A Parallel SAT Solver

Submitted in partial fulfillment of the requirements

of the degree of

Bachelor of Technology and Master of Technology

by

Sahil Agarwal

(*Roll no.* 10D070017)

Supervisor:

Prof. Madhav Desai

Department of Electrical Engineering

Indian Institute of Technology Bombay

2015

Sahil Agarwal/ Prof. Madhav Desai (Supervisor): **"A Parallel SAT Solver"**, *Dual Degree Dissertation*, Department of Electrical Engineering, Indian Institute of Technology Bombay, June 2015.

Abstract

SAT solvers are used in various domains like artificial intelligence, circuit design, automatic theorem proving, etc. With increasingly large search spaces, greater speed in solving is required. Modern SAT solvers are using available multi-core systems to achieve speedup in SAT solving.

In this work I focus on massive parallelization of SAT solvers. I present a *divide and conquer* parallel SAT solver solving decomposed subproblems in parallel using GNU Parallel [1] with MiniSat 2.2 [2] used as the sequential solver. The main features of the program are described. The performance is measured over different number of computing nodes. It is compared to state-of-the-art parallel solvers like ManySAT 2.0 [3].

Index terms: shared memory, divide and conquer, GNU Parallel

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Boolean Satisfiability Problem

The Boolean Satisfiability Problem (SAT) is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. An interpretation is the assignment of TRUE or FALSE values to variables of the given formula. If an assignment exists such that the formula evaluates to TRUE, the formula is called satisfiable. If no such assignment exists, the function expressed by the formula is identically FALSE for all possible variable assignments and the formula is unsatisfiable. A *SAT solver* is the algorithm which solves the SAT problem.

Definitions and Terminology: A propositional logic formula, also called Boolean expression, is built from variables, operators AND (conjunction, also denoted by $\wedge$), OR (disjunction, $\vee$), NOT (negation, $\neg$), and parentheses. A literal consists of a variable A and is either positive (A) or negated ($\neg A$). A clause is a disjunction of literals. A formula is in the conjunctive normal form (CNF) if it is a conjunction of clauses. A boolean formula is generally given to the SAT solver in CNF.

1.2 Overview of SAT solvers

1.2.1 DPLL Algorithm

Most modern SAT-solvers are based on the Davis-Putnam-Loveland-Logemann (DPLL) algorithm [4]. It is a complete, backtracking-based search algorithm for solving the CNF-SAT

problem. [1]

The *backtracking algorithm* runs by choosing a literal, assigning a truth value to it, simplifying the formula and then recursively checking if the simplified formula is satisfiable; if this is the case, the original formula is satisfiable; otherwise, the same recursive check is done assuming the opposite truth value. This is known as the *splitting rule*, as it splits the problem into two simpler sub-problems.

The DPLL algorithm enhances over the backtracking algorithm by using the following rules at each step:

- *Unit propagation* - If a clause is a unit clause, i.e. it contains only a single unassigned literal, this clause can only be satisfied by assigning the necessary value to make this literal true.

- *Pure literal elimination* - If a propositional variable occurs with only one polarity in the formula, it is called pure. Pure literals can always be assigned in a way that makes all clauses containing them true. Thus, these clauses do not constrain the search anymore and can be deleted.

Unsatisfiability of a given partial assignment is detected if one clause becomes empty, i.e. if all its variables have been assigned in a way that makes the corresponding literals false. This is known as the *conflict clause*. Satisfiability of the formula is detected either when all variables are assigned without generating the empty clause, or, in modern implementations, if all clauses are satisfied. Unsatisfiability of the complete formula can only be detected after exhaustive search.

1.2.2 Sequential SAT solvers

Modern SAT solvers are based on the classical DPLL search procedure, combined with heuristics such as:

1. Activity-based variable selection heuristics (e.g. VSIDS) [5] keep track of the frequency of occurrence of variables during the search. This provides more information about variables which are potentially important to the search, thereby helping to intensify the search.

2. The idea to analyze the conflict clause further led to the conflict driven clause learning (CDCL) [6]. Resolving the conflict clause and the clauses which have been used in the

[1] Henceforth, the input formula is in CNF unless specified otherwise.

implications, new clauses are learned. These learned clauses can be added to the given formula leading to an improved backtracking behavior, where larger parts of the search tree are closed by a single conflict. The most commonly used version of this algorithm is described in [7].

The most commonly used SAT-solver that implements most of the mentioned techniques is MiniSat [2]. This solver provides a basis for many sequential and parallel SAT-solvers including the one implemented in this work.

1.2.3 Parallel SAT solving

Parallel SAT solvers are based on one of the following two approaches [8]

1. *Divide and Conquer*: The recursive application of the split rule in the DPLL algorithm provides a natural way to parallelize the search for a satisfying assignment . These solvers either divide the search space using certain heuristics, or decompose the formula using decomposition techniques. The first parallel SAT-solvers used single-core CPUs which communicated via a network. When shared memory architectures became available, shared memory communication was utilized.

2. *Portfolio*: Instead of implementing cooperative parallelism by splitting the search space, competitive parallelism is applied where all parallel solvers try to find a solution for the same SAT instance. Portfolios take advantage of the main weakness of modern solvers - their sensitivity to parameter tuning. Each processor unit runs a version of the solver with predetermined parameters, working on the full problem instance

1.2.4 Relevant Parallel SAT solvers

A popular decomposition-based parallel solver is PMSAT [9]. It is based on MiniSat 1.14 and MPI (Message Passing Interface). The solver uses a master-slave approach with a fixed number of slaves. The master creates the assumptions that are used to split an instance. For k slaves, $3k$ splitting variables are selected such that 2^{3k} jobs have to be handled. The master stores the splitting variables and sends a partial interpretation to the next free slave. Load balancing is implemented by providing sufficiently many tasks. If a slave has shown that its subformula is undecidable, it returns the 50 most active learned clauses of length less than 21 to the master.

Because these clauses are logical consequences of the input formula, the master can forward them to the running slaves. Additionally, the master removes tasks which became unsatisfiable based on the newly received clauses from its tasks queue.

ManySAT [3] was the first portfolio solver, that created portfolios based on varying parameters such as restart policies, randomness in the decision heuristic and variable polarity heuristic. It is built on top of MiniSat with a shared-memory model using OpenMP. The communication between the solvers is organized through lockless queues which contain all the learned clauses that a particular core wants to share. Since it uses MiniSat, each core runs a CDCL-based DPLL solver that broadcasts all learned clauses of up to a fixed length (this length is determined empirically). Our solver's performance is compared with that of ManySAT.

1.3 Motivation and Objectives

Until now, modern CPUs contain only few cores. Recent parallel SAT-solvers for such shared-memory architectures are mainly based on the techniques developed for sequential SAT-solvers. As soon as the individual solvers running on different cores share memory, the efficiency of the individual solvers decreases. None of these parallel solvers seem to scale well if hundreds or even thousands of cores become available.

In this work, I evaluate scaling results for my implementation of a parallel SAT solver intended to scale to hundreds or thousands of cores, using a combination of shared and distributed memory. The work is evaluated based on the following questions

1. Scaling to a large number of cores: How many cores is it possible to scale our solver to?
2. Demonstrating parallel scaling: Does adding more cores make the solver faster?
3. Speedups over existing solvers: Is performance better than existing parallel solvers?

Chapter 2

Solver Description

In this chapter, we describe the the structure and implementation of our parallel SAT solver. The solver implements a divide and conquer approach. The search space is divided using Orthonormal decomposition (Generation). The generated sub-problems are solved in parallel over many machines/cores using GNU Parallel (Solution). The idea behind this approach is that time taken in choosing splitting variables will be much lesser than solving the reduced subproblem. So, the job of solving the reduced sub-problems is performed over many computing nodes.

[Discussion] To simplify the work, the two parts are divided into Generation and Solution (better word). Generation is a sequential program which takes problem instance and ouputs a list of assumptions to text files. Solution takes the original problem and the assumptions and solves them in parallel. The independence of the two parts leads to easy optimization and mix-and-match to observe performance.

2.1 Part1 - Assumptions Generation

The given problem is split into several sub-problems using a DPLL-like approach.

Variable Selection

- *Fixed heuristic* - The problem is split using arbitrary variables without taking into account the given problem.
- *Frequency-based heuristic* - At each stage of decomposition, the most frequent variable in the current sub-problem is chosen for further decomposition.

- *Frequency-based (simple)* - By choosing the more frequent variables in the input problem, the assumptions will simplify more clauses and thus, hopefully, reduce the problem to a greater extent.

Assumptions generation

Given a set of selected variables, specific values are assigned to them and the resulting search subspace is ready to be analyzed. This assignment procedure is termed the assumptions generation process. The assumptions generation process is akin to the generation of the *guiding path*, a concept used in [9]. However here the program generates at once all the paths associated with all search spaces to be used. Considering the selected variables, there is more than one way to generate assumptions and to explore the assignments tree. In my implementation, all possible assumptions are generated from the variables chosen by varying the polarity of each literal. Each assumption has equal number of literals.

2.1.1 Implementation

Modifying my SAT solver *(slow)*

I used the core of my sequential solver to implement generation (for fixed and frequency-based heuristic). The idea was to spend least time possible on the sequential part (Generation) and offload most of the work to the parallel part. The modifications from the sequential implementation are listed below:

- *Decomose()* recursively calls itself without checking if the current sub-problem is SAT or UNSAT.
- *simpleSimplify()* just simplifies the problem given the assumptions. No unit propagation is performed.
- *SplitVar()* decides the splitting variable according to the given heuristic.
- *PrintAssums()* prints the assumptions into text files.

Parsing cnf-file into array

The pseudo-code of the program is shown in Figure 2.1. The time taken to execute this program is negligible for large problems as compared to the above implementation. The performance of Part2 also doesn't vary much. Therefore, this will be used for measuring performance of the SAT solver.

```
class lit { int id; int occur};
main(){
      parse parameters; (no. of subproblems)
      read input file into array of size nvars and type lit;
      sort array according to lit.occur;
      decide variables (according to no. of subproblems);
      print assumptions;
}
```

Figure 2.1: Part1 program

2.2 Parallel Solving

This part of the solver takes the original problem and assumption files as input and invokes Minisat on them in parallel using GNU Parallel.

2.2.1 Implementation

Below is the description of the important components

- *part2.cpp* - Takes a CNF problem and a list of assumptions as input and solves using MiniSat.
- *call_msat()* - A bash function; takes a CNF problem and a list of assumptions as input and calls the executable of part2.cpp. Returns SAT/UNSAT
- *GNU parallel command* -

  ```
  parallel -j8 call_msat <original_problem>{}\
           ::: <generated_assumptions> | parallel --halt 2  exit
  ```

  The above command calls the bash function call_msat() on the generated assumptions and stops the execution if SAT is detected.

Chapter 3

Performance Evaluation

In this chapter, I compare the performance (computation time) of MiniSat 2.2, ManySAT 2.0 and my solver, on several benchmark instances.

3.1 Experimental setup

3.1.1 Performance metrics

The sequential MiniSAT measures computation time as the CPU time from the beginning to the end of the execution. For my solver, I am measuring the total wall clock time in executing Part1 (sequential generation) and Part2 (solving in parallel). All times presented in this section are in seconds.

Regarding the performance measures related with time, I present the relative speedup, relative efficiency and serial fraction. These values are calculated using the following formulas [9]:

Definition 3.1.1. I will denote T_1 as the execution time of the sequential program (MiniSat 2.2), T_p as the execution time of the parallel program (my solver), speedup as s_p and efficiency as e_p, defined respectively as

$$s_p = \frac{T_1}{T_p} \tag{3.1}$$

$$e_p = \frac{s_p}{p} = \frac{T_1}{T_p \times p} \tag{3.2}$$

3.1.2 Benchmarks and execution environment

For the initial tests, 3 instances were used as benchmarks, their abbreviated name, solution type, number of variables and clauses are presented in Table 3.1. These benchmarks are available at `http://www.miroslav-velev.com/sat_benchmarks.html` The tests were made on the

Table 3.1: Benchmark instances

Instance	Abbreviation	Solution	Variables	Clauses
engine_4.cnf	engine_4	UNSAT	6944	66621
engine_6_nd_case1.cnf	engine_6	UNSAT	45435	610120
12pipe_bug1_q0.cnf	12pipe	SAT	138917	4678756

machines in Embedded Systems Lab in Electrical Engineering Department. Each machine has an Intel(R) Core (TM) i7-2600 processor at 3.4 GHz, 4 GB of RAM, Linux operating system. There are 4 cores per computer and 2 processing units per core (hyperthreading). All computers have access to the same hard drive shared via the Network File System (NFS).

3.2 Performance

3.2.1 Performance of existing solvers

The computation times for MiniSat and ManySAT are given in Table 3.2

Table 3.2: Performance of MiniSat and ManySAT

Benchmark	MiniSat time	ManySAT time
engine_4	1.6	0.78
engine_6	32	21.5
12_pipe	2.2	4.3

The parameters used for MiniSat and ManySAT are given in the following commands in Figure 3.1 (taken from README files of both). I have used the same MiniSat parameters for my solver.

```
minisat <cnf-file> -no-luby -rinc=1.5 -phase-saving=0 -rnd-freq=0.02

manysat <cnf-file> -no-luby -rinc=1.5 -phase-saving=0
    -rnd-freq=0.02 -ncores=4 -limitEx=10 -det=0 -ctrl=0
```

Figure 3.1: Parameters used for MiniSat and ManySAT

3.2.2 Performance of my solver on shared memory

Performance of my solver is shown in Tables 3.3 and 3.4. The experiments are carried on 1 machine (8 cores). (!! indicates memory overflow)

- *engine_4*: The performance is best when using assignments of size 6 (number of subproblems = 64).
- *12pipe*: The algorithm is failing for a reasonably easy SAT instance in this experiment!

Table 3.3: Execution times for engine_6

No. of subproblems	**T_gen**	**T_sol**	**Total**	**s_p**	**e_p**
2	0.22	30.9	31.12	1.03	0.13
4	0.22	21.67	21.89	1.46	0.18
8	0.23	24.5	24.73	1.29	0.16
16	0.24	24.06	24.3	1.32	0.17
32	0.26	16.5	16.76	1.91	0.24
64	0.3	14.3	14.6	2.19	0.27
128	0.39	16.1	16.49	1.94	0.24
256	0.57	20.14	20.71	1.55	0.19
512	0.93	26.79	27.72	1.15	0.14
1024	1.56	41.55	43.11	0.74	0.09

Table 3.4: Execution times for12pipe

No. of subproblems	**T_gen**	**T_sol**	**Total**	**s_p**	**e_p**
2	1.71	!!			
4	1.73	!!			
8	1.73	!!			
16	1.82	!!			
32	1.8	!!			
64	3.8	!!			
128	5.26	109	114.26	0.280063	0.035008
256	4.74	85	89.74	0.356586	0.044573
512	4.56	418	422.56	0.075729	0.009466
1024	6.67	335	341.67	0.093658	0.011707

Chapter 4

Conclusion and Future Work

4.1 Conclusion

I have presented a new parallel SAT-solver, that is based on MiniSAT and uses GNU Parallel, to be executed in clusters or in a grid of computers.

The development of the solver gave an indication of the potential contribution of parallel computing in SAT solving. It showed how a simple idea like domain decomposition can introduce improvements into the search and decrease the time spent.

4.2 Future work

1. *Pre-processing* the input problem to guess best number of sub-problems to decompose into
2. *Incremental SAT solving*
3. *Large-scale parallelism*

Appendix A

SAT solver based on Orthonormal Decomposition

A.1 Introduction

I have developed a SAT solver, from scratch, to implement the parallel algorithm based on *Orthonormal Decomposition*. This algorithm is a generalized version of the DPLL algorithm [4]. A sequential form of the algorithm has been implemented currently, to verify the correctness of the approach. The solver implementation is modular, allowing *future users* to make domain specific extensions or adaptations of current techniques. The code is available at `https://github.com/sahilag/DDP/tree/master/ONexpansion/new`

A.2 Implementation

Class *Solver* - It is the basis of the entire solver. It contains the data structure C which stores the clauses. Its member functions help interect with the class and can be used to solve SAT problems.

Below is the description of the important components of the solver:

- *Main*() (Algorithm 1) takes as input the CNF file and stores the clauses in C. Calls Solve() and outputs SAT/UNSAT.
- C - It refers to the data structure which stores the clauses using *sparse matrix representation.*

- *Solver::Solve*() (Algorithm 2) is called by the main program. It takes as input C and n_0. It decides whether to decompose further or solve by direct search over all assignments. It returns whether the input problem is SAT or UNSAT.
- *Solver::Decompose*() (Algorithms 3 and 4) decides an ON set of terms and decomposes the input problem. It calls *Solve*() for each of the reduced problems.
- *Solver::Simplify*() consists of functions *unitpropagate*() and *pureliterals*() which simplify the problem and remove satisfied clauses.
- *Solver::SolveMinisat*() is a function that takes as input C and returns SAT/UNSAT. It uses the Minisat solver [2].

The sequential algorithm uses Algorithms 2 and 3. The parallel implementation will only need a slight modification in the decompose() function as shown in Algorithm 4.

Algorithm 1 Main()

Input: CNF file input.cnf
  Solver S
  $S.C \leftarrow$ input.cnf
  return *S.Solve*(C)
Output: SAT/UNSAT

Algorithm 2 Base Algorithm - Solver::Solve()

Input: C, n_0
  $C \leftarrow$*Simplify*(C)
  if $nVars \geq n_0$ **then**
    Decompose(C)
  else
    return *SolveMinisat*(C)
  end if

Algorithm 3 Solver::Decompose()

Input: C
  Decide an ON set of terms T.
  for all $t_i \in T$ **do**
    Compute reduced problem $C_i = C/t_i$
    if no conflict **then**
      return $Solve(C_i)$
    end if
  end for

Algorithm 4 Solver::Decompose-Distribute()

Input: C
  Decide an ON set of terms T.
  for all $t_i \in T$ **do** ▷ distribute each subproblem to different node
    Compute reduced problem $C_i = C/t_i$
    if no conflict **then**
      return $Solve(C_i)$
    end if
  end for

Appendix B

Reductions to SAT

B.1 Introduction

SAT is an NP-Complete problem. Therefore, all NP problems are reducible to SAT. This section covers the reductions of some well known hard problems to the SAT problem.

B.2 Clique problem

In graph theory, a clique in an undirected graph is a subset of its vertices such that every two vertices in the subset are connected by an edge. The clique problem which is solved here is the decision problem of testing whether a graph contains a clique larger than a given size.

Problem: Determine whether a graph G on n vertices has a clique of size k

Solution [10]:

Convert the graph to a CNF using the following rules:

Variables:

$y_{i,r}$ (true if node i is the r^{th} node of the clique) for $1 \leq i \leq n$, $1 \leq r \leq k$.

Clauses:

- For each r, $y_{1,r} \vee y_{2,r} \vee \ldots \vee y_{n,r}$ (some node is the r^{th} node of the clique).
- For each i, $r \leq s$, $\neg y_{i,r} \vee \neg y_{i,s}$ (no node is both the r^{th} and the s^{th} node of the clique).
- For each $r \neq s$ and $i < j$ such that (i, j) is not an edge of G, $\neg y_{i,r} \vee \neg y_{j,s}$. (If there's no edge from i to j then nodes i and j cannot both be in the clique).

B.3 Hamiltonian path problem

A Hamiltonian path is a path in an undirected or directed graph that visits each vertex exactly once.

Problem: Determine whether a graph G on n vertices has a Hamiltonian path

Solution: Convert the graph to a CNF using the following rules:

Variables:

$y_{i,r}$ (true if node i is the r^{th} node of the path) for $1 \leq i, r \leq n$.

Clauses:

- For each r, $y_{1,r} \vee y_{2,r} \vee \ldots \vee y_{n,r}$ (some node is the r^{th} node of the path).
- For each i, $r \leq s$, $\neg y_{i,r} \vee \neg y_{i,s}$ (no node is both the r^{th} and the s^{th} node of the path).
- For each $i \neq j$ and $r < n$ such that (i, j) is not an edge of G, $\neg y_{i,r} \vee \neg y_{j,r+1}$. (If there's no edge from i to j then they cannot be consecutive nodes in the path).

The above solution can be used to reduce the Hamiltonian cycle problem to SAT by replacing the clause $\neg y_{i,n-1} \vee \neg y_{j,n}$ by the clause $\neg y_{i,n-1} \vee \neg y_{j,1}$.

B.4 Future work

More interesting and hard problems like the Travelling Salesman Problem and the Discrete Logarithm Problem can be looked at.

Bibliography

[1] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36 (1):42–47, Feb 2011. URL `http://www.gnu.org/s/parallel`.

[2] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-20851-8. doi: 10.1007/978-3-540-24605-3_37. URL `http://dx.doi.org/10.1007/978-3-540-24605-3_37`.

[3] Youssef Hamadi and Lakhdar Sais. Manysat: a parallel sat solver. *JOURNAL ON SATISFIABILITY, BOOLEAN MODELING AND COMPUTATION (JSAT)*, 6, 2009.

[4] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962. ISSN 0001-0782. doi: 10.1145/368273.368557. URL `http://doi.acm.org/10.1145/368273.368557`.

[5] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM. ISBN 1-58113-297-2. doi: 10.1145/378239.379017. URL `http://doi.acm.org/10.1145/378239.379017`.

[6] João P. Marques Silva and Karem A. Sakallah. GRASP-;a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7597-7. URL `http://dl.acm.org/citation.cfm?id=244522.244560`.

[7] Lintao Zhang, Conor F. Madigan, and Matthew H. Moskewicz. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

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

[9] Luís Gil, Paulo Flores, and Luís Miguel Silveira. Pmsat: a parallel version of minisat. *Journal on Satisfiability, Boolean Modeling and Computation*, page 2008.

[10] Lance Fortnow. Reductions to SAT. URL `http://blog.computationalcomplexity.org/2006/12/reductions-to-sat.html`.

Acknowledgements

I would like to thank ...

Signature: .....

Sahil Agarwal

10D070017

Date: ..... June 2015