Understanding SMT Solvers

Exploring Parallelization in Floating-Point Problems

Tristan Schmidt[1]

Supervisor(s): Soham Chakraborty[1]**, Dennis Sprokholt**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Tristan Schmidt
Final project course: CSE3000 Research Project
Thesis committee: Soham Chakraborty, Dennis Sprokholt, Andy Zaidman

Abstract

To solve floating-point SMT problems, a variety of algorithms can be used, but there is not one algorithm that truly stands out at solving any kind of problem, as most have their own specific subset of problems where they perform well. A solution to maintaining efficiency in solving any kind of floating-point SMT problem is to run many of these domain specific solvers in parallel, allowing the best to always come out on top. This paper aims to provide an insight into possible improvements using parallelization by running multiple solvers (in parallel) on a large and diverse set of problems. We show that by parallelizing diverse solvers we can obtain a significant speedup over individual solvers on complex problems.

1 Introduction

The floating-point data type and its arithmetic are fundamental building blocks of many computer programs. It is of high importance to test and verify the functionality of programs to prevent any undefined or unknown behaviour, which is commonly caused by rounding errors and other arithmetic anomalies. Verifying programs is one of many tasks that SMT solvers help with. For the floating-point data type it is a particularly important one, as it is one of the most common primitive data types. Even though SMT solvers perform well at these problems, it is often the case that a solver stands out in a subset of problems, there isn't one that stands out in all of them. Many different solvers exist, each with their own unique architecture, but there isn't one that is optimal for solving all floating-point SMT problems.

Parallelization is a simple but promising improvement [12] to solve floating-point problems faster. It can be implemented by making use of the multiple cores modern day computer processors have, where each processor can run one solver, so that they will run simultaneous / in parallel. This technique does not limit the CPU access of other solvers so that, in theory, the fastest solver will always yield the solution of a problem first. A *portfolio solver* approach like this is sometimes used internally in solvers, but the technique is not often applied between different solvers. With this approach, we aim to make a solver that is always at least as fast as the fastest solver (available to us), which will be verified by testing our solver on benchmarks from the 2018 SMT-COMP [21].

This paper aims to explore and address the possible optimizations in solving floating-point SMT problems by applying parallelization. Specifically we use a tool to run diverse and well-known SMT solvers in parallel to see how they perform on different benchmarks, from this we will conclude the effect and possible optimizations obtained from parallelization, as well as an analysis on how the underlying solvers relate to the results they produced. Specifically we contribute the following:

- We provide a detailed overview of the current state-of-the-art floating-point SMT solvers.
- We discuss the current advancements of parallelism in SMT solving and its limitations.
- We use a portfolio solver that deploys different solvers to explore the possible benefits of parallelization.
- We compare our results to individual floating-point solvers, which show improvements in the amount of problems that can be solved and the time used in total when given a time limit for each problem.

Our results show that in problems that take more time to solve, parallelization matches the cumulative solving capabilities of every individual solver, resulting in less time spent solving, while being able to solve more problems overall. As opposed to larger problems, parallelization does not improve the solving speed of smaller problems.

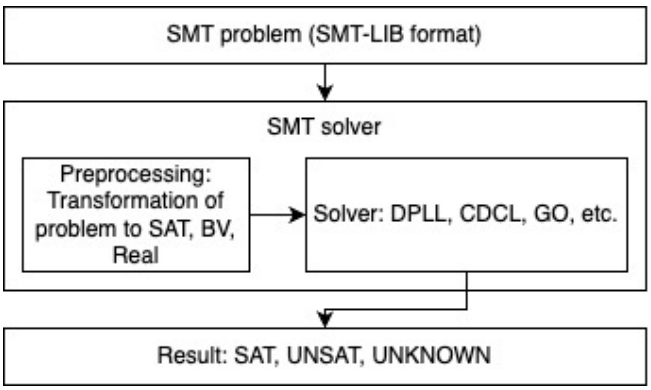
2 Background

This section presents the most relevant background theory and terminology regarding SMT solving, SMT solvers and floating-point numbers.

Satisfiable Modulo Theories (SMT) problems question whether a (mathematical) formula is satisfiable or not, that is, if there is an arrangement of values such that the said formula is true. An *SMT problem* can involve different data types such as real numbers, integers, strings or floating-point numbers, and their relative operators. An *SMT solver* (Figure 1) then tries to find solutions for these problems. The formula $x = y + 5$ could be an SMT problem of the integer data type, where an answer given by an SMT solver could be $x = 7$ and $y = 2$.

As there is no efficient solution (an algorithm in polynomial time) to many of these problems, some SMT solvers solve these problems by reducing them to a boolean satisfiability (SAT) problem [10; 3]. With this generalization, only an algorithm for a reduction to SAT is needed, which simplifies writing a general solver. This also means that this approach could be less efficient when writing a specialized solver that is focused on less data types, and that many SMT problems are more complex than SAT.

Figure 1: Architecture of an SMT solver



SMT problems can be of two types: sat / satisfiable or unsat / unsatisfiable. To prove that floating-point numbers are not associative, we can turn it into a satisfiable SMT problem by writing it in the SMT-LIB standard:

```
(declare-fun x () (_ FloatingPoint 11 53))
(declare-fun y () (_ FloatingPoint 11 53))
(declare-fun z () (_ FloatingPoint 11 53))
(assert ( not (=
    (fp.add RNE x (fp.add rm y z))
    (fp.add RNE (fp.add rm x y) z)
)))
(check-sat)
```

When solving this problem with an SMT solver, the solver will return sat. When we express an assertion that is clearly false, such as asserting that $x \neq x$, an SMT solver will return unsat:

```
(declare-fun x () (_ FloatingPoint 11 53))
(assert (not (= x x)))
(check-sat)
```

Both of these problems can be solved by different solvers relatively quick, but when problems become more complex (more assertions and variables), one solving strategy tends to become faster than the other, parallelism can find this better strategy.

By the means of standardizing SMT solving, standards have been made for specifying SMT equations and data types. The specification, known as SMT-LIB [2], also provides support for floating-point numbers, a so called *theory*. An SMT theory describes and defines the logical system and behaviour of a data type. The theory includes details such as: the functions a floating-point solver should support, the ways that these functions should behave and the construction of a floating point number, specifically that it is represented by a bit vector / bit array, which is used to represent a number using a significant and an exponent. SMT-LIB also manages SMT-COMP [21], a competition for SMT solvers, which includes large sets of SMT problems which are called benchmarks. These benchmarks are often useful for testing the performance of an SMT solver based on the accuracy of its solutions and its efficiency in terms of computation time and correctness.

2.1 Solvers

The most common approaches to solving floating-point problems include reducing the floating-point problem to either a bit vector problem (and from there to a SAT one) [17; 1; 6] or a real number problem [15; 8]. The first approach works in theory, as floating-point operations are composed of bit vector operations, but unfortunately this approach becomes inefficient really quick when the input equation grows in size. Trying to solve an equation by transforming it into a real number equation works sometimes, however it might return inaccurate results because floating-point arithmetic doesn't follow real number arithmetic. As an example, floating-point numbers are not associative, while real numbers are, so $x+(y+z)$ does not always equal $(x+y)+z$, these problems are mostly caused by the rounding after calculations on floating-point numbers [9].

Z3

The Z3 solver [6] is one of the largest and most general solvers. It supports almost all data types that are supported by the SMT-LIB theories, including floating-point. It has been proven to be an overall well performing solver, winning many first places in the SMT-COMP across the years, while also performing well in the QF_BVFP (Quantifier-Free Bit Vector and Floating Point) category, where solvers should support a mix of both quantifier free bit vector and floating-point problems. Z3 uses the DPLL(T) [10] algorithm to turn SMT problems of a type T into a SAT problem and solve it from there using the DPLL algorithm. Although the process of transforming a floating-point problem into a SAT problem is not documented, it becomes evident from the code[1] that Z3 turns the floating-point problem into an equivalent bit vector problem, which it solves using bit blasting. Bit blasting turns a bit vector formula into an equivalent SAT problem by taking every bit from from the bit vector variables and turning them into a propositional formula of the same form.

CVC5

The CVC5 solver [1] is a general purpose solver, similar to Z3. In the 2024 SMT-COMP it has won every category in the single query track[2], and has performed well in the floating-point arithmetic category. CVC5 implements the CDCL(T) [3] (Conflict Driven Clause Learning) algorithm for a data type T. CDCL is inspired by and similar to DPLL [3], but it implements non-chronological backjumping, meaning that it can go up multiple levels in its decision tree per iteration when it reaches a conflict. CVC5 implements a similar bit blasting technique[3] to Z3. It uses SymFPU [20] to translate floating-point operations an equivalent bit vector representation. After reducing to a bit vector problem, it uses bit blasting to reduce it to a SAT problem, from where it is solved using CDCL(T).

Bitwuzla

The Bitwuzla [17] solver is aimed at solving SMT problems regarding data types involving bits specifically, including floating-point problems. In the 2024 (as well as 2023) SMT-COMP, it has won all categories in the QF_FPA single query track[4]. Bitwuzla solves floating-point problems in two ways. It first reduces it to a bit vector problem using SymFPU, similar to CVC5. When solving a bit vector instance, Bitwuzla either resorts to the classic bit blasting strategy, or a propagation based local-search method [16].

GoSAT

The GoSAT [15] solver is an SMT solver that specifically and only supports floating-point problems. GoSAT has been

[1] Github: Z3Prover/z3 src/tactic/fpa/qffp_tactic.cpp

[2] https://smt-comp.github.io/2024/results/largest-contribution-single-query/

[3] Github: cvc5/cvc5 src/theory/fp/fp_word_blaster.cpp

[4] https://smt-comp.github.io/2024/results/qf_fparith-single-query/

shown to perform particularly well compared to other solvers on the griggio benchmark, a set of problems known to be specifically hard for conventional SMT solvers. GoSAT, as opposed to Z3, CVC5 and Bitwuzla, does not reduce a floating-point problem to a bit vector one, but instead uses global optimization to solve the problem by turning it into a minimization problem.

3 Method

This section presents a parallel solver that deploys different solvers over multiple CPU cores (Figure 2). Reasons to include different solvers will be presented, as well as possible limitations of the solver and what metrics will be gathered and why.

3.1 Running Solvers in Parallel

Architecture

The choice of language for the solver is python3.13.3. Python is suitable for its ease of multi-threading and wide support of libraries. All solvers that will be used have either a native library or binary executable available, and thus we are not limited by the relatively slow speed of Python itself.

Solvers

CVC5. We include CVC5 as it has proven to be a strong general solver. It has shown promising results in many of the most recent editions of SMT-COMP, in both overall results as well as floating-point specific results.

Bitwuzla. We will include Bitwuzla as it is one of the strongest bit and floating-point type specific solvers, as it has won the floating-point SMT-COMP tracks of the most recent years. Next to being a promising solver, Bitwuzla deploys a strategy that is different compared to the general bit blasting algorithm that utilizes CDCL or DPLL, allowing for possible positive / faster results where conventional solvers will perform worse.

GoSAT. We will include GoSAT as it has shown to be a good solver for problems where other conventional (bit blasting) solvers will perform significantly slower, such as the griggio benchmark [15]. As results for GoSAT have mainly been aimed at this benchmark in the past, it could also provide interesting insights when running GoSAT on other benchmarks.

Z3. We will include Z3 as it remains to be one of most well known general solvers. For this reason, Z3 has been included in various papers, which will help us compare and confirm our own results (Section 6). General solvers such as CVC5 have a very similar architecture to Z3, yet they have shown to be faster [20] on floating-point problems. Therefore, our parallel solver might not benefit as much from Z3 as it may from other solvers.

3.2 Benchmarks

We have taken the benchmarks from the 2018 SMT-COMP[5] to compare the parallel solver to other solvers. These benchmarks provide a diverse set of tests which have been used before to test different solvers [15; 20]. The QF_FP benchmark set is used, where QF stands for quantifier free, as quantifiers are known to pose problems regarding unsupported operators for some solvers[15]. The problems in these benchmarks are generated from / for software verification tools [19; 14] [6] [7], randomized problems involving common floating-point operators (wintersteiger benchmark) and manually crafted problems [11].

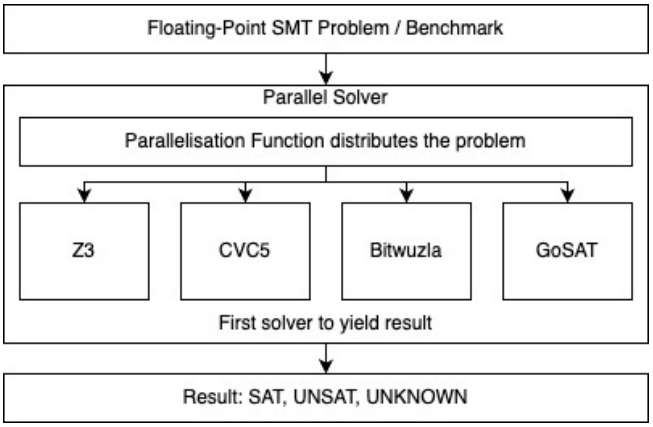
3.3 Measurements

We measure the performance of a solver by its speed. For every result yielded by the parallel solver, the (CPU) execution time will be measured, as well as the result obtained. From these metrics we can conclude the effect of the parallelization, and compare this to the performance of the individual solvers on the same problems.

3.4 Limitations

As GoSAT uses global optimization, it can never prove that a problem is unsatisfiable, as it does not propagate variables and reach conflicts. Because of this GoSAT returns unknown when a result is (assumed to be) unsatisfiable[8]. It is clear that this solver trades in absolute certainty / completeness for a (possibly) faster solving time.

Figure 2: Architecture of the parallel solver



4 Experimental Results

This section presents the experimental results from running the benchmarks on our parallel solver as well as other solvers for comparison. We present the results per benchmark in a graph where we compare the time spent per problem, as well as a bar plot with the contributions of each solver to the parallel solving.

[5] See: https://smt-comp.github.io/2018/benchmarks.htmlSMT-COMP 2018 benchmarks

[6] Vector: https://www.vector.com/int/en/products/products-a-z/software/vectorcast

[7] SPARK: https://www.adacore.com/about-spark

[8] See: ff

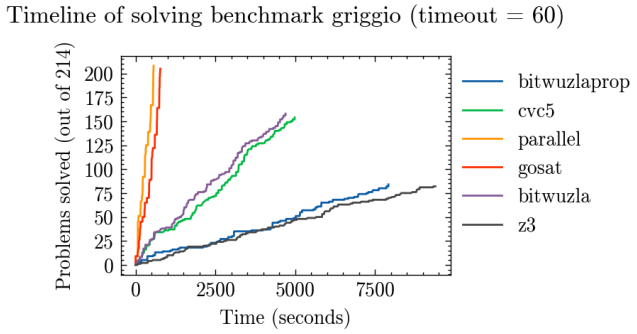
4.1 Configuration

All benchmarks have been ran on an 8 core M2 CPU, with 16gb of RAM available. Benchmarks have been run without any foreground processes running that could interfere with the CPU access of the solvers. The 8 cores ensure that all solvers can have their own core, which is also ensured by pythons multiprocessing module that spawns the processes. All solvers have a 60 second timeout / limit to solving a single problem, and share the same RAM with a size of 16GB.

4.2 Benchmarks

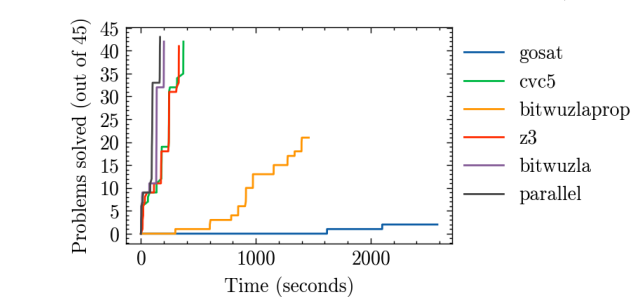
We show the results of the parallel solver next to those of the individual solvers in a plot that represents the amount of problems solved over time. From this graph we can see differences between how many problems each solver can solve, as well as how fast it solves these.

Figure 3: Results of the griggio benchmark.



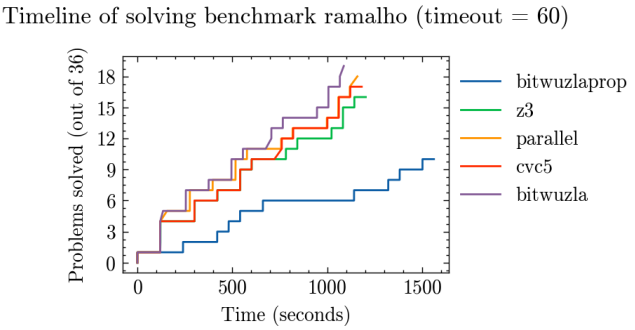
When comparing the individual solvers in the griggio benchmark (Figure 3), it can be seen that GoSAT has solved the most problems in the least amount of time. Bitwuzla and CVC5 show to be performing relatively similar. Looking at the parallel solver, it can be seen that it has solved more problems than any individual solver, as well as that it has done this in less time.

Figure 4: Results of the schanda benchmark.



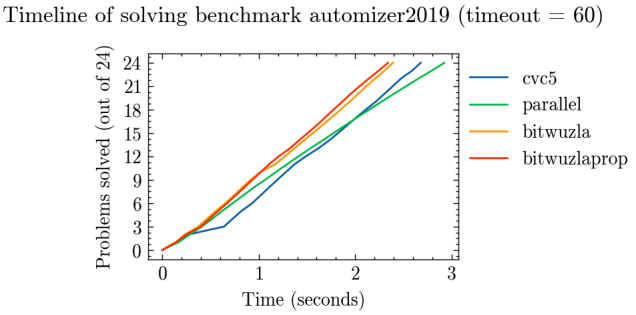
For the schanda benchmark (Figure 4), both CVC5, Z3 and Bitwuzla perform very well compared to the Bitwuzla propagation and GoSAT solver. The GoSAT solver does not perform well because it does not support most of the floating-point operations that are used in these benchmarks. The parallel solver has solved more problems than any of the individual solvers in less time.

Figure 5: Results of the ramalho benchmark.



For the ramalho benchmark (Figure 5), it can be seen that Bitwuzla solves the largest amount of problems. All solvers can be seen timing out on approximately half of the problems in the benchmark, including the parallel solver, which does not provide better results than Bitwuzla. GoSAT is not showed, as it does not support many of the operations that are used in these benchmarks and because of this has solved 0 problems.

Figure 6: Results of the automizer2019 benchmark. Z3 is not shown for clarity as it took 100x as long as the other solvers to solve the problems.



For both the automizer2019, vector and wintersteiger benchmark (Figure 6, Figure 7, Figure 8), it can be seen that all solvers have solved every problem in the benchmarks, with no significant differences between the times that the solvers take. The parallel solver starts solving with a similar trend, but it becomes slower once it has progressed further through the benchmark. GoSAT does not support the operations that are used in these benchmarks.

By looking at the the distribution of the amount of problems that are solved by each solver per benchmark (Figure 9), we can get a better insight into what solvers contribute to the solving of each benchmark. Bitwuzla shows to have solved most of the problems across all benchmarks.

Figure 7: Results of the vector benchmark.

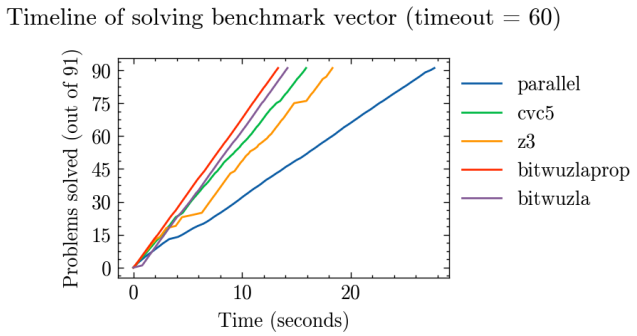
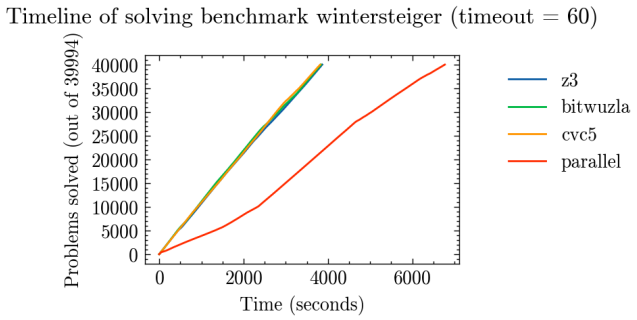


Figure 8: Results of the wintersteiger benchmark.

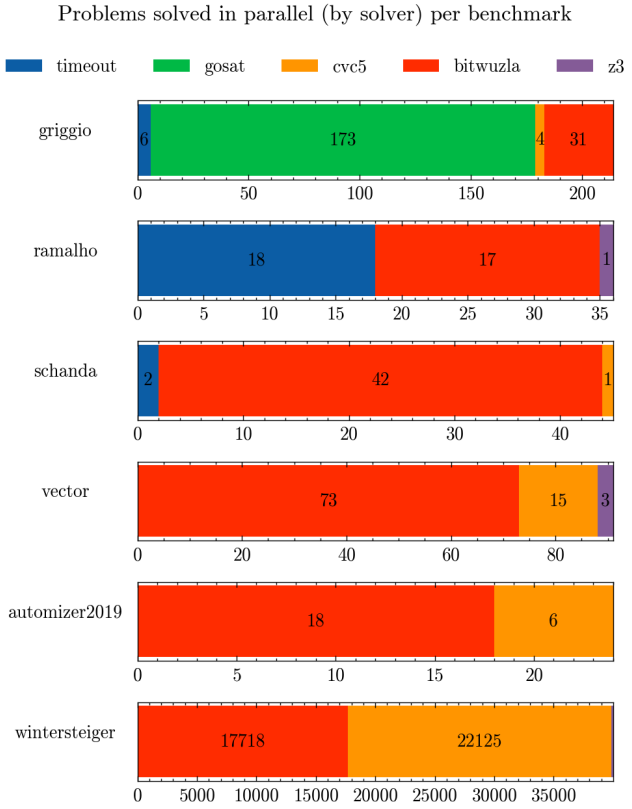


Although the majority of problems is solved by one solver for each benchmark, the solvers that solve the rest of the benchmarks still make a significant and surprising contribution. For the griggio benchmark it is clear that GoSAT is the most efficient solver, being 6x as efficient as Bitwuzla (Table 1). Yet, Bitwuzla manages to solve 14% of the problems, reducing the solving time from 13 minutes down to 10 minutes.

Looking at the table of the time spent solving per benchmark (Table 1), we can see that our parallel solver outperforms every individual solver, as should be theoretically possible. Bitwuzla and CVC5 show to be the overall most competitive solvers, where our solver is 1.14x to 1.35x faster, across all benchmarks. When we compare our solver to GoSAT it shows that our solver is 29.65x faster. This has no significance as the increased time in GoSAT's solving is caused by timeouts, as it is unable to support some floating-point operands, which is a known limit of GoSAT and prevents it from solving a large portion of problems. If we take into account these timeouts, and exclude the problems that GoSAT does not support in the calculation, we can see that our solver obtains a speedup of 1.37x. When comparing our solver to Z3, a speedup of of 3.52x can be seen. The parallel solver gives us a 2.30x speedup over the Bitwuzla propagation solver.

Because of their similar bit blasting tactic Z3, CVC5 and Bitwuzla also show similar behaviour in solving (Figure 3, Figure 4), where Z3 most of the times performs slightly worse than the others, which is possible because of the structure of

Figure 9: The distribution of problems solved for each solver. Timeouts are shown in blue. Numbers in the bars indicate the amount of problems that are solved by the corresponding solver. For contributions smaller than a hundredth of all problems, labels have been hidden.



the code itself and not the algorithm it uses, as discussed by other papers [20; 4].

While running different problems from the wintersteiger benchmark with the Bitwuzla propagation solver, we encountered multiple timeouts, from which we estimated that the entire benchmark would take multiple days to complete, as it contains 39994 problems. We concluded that it was infeasible to run the entire benchmark, the results are marked with an x.

5 Responsible Research

The proper working of SMT solvers is of high importance, as this is what the essence of software verification is about. As we have not introduced a new algorithm for solving itself, the accuracy and integrity of our portfolio solver depends directly on the solvers that we have used, considering our code is correct. Our work is publicly available[9] to provide this integrity as well as reproducibility. The solvers that are used are all publicly available [15; 1; 6; 17], as well as the data that we have used to verify our solver [21]. To obtain fair results we have ran our experiments in

[9]Repository: https://github.com/TrizlyBear/parallelsmt

Table 1: The time every solver took to solve all problems of a benchmark. Numbers that are bold show the fastest solver. Crossed out numbers indicate that the solver does not support (some) operations in the benchmark. Speedup times are proportional to the parallel solver, formatted as the speedup time *with* / *without* solvers that timeout because of unsupported operations. The speedup column shows the speedup of our solver compared to the individual solvers across all benchmarks, whereas the column shows the speedup of our solver for a single benchmark compared to all solvers.

Model	griggio	schanda	ramalho	vector	automizer2019	wintersteiger	Speedup
bitwuzlaprop	7923.54s	1461.23s	1561.95s	**13.32s**	**2.34s**	x	2.30/2.30
cvc5	4985.38s	370.45s	1181.67s	15.90s	2.68s	**3817.71s**	1.35/1.35
gosat	775.48s	~~2580.75s~~	~~2160.00s~~	~~5460.00s~~	~~1440.00s~~	~~1198390.46s~~	29.65/1.37
bitwuzla	4697.98s	200.75s	**1087.13s**	14.19s	2.39s	3839.68s	1.14/1.14
z3	9394.44s	330.98s	1203.06s	18.33s	427.09s	3864.78s	3.52/3.52
parallel	**564.73s**	**166.95s**	1158.85s	27.71s	2.92s	6770.50s	1.00/1.00
Speedup	7.49/7.49	3.72/2.61	1.20/1.08	1.79/0.55	8.45/3.06	2.38/0.57	

equal environments, with minimal interference of other processes.

6 Related Work

This section presents relevant research to parallelization and other benchmark results of the relevant solvers. We present an overview of parallelization and its benefits compared to running SMT solvers sequentially, as well as comparisons of relevant benchmark results compared to ours.

6.1 Parallelization

Parallelization has already been shown to have a positive effect on SMT solving. Using the Z3 solver it has been shown [12] that an overall speedup of 1.06 can be achieved, but when looking at harder problems that can not be solved under a minute, the average speedup is 3.2. This shows that parallelization becomes significantly more useful when solving harder problems, our findings reach the same conclusion. The strategy used to achieve this speedup is lemma sharing. When an instance of a solver finds a lemma, it adds this lemma to a queue that is available to all solver instances, from where other solvers can use any new lemmas to specify their search domain further.

ManySAT [13], a parallel SAT solver, has also shown to make significant improvements (compared to sequential solvers) by running solvers in parallel. The solver deploys the same (CDCL based) solver with different configuration on different cores.

6.2 Benchmarks

The results of this study confirm different performance related findings from previous studies. We can confirm the fast behaviour of GoSAT on the griggio benchmark compared to a slow performance on other benchmarks that the SymFPU paper has shown before [20]. Our results of the Bitwuzla, CVC5 and Z3 solver also show similar times compared to the experiments results shown in the Bitwuzla system description [17]. Even though it is not mentioned in the paper whether Bitwuzla used a bit-blasting or propagation-based approach, we believe that it uses the bit-blasting approach as this has shown to be the most effective method of the two (Table 1). We have not found any benchmark results of the propagation-based approach [16] in other studies.

7 Conclusion and Future Work

We have addressed the workings, issues and state-of-the-art solvers of floating-point SMT problems (Section 2) from which we have constructed a parallel / portfolio solver (Section 3). From our experiments we have concluded that parallelization provides promising results and optimizations when it comes to solving floating-point SMT problems. Across all experiments 4 our solver performs better than any individual solver, with speedups between 1.14x and 1.37x over the most competitive solvers and a speedup of 2.30x+ over other solvers. Our solver achieved speedups of 2.61x+ on problems that are not easily solvable (over 2+ seconds), contrasted by the speed decrease that our solver shows on problems that are easily solvable. Our solver has also shown that solvers which do not stand out in the overall results of a benchmark, might still be the best performing for some specific problems, reducing overall solving time when using parallelization.

While our parallelization of some promising solvers has shown positive results, there are other promising solvers such as Boolector [18], MathSAT [5] and Yices [7]. Comparing a parallel solver to these solvers as well as including them in a parallel solver could provide more significant insights, faster results to problems, and an even more diverse range of algorithms that can be applied. GoSAT [15] has shown to be one of the fastest solvers (Section 1), yet it can not solve many problems because of its lack of support for some floating-point operations and constructors. An implementation of these features would make it possible to solve a significantly larger subset of problems.

References

[1] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar.

cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.

[3] Roberto J. Bayardo and Robert C. Schrag. Using csp look-back techniques to solve real-world sat instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, page 203–208. AAAI Press, 1997.

[4] Martin Brain and Marina De Vos. The significance of memory costs in answer set solver implementation. *Journal of Logic and Computation*, 19(4):615–641, 09 2008.

[5] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[6] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In Jakob Ramakrishnan C. R. and Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, 2008.

[7] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[8] Zhoulai Fu and Zhendong Su. XSat: A Fast Floating-Point Satisfiability Solver. In *Proceedings of the 28th International Conference on Computer Aided Verification, Part II*, pages 187–209. Springer, 2016.

[9] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23:5 – 48, 1991. Cited by: 1183.

[10] George Hagen, Robert Nieuwenhuis, Albert Oliveras, Cesare Tinelli, and Harald Ganzinger. Dpll(t): Fast decision procedures. In Doron A Alur Rajeev and Peled, editors, *Computer Aided Verification*, pages 175–188. Springer Berlin Heidelberg, 2004.

[11] Leopold Haller, Alberto Griggio, Martin Brain, and Daniel Kroening. Deciding floating-point logic with systematic abstraction. In *2012 Formal Methods in Computer-Aided Design (FMCAD)*, pages 131–140, 2012.

[12] Youssef Hamadi, Leonardo de Moura, and Christoph M. Wintersteiger. A concurrent portfolio approach to smt solving. In Oded Bouajjani Ahmed and Maler, editors, *Computer Aided Verification*, pages 715–720. Springer Berlin Heidelberg, 2009.

[13] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modelling and Computation*, 6:245–262, 2010.

[14] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. Ultimate automizer and the search for perfect interpolants. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 447–451, Cham, 2018. Springer International Publishing.

[15] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. gosat: Floating-point satisfiability as global optimization. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 11–14, 2017.

[16] Aina Niemetz and Mathias Preiner. Ternary propagation-based local search for more bit-precise reasoning, 2020.

[17] Aina Niemetz and Mathias Preiner. Bitwuzla. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965, pages 3–17. Springer, 2023.

[18] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *J. Satisf. Boolean Model. Comput.*, 9(1):53–58, 2014.

[19] Mikhail Ramalho, Felipe R. Monteiro, Jeremy Morse, Lucas Cordeiro, Bernd Fischer, and Denis Nicole. Esbmc 5.0: An industrial-strength c model checker. In *33rd IEEE/ACM International Conference on Automated Software Engineering*, 2018.

[20] Florian Schanda, Youcheng Sun, and Martin Brain. Building better bit-blasting for floating-point problems. In Lijun Vojnar Tomáš and Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 79–98. Springer International Publishing, 2019.

[21] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. The SMT competition 2015-2018. *J. Satisf. Boolean Model. Comput.*, 11(1):221–259, 2019.