PARALLEL SAT SOLVING

Jan Eberhardt, Jakub Lichman

Department of Computer Science
ETH Zurich
Zurich, Switzerland

The hard page limit is 6 pages in this style. Do not reduce font size or use other tricks to squeeze. This pdf is formatted in the American letter format, so the spacing may look a bit strange when printed out.

ABSTRACT

Describe in concise words what you do, why you do it (not necessarily in this order), and the main result. The abstract has to be self-contained and readable for a person in the general area. You should write the abstract last.

1. INTRODUCTION

Motivation. Boolean satisfiability problem (SAT) belongs to the most important problems in program analysis, verification and other disciplines of theoretical computer science. It is particularly used in background of many applications, especially ones in the field of automated planning and scheduling, model checking(formal verification) and theorem proving.

Last decade brought many improvements to SAT world in form of advanced heuristics, preprocessing and inprocessing techniques and data structures that allow efficient implementation of search space pruning.

However, past 10 years were also rich on improvements in parallelism. Current trends in computer hardware design decreased performance per processing unit and pack more units on a single processor. It is caused by thermal wall which stopped further increase of clock speed. However, algorithms for SAT solving like DPLL and CDCL were invented before wide use of parallelism and therefore were designed for sequential execution. Since SAT is a NP-complete problem, we consider it the right candidate for running in parallel.

In our approach, we are trying to speed up SAT solving by running it on multiple cores with different techniques of search space partitioning. Final comparison is done between different parallel versions and sequential one. Parallel versions are mainly based on DPLL algorithm except the one which uses also CDCL, but locally. All algorithms are unlimited in number of cores they can run on. However, some scale better than the others.

Experiments were run on the cluster where we were allowed to use at most 48 cores. Tests were taken from SATLIB - The Satisfiability Library [1] and some were also created by our own random generator of formulas. Results show nice speedups in parallel versions against sequential one. However, parallel DPLL algorithm is still not able to outperform sequential CDCL. It shows how good CDCL actually is in comparison with DPLL.

Related work. Tomas Balyo et al. in their paper [2] propose HordeSat solver which can run up to 1024 cores and is based on CDCL algorithm. Their parallel approach is different from ours because it is portfolio based but with sign of search space partitioning. Most of the previous SAT solvers designed for computer clusters or grids use explicit search space partitioning. Examples of such solvers are GridSAT [3], PMSAT [4] or ManySat [5]. Paper that is probably closest to ours [6] firstly introduced work stealing for dynamic load-balancing.

2. BACKGROUND: WHATEVER THE BACKGROUND IS

CNF. A *boolean variable* is a variable that can be assigned either to *true* or *false*. A *literal* of a boolean variable x is considered to be in positive x or negative $\overline{x}$ form. A *clause* is then disjunction (OR) of literals. A *conjunctive normal form* (CNF) is a conjunction of such a clauses. CNF is usually represented by a number of variables and clauses. However, measuring difficulty of CNF by these two factors is very inaccurate.

SAT. SAT solver is a program that is able to decide whether given formula is satisfiable. More formally, given formula F is satisfiable *iff* there *exists* assignment of literals θ that makes whole formula (*CNF*) true (*satisfiable*). If there does not exits such an assignment then formula is *unsatisfiable*. Furthermore, every SAT solver should be able in *satisfiable* case provide valid assignments of literals as well.

History. First algorithm was develop in 1960 by Martin Davis and Hilary Putnam [7] for checking the validity

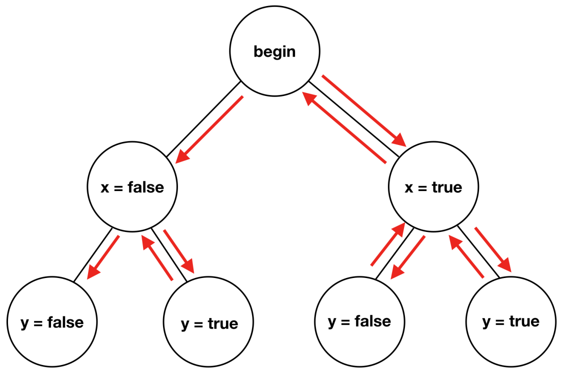

Fig. 1. The way how DPLL explores search space of CNF with two variables *x* and *y*.

of a first-order logic formula using a resolution-based decision procedure for propositional logic. Since Davis-Putnam algorithm was able to handle just valid formulas, more general approach for SAT solving was needed. In 1962 was developed new, complete algorithm that was able to handle all types of formulas. The algorithm is called *DPLL* [8] after Davis, Putnam, Logemann, and Loveland. It is backtracking-based search algorithm that still forms the basis for most efficient complete SAT solvers. Since DPLL invention, there were many algorithms proposed, which improved runtime of SAT solving significantly.

DPLL. DPLL solves a SAT problem by modelling 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 it later turns out that this decision was wrong it backtracks to that point and picks the negated assignment for variable v_i.

CDCL. The algorithm performs as well as DPLL a depth-first search of the space of partial truth assignments. In addition to DPLL, Conflict-Driven Clause Learning (*CDCL*) adopts a pruning technique called learning. While in DPLL we can solve same clauses multiple times, CDCL remembers them and in the next encounter avoids them. More formally, learning extracts and memorizes information from the previously searched space to prune the search in the future [9]. Learning is done by adding clauses to the existing clause database. Clauses are analyzed and stored whenever search reaches conflict state. If it cannot be resolved by backtracking then the formula is unsatisfiable. If all the variables are assigned and no conflict happened then the formula is satisfiable. [2]

3. PARALLELIZING DPLL

DPLL is a simple backtracking algorithm that explores search space without any advanced techniques like learning and pruning. As an result of this, it does not require any transfer of information between stages of solving and therefore it can be nicely parallelized. Parallelization is not trivial but far easier than in case of CDCL where in addition to load balancing we need to deal with learned clause sharing.

DPLL Branches. As shown in figure 1 the DPLL algorithm explores search space with depth first search. However it needs to make a decision at some point. If there are no more literals that can be assigned trivially, we need to pick the one and assume it to be either *true* or *false*. That is exactly the point where we can do so called DPLL branching, i.e. let some other node solve the other branch. We can call this point branching point and it represents one node in DPLL search space graph.

In our work we considered two approaches to load balancing. First one is a master worker communication pattern, where one node called *master* is in charge of storing partial models and eventually passing them to *workers* that will solve them. Second approach is a work stealing communication model where each node (*worker*) runs on its own and therefore 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 partial model from it.

Master Worker Model. First approach that we have decided to design and implement is master worker model. We considered this model to be easiest to implement and therefore we picked it first. The biggest challenge in parallelizing DPLL is load balancing. The work cannot be split equally between workers at the beginning because it would require knowledge of a depth of each branch that cannot be inferred from a formula at the beginning. We want to avoid situations where work is not divided equally because we want to fully exploit computing power of all nodes.

Master is one of *n* nodes available and other *n-1* nodes become *workers*. At the beginning are all workers marked as free. Master starts solving model by passing empty model to the first free worker. Each worker in every branching point takes one branch and sends the other to the master. When master receives partial model, it checks whether there exists some free worker. If not then it stores the model into its local stack and sends it back in case some worker finished its branch with *unsat* and therefore needs more work. If there is still some free worker then master bypasses the partial model to it. This procedure is repeated until some worker does not find *sat* model and sends it to the master. Master then outputs the model and stops all workers. If no worker found *sat* model, master has empty stack and all workers are free we know that formula is *unsat*.

While master worker model is easy to understand and implement it comes with several drawbacks. First of them is lack of one node. Master worker model has one node used only as an manager and therefore this node is not actually doing any computation. Next issue is related to scaling. Master with certain number of nodes becomes bottleneck because there is too much communication it needs to process and therefore nodes need to wait longer for getting new model. Last problem is related to amount of communication in this model. There is transfer of models and meta data every time worker branches or wants to get a new model. There is also transfer at the beginning and at the end but this communication is necessary while the one mentioned before can be reduced.

Work Stealing Model. Because of the drawbacks mentioned above we have decided to reduce and decentralize communication by implementing work stealing model which provides desired features. The model treats all nodes *equally* and therefore there is *no master* just workers. All workers contain their own stacks with models to process. This reduces communication overhead because all produced models are stored locally instead of their transfer and storage on the master. However, existence of local stack breaks load balancing because some workers can work on bigger subtrees of search space than the others.

Inequalities in load balancing are resolved by mechanism called work stealing. In case some worker finished its work and no other worker has solved the formula than it tries to "steal" model from other workers. Process of stealing can be formally defined as follows: Try to steal from random worker until you do not find a model or until someone else has not find it. We need to define two more rules(phases) for algorithm to work. First phase is initial work distribution and second one is ending condition. In initial phase worker 0 will take role of the master and distribute starting models to all workers. Then will all workers try to solve their subtrees and in case of running out of work they will do stealing (phase 2). When some worker will find a *sat* model then it will send it to the worker 0 and it will stop all workers and output final assignment (phase 3). We have to have master node at the end because there can be more workers that find *sat* model at the same time. In our solution, worker 0 prints first obtained sat model and ignores the others.

However, our current work stealing implementation has one drawback. It can only handle satisfiable cases. The stopping criteria in the unsatisfiable case was very trivial in master worker model because it was moment when master's stack was empty and all workers were free. In work stealing no such a moment exits. We do not have central node that controls the others and therefore knows in which stage each worker is. We currently cannot detect if all workers are trying to steal (no worker has work left) and therefore we restrict ourselves just to sat formulas but we plan to solve this issue in our future work.

Implementation. We implemented both communication models in C++ with a help of the MPI [10]. Our DPLL solver is implemented in C++ as well and therefore we can directly interact with MPI within the solver. We tried to keep implementation of a DPLL algorithm and communication within a nodes as separate as possible for better readability. The implementation of a parallel DPLL is straight forward and tries to follow good object oriented programming principles. We adapted test driven development with python wrapper that tests our program as black box. We split program into classes with well defined interfaces according to their functions and applied proper encapsulation and information hiding.

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, literal and variable values. To implement the backtracking algorithm we used recursion, but we allocate all necessary data structures on the heap. We didn't 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. [1] We ran each formula through z3 [11] and compared the result of the 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 the 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, since we essentially solved the same set of subproblems but on the different nodes and in completely different order. The difference between parallel and stealing version is in load balancing and not in searched space, i.e. both versions solve the same partial models but they are distributed over working nodes in a different way.

4. EXPERIMENTAL RESULTS

In this section we are going to present our experimental results for both versions of our parallel DPLL algorithm as well as their detailed description.

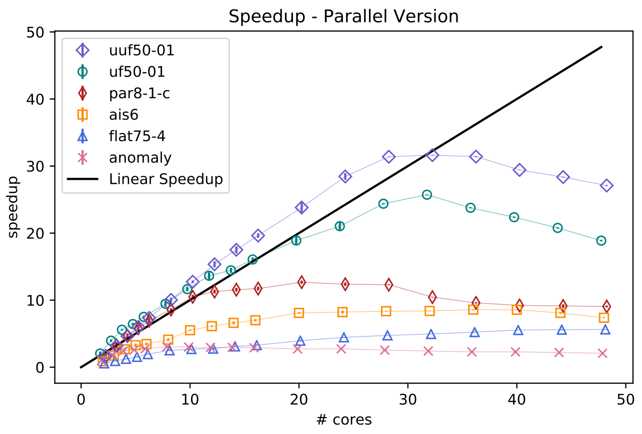

Fig. 2. Average speedup of master worker parallel DPLL implementation compared to sequential DPLL. The 95% confidence intervalls are shown as error bars but too small to be visible in some cases.

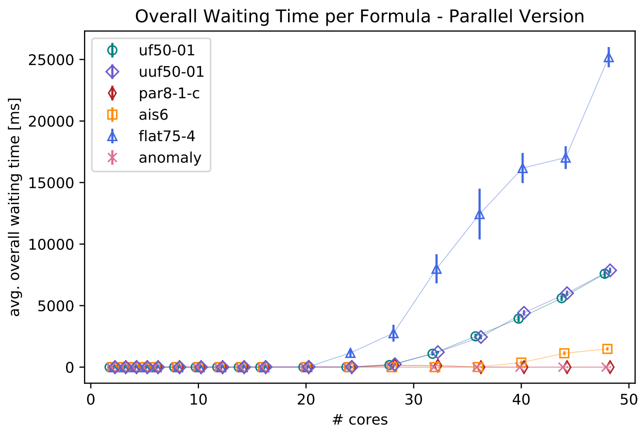

Fig. 3. Average overall waiting time of workers per cnf in parallel version. All waiting times per worker are summed up. The 95% confidence interval are shown as error bars.

Experimental Setup. We ran both communication models on the Euler super compute cluster [12] on up to 48 cores, which was the maximum accessible number of cores to us, and requested 1 Gigabyte of memory per core. 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 the variation of runtimes between them. It is caused by a completely random structure of a formula generated by our generator. Therefore we used a set of 14 formulas taken from the SATLIB as a benchmark set. The set contains different real world problems modeled as a boolean formulas from various domains such as planning problems, all-interval series encodings, flat graph coloring, inductive inference, etc. Our benchmark set also contains few random formulas that were taken from SATLIB 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 and included corresponding figures for the other 8 formulas in the Appendix section. Subset of representatives is described in table 1.

Master Worker. We compared the runtimes of the master worker communication pattern with the sequential version of the DPLL. The achieved speedup is shown in Figure 2. It can be inferred from the figure that achieved speedup factor heavily depends on the formula. However size of the problem together with the time required to solve a formula with the sequential algorithm are not the only influential factors. 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 with the randomly generated formula (uf50-01). For these types of formulas it is trivial to explain why speedup goes down with more than 32 cores. It is caused by moments in solving process when master is no longer able to serve that many workers at the same time and therefore some workers has to wait longer for obtaining next partial model. This problem starts to occur when number of cores overcome 30. Another graph that shows this problem is the overall average waiting time shown in Figure 3. Waiting time in our paper is considered to be time that some worker spends waiting for new partial model from master. The overall waiting time is then sum of all waiting time periods over all workers and then averaged over different runs of the same formula.

The cases where speedup is not so significant are real world problems and therefore their decision trees have some structures. The small speedup is then caused by the way a decision tree is traversed. While the sequential implementation deterministically does a depth-first search and therefore gives us ordering guarantees, parallel version sends all subbranches to a master and gives no guarantees that it will receive them in the next call to a master. It might happen that we are ”close” to a solution but then we receive 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

Formula	seq. runtime [ms]	# vars/clauses
uf50-01 (sat)	7890.9 ± 44.31	50/218
par8-1-c (sat)	2185.0 ± 32.88	64/254
ais6 (sat)	2964.0 ± 41.54	61/581
flat75-4 (sat)	26284.6 ± 172.65	225/840
anomaly (sat)	279.7 ± 6.74	48/261
uuf50-01 (unsat)	13404.2 ± 102.94	50/218

Table 1. Overview of benchmark subset formulas.

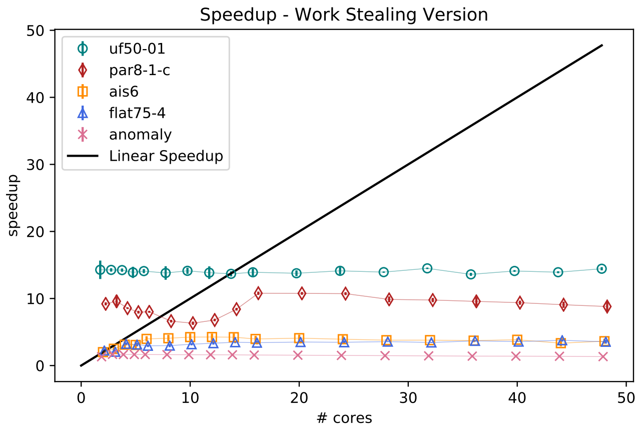

Fig. 4. Speedup of work stealing parallel DPLL implementation compared to sequential DPLL.

we were lucky enough in decision tree traversal and therefore we reached final solution with less iterations than in sequential case.

Work Stealing. Similar to the previous comparison, we compared our parallel work stealing algorithm to the sequential version of the DPLL. The speedup that we achieved is shown in Figure 4. With work stealing pattern we achieved super high speedups for small number of cores but essentially did not gain from adding more cores after some limit. The reason for it is again the way how we traverse the decision tree. In this pattern we do something that is similar to a breath first search globally and locally per node, the traversal order of the subtree is the same as in the sequential implementation, so depth first search. 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, wrong subtree. We do not have this guarantee for the master worker pattern because here workers jump out of their search space context every time they request new model from master. It does not guarantee that model that worker receives is in the same context (subtree) as previous one. Therefore it can easily happen that all workers are working on wrong subtree and they will move to the correct one just after finishing the wrong one. As an result of this, we outperform master worker with work stealing version easily.

Communication Overhead. One of the reasons why we developed work stealing mechanism was that it should drastically reduce communication (bytes transferred) between cores. Our experimental results proved it and they are shown in Figure [xxx]. It can be inferred from the figure that communication overhead rises with number of cores in master worker model but stays more or less the same in stealing one. It is caused by reduced model transfer mentioned in chapter 3. However we observed that amount of communication is definitely not a limiting factor in master worker model. It is master that becomes bottleneck in higher number of cores and communication is in comparison to master serving free.

5. DISCUSSION

In this section we are going to discuss some approaches that we tried during development but they did not work as expected or we had to stop investigating possible solutions because of time constraints.

DPLL-CDCL Hybrid Parallel Solver. We have implemented a fully working CDCL solver. Its correctness was proved the same way as in DPLL ones. Even tough CDCL solver is sequential, it was able to outperform both DPLL solvers on all of our 14 benchmark formulas with highest number of cores. The reason of that is because CDCL is far better algorithm than DPLL even if DPLL runs on 48 cores. Quite naturally we tried to plug that solver in 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 patterns we pick) and we only branched a certain number of times per worker.
- After ”branching limit” was reached we switched to CDCL and solve the whole subtree of the original decision tree locally.
- When it was done we either found a solution or get a new model (either from the master or another worker) and solved it again with CDCL.

However this hybrid approach showed worse performance than we expected. There are two reasons why. Firstly we essentially crippled the CDCL solver by not giving him the full formula but only a modified part of it. That means if we already decided something and then pass that subproblem to the CDCL solver, the subproblem has not really a correlation to the original problem. The problem that we solve with CDCL can be a completely different one and therefore a lot more difficult to solve then the original one. Secondly, yet again we break something because of the way we split the work. It might happen that all workers are busy solving some problem that is a lot more difficult than the actual problem and the subproblem that contains the correct solution is not solved by any worker at that point in time. That means in the worst case we solve lots of partial models that are more difficult than the original problem, before we solve the correct problem and find the solution.

We ran everything we discussed in Section 4 with the hybrid parallel solver for different branching limits. We do not include any further information in this report because of space limitations.

6. FUTURE WORK

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

Improvements. As presented in chapter 4 during evaluation and measurements phase of our project we detected several drawbacks in our approach that could be improved. First and the most important one is tree traversal. We need to ensure equal distribution of workers over search space. We have to avoid situations where one or no worker is working on the left part and all the other workers on the right part. Next improvement should enhance stealing effectiveness. Stealing should no longer be completely random. We need to somehow remember our stealing attempts and steal from workers that have the most models in their local stacks.

Extensions. Work stealing model is able to handle just *sat* cases and we would like to extend it to handle *unsat* cases as well. It would allow us to do better benchmarks because in *unsat* case we need to traverse the whole search tree. Another interesting extension would be to parallelize CDCL algorithm. It would require clause sharing that is hard to implement but it should lead to performance that is at least comparable with state-of-the-art solvers.

7. CONCLUSION

SAT solving is after six decades of intensive research still hot topic in theoretical computer science. In our approach we tried to expose more processor cores to get more computing power for SAT problem that is NP-complete. We parallelized DPLL algorithm with two different communication strategies. Work stealing strategy outperformed master worker one since it does not contain central node as well as reduces communication overhead to minimum. However newer and much better CDCL algorithm was still able to outperform our both parallel DPLL versions even tough it ran sequentially. We also tried hybrid approach with global DPLL and local CDCL but it did not bring desired performance increase. As our future work we mainly plan to parallelize CDCL since we believe that this approach will bring performance that is comparable with moder state-of-the-art solvers.

8. REFERENCES

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

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

[8] Martin Davis, George Logemann, and Donald Loveland, “A machine program for theorem-proving,” *Commun. ACM*, vol. 5, no. 7, pp. 394–397, July 1962.

[9] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik, “Efficient conflict driven learning in a boolean satisfiability solver,” in *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, Piscataway, NJ, USA, 2001, ICCAD ’01, pp. 279–285, IEEE Press.

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

[11] 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.

[12] “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.

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.

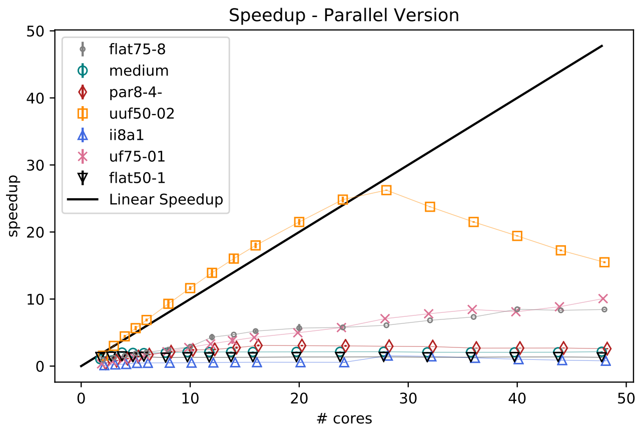

Fig. 5. Average speedup of master worker parallel DPLL implementation compared to sequential DPLL. The 95% confidence intervalls are shown as error bars but too small to be visible in some cases. Other subset.

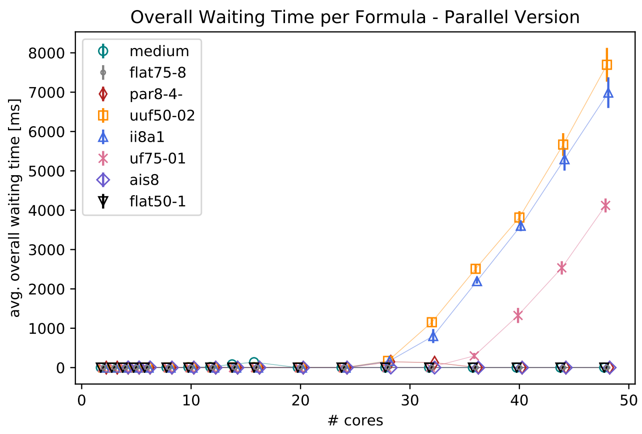

Fig. 6. Average overall waiting time of workers per cnf in parallel version. All waiting times per worker are summed up. The 95% confidence interval are shown as error bars. Other subset.

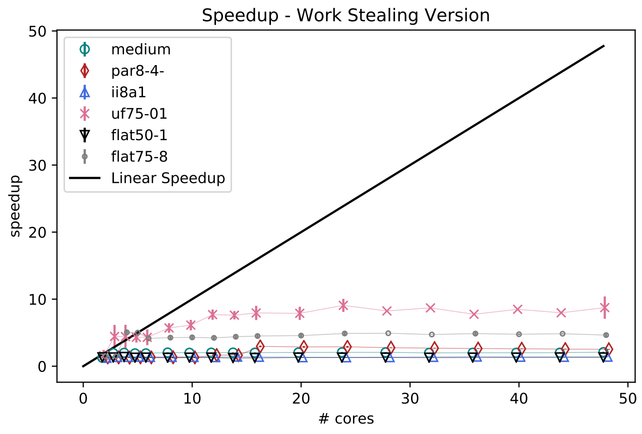

Fig. 7. Speedup of work stealing parallel DPLL implementation compared to sequential DPLL. Other subset.

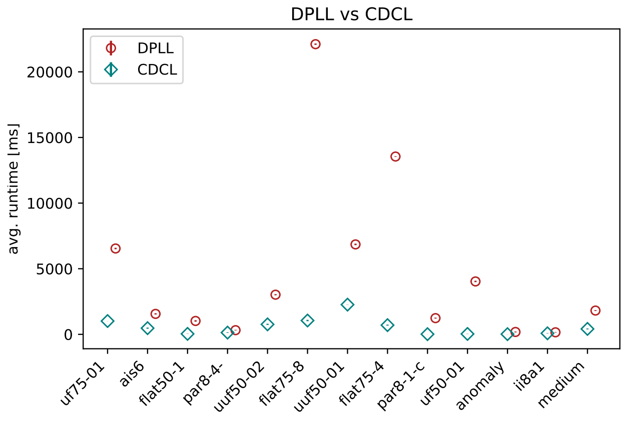

Fig. 8. DPLL vs CDCL sequential runtime. Average runtime with 95% confidence intervals.

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.