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 cluster where we were allowed to use at most 48 cores. Tests were taken from SATLIB - The Satisfiability Library **??** and they 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 **??** 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 [9], PMSAT [11], GradSat [8], C-sat [26], ZetaSat [6] and SatCiety [28]. Paper that is probably closest to ours [20] firstly introduced work stealing for dynamic load-balancing.

2. BACKGROUND: WHATEVER THE BACKGROUND IS

In this section we are going to have a look at history of SAT solving algorithms as well as new challenges that parallel computing brings to world of SAT solvers.

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.

Give a short, self-contained summary of necessary background information. For example, assume you present an implementation of sorting algorithms. You could organize into sorting definition, algorithms considered, and asymptotic runtime statements. The goal of the background section is to make the paper self-contained for an audience as large as possible. As in every section you start with a very brief overview of the section. Here it could be as follows: In this section we formally define the sorting problem we consider and introduce the algorithms we use including a cost analysis.

Sorting. Precisely define sorting problem you consider.

Sorting algorithms. Explain the algorithm you use including their costs.

As an aside, don't talk about ”the complexity of the algorithm.” It's incorrect, problems have a complexity, not algorithms.

3. YOUR PROPOSED METHOD

Now comes the “beef” of the report, where you explain what you did. Again, organize it in paragraphs with titles. As in every section you start with a very brief overview of the section.

In this section, structure is very important so one can follow the technical content.

Mention and cite any external resources that you used including libraries or other code.

4. EXPERIMENTAL RESULTS

Here you evaluate your work using experiments. You start again with a very short summary of the section. The typical structure follows.

Experimental setup. Specify the platform (processor, frequency, maybe OS, maybe cache sizes) as well as the compiler, version, and flags used. If your work is about performance, I strongly recommend that you play with optimization flags and consider also icc for additional potential speedup.

Then explain what kind of benchmarks you ran. The idea is to give enough information so the experiments are reproducible by somebody else on his or her code. For sorting you would talk about the input sizes. For a tool that performs NUMA optimization, you would specify the programs you ran.

Results. Next divide the experiments into classes, one paragraph for each. In each class of experiments you typically pursue one questions that then is answered by a suitable plot or plots. For example, first you may want to investigate the performance behavior with changing input size, then how your code compares to external benchmarks.

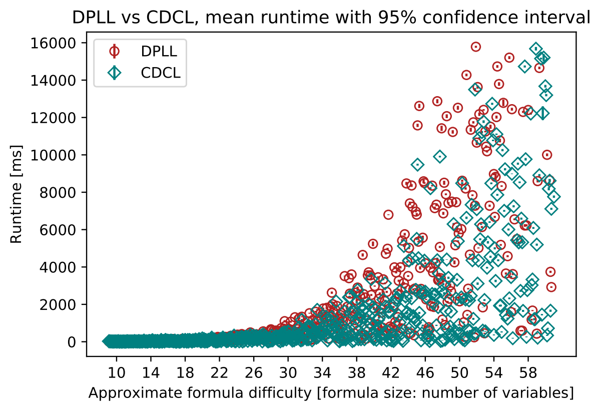

Fig. 1. TODO.

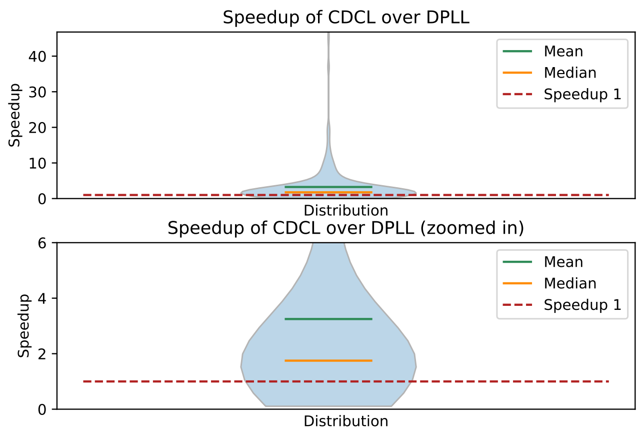

Fig. 2. Speedup of CDCL compared to DPLL. Both algorithms were run sequentially.

For some tips on benchmarking including how to create a decent viewgraph see pages 22–27 in [1].

Comments:

- Create very readable, attractive plots (do 1 column, not 2 column plots for this report) with readable font size. However, the font size should also not be too large; typically it is smaller than the text font size. An example is in Fig. **??** (of course you can have a different style).

- Every plot answers a question. You state this question and extract the answer from the plot in its discussion.

- Every plot should be referenced and discussed.

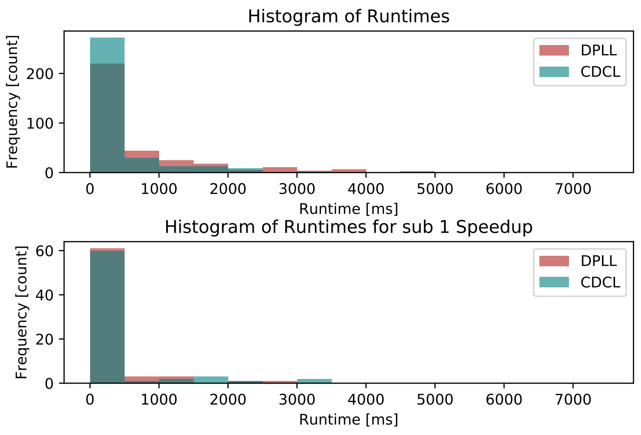

Fig. 3. TODO.

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: [2] and [3].
- 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] M. Püschel, “Benchmarking comments,” online: http://people.inf.ethz.ch/markusp/teaching/263-2300-ETH-spring11/slides/class05.pdf.

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

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