PARALLEL SAT SOLVING

Jan Eberhardt, Jakub Lichman

Department of Computer Science
ETH Zurich
Zurich, Switzerland

ABSTRACT

The boolean satisfiability problem, or SAT for short, is an interesting problem with various applications. In this report we describe how we implemented and parallelized a simple SAT solver. We compare different parallelization variations and analyze their respective trade offs. We achieved linear or even super linear speedups until up to 28 CPU cores compared to our sequential implementation.

1. INTRODUCTION

Motivation. The boolean satisfiability problem (SAT) belongs to the most important problems in program analysis, verification and other disciplines of computer science. It is used in the background of many applications, especially ones in the field of automated planning and scheduling, model checking (formal verification) and theorem proving. The last decade brought many improvements to the SAT solving world in the form of advanced heuristics, preprocessing and processing techniques, and data structures that allow efficient implementations of search space pruning. The past 10 years were also rich in improvements in parallelism. Current trends in computer hardware design decreased performance per processing unit and pack more units on a single processor. However, algorithms such as DPLL and CDCL were invented before the wide use of parallelism and therefore designed for sequential execution. Since the SAT problem is NP-complete, we consider it a good candidate to be parallelized.

In our approach, we are trying to speed up SAT solving by running it on multiple cores with different techniques of search space partitioning.

The remainder of this report is structured as follows: The next subsection provides related work. In Section 2 we introduce core SAT concepts. Section 3 explains how we parallelized DPLL and Section 4 contains the experimental evaluation. In Sections 5, 6 and 7 we talk about things that we tried that did not work out, about possible extensions to our work and conclude.

Related work. Tomas Balyo et al. proposed the "HordeSat" solver, which can run on up to 1024 cores and is based on the CDCL algorithm. Their parallel approach is different from ours, because it is portfolio based but with signs of search space partitioning. [1]

Most of the previous SAT solvers designed for computer clusters or grids use explicit search space partitioning. Examples of such solvers are GridSAT [2], PMSAT [3] or ManySat [4].

Jurkowiak et al. proposed a work stealing implementation for dynamic load-balancing. [5] Our work stealing approach is similar to what they proposed.

Berger et al. parallelized DPLL in Haskell. [6] Our DPLL search space partitioning is similar to what they proposed, but our solver is not limited to one machine and we can run on multiple compute nodes.

2. BACKGROUND

CNF. A *boolean variable* is a variable that can hold either *true* or *false* as a value. A *literal* of a boolean variable x is either the positive variable x or its negation $\overline{x}$. A *clause* is a disjunction (OR) of literals. A formula in *conjunctive normal form* (CNF) is a conjunction of such clauses. Every propositional logic formula can be transformed into an equivalent formula in conjunctive normal form. For the sake of this report we assume that the formula is already in the CNF format. However a transformation would be simple and straight forward based on basic propositional logic equivalence rules.

SAT Solver. A SAT solver is a program that is able to decide whether a given formula is satisfiable. More formally, a given formula F is *satisfiable iff* there *exists* a variable to value assignment θ that makes the formula true. The assignment θ is then called a model of the formula F. If there does not exits such an assignment, the formula is *unsatisfiable*. Furthermore, a SAT solver should be able to provide a valid variable assignment θ in the *satisfiable* case.

History. The first SAT solving algorithm was develop in 1960 by Martin Davis and Hilary Putnam [7]. They used a resolution-based decision procedure. In 1962 Davis, Putnam, Logemann and Loveland developed an extension of the DP algorithm, the DPLL algorithm. DPLL is a back-

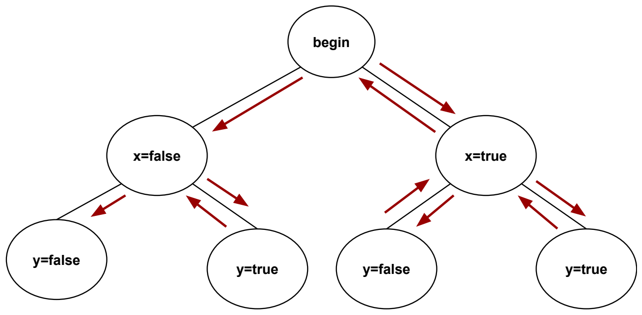

Fig. 1. How DPLL explores the search space of a formula with two variables *x* and *y*.

tracking-based search algorithm that still forms the basis for many extensions and implementations of SAT solvers.

DPLL. DPLL solves a SAT problem by modeling it as a decision tree, variables can either be assigned to *true* or *false*. In every step DPLL tries to find variables that are "automatically" assigned because of a previous decision, if there are none it will pick a variable v_i and assume a value for it. If the path that is taken based on this decision turns out to be unsatisfiable, it backtracks to the decision and picks the negated assignment for variable v_i. The backtracking search continues until a solution is found or the whole decision tree is traversed. Figure 1 shows how DPLL's decision tree could look like for a simple formula with 2 variables.

CDCL. Similar to DPLL the *Conflict-Driven Clause Learning* (CDCL) algorithm performs a depth-first search on the space of partial variable assignments. In addition to DPLL, CDCL adopts a pruning technique called learning. In DPLL it can happen that the same combination of variable assignments results in multiple conflicts and therefore multiple unsatisfiable paths. CDCL remembers the variable assignments that cause a conflict and avoids them in future decisions, so to say pruning the search space. This learning is done by analyzing the state whenever a conflict is reached and adding clauses based on that state to the formula. If a conflict cannot be resolved by backtracking, then the formula is unsatisfiable. If all variables are assigned and no conflict occurred, then the formula is satisfiable.

3. PARALLELIZING DPLL

In this section we describe how we parallelizled DPLL.

As a result of the simplicity of DPLL, it can be parallelized relatively easy. The parallelization is not trivial, but far easier than in the case of CDCL, where in addition to load balancing one would need to deal with learned clause sharing.

DPLL Branches. As shown in Figure 1 the DPLL algorithm explores the search space with depth first search. Whenever there are no more variables that can be assigned trivially, one variable assignment is picked. That is exactly the point where we can do so called DPLL branching: one core continues on one branch and lets some other core solve the other branch. We can call this point a branching point and it represents one node (or decision) in the DPLL decision tree. The decision and the whole subtree starting at that decision can be represented as a partial variable to value assignment, called a *partial model*.

The biggest challenge when using this form of branching is load balancing. The work cannot be split equally between workers at the beginning, because it would require knowledge of the depth of each subtree. This information cannot be inferred from a formula without solving it. We want to avoid situations where work is not divided equally because we want to fully exploit the computing power of all cores. In our work we considered two approaches to load balancing. The first one is a master worker communication model, where one actor, called *master*, is in charge of storing partial models and passing them to *workers* that will solve them. The second approach is a work stealing communication model, where each actor (or *worker*) runs on its own and all workers are equal. If some worker runs out of work and the formula is not solved yet, it picks a random other worker and tries to steal a partial model from it, such that it can continue doing some work.

Master Worker Model. In the master worker model, the *master* is one of the *n* available cores and the other *n-1* cores are *workers*. At the beginning all workers are marked as free. The master starts solving by passing an empty partial model to the first free worker. When a worker reaches a branching point it takes one branch and sends the other branch as a partial model to the master. When the master receives a partial model, it checks whether there exists some free worker. If not it stores the partial model into its local queue. Otherwise it sends it to a worker, which finished solving a branch with *unsat* and therefore is free and needs more work. This procedure is repeated until some worker finds a *sat* model and sends it to the master. The master then outputs the model and stops all workers. If no worker found a *sat* model, the master has an empty stack and all workers are free, then we know that the formula is *unsat*.

While the master worker model is easy to understand and implement, it comes with several drawbacks. The first of them is the lack of one core. In the master worker model, one core (the master) is only used as a manager and therefore this core is not actually doing any computation. The next issue is related to scaling. The master can become a bottleneck with a certain number of cores, because there is too much communication it needs to process. As a result workers need to wait longer for getting new partial models. The last problem is also related to the amount of communication in this model. Partial models and meta data are transferred every time a worker branches or requests a new partial model.

Work Stealing Model. Because of the drawbacks mentioned above we have decided to reduce and decentralize the communication by implementing a work stealing model. The model treats all cores equally and therefore there is no master, just workers. All workers contain their own stacks with partial models to be processed. This reduces the communication overhead because all produced models are stored locally instead of sending them to and storing them at the master. However, the existence of this local stack breaks load balancing because some workers can work on bigger subtrees of the search space than others.

Inequalities in load balancing are resolved by a mechanism called *work stealing*. If some worker finished its work and no other worker has solved the formula yet, than it tries to "steal" a partial model from one of the other workers. The process of stealing can be defined as follows: If a worker has an empty local stack, it tries to steal from a randomly selected other worker until it finds a model or until someone else has found one. The stealing is always performed from the bottom of some other worker's stack, while the local solving of partial models is done from the top of the local stack. There are two more phases in the communication model. The first phase is the initial work distribution and the third one is the stopping criteria. In the first phase worker 0 will take the role of the master and distribute starting models to all other workers. Then all workers will try to solve their subtrees and in the case of running out of work they will try to steal work from some other worker (phase 2). When some worker finds a *sat* model, it will send it to worker 0, who stops all workers and outputs the final model (phase 3). A master is necessary at the end because there can be multiple workers that find a *sat* model at the same time. In our implementation, worker 0 prints the first obtained *sat* model and ignores the others.

Our current work stealing implementation has one drawback: It can only handle satisfiable formulas. The stopping criteria in the unsatisfiable case was very trivial in the master worker model because there exists a moment when the stack at the master was empty and all workers were free. In the work stealing model, no such moment exits. We do not have a central actor that controls the others and so knows in which stage each worker is and therefore we restrict ourselves just to satisfiable formulas with this communication model.

Implementation. We implemented both communication models in C++ with the help of MPI [8]. Our DPLL solver is implemented in C++ as well. We tried to keep the implementation of the DPLL algorithm and communication within nodes as separate as possible for better readability. We used *vector*s from the C++ standard library to represent sets of clauses and sets of literals. We used *unsigned int*s as variable names, and *boolean* types for sign and value of the literals and variables. To implement the backtracking algorithm we used recursion, but we allocate all necessary data structures on the heap. We did not experience any stack overflow issues.

Correctness. We tested our sequential DPLL solver on hundreds of formulas that we either randomly generated or took over from a SATLIB formula collection. [9] We ran each formula through z3 [10] and compared the result of z3 with our result. The following cases have to be considered for each test case: If both solvers return *unsat*, we pass the test case. If one solver returns *sat* and the other one *unsat*, we fail the test case. If both solvers return *sat*, we still need to check if the model our solver has returned is correct. To do that we can conjoin the model to the original formula and again run it through z3.

We ran both parallel implementations through the same set of formulas and checked correctness for each of them. Assuming that our sequential implementation is correct, it is straight forward that our parallel implementation is correct as well: we essentially solve the same set of subproblems but on different cores and in a potentially different order. The difference between the master worker and work stealing model is just the load balancing and not the search space. Both parallel versions solve the same partial models but they are distributed over the working nodes in a different way.

4. EXPERIMENTAL RESULTS

In this section present our experimental results for both of our parallel DPLL models.

Experimental Setup. We ran both communication models on the Euler super compute cluster [11] on up to 48 cores and requested 1 Gigabyte of memory per core. 48 cores was the maximum number of cores accessible to us. Euler contains 5 different types of nodes but each of them contains at least two 12-core Intel Xeon processors (2.5-3.7 GHz) and between 64 and 512 GB of DDR4 memory clocked at 1866, 2133 or 2666 MHz.

Benchmark Formulas. During our correctness testing and debugging phase of the core algorithm, we realized that random formulas generated by our own random formula generator are not suitable for performance testing. Even if we picked formulas that contained the same amount of variables, clauses and literals per clause, we observed large variations in run time between them. This is caused by the completely random structure of formulas generated by our generator. Therefore we used a set of 14 formulas taken from a SATLIB formula collection as a benchmark set. [9] The set contains different real world problems modeled as boolean formulas from various domains such as planning problems, all-interval series encodings, flat graph coloring, inductive inference, etc. Our benchmark set also contains a few random formulas that were taken from the SATLIB formula collection as well. Table 2 in the Appendix section contains a detailed listing of the used formulas together with their description. For the coming subsections we reduced the set of benchmark formulas to a representable subset of size 6 for readability reasons. We included the corresponding figures for the other 8 formulas in the Appendix section.

Master Worker. We compared the run times of the master worker communication pattern with the sequential version of DPLL. The achieved speedup is shown in Figure 2. It can be inferred from the figure that the achieved speedup factor heavily depends on the formula. But it is not just the "size" of the problem or time required to solve a formula with the sequential algorithm that influences the speedup factor. As shown in Table 1 the formulas where we reached the highest speedup are not necessarily the ones with the longest time to solve sequentially or the largest number of variables (highest upper bound on the depth of the decision tree). Note that the best speedup in this subset is achieved for the randomly generated formulas (uf50-01 and uuf50-01).

For the random formulas it is easy to explain why the speedup goes down with more than 32 cores. The decrease is caused by moments in the solving process where the master is no longer able to serve all the workers at the same time. Therefore some workers have to wait longer to obtain the next partial model. This problem starts to occur when the number of cores is larger than 30. The overall average waiting time of all workers, shown in Figure 3, shows this problem. The waiting time is considered to be the time that some worker spends waiting for a new partial model from the master. The overall waiting time is the sum of all waiting time periods of all workers, averaged over different runs of the same formula.

For some real world problems the speedup is smaller. The small speedup is caused by the way the decision tree is traversed. While the sequential implementation deterministically does a depth-first search and therefore gives us some ordering guarantees, the master worker implementation sends all subtrees to the master in the form of partial models and gives no ordering guarantees. It might happen that a worker is "close" to a solution but then receives a partial model in a completely different part of the decision tree that might be unsatisfiable all together. The size of the subtree where we will not find a solution depends on the structure of the formula. This is also the explanation why we achieved super linear speedup in some other cases: There we were lucky enough to reach the final solution with less iterations than in the sequential case.

Work Stealing. Similar to the previous comparison, we compared our parallel work stealing model to the sequential DPLL version. The speedup that we achieved is shown in Figure 4. With the work stealing model we achieved large

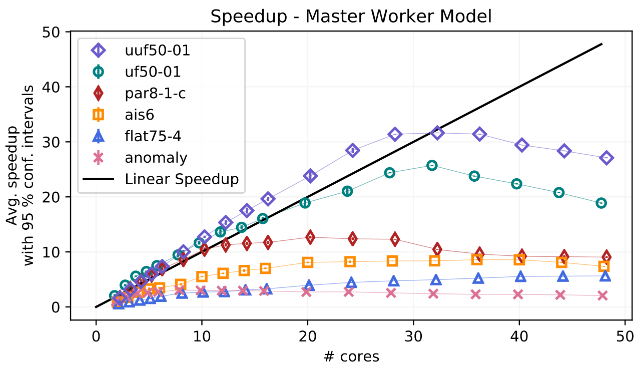

Fig. 2. Per formula average speedup of master worker parallel DPLL implementation compared to sequential DPLL.

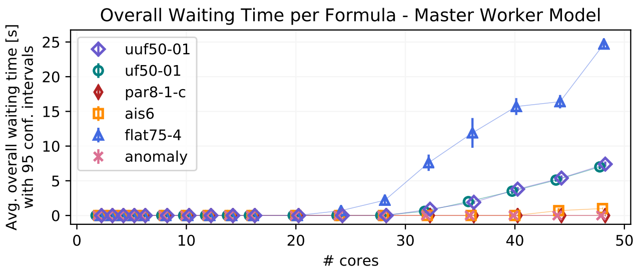

Fig. 3. Average overall waiting time of workers per formula in the master worker model. All waiting times per worker are summed up.

Formula	seq. runtime [ms]	# vars/clauses
uuf50-01 (unsat)	13'404.2 ± 102.94	50/218
uf50-01 (sat)	7'890.9 ± 44.31	50/218
par8-1-c (sat)	2'185.0 ± 32.88	64/254
ais6 (sat)	2'964.0 ± 41.54	61/581
flat75-4 (sat)	26'284.6 ± 172.65	225/840
anomaly (sat)	279.7 ± 6.74	48/261

Table 1. Overview of sequential runtime of benchmark subset formulas. The second column shows the average runtime with the 95 percentiles.

speedups for a small number of cores but essentially did not gain from adding more cores after some limit. The reason for this is again the way how we traverse the decision tree. In theory we should do something that is similar to a global breath-first search and local (per node) depth first search in this model. However because of phase 0 mentioned in Section 3, the "global breath-first search" part is not really a breath-first search. Each worker starts in a node in the decision tree that is one level shifted inwards from the rightmost subbranch of the tree. That means that if the formula is satisfiable, we are guaranteed that at least one worker is working on the correct subtree already after the first decision and will never solve models of the other subtree.

As an result, we outperform the master worker model with the work stealing model with only a few cores. But do not gain that much from adding more cores since the search space is split in a non-uniform way. The overall waiting time of the workers for the work stealing model is shown in Figure 5. The waiting time does no longer go up when increasing the number of cores.

Communication Overhead. One of the reasons why we introduced a work stealing mechanism, was that it should drastically reduce the amount of communication (bytes transferred) between cores. Our experimental results prove that and they are shown in Figure 6. From the upper subplot in Figure 6 it can be inferred that the amount of communication is quite high in the master worker model, but does not necessarily increase when increasing the number of cores. This constant behavior is caused by the fact that every worker sends every partial model it finds and that it does not directly solve, back to the master, regardless of the number of cores that are used. If more cores are used however, it can happen that parts of the decision tree that were not traversed with a smaller number of cores are suddenly traversed. In the lower subplot of Figure 6 the amount of overall communication for the work stealing model is shown. The amount of communication is reduced, compared to the master worker model. This reduction is caused by the reduced amount of partial model transfers, as mentioned in Section 3. However as we discussed in the previous subsection, the amount of communication is not the only limiting factor. Master becoming the bottleneck when adding more cores is the main factor.

5. DISCUSSION

In this section we discuss some approaches that we tried during our project but that did not work as expected or we had to stop investigating in because of time constraints.

DPLL-CDCL Hybrid Parallel Solver. We have implemented a fully working CDCL solver. Its correctness was proven with the same testing infrastructure that we used to test our DPLL implementation. The sequential version of the CDCL solver was faster than the sequential DPLL implementation for all of our benchmark formulas. Quite naturally we tried to plug that solver in locally at the workers to boost the performance. The way we did it is the following: First we run DPLL in parallel (it does not matter which of the two communication models we pick) and we only branch a certain number of times per worker. After that ”branching limit” is reached we switch to CDCL and solve the whole subtree of the original decision tree locally. When CDCL terminates, we either found a solution or get a new model (either from the master or another worker) and solve it again with CDCL. This hybrid approach resulted in

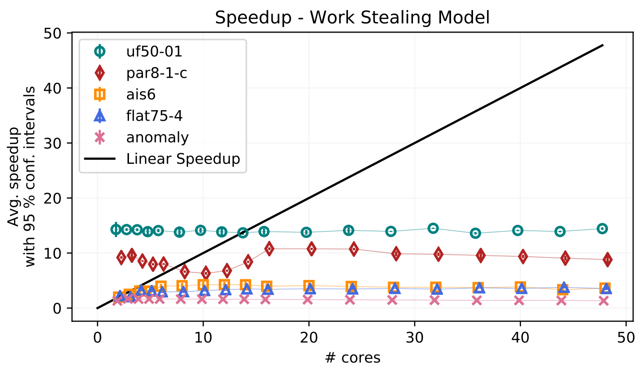

Fig. 4. Speedup of work stealing parallel DPLL implementation compared to sequential DPLL.

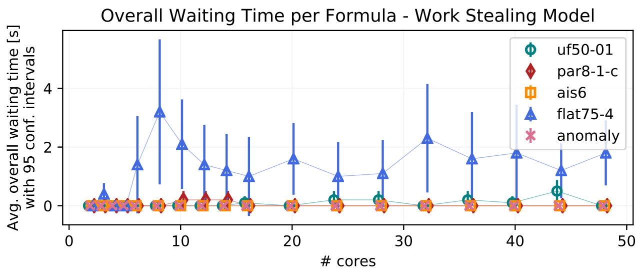

Fig. 5. Average overall waiting time of workers per cnf in the work stealing model. All waiting times per worker are summed up.

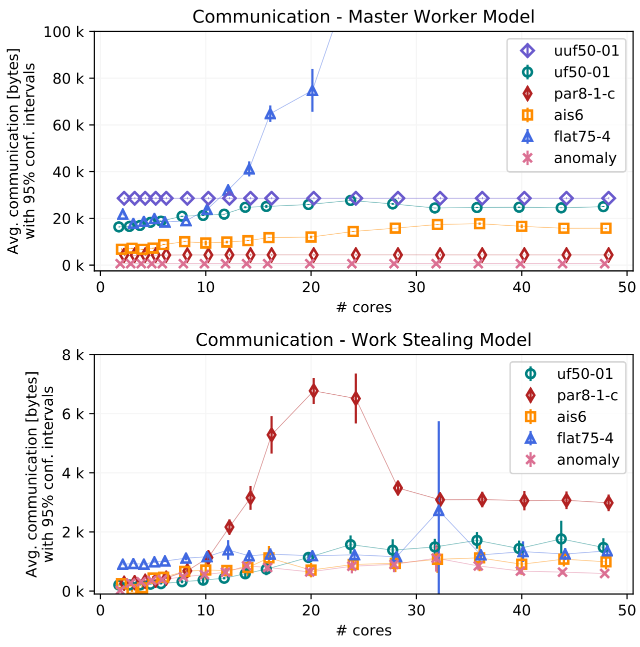

Fig. 6. Overall amount of transferred bytes per formula for both the master worker model and the stealing model.

worse performance than we expected. The reason for this is the following: When running DPLL globally, we essentially create multiple smaller problems. Smaller problems for the DPLL algorithm, which means they are easier and faster to solve than the original problem for a DPLL solver. For the CDCL algorithm on the other hand those subproblems are not necessary easier. Some of those subproblems might actually be a lot harder than the original problem, because with the made assumption by a DPLL decision step we could add a new conflict that was not there in the original problem.

We ran everything we discussed in Section 4 with the hybrid parallel solver. We do not include any further information in this report because of space limitations. Overall the performance of the DPLL-CDCL hybrid parallel solver was worse than both sequential CDCL and parallel DPLL with both of the two parallelization models.

6. FUTURE WORK

In this section we discuss possible extensions and improvements of our work.

Improvements. As presented in Section 4 we detected several drawback in our approach during the evaluation and measurements phase of our project.

The first and most important one is the decision tree traversal. One would need to ensure a more uniform distribution of the search space among the workers for both models. For instance we should avoid situations where one or no worker is working on the left part of the decision tree and all the other workers on the right part.

A second improvement would be to enhance the stealing effectiveness. Stealing should no longer be completely random. One could try to remember the stealing attempts and steal from workers that have the most partial models in their local stacks.

Extensions. Our current work stealing model implementation is only able to handle *sat* formulas. For handling *unsat* formulas one would need to detect if all workers are trying to steal work.

Another interesting extension would be to fully parallelize the CDCL algorithm. It would require sharing of learned clauses, which could be implemented by extending both of our communication models.

7. CONCLUSION

SAT solving is after six decades of intensive research still a hot topic in computer science. In our approach we tried to leverage more processor cores to get more computing power for solving SAT problems. We parallelized the DPLL algorithm with two different communication models. A master worker model with a central management actor (the master) and a work stealing model where we try to reduce the communication overhead to a minimum. Even though we identified deficiencies in both of our models we were still able to outperform our sequential DPLL implementation with both models. For some input formulas we were even able to outperform our sequential CDCL implementation.

8. REFERENCES

[1] Tomas Balyo, Peter Sanders, and Carsten Sinz, “Hordesat: A massively parallel portfolio sat solver,” *Theory and Applications of Satisfiability Testing*, vol. 18, pp. 156–172, 2015.

[2] W. Chrabakh and R. Wolski, “chaff-based distributed sat solver for the grid.,” *ACM/IEEE conference on Supercomputing.*, vol. 37, 2003.

[3] L. Gil, P. Flores, and Silveira L.M., “Pmsat: a parallel version of minisat.,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 73–101, 2008.

[4] Y. Hamadi, S. Jabbour, and L. Sais, “Manysat: a parallel sat solver.,” *Journal on Sat- isfiability, Boolean Modeling and Computation*, vol. 6, pp. 245262, 2008.

[5] Bernard Jurkowiak, Chu Min, and LiGil Utard, “A parallelization scheme based on work stealing for a class of sat solvers,” *Journal of Automated Reasoning*, vol. 34, pp. 73101, 2005.

[6] Till Berger and David Sabel, “Parallelizing dpll in haskell,” 02 2013.

[7] Martin Davis and Hilary Putnam, “A computing procedure for quantification theory,” *J. ACM*, vol. 7, no. 3, pp. 201–215, July 1960.

[8] “MPI Forum,” online: http://mpi-forum.org/, Accessed: 15.12.2017.

[9] “SATLIB - Benchmark Problems, University of British Columbia,” online: http://www.cs.ubc.ca/˜hoos/SATLIB/benchm.html, Accessed: 15.12.2017.

[10] Leonardo De Moura and Nikolaj Bjørner, “Z3: An efficient smt solver,” in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Heidelberg, 2008, TACAS’08/ETAPS’08, pp. 337–340, Springer-Verlag.

[11] “Euler High-Performance-Cluster, ETH Zurich,” online: https://scicomp.ethz.ch/wiki/Euler, Accessed: 15.12.2017.

9. APPENDIX

This section contains additional information that is not strictly part of the report.

Source Code Repository. All of our solver, benchmarking and plotting source code can be found in he following github repository: *https://github.com/limo1996/SAT-Solver/releases/tag/project-submission*

Benchmark Formulas. Table 2 contains a list and description of the formulas that we used as our benchmark set.

Master Worker. The analogous plots to Figures 2 and 3 for the second subset are shown in Figures 7 and 8.

Work Stealing. The analogous plots to Figures 4 and 5 for the second subset are shown in Figures 9 and 10.

Communication Overhead. The analogous plot to Figure 6 for the second subset is shown in Figure 11

CDCL. In Section 5 we stated that we implemented the CDCL algorithm and that it is faster than sequential DPLL on all of our benchmark formulas. Figure 12 shows the run times of our CDCL and our DPLL implementation for all formulas in of our benchmark set.

Overall Fastest Implementation. In Table 3 we list all of the benchmark formulas with the implementation that solves them in the least amount of time. For half of the formulas it is sequential CDCL, for the other half the master worker parallel DPLL implementation with varying number of cores.

File names	Description
anomaly, medium	SAT-encoded blocks world planning problems
ais6, ais8	SAT-encoded All-Interval Series problems
flat50-1, flat75-4, flat 75-8	SAT-encoded "Flat" Graph coloring problems (J. Coberson's flat graph generator is used
ii8a1	Inductive inference, stem from a formulation of boolean function synthesis problems
par8-1-c, par8-4-	Instances for learning the paritiy function
uf50-01, uf75-01	Uniform random 3-sat, satisfiable
uuf50-01, uf50-02	Uniform random 3-sat, unsatisfiable

Table 2. Description of benchmark problems.

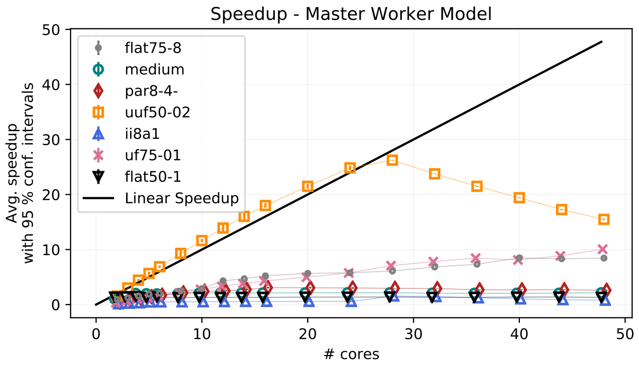

Fig. 7. Average speedup of master worker parallel DPLL implementation compared to sequential DPLL. Second subset.

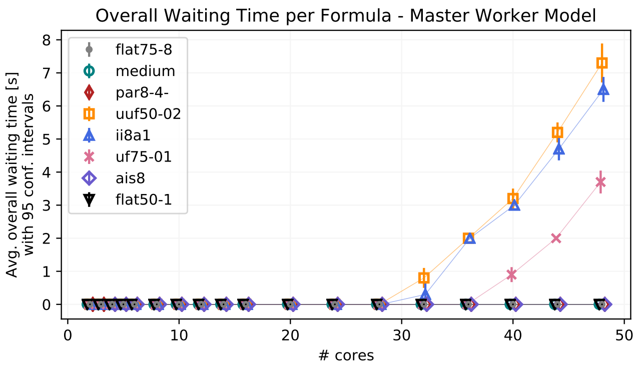

Fig. 8. Average overall waiting time of workers per formula in the master worker model. All waiting times per worker are summed up. Second subset.

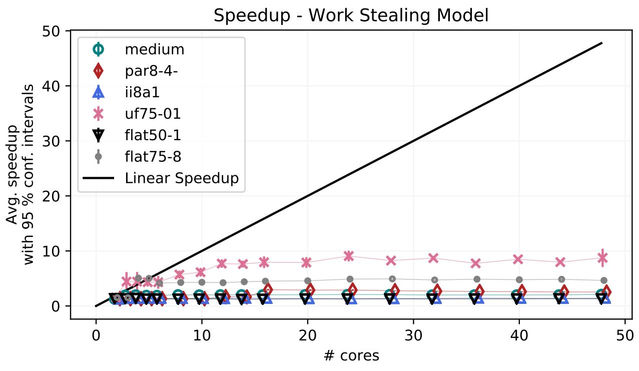

Fig. 9. Average speedup of working stealing parallel DPLL implementation compared to sequential DPLL. Second subset.

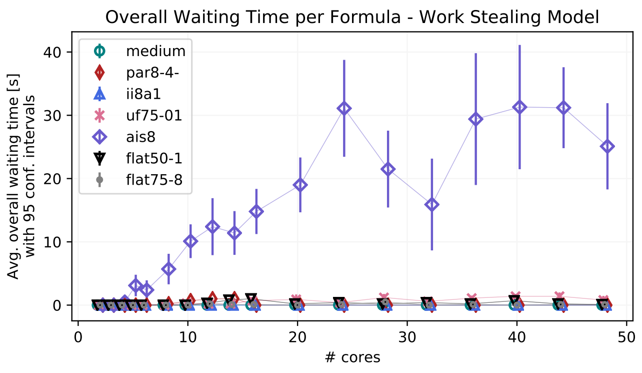

Fig. 10. Average overall waiting time of workers per formula in the work stealing model. All waiting times per worker are summed up. Second subset.

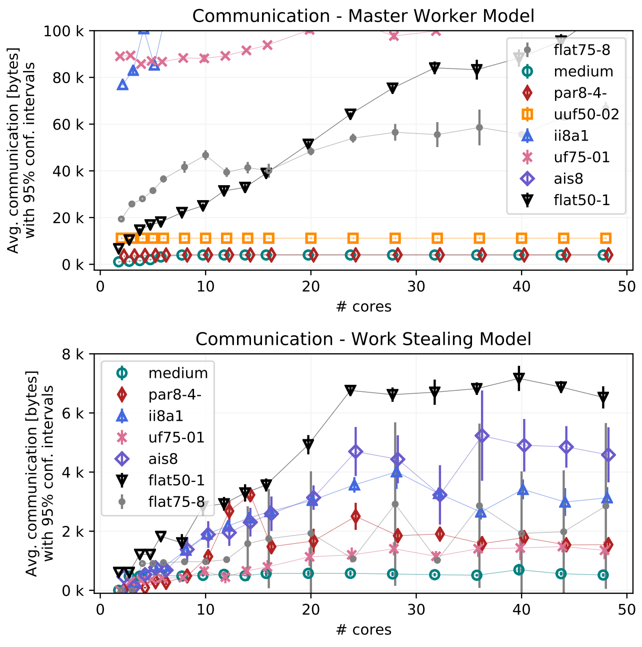

Fig. 11. Overall amount of transferred bytes per formula for both the master worker model and the stealing model. Second subset.

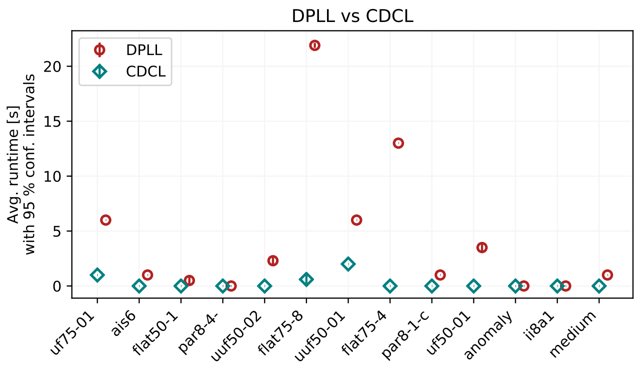

Fig. 12. Average runtimes of DPLL and CDCL.

Formula	Best Solver Configuration	Runtime [ms]
anomaly	CDCL sequential	16.2 ± 4.18
medium	CDCL sequential	414.8 ± 32.94
ais6	DPLL master worker, 14 cores	343.3 ± 3.58
ais8	DPLL master worker, 48 cores	3015.3 ± 58.85
flat50-1	CDCL sequential	36.4 ± 9.07
flat75-4	CDCL sequential	702.8 ± 38.18
flat75-8	CDCL sequential	1054.2 ± 50.72
ii8a1	DPLL master worker, 6 cores	72.2 ± 0.79
par8-1-c	CDCL sequential	15.2 ± 2.28
par8-4-	DPLL master salve, 2 cores	132.2 ± 2.22
uf50-01	CDCL sequential	31.5 ± 9.21
uf75-01	DPLL master worker, 32 nodes	297.4 ± 7.99
uuf50-01	DPLL master worker 24 nodes	187.5 ± 5.59
uuf50-02	DPLL master worker 24 nodes	148.9 ± 2.85

Table 3. Benchmark formulas with best performing configuration. The third column shows the average runtime with the 95 percentiles.