iff x is $false$ and y is $false$; otherwise it is $true$. The negation $\overline{x}$ of a propositional expression x is $true$ iff x is $false$; otherwise it is true.

To decide whether there exists an assignment such that an arbitrary formula evaluates to *true* is known to be NP-complete. That means that there is no known algorithm to produce an answer in polynomial time and solve the so called SAT (for satisfiability) problem. However, there have been major progresses within the last decade regarding *SAT solvers*, which use algorithms based on *resolution* (see below) and canny search space pruning techniques so that results for many SAT instances can be computed in reasonable time despite NP-completness [5, 6, 10].

In order to apply a SAT solver the formula has to be in *conjunctive normal form* (CNF). A CNF is a conjunction of clauses and a clause is a disjunction of literals. A literal again is a variable v in its positive phase v or negative phase $\overline{v}$. In order for a CNF to be satisfiable there has to be an assignment such that each clause evaluates to *true*; i.e. in each clause there has to be one literal evaluating to *true*.

Every propositional logic formula can be transformed into an equivalent CNF through expansion and application of the boolean distributive law. That, however, has in the worst case an exponential complexity and is therefore not effectively applicable. But there is a method with polynomial complexity to transform any formula f into a CNF that is satisfiable by the same assignments as f: the so called Tseitin transformation [9]. To achieve that, a set of fresh variables has to be introduced which is not appearing in the original formula.

The formula $f = (a \wedge \overline{b}) \vee \overline{(\overline{a} \vee c)}$, for example, would be transformed into a CNF as follows: For each non-atomic sub formula (i.e. a propositional expression within the formula which is not a literal on its own) and for the whole formula itself we have to introduce one distinct fresh variable. In our example we need three. One for the sub formula $(a \wedge \overline{b})$, one for the sub formula $(\overline{a} \vee c)$ and one for the whole formula. We name our set of fresh variables $U = \{d, e, f\}$ and begin to construct the new formula $\textsc{Cnf}(f, U)$.

$$f = (a \wedge \overline{b}) \vee \overline{(\overline{a} \vee c)}$$

$$\begin{aligned}\textsc{Cnf}(f, U) = d \, &\wedge (d \leftrightarrow (e \vee \overline{f})) \\ &\wedge (e \leftrightarrow (a \wedge \overline{b})) \\ &\wedge (f \leftrightarrow (\overline{a} \vee c))\end{aligned}$$

It should be obvious that every satisfying assignment of $\textsc{Cnf}(f, U)$ is also a satisfying assignment of f, if we ignore the fresh variables. But we still do not have a CNF yet. Some rewriting has to be done, so we substitute $(x \vee \overline{y}) \wedge (\overline{x} \vee y)$ for every $(x \leftrightarrow y)$ and get

$$\begin{aligned}\textsc{Cnf}(f, U) = d\ &\wedge (d \vee \overline{(e \vee \overline{f})}) \wedge (\overline{d} \vee (e \vee \overline{f}))\\ &\wedge (e \vee \overline{(a \wedge \overline{b})}) \wedge (\overline{e} \vee (a \wedge \overline{b}))\\ &\wedge (f \vee \overline{(\overline{a} \vee c)}) \wedge (\overline{f} \vee (\overline{a} \vee c)).\end{aligned}$$

Now we can use De Morgan's laws $\overline{(x \vee y)} \Leftrightarrow (\overline{x} \wedge \overline{y})$ and $\overline{(x \wedge y)} \Leftrightarrow (\overline{x} \vee \overline{y})$ and the distributive law to get the final CNF, which could be handled by a SAT solver:

$$\begin{aligned}\textsc{Cnf}(f, U) = d\ &\wedge (d \vee \overline{e}) \wedge (d \vee f) \wedge (\overline{d} \vee e \vee \overline{f})\\ &\wedge (e \vee \overline{a} \vee b) \wedge (\overline{e} \vee a) \wedge (\overline{e} \vee \overline{b})\\ &\wedge (f \vee a) \wedge (f \vee \overline{c}) \wedge (\overline{f} \vee \overline{a} \vee c)\end{aligned}$$

2.2 Preprocessing

Once a SAT instance is available in CNF it can be processed by a SAT solver which will return whether the formula represented by the CNF is satisfiable or not. The method according to which a SAT solver works is in most cases based upon the DPLL algorithm [2] and its extensions [5, 6, 10]. This method will not be discussed in this work as it has no relevance for the understanding of the preprocessing concepts.

As stated in the introduction of this work, it could be shown that the performance of SAT solvers (i.e. runtime until a result is returned) can be improved when the CNF undergoes certain preprocessing procedures before the actual SAT algorithm starts [1, 3, 4, 8]. These procedures mostly aim at reducing the number of clauses, literals and variables in the CNF, which (in most cases) leads to a faster processing through the SAT solver. Sometimes it is even possible to determine whether an instance is not satisfiable by simply running the preprocessing steps.

In order for the preprocessing to be worthwhile it is necessary that its runtime does not exceed the runtime saved during the subsequent SAT solver execution. Hence, it is of great interest to have fast preprocessing routines which at the same time reduce the CNF as much as possible. During the developement of the *MiraXT2008* SAT solver some preprocessing steps emerged to be very gainful but also time-consuming. A selection of these will be explained in the following subsections before we describe (in Section 3) how we parallised these routines with the goal of reducing their runtimes.

Subsumption Consider two different clauses c_1 and c_2 of a CNF and let c_1 include (among others) all literals which are included in c_2:

$$c_1 = c_2 \vee R_1$$

with R_1 being the rest of c_1 (i.e. a clause execlusively comprising all literals included in c_1 but not in c_2). If this is the case we say "c_2 (syntactically) subsumes c_1" or "c_1 is (syntactically) subsumed by c_2".

As mentioned above, in order for the whole CNF to be satisfiable each clause has to be satisfiable by the same variable assignment. Therefore, if the CNF including c_1 and c_2 is satisfiable, both c_1 and c_2 have to be satisfied by the same assignment. It is obvious, however, that if c_2 is satisfied, $c_1 = c_2 \vee R_1$ is satisfied as well. Hence, we do not alter the satisfiability of the CNF when we simply remove a subsumed clause from the conjunction of clauses and thereby reduce the CNF by one clause.

An algorithm could traverse the CNF and check for each clause if it is subsumed by any other clause of the CNF. If a subsuming clause is found, the subsumed clause could be removed and the algorithm could continue with the next clause of the CNF. Alternatively, one could also check for each clause if it subsumes (as opposed to "is subsumed by") any other clause and remove all subsumed clauses found in this "backwards" manner before continuing with the next clause of the CNF.

The *MiraXT2008* SAT solver implements the "backwards" version of subsumption where the algorithm checks for each clause if it subsumes any other clause. When searching for subsumed clauses, one does not need to check *all* other clauses of the CNF (in the following also referred to as the *clause database*). As all literals from the subsuming clause have to be included in the subsumed clause, it is sufficient to check the shortest *occurs list* of all literals included in the currently examined (potentially subsuming) clause. An occurs list $occurs(l)$ of a literal l is a list of all clauses from the *clause database* in which the literal l occurs. Thus, *MiraXT2008's* subsumption routine checks for each clause c which literal in c has the shortest occurs list and afterwards checks if c subsumes any clause referred to in that particular occurs list.

Searching for subsumed clauses can be made more efficient by the following implementation technique: each clause gets a 32-bit *signature* depending on the variables it contains (it makes no difference whether a variable appears in its positive or its negative phase). A hash function returns for each variable a 32-bit word only consisting of 0's except for one bit among the 16 most significant bits (MSB) and one bit among the 16 least significant bits (LSB), which are set to 1 depending on the variable. The signature of a clause is the bitwise OR of all its variables' hash function values.

Hence, if a clause c_1 is subsumed by another clause c_2 (i.e. c_1 includes at least every literal included in c_2), the signature bit word of c_1 must at least have a 1 at every positon at which the signature bit word of c_2 has a 1. The result of a bitwise AND between c_2's signature and the bitwise complement of c_1's signature has to be 0^{32} if this is the case. As the hash function is not injective and not sensitive to the variables' phases, the signature check can only be used to verify whether a clause does *not* subsume another one. Before even executing the signature check one can also check the lengths of the clauses: a clause can never be subsumed by another clause with a greater number of literals.

After the lengths and the signature checks return positive one has to compare the two clauses literal by literal which is the most time consuming part of subsumption. For further details regarding the clause signature the reader is referred to [3].

Self-subsuming resolution In order to understand the CNF preprocessing method called *self-subsuming resolution* one first has to be familiar with the concept of *resolution* in general: Consider two different clauses c_1 and c_2 and let c_1 include a variable v as a literal in positive phase and c_2 include the same variable v as a literal in negative phase:

$$c_1 = v \vee R_1, \qquad c_2 = \overline{v} \vee R_2$$

with R_1 and R_2 being the remaining literals except v and $\overline{v}$ of c_1 and c_2, respectively. Now it can be shown that the new clause $(R_1 \vee R_2)$ is satisfiable iff $(c_1 \wedge c_2)$ is satisfiable:

"$\Rightarrow$": Notice that v does neither occur in R_1 nor in R_2 and can thus be assigned arbitraily without altering whether $(R_1 \vee R_2)$ is satisfied or not. If $(R_1 \vee R_2)$ is satisfied, either R_1 or R_2 is satisfied. If R_1 is satisfied, c_1 is instantly satisfied and v can be assigend to $false$ such that c_2 is satisfied as well. In the same way, if R_2 is satisfied, c_2 is instantly satisfied and v can be assigend to *true* such that c_1 is satisfied as well. Thus, $(c_1 \wedge c_2)$ can be satisfied. $\square$

"$\Leftarrow$": If $(c_1 \wedge c_2)$ is satisfied, both c_1 and c_2 have to be satisfied. Exactly one of the two clauses is satisfied due to the assignment of the variable v; either c_1 if v has been assigned to *true* or c_2 if v has been assigened to $false$. The other clause can hence only by satisfied due to its rest R_i being satisfied. Thus, by either R_1 or R_2 being satisfied $(R_1 \vee R_2)$ is satisfied. $\square$

$(R_1 \vee R_2)$ is called the *resolvent* of c_1 and c_2. Since the conjunction of two clauses is satisfiable iff their resolvent is satisfiable, two clauses of a CNF can be replaced by their resolvent without changing the CNF's satisfiabilty.

Self-subsuming resolution incorporates this concept of resolution. Eén and Biere found out that in SAT instances there are often clauses which *almost* subsume another clause [3]. By "almost" they mean that a clause c_1 would subsume another clause c_2 if there was not *one* literal occuring in c_1 in the opposite phase of its occurence in c_2. Consider the following concrete example:

$$c_1 = a \vee b \vee \overline{c}, \qquad c_2 = \overline{a} \vee b \vee \overline{c} \vee \overline{d} \vee e$$

c_1 would subsume c_2 if c_2 included a instead of $\overline{a}$ or if c_1 included $\overline{a}$ instead of a. Thus, subsumption is out of the question for simplifying the CNF. But there is another possibility for simplification by removing $\overline{a}$ from c_2 without changing the satisfiability of the whole CNF. This is possible because through resolution one can get the clause $(b \vee \overline{c} \vee \overline{d} \vee e)$ as the resolvent of c_1 and c_2. Removing the respective literal from the *almost* subsumed clause c_2 is called *strengthening* c_2 *using* c_1.

Finding clauses which can be strengthened as just described would involve a literal-by-literal clause checking procedure very similar to the one needed to find subsumed clause. This is why it suggests itself to integrate into the algorithm finding subsumed clauses an algorithm finding *almost* subsumed clauses as well.

3 Parallel algorithm and implementation

In the following description of the algorithm we will not go into all details of the implementation which are much easier to comprehended on review of the *MiraXT2008* source code [7]. We will restrain ourselves to merley describing the data structures and programming techniques down to a level on which their characteristic properties become apparent.

3.1 The Clause database

In *MiraXT2008* every literal is associated with an integer number. Clauses are maintained as vector objects of integers each of which is representing a literal occuring in the respective clause. The clause vectors are all stored in another vector called the *clause database*, from which every clause can be obtained by its clause number (i.e. its position in the *clause database* vector). Stored for quick access at a designated position within each clause is the clause's *length*. The *signatures* (see chapter 2.2) of the clauses are saved in a separate signature vector of the same size as the *clause database* vector.

3.2 Subsumption algorithm without parallelisation

When the concept of subsumption was introduced in chapter 2.2 it has been said that the subsumption algorithm is applied to the whole CNF before the actual SAT algorithm starts. This being the case, there are also several points during preprocessing at which it is worthwhile to invoke subsumption again on another subset of clauses. Hence, when our subsumption function starts it is always provided an integer vector called *ClauseSet* containing the clause numbers of all clauses it is meant to check. Remember that *MiraXT2008* uses the "backwards" subsumption method (see chapter 2.2). Thus, our subsumption algorithm checks for all clauses referred to in *ClauseSet* if there are clauses subsumed by them in their shortest occurs lists and, if there are, deletes those clauses. This is done in the following manner (refer to chapter 2.2 for an explaination of *occurs lists* and *clause signatures*):

1. Get new *clause*: If *ClauseSet* is empty, we are done. Otherwise obtain the clause referred to by the last entry of *ClauseSet* from the *clause database*. In the following, this clause will be called *clause*. Afterwards continue with step 2. (The reason why we are traversing *ClauseSet* from right to left instead of left to right is merely due to a simpler parallelisation implementation. See chapter 3.3. For the sequential version it does not make a difference.)
2. Get shortest *occurs list*: Identify the literal of *clause* with the shortest *occurs list*. Continue with step 3.
3. Get first *otherClause*: Obtain from the *clause database* the clause referred to by the first entry of *clause's* shortest *occurs list*. In the following, this clause will be called *otherClause*.
4. Lengths check: Begin checking whether *otherClause* could be subsumed by *clause* or strengthened using *clause*. This is done by checking the two clauses' lengths. If *clause* is longer than *otherClause*, no subsumption or strengthening of *otherClause* is possible. If this is the case, continue with step 9. Otherwise continue with step 5.
5. Signature check: Check by comparison of *clause's* and *otherClause's* signatures whether a subsumption or strengthening of *otherClause* could be possible. If not, continue with step 9. Otherwise continue with step 6.
6. Literal-by-literal check: When both the lengths check (step 4) and the signature check (step 5) have returned positive, it is necessary to conduct a literal-by-literal check between *clause* and *otherClause*. This is done by trying to find every literal of *clause* within *otherClause*. If this search is successful for every literal of *clause*, *otherClause* is subsumed by *clause*: continue with step 7.
   The algorithm is, however, not only sensitive for the identical occurence of literals from *clause* within *otherClause* but also for their inverted phases. If the inverted phase of a literal from *clause* is found in *otherClause*, subsumption is no longer possible. Should, however, all other literals from *clause* be occuring identically in *otherClause*, *otherClause* can be strengthened using *clause*. I.e. the *one* literal in *otherClause*, which occurs in *clause* in its inverted phase, can be removed from *otherClause*. Thus, if the first such inverted literal is found in *otherClause* the search continues and tries to validate the congruence of *clause's* remaining literals with *otherClause*. Should this succeed, continue with step 8.
   If a literal from *clause* could not be found in *otherClause* neither in its original nor in its inverted phase or if more than one literal from *clause* has been found in *otherClause* in its inverted phase, neither subsumption nor strengthening of *otherClause* is possible: continue with step 9.
7. Subsumption: If the literal-by-literal check resulted in *otherClause* being subsumed by *clause*, *otherClause* can be removed from the *clause database* because (as shown in chapter 2.2) *otherClause* would be instantly satisfied by any assignment which satisfies *clause*. Afterwards continue with step 9.
8. Self-subsuming resolution: If the literal-by-literal check returned that *otherClause* can be strengthened using *clause*, the respective literal can be removed from *otherClause*. Afterwards continue with step 9.

9. Get new *otherClause*: If there are still unchecked clauses in *clause's* shortest *occurs list* make *otherClause* the next one of these and continue with step 4. If all clauses referred to by this particular *occurs list* have already been checked, remove *clause* from *ClauseSet* and continue with step 1.

3.3 Subsumption algorithm with parallelisation

The most time-consuming part of the algorithm described in the preceding section clearly consists of the literal-by-literal checks. It therefore appears most promising to distribute this effort among several threads running in parallel on different processor cores. In our approach we have each subsumption thread functioning very similar to the sequential manner with the distinction that each clause of *ClauseSet* is only checked by exactly one thread. If there are, for example, two threads and *ClauseSet* has the size of n, one thread would check the clauses at positions $n-1, n-3, n-5, ...$ while the other one would check those at positions $n-2, n-4, n-6, ...$ (Notice that there is no position n as the first position of an array is 0.)

Our subsumption threads get initiated in the very beginning of the program and are merely put into a suspend mode when they are not needed. This keeps the thread maintenance runtime overhead at a minimum. Once (re-)activated the subsumption threads start to examine their designated clauses from *ClauseSet* as described above. This goes on until one of them would need to update the *clause database* either by removing one or more subsumed clauses or by removing a literal from a strengthened clause. In this case all subsumption threads have to be stopped before the change to the *clause database* can be made, because it is not possible to ensure that the clauses or literals which are about to be removed are not currently read by another subsumption thread.

The way this issue is handled is as follows: Each subsumption thread owns a private integer variable *clauseSetPtr* which stores the position in *ClauseSet* of the clause the thread is currently examining. A subsumption thread which identifies the necessity of updating the *clause database* sets a global integer variable *stopSubsumptionPtr*, which is otherwise -1, to the value of its own *clauseSetPtr*. (Now, it becomes clear why we are having the threads traverse *ClauseSet* from right to left, such that the *ClauseSet* positions of the clauses are getting smaller as the threads proceed.) Each subsumption thread checks whether *stopSubsumptionPtr* is still smaller than its own *clauseSetPtr* before beginning to examine the clause currently referred to by its own *clauseSetPtr*. If *stopSubsumptionPtr* is greater than the thread's own *clauseSetPtr*, the thread enters its suspend mode. Furthermore, a thread will only set *stopSubsumptionPtr* to its own *clauseSetPtr* if *clauseSetPtr* is greater than the current *stopSubsumptionPtr*.

This routine ensures that once the subsumption threads have entered their suspend mode *stopSubsumptionPtr* is storing the rightmost (i.e. greatest) position in *ClauseSet* at which there is the number of a clause responsible for an update of the *clause database*. If *stopSubsumptionPtr* is -1, all clauses of *ClauseSet* have been checked and we are done. Otherwise, the algorithm can now do the changes to the *clause database* according to the clause referred to by *stopSubsumptionPtr* without having to worry about possibly interfering with the subsumption threads, as these are suspended. Afterwards, all entries of *ClauseSet* at the positions geater than or equal to *stopSubsumptionPtr* are removed from *ClauseSet*, as there are surely no clauses referred to by those entries which still need to be checked. Then *stopSubsumptionPtr* is reset to -1 and the subsumption threads are reactivated. This procedure continues until the threads eventually enter their suspend mode leaving *stopSubsumptionPtr* still set to -1. [1]

An important characterisic of this parallelised routine is that it is deterministic. I.e. it will always process the clauses of *ClauseSet* in the exact same order as its sequential equivalent would; no matter how many parallel threads are being used.

A hypothesis of ours has been that parallelisation is only worthwhile when *ClauseSet* surpasses a certain size, because even though we were able to keep the thread maintenance low by merely putting them into the suspend mode when not needed, stopping and reactivating them still requires some overhead. Hence, our program is capable of executing both the parallel algorithm described in this subchapter and the sequential one described in the preceeding subchapter. Which one is applied depends on whether the size of *ClauseSet* surpasses a threshold which can be set as a compile parameter.

3.4 Sorting of ClauseSet

Another hypothesis we had was that the gain through parallelisation would be bigger if the separate subsumption threads were able to complete bigger parts of *ClauseSet* without being interrupted. We assumed that this could be achieved by sorting the entries of *ClauseSet* in ascending order according to the lengths of the referred clauses. Starting with the rightmost clauses of *ClauseSet* the algorithm would first check the longest clauses for which it is less likely to find subsumed clauses than for short ones. Our hope was that the longer clauses could be checked continuously with very little interruptions of the threads before the shorter clauses are checked.

[1] In an earlier version we circumvented the problem of having to stop the subsumption threads at every detection of a subsuming clause by not removing the subsumed clauses right after they had been found but by storing their numbers in a list we maintained until all clauses in *ClauseSet* had been checked. The thusly stored clauses could then be removed from the *clause database* all at one time after the subsumption threads had been able to run through their designated clauses without interruption. Using this method, the speed-up of a parallel version compared to its sequential equivalent was as good as a speed-up through parallelisation can get: The runtime of the parallel version was the runtime of the sequential one devided by the number of subsumption threads. It turned out, however, that removing the subsumed clauses right after their detection is more beneficial for the overall subsumption process (with respect to the number of clauses/literals being removed).

Analyses of the benchmarks we used[2] showed that for SAT instances with less than 4,000,000 clauses the average length of clauses appearing in *ClauseSet* was approximately 4.5 with an average deviation of approximately 1.4. Less than 1% of the clauses were longer than 20. So we decided to use a very efficient sort algorithm which would only sort the clauses within *ClauseSet* which have a length below a certain threshold (called *sortLimit*). The very few clauses with a length longer than *sortLimit* are merely appended to the right end of *ClauseSet* without being sorted. The sort algorithm's complexity is linear in the size of *ClauseSet*. Its functionality is explained in Figure 1.

4 Results

4.1 Parameters

Our main goal was to examine whether the parallel subsumption algorithm really outperforms the sequential one. Furthermore, we were interested in whether the sorting of *ClauseSet* (see section 3.4) could improve the parallel algorithm's performance even further. There were two adjustable integer parameters in our experiment setting whose impacts on the algorithm's behaviour we wanted to examine. The first one was the parameter called *sortLimit* (see section 3.4). The second one was the threshold of *ClauseSet's* size, beyond which the program would use the parallel algorithm instead of the sequential one (see section 3.3). In the following, we will refer to this threshold as *csSizeLimit*.

sortLimit Choosing a small value for *sortLimit* results in a greater number of unsorted clauses in the rightmost end of *ClauseSet* after completion of the sort algorithm described in section 3.4. These are all clauses which have a length greater than or equal to *sortLimit*.

An excessively high *sortLimit* value would therefore undo the whole intention behind the sort algorithm leaving most clauses unsorted. An excessively low value, however, could be expected to be disadvantageous as well, as it could increase the *SortLists* vector to a size where the addressing effort would outweigh all other runtime savings.

As the analyses of our test benchmarks showed that the average size of all clauses appearing in *ClauseSet* was approximately 4.5 with an average deviation of approximately 1.4, we decided to try four different values for *sortLimit* in our tests: 5, 10, 20, and 30.

csSizeLimit As described in section 3.3, when the subsumption procedure needs to be executed, our program chooses whether to use the parallel or the sequential algorithm depending on the size of *ClauseSet*, because if *ClauseSet*

[2] SAT-Race 2006: `http://fmv.jku.at/sat-race-2006`
SAT 2007 competition: `http://www.satcompetition.org` (industrial)
SAT-Race 2008: `http://www-sr.informatik.uni-tuebingen.de/sat-race-2008`

```
vec<int> $SortLists[sortLimit]$;

// Copy all clause references from $ClauseSet$ into the sort vectors.
**for** all elements $i$ of $ClauseSet$ **do**
  **if** the $clauseLength$ of $ClauseSet[i]$ is smaller than $sortLimit$
    $SortLists[clauseLength]$.push($ClauseSet[i]$);
  **else**
    $SortLists[sortLimit-1]$.push($ClauseSet[i]$);
  **end if**
**end for**

// Clear $ClauseSet$.
$ClauseSet$.clear();

// Copy all clauses from the sort vectors into empty $ClauseSet$.
**for** $j = 0$ to $sortLimit$ **do**
  **for** all elements $i$ of $SortLists[j]$ **do**
    $ClauseSet$.push($SortLists[j][i]$);
  **end for**
**end for**
```

Fig. 1. The ***ClauseSet* sort algorithm** works with an array of *sortLimit* vector objects called *SortLists*. The algorithm runs once through all elements of *ClauseSet* and copies each element into the vector of *SortLists* corresponding to the length of the clause referred to by this element. After this is done, the vector at *SortLists*[0], for example, contains all clause references from *ClauseSet* to clauses of length 1, just as *SortLists*[1] contains those to clauses of length 2, and so forth. The last vector *SortLists*[*sortLimit*-1] contains all entries of *ClauseSet* refering to clauses whose lengths are greater than or equal to *sortLimit*. (The entries of *SortLists*[*sortLimit*-1] are unsorted but as mentioned in section 3.4 less than 1% of all clauses have a length greater than 20.) Now, *ClauseSet* can be cleared and the entries are copied back from the sort vectors beginning with the smallest ones.

is below a certain size, activating and maintaining the threads could require more effort than checking *ClauseSet* in the sequential manner. *csSizeLimit* is the parameter determining this threshold.

When we began our experiments we were primarily interested in results for SAT instances with less than 1,000,000 clauses (*typical*) and instances with 1,000,000 - 4,000,000 clauses (*large*). Analyses for these instances in our benchmark sets showed that each time, when subsumption is started, the size of *clauseSet* is on average 1781 for the typical instances and 95 for the large instances. This indicates that it must quite often (apparently more often for the large instances) be the case that the algorithm has to check rather small *ClauseSets*. Remember that subsumption is not only executed once for the whole CNF at the beginning of preprocessing but repeatedly for smaller sets of clauses in course of the overall preprocessing routine (see section 3.2). For our experiments we chose values for *csSizeLimit* ranging from 5,000 to 3,000,000.

4.2 Effects on reduction quality

As the parallel algorithm is deterministic (see section 3.3), changing *csSizeLimt* cannot have any effect on how much of the original CNF is reduced during preprocessing. The clauses in *ClauseSet* are always checked in the same order. Changing *sortLimit*, however, can have this effect because it determines how many clauses in *ClauseSet* will be sorted and thereby changes the order in which the clauses will be processed by the self-subsuming resulution procedure. (The order does not make a difference for subsumption.)

It could, however, be shown that the sorting of *ClauseSet* has no significant impact on the algorithm's capability of reducing the number of clauses or literals of the SAT instances. With or without sorting, the percentages by which both the number of clauses and the number of literals are reduced through preprocessing do not differ signifficantly (see Tables 1 and 2).

4.3 Effects on runtime

Knowing that the sorting of *ClauseSet* does not impair the reduction qualities of our preprocessing program, the next issue of interest was to examine whether the parallel algorithm would yield shorter runtimes than the sequential one and whether the sorting of *ClauseSet* would shorten the runtimes even further.

For the 329 typical SAT instances (less than 1,000,000 clauses) from our benchmark sets, we could show that using the parallel algorithm with a certain *csSizeLimit* value does improve the runtime. Sorting *ClauseSet*, however, only resulted in longer runtimes (see Figure 2).

For the 71 large instances (between 1,000,000 and 4,000,000 clauses) using the parallel algorithm could only yield a slight advantage over a predominantly sequential variant. Sorting *ClauseSet* did not appear to be as disadvantageous as for the typical instances but neither did it exhibit a noteworthy advantage over the "no sorting" variant (see Figure 3).

	average reduction of clauses in SAT instance				
	no sorting	sorting with *sortLimit* = 5	10	20	30
329 instances with <1,000,000 clauses	-47.4794%	-47.6286%	-47.6337%	-47.6343%	-47.6343%
71 instances with 1,000,000 – 4,000,000 clauses	-51.3497%	-51.3691%	-51.3692%	-51.3692%	-51.3692%
30 instances with >4,000,000 clauses	-10.0436%	n.a.	-10.0453%	n.a.	n.a.

Table 1. The precentage values indicate how much the numbers of clauses have been reduced through preprocessing. -47.4794%, for example, means that for the 329 SAT instances with initially less than 1,000,000 clauses their numbers of clauses could in average be reduced by 47.4794 percent of their original sizes when no sorting of *ClauseSet* was applied. As one can see, the reduction of clauses does only increase very slightly when *sortLimit* is increased. On the basis of our results for the instances with less than 4,000,000 clauses we decided against running tests with different *sortLimit* values for the very large SAT instances comprising more than 4,000,000 clauses.

	average reduction of literals in SAT instance				
	no sorting	sorting with *sortLimit* = 5	10	20	30
329 instances with <1,000,000 clauses	-35.2214%	-35.2191%	-35.2310%	-35.2316%	-35.2316%
71 instances with 1,000,000 – 4,000,000 clauses	-38.5203%	-38.5273%	-38.5280%	-38.5280%	-38.5280%
30 instances with >4,000,000 clauses	-6.6371%	n.a.	-6.5636%	n.a.	n.a.

Table 2. Similar to our findings presented in Table 1, the reduction of literals does also only increase very slightly when *sortLimit* is increased.

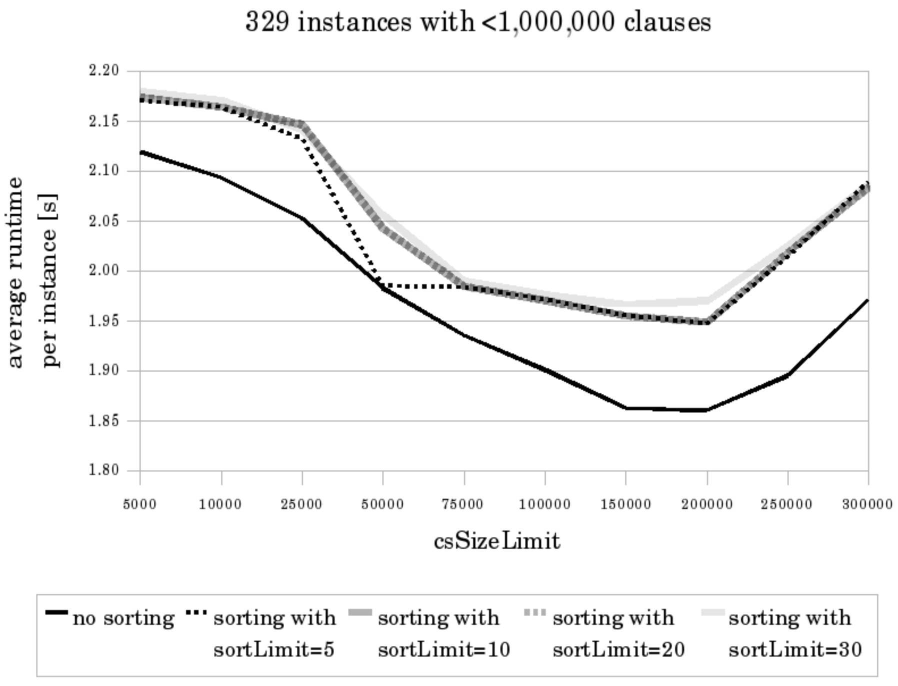

Fig. 2. For SAT instances with less than 1,000,000 clauses, the algorithm which does not sort the clauses of *clauseSet* is clearly faster than its sorting counter parts. The minimum average runtime for all the instances is achieved when *csSizeLimit* is set to some value between 150,000 and 200,000. If *csSizeLimit* is increased further than that, the runtime increases again. This indicates, as a higher value of *csSizeLimit* means less parallelisation, that the parallel algorithm performs better than a purely sequential one.

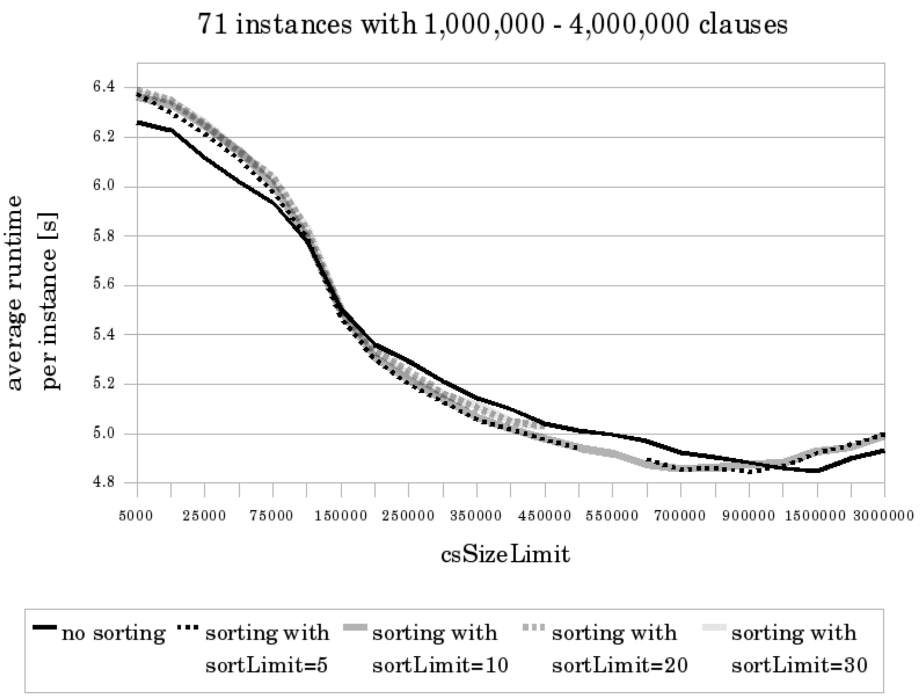

Fig. 3. For SAT instances with 1,000,000 - 4,000,000 clauses, the minimum average runtime is achieved at 4.84 s with the algorithm using the sort procedure with a *sortLimit* value of 5 and a *csSizeLimit* value of 900,000. The "no sorting" variant, however, performed almost as well (4.85 s) with a *csSizeLimit* of 1,500,000. Both these values of *csSizeLimit* are rather high considering that the average size of *ClauseSet* is 95 for these instances. The results imply that the overall program works best if the parallel algorithm is only applied very rarely. But still, the runtime increases again when *csSizeLimit* is set to a greater value than 1,500,000, which speaks at least for a small gain through parallelisation. (The *sortLimit*=20 and *sortLimit*=30 variants were only tested with *csSizeLimit* values up to 450,000.)

In the final phase of this research project we decided to expand our experiment by also running some tests on SAT instances from our benchmark sets with more than 4,000,000 clauses (*huge*). Having seen that there has been little difference between the algorithms using different *sortLimit* settings we only compared one variant using a *sortLimit* value of 10 to the "no sorting" variant. We also restrained ourselves to only four different *csSizeLimit* values ranging from 200,000 to 1,000,000.

The results show that the sorting of *ClauseSet* for these huge instances indeed has a beneficial effect on the runtime. Using parallelisation could also be shown to be catering for shorter runtimes as the average runtime decreases with the decrease of *csSizeLimit* (see Figure 4).

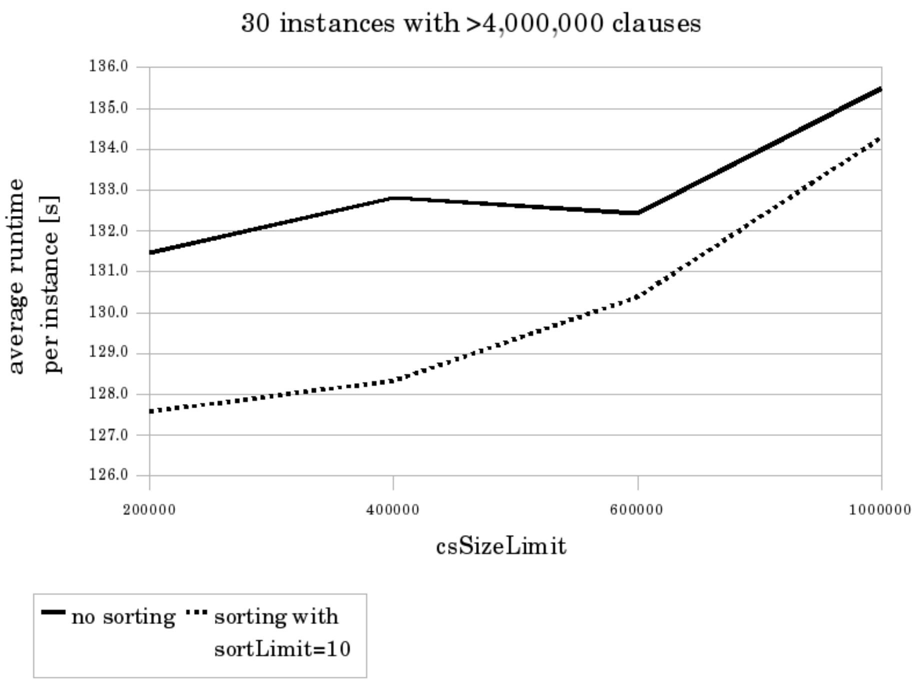

Fig. 4. In this rather non-exhaustive run of tests we found that the algorithm encorporating the sorting of *ClauseSet* achieved better runtimes than the one without sorting. Unfortunately, our values for *csSizeLimit* do not reveil the turning points of the curves, which might be located at values even smaller than 200,000.

5 Discussion

The results show that parallelisation is indeed capable of reducing the runtime of a subsumption/self-subsuming resolution algorithm as used in our program. Achieving those results, however, has been a very painstaking endeavour. If it had not been for the elaborate means by which we suspend and reactivate the threads, the overhead for maintaining the threads would have outweighed all runtime savings gained through paralellisation.

Moreover, as shown in section 4.3, it was vital to find an adequate ratio between the application of the parallel and the sequential algorithm in order to make parallelisation worthwhile. Using the parallel variant apperears to be only viable for rather large sets of clauses. However, it also seems to depend on the initial size of the SAT instances, what “large” means in this context. Remember, that we got different optimal *csSizeLimit* values for SAT instance of different initial sizes.

In future studies, one might try to further investigate to what extend the properties of SAT instances can be used to define in advance what degree of parallelisation is best suited for their preprocessing. The studies presented in this work have undoubtedly shown that parallelisiation has a potential for preprocessing.

Another promising prospect is that there are more preprocessing methods which are very similar to those of the subsumption algorithm. I.e. sets of clauses have to be traversed and checked for certain conditions. One of these other preprocessing methods for example is the one called *clause distribution* [8]. The same paralellisation principle we applied to the subsumption algorithm could be applied to those preprocessing methods, as well. Whether this will also lead to better runtime results is again something which has to be investigated by future studies.

References

1. F. Bacchus and J. Winter, *Effective preprocessing with hyper-resolution and equality reduction*, in SAT, 2003, pp. 341–355.
2. M. Davis, G. Logemann, and D. Loveland, *A machine program for theorem-proving*, Commun. ACM, 5 (1962), pp. 394–397.
3. N. Eén and A. Biere, *Effective preprocessing in SAT through variable and clause elimination*, in SAT, 2005, pp. 61–75.
4. I. Lynce and J. Marques-Silva, *Probing-based preprocessing techniques for propositional satisfiability*, in Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI ’03), Washington, DC, USA, 2003, IEEE Computer Society, p. 105.
5. J. P. Marques-Silva and K. A. Sakallah, *GRASP - A New Search Algorithm for Satisfiability*, in Proceedings of IEEE/ACM International Conference on Computer-Aided Design, 1996, pp. 220–227.
6. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, *Chaff: Engineering an Efficient SAT Solver*, in Proceedings of the 38th Design Automation Conference (DAC ’01), 2001.

7. T. Schubert, *MiraXT*. `http://ira.informatik.uni-freiburg.de/~schubert/html/miraxt.html`, 2008.
8. S. Subbarayan and D. K. Pradhan, *NiVER: Non increasing variable elimination resolution for preprocessing SAT instances*, in Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT), Springer, 2004, pp. 276–291.
9. G. S. Tseitin, *On the complexity of derivations in the propositional calculus*, Studies in Mathematics and Mathematical Logic, 2 (1968), pp. 115–125.
10. H. Zhang, *SATO: an efficient propositional prover*, in Proceedings of the International Conference on Automated Deduction (CADE '97), volume 1249 of LNAI, 1997, pp. 272–275.