DEGREE PROJECT, IN COMPUTER SCIENCE , FIRST LEVEL
*STOCKHOLM, SWEDEN 2015*

# Factoring integers with parallel SAT solvers

## AN EFFICIENCY AND EFFECTIVENESS EVALUATION OF PARALLEL VERSUS SEQUENTIAL SAT SOLVERS FOR INTEGER FACTORIZATION.

ERIK F, DANIEL L

**KTH ROYAL INSTITUTE OF TECHNOLOGY**

**CSC SCHOOL**


KTH Computer Science
and Communication

# Factoring integers with parallel SAT solvers

An efficiency and effectiveness evaluation of parallel versus sequential SAT solvers for integer factorization.

Erik Forsblom
Daniel Lundén

Degree Project in Computer Science, DD143X
Supervisor: Per Austrin
Examiner: Örjan Ekeberg

CSC, KTH May 7, 2015

**Abstract**

Factoring integers is a well known problem that at present cannot be solved in polynomial time. Therefore, other approaches for solving factorization problems are of interest. One such approach is to reduce factorization to SAT and solve it with a dedicated SAT solver. In this study, parallel SAT solvers are examined in this context and evaluated in terms of speedup, efficiency and effectiveness versus sequential SAT solvers. The methodology used was an experimental approach where different parallel and sequential solvers were benchmarked on different reductions from integer factorization to SAT. The benchmarks concluded that parallel SAT solvers are not better suited for solving factorization problems than sequential solvers. The performance boosts achieved by the fastest parallel solver barely corresponded to the extra amount of available parallel resources over the fastest sequential solver.

**Sammanfattning**

Att faktorisera heltal är ett välkänt problem som för närvarande inte kan lösas i polynomisk tid. Därför är andra tillvägagångssätt för att lösa faktorisering av intresse. Ett sådant tillvägagångssätt är att reducera faktorisering till SAT och sedan lösa problemet med en dedikerad SAT-lösare. I denna studie undersöks parallella SAT-lösare i detta sammanhang och utvärderas i förhållande till uppsnabbning, effektivitet och ändamålsenlighet jämfört med sekventiella SAT-lösare. Den metod som användes var en experimentell sådan där olika parallella och sekventiella lösare jämfördes på olika reduktioner från heltalsfaktorisering till SAT. Genom testerna erhölls slutsatsen att parallella SAT-lösare inte är bättre lämpade för att lösa heltalsfaktorisering än sekventiella lösare. Prestandavinsterna som uppnåddes av den snabbaste parallella lösaren motsvarade knappt den extra mängd parallella resurser som denna hade över den snabbaste sekventiella lösaren.

# Contents

# Chapter 1

# Introduction

The Integer factorization problem (FACT) is a well known computational problem for which no polynomial time algorithm is yet known. This fact is exploited extensively in cryptography to provide secure communication of data. For example, in the popular cryptosystem RSA, this is achieved through encryption using the product of two large prime numbers. To decrypt a message, knowledge of these prime numbers is required. If only their product is known – as is the case if a malicious user tries to decrypt an encrypted message – it is believed to be computationally infeasible to find the prime factors given that they are sufficiently large, rendering the communication secure. Currently, specialized sub-exponential time algorithms such as the general number field sieve are being used to factorize integers. These algorithms are however not even remotely capable of breaking cryptosystems such as RSA due to their sub-exponential time complexity and the size of numbers used [1].

This leads to another well known computational problem – the Boolean satisfiability problem (SAT). It was the first computational problem proved to be NP-complete in 1971 by Stephen Cook [2] and many other well known problems have in turn been proved to be NP-complete through reductions to SAT. Due to its age and status as a fundamental problem in computer science, algorithms for solving SAT have been improved substantially over the years. Because FACT is known to be in NP and SAT is known to be NP-complete, there exist reductions from the former to the latter. One such possible reduction is simple boolean circuit reductions enabling the use of SAT solvers for problem instances generated by FACT. This leads to the thematic question for this study, which is whether it is possible, by reducing FACT to SAT, to solve FACT fast enough with a SAT solver to be deemed an improvement over specialized algorithms for the problem.

Previous studies [3, 4, 5] show that this approach for solving FACT cannot yet compete with specialized FACT algorithms such as the general number field sieve mentioned above. The most central aim of this study is therefore not to directly compare SAT solvers to FACT algorithms. Instead, the focus will lie on evaluating parallel SAT solvers on the specific problem instances generated

by reductions from FACT. The evaluation will be made with respect to the performance of sequential SAT solvers and the very simple FACT algorithm trial division as a reference point.

## 1.1 Problem Statement

The formal problem statements for this study follow below. They are presented in descending order of importance.

- Are parallel SAT solvers better suited for solving problem instances generated by FACT than sequential SAT solvers?
- What solving technique(s) used for parallel SAT solvers has the biggest performance impact for solving SAT instances generated by reductions from FACT?
- Are there any reduction technique(s) from FACT to SAT that are better suited when solving problem instances with parallel SAT solvers?
- Can parallel SAT solvers compete with standard trial division for solving FACT?

The last statement is included entirely for good measure and for providing a reference point to the parallel solvers in the study. It is already established [5] that it is not possible for solvers to compete with trial division for the relatively small numbers used here (See delimitations below).

## 1.2 Purpose

The research topics presented above are of interest due to the fact that it may be the case that SAT instances generated by reductions from FACT are particularly well suited for being solved with parallel SAT solvers. This is hinted at in a report by Hamadi and Wintersteiger from 2013 [6] and a presentation by Srebrny in 2004 [7]. The indicative measurements of speedup, efficiency and effectiveness [6] used in this study are the tools used for determining if current parallel solvers are in fact more suited for integer factorization than current sequential solvers. Hopefully, the results of this study will enable further research in this specific area by individuals that possess more in-depth knowledge of SAT solvers. For example, an optimal future result would be the development of new parallel SAT solvers performing competitively against specialized algorithms for FACT. An estimation by Hamadi and Wintersteiger suggests that designing and implementing such a specialized SAT solver for FACT would require about ten years of research [6].

## 1.3 Delimitations

Due to the nature of the available reductions from FACT to SAT, the problem instances used for the experiments of this study will consist of composite numbers being a product of exactly two primes of equal bit length (explained further in background section). Technically, solving FACT where composite numbers are built by more than two primes are also desirable, but the most practical application of FACT is breaking common cryptosystems where the composite numbers are of the form described above (product of two primes).

Since algorithms with sub-exponential worst case performance will be examined in this study, extra caution will have to be taken as to not exaggerate the size of the problem instances. As is written in the method section, the maximum problem instance size chosen for the integers used was 32-bits. This size limit was mainly acquired by own experiments prior to the execution of the method, but also from examining a similar recent study [5].

The development of SAT solvers is such a large research topic that if a specific solver for this study were to be developed either by implementing solvers from scratch or by tweaking existing solvers, it could not hope to rival already existing solvers. Therefore, this study will only use common publicly available solvers with default parameter settings.

## 1.4 Outline of report

In the the background chapter, relevant background information for the problem at hand can be found. Here, preliminaries and required definitions for the study will be described alongside a description of modern approaches for solving SAT instances. This provides all necessary background information for the reader to fully understand the contents of the thesis.

Following this, the method chapter provides a detailed description of the execution of the study as well as justifications for the choice of methodology.

Next, the results chapter presents the data obtained during testing.

Following this, the results are discussed with regard to the methodology used, as well as potential error sources during the experimentation.

Finally, conclusions based on the result and discussion are presented, and the problem statements are dealt with.

# Chapter 2

# Background

In this section, a brief overview of the fundamentals relevant for this study will first be given. This will be followed by a background for both sequential and parallel SAT solving. Lastly, relevant performance measurements used will be explained in detail.

## 2.1 Integer factorization problem (FACT)

By the fundamental theorem of arithmetic, every integer $n > 1$ is either prime itself or a unique product of primes of arbitrary powers.

$$n = p_1^{\alpha_1} p_2^{\alpha_2} p_3^{\alpha_3} \dots p_n^{\alpha_n} = \prod_{i=1}^{n} p_i^{\alpha_i} \tag{2.1}$$

Integers that are the product of two primes are called semiprimes. These integers are used as problem instances for factorization in this study.

Current algorithms for finding the factors of an arbitrary integer take, in the worst case, sub-exponential time in the number of bits for the integer being factored. Thus, for very large numbers, finding the factors will take a very long time. This is, as mentioned in the introduction, the foundation in many cryptosystems where the security of communication is guaranteed based on this simple principle.

The most trivial approach for factoring integers is by trial division. Simply divide the number to be factored by all numbers greater than 1 and less than or equal to the square root of the number[1]. When the remainder of the division evaluates to zero, a factor has been found. Remove the factor from the number and iteratively repeat the above to find the remaining factors.

State-of-the-art software for factoring integers use a combination of different algorithms, but the common denominator for factoring large integers is the general number field sieve. The general number field sieve is the major algorithm

[1] If composite, a number $n$ always has a factor less than or equal to $\sqrt{n}$.

used to factor the largest integers, such as the ones seen in the RSA factoring challenges [1]. The execution process of the algorithm will not be covered here as it is not used for this study.

## 2.2 Boolean satisfiability problem (SAT)

The boolean satisfiability problem was the first problem proven to be NP-complete. The problem is, given a formula expressed in propositional calculus, to determine if there exists an assignment of variables for the formula such that the formula as a whole evaluates to true. For example, the below formula expressed in propositional calculus is satisfiable since the assignment $a = \text{true}, b = \text{true}, c = \text{false}$ evaluates the whole formula to being true.

$$(a \vee b) \rightarrow (c \wedge \neg a) \vee b \tag{2.2}$$

Variations for this problem exists, restricting the usage of propositional calculus in the formula. The most common variation, which will be used in this study, is the problem of finding an assignment for a formula expressed in Conjunctive normal form (CNF). CNF is, as above, a formula expressed in propositional calculus but with the restriction that it has to be a conjunction of clauses. A clause is a disjunction of literals, and a literal can either be a variable or the negation of a variable. One important property of CNF is that every propositional formula can be converted to an equivalent CNF formula. Another important property is the plain structure of the CNF formulas, which lead to easier implementation of algorithms taking propositional formulas as input. In the cause of this study, these algorithms are the SAT solvers. The following formula is an example of a SAT formula expressed in CNF.

$$(\neg a \vee b \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \tag{2.3}$$

## 2.3 Reductions from FACT to SAT

The most basic reduction from FACT to SAT is based on the elementary school multiplication algorithm, known as long multiplication. The product $n$ of two given numbers $p$ and $q$ can easily be calculated with this method in polynomial time. If the multiplication is carried out in base 2, it is possible to express the multiplication as a formula in propositional calculus, where the bits of the numbers are encoded as variables in the formula. This is done by modelling binary multipliers in propositional calculus. Binary multipliers are used in the arithmetic logic unit of modern computers for multiplication of integers. A starter friendly description of how one can perform this reduction and obtain a propositional formula in CNF for solving an instance of FACT has been written by Mateusz Srebrny [7].

Through the propositional formula obtained, given a known product $n$, finding the factors of $n$ now simply reduces to plugging the bits of $n$ into the propo-

sitional formula and finding an assignment of the bits for $p$ and $q$ so that the formula is satisfied. If one such assignment can be found, the assignment variables represents the factors $p$ and $q$ of $n$. Necessary restrictions are of course needed for practical application. For example, every number has a factor 1 which is not an acceptable solution for the problem.

The SAT instances obtained by the above method can be considered "hard", especially if $p$ and $q$ are both prime and are of equal bit length (as is the case in this study). This is a result of the presumed hardness of FACT itself, but it also follows intuitively since the solution space is very sparse. In fact, only a single solution exists if $p$ and $q$ are prime.

## 2.4 Solving SAT sequentially

Solving SAT is a large research topic, well beyond the scope of this study. Still, certain techniques are more commonly used than others. In this section, the most common approaches for sequential SAT solving will be described very briefly.

### 2.4.1 DPLL algorithm

The Davis-Putnam-Logemann-Loveland (DPLL) [8] algorithm is a chronological backtracking search algorithm introduced in 1962 for solving the satisfiability problem, where the propositional logic formula is in CNF. It works by initially choosing – or "guessing" – a value for a literal, and then finding all *unit clauses* that resulted from assigning the given boolean value to the chosen literal. A unit clause is a clause where exactly one literal in the clause is still unassigned [8]. For each found unit clause, the last unassigned literal is assigned such that the clause evaluates to true. If the resulting assignment of literals cannot satisfy all clauses, the algorithm will backtrack recursively and try a different assignment. The algorithm terminates either when all clauses are satisfied, i.e. when a solution has been found, or when the algorithm has tried all assignment combinations without finding a solution.

Most modern SAT solvers are still based on this over 50 years old algorithm [9].

### 2.4.2 Conflict-Driven Clause Learning

Building on the DPLL algorithm, Conflict-Driven Clause Learning (CDCL) is a more modern and effective approach to solving the SAT problem [10]. CDCL is a substantially more complex method than DPLL, which besides just implementing the basic DPLL functionality also includes techniques such as learning new clauses from conflicts during backtracking, exploiting the structure of conflicts during clause learning, using lazy data structures for formula representation,

ensuring branching heuristics have low overhead[2], periodically restarting backtrack searches and several others [10].

There exists a vast amount of CDCL SAT solvers, where the main difference lies in the usage and parameters of the above mentioned techniques.

### 2.4.3 Look-ahead

Look-ahead is another technique commonly used by SAT solvers. It consists of a basic DPLL algorithm with the addition of a more complicated *look-ahead procedure* that is used to determine the most effective choice of the next branching variable in the DPLL algorithm [11]. This effectiveness is determined by evaluating the result of the look-ahead procedure on different variables. The evaluation is performed with a *look-ahead evaluation function* [11], whose implementation varies between solvers.

### 2.4.4 Stochastic Local Search

The general concept of stochastic local search can be described as follows [12]: for a given instance of a combinatorial problem, the local search process is started by selecting an initial candidate solution (often randomly) from the search space. Following this initialization, the process then iteratively moves to neighboring candidate solutions, where the decision in each iteration is based on a limited amount of local information. In stochastic local search, both decisions and initialization can be randomized.

## 2.5 Solving SAT in parallel

Parallel SAT solvers have emerged, as many other parallel algorithms, due to the thermal wall being hit for development of sequential hardware [6]. Because of this, it has become a necessity to develop algorithms for parallel hardware (i.e. multi core CPUs) to achieve faster run times. Compared to sequential algorithms, parallel algorithms have to take additional problematic factors into account such as load balancing, non determinism and effective communication between the parallel units executing the algorithm. Analogous with sequential solvers, parallel solving of SAT is naturally an equally large research topic outside the scope of this study. Therefore, only an overview will be presented here.

### 2.5.1 Divide and conquer

The very first implementations of parallel SAT solvers took a divide and conquer approach, splitting the search space into smaller subspaces that could be assigned to parallel working units. This intuitively seems like the natural choice

[2]Computational overhead is the excess use of resources during computation time, such as invoking functions or methods, which causes the setup of a stack frame.

for a parallel SAT algorithm based on CDCL/DPLL, since the sequential version of the algorithm is indeed traversing a tree of possible variable assignments. However, in recent years portfolio-based parallel SAT solvers (described next) have become prominent.

### 2.5.2 Portfolio

Hamadi and Winterstiger [6] mention in their report from 2013 that the most recently developed parallel SAT solver with a divide and conquer approach was developed in 2008. A long time ago considering the development rate of SAT solvers. The phasing out of these solvers was not accomplished because they were inherently bad. Instead, portfolio-based solvers directly utilize the characteristics of sequential solvers in a way that divide and conquer solvers cannot. This fact combined with tricky implementation details regarding load balancing for divide and conquer algorithms have lead to portfolio-based solvers becoming easier to implement and optimize, and therefore more popular.

Current state-of-the-art parallel SAT solvers all use variations of the portfolio approach. The basic functionality of portfolio-based solvers utilize one sequential solver on each available working unit. The speedup is achieved by differentiating the individual sequential solvers on their parameters [6], or even using altogether different type of solvers. This enables the different solvers to act in a complementary manner, which results in the individual solvers competing for solving the problem instance. Sharing of learned clauses between the different solvers is also an important technique that is frequently used.

The rise of portfolio solvers began with an observation regarding the technique of restarts for sequential solvers. Researchers noticed that frequent restarts of sequential solvers led to performance improvements [13]. Since restarts improved performance, starting many solvers in parallel would doubtlessly also improve performance. Hence, portfolio solvers were implemented to take full advantage of these noticed performance gains.

## 2.6 SAT performance for solving FACT

Several earlier studies [3, 4, 5] have shown that approaching FACT by first reducing it to SAT and then utilizing existing (sequential) SAT solvers appears to be substantially inferior to simply approaching the problem with standard FACT algorithms.

Schoenmackers and Cavender [3] demonstrated this inferiority in 2004 through experimentation by implementing reductions from a delimited version of FACT to SAT and converting the resulting boolean formula to conjunctive normal form before using the SAT solver zChaff to solve the CNF-SAT instance. With their specific test instances, they concluded that a brute force FACT algorithm was in fact faster than the reduction and SAT solving approach.

A previous bachelor thesis [4] from 2014 hints at the same result by comparing different reductions from FACT to CNF-SAT and solving with the SAT

solver MiniSat – however, it was also noted that the choice of full adder design used in the reduction may have a sizable impact on the runtime.

Another previous bachelor thesis [5] from 2014 found that FACT reductions to SAT followed by the application of SAT solvers was slower than the application of actual FACT solvers for the generalized FACT problem.

## 2.7 Speedup, efficiency and effectiveness

When comparing the performance of a parallel SAT solver to a sequential SAT solver, Hamadi and Wintersteiger [6] present three measurements. Firstly, the most obvious measurement, known as effectiveness, is simply the time required for the SAT solver to solve a problem.

Next, the speedup $S$ of a parallel solver that runs in time $T_p$ over a sequential solver that runs in time $T_s$ is defined as follows.

$$S = \frac{T_s}{T_p} \tag{2.4}$$

This fraction often does not provide an indicative measure of performance, because it does not account for the additional resources available to the parallel SAT solver. Therefore the run-time efficiency of a parallel solver is considered, where $E$ is the efficiency of the solver and $r$ represents the number of resources available to the solver. The efficiency is relevant because a very efficient solver may often be preferable to an inefficient solver, even if the speedup of the efficient solver is somewhat lower than that of the inefficient solver. The efficiency is defined as follows.

$$E = \frac{S}{r} = \frac{T_s}{r \cdot T_p} \tag{2.5}$$

As for the availability of resources, $r$, Hamadi and Wintersteiger [6] suggest that it can be computed in a calibration test, prior to measuring the main experiment. Assume that the hardware on which the testing is performed has $n$ cores. Then $n$ copies of a sequential SAT solver can be run simultaneously, independent of each other. Let $T_{ns}$ be the observed runtime that is required for them to complete their computation. Then $r$, the resource availability to a parallel solver, may be calculated as follows.

$$r = n \cdot \frac{T_s}{T_{ns}} \tag{2.6}$$

For instance, when running a single sequential SAT solver on a single core machine (that is, $n = 1$), $T_s$ would equal $T_{ns}$ since only one core is available and no parallelism is possible. Thus the value $r$ would equal 1. If instead four sequential SAT solvers were run on a quad-core machine ($n = 4$) – and say the calibration experiment still resulted in $T_s = T_{ns}$ – this would result in $r = 4$, meaning that the machine was able to perfectly utilize its parallel resources. Generally, as $n$ increases, so does the value $T_{ns}$ (due to all cores sharing the

same memory) making it greater than $T_s$. Thus the value $r$ will usually be slightly lower than the number of cores $n$.

# Chapter 3

# Method

The problem statements introduced previously were explored by running benchmarks where different parallel SAT solvers were tested and compared to sequential SAT solvers and FACT algorithms based on the same input. The input for the SAT solvers were problem instances generated by different available reductions from FACT.

As such, this study uses an experimental approach rather than a theoretical one [14]. The main reason for this is the relative infancy of the problem context (factoring integers with SAT), which can be seen by the lack of research papers on the subject. Another reason is the purely general knowledge base, which lacks a detailed understanding of modern SAT solving, that the writers have attained for the subject during the short interval of time in which this study was performed. In fact, even with a deeper understanding of the implementation of SAT algorithms, a theoretical approach to the problem statement would still be an extremely hard and complicated undertaking due to the heuristic nature and exponential worst case time complexity of SAT algorithms.

By using an experimental approach, the main benefit is that the research process is simplified greatly. The disadvantage with this approach, as is the case for experimental approaches in general, is also quite obvious - the reliability of the results have to be questioned in much greater detail.

## 3.1 Testing environment

For reproducibility purposes, here follows the relevant specifications for the machine on which the experiments were run.

**Operating System** Ubuntu 14.04 LTS, 64-bit

**CPU** Intel Core i5-2500K CPU @ 3.30GHz x 4

**Memory** 2x 4096 MB DDR3, 1600 MHz

**Motherboard** ASUS P8P67 PRO

## 3.2 Problem instance generation

### 3.2.1 Generating semiprimes

For the purposes of this project, SAT problem instances had to meet certain requirements in order to be potential test instances. As mentioned in the background section, the hardest problem instances are generally semiprimes where the factors are of equal or similar bit length. Therefore all SAT instances used in this project are the results of reductions from FACT using these specific types of semiprimes.

The problem instances for this study were generated by first finding all prime numbers having a bit length lower than 16. This was done simply by using the sieve of Eratosthenes. After this step, tough semiprimes were generated by multiplying primes of equal bit length. The amount of semiprimes generated for each bit length was set to 100, and the semiprimes generated had a bit length between 17 and 32. This produced a grand total of 1600 semiprimes. The problem instances produced were also sorted by size. The algorithm itself was programmed in C++ and can be found in appendix A.3.

The reason for using input of the above specified bit length as a maximum input size was to limit the required time for running multiple SAT solvers on a large sample of inputs for several different bit lengths. This is also mentioned in the Delimitations section found in the Introduction chapter.

### 3.2.2 Reductions to CNF instances

The generation of corresponding SAT instances for the semiprimes were performed using two different reduction tools, in order to ensure that the results were not be subject to a specific bias or tendency of one certain reduction from FACT to SAT. The tools used are Tough SAT Project [15] and CNF Generator for Factoring Problems [16], both of which enable generation of SAT instances using boolean circuit reductions based on an integer to be factored. The CNF Generator for Factoring Problems allows several different variations in reductions – all of which have been used when generating SAT instances. It was not clear what kind of formula simplifications were made regarding the two tools used, so additional measures was required. To render the reductions unbiased with respect to simplification, a popular SAT simplifier/preprocessor named SatELite [17] was used. Because of this, a total of 14 reductions were generated:

- N-Bit Carry-save (CNF Generator)
- N-Bit Wallace (CNF Generator)
- N-Bit Recursive (CNF Generator)
- Fast Carry-save (CNF Generator)
- Fast Wallace (CNF Generator)

- Fast Recursive (CNF Generator)
- Tough SAT (Tough SAT project)
- Simplified versions of the above

The naming of the 6 first reductions above refers to the type of adder used for the multiplication circuit and the multiplication circuit itself. The first part is the adder type, and the second is the multiplicator type. More information regarding the different reductions can be found on the respective web pages for the tools used.

The generation itself was carried out using Haskell (CNF Generator) and Python (ToughSAT) source code freely available from respective web pages [15, 16]. Both tools generate CNF in the DIMACS format [18]. DIMACS is the de-facto standard input format for most SAT solvers. The shell script demonstrating how these tools were used is available in appendix A.4.

## 3.3 Problem instance solving

### 3.3.1 Precautions

All user started active programs were terminated and the network connection was disabled before running any tests. This was to ensure that no user programs or network communications stole computing resources from the solvers, which could result in biased results.

### 3.3.2 Assumptions

It was assumed that all of the solvers and algorithms were correctly implemented, as in that they would never return an incorrect result under normal execution conditions. In other words, since all of the problem instances used were satisfiable, it was assumed that if a solver had finished running, it had successfully solved the problem instance.

It was also assumed that all the used reductions were in fact correct reductions that only had one possible satisfying variable assignment that corresponded to the true solution.

Based on these assumptions, no verification that the solver output did indeed conform to the sought factors were performed.

### 3.3.3 Calibration test

To get correct values for the efficiency of a parallel solver, an estimated available resource value $r$ was required (See background section). The estimation was performed by running a simple bash script that measured the time it took ($T_s$) for MiniSat to solve a single problem instance of suitable difficulty, and the time it took ($T_{4s}$) to solve four (the number of available CPU cores, see testing

environment) instances of the same problem in parallel. The available resources $r$ was then calculated as $r = 4 \cdot (T_s/T_{4s})$. The script is available in appendix A.2.

### 3.3.4 Solvers and algorithms

A number of SAT solvers and one FACT algorithm were examined. This was, as with the different reductions, to ensure unbiased results.

#### Sequential solvers

- MiniSat 2.2.0 [19]
- Lingeling ayv-86bf266-140429 [20]

MiniSat was chosen since it is a popular and accessible sequential CDCL solver. It is also used as a basis for many other well known solvers. For example, the most prominent portfolio-based parallel SAT solver ManySAT below is based on MiniSat.

Lingeling is another popular CDCL solver [21, 22, 23, 24, 25] recognized from its participation in the SAT competitions [26] and it is therefore included here. Another reason for its inclusion is because of its affinity with the parallel solvers Plingeling and Treengeling which are included below.

#### Parallel solvers

- ManySAT 2.0 [27]
- Plingeling ayv-86bf266-140429 [20]
- Treengeling ayv-86bf266-140429 [20]

ManySAT is a popular parallel SAT solver. It is a standard portfolio solver which smart differentiation regarding restart policies, clause learning, clause sharing and branching heuristics for the sequential solvers. ManySAT is designed to run on up to 4 CPU cores and as the creators of the solver mentions in their initial technical report [28], super-linear speedups[1] are indeed achieved on average when compared to the top performing sequential solvers [28].

Plingeling and Treengeling are the parallel versions of Lingeling [21, 22, 23, 24, 25]. As Lingeling, these two are regular participators in SAT Competitions. Plingeling is a portfolio-based CDCL solver comparable to ManySAT [21] whereas Treengeling is implemented by combining the best aspects of CDCL and look-ahead solving [24] in a portfolio solver. The Plingeling and Treengeling versions from the 2014 SAT competition that are used in this study interestingly enough also incorporates a new local search solver called YalSAT [25] in their solving procedure.

[1] A speedup greater than a factor $r$, where $r$ is the amount of available parallel resources (most commonly CPU cores)

**FACT algorithm**

- Trial division

The source code for the trial division algorithm used can be found in appendix A.1.

### 3.3.5 Measuring performance

All of the measures used for this study are based on wall clock time. Therefore, only time was measured during the benchmarking. The tool used for this was the *date* utility in Ubuntu. The raw data for the benchmarks was measured in nanoseconds. Each solver was run on all of the 1600 problem instances for all of the 14 reductions.

To ensure that the benchmarking would terminate at some point, the *timeout* utility was also used as a precaution. To see details about how the benchmarking was executed, see the shell script in appendix A.5.

## 3.4 Aggregating test data

For each SAT solver and the trial division algorithm, the measurement data produced during testing was aggregated on the mean value based on bit length. Because there were 100 test instances per used bit length, the aggregations were assessed to provide a sufficiently smooth result. Categorizing input instances based on bit length is also a logical approach because of the reductions used, which are based on the bit configuration of the product and the factors. The bit length is also the general focus of the problem in the RSA factoring challenges [1] and it is the measurement used when analyzing time complexity for factoring.

## 3.5 Analyzing results

The results acquired by running the tests gave the effectiveness of each examined solver for each reduction. The examination of the parallel solvers however required both the speedup and efficiency compared to some sequential solver. The sequential solver chosen for this purpose was simply the one with the lowest total running time.

The speedup and the efficiency were then calculated by $S = T_s/T_p$ and $E = S/r$.

# Chapter 4

# Results

In this section, the results achieved by applying the previously explained method are given. Firstly, the supporting results of the calibration test are presented. After this, the effectiveness (raw data) results are presented with a summarizing table and graphs for each solver comparing different reductions. A summarizing graph and table with the best solver/reduction combinations in terms of effectiveness are also presented. Lastly, the speedup and efficiency for the parallel solvers are listed in a table.

## 4.1 Calibration test

The execution of the calibration test concluded that an estimation for the amount of available parallel resources $r$ in the testing environment with a 4-core processor was $r \approx 3.91$.

## 4.2 Effectiveness

Table 4.1 shows the total effectiveness (running time) results for all combinations of solvers and reductions. The green cells show the fastest reduction per solver and the red cells the slowest. To compress the table, abbreviations are used in place of the real solver and reduction names. The names for the reductions are represented as follows:

- S - Simplified
- N - N-bit
- F - Fast
- C - Carry-save
- W - Wallace

- R - Recursive
- TS - ToughSAT

The abbreviations for the solvers are self-explanatory.

Further below, figures 4.1-4.10 show more detailed graphs demonstrating how the run time increases with bit length for each solver and reduction. To make the result more comprehensible, there are two graphs for each solver. One shows the ordinary reductions, and the other shows their simplified versions. The graphs are meant to visualize the results shown in table 4.1.

Lastly in figure 4.11, the running times for trial division are demonstrated analogous with the style used for the solver graphs.

Table 4.1: All solvers combined with all reductions.

| | **Mini** | **Many** | **Ling** | **Pling** | **Tree** | **Total** |
|---|---|---|---|---|---|---|
| **NC** | 715 s | 205 s | 637 s | 390 s | 787 s | 2734 s |
| **SNC** | 629 s | 185 s | 717 s | 395 s | 866 s | 2792 s |
| **FC** | 780 s | 224 s | 754 s | 389 s | 934 s | 3081 s |
| **SFC** | 666 s | 208 s | 762 s | 382 s | 914 s | 2932 s |
| **NW** | 852 s | 245 s | 1360 s | 600 s | 1357 s | 4414 s |
| **SNW** | 789 s | 217 s | 1383 s | 624 s | 1374 s | 4387 s |
| **FW** | 876 s | 262 s | 1475 s | 663 s | 1455 s | 4731 s |
| **SFW** | 789 s | 233 s | 1384 s | 634 s | 1437 s | 4477 s |
| **NR** | 737 s | 225 s | 1474 s | 619 s | 1496 s | 4551 s |
| **SNR** | 676 s | 196 s | 1498 s | 623 s | 1544 s | 4537 s |
| **FR** | 877 s | 264 s | 6969 s | 713 s | 1798 s | 10621 s |
| **SFR** | 759 s | 223 s | 4490 s | 800 s | 1792 s | 8064 s |
| **TS** | 1379 s | 350 s | 2407 s | 1326 s | 2390 s | 7852 s |
| **STS** | 895 s | 208 s | 2514 s | 1360 s | 2413 s | 7390 s |
| **Total** | 11419 s | 3245 s | 27824 s | 9518 s | 20557 s | |

### 4.2.1 MiniSat

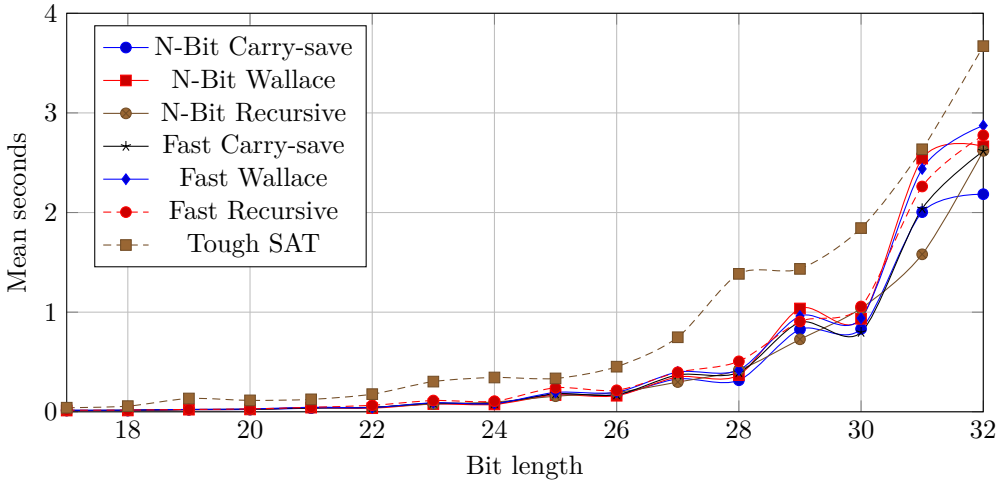


Figure 4.1: MiniSat with default reductions

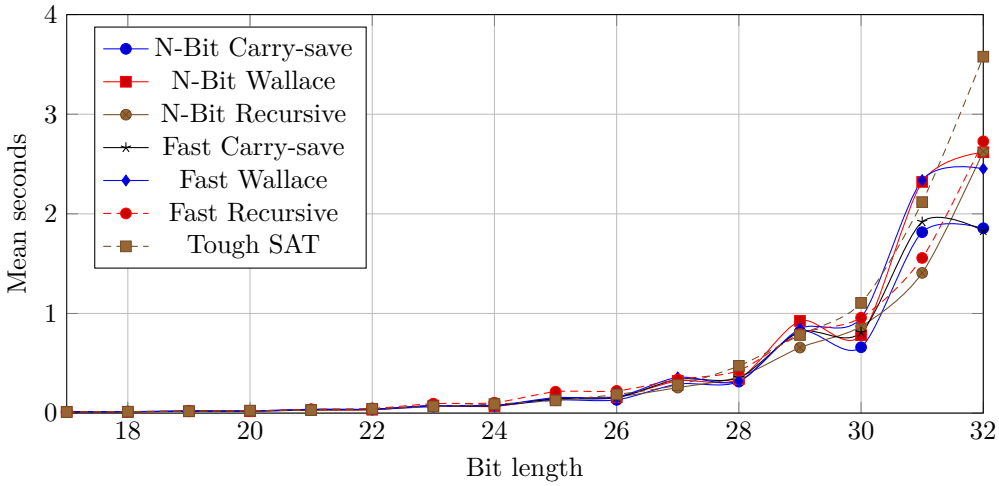


Figure 4.2: MiniSat with simplified reductions

### 4.2.2 ManySAT

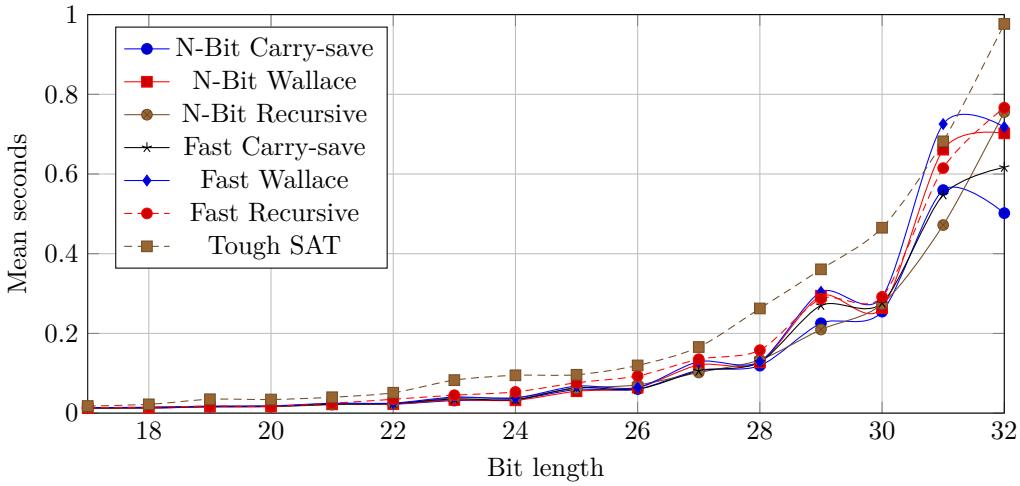


Figure 4.3: ManySAT with default reductions

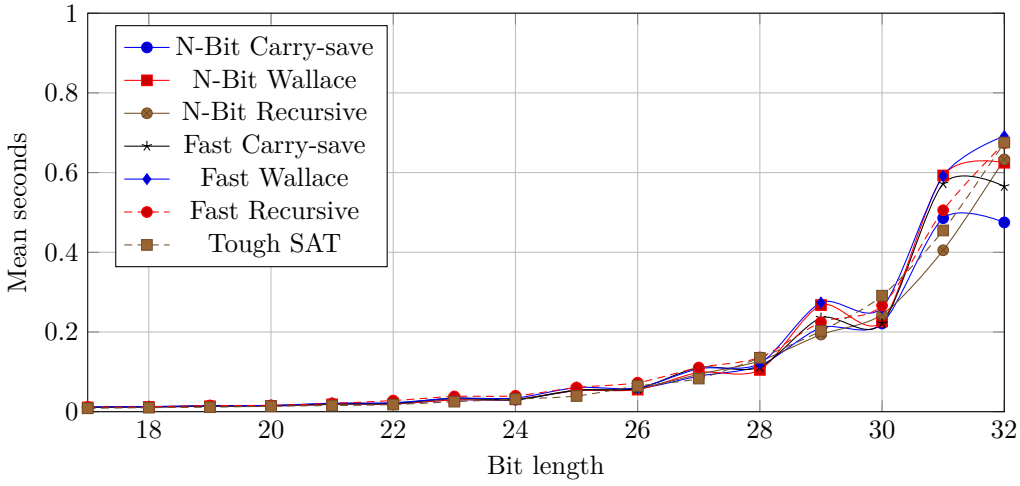


Figure 4.4: ManySAT with simplified reductions

### 4.2.3 Lingeling

In figure 4.5 and 4.6, a strange behaviour can be observed for the Fast Recursive reduction. To ensure this was not caused by a temporarily unstable testing

environment, the tests were run twice for this reduction. Equal results were obtained.

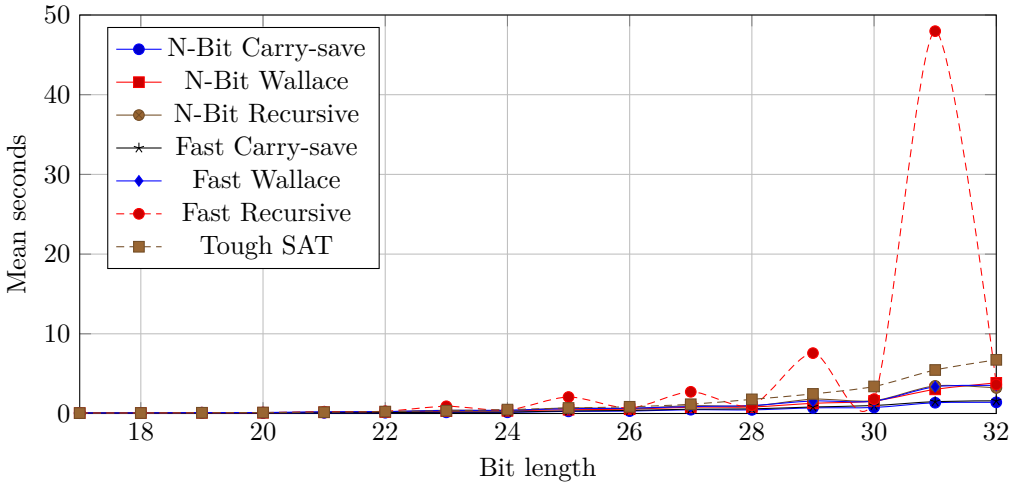


Figure 4.5: Lingeling with default reductions

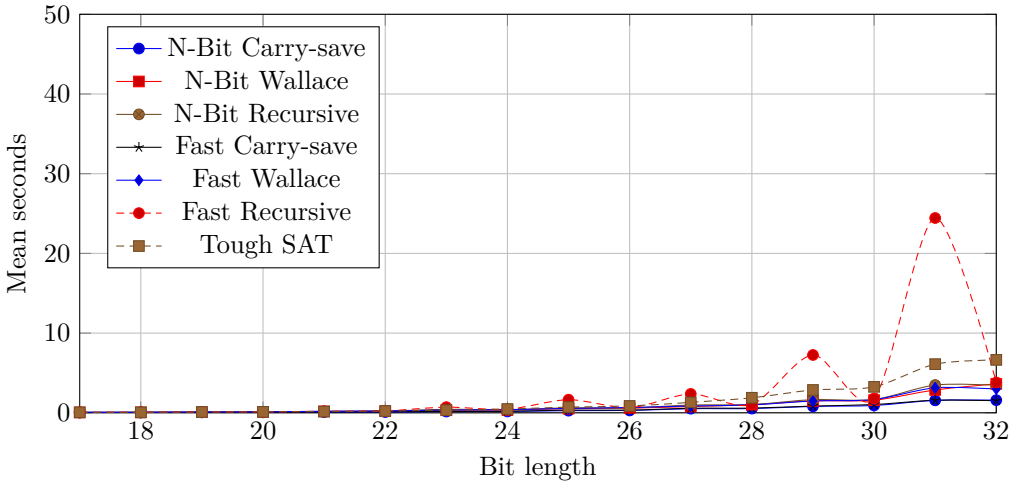


Figure 4.6: Lingeling with simplified reductions

### 4.2.4 Plingeling

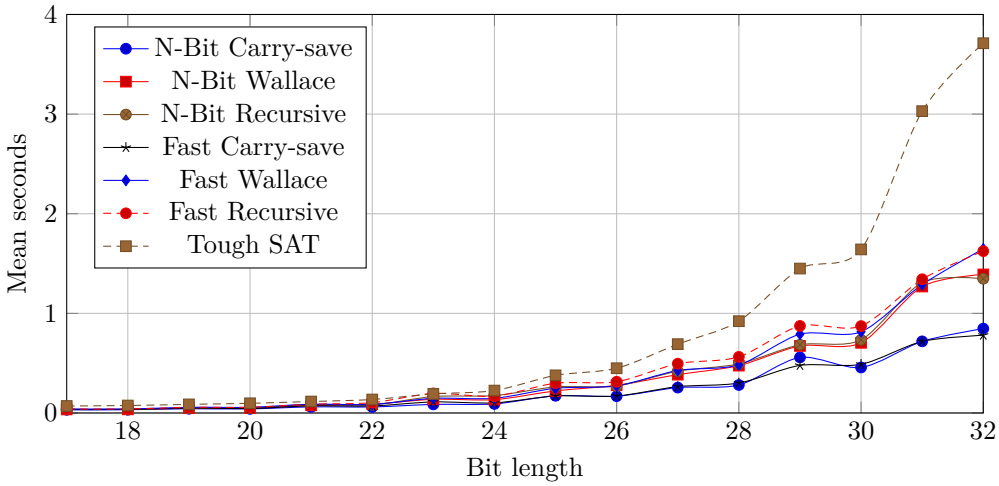


Figure 4.7: Plingeling with default reductions

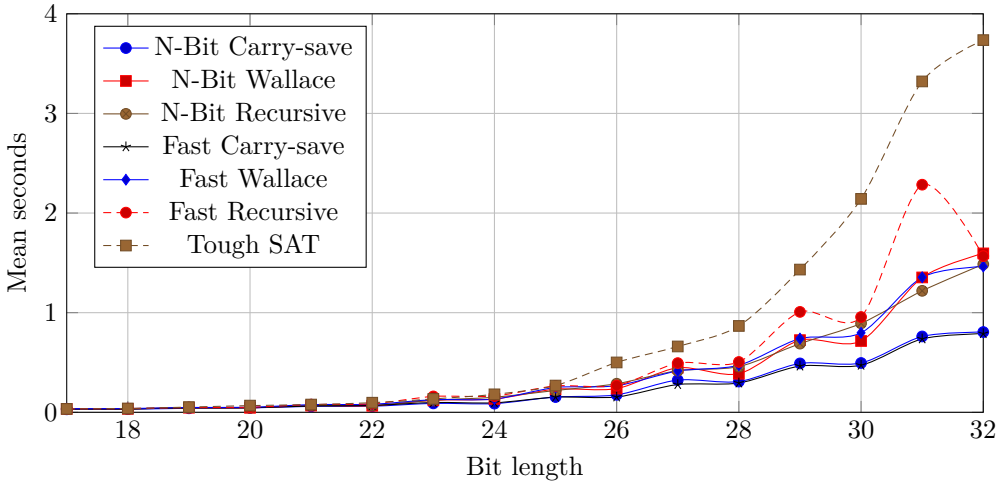


Figure 4.8: Plingeling with simplified reductions

### 4.2.5 Treengeling

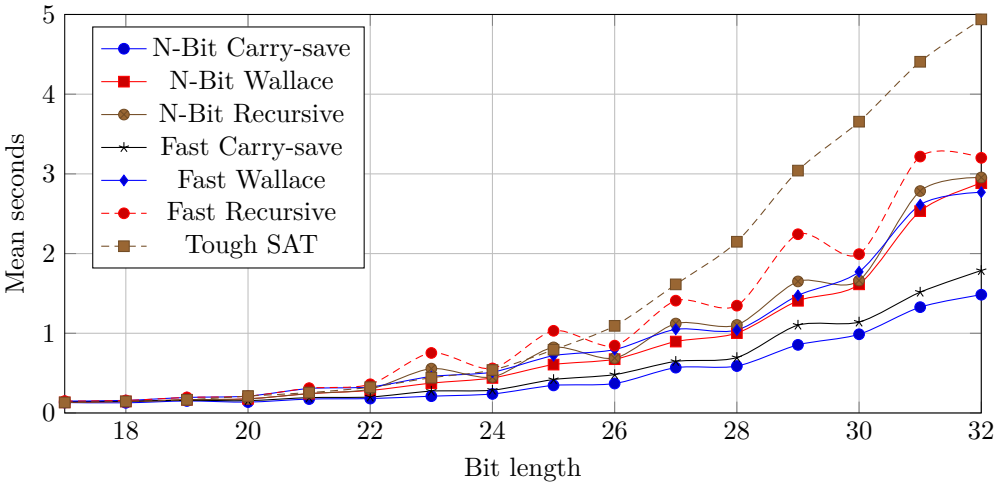


Figure 4.9: Treengeling with default reductions

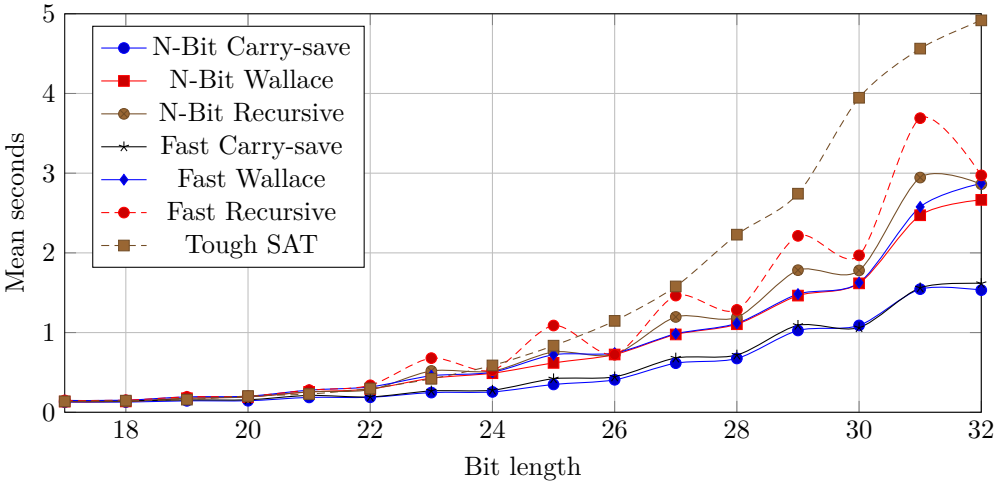


Figure 4.10: Treengeling with simplified reductions

### 4.2.6 Trial division

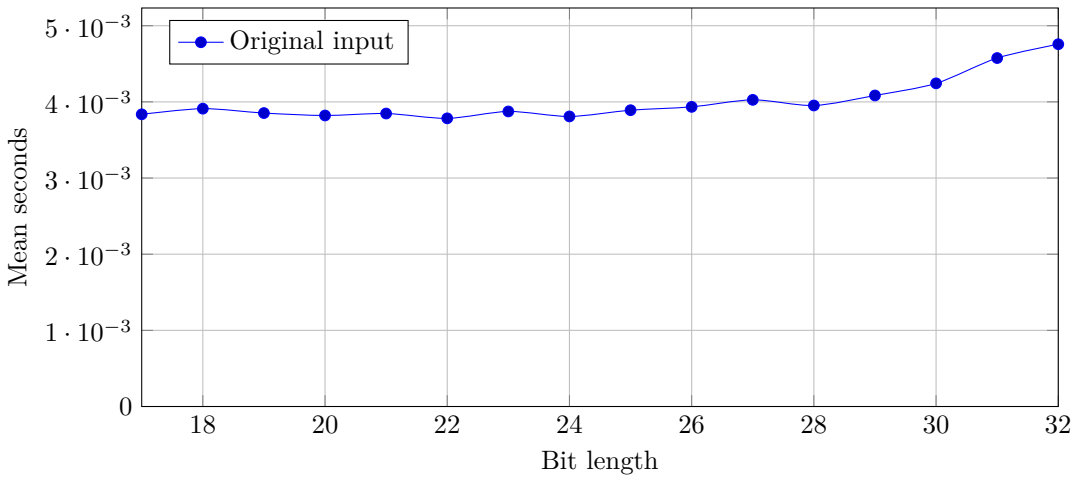


Figure 4.11: Trial division

## 4.3 Summarized effectiveness

In figure 4.12, a comparison graph consisting of all the solvers using their best performing reductions can be found. Further below in table 4.2 follows the total running time for the same optimal solver-reduction combinations shown in the graph.

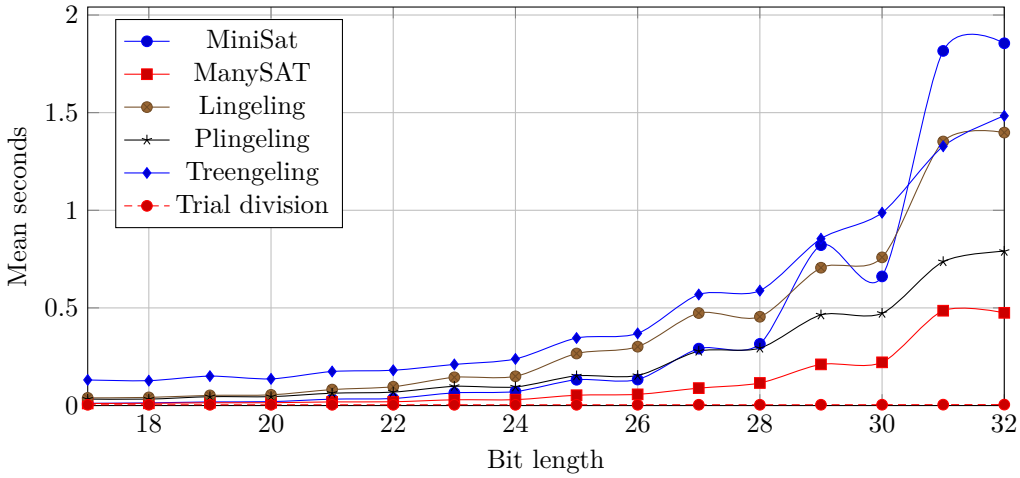


Figure 4.12: All solvers with their top performing reductions

Table 4.2: All solvers total time with their top performing reductions

| Solver | Reduction | Total time |
|---|---|---|
| Trial division | Original indata | 6 s |
| ManySAT | Simplified N-bit Carry-save | 185 s |
| Plingeling | Simplified Fast Carry-save | 382 s |
| MiniSat | Simplified N-bit Carry-save | 629 s |
| Lingeling | N-bit Carry-save | 637 s |
| Treengeling | N-bit Carry-save | 787 s |

## 4.4 Speedup and efficiency for parallel solvers

Table 4.3 lists the speedup and efficiency[1] for all of the parallel solvers compared to MiniSat, which achieved the lowest total running time of the two sequential solvers.

Table 4.3: The efficiency for the parallel solvers compared to MiniSat, the fastest sequential solver.

| Solver | Speedup | Efficiency |
|---|---|---|
| ManySAT | 3.39 | 0.87 |
| Plingeling | 1.65 | 0.42 |
| Treengeling | 0.89 | 0.20 |

[1] An efficiency of 1 means that the parallel solver performed in line with its available resources compared to MiniSat. A lower score indicates problems utilizing the extra resources available.

# Chapter 5

# Discussion

## 5.1 General

One major purpose of this thesis was to evaluate whether parallel SAT solvers were particularly well suited for solving SAT instances generated by reductions from FACT. Based on the results in the experiments carried out in this study, it is clear that there is no major advantage to parallel SAT solving compared to sequential SAT solving other than the natural benefits of parallelism. It also appears that there are no major downsides to parallel SAT solving. No particular reason to suspect that there would be any such downside was identified, but it could still have been the case that the SAT instances generated for this study could have given particularly poor results for parallel solving.

In the experiments performed in this study, no parallel SAT solver achieved a speedup greater than or equal to that of the available resources[1] when compared to the fastest sequential SAT solver, MiniSat. This indicates that the parallel SAT solvers used were not able to fully utilize the resources available to them.

An interesting trend in the results for all SAT solvers used – both sequential and parallel – is that input of odd bit length appears to be more difficult to solve than input of even bit length. This can be seen in the diagrams in the results section. The impact of this supposed difficulty with odd numbers decreases as input size increases, because the time complexity with respect to input size is exponential. However, there appears to be an upper limit around the bit lengths of 26 and 28 – which differs for different solvers – under which the input sizes of odd bit lengths require longer solving times than the successive even bit length input.

[1] Recall that the available resources were calculated to be $r \approx 3.91$ using 4 cores. ManySAT was most efficient in terms of using parallel resources, with a speedup of 3.39 and efficiency of 0.87.

## 5.2 Solvers

Another goal for this thesis was to determine if any particular technique(s) used by parallel SAT solvers was more efficient for solving FACT reductions than others. Of the three parallel solvers used in this study, ManySAT was the clear winner. This is most likely due to ManySAT being more of a general purpose solver than the other two, which are tailored for being competitive on the instances used in the SAT competitions. ManySAT also performed far more stable in general on all of the used reductions, further proving its more general purpose nature over the other two.

Of the two other parallel solvers, Plingeling did much better than Treengeling. Since Plingeling had more in common with ManySAT than with Lingeling, one possibility is that portfolio-based CDCL solvers currently are the most effective parallel solvers for SAT instances generated by reductions from FACT. In fact, Treengeling actually performed *worse* than both of the sequential solvers, achieving a speedup (slowdown) factor of 0.89. This hints at the conclusion that the combination of look-ahead and CDCL is not very effective for the problem instances found in this study. This could however also be the result of Treengeling specifically performing badly, other similar solvers may of course perform better.

The two corresponding sequential solvers MiniSat and Lingeling did not at a first glance follow the pattern of the parallel solvers, since the total running times for these two were quite equal. This was most likely caused by Lingeling agreeing strongly with the reduction named N-bit Carry-save, since MiniSat showed stable results for a far wider range of reductions than Lingeling. To be precise, MiniSat had an astounding 13/14 reductions completing in under 1000 seconds. Lingeling on the other hand, only had 4/14 reductions under 1000 seconds, which all used the Carry-save multiplier. This demonstrates the more general purpose nature of MiniSat.

## 5.3 Reductions

Yet another goal for the study was to determine if any reduction(s) performed better than the others. From the results it is clearly distinguishable that N-bit Carry-save is the most promising reduction. It and its simplified version gave the fastest total running times for all solvers except for Plingeling, where the Fast Carry-save reductions (simplified and unsimplified) achieved a marginally faster run time. The reductions using Carry-save was overall found at the top of the tables showing running times. This is most likely due to the straightforward nature of these reductions combined with the relatively small numbers used in this study. It is quite likely that the Wallace and especially the Recursive multipliers [16] performs better for larger numbers (i.e. larger bit lengths) than the ones used. To examine this thoroughly, more computational power than what was available would be required. This delimitation was mentioned in the introduction.

A very strange oddity found in the result was the extremely long running times (which occurred consistently) for the Fast Recursive reductions on odd bit lengths when paired with Lingeling. The same behaviour could also be seen for Treengeling, however not at all in the same magnitude. As mentioned in the General section above, this anomaly was also to some extent true in general. Plingeling, which is also related to the above two, did however not suffer from this anomaly.

Some other interesting results already partly mentioned in the solver section above were the stable overall performance for all reductions when paired with MiniSat and ManySat. The performance of Nbit Recursive for both of these solvers was especially interesting, as well as the performance for the simplified version of ToughSAT paired with ManySAT. In general, ToughSAT and the Recursive multipliers were otherwise found in the bottom of the tables showing total running times.

Simplifying the formulas with SATELite achieved mixed results regarding performance. For certain combinations of reductions and solvers, the gains were large, and for others the simplification instead resulted in a slower running time. Not much can therefore be said about SATELite. One thing that may explain the varied results is that most solvers themselves incorporate simplifying techniques in their solving process.

## 5.4 Trial division

Not surprisingly, due to the results achieved by previous studies, simple trial division outperformed all parallel solvers by a huge margin. It also performed quite stable for all problem instances, only increasing in run time slightly for the larger instances.

## 5.5 Error sources

One potential error source is the variability of resources in the testing environment in which the SAT solvers were run. Although actions were taken to avoid any major exposure to this, as mentioned in the Method section, the environment used (Ubuntu 14.04) of course still has scheduled background processes that vary in resource usage. This may have minor impacts of varying size on the results.

An aspect that potentially misleads results may be the bit lengths used in this study. Although not quite an error source, it could have an impact on the result with regards to the purpose of the study. The purpose of using reductions from FACT to SAT is of course that SAT solvers should eventually be able to compete with specialized integer factorization algorithms. For this purpose, input of vastly greater size than 32 bits would have to be used in order to study the behavior of SAT solvers used on such large input. It may be the case that the results presented in this thesis do not demonstrate sufficient patterns

or tendencies with regard to larger input sizes. As mentioned in the previous section, greater computing power would be required in order to carry out this study in a reasonable time frame with larger input.

Another aspect which may make the results misleading is the fact that reductions from FACT to SAT took a considerable amount of time – but this was not taken into account when measuring the time for the SAT solving process. In a completely fair comparison, the starting point for measuring runtime should perhaps begin upon receiving the input in the form of a semiprime integer and end when a solution has been found. In the experiments performed for this thesis, the input integer was first reduced to a SAT instance – and time measurement started only as the SAT solver began solving the given SAT instance, and ended when a solution was found (or time limit exceeded).

# Chapter 6

# Conclusions

## 6.1 Problem statement conclusion

Below follows the conclusions to the initial problem statements.

For the first problem statement – regarding whether parallel SAT solvers are better suited than sequential SAT solvers for solving integer factorization – it can be concluded that for the generated input used in this study, parallel solvers were only superior to sequential solvers in terms of the expected performance increase from parallelism. The only noted speedup for parallel SAT solvers was in fact, as mentioned, lower than the amount of resources available to the parallel SAT solvers.

The second problem statement – regarding which technique(s) used for parallel SAT solving has the biggest performance impact for solving the generated input for this study – can not be answered properly because of the insufficient sample size of different parallel SAT solvers. The observation which can be made for this study is that the portfolio-based CDCL parallel SAT solver ManySAT[1] was the highest performing parallel SAT solver amongst the tested solvers, with another portfolio-based CDCL solver Plingeling taking second place. Thus, portfolio-based CDCL techniques seemed to have the biggest impact on performance in the context of this study.

As for the third problem statement – regarding whether there are any reductions techniques from FACT to SAT particularly well suited when solving problem instances with parallel SAT solvers – it appeared that the N-bit Carry-save reduction yielded the highest performance for both parallel and sequential SAT solvers. The N-bit Carry-save was however closely followed by the Fast Carry-save reduction. This superiority to other tested reductions was quite consistent.

Finally, the last problem statement – regarding whether parallel SAT solvers can compete with standard (sequential) trial division for solving FACT – it can be stated with certainty that parallel SAT solvers are not yet able to perform

[1] See method section for a summary of techniques used in ManySAT.

anywhere near as well as the standard trial division algorithm, at least for the relatively low bit lengths examined here.

## 6.2 Significance of results

The results of this thesis align well with previous results of similar theses [3, 4, 5] in the sense that the approach of reducing FACT to SAT remains an inferior method to simply using specialized FACT algorithms. This result holds for parallel SAT solvers, for which the best contender (ManySAT) only experienced a speedup of slightly less than the amount of available resources to the parallel solvers.

It has also been shown that for the selection of SAT solvers and the input size range used in this study, the overall best reduction method is the N-bit Carry-save [16] technique – and this knowledge may be advantageous for further testing within this area.

## 6.3 Further research

The most straightforward extension of this study is to create a larger statistical basis by increasing the amount of solvers tested, choosing multiple ones for each specific solving technique (such as CDCL, look-ahead and stochastic local search) used in different portfolio solvers. This may provide more insight into the second problem statement of this thesis, of which the goal was to identify what parallel solving techniques were best fitted for handling FACT problem instances. This thesis was almost entirely devoted to portfolio CDCL solvers, which, since most widely used, were believed to provide the best results. This may however not be the case. Note that it would probably be best to first examine solving techniques found in sequential solvers independently and then move on to parallel techniques, since parallel portfolio solving techniques are built upon these. The most efficient solution would probably be to separate these two undertakings into different studies.

Another possibility for a further study would be to examine larger problem instances. This especially with regard to the Wallace and Recursive multipliers which were inferior to the more basic reductions for the smaller problem instances found in this study. They may very likely prove to be superior on larger instances since this is their main purpose in other applications. It may even be that these reductions (or others) can compete with trial division for larger bit lengths. This approach was not doable within reasonable time with the hardware available for this study.

Yet another interesting topic to explore is how difficult it is to solve other kinds of SAT instances compared to SAT instances generated by reductions from hard FACT instances (semiprimes). For instance, comparisons could be made to some of the problem groups found in the SAT competitions. It could perhaps be assumed that SAT instances derived from hard FACT instances would also

be hard – but this is not necessarily true.

It is also worth mentioning that many other more complicated multiplication algorithms and binary multipliers could potentially be used as a means to achieve a suitable SAT formula. This is definitely an area worthy of more in-depth studies.

# Bibliography


[1] RSA Laboratories - The RSA Factoring Challenge FAQ. http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge-faq.htm. Accessed: May 7, 2015.

[2] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[3] Stefan Schoenmackers and Anna Cavender. Satisfy This: An Attempt at Solving Prime Factorization using Satisfiability Solvers. http://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/TheS&Ateam/SATeamFinalPaper.pdf. Accessed: May 7, 2015.

[4] John Eriksson and Jonas Höglund. A comparison of reductions from FACT to CNF-SAT, 2014.

[5] Jonatan Asketorp. Attacking RSA moduli with SAT solvers, 2014.

[6] Youssef Hamadi and Christoph Wintersteiger. Seven challenges in parallel sat solving. *AI Magazine*, 34(2):99, 2013.

[7] Mateusz Srebrny. Factorization with SAT – classical propositional calculus as a programming environment. http://www.mimuw.edu.pl/~mati/fsat-20040420.pdf, 2004. Accessed: May 7, 2015.

[8] Adnan Darwiche and Knot Pipatsrisawat. Complete Algorithms. http://reasoning.cs.ucla.edu/fetch.php?id=97&type=pdf, 2008. Accessed: May 7, 2015.

[9] Vladimir Lifschitz. The Davis-Putnam-Logemann-Loveland Procedure. http://www.cs.utexas.edu/users/vl/teaching/lbai/dpll.pdf, 2011. Accessed: May 7, 2015.

[10] Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. http://www.cs.utexas.edu/~isil/cs395t/CDCL.pdf, 2008. Accessed: May 7, 2015.

[11] Marijn Heule, Mark Dufour, Joris Van Zwieten, and Hans Van Maaren. March_eq: implementing additional reasoning into an efficient look-ahead sat solver. In *Theory and Applications of Satisfiability Testing*, pages 345–359. Springer, 2005.

[12] Holger Hoos H. and Thomas Stützle. *Stochastic Local Search, Foundations and Applications*. Morgan Kaufman / Elsevier.

[13] Steffen Hölldobler, Norbert Manthey, V Nguyen, J Stecklina, and P Steinke. A short overview on modern parallel sat-solvers. In *Proceedings of the International Conference on Advanced Computer Science and Information Systems*, pages 201–206, 2011.

[14] Gordana Dodig-Crnkovic. Scientific methods in computer science. In *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, Skövde, Suecia*, pages 126–130, 2002.

[15] Henry Yuen and Joseph Bebel. ToughSAT Generation. `https://toughsat.appspot.com/`, 2011. Accessed: May 7, 2015.

[16] Paul Purdom and Amr Sabry. CNF Generator for Factoring Problems. `http://www.cs.indiana.edu/cgi-pub/sabry/cnf.html`. Accessed: May 7, 2015.

[17] SATELite. `http://minisat.se/SatELite.html`. Accessed: May 7, 2015.

[18] Benchmarks submission guidelines. `http://www.satcompetition.org/2009/format-benchmarks2009.html`. Accessed: May 7, 2015.

[19] Niklas Eén and Niklas Sörensson. MiniSat Page. `http://minisat.se/`. Accessed: May 7, 2015.

[20] Armin Biere. Lingeling, Plingeling and Treengeling . `http://fmv.jku.at/lingeling/`. Accessed: May 7, 2015.

[21] Armin Biere. Lingeling, plingeling, picosat and precosat at sat race 2010.

[22] Armin Biere. Lingeling and friends at the sat competition 2011.

[23] Armin Biere. Lingeling and friends entering the sat challenge 2012.

[24] Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013.

[25] Armin Biere. Yet another local search solver and lingeling and friends entering the sat competition 2014.

[26] SAT Competitions. `http://www.satcompetition.org/`. Accessed: May 7, 2015.

[27] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a Parallel SAT Solver. http://www.cril.univ-artois.fr/~jabbour/manysat.htm. Accessed: May 7, 2015.

[28] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2008.

# Appendix A

# Source code

## A.1 Trial division

```
1  /**
2   * A simple trial division algorithm.
3   *
4   * Author: Daniel Lundén
5   */
6
7  #include <string>
8  #include <stdexcept>
9  #include <iostream>
10 #include <vector>
11 #include <cmath>
12 #include <fstream>
13
14 using namespace std;
15
16 int main(int argc, char *argv[]) {
17   if(argc == 1) {
18     cout << "Error, file name needed as first parameter" << endl;
19     return 0;
20   }
21   unsigned long number;
22
23   try {
24     ifstream in;
25     in.open(argv[1]);
26     string tmp;
27     in >> tmp;
28     number = stoul(tmp);
29     in.close();
30   } catch (const exception& ia) {
31     cout << "Invalid file name" << endl;
32     return 0;
33   }
34
35   if(number == 1) {
```

```
36      cout << "Factors: 1" << endl;
37      return 0;
38    }
39
40    vector<unsigned long> factors;
41    for(unsigned long l = 2;; ++l) {
42      if(l > number/2) {
43        if(number != 1) {
44          factors.push_back(number);
45        }
46        break;
47      } else if (number == l) {
48        factors.push_back(l);
49        break;
50      }
51
52      if(number % l == 0) {
53        factors.push_back(l);
54        number = number / l;
55        while(number % l == 0) {
56          factors.push_back(l);
57          number = number / l;
58        }
59      }
60    }
61
62    cout << "Factors: ";
63    for(int i = 0; i < factors.size(); ++i) {
64      cout << factors[i];
65      if(i != factors.size()-1) {
66        cout << ", ";
67      }
68    }
69    cout << endl;
70  }
```

## A.2 Calibration test

```
1  #!/bin/bash
2
3  TS_SUM=0
4  TN_SUM=0
5
6  for i in $(seq 30); do
7    START1=$(date +%s%N)
8    ../minisat/core/minisat_static sample.in &> /dev/null
9    END1=$(date +%s%N)
10
11   START2=$(date +%s%N)
12   ../minisat/core/minisat_static sample.in &> /dev/null &
13   ../minisat/core/minisat_static sample.in &> /dev/null &
14   ../minisat/core/minisat_static sample.in &> /dev/null &
15   ../minisat/core/minisat_static sample.in &> /dev/null &
16   wait
```

```
  END2=$(date +%s%N)

  let TS_SUM=TS_SUM+$((END1 - START1))
  let TN_SUM=TN_SUM+$((END2 - START2))
done

echo $(echo "4*$TS_SUM/$TN_SUM" | bc -l)
```

## A.3 Semiprime generator

```
/**
 * Generate semiprimes with specified bit lengths.
 * The prime factors are generated with the sieve of Eratosthenes.
 *
 * Author: Daniel Lundén
 *
 */

#include <fstream>
#include <iostream>
#include <cmath>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <string>
#include <sstream>

#define MAX_PRODUCT_BIT_LENGTH 32
#define MIN_PRODUCT_BIT_LENGTH 17
#define PRODUCTS_PER_BIT_LENGTH 100
#define OUTPUT_PATH "base/"

using namespace std;

int main() {
  const unsigned long max = 1ul << (MAX_PRODUCT_BIT_LENGTH + 1)/2;
  vector<bool> primetable(max);
  primetable[0] = false;
  primetable[1] = false;
  for(unsigned long i = 2; i < max; ++i) {
    primetable[i] = true;
  }

  // Sieve
  unsigned long p = 2;
  const unsigned long limit = sqrt(max);
  while(p <= limit) {
    // Remove multiples of current prime
    for(unsigned long mult = 2; mult*p < max; ++mult) {
      primetable[mult*p] = false;
    }

    // Find next prime
```

```
45      ++p;
46      for(;primetable[p] == false; ++p) {}
47    }
48
49    // Bucket primes by their bit lengths
50    vector<vector<unsigned long>> primes((MAX_PRODUCT_BIT_LENGTH + 1)/2 +
          1);
51    for(unsigned long i = 2; i < max; ++i) {
52      if(primetable[i]) {
53        int bits = log2(i) + 1;
54        primes[bits].push_back(i);
55      }
56    }
57
58    // Produce semiprimes to files
59    srand(time(NULL));
60    vector<unsigned long> semiprimes;
61    for(int i = MIN_PRODUCT_BIT_LENGTH; i <= MAX_PRODUCT_BIT_LENGTH; ++i)
          {
62      int factor_bitlength = (i+1)/2;
63      for(int j = 0; j < PRODUCTS_PER_BIT_LENGTH; ++j) {
64        unsigned long prod = 0;
65        int bitlength;
66        do {
67          unsigned long f1 = primes[factor_bitlength][rand() % primes[
              factor_bitlength].size()];
68          unsigned long f2 = primes[factor_bitlength][rand() % primes[
              factor_bitlength].size()];
69          prod = f1*f2;
70          bitlength = log2(prod) + 1;
71        } while (bitlength != i);
72        semiprimes.push_back(prod);
73      }
74    }
75    sort(semiprimes.begin(), semiprimes.end());
76
77    ofstream out;
78    for(int i = 0; i < semiprimes.size(); ++i) {
79      stringstream ss;
80      ss << OUTPUT_PATH << (i+1) << ".in";
81      out.open(ss.str());
82      out << semiprimes[i];
83      out.close();
84    }
85  }
```

## A.4 Test generator

```
1  #!/bin/bash
2
3  echo "Setting up folders..."
4  mkdir base &> /dev/null
5
6  mkdir fast_carry &> /dev/null
```

```
mkdir fast_recursive &> /dev/null
mkdir fast_wallace &> /dev/null
mkdir nbit_carry &> /dev/null
mkdir nbit_recursive &> /dev/null
mkdir nbit_wallace &> /dev/null
mkdir tough_sat &> /dev/null
mkdir s_fast_carry &> /dev/null
mkdir s_fast_recursive &> /dev/null
mkdir s_fast_wallace &> /dev/null
mkdir s_nbit_carry &> /dev/null
mkdir s_nbit_recursive &> /dev/null
mkdir s_nbit_wallace &> /dev/null
mkdir s_tough_sat &> /dev/null
echo "Done!"

ghc --make cnf1.hs &> /dev/null

echo "Generating semiprimes..."
g++ -std=c++0x -O3 semiprime_generator.cpp
./a.out
echo "Done!"

generate() {
  for i in $(ls base); do
    ./cnf1 $(cat base/$i) $1 > $2/$i
    ./SatELite_v1.0_linux $2/$i s_$2/$i &> /dev/null
  done
}

echo "Generating fast carry CNF..."
generate "fast carry-save" fast_carry
echo "Done!"

echo "Generating fast recursive CNF..."
generate "fast recursive" fast_recursive
echo "Done!"

echo "Generating fast wallace CNF..."
generate "fast wallace" fast_wallace
echo "Done!"

echo "Generating n-bit carry CNF..."
generate "n-bit carry-save" nbit_carry
echo "Done!"

echo "Generating n-bit recursive CNF..."
generate "n-bit recursive" nbit_recursive
echo "Done!"

echo "Generating n-bit wallace CNF..."
generate "n-bit wallace" nbit_wallace
echo "Done!"

echo "Generating ToughSAT CNF..."
for i in $(ls base); do
  python -c "from factoring_j import *; print generate_instance2($(cat
      base/$i), False)" > tough_sat/$i
```

```
  ./SatELite_v1.0_linux tough_sat/$i s_tough_sat/$i &> /dev/null
done
echo "Done!"

echo "Removing executables..."
rm a.out
rm factoring_j.pyc
rm cnf1
rm cnf1.hi
rm cnf1.o
echo "Done!"

echo "All operations completed!"
```

## A.5 Test runner

```
#!/bin/bash

# Function for running tests. The parameters are:
# $1 = Solver path
# $2 = Indata type
# $3 = Output file name
runtests() {
  echo "$1 - $2"
  rm results/$2/$3 &> /dev/null
  echo -e 'instance_num\ttime' >> results/$2/$3;
  for i in $(seq $(ls -1 indata/base | wc -l)); do
    START=$(date +%s%N)
    timeout 10m solvers/$1 indata/$2/$i.in 1> /dev/null
    TIMEOUT="$?"
    END=$(date +%s%N)

    if [[ "$TIMEOUT" = "124" ]]; then
      # echo timeout >> results/$2/$3
      echo "$i / $(ls -1 indata/base | wc -l)"
      return
    else
      echo -n $i >> results/$2/$3
      echo -en '\t' >> results/$2/$3
      echo $((END - START)) >> results/$2/$3
    fi

    echo -ne "$i / $(ls -1 indata/base | wc -l) \r"
  done
  echo "$(ls -1 indata/base | wc -l) / $(ls -1 indata/base | wc -l)"
}

echo "Setting up folders..."
mkdir results/base &> /dev/null
mkdir results/fast_carry &> /dev/null
mkdir results/fast_recursive &> /dev/null
mkdir results/fast_wallace &> /dev/null
mkdir results/nbit_carry &> /dev/null
mkdir results/nbit_recursive &> /dev/null
```

```
mkdir results/nbit_wallace &> /dev/null
mkdir results/tough_sat &> /dev/null
mkdir results/s_fast_carry &> /dev/null
mkdir results/s_fast_recursive &> /dev/null
mkdir results/s_fast_wallace &> /dev/null
mkdir results/s_nbit_carry &> /dev/null
mkdir results/s_nbit_recursive &> /dev/null
mkdir results/s_nbit_wallace &> /dev/null
mkdir results/s_tough_sat &> /dev/null
echo "Done!"

# Compile trial division
g++ -std=c++0x -O3 solvers/trial_division.cpp

# ----------------------- TEST RUNNING AREA -------------

# Trial division
runtests ./a.out base trial_division.raw

# MiniSAT
runtests minisat/core/minisat_static fast_carry minisat.raw
runtests minisat/core/minisat_static fast_recursive minisat.raw
runtests minisat/core/minisat_static fast_wallace minisat.raw
runtests minisat/core/minisat_static nbit_carry minisat.raw
runtests minisat/core/minisat_static nbit_recursive minisat.raw
runtests minisat/core/minisat_static nbit_wallace minisat.raw
runtests minisat/core/minisat_static tough_sat minisat.raw
runtests minisat/core/minisat_static s_fast_carry minisat.raw
runtests minisat/core/minisat_static s_fast_recursive minisat.raw
runtests minisat/core/minisat_static s_fast_wallace minisat.raw
runtests minisat/core/minisat_static s_nbit_carry minisat.raw
runtests minisat/core/minisat_static s_nbit_recursive minisat.raw
runtests minisat/core/minisat_static s_nbit_wallace minisat.raw
runtests minisat/core/minisat_static s_tough_sat minisat.raw

# ManySAT
runtests manysat2.0/core/manysat2.0_static fast_carry manysat.raw
runtests manysat2.0/core/manysat2.0_static fast_recursive manysat.raw
runtests manysat2.0/core/manysat2.0_static fast_wallace manysat.raw
runtests manysat2.0/core/manysat2.0_static nbit_carry manysat.raw
runtests manysat2.0/core/manysat2.0_static nbit_recursive manysat.raw
runtests manysat2.0/core/manysat2.0_static nbit_wallace manysat.raw
runtests manysat2.0/core/manysat2.0_static tough_sat manysat.raw
runtests manysat2.0/core/manysat2.0_static s_fast_carry manysat.raw
runtests manysat2.0/core/manysat2.0_static s_fast_recursive manysat.raw
runtests manysat2.0/core/manysat2.0_static s_fast_wallace manysat.raw
runtests manysat2.0/core/manysat2.0_static s_nbit_carry manysat.raw
runtests manysat2.0/core/manysat2.0_static s_nbit_recursive manysat.raw
runtests manysat2.0/core/manysat2.0_static s_nbit_wallace manysat.raw
runtests manysat2.0/core/manysat2.0_static s_tough_sat manysat.raw

# Lingeling
runtests lingeling-ayv-86bf266-140429/binary/lingeling fast_carry
    lingeling.raw
runtests lingeling-ayv-86bf266-140429/binary/lingeling fast_recursive
    lingeling.raw
```

```
runtests lingeling-ayv-86bf266-140429/binary/lingeling fast_wallace
    lingeling.raw
runtests lingeling-ayv-86bf266-140429/binary/lingeling nbit_carry
    lingeling.raw
runtests lingeling-ayv-86bf266-140429/binary/lingeling nbit_recursive
    lingeling.raw
runtests lingeling-ayv-86bf266-140429/binary/lingeling nbit_wallace
    lingeling.raw
runtests lingeling-ayv-86bf266-140429/binary/lingeling tough_sat
    lingeling.raw
runtests lingeling-ayv-86bf266-140429/binary/lingeling s_fast_carry
    lingeling.raw
runtests lingeling-ayv-86bf266-140429/binary/lingeling s_fast_recursive
     lingeling.raw
runtests lingeling-ayv-86bf266-140429/binary/lingeling s_fast_wallace
    lingeling.raw
runtests lingeling-ayv-86bf266-140429/binary/lingeling s_nbit_carry
    lingeling.raw
runtests lingeling-ayv-86bf266-140429/binary/lingeling s_nbit_recursive
     lingeling.raw
runtests lingeling-ayv-86bf266-140429/binary/lingeling s_nbit_wallace
    lingeling.raw
runtests lingeling-ayv-86bf266-140429/binary/lingeling s_tough_sat
    lingeling.raw

# Plingeling
runtests plingeling-ayv-86bf266-140429/binary/plingeling fast_carry
    plingeling.raw
runtests plingeling-ayv-86bf266-140429/binary/plingeling fast_recursive
     plingeling.raw
runtests plingeling-ayv-86bf266-140429/binary/plingeling fast_wallace
    plingeling.raw
runtests plingeling-ayv-86bf266-140429/binary/plingeling nbit_carry
    plingeling.raw
runtests plingeling-ayv-86bf266-140429/binary/plingeling nbit_recursive
     plingeling.raw
runtests plingeling-ayv-86bf266-140429/binary/plingeling nbit_wallace
    plingeling.raw
runtests plingeling-ayv-86bf266-140429/binary/plingeling tough_sat
    plingeling.raw
runtests plingeling-ayv-86bf266-140429/binary/plingeling s_fast_carry
    plingeling.raw
runtests plingeling-ayv-86bf266-140429/binary/plingeling
    s_fast_recursive plingeling.raw
runtests plingeling-ayv-86bf266-140429/binary/plingeling s_fast_wallace
     plingeling.raw
runtests plingeling-ayv-86bf266-140429/binary/plingeling s_nbit_carry
    plingeling.raw
runtests plingeling-ayv-86bf266-140429/binary/plingeling
    s_nbit_recursive plingeling.raw
runtests plingeling-ayv-86bf266-140429/binary/plingeling s_nbit_wallace
     plingeling.raw
runtests plingeling-ayv-86bf266-140429/binary/plingeling s_tough_sat
    plingeling.raw

# Treengeling
```

```
123 runtests treengeling-ayv-86bf266-140429/binary/treengeling fast_carry
        treengeling.raw
124 runtests treengeling-ayv-86bf266-140429/binary/treengeling
        fast_recursive treengeling.raw
125 runtests treengeling-ayv-86bf266-140429/binary/treengeling fast_wallace
         treengeling.raw
126 runtests treengeling-ayv-86bf266-140429/binary/treengeling nbit_carry
        treengeling.raw
127 runtests treengeling-ayv-86bf266-140429/binary/treengeling
        nbit_recursive treengeling.raw
128 runtests treengeling-ayv-86bf266-140429/binary/treengeling nbit_wallace
         treengeling.raw
129 runtests treengeling-ayv-86bf266-140429/binary/treengeling tough_sat
        treengeling.raw
130 runtests treengeling-ayv-86bf266-140429/binary/treengeling s_fast_carry
         treengeling.raw
131 runtests treengeling-ayv-86bf266-140429/binary/treengeling
        s_fast_recursive treengeling.raw
132 runtests treengeling-ayv-86bf266-140429/binary/treengeling
        s_fast_wallace treengeling.raw
133 runtests treengeling-ayv-86bf266-140429/binary/treengeling s_nbit_carry
         treengeling.raw
134 runtests treengeling-ayv-86bf266-140429/binary/treengeling
        s_nbit_recursive treengeling.raw
135 runtests treengeling-ayv-86bf266-140429/binary/treengeling
        s_nbit_wallace treengeling.raw
136 runtests treengeling-ayv-86bf266-140429/binary/treengeling s_tough_sat
        treengeling.raw
137
138 # ------------------------------------------------------
139
140 # Remove trial division executable
141 rm solvers/a.out &> /dev/null
```