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.

:why it is important? - because it is used it model checking(formal verification), automated planning and scheduling, Combinatorial design and many more. Many problems are transformed into boolean formulas which are than solved with SAT solvers.

:why you are doing? - Current trends in computer hardware design decrease performance per processing unit and pack more units on a single processor. It happened because thermal wall 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 NP-complete problem, it is the right candidate for running it in parallel.

:what are you doing? - In our approach, we are trying to exploit more cores(bring parallelism to SAT solving). This should lead to significant increase of speed and scalability.

Do not start the introduction with the abstract or a slightly modified version. It follows a possible structure of the introduction. Note that the structure can be modified, but the content should be the same. Introduction and abstract should fill at most the first page, better less.

Motivation. The first task is to motivate what you do. You can start general and zoom in one the specific problem you consider. In the process you should have explained to the reader: what you are doing, why you are doing, why it is important (order is usually reversed).

For example, if my result is the fastest sorting implementation ever, one could roughly go as follows. First explain why sorting is important (used everywhere with a few examples) and why performance matters (large datasets, realtime). Then explain that fast implementations are very hard and expensive to get (memory hierarchy, vector, parallel).

Now you state what you do in this paper. In our example: presenting a sorting implementation that is faster for some sizes as all the other ones.

Related work. Next, you have to give a brief overview of related work. For a report like this, anywhere between 2 and 8 references. Briefly explain what they do. In the end contrast to what you do to make now precisely clear what your contribution is.

2. BACKGROUND: WHATEVER THE BACKGROUND IS

//Jakubs version SAT solver is program that is able to decide whether given formula is satisfiable i.e. there exists assignment of variables that makes whole formula true-satisfiable or not-unsatisfiable. First algorithm was develop in 1960 by Martin Davis and Hilary Putnam for checking the validity 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 procedure 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 after DavisPutnamLogemannLoveland. Its backtracking-based search algorithm that still forms the basis for most efficient complete SAT solvers. Since DPLL invention there were many algorithms proposed, which improve runtime of SAT solving significantly.

SAT. The SAT problem is the following: Given a propositional logic formula F we try to find a valid assignment of variables v_i of F. A valid assignment is one that makes the formula F true. More formally: //TODO

The question that a SAT solver needs to answer is the following: Given a formula F, is it satisfiable? And if so what would be a valid variable assignment?

//TODO Nice small figure here

DPLL. The Davis Putnam Logemann Loveland algorithm, or short DPLL, was introduced by M. Davis, G. Logemann and D. Loveland in 1962. [1] It is an extension of the Davis and Putnam, or short DP, algorithm. [2] DPLL solves a SAT problem by modelling it as a decision tree, variables can either be assigned `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. //TODO [3]

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 [4]. 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. [5] We ran each formula through z3 [6] 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. [7].

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

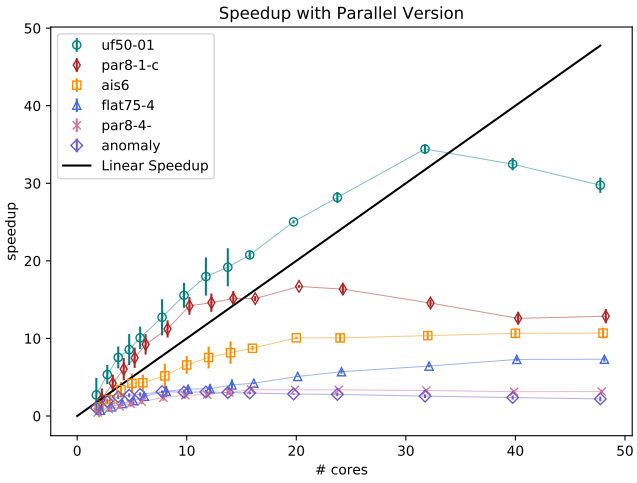

Fig. 1. Speedup of master slave parallel DPLL implementation compared to sequential DPLL.

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.

achieve is shown in Figure 1 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?

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

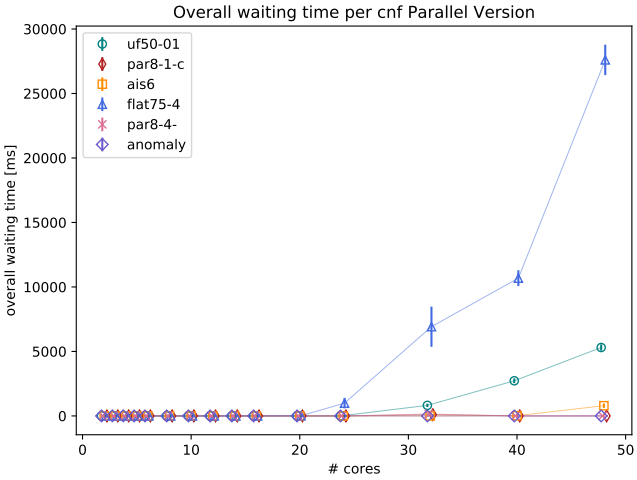

Fig. 2. Overall waiting time of workers per cnf in parallel Version. All waiting times per worker are summed up.

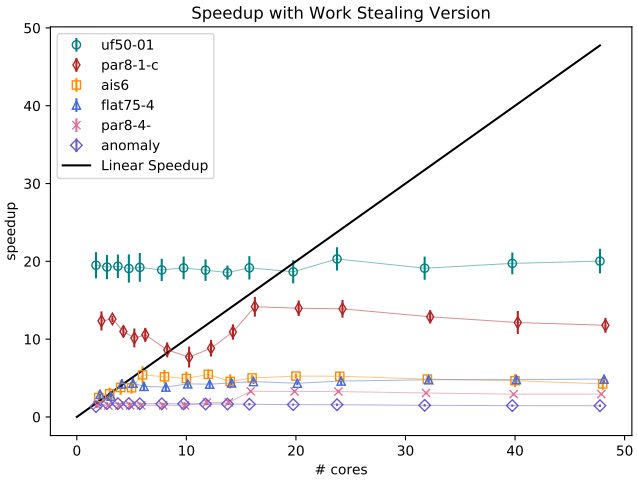

Fig. 3. Speedup of work stealing parallel DPLL implementation compared to sequential DPLL.

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: [8] and [9].
- 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] Martin Davis, George Logemann, and Donald Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, July 1962.

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

[3] 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.

[4] "MPI Forum," online: http://mpi-forum.org/, Accessed: 15.12.2017.

[5] "SATLIB - Benchmark Problems, University of British Columbia," online: http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html, Accessed: 15.12.2017.

[6] 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.

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

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

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