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

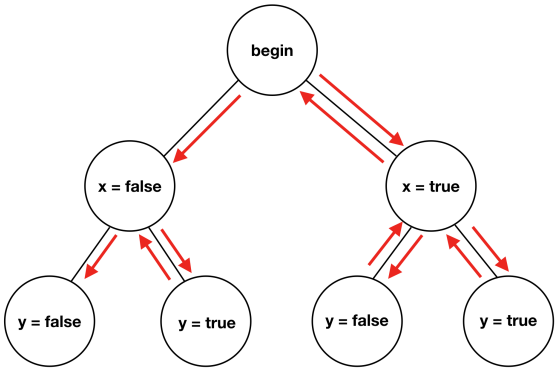

Fig. 1. The way how DPLL explores search space of CNF with two variables *x* and *y*.

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

We decided to parallelize DPLL because it is a relatively simple backtracking algorithm and therefore it does not require any advanced communication. Subtasks can be solved individually and therefore also on different nodes.

DPLL Branches. As introduced in Section 2 the DPLL algorithm at some point needs to make a decision. If there are no more variables that can be assigned trivially, we need to pick one variable and just assume that it is either `true` or `false`. That is exactly the point where we can let some other node solve the other branch. We looked at two different ways on how to split the work between multiple nodes. Firstly a master slave communication pattern, where one node is in charge of storing partial models and eventually passing them on to a slave or worker that will solve it. And secondly a work stealing scheduling communication model where each node runs on its own and if it runs out of work and the formula is not solved yet, it picks a random other node and requests a partial model to work on from that node.

Master Slave Model. //TODO introduce the model and put a nice and compact figure here.

Work Stealing Scheduler. //TODO introduce the model and put a nice and compact figure here. Our current work stealing implementation can only handle satisfiable cases. The stopping criteria in the unsatisfiable case was very trivial in the master slave model, however in the work stealing scheduler it is not trivial: How does one detect if all workers are asking other workers for work, but no worker has work left? Currently we just ignore this case.

Implementation. We implemented both communication models in C++ with MPI [10]. Our DPLL solver is also implemented in C++ and therefore we can directly interact with MPI from within the solver. Our DPLL implementation is straight forward and simple, 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 values for sign and value of the literal and variable. To implement the backtracking algorithm we use recursion, but 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 ourselves or took over from a DIMACS 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 also for each of them that they are correctly solved. Assuming that our sequential implementation is correct, it is straight forward that our parallel implementation is also correct, since we essentially just solve subproblems with the sequential version. All steps of the algorithm stay unchanged, we just store the "other" branch (reformulate this) in a different data structure and resume it potentially at a different point in time.

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 XX formulas in the DIMACS CNF format as a benchmark set. The set contains different real world problems from various domains such as //TODO.

Master Slave. We compared the runtimes of the master slave communication pattern with sequential DPLL. //Question to answer: How well do we scale? The speedup that we achieve is shown in Figure 2 on a representable subset of the benchmark set. The speedup 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 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 or longest time to solve sequentially. Note that the best speedup in this subset is actually achieved for the formula that is randomly generated.

//TODO

//Question to answer: when does master become a bottleneck

Stealing Scheduling. Similar to the previous comparison, we compared the parallel work stealing algorithm to sequential DPLL. //Question to answer: How well do we scale?

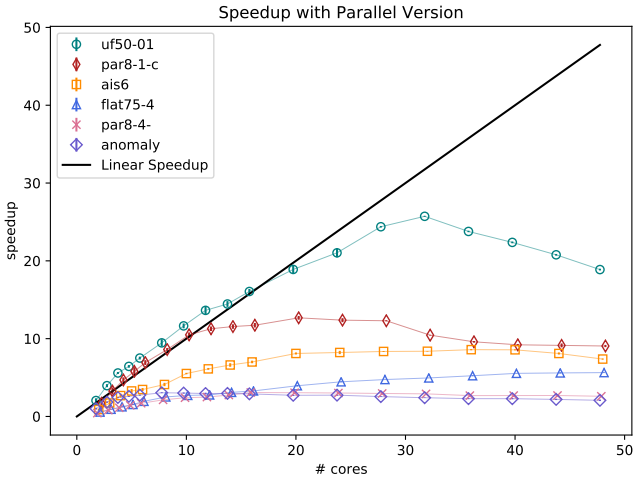

Fig. 2. Speedup of master slave parallel DPLL implementation compared to sequential DPLL.

formula	avg. runtime	# vars/clauses	# decisions
uf50-01	todo	50/218	286
par8-1-c	todo	64/254	32
ais6	todo	61/581	30
flat75-4	todo	225/840	31
par8-4-	todo	67/266	19
anomaly	todo	48/261	5

Table 1. Overview of benchmark subset formulas.

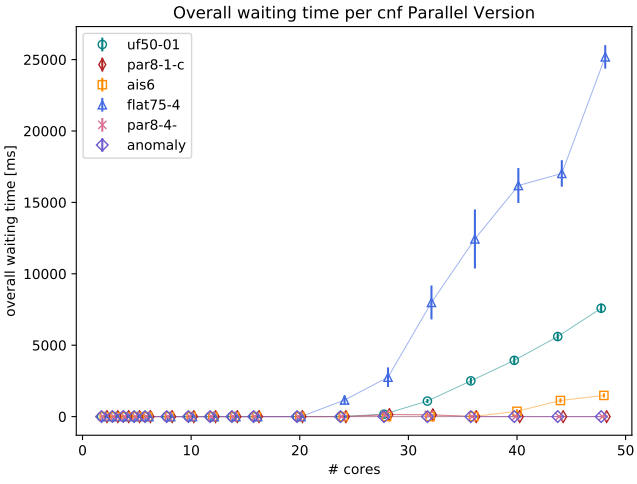

Fig. 3. Overall waiting time of workers per cnf in parallel Version. All waiting times per worker are summed up.

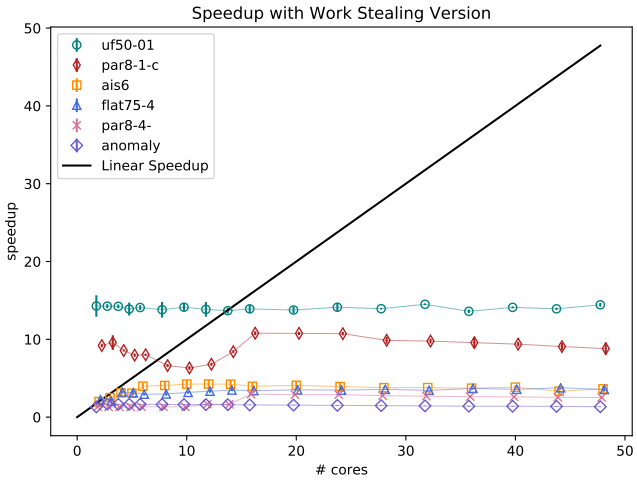

Fig. 4. Speedup of work stealing parallel DPLL implementation compared to sequential DPLL.

5. 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.

6. FURTHER COMMENTS

Here we provide some further tips.

Further general guidelines.

- For short papers, to save space, I use paragraph titles instead of subsections, as shown in the introduction.
- It is generally a good idea to break sections into such smaller units for readability and since it helps you to (visually) structure the story.
- The above section titles should be adapted to more precisely reflect what you do.
- Each section should be started with a very short summary of what the reader can expect in this section. Nothing more awkward as when the story starts and one does not know what the direction is or the goal.
- Make sure you define every acronym you use, no matter how convinced you are the reader knows it.
- Always spell-check before you submit (to us in this case).
- Be picky. When writing a paper you should always strive for very high quality. Many people may read it and the quality makes a big difference. In this class, the quality is part of the grade.
- Books helping you to write better: [13] and [14].
- Conversion to pdf (latex users only):

  dvips -o conference.ps -t letter -Ppdf -G0 conference.dvi

  and then

  ps2pdf conference.ps

Graphics. For plots that are not images *never* generate the bitmap formats jpeg, gif, bmp, tif. Use eps, which means encapsulate postscript. It is scalable since it is a vector graphic description of your graph. E.g., from Matlab, you can export to eps.

The format pdf is also fine for plots (you need pdflatex then), but only if the plot was never before in the format jpeg, gif, bmp, tif.

7. 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.