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

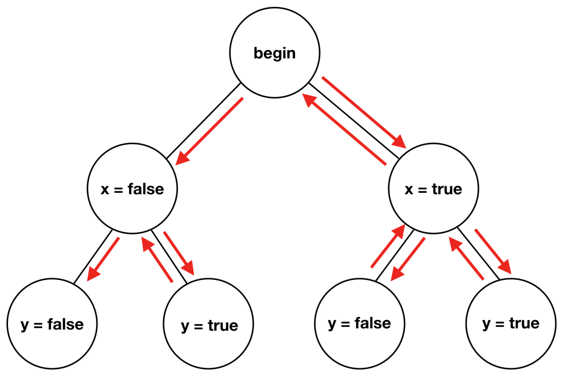

Fig. 1. The way how DPLL explores search space of CNF with two variables *x* and *y*.

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 that, 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 show 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 slave communication pattern, where one node called *master* is in charge of storing partial models and eventually passing them to *slaves* (*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 Slave 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. Its impossible to split work equally at the beginning because it cannot be inferred from a formula how ”deep” each branch actually is. 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 the *n* nodes available and other *n-1* nodes become *slaves* (*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*.

Work Stealing Scheduler. 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 slave 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 slave 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 DPLL algorithm and communication within a nodes as separate as possible for better readability. The implementation of parallel DPLL is straight forward and tries to follows 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 use `vectors` from the C++ standard library to represent sets of clauses and sets of literals. We use `unsigned ints` 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 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.

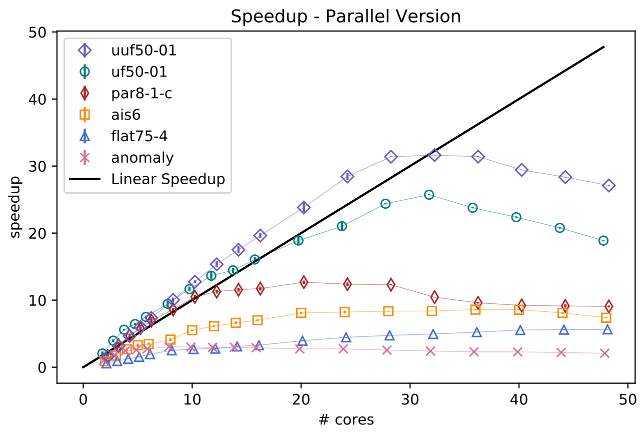

Fig. 2. Average speedup of master slave parallel DPLL implementation compared to sequential DPLL. The 95% confidence intervalls are shown as error bars but too small to be visible in some cases.

4. EXPERIMENTAL RESULTS

We ran both communication models on the Euler super compute cluster. [12].

Experimental Setup. We ran our implementation on up to 48 cores, which was the maximum accessible to us on Euler, and requested 1 Gigabyte of memory per core. //TODO talk about processors, clock frequencies, OS etc necessary?

During our correctness testing and debugging phase of the core algorithm, we realized that random formulas are not a particular good fit to test performance. At least not the kind of random formulas that we generated with our own random formula generator. Even though they used the same amount of variables, clauses and literals per clause the runtimes varied quite a bit because the problems had a completely randoms structure. We used a set of 14 formulas in the DIMACS CNF format as a benchmark set. The set contains different real world problems from various domains such as planning problems, all-interval series encodings, flat graph coloring, inductive inference, etc. We also included a couple of random formulas in our benchmark set. Table 2 in the Appendix section contains a detailed listing of the used formulas. For the coming subsections we reduced the set of formulas to a representable subset of size 6. Corresponding figures for the other 8 formulas can be found in the Appendix section.

Master Slave. We compared the runtimes of the master slave communication pattern with sequential DPLL. The speedup that we achieve is shown in Figure 2. The speedup

Formula	runtime seq. DPLL [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.

factor that we achieved 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 largest number of variables, and therefore highest upper bound on the depth of the decision tree, or longest time to solve sequentially. Note that the best speedup in this subset is actually achieved for the formula that is randomly generated (uf50-01). For the random generated formula we can explain why the speedup goes down after about 32 cores: After about 32 cores the master in our master-slave communication pattern starts to be a bottleneck and workers have to start waiting for pieces of work. The overall average waiting time is shown in Figure 3. The waiting time is summed up over all the workers per solved formula and then averaged over different runs of the same formula.

The speedup is so small in some other cases because of the way the decision tree is traversed. The sequential implementation deterministically does a depth-first search. In this parallel version however we do not have any ordering guarantees. It might happen that we were ”close” to a solution but then solve some other partial models 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 achieve super linear speedup in some other cases: there we are lucking and because of the traversal order find a solution faster than in the sequential traversal.

Stealing Scheduling. Similar to the previous comparison, we compared the parallel work stealing algorithm to sequential DPLL.

The speedup that we achieve is shown in Figure 4. With this work stealing pattern we achieve quite high speedups for small numbers of cores but essentially do not gain from adding more cores after some limit. The reason for this speedup 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 some worker is work-

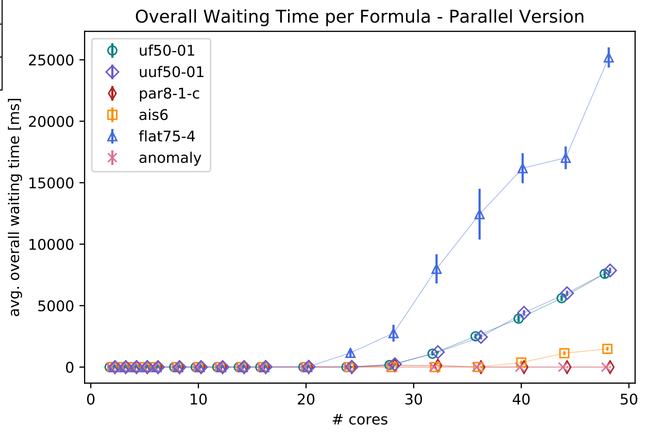

Fig. 3. Average overall waiting time of workers per cnf in parallel version. All waiting times per worker are summed up. The 95% confidence interval are shown as error bars.

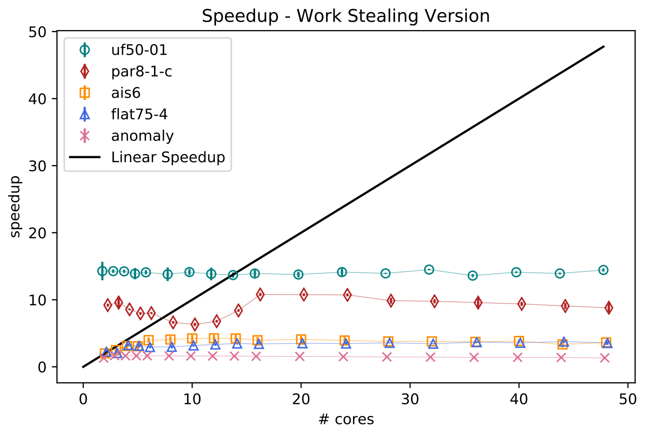

Fig. 4. Speedup of work stealing parallel DPLL implementation compared to sequential DPLL.

ing on the correct subtree already after the first decision and will never solve models of the other, wrong subtree. This guarantee we do not have for the master slave pattern and we therefore beat the master slave pattern easily.

5. DISCUSSION

In this section we will mention some of the things that we tried along the way but did not work out or we had to stop looking closer into because of time constraints.

CDCL DPLL Hybrid Parallel Solver. We have implemented a fully working CDCL solver. If run sequentially it outperforms the DPLL solver on all of our 14 benchmark formulas. Quite naturally we tried to plug that solver in at the workers. The way we did that is the following:

- First we run DPLL in parallel (it does not matter which of two communication patterns we pick). 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 that is done we either found a solution or again get a new model (either from the master or another worker) and solve it with CDCL.

With this hybrid approach we achieved really bad performance. There are two reasons why. Firstly we essentially cripple the CDCL solver by not giving him the full formula but only modified a part of it. That means if we already decided something and then pass on 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 this 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 discuss possible extensions of our work.

7. CONCLUSIONS

Here you need to summarize what you did and why this is important. *Do not take the abstract* and put it in the past tense. Remember, now the reader has (hopefully) read the report, so it is a very different situation from the abstract. Try to highlight important results and say the things you really want to get across such as high-level statements (e.g., we believe that .... is the right approach to .... Even though we only considered x, the .... technique should be applicable ....) You can also formulate next steps if you want. Be brief. After the conclusions there are only the references.

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.

[13] N.J. Higham, *Handbook of Writing for Mathematical Sciences*, SIAM, 1998.

[14] W. Strunk Jr. and E.B. White, *Elements of Style*, Longman, 4th edition, 2000.

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.

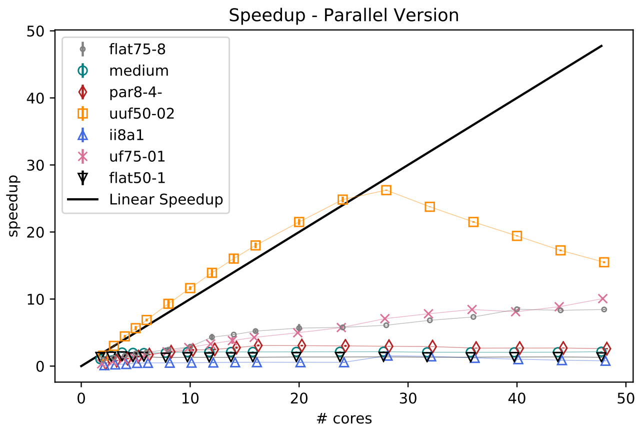

Fig. 5. Average speedup of master slave parallel DPLL implementation compared to sequential DPLL. The 95% confidence intervalls are shown as error bars but too small to be visible in some cases. Other subset.

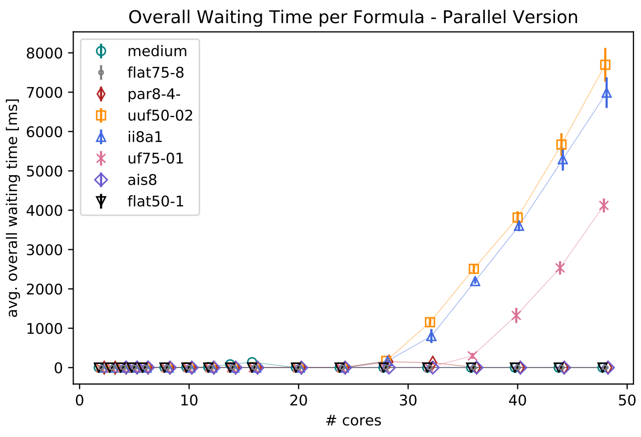

Fig. 6. Average overall waiting time of workers per cnf in parallel version. All waiting times per worker are summed up. The 95% confidence interval are shown as error bars. Other subset.

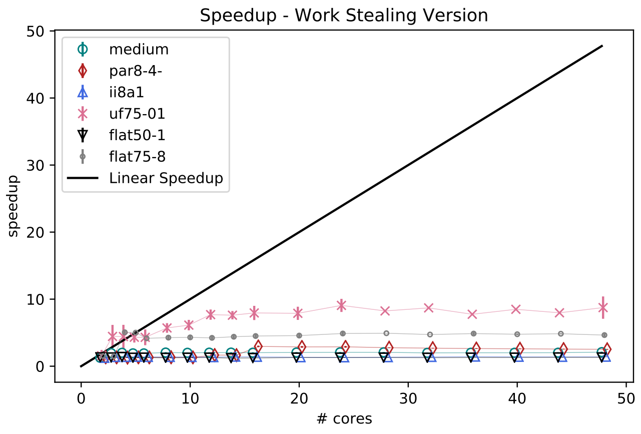

Fig. 7. Speedup of work stealing parallel DPLL implementation compared to sequential DPLL. Other subset.

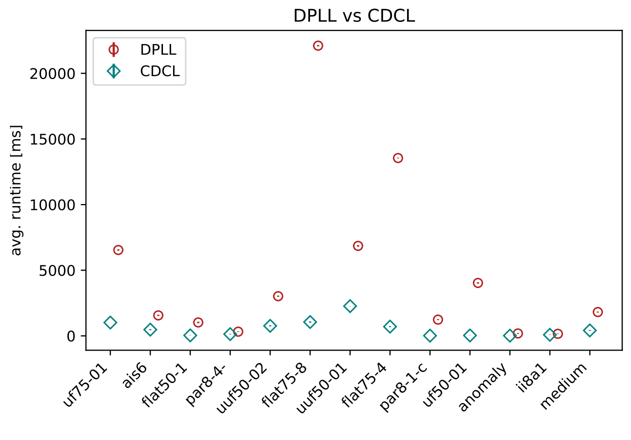

Fig. 8. DPLL vs CDCL sequential runtime. Average runtime with 95% confidence intervals.

Formula	Best Solver Configuration	Runtime [ms]
anomaly	CDCL sequential	16.2 ± 4.18
medium	CDCL sequential	414.8 ± 32.94
ais6	DPLL master slave, 14 cores	343.3 ± 3.58
ais8	DPLL master slave, 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 slave, 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 slave, 32 nodes	297.4 ± 7.99
uuf50-01	DPLL master slave 24 nodes	187.5 ± 5.59
uuf50-02	DPLL master slave 24 nodes	148.9 ± 2.85

Table 3. Benchmark formulas with best performing configuration.