Parallel MUS Extraction⋆

Anton Belov[1], Norbert Manthey[2], and Joao Marques-Silva[1,3]

[1] Complex and Adaptive Systems Laboratory University College Dublin
[2] Institute of Artificial Intelligence, Technische Universität Dresden, Germany
[3] IST/INESC-ID, Technical University of Lisbon, Portugal

Abstract. Parallelization is a natural direction towards the improvements in the scalability of algorithms for the computation of Minimally Unsatisfiable Subformulas (MUSes), and group-MUSes, of CNF formulas. In this paper we propose and analyze a number of approaches to parallel MUS computation. Just as it is the case with the parallel CDCL-based SAT solving, the communication, i.e. the exchange of learned clauses between the solvers running in parallel, emerges as an important component of parallel MUS extraction algorithms. However, in the context of MUS computation the communication might be unsound. We argue that the assumption-based approach to the incremental CDCL-based SAT solving is the key enabling technology for effective sound communication in the context of parallel MUS extraction, and show that fully unrestricted communication is possible in this setting. Furthermore, we propose a number of techniques to improve the quality of communication, as well as the quality of job distribution in the parallel MUS extractor. We evaluate the proposed techniques empirically on industrially-relevant instances of both plain and group MUS problems, and demonstrate significant (up to an order of magnitude) improvements due to the parallelization.

1 Introduction

A minimally unsatisfiable subformula (MUS) of an unsatisfiable CNF formula is any minimal, with respect to set inclusion, subset of its clauses that is unsatisfiable. MUSes, and the related *group-MUSes* [15,21], find a wide range of practical applications [21,3], and so the development of efficient MUS extraction algorithms is currently an active area of research (see [16] for a survey, [26,4,24] for recent developments). State-of-the-art MUS extraction algorithms use SAT solvers as $\mathcal{NP}$ oracles, and typically perform a large number of SAT solver calls — each call with a different subformula of the original input formula. The fact that many of these calls are independent suggests that MUS computation problem might be a good candidate for parallelization.

A number of successful approaches to the parallelization of SAT solving have been developed (see [13,19] for the recent overviews). In one of the widely used variants of parallel SAT solvers, namely *portfolio solvers*, several incarnations of a sequential solver, possibly with different configurations, are executed on the same input formula in

⋆ The first and third authors are financially supported by SFI PI grant BEACON (09/IN.1/I2618), and by FCT grants ATTEST (CMU-PT/ELE/0009/2009) and POLARIS (PTDC/EIA-CCO/123051/2010).

M. Järvisalo and A. Van Gelder (Eds.): SAT 2013, LNCS 7962, pp. 133–149, 2013.
© Springer-Verlag Berlin Heidelberg 2013

parallel. An essential component of portfolio solvers built on top of CDCL-based SAT solvers is the mechanism for the exchange of learned clauses — the *communication* — between the sequential sub-solvers. As such, it is plausible, and, as we show, indeed the case, that communication is an important aspect of any parallel MUS extraction solution geared towards industrially-relevant problems. However, while the communication is sound in portfolio-based parallel SAT solvers (since the sub-solvers work on the same input formula), this is not necessarily the case in a parallel MUS extractor, since now the formulas might differ.

In this paper we analyze a number of approaches to the parallel MUS computation. We notice that some of the simpler of these approaches result in performance degradation with respect to a sequential solution, however we do confirm the importance of communication for the parallel MUS extraction. More importantly, we argue that the *assumption-based approach* to the incremental CDCL-based SAT solving, introduced in [9], is the key enabling technology for effective sound communication in scalable parallel MUS extraction algorithms, and suggest that this might also be the case in more general settings where CDCL-based SAT solvers work on different related formulas in parallel. We carefully analyze the communication aspect in the context of parallel MUS extraction, and show that in this setting fully *unrestricted* communication is possible. Furthermore, we propose a number of effective clause filtering techniques, and an improved job distribution scheme based on the analysis of unsatisfiable cores. We evaluate the proposed algorithms and techniques empirically, and demonstrate significant speedups and scalability (e.g. median 2.94x, with up to 132x, speed up on 4 cores) on a set of industrially-relevant MUS and group-MUS extraction benchmarks.

2 Preliminaries and Background

We assume the familiarity with propositional logic, its clausal fragment, and commonly used terminology of the area of SAT. We focus on formulas in CNF (*formulas*, from hence on), which we treat as (finite) (multi-)sets of clauses. We assume that clauses do not contain duplicate variables. Given a formula F we denote the set of variables that occur in F by $Var(F)$, and the set of variables that occur in a clause $C \in F$ by $Var(C)$. An *assignment* τ for F is a map $\tau : Var(F) \to \{0,1\}$. Assignments are extended to formulas according to the semantics of classical propositional logic. If $\tau(F) = 1$, then τ is a *model* of F. If a formula F has (resp. does not have) a model, then F is *satisfiable* (resp. *unsatisfiable*). By $F|_\tau$ we denote the *reduct* of the formula F wrt. the assignment τ – that is the formula obtained from F by removing the satisfied clauses and falsified literals from the remaining clauses. The *resolution rule* states that, given two clauses $C_1 = (x \vee A)$ and $C_2 = (\neg x \vee B)$, the clause $C = (A \vee B)$, called the *resolvent* of C_1 and C_2, can be inferred by *resolving* on the variable x. We write $C = C_1 \otimes_x C_2$. Note that $\{C_1, C_2\} \models C$.

A CNF formula F is *minimally unsatisfiable* if (i) F is unsatisfiable, and (ii) for any clause $C \in F$, the formula $F \setminus \{C\}$ is satisfiable. The set of minimally unsatisfiable CNF formulas is denoted by MU. A CNF formula F' is a *minimally unsatisfiable subformula (MUS)* of a formula F if $F' \subseteq F$ and $F' \in \mathsf{MU}$. The set of MUSes of a CNF formula F is denoted by $\mathsf{MUS}(F)$. A clause $C \in F$ is *necessary* for F (cf. [14])

if F is unsatisfiable and $F \setminus \{C\}$ is satisfiable. Necessary clauses are often referred to as *transition* clauses. The set of all necessary clauses of F is precisely $\bigcap \mathsf{MUS}(F)$. Thus $F \in \mathsf{MU}$ if and only if every clause of F is necessary. The problem of deciding whether a given CNF formula is in MU is DP-complete [23]. Motivated by several applications, minimal unsatisfiability and related concepts have been extended to CNF formulas where clauses are partitioned into disjoint sets called *groups* [15,21].

The basic deletion-based MUS extraction algorithm operates in the following manner. Starting from an unsatisfiable formula F, the algorithm picks a clause $C \in F$, and tests the formula $F \setminus \{C\}$ for satisfiability. If the formula is unsatisfiable, C is removed from F, i.e. we let $F = F \setminus \{C\}$. Otherwise, C is necessary for F, and so for every unsatisfiable subformula of F, and hence C is included in the computed MUS. Once all clauses of the input formula F are tested for necessity in this manner, the remaining clauses constitute an MUS of F. While the basic deletion algorithm is neither theoretically nor empirically effective (see, for example, [16,4]), the addition of *clause-set refinement* and *model rotation*[17] makes it the top performing algorithm for industrially relevant instances. Clause-set refinement takes advantage of the capability of modern SAT solvers to produce an unsatisfiable core: since a core includes at least one MUS, all clauses outside the core can be removed from the formula after a single UNSAT outcome. Model rotation, on the other hand, allows to detect multiple necessary clauses in the case of SAT outcome: when $F \setminus \{C\} \in \mathsf{SAT}$, the model τ returned by the SAT solver serves as a witness of the necessity of C in F, and model rotation attempts to (cheaply) modify τ to obtain a witnesses for other clauses of F, possibly declaring multiple clauses of F necessary after a single SAT outcome.

Although the modern sequential CDCL-based SAT solvers are rooted in the DPLL algorithm [8], the addition of *clause learning* [18] and *back-jumping*, drastically changes the behaviour of the algorithm. We refer the reader to a tutorial introduction of CDCL in [25] that illuminates these changes. Additional enhancements to CDCL present in the modern SAT solvers include *restarts* [10], advanced data-structures and decision heuristics [20], and sophisticated heuristics to control the quality of learned clauses, based, for example, on the idea of *literal block distance* [2]. Many details of the modern CDCL-based SAT solving can be found in [7]. Although the specifics of the clause learning mechanism in CDCL-based SAT solvers are not crucial for the understanding of this paper, one aspect that is important is that new learned clauses are derived from the clauses of the input formula and the previously learned clauses used in a conflict via a sequence of resolutions steps. This ensures that the learned clauses are logically entailed by the clauses of the input formula.

A detailed overview of the modern parallel SAT solving systems can be found in [13,19]. Here, we focus on a widely used variant of parallel SAT solvers, namely *portfolio solvers*, as for example MANYSAT [12], PLINGELING [6] or PENELOPE [1]. This type of solvers execute several incarnations of a sequential solver, possibly with different configurations, on the same input formula in parallel. An important component of this type of solvers is the exchange of learned clauses between the sequential sub-solvers — the *communication*. The quality of the exchanged clauses becomes particularly important, and several heuristics have been proposed. In [12] only the clauses of a fixed size have been shared, while in [11] this *sharing filter* has been improved

further to a *quality based heuristic*: sharing limits are relaxed if not sufficiently many clauses are exchanged, and when too many clauses are shared, the sharing limits are tightened again.

Incremental SAT Solving. In many practical applications SAT solvers are invoked on a sequence of related formulas. The *incremental* SAT solving paradigm is motivated by the fact that the clauses learned from subformulas common to the successive formulas can be reused. The most widely used approach to the incremental SAT is the so-called *assumption-based* approach introduced in [9]. In this approach, a SAT solver provides an interface to add clauses, and an interface to determine the satisfiability of the set of currently added clauses, F, together with a user-provided set of *assumption* literals $A = \{a_1, \ldots, a_k\}$ – that is, to test the satisfiability of the formula $F \cup \bigcup_{a_i \in A}\{(a_i)\}$. An important feature of the approach of [9] is that the assumptions are *not* added to the formula prior to solving, but, instead, are used as the top-level decisions. As a result, any clause learned while solving the formula F under the assumptions A can be used to solve a formula $F' \supseteq F$ under a possibly different set of assumptions A'.

Assumption-based incremental SAT solving is often used to emulate arbitrary modifications to the input formula. Given a formula $F = \{C_1, \ldots, C_n\}$, the set $A = \{a_1, \ldots, a_n\}$ of fresh *assumption variables* is constructed (i.e. $Var(F) \cap A = \emptyset$), and the formula $F_A = \{(a_i \vee C_i) \mid C_i \in F\}$ is loaded into an incremental SAT solver. Then, for example, in order to establish the satisfiability of $F' \subseteq F$, the formula F_A is solved under assumptions $\{\neg a_i \mid C_i \in F'\} \cup \{a_j \mid C_j \notin F'\}$. In effect, the assumptions *temporarily* remove clauses outside of F'. If the outcome is `sat`, the model, restricted to $Var(F)$, is a model of F'. If the outcome is `unsat`, SAT solvers return a set of literals $A_{core} \subseteq A$ such that F_A is unsatisfiable under the assumptions $\{\neg a_i \mid a_i \in A_{core}\} \cup \{a_j \mid a_j \notin A_{core}\}$, and so the formula $\{C_i \mid a_i \in A_{core}\}$ is an *unsatisfiable core* of F'. Using the incrementality, any clause $C_i \in F$ can be removed *permanently* by adding the unit (a_i) to the SAT solver. Conversely, the addition of the unit $(\neg a_i)$ permanently asserts, or *finalizes*, C_i. Importantly, in this setting the negated assumptions are *not* resolved out of the learned clauses, whereas the negated unit clauses *are*. For example, if $\neg a_1$ and $\neg a_2$ are assumptions, and $(\neg a_3)$ is a unit clause, and if D is a learned clause whose derivation used clauses containing a_1, a_2, a_3, then the literals a_1 and a_2 are in D, whereas a_3 is resolved out of D. If a unit (a_2) is later added to the formula, then D is satisfied and is not used for further reasoning.

An alternative to the approach of [9], discussed extensively in [22], is to add assumptions as *temporary* unit clauses to the SAT solver's formula. To be usable for the subsequent incremental invocations, the clauses learned from any of these units must be extended with the negation of the assumption literals used to derive them. Although for the applications discussed in [22] this post-processing step pays off, as we argue shortly the approach of [9] appears to be the key to the efficient parallel MUS extraction.

3 Parallel MUS Extraction Algorithm

In this paper we describe a *low-level* parallelization of a particular MUS extraction algorithm, that is, while the high-level flow of the algorithm is unchanged, we off-load various satisfiability tests required by the algorithm to multiple threads. Clearly, such

low-level parallelization can be further integrated into a *high-level* parallel MUS extractor, that would run different (parallelized on the low-level) MUS extraction algorithms in parallel. Such high-level parallelization is a subject of future research.

The parallel MUS extraction algorithm proposed in this paper is based on the hybrid MUS extraction algorithm [17] (a variant of the deletion-based algorithm, augmented with the clause-set refinement and model rotation). The algorithm maintains a single master thread, and one or more worker threads. The master keeps a current snapshot of the working formula, and distributes the work items to the workers. A work item is simply a clause (or a group) that needs to be tested for necessity with respect to the current working formula. Thus, each worker owns a SAT solver, and given a work item $\langle F, C_i \rangle$, the responsibility of the worker is to invoke the SAT solver on the formula $F \setminus \{C_i\}$ and provide the result to the master. Once all available workers are started, the master waits for some or all of the workers to finish processing their work item. The results of finished workers are then aggregated, the master's working formula is updated, and any currently running workers whose work item's status has already been determined (the *redundant* workers) are aborted. Finally, the master proceeds to assign the next work items to the available workers, until no more work items are left.

There is a number of degrees of freedom within this framework: (i) *synchronous* vs. *asynchronous* execution — in the synchronous mode the master waits for all workers to finish their current task before advancing to the next iteration, while in the asynchronous execution the master processes the results as they come in from the workers; (ii) *work distribution* — whether the workers test the necessity of the same or a different clause; (iii) *communication between workers* — whether the workers are allowed to exchange the learned clauses or not. In the rest of the paper we denote various configurations by three letter acronyms: `S` (resp. `A`) for synchronous (resp. asynchronous) execution, followed by `S` (resp. `D`) for same (resp. different) clause distribution, followed by `N` (resp. `C`) for absence (resp. presence) of communication between the workers.

Perhaps the simplest reasonable configuration is the asynchronous execution on the same clause, i.e. all workers are given the same task $\langle F, C_i \rangle$ (the `AS_` configurations). This configuration is akin to portfolio-like solutions for parallel SAT solving in that it takes advantage of the fact that the run times of different incarnations of the same SAT solver working on the same formula may vary significantly. Since all workers test the same clause, once some workers are finished (there may be more than one), all others become redundant. Clearly, this configuration should benefit from communication — a comparison of the results for `ASN` and `ASC` configurations (Table 1 in Sec. 5) confirms that this is the case. Since in `ASC` all workers work on exactly the same formula $F \setminus \{C_i\}$, they can freely exchange learned clauses. The main drawback of `ASC` is that, despite the communication, the workers largely duplicate each other efforts. As a result, and since the parallel execution incurs a non-trivial overhead on the system (mostly due to memory accesses), this configuration performs worse than a sequential solution.

The next configurations we consider are those with synchronous execution on different clauses (the `SD_` configurations). Here the workers are given the tasks $\langle F, C_1 \rangle$, $\langle F, C_2 \rangle, \ldots$ with the goal to distribute the work of checking the necessity of the clauses between the workers. Note that since the master algorithm employs both the clause set refinement and the model rotation, some of the workers will end up executing

redundant tasks. We will address this drawback shortly, however meanwhile let us discuss the communication aspect of SDC. The main observation is that the communication between workers is *not* sound. To be specific, consider two workers $\texttt{W}_1$, $\texttt{W}_2$ working on the tasks $\langle F, C_1 \rangle$, and $\langle F, C_2 \rangle$, respectively. That is, $\texttt{W}_1$ is running its SAT solver on the formula $F \setminus \{C_1\}$, while $\texttt{W}_2$ is working on the formula $F \setminus \{C_2\}$. The problem now is that some clauses derived from C_1 by $\texttt{W}_2$ might not be valid logical consequences of the formula $F \setminus \{C_1\}$ solved by $\texttt{W}_1$. One could envision a number of ways to circumvent this problem. For example, we could prohibit $\texttt{W}_1$ from sending any clause derived from C_2 to $\texttt{W}_2$ (and vise versa), however in this case the workers need to be aware of what other workers are doing. Another option would be to force $\texttt{W}_1$ to refuse any clause derived from C_1 — this solution would require to augment every exchanged clause with some information regarding its origin, and to analyze every received learned clause. Finally, one could resort to an ($\mathcal{NP}$-complete) implication test for each received clause. Clearly, neither of these solutions are satisfactory. Yet, a natural solution does exist: use *assumption-based incremental SAT solvers*. Before we come back to this important point, we note another drawback of SD_ configurations: due to the fact that the run times of the SAT calls executed by workers may vary significantly, the algorithm waits for the completion of the longest running call while other workers are idle. Worse, it might be that the longest call is redundant given the results of some of the workers that finished their SAT calls faster. Thus, SD_ configurations might be hampered by a low CPU utilization, and a large percentage of "wasted" SAT calls.

It should be no surprise then, that a scalable parallel MUS extraction algorithm requires both the asynchronous execution, a work distribution strategy that reduces the duplication of workers' efforts, and sound communication — this the configuration ADC. The problem of sound communication in the asynchronous context not only remains, but becomes exacerbated, as we now might have a situation where $\texttt{W}_1$ is processing a work item $\langle F, C_1 \rangle$, while $\texttt{W}_2$ is working on $\langle F', C_2 \rangle$ with $F' \subset F$, and so the clauses derived from $F \setminus F'$ by $\texttt{W}_1$ might not be valid for $\texttt{W}_2$.

We now argue that assumption-based incremental SAT solving (i.e. the approach introduced in [9]), often seen as simply an implementation technique, is in fact a key enabling technology for the scalable parallel MUS extraction algorithms. Recall from Sec. 2 that in the incremental SAT setting the test of the satisfiability of some subformula F_1 of an input formula F can be performed without removing any of the clauses of F. Instead, the clauses outside of F_1 can be temporarily disabled using assumptions. As a result, clauses that are learned while analyzing a different subformula F_2 of F are valid (though might also be temporarily disabled) for the analysis of F_1. Consider now the organization of the parallel MUS extraction algorithm described above, whereby the workers are provided with *incremental* SAT solvers. During the initialization, all workers are given the same augmented formula $F_A = \{(a_i \vee C_i) \mid C_i \in F\}$ constructed by adding assumption literals to the clauses (or groups) of the input formula F. The workers invoke their incremental SAT solvers on F_A under a set of assumptions that represent the subformula assigned to them by the master. As the construction of an MUS of F progresses, the master determines that some clauses of F need to be either permanently removed or finalized — to achieve this, the master adds the corresponding unit clauses to the SAT solvers of the workers. Now, consider again the configuration

SDC — notice that since the execution is synchronous, all workers have *exactly the same* input formula $F_A \cup U$, where U is the set of unit clauses added to the formula in order to delete and to finalize some clauses. Since the only difference between the worker's execution is the set of assumptions under which they test the satisfiability of $F_A \cup U$, the workers are free to exchange the learned clauses, without any limitations and any additional reasoning. In the asynchronous setting, ADC, the situation is not quite straightforward, since then, while $\mathtt{W}_1$ is working on the formula $F_A \cup U$, the worker $\mathtt{W}_2$, which "ran ahead" of $\mathtt{W}_1$, might be working on the formula $F_A \cup U \cup U'$, where U' is a set of unit clauses added by the master since the beginning of execution of $\mathtt{W}_1$. Since the input formula of $\mathtt{W}_1$ is a subformula of the input formula of $\mathtt{W}_2$, the clauses learned by $\mathtt{W}_1$ will be valid for $\mathtt{W}_2$. However, it is not clear that $\mathtt{W}_2$ can send its learned clauses "back" to $\mathtt{W}_1$, since $\mathtt{W}_2$ has additional clauses in its formula. We will prove that fully *unrestricted* communication is sound in the asynchronous setting as well.

Notice that in the assumption-based incremental SAT setting the assumption literals provide an automatic way of *tagging* the learned clauses with the information of their origins. At the same time, SAT solving under assumptions automatically ensures that any clause previously learned from a currently disabled clause is disabled. The alternative approach to the incremental SAT, whereby the assumptions are added as temporary unit clauses (cf. Sec. 2), would, in our setting, require the reconstruction step described in [22] for every learned clause sent to another solver, which is not likely to scale. Our observations suggest that the approach of [9] to the assumption-based incremental SAT solving might be the key to the effective communication in more general scenarios where the CDCL-based SAT solvers work on different related formulas in parallel.

Formal Description of the Algorithm. By $F = \{C_1, \ldots, C_n\}$ we denote the input CNF formula, whose MUS M is to be computed. Let $A_F = \{a_1, \ldots, a_n\}$ be a set of fresh *assumption* variables (or, *assumptions*). Each assumption variable a_i will be implicitly associated with the clause C_i. As in Sec. 2, by F_A we will denote the formula $F_A = \{(a_1 \vee C_1), \ldots, (a_n \vee C_n)\}$. Given any $A \subseteq A_F$, let $cls(A) = \{C_i \mid a_i \in A\}$. Throughout the execution, the master maintains two sets of assumptions: the set of *necessary* assumptions $A_{nec} \subseteq A_F$ that corresponds to clauses that are declared to be necessary (i.e. part of the computed MUS), and the set of *unnecessary* assumptions $A_{unnec} \subseteq A_F$ that corresponds to the clauses that will not be included in the computed MUS. The sets A_{nec} and A_{unnec} are disjoint. For convenience, we let $A_{unk} = A_F \setminus (A_{nec} \cup A_{unnec})$ to denote the set of assumptions that correspond to clauses whose status is unknown. The state of the master is described by a pair $\langle A_{nec}, A_{unnec} \rangle$ of the sets of the necessary and the unnecessary assumptions. Given such a state pair (or, simply, a *state*) $S = \langle A_{nec}, A_{unnec} \rangle$, by $F(S)$ we denote the formula

$$F(S) = F_A \cup \{(\neg a_i) \mid a_i \in A_{nec}\} \cup \{(a_j) \mid a_j \in A_{unnec}\}. \tag{3.1}$$

The pseudocode of the algorithm is presented in Alg. 1. Each of the workers $\mathtt{W}_w$, $w = 1, \ldots, nw$, runs on its own thread, and has its own incremental SAT solver which is initialized with the formula F_A. As the master progresses, it adds negative units to finalize the necessary clauses, and positive units to remove the unnecessary clauses from the workers' SAT solvers. Since in the asynchronous configurations the formulas inside the SAT solvers may diverge, it will be convenient to view each worker $\mathtt{W}_w$ as

Algorithm 1. Parallel MUS extraction algorithm (with incremental SAT)

Input : F — unsatisfiable CNF Formula; nw — number of worker threads
Output: $M \in \mathsf{MUS}(F)$

```
1  initializeWorkers(F, {W_1, ..., W_nw})             // each W_i is a worker
2  ⟨A_nec, A_unnec⟩ ← ⟨∅, ∅⟩                            // initial state
3  while A_unk ≠ ∅ or there are running workers do    // A_unk ≜ A_F \ (A_nec ∪ A_unnec)
4      if A_unk ≠ ∅ then                              // there are untested clauses
5          foreach idle W_w do
6              a_i ← pickAssumption(A_unk)
7              W_w.updateState(⟨A_nec, A_unnec⟩)
8              W_w.startTask(a_i)
9      sleepUntilFinished()
10     Res = { W_w.getResult() | W_w is finished }
11     ⟨Δ_{A_nec}, Δ_{A_unnec}⟩ ← mergeResults(Res)
12     ⟨A_nec, A_unnec⟩ ← ⟨A_nec ∪ Δ_{A_nec}, A_unnec ∪ Δ_{A_unnec}⟩
13     abortRedundantWorkers()
14 return M ≜ cls(A_nec)                              // M ∈ MUS(F)
```

having its own version of a state-pair $S^w = \langle A^w_{nec}, A^w_{unnec}\rangle$, with $F(S^w)$ (as per 3.1) being exactly the set of input clauses in $\mathtt{W}_w$'s SAT solver. Details of the functions used in Alg. 1 are discussed below:

`pickAssumption(`A_{unk}`)`: for `_S_` configurations, the function picks $a_i \in A_{unk}$, and returns the same a_i for each invocation in the **foreach** loop on line 5; for `_D_` configurations, for each invocation the function returns a different $a_i \in A_{unk}$, if possible, and the last picked a_i if not.

$\mathtt{W}_w$`.updateState(`$\langle A_{nec}, A_{unnec}\rangle$`)`: sets S^w to be identical to S; on the implementation level this causes the addition of unit clauses to $\mathtt{W}_w$'s SAT solver.

$\mathtt{W}_w$`.startTask(`a_i`)`: starts $\mathtt{W}_w$'s SAT solver. If $S^w = \langle A^w_{nec}, A^w_{unnec}\rangle$ is the state of the worker $\mathtt{W}_w$ at the moment of invocation, the set of clauses in $\mathtt{W}_w$'s SAT solver is $F(S^w)$, and the SAT solver is invoked under assumptions $\{a_i\} \cup \{\neg a_j \mid a_{j\neq i} \in A_{unk}\}$. Thus, the worker tests whether the clause C_i, associated with the assumption a_i, is necessary for the formula $cls(A^w_{nec} \cup A^w_{unk})$. We will say that a_i is $\mathtt{W}_w$'s *task literal*, and that, until the SAT solver run has finished, $\mathtt{W}_w$ is processing its *task*.

`sleepUntilFinished()`: for synchronous configurations (`S__`) this function waits until all workers finished their tasks; for asynchronous configurations (`A__`) this function waits until at least one worker has finished its task (but there might still be more than one finished worker when this function returns).

$\mathtt{W}_w$`.getResult()`: retrieves the outcome of the SAT test performed by a finished worker $\mathtt{W}_w$. Since a worker's state might be out of sync with the master's, for notational convenience we assume that the returned result is a tuple $R^w = \langle S^w, a_i, st, \tau, A^w_{core}\rangle$, where S^w is the state of the worker, a_i is the worker's task literal, st is the outcome of the SAT call (`sat` or `unsat`), τ is the model returned by the SAT solver in case

$st = \texttt{sat}$, and $A^w_{core} \subseteq A^w_{unk}$ is a set of assumption literals in the final conflict clause in case $st = \texttt{unsat}$.

`mergeResults(`Res`)`: is responsible for the analysis of the set Res of the results of finished workers. The function returns a tuple $\langle \Delta_{A_{nec}}, \Delta_{A_{unnec}} \rangle$ of assumptions that correspond to the newly discovered necessary and unnecessary clauses, initialized with $\langle \emptyset, \emptyset \rangle$. For each $R^w = \langle S^w, a_i, st, \tau, A^w_{core} \rangle \in Res$ with $st = \texttt{sat}$, the function appends a_i to $\Delta_{A_{nec}}$, and executes the model rotation algorithm on the formula $cls(A_{nec} \cup A_{unk})$ with the assignment τ to detect additional necessary clauses. For each such clause, the corresponding assumption variable is added to $\Delta_{A_{nec}}$. For each result tuple with $st = \texttt{unsat}$ the function first checks whether A^w_{core} is a subset of A_{unk} — in the asynchronous mode this might not be the case, since the state of the worker S^w might be out-of-date with respect to the state of the master. All `unsat` results for which $A^w_{core} \not\subseteq A_{unk}$ are discarded, and from the remaining `unsat` results one set A^w_{core} is selected[1]. The function then sets $\Delta_{A_{unnec}}$ to be $A_{unk} \setminus A^w_{core}$.

`abortRedundantWorkers()`: aborts all workers whose task literal is not in A_{unk}. The learned clauses accumulated by a worker's SAT solver remain in the solver.

Proof of Correctness and the Soundness of Unrestricted Communication. The correctness of the presented algorithm hinges on the following loop invariant.

Invariant 3.1. *For $v = 1, \ldots,$ let $S_v = \langle A^v_{nec}, A^v_{unnec} \rangle$ denote the state of the master prior to the v-th test of the main loop guard (line 3 of Alg. 1). Then, the formula $cls(A^v_{nec} \cup A^v_{unk})$ is unsatisfiable, and every clause in $cls(A^v_{nec})$ is necessary for it.*

To prove the invariant we need to establish a correctness property of the results returned by the finished workers. The property holds trivially in the configurations without communication, however a subtlety arises when the *unrestricted* communication is enabled. This is best demonstrated by the following example.

Example 1. Let the (already augmented) formula F_A be $\{(a_1 \vee \neg b \vee \neg c \vee \neg d), (a_2 \vee \neg b \vee c \vee d), (a_3 \vee b \vee c \vee \neg d), (a_4 \vee b \vee c \vee d), (a_5 \vee \neg c \vee \neg d), (a_6 \vee c \vee \neg d), (a_7 \vee \neg c \vee d)\}$. Assume that there are three workers, and $\texttt{W}_1$ already determined that C_1 is unnecessary:

$\texttt{W}_w$	Task	Assumptions	A^w_{nec}	A^w_{unnec}
$\texttt{W}_1$	a_4	$\{\neg a_2, \neg a_3, a_4, \neg a_5, \neg a_6, \neg a_7\}$	$\{\}$	$\{a_1\}$
$\texttt{W}_2$	a_2	$\{\neg a_1, a_2, \neg a_3, \neg a_4, \neg a_5, \neg a_6, \neg a_7\}$	$\{\}$	$\{\}$
$\texttt{W}_3$	a_3	$\{\neg a_1, \neg a_2, a_3, \neg a_4, \neg a_5, \neg a_6, \neg a_7\}$	$\{\}$	$\{\}$

Assume $\texttt{W}_1$ finishes its task first: it returns `sat`, and a model τ that witnesses the clause C_4. In `mergeResults()` the master applies model rotation, and determines that C_2 is also necessary. The master now has $A_{nec} = \{a_2, a_4\}$ and $A_{unnec} = \{a_1\}$. Since $\texttt{W}_2$'s task is a_2, it becomes redundant and is aborted. Assume $\texttt{W}_1$ is given a_5, and $\texttt{W}_2$ some other task (not shown). Note that the master adds the units $\{(\neg a_2), (\neg a_4)\}$ to $\texttt{W}_1$'s solver, and the units $\{(a_1), (\neg a_2), (\neg a_4)\}$ to $\texttt{W}_2$'s, prior to the call.

$\texttt{W}_1$	a_5	$\{\neg a_3, a_5, \neg a_6, \neg a_7\}$	$\{a_2, a_4\}$	$\{a_1\}$

Then, during conflict analysis the learned clause $(c \vee d)$ can be generated by $\texttt{W}_1$ from $(a_2 \vee \neg b \vee c \vee d)$, $(a_4 \vee b \vee c \vee d)$, and the two units $(\neg a_2)$, $(\neg a_4)$. When this clause

[1] In our implementation we select a set A^w_{core} of the smallest size.

is received by $\mathtt{W}_3$, it learns $(a_5 \vee a_6 \vee a_7)$ from $F(S^3)$ by resolving $(c \vee d)$ with $(a_5 \vee \neg c \vee \neg d)$, $(a_6 \vee c \vee \neg d)$ and $(a_7 \vee \neg c \vee d)$, resulting in the set $A^3_{core} = \{a_5, a_6, a_7\}$. But, the formula $cls(A^3_{core})$ is satisfiable ! As we see shortly, the master must conjoin A^3_{core} with A_{nec} to get the "real" unsatisfiable core. ■

Lemma 1. *Let $S^w = \langle A^w_{nec}, A^w_{unnec}\rangle$ be the state of a worker $\mathtt{W}_w$ at the time of the invocation of the function $\mathtt{W}_w$ `.startTask(`a_i`)`. Let A be any subset of $A^w_{unk} \setminus \{a_i\}$, and D be any set of clauses implied by the formula $F(S^w) \cup \{(\neg a_j) \mid a_j \in A\}$. Furthermore, let $\langle st, \tau, A^w_{core}\rangle$ be the outcome of a SAT solver call on the formula $F(S^w) \cup D$ under the set of assumptions $P = \{a_i\} \cup \{\neg a_j \mid a_{j \neq i} \in A^w_{unk}\}$. Then,*

(i) if $st = \mathtt{sat}$, then the formula $F(S^w)$ is satisfiable P, and τ is a model of $F(S^w)$ (that respects P).
(ii) if $st = \mathtt{unsat}$, then the formula $F(S^w)$ is unsatisfiable under the assumptions $P' = \{\neg a_j \mid a_j \in A \cup A^w_{core}\}$.

The set D in Lemma 1 is intended to represent the set of "extra" clauses that a worker $\mathtt{W}_w$ has received from other workers during the execution of its task, and the set $A \subseteq A^w_{unk}$ of assumptions to correspond to the clauses that were discovered to be necessary by the master since $\mathtt{W}_w$ started its task. Notice the addition of the set A to the set of assumptions P' in part (ii) of the lemma (cf. Example 1). We now argue that all clauses received by any worker $\mathtt{W}_w$ satisfy the condition of Lemma 1.

Lemma 2. *Let $S^w = \langle A^w_{nec}, A^w_{unnec}\rangle$ be the state of a worker $\mathtt{W}_w$ that has successfully completed its task a_i (i.e. it has not been aborted by the master), and let $S = \langle A_{nec}, A_{unnec}\rangle$ be the state of the master by the time it calls $\mathtt{W}_w$ `.getResult()` (line 10, Alg. 1). Then, for every clause C in $\mathtt{W}_w$'s SAT solver,*

$$F(S^w) \cup \{(\neg a_j) \mid a_j \in A_{nec} \setminus A^w_{nec}\} \models C. \tag{3.2}$$

Proof (sketch). The complete proof involves an inductive argument on the global sequence of generated learned clauses. The base case is the non-trivial part of the proof and is established using the following observations. Let $\mathtt{W}_1$ and $\mathtt{W}_2$ be two workers with the states S^1 and S^2 respectively, such that $S^1_{unk} \supset S^w_{unk} \supset S^2_{unk}$, i.e. $\mathtt{W}_1$ is "behind" $\mathtt{W}_w$ and $\mathtt{W}_2$ is "ahead" of $\mathtt{W}_w$. Since $F(S^1) \subset F(S^w)$, any clause C learned by $\mathtt{W}_1$ from its input formula satisfies (3.2). Let C be a clause learned by $\mathtt{W}_2$ from its input formula $F(S^2)$, i.e. C is implied by the formula

$$F(S^2) = F(S^w) \cup \{(\neg a_j) \mid a_j \in A^2_{nec} \setminus A^w_{nec}\} \cup \{(a_k) \mid a_k \in A^2_{unnec} \setminus A^w_{unnec}\}.$$

The assumptions a_k occur only positively in $F(S^2)$, so, due to the units (a_k), clauses with a_k will not be used as conflict. Hence, a_k do not occur in C, and so C is implied by

$$F(S^w) \cup \{(\neg a_j) \mid a_j \in A^2_{nec} \setminus A^w_{nec}\}.$$

We conclude that (3.2) holds by taking into account that $A^2_{nec} \subseteq A_{nec}$. □

Lemma 3 below establishes the correctness of the results returned by the workers by putting together Lemmas 1 and 2, and taking into account the fact that for any worker

$\mathtt{W}_w$ with state S^w that has *successfully* completed a task a_i, we have that $a_i \in A_{unk}$, as otherwise the master would have aborted $\mathtt{W}_w$ during `abortRedundantWorkers()` call at the end of the *previous iteration*. In particular, $a_i \notin (A_{nec} \setminus A^w_{nec})$, and so the set $A_{nec} \setminus A^w_{nec}$ from (3.2) satisfies the condition imposed on the set A in Lemma 1. Notice that in Example 1, if $\mathtt{W}_2$ was *not* aborted when a_2 was found necessary, on the reception of the clause $(c \vee d)$ from $\mathtt{W}_1$ it would return `unsat`, instead of `sat`.

Lemma 3. *Let* $S^w = \langle A^w_{nec}, A^w_{unnec} \rangle$ *be the state of a worker* $\mathtt{W}_w$ *that has successfully completed its task* a_i*, and let* $S = \langle A_{nec}, A_{unnec} \rangle$ *be the state of the master by the time it obtains* $\mathtt{W}_w$*'s result tuple* $R = \langle S^w, a_i, st, \tau, A^w_{core} \rangle$*. Then, if* $st =$ `sat`*, then* τ *is a model of the formula* $cls(A_{nec} \cup A_{unk}) \setminus \{C_i\}$*; if* $st =$ `unsat`*, then the formula* $cls(A_{nec} \cup A^w_{core})$ *is unsatisfiable.*

Using Lemma 3 and the definition of `mergeResults()` we establish Invariant 3.1.

Theorem 1. *Algorithm 1 terminates on any unsatisfiable input formula* F*, and the set* $M = cls(A_{nec})$ *returned by the algorithm is an MUS of* F*.*

SAT Solver Modifications. To exchange the learned clauses, a globally accessible clause pool, implemented as a ring buffer, is created. Each incremental SAT solver incarnation adds its learned clauses to the pool and receives the clauses submitted by other solvers. A solver incarnation sends a learned clause immediately after its generation if the clause passes the heuristic filters (discussed below). Clauses are received from the pool prior to a decision on decision level 0.

Improving Communication. Although, as shown above, the unrestricted communication is sound, clause sharing has to be restricted due to following reasons: (i) the received clauses might be redundant; (ii) additional clauses slow down the reasoning of a SAT solver incarnation; (iii) the usefulness of the new clauses cannot be determined in advance. Thus, we restrict the communication to the clauses that appear to be *promising* by adding two sharing filters to the system: learned clauses are shared if (i) their size or (ii) the *literal block distance* (LBD) [2] are less than a certain threshold. The default configuration uses a size limit of 10 literals and an LBD limit of 5. Following the ideas in [11] we also added a configuration `DYN`, in which these sharing thresholds are controlled dynamically. As previously discussed, the exchanged learned clauses may include a large number of assumption literals, which affect both the size and the LBD value of the clauses. For example, a clause with a single non-assumption literal and a large number of assumption literals might be filtered out due to its size. Clearly, such clause will be extremely useful to other solvers, as it might trigger unit propagation once the assumptions are assigned. Thus, from the filtering point of view, the assumption literals are superfluous, and so we added the configuration `PRASS` in which these literals are ignored ("protected") in the analysis by the sharing filters. Since the activity of learned clauses is initialized, we also allow this for received clauses (`BUMP`).

Core-Based Scheduling. Work duplication remains an important problem in our algorithm. To reduce the duplication we implemented a scheduling scheme based on the analysis of unsatisfiable cores returned by the workers. The scheme relies on the intuitive observation that clauses that appear in the intersection of unsatisfiable cores during

the execution of the algorithm (including those discarded by mergeResults()) are likely to be necessary. In *core-based scheduling* CBS we prioritize clauses based on their core membership count.

4 Related Work

To our knowledge the only published work on the parallelization of MUS extraction algorithms is the recently published papers by Wieringa [26] and Wieringa and Heljanko [27]. In both papers an MUS extractor is built on top of a parallel incremental SAT solver. As the focus of [26] is the analysis of model rotation, the parallelization aspects, and importantly, the communication aspects, are not discussed in sufficient detail. In [27] the authors present a parallel incremental SAT solver Tarmo, and use the MUS extraction problem as a case study to demonstrate its effectiveness. To this extent, the authors implemented the MUS extraction algorithm described in [26] on top of Tarmo — we will refer to this combination as TarmoMUS, as this is the name of the MUS extractor distributed by the authors. The essential difference between the algorithm proposed in this paper, and TarmoMUS is that in our algorithm the communication is unrestricted (modulo the filtering techniques discussed above), whereas in TarmoMUS the communication is restricted to be “forward” only. This restriction both incurs an additional implementational overhead, and reduces the quality and usefulness of exchanged clauses. In Sec. 5 we demonstrate that our algorithm scales significantly better that of [27] — we attribute this difference to the unrestricted communication. Additional important technique in our algorithm that is absent from TarmoMUS is the assumption “protection” during clause filtering.

5 Experimental Evaluation

The algorithm described in this paper was implemented in C++ with pthreads, and the resulting tool, pMUSer2, was evaluated on a subset of benchmarks used in the MUS track of SAT Competition 2011. The subset consists of 175 MUS and 201 group-MUS instances on which the sequential MUS extractor MUSer2 [5] takes more than 10 seconds of CPU time. The experiments were performed on an HPC cluster, where each node is a dual quad-core Intel Xeon E5450 3 GHz with 32 GB of memory. All tools were run with a timeout of 1800 seconds and a memory limit of 16 GB per input instance. All communicating configurations use the PRASS option by default.

Table 1 summarizes the results of various configurations of the parallel, as well as the sequential, algorithms. Clearly, adding communication to the asynchronous configurations of the increases the performance in terms of solved instances and overall solving time. The high average run time of the ADC configuration can be explained with the overhead introduced of less useful shared clauses. The table also shows clearly that using incremental SAT as the basis for parallel MUS is essential – without this technique only the configurations AS$_-$ would be possible, but these two configurations show the worst performance. For the AS$_-$ and SD$_-$ configurations, also a slight decrease in the calls to the SAT solver can be recognized. Note, that this number of calls includes all solved instances, so that the increase from ADN to ADC is still plausible: the additional

Table 1. The table compares different algorithm configurations and shows the following statistic; the number of *solved instances* from the benchmark set; the *average run time* for the full benchmark set (including timeout instances) and, for parallel solvers, the *average CPU utilization*; the *penalized runtime*, i.e. the runtime of solving the full benchmark set with a penalty of factor 10 added for all instances that could not be solved; the total number of *sent clauses* shows how many learned clauses are provided for other solver incarnations; the total number of *SAT solver calls*; the percentage of SAT calls whose result has been ignored (*wasted calls*); the percentage of calls that have been *aborted*. The last column indicates whether the results are for plain MUS or for group MUS instances.

Configuration	Solved Instances	average run time	average cpu utilization	penalized run time	sent clauses	SAT solver calls	wasted solver calls	aborted solver calls	groups
MUSer2	144	186.46	100	590 K	–	413 K	–	–	–
ASN(4)	135	157.01	80.30	747 K	–	1458 K	21.80	54.13	–
ASC(4)	137	146.37	80.65	709 K	433 K	1453 K	21.88	54.04	–
SDN(4)	143	154.93	47.27	603 K	–	517 K	15.91	–	–
SDC(4)	141	138.31	46.70	636 K	433 K	512 K	16.19	–	–
ADN(4)	146	126.45	91.90	544 K	–	488 K	5.76	11.70	–
ADC(4)	150	154.09	90.79	476 K	1186 K	602 K	5.84	12.26	–
ADC-PRASS(4)	148	104.95	90.15	504 K	548 K	585 K	5.15	12.12	–
ADC+DYN+BUMP(4)	153	133.98	91.55	419 K	7071 K	661 K	7.01	11.83	–
ADC(8)	151	128.05	87.76	454 K	867 K	672 K	7.78	13.30	–
ADC+CBS(8)	151	117.22	86.71	452 K	1283 K	552 K	4.10	3.53	–
ADC+DYN+BUMP+CBS(8)	155	136.35	88.27	383 K	5564 K	635 K	5.56	3.13	–
MUSer2	194	123.56	100	150 K	–	325 K	–	–	✓
ADC(4)	198	106.92	90.19	75 K	487 K	530 K	14.35	33.39	✓
ADC+DYN+BUMP+CBS(4)	197	110.11	93.79	94 K	1688 K	350 K	11.68	7.29	✓
ADC(8)	198	100.76	86.99	74 K	333 K	692 K	20.93	33.26	✓
ADC+DYN+BUMP+CBS(8)	198	88.52	88.39	71 K	1178 K	439 K	9.92	8.66	✓

SAT solver calls depend on the additionally solved instances. Since the communication speeds up single SAT calls, more solver calls are wasted, because a competing solver has finished its task faster and thus aborts redundant solvers. Note, that the CPU utilization of the `SD_` (synchronous) configurations is almost half of that of asynchronous configurations. Therefore, using asynchronous SAT solver calls is important to the scalability of the parallel algorithm and further motivates the analysis of communication for this setting. Comparing the results of the basic configurations with the sequential solver `MUSer2` we note that only the configurations `AD_` improve the performance from 144 to 150 solved instances with an improved average run time.

In Sec. 3 we discussed several improvements of the algorithm, which are also evaluated in Table 1. Disabling the `PRASS` option reduces the performance, and also reduces the number of shared clauses significantly – underlining that ignoring assumption literals is a must for successful clause sharing. Optimizing clause sharing with `BUMP` and `DYN` improves `ADC` further to 153 solved instances and an improved average run time. Note that this configuration also shares significantly more clauses than `ADC`, but

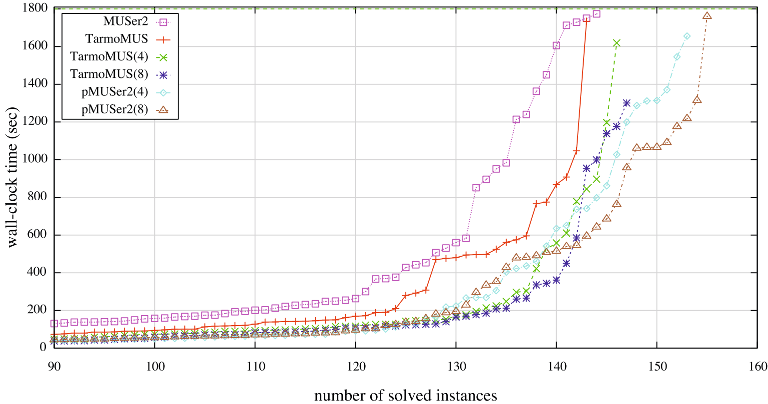

Fig. 1. Comparison of extractors on plain MUS instances

wastes more SAT calls. The core-based scheduling heuristic CBS does not improve the performance on 4 cores. The best four core configuration ADC+DYN+BUMP is referred as pMUSer2(4). In the eight core setting, the addition of CBS improves the number of SAT calls and reduces the percentage of wasted and aborted calls, without improving the overall performance significantly compared to ADC. However, adding sharing optimizations DYN and BUMP to ADC+CBS improves the overall performance: four more instances can be solved, the average run time decreases and the number of wasted and aborted SAT calls is also smaller than for any other eight core configuration. The best eight core configuration we found is ADC+DYN+BUMP+CBS, which we refer to as pMUSer(8). The behaviour of the parallel MUS extraction algorithm in the context of group-MUS extraction is quite similar. Again, the configuration ADC gives the best results and can solve already 198 out of 201 instances. For four cores, adding sharing or scheduling optimizations does not increase the performance, but again CBS helps to reduce the number of SAT calls as well as wasted and aborted SAT calls and DYN increases the number of shared clauses. When adding more cores, also 198 instances can be solved also by ADC+DYN+BUMP+CBS – suggesting that if the timeout were increased slightly, the four core variant could have solved these instances as well.

Figure 1 depicts the comparative behaviour of MUSer2 and pMUSer with 4 and 8 cores on the plain MUS instances. In addition, we evaluated the parallel MUS extractor TarmoMUS [27], discussed in Sec. 4. While in the sequential mode TarmoMUS is notably faster than MUSer2[2], the plot demonstrates that our algorithm scales significantly better with the number of cores, than TarmoMUS. For example, already a 4-core configuration of pMUSer2 outperforms the 4-core configuration of TarmoMUS. The statistics to compare the scalability of the algorithms are presented in Table 2. For both average and median speedup pMUSer2 gives much better results on plain instances. From sequential to four cores, MUSer2 scales linear in average. Obtained speedups range from 0.49 up to 132.59, showing that the parallelization can result in super-linear speedups

[2] Our analysis suggests that this is due to different versions of the SAT solvers used by the tools.

Table 2. Relative speedup with the addition of parallel resources: *minimum*, *average*, *maximum* and *median*. As a basis for the calculation the *commonly* solved instances have been used.

Solver 1	Solver 2	Min.	Avg.	Max.	Median	Common	Groups
TarmoMUS	TarmoMUS(4)	0.55	1.44	4.40	1.17	141	–
TarmoMUS	TarmoMUS(8)	0.41	1.74	6.92	1.29	141	–
MUSer2	pMUSer2(4)	0.49	4.09	132.59	2.94	143	–
MUSer2	pMUSer2(8)	0.28	4.01	97.66	3.38	142	–
MUSer2	pMUSer2(4)	0.46	1.41	4.07	1.33	194	✓
MUSer2	pMUSer2(8)	0.44	1.88	9.20	1.49	194	✓

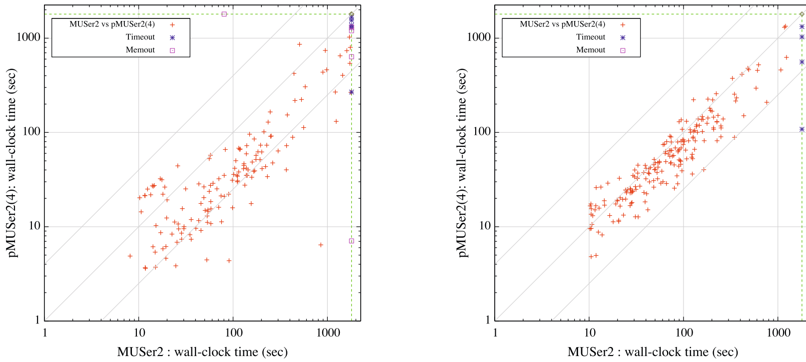

Fig. 2. Wall-clock time, sequential vs. 4 cores: left — plain MUS; right – group MUS

(consider also the scatter plots in Figure 2). For `TarmoMUS` the average, maximum and median speed-ups are lower, and when more resources are added, the performance increases only slightly. Neither `pMUSer2` nor `TarmoMUS` scale well to eight cores.

6 Conclusion

We argued that assumption-based incremental SAT solving is essential to ensuring the scalability of the proposed parallel MUS extraction algorithm. We proved the soundness of unrestricted communication in our algorithm, and proposed a number of optimizations focused on improving the quality of communication and job distribution. While the algorithm scales extremely well from a single-core to the 4-core setting, we did not observe similar improvements going from the 4-core setting to the 8-core. In our view, the main obstacle to the scalability to a high number of cores is the fact that as the number of cores grows, the workers are more likely to duplicate each others efforts. While the situation is somewhat improved by the core-based scheduling, the solution is not yet satisfactory, and requires further research. Additional avenue for improvement might lie in the high-level parallelization, whereby different (parallelized on the low-level) MUS extraction algorithms are executed in parallel.

Acknowledgements. We thank Allen Van Gelder and the anonymous referees, for their comments and suggestions that helped us to improve the presentation of our ideas.

References

1. Audemard, G., Hoessen, B., Jabbour, S., Lagniez, J.-M., Piette, C.: Revisiting clause exchange in parallel SAT solving. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 200–213. Springer, Heidelberg (2012)
2. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: Proceedings of the 21st International Jont Conference on Artifical Intelligence, IJCAI 2009, pp. 399–404. Morgan Kaufmann Publishers Inc., San Francisco (2009)
3. Belov, A., Chen, H., Mishchenko, A., Marques-Silva, J.: Core minimization in SAT-based abstraction. In: DATE (2013)
4. Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Communications 25(2), 97–116 (2012)
5. Belov, A., Marques-Silva, J.: MUSer2: An efficient MUS extractor. Journal of Satisfiability 8, 123–128 (2012)
6. Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. FMV Report Series Technical Report 10/1, Johannes Kepler University, Linz, Austria (2010)
7. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
8. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM 5, 394–397 (1962)
9. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. 89(4), 543–560 (2003)
10. Gomes, C.P., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. Journal of Automated Reasoning 24(1-2), 67–100 (2000)
11. Hamadi, Y., Jabbour, S., Sais, L.: Control-based clause sharing in parallel sat solving. In: Proceedings of the 21st International Jont Conference on Artifical Intelligence, pp. 499–504. Morgan Kaufmann Publishers Inc., San Francisco (2009)
12. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. JSAT 6(4), 245–262 (2009)
13. Hölldobler, S., Manthey, N., Nguyen, V.H., Stecklina, J., Steinke, P.: A short overview on modern parallel SAT-solvers. In: Wasito, I., et al. (eds.) Proceedings of the International Conference on Advanced Computer Science and Information Systems, pp. 201–206 (2011) ISBN 978-979-1421-11-9
14. Kullmann, O., Lynce, I., Marques-Silva, J.: Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable sub-clause-sets and the lean kernel. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 22–35. Springer, Heidelberg (2006)
15. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reasoning 40(1), 1–33 (2008)
16. Marques-Silva, J.: Minimal unsatisfiability: Models, algorithms and applications. In: ISMVL, pp. 9–14 (2010)
17. Marques-Silva, J., Lynce, I.: On improving MUS extraction algorithms. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 159–173. Springer, Heidelberg (2011)
18. Marques Silva, J.P., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Trans. Computers 48(5), 506–521 (1999)
19. Martins, R., Manquinho, V., Lynce, I.: An overview of parallel SAT solving. Constraints 17, 304–347 (2012)

20. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. 38th Annual ACM/IEEE Design Automation Conf. (DAC), pp. 530–535. ACM (2001)
21. Nadel, A.: Boosting minimal unsatisfiable core extraction. In: FMCAD, pp. 121–128 (October 2010)
22. Nadel, A., Ryvchin, V.: Efficient SAT solving under assumptions. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 242–255. Springer, Heidelberg (2012)
23. Papadimitriou, C.H., Wolfe, D.: The complexity of facets resolved. J. Comput. Syst. Sci. 37(1), 2–13 (1988)
24. Ryvchin, V., Strichman, O.: Faster extraction of high-level minimal unsatisfiable cores. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 174–187. Springer, Heidelberg (2011)
25. Van Gelder, A.: Generalized conflict-clause strengthening for satisfiability solvers. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 329–342. Springer, Heidelberg (2011)
26. Wieringa, S.: Understanding, improving and parallelizing MUS finding using model rotation. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 672–687. Springer, Heidelberg (2012)
27. Wieringa, S., Heljanko, K.: Asynchronous multi-core incremental SAT solving. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 139–153. Springer, Heidelberg (2013)