Parallelizing Partial MUS Enumeration

Wenting Zhao
and Mark H. Liffiton
Illinois Wesleyan University, Bloomington IL 61701, USA
{wzhao,mliffito}@iwu.edu
http://www.iwu.edu/~mliffito/marco/

***Abstract*—The problem of enumerating minimal unsatisfiable subsets of a constraint system (MUSes) is a natural candidate for parallelization: as an enumeration problem, it allows for concurrent solving of independent subproblems, and as a typically intractable problem w.r.t. completion (which parallelization cannot transcend), the speed or rate of output (which parallelization *can* improve) is often the most important performance characteristic. In this work, we explore the parallelization of partial MUS enumeration (aiming to enumerate *some* MUSes within given resource constraints) via two extensions to a recently-developed sequential algorithm – one employing an existing parallel single-MUS extraction algorithm, the other parallelizing the entire enumeration algorithm – and we discuss variants and implementation details as well. Results of experiments run with up to 16 cores show that the full parallelization of the entire enumeration algorithm scales well, reaching an average of 92% of perfect scaling with 4 cores and 70% at 16 cores. Evaluating variants and implementation details illuminates how those choices impact performance, including a potentially counterintuitive result that sharing results between threads to avoid duplicate work is not beneficial in the general case.**

I. Introduction

With multi-core processors now commonplace, algorithms must be parallelized to make full use of the computing resources available in a given system. Today, four-core processors are typical in consumer-level machines, 16-core configurations and beyond are common in servers and workstations, and core counts will continue to increase as CPU manufacturing processes improve. Compared to a sequential algorithm running on a single core, a parallel algorithm on a k-core machine could provide up to k times the performance on average, and while such perfect scaling is not often achievable, it provides an ideal against which parallel algorithms can be measured.

In this work, we explore parallel algorithms for the *partial minimal unsatisfiable subset enumeration problem* (partial MUS enumeration). Given an unsatisfiable constraint system, an MUS of that system can be seen as an explanation of its infeasibility, and MUSes are used in a wide range of applications (for a survey, see [3]). Enumeration problems with a large search space like MUS enumeration lend themselves well to parallelization; enumeration can often be decomposed into many independent subproblems, which often allows parallel algorithms to exhibit better scaling, making efficient use of multiple cores. Additionally, MUS enumeration is typically intractable with respect to completion – a constraint system with n constraints can have on the order of 2^n MUSes – and so applications of MUSes tend to not depend on finding *all* MUSes, but rather the rate of output or the number produced within some time limit is of greatest importance.

We base our work on the MARCO algorithm [7], a recently-developed sequential MUS enumeration algorithm. MARCO produces MUSes at a high initial rate, making it well suited to applications for which multiple MUSes are desired quickly, as compared to an approach like the CAMUS algorithm [8] that can complete the enumeration in less time when complete enumeration is tractable but more often fails to return any MUSes within a reasonable time limit.

MARCO can use any single-MUS extraction algorithm as a black-box solver, and most of its runtime is spent in running that solver; therefore, we investigate one approach to parallelizing MARCO that employs an existing parallel single-MUS extraction algorithm with no changes made to MARCO itself. This is convenient, but its scaling relies solely on that of the MUS extraction algorithm. Therefore, we have also explored an extension of MARCO that parallelizes the entire algorithm itself in a flexible and highly scalable fashion.

We should note that MARCO simultaneously enumerates *minimal correction subsets* (MCSes) as it produces MUSes, and there has been recent interest in exploring this problem further [9]. While the parallelization we present improves MCS enumeration as well, that is not the aim of this work, and we focus primarily on performance increases in enumerating MUSes alone.

Three recent articles have explored MUS enumeration beyond MARCO. Bacchus and Katsirelos [2] present a new algorithm for single-MUS extraction based on the duality between MCSes and MUSes, and they integrate this algorithm into MARCO to reduce the latency of computing *an* MUS. Then, they extended the new single-MUS extraction algorithm to find a "collection" of MUSes incrementally [1]. Compared to MARCO iteratively finding each MUS individually, their algorithm performs local search to find more MUSes once the first is computed. For the task of partial MUS enumeration, when enumerating MUSes within a set time limit, their algorithm produced more MUSes than MARCO more often (170 instances) than not (82 instances). Zielke and Kaufmann [15] directly extend MARCO to produce MCSes faster with the goal of using those to improve the enumeration of MUSes. Their algorithm produces far more MCSes than MARCO, but in terms of MUS output, the results are mixed. It produces more MUSes than MARCO in 55% of their test cases, but

the performance loss relative to MARCO in the other 45% is greater than the gain in those 55%. As we will see, these and other enumeration algorithms could be integrated into the parallelization we present here; however, as these approaches add complexity to MARCO and some have produced mixed results on the task of MUS enumeration, we do not incorporate them into the evaluation here, instead focusing on parallelizing the algorithm underlying all of them.

MARCO uses a SAT solver for maintaining internal state and directing its search (separate from the constraint system in which it is enumerating MUSes), and it would be possible to use a parallel SAT solver [10] as a drop-in replacement there as well. Usually, the time spent in this solver is a small fraction of MARCO's total runtime, however. Even in the unlikely case that the parallel SAT solver achieves perfect scaling, the impact on the entire algorithm's performance would be minimal, and so we do not explore this option in this work.

To the best of our knowledge, no previous work has presented a parallel algorithm for the MUS enumeration problem. The closest work of which we are aware is by Jannach, et al. [5], who parallelized an algorithm for model-based diagnosis (MBD). After mapping concepts from the diagnosis domain into our own ("diagnosis"→MCS; "conflict"→MUS), their work can be seen as an algorithm for enumerating all MCSes of a particular type of constraint system, finding and using some MUSes during execution. Thus there are some connections, but the focus is different and the work does not directly apply to the problem of interest here.

In the remainder of this paper, we first set out formal definitions in Section II, followed by a brief overview of the sequential MARCO algorithm in Section III. Addressing the "simple" parallelization of MARCO, existing work on parallel single-MUS extraction is covered in Section IV. We describe the full parallelization of MARCO in Section V, present experimental results in Section VI, and conclude in Section VII.

II. Preliminaries

We are interested in *constraint-agnostic* approaches for analyzing infeasible constraint systems that can be applied to any type of constraints. Hence, here we define a constraint system, C, without specifying the type of constraints it contains. Consider C as an ordered set of n abstract constraints, where $C = \{C_1, C_2, ..., C_n\}$ over a set of variables. Each C_i places some restrictions on values that can be assigned to certain variables. C is *satisfied* by an assignment to the variables of C if all restrictions of every C_i are met by the assignment; we call such an assignment a *model*. We say C is *satisfiable* or *SAT* if such an assignment exists; otherwise, it is *unsatisfiable*, *infeasible*, or *UNSAT*. When a constraint system is determined to be *UNSAT*, it is often useful to perform further analyses on its infeasibility.

One common way to analyze an unsatisfiable constraint system is to extract the following types of subsets with particular properties:

- A *minimal unsatisfiable subset* (MUS) of a constraint system C is a subset $M \subseteq C$ such that M is unsatisfiable and $\forall c \in M, M \setminus \{c\}$ is satisfiable.
- A *maximal satisfiable subset* (MSS) of a constraint system C is a subset $M \subseteq C$ such that M is satisfiable and $\forall c \in C \setminus M, M \cup \{c\}$ is unsatisfiable.
- A *minimal correction set* (MCS) of a constraint system C is a subset $M \subseteq C$ such that $C \setminus M$ is satisfiable and $\forall S \subset M, C \setminus S$ is unsatisfiable.

As a concrete example, we can consider the *Boolean satisfiability problem*, as it is also relevant to the internals of the algorithms we will be discussing. Such a constraint system C is defined over a set of Boolean variables and represented as a Boolean logic formula in conjunctive normal form (CNF), where C is a conjunction of clauses, $C = \bigwedge_{i=1..n} C_i$; each clause C_i is a disjunction of literals, $C_i = l_{i1} \vee l_{i2} \vee \cdots \vee l_{ik_i}$; and each literal is either a Boolean variable x or its negation $\neg x$.

III. Sequential MUS Enumeration (MARCO)

This work builds on the sequential MARCO algorithm for MUS enumeration [6], [7], [11]. MARCO analyzes an infeasible constraint system C by exploring its power set $\mathcal{P}(C)$. MARCO exploits the connection between the power set of a constraint system and Boolean algebra; it uses a Boolean formula to keep track of the subsets whose satisfiability has been determined to reduce the future search space.

Specifically, the MARCO algorithm builds a function, $\mathcal{F} : \mathcal{P}(C) \rightarrow \{0, 1\}$, that maps each element of the power set of C to False (already explored) or True (unexplored). The function $\mathcal{F}$ thus takes a subset, $seed \subseteq C$, as an input, and it returns 1 if and only if the satisfiability of $seed$ is unknown and remains to be checked, 0 otherwise. Therefore, any model of $\mathcal{F}$ indicates an unexplored subset of C. Once an MUS/MSS is found, an appropriate blocking clause is added to $\mathcal{F}$. Thus, this formula keeps track of which subsets have been explored and which have not.

The MARCO algorithm iteratively solves $\mathcal{F}$ to identify unexplored subsets. Each such subset found is checked for satisfiability using a constraint solver. If the subset is UNSAT, MARCO calls a **shrink** subroutine to extract an MUS; otherwise, if MARCO is run in its optimal configuration, the SAT subset is guaranteed to be an MSS. After an MUS/MSS is computed, its corresponding blocking clause is added to $\mathcal{F}$. MARCO maintains two solvers to implement these procedures: one solver, Map, holds $\mathcal{F}$ and generates models of it, and the other solver determines the satisfiability of subsets. MARCO terminates when $\mathcal{F}$ is no longer satisfiable; at that point, the satisfiability of every subset of C is known and all MUSes/MSSes have been found.

IV. Parallel Single-MUS Extraction (Related Work)

While we are aware of no other work on enumerating MUSes in parallel, parallel algorithms for single-MUS extraction have been presented, and one simple approach to

parallelizing partial MUS enumeration would thus be to use a parallel MUS extraction algorithm as the **shrink** subroutine in MARCO.

Wieringa [13] first developed a parallel MUS extraction algorithm, which we denote by TarmoMUS in this paper. In a later paper, Wieringa and Heljanko [14] presented a parallel incremental SAT solver, Tarmo, and used the MUS extraction problem as a case study to show the effectiveness of their solver. Belov and Manthey [4] further explored the parallelization of MUS extraction with different levels of flexibility. They investigated the implementation and communication between threads in detail, and they presented a parallel version of the MUSer2 MUS extraction algorithm named pMUSer2.

Both TarmoMUS and pMUSer2 are built on the deletion-based approach to extracting an MUS. This approach iteratively removes each clause from an unsatisfiable Boolean formula and tests its satisfiability after removing the clause. If the formula becomes satisfiable, the removed clause is necessary for the MUS; thus we return it to the formula and move to the next constraint. If the formula remains unsatisfiable, the removed constraint is not needed for the MUS, and it is discarded. Wieringa developed TarmoMUS, built on top of an external SAT solver, Tarmo. Tarmo is made to solve in parallel multiple copies of a given formula with different clauses enabled and disabled. Due to the similarity between the formulas it is solving, Tarmo can share information between the parallel solvers, resulting in an efficiency gain. TarmoMUS has two master threads: one creates the formulas with different clauses removed and passes them to Tarmo, Tarmo tests the necessity of those clauses potentially in parallel, and the other thread receives results from Tarmo and performs model rotation to identify further necessary clauses.

Belov and Manthey explored a variety of choices one can make when parallelizing deletion-based MUS extraction. The first choice is whether the algorithm executes asynchronously or synchronously. In the synchronous execution, the master thread waits for all worker threads to finish the current level of the iteration before assigning any of them new task, whereas in the synchronous execution, the master processes results as they are sent from any worker thread. Another choice comes from the work distribution – whether all workers check the necessity of the same clause (benefiting from the different paths each solver may take through the search space) or each gets a different clause to test. The last choice regards communication, whether workers share learned clauses between themselves or not. Additional learned clauses received from other solvers can improve a solver's performance, but this has the downside of added delays for communication. Across all configurations, they showed that the best performance is obtained by workers checking different clauses in a formula, sharing learned clauses, and using asynchronous execution. Their results show pMUSer2 outperforms the previous approaches, and thus we consider it the state of the art in parallel MUS extraction.

V. MARCOs: Parallel MUS Enumeration

Our parallelization of MARCO is dubbed MARCOs (an acronym for **Ma**pping **R**egions of **Co**nstraints **S**imultaneously, as well as the "plural" of MARCO – it is pronounced as the name "Marcos"). MARCOs features limited communication between threads, resulting in robust scaling over many cores without substantial overhead. Fundamentally, it exploits the fact that each call to **shrink**, where the bulk of MARCO's time is spent, can be executed independently in parallel. MARCOs identifies multiple unexplored seeds simultaneously, and each UNSAT seed can be passed to **shrink** to extract an MUS in parallel with others. The sole *required* communication is the reporting of results (MUSes and MSSes generated). MARCOs employs a master-worker architecture, with a master thread that initializes worker threads and manages the communication:

1) The master thread creates k worker threads (one per core) running separate copies of the sequential MARCO algorithm.
2) Worker threads send generated results (MUSes and MSSes) to the master thread as they are found.
3) The master thread filters duplicate results and outputs any new results it has not yet received.
4) As soon as any one thread reports that the enumeration is complete (no unexplored subsets remain), the master thread terminates the algorithm.

The communication required can be accomplished by passing messages via unidirectional IPC mechanisms. It requires no shared-memory data structures, so no locks or additional synchronization mechanisms are necessary. In our implementation, we create the threads with Python's `multiprocessing` library and IPC is performed with `multiprocessing.Pipe` objects.

The correctness and completeness of MARCOs depend directly on its worker threads. Completeness and correctness proofs for MARCO are found in [7]. Additional implementation details as well as variants and optimizations follow.

A. Checking for Duplicates

To check whether each received result has been previously computed by other workers, the master thread must maintain a set of all results with fast lookups. In our tests, the memory required to store these results directly in a standard set data structure was prohibitive. Instead, we found that reusing the "map" concept from MARCO was far more memory efficient while retaining adequate speed. Specifically, the master thread maintains its own Map solver for a formula $\mathcal{F}$. For each result received from a worker, the master thread evaluates whether the assignment corresponding to that MUS/MSS is a model of $\mathcal{F}$. If it is, the MUS/MSS is new. The master thread then outputs it as a new result and adds a blocking clause as appropriate to its copy of $\mathcal{F}$ to mark it as explored. If the received result does not correspond to a model of $\mathcal{F}$, the master thread simply discards it and waits for the next. Note that the master thread can only filter out duplicate results after

they have been received, and thus workers could still waste time in computing them.

B. Avoiding Duplicate Work: Randomization

An additional way to prevent MARCOs from generating duplicate results is to bias worker threads to work on different tasks in the first place. To accomplish this, when first initializing the Map solver and the constraint solver in each worker, we randomize the initial activity of the variables in each solver, thus randomizing the solvers' variable orderings.

Though the first maximal seed returned by the Map solver for each worker will be the same – the entire constraint system – the randomized constraint solvers are likely to extract different UNSAT cores from it; therefore, all worker threads could have different seeds passed to the first call to their **shrink** subroutines. And in the Map solvers, randomization can lead each worker thread to generate different seeds even when they have equivalent $\mathcal{F}$ formulas. We show in Section VI that this randomization is crucial for good performance and scaling.

C. Avoiding Duplicate Work: Sharing Results

A further method for avoiding duplicated work between threads is to share information between the worker threads in addition to the one-way communication to the master. When the master receives each new result, it can send the MUS/MSS to all workers other than the one that produced that result. Workers can then block each received result in their *own* Map solvers, marking the corresponding subsets as explored.

We initially placed results received by each worker in a queue that the worker would check before producing each new seed, which is when the shared results are most useful. However, a worker may spend a very long time in a single call to the **shrink** subroutine, and a large number of results can be received by the worker in that time, potentially exceeding memory limits. To avoid this, we created a second thread for each worker that handles incoming results concurrently with its normal execution of the MARCO algorithm. This requires synchronizing the Map solver with thread-safe mutexes, as the solver object is now shared between two threads.

Note that this method for avoiding duplicate work introduces a trade-off: any saved work comes at the cost of increased overhead due to the additional communication and synchronization. We show in Section VI that this trade-off generally resolves in favor of *not* adding this sharing, due primarily to the fact that duplicate results are rare.

In addition to the above strategy for using shared results, a worker thread could identify when a newly-received result might make its current work redundant. For example, if a worker is running **shrink** on a seed that is a superset of a just-received MUS, it may produce the same MUS again. Even in this situation, the call to **shrink** could produce a different MUS, however, as each unsatisfiable seed is likely to contain multiple MUSes. Given this uncertainty along with the rarity of duplicate results overall and the fact that receiving a result subsuming the current work at any point is even more rare, we chose not to explore this option.

VI. Experimental Results

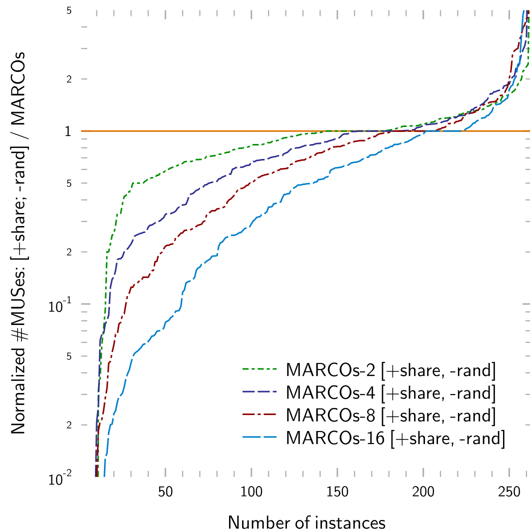
We investigated the performance and scaling of the proposed algorithms across a range of instances on 2 to 16 cores. We first evaluated the variants of MARCOs proposed in Section V, specifically investigating the effects of randomization and result-sharing among worker threads and determining the best variant based on experimental results. We then compared the best variant of MARCOs to the "simple" parallelization of MARCO obtained by using the parallel MUS extraction algorithm pMUSer2 [4] as the **shrink** subroutine (Section IV) – referred to as MARCO(pMUSer) in this section. In all cases, we focus on the number of MUSes that an algorithm produces within a set 10 minute time limit, as the motivation for parallelization is primarily to increase this rate. As the output rate of MARCO has been shown to remain consistent during its execution [7], 10 minutes is sufficient to characterize the performance of the tested algorithms on each instance.

We collected a large set of benchmarks from sources including the MUS track of the 2011 SAT competition[1], automotive product configuration [12], and circuit diagnosis applications. To limit over-representation of any particular type of instance, we grouped benchmarks into "families" by filename and randomly selected five instances from each family; when only five or fewer were available in a family, all were selected. This selection produced a set of 309 unsatisfiable Boolean CNF benchmarks ranging in size from 26 to 4.4 million variables and from 70 to 16.0 million constraints (clauses).

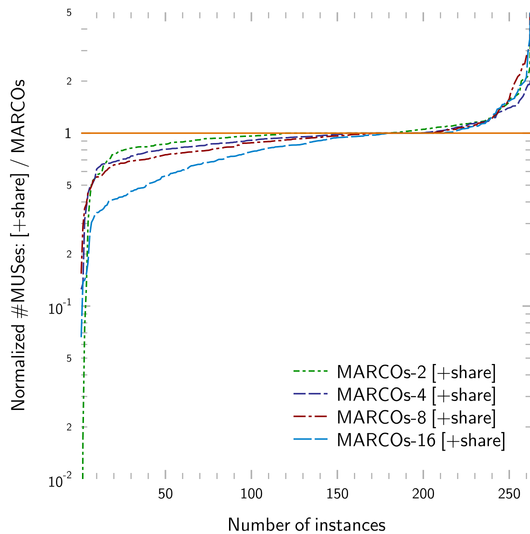
From these 309, we excluded from our analysis the 25 instances for which none of the tested implementations were able to produce *any* MUS. Additionally, because we are focused on the partial MUS enumeration problem, most applicable when complete enumeration is infeasible, we also excluded 21 instances for which sequential MARCO was able to complete the enumeration within the given time limit. For the problem of complete MUS enumeration, it is shown in [7] that CAMUS generally outperforms MARCO; thus, in applications where complete enumeration is tractable, MARCO would not be the algorithm to use. And as parallelization is only able to offer constant-factor speedups, it will not enable complete enumeration in many more instances than a sequential algorithm; in our experiments the parallel algorithms completed enumeration in only one more instance than sequential MARCO. The result of this filtering was a set of 263 unsatisfiable instances for which sequential MARCO fails to complete the enumeration and at least one algorithm produced at least one MUS.

We ran all experiments on Amazon Elastic Compute Cloud (EC2) "c3.8xlarge" instances with Intel Xeon E5-2680 v2 processors and 60GiB of RAM. Each machine had 16 processor cores and 32 logical cores due to hyperthreading; each execution of an algorithm was specifically allocated and restricted to k cores (and their corresponding $2 \cdot k$ logical cores) for some value of k between 1 and 16 as noted by the -k suffix appended to algorithm names below. MARCOs was run with k worker threads in addition to the master,

[1] Available: http://www.cril.univ-artois.fr/SAT11/



(a) With result-sharing, no randomization



(b) No result-sharing, with randomization

Fig. 1. Cumulative distribution plots of normalized MUS output counts for MARCOs variants. Both cases are normalized to MARCOs with no result-sharing and with randomization on each of 2, 4, 8, and 16 cores.

and MARCO(pMUSer) ran pMUSer2 with k threads. Every execution of an algorithm on a single instance was given a timeout of 600 seconds and a memory limit of $3500 \cdot k$ MB of RAM (hence from 3.5GB for the sequential algorithm up to 56GB for a 16-core parallel execution).

A. *MARCOs configurations*

We first analyzed how the variants of MARCOs proposed in Section V affected performance. Specifically, we evaluated the effects of randomization and of sharing results between worker threads. We found that the best configuration used randomization, as expected, but sharing results between worker threads was unexpectedly detrimental to performance in most cases.

In Fig. 1, we show MUS output counts for variants of MARCOs run on 2, 4, 8, and 16 cores *normalized to the number output by the best configuration on the same number of cores*. Each line shows ratios of the output counts of a variant to the best configuration, both run on k cores. The data are sorted to produce cumulative distribution plots (similar to a “cactus plot”) that illustrate the distribution of those values. Each point below $y = 1.0$ represents an instance for which the variant produces MUSes more slowly than the best configuration, while those above represent instances where the variant is faster.

In Fig. 1a, we examine the effect of removing randomization. Without randomization ([-rand]), sharing results becomes critical – otherwise each worker thread will make the same choices and produce the same results, producing no benefit from parallelization – therefore, the specific variant examined also enables result-sharing ([+share]). It is clear that except for a few instances where the variant without randomization is able to generate result more quickly, randomization provides benefits in most cases, with the non-randomized variant being slower in roughly 70% of all instances, often by an order of magnitude. The effect becomes more pronounced as the number of threads is increased.

Fig. 1b presents data examining the effect of result-sharing (and with randomization enabled). Intuitively, enabling result-sharing between worker threads will help avoid duplicate, redundant work and thus boost the performance: without sharing results, all workers will have to compute the complete set of results independently, whereas with sharing each result only needs to be found by one worker. In our experiment, the effect on performance is in fact the opposite. In this case again, roughly 70% of instances exhibit decreased performance with the variant, though the effect is not as drastic as with disabling randomization. This implies that the cost of sharing results (the increased communication overhead) generally outweighs any benefit it provides. Moreover, even when the number of worker threads is increased to 16, the relative performance of the result-sharing variant decreases, despite the fact that duplicate results will be more frequent with more simultaneous workers.

To help explain this, Fig. 2 shows a reverse cactus plot of the MUS duplicate rates (#duplicate MUSes / #total MUS results) for MARCOs without result-sharing on 2, 4, 8, and 16 cores. The number of duplicate MUSes puts an upper bound on the amount of benefit result-sharing can provide; if there are few duplicate MUSes, there is little redundant work for result-sharing to possibly prevent. In most instances, there are very few duplicate MUSes. For each of 2, 4, 8, and 16 cores, the number of instances with fewer than 10% duplicate MUSes is 93%, 86%, 85%, and 79% of the instances, respectively. In the worst case, when run on 16 cores, the median duplicate rate is 0.39%, and in 38% of instances no duplicate MUSes are produced at all.

Our explanation for this is that the search space is so vast in most cases (recall the number of MUSes can be exponential in the size of the instance) that randomization alone is sufficient to prevent most duplicate work *early in the enumeration*. Indeed, results on the instances for which enumeration can be

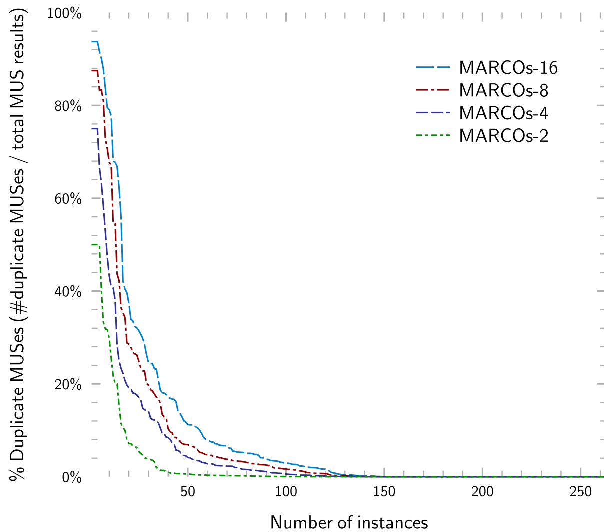


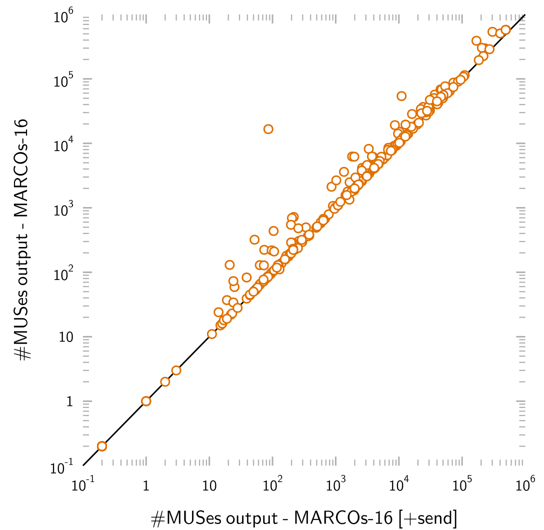
Fig. 2. Reverse cactus plot of MUS duplicate rates for MARCOs with randomization and no result-sharing on 2–16 cores.

completed suggest that result-sharing has a beneficial impact on total runtime in those cases; when the enumeration nears completion and the unexplored seeds become fewer, duplicate work will become more common and sharing results *will* increase efficiency. For most applications of *partial* MUS enumeration, however, this will not happen in any reasonable time frame due to the sheer size of the result set, and so sharing results cannot produce enough benefit to outweigh its cost.

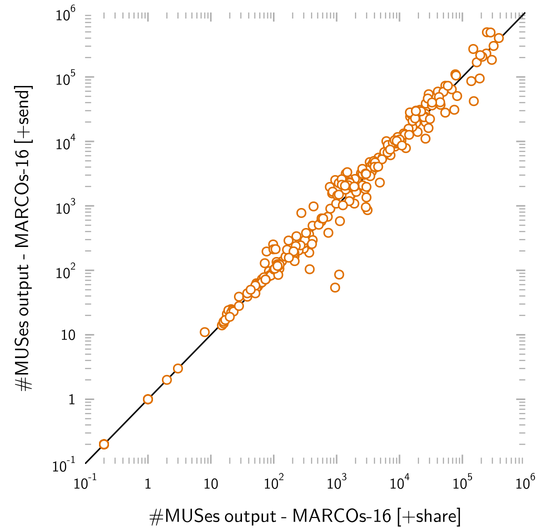
B. Costs and Benefits of Result-Sharing

We investigated result-sharing further by looking separately at the effects of 1) sending results to worker threads and 2) incorporating those results into each worker's Map solver. We ran an experiment with an "intermediate" implementation between the result-sharing and no result-sharing variants: in this implementation, the master thread sends all results to all worker threads, but the worker threads do not actually add those results to the Map solver. We call this variant "result-sending" or [+send]. With this, we can measure the cost of the communication alone, separately from the cost and benefit of integrating them into each worker's Map.

The following table summarizes the performance differences between the no-result-sharing, result-sharing, and intermediate ("result-sending") variants in terms of MUSes produced. In the table, we present the geometric mean of the ratios between a variant's MUS output count and that of MARCOs without result-sharing for each of 2, 4, 8, and 16 cores. Fig. 3 presents scatter plots of the MUS output counts for these variants on 16 cores, where the effects of communication and sharing are largest.



(a) MARCOs [+send] vs MARCOs with no result-sharing.



(b) MARCOs [+share] vs MARCOs [+send].

Fig. 3. Scatter plots comparing #MUSes produced.

	average ratio to MARCOs-k	
k	[+send]	[+share]
2	0.979	0.962
4	0.964	0.919
8	0.951	0.907
16	0.818	0.786

From the table, we can see that just sending the results has a fairly small effect (a 2-5% reduction in output) for 2 through 8 cores, while at 16 cores the communication results in a roughly 18% reduction in output. Fig. 3a, showing detail for the 16-core case, shows that even at 16 cores, the majority of instances show little effect from the communication, with most points lying very near the diagonal. In our experiments, the largest performance reductions occur when the size of each result is extremely large (when the MUSes/MSSes have

hundreds of thousands of clauses or more). Note that there is no clear correlation between number of results sent and the communication cost; many instances with hundreds of thousands of results sent show little difference in total output whether they are sent or not.

Comparing the overall performance of sending the results ([+send]) and *using* those sent results ([+share]) isolates the effect of using the results. From the table again, we can see that using the shared results produces an additional *decrease* in performance. Fig. 3b conveys this as well, with mixed results between the two variants. This suggests that adding incoming results to the Map solver takes enough time that it often outweighs the benefit of possibly preventing those results from being computed again in the future. Even if sending the results were "free," result-sharing would slightly reduce performance overall. This is again explained by the duplicate rate data; the cost of sharing results is always present here, even though it can only provide a large benefit in a small set of instances.

C. Scaling across 2 to 16 cores

We examined the scaling of the best variant of MARCOs (with randomization and no result-sharing) and of MARCO(pMUSer). Fig. 4 presents cumulative distribution plots of normalized MUS output counts (as in Fig. 1) to provide a clearer view of each algorithm's scaling. The data here are ratios of an algorithm's output count on k cores to the output count of sequential MARCO on a single core. Perfect scaling would produce k times the outputs of the sequential algorithm, and this is shown in the charts by the labeled horizontal lines. There are a few instances where a parallel algorithm generated some results and sequential MARCO found nothing. Those data points lie at infinity, as the denominator of the ratio is zero in those cases.

Here we can see that as the number of cores increases, the MUS output counts grow proportionally, but diminishing returns are apparent. Additionally, we can see that MARCOs is much closer to perfect scaling than MARCO(pMUSer) is: MARCOs is more often near k times sequential's performance, more often *above* k times, and almost never below sequential's output count.

We summarize the scaling results in the following table.

	MARCOs-k		MARCO(pMUSer-k)	
k	average ratio to sequential	as % of perfect scaling	average ratio to sequential	as % of perfect scaling
2	1.804	90.2%	1.308	65.4%
4	3.660	91.5%	1.647	41.2%
8	6.589	82.4%	1.702	21.3%
16	11.163	69.8%	1.706	10.7%

In the table, we present the geometric mean of the ratios shown in Fig. 4 for each algorithm and for each of 2, 4, 8, and 16 cores[2]. From the average ratios, we compute the percentage of perfect scaling that each represents. The data show that MARCOs reaches over 90% of perfect scaling when run on 2 or 4 cores, and it is still at nearly 70% on 16 cores. The average performance of MARCO(pMUSer) does not improve much after 4 cores, and its scaling is worse than that of MARCOs at all core counts. The empirical analysis of pMUSer2 presented in [4] shows results consistent with these, both in the rough magnitude of the performance improvements up to 4 cores and the minor gain going from 4 to 8.

Overall, MARCOs scales well, making efficient use of multiple cores for enumerating MUSes. Parallelizing MARCO by plugging in a parallel MUS extraction algorithm, on the other hand, exhibits much worse scaling due to the difficulty of parallelizing the task of extracting one MUS.

VII. Conclusions & Future Work

We have explored parallel approaches to the partial MUS enumeration problem, presenting two different extensions of MARCO, a recently-developed sequential MUS enumeration algorithm. The first approach is a simple drop-in replacement of the sequential MUS extraction tool used in MARCO with a parallel version, pMUSer2 [4]; this provides a simple, immediate parallelization of the most time-consuming step in MARCO. The second is a complete parallelization of MARCO itself, which we have dubbed MARCOs, alongside which we have discussed several implementation choices that can have a large impact on its performance.

Experiments show that employing pMUSer2 in otherwise-unmodified MARCO fails to scale well, suffering from the fact that parallelizing single-MUS extraction is difficult, and thus far it has not come close to perfect scaling. On the other hand, MARCOs avoids this problem by extracting multiple MUSes simultaneously, and results show that it scales well, consistently reaching a large fraction of the performance of perfect scaling. Experiments designed to determine the impacts of various implementation choices show that randomization is crucial to achieving good scaling and that the randomization then makes sharing results between threads to avoid redundant work unnecessary in most cases.

As this is the first work on parallelizing partial MUS enumeration, many avenues are open for further research. MARCOs reaches a large percentage of the performance of perfect scaling, though there is some room to improve its scaling. Any improvements in sequential MUS enumeration can be easily adapted into its parallel framework with equivalent scaling. Furthermore, the flexible nature of the framework enables the exploration of "mixed strategies," in which each thread runs a different heuristic or a different solver altogether, which may enable synergies between threads that produce results in more instances and could improve performance beyond k times that of MARCO as it exists (this would

[2] Invalid ratios, when one or both algorithms output no MUSes, are necessarily excluded. Any inclusion of those data could only increase the reported average performance ratios, as more instances had zero outputs for sequential and non-zero for parallel than the other way around.

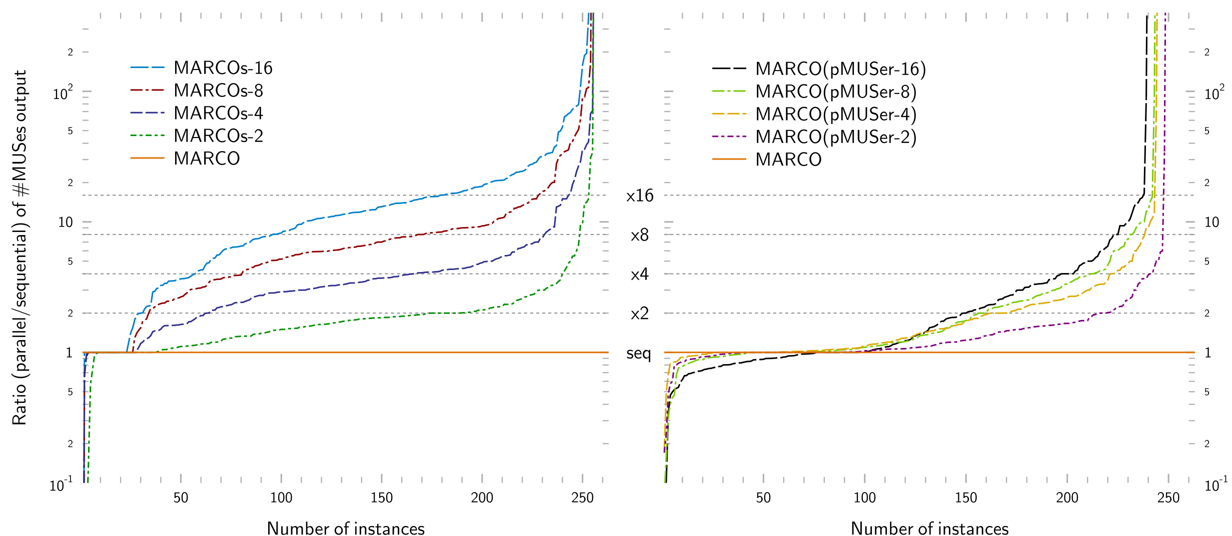


Fig. 4. Cumulative distribution plots of MUS output counts for each parallel algorithm normalized to sequential MARCO. **Left:** MARCOs. **Right:** MARCO(pMUSer). Horizontal lines indicate where perfect scaling for each of 2, 4, 8, and 16 cores would lie.

not produce "more than perfect" scaling, however, as these mixed strategies could always be scheduled sequentially as in a portfolio solver).

This work focused on MUS enumeration, but both MARCO and MARCOs enumerate MSSes/MCSes while they are producing MUSes, so MARCOs is thus a parallel MCS enumeration algorithm as well, suggesting that comparisons should be made to existing algorithms in that area. And finally, MARCO is just one target for parallelization of MUS enumeration, and other algorithms such as CAMUS [8], which has better performance for *complete* MUS enumeration, could be parallelized as well.

Acknowledgments

We would like to thank the SAT2016 and CP2016 reviewers for their helpful comments. This work is partially supported by the AWS Cloud Credits for Research program.

References

[1] F. Bacchus and G. Katsirelos. Finding a collection of muses incrementally. In C.-G. Quimper, editor, *Proceedings of the 13th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2016)*, pages 35–44. Springer International Publishing, 2015.

[2] F. Bacchus and G. Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In D. Kroening and S. C. Păsăreanu, editors, *Computer Aided Verification: 27th International Conference, CAV 2015*, pages 70–86, Cham, 2015. Springer International Publishing.

[3] A. Belov, I. Lynce, and J. Marques-Silva. Towards efficient MUS extraction. *AI Communications*, 25(2):97–116, 2012.

[4] A. Belov, N. Manthey, and J. Marques-Silva. Parallel MUS extraction. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT-2013)*, pages 133–149. Springer, 2013.

[5] D. Jannach, T. Schmitz, and K. M. Shchekotykhin. Parallelized hitting set computation for model-based diagnosis. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI-2015)*, pages 1503–1510, 2015.

[6] M. H. Liffiton and A. Malik. Enumerating infeasibility: Finding multiple MUSes quickly. In *Proceedings of the 10th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2013)*, pages 160–175, May 2013.

[7] M. H. Liffiton, A. Previti, A. Malik, and J. Marques-Silva. Fast, flexible MUS enumeration. *Constraints*, 21(2):223–250, 2015.

[8] M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, Jan. 2008.

[9] J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov. On computing minimal correction subsets. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI-2013)*, pages 615–622. AAAI Press, 2013.

[10] R. Martins, V. Manquinho, and I. Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, 2012.

[11] A. Previti and J. Marques-Silva. Partial MUS enumeration. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI-2013)*, pages 818–825, July 2013.

[12] C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(1):75–97, 2003.

[13] S. Wieringa. Understanding, improving and parallelizing MUS finding using model rotation. In *Principles and Practice of Constraint Programming (CP 2012)*, pages 672–687. Springer, 2012.

[14] S. Wieringa and K. Heljanko. Asynchronous multi-core incremental sat solving. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*, pages 139–153. Springer, 2013.

[15] C. Zielke and M. Kaufmann. A new approach to partial MUS enumeration. In M. Heule and S. Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015: 18th International Conference*, pages 387–404. Springer International Publishing, 2015.