HyParSAT: A Hybrid Parallel Complete SAT Solver Using Parallel Java 2

By

Jiten Patel

Email: jrp5375@rit.edu

A Masters Project report submitted in partial fulfillment of the requirements for the degree of Master of Science Rochester Institute of Technology 2013

Committee
Chairperson of the Supervisory Committee: Professor Alan Kaminsky
Reader of the Supervisory Committee: Professor Edith Hemaspaandra
Observer of the Supervisory Committee: Professor Zack Butler

Department of Computer Science Rochester Institute of Technology
ABSTRACT

HyParSAT: HYBRID PARALLEL COMPLETE SAT SOLVER USING PARALLEL JAVA 2

In this project, I have proposed and thoroughly discussed HyParSAT which is a new hybrid approach of a parallel complete SAT solver for shared memory multiprocessor (SMP) parallel computers. It is highly configurable and scalable with respect to the number of available processors. Almost all of the existing parallel SAT solvers designed for SMP computers, uses either a portfolio approach or a divide and conquer approach. HyParSAT combines both approaches in order to achieve benefits from both. The main solver behaves as a grand-master and consists of four independent portfolios that are designed to run asynchronously. Each of these four independent portfolios behaves as a master, and spawns multiple slave processes each of which runs the Conflict Driven Clause Learning (CDCL) sequential SAT solving algorithm. Moreover, each portfolio uses a guiding path divide and conquer strategy to split the problem’s search space into multiple pseudo non-overlapping search space chunks that are assigned to slave processes using a dynamic scheduling scheme to manage load balancing. Each slave process treats the assigned search space chunk as an independent SAT instance, and is now forced to search for a solution in a different part of the problem search space.

HyParSAT has been implemented in 100% Java using Parallel Java 2 (PJ2) API. Its performance has also been compared with MiniSat sequential solver as well as ManySAT parallel solver. Performance of HyParSAT has been evaluated on various structured industrial SAT instances selected from SAT Race 2010 and some other resources.

Keywords: HyParSAT, Parallel Computing, PJ2, DPLL, CDCL, ManySAT, zChaff, and MiniSat.
Acknowledgement

I would like to express my deepest gratitude to my project adviser, Dr. Alan Kaminsky, for his essential guidance and encouragement. I am thankful to him for generously accepting me as his student. He has always been there to answer my questions and discuss the issues regarding the design and implementation phases of my work. This project would have never been possible without his invaluable advice and help.

I would also like to thank Dr. Edith Hemaspaandra and Dr. Zack Butler for accepting the request of being my project committee members. I would have never been able to successfully finish my capstone project proposal and final report without their valuable feedbacks.

I would like to express my hearty gratitude to my graduate adviser, Dr. Hans-Peter Bischof and rest of the office staff of computer science department for helping me throughout my master’s degree at Rochester Institute of Technology.
# Table of Contents

1. Introduction ........................................................................................................... 7

2. SAT Problem and SAT Solvers........................................................................... 8
   2.1 SAT Problem.................................................................................................... 9
   2.2 SAT Solvers.................................................................................................... 9
      2.2.1 Complete SAT Solvers........................................................................ 9
      2.2.2 Incomplete SAT Solvers.................................................................... 10

3. Implementation Aspects of CDCL Algorithm............................................... 10
   3.1 Boolean Constraint Propagation (BCP) .................................................. 11
   3.2 Branching Heuristics................................................................................. 11
   3.3 Clause Leaning............................................................................................ 12
   3.4 Random Restart.......................................................................................... 12
   3.5 Restricted Clause Learning and Clause Deletion Strategies............... 13
   3.6 Non-Chronological Back-Tracking......................................................... 13

4. Parallel Complete SAT Solvers....................................................................... 14

5. HyParSAT: Hybrid parallel complete SAT solver.................................... 15
   5.1 System architecture..................................................................................... 15
   5.2 Diversity of Portfolios................................................................................. 17
   5.3 Master-Slave Design Using Parallel Java 2 (PJ2) .................................... 18
   5.4 Load balancing............................................................................................ 19
   5.5 Data Structures............................................................................................ 20
      5.5.1 Literal.................................................................................................... 20
      5.5.2 Clause................................................................................................... 21
      5.5.3 Assignment Vector............................................................................... 21
      5.5.4 CNFFormula: The Global Clause Database........................................ 22
      5.5.5 Solver Configuration............................................................................ 22
      5.5.6 RANDBranchingHeuristic............................................................... 23
      5.5.7 VSIDSPriorityQueue.......................................................................... 24
      5.5.8 Global Variables.................................................................................. 24
   5.6 CDCL Algorithm of HyParSAT............................................................... 25
   5.7 Important Functions..................................................................................... 25
      5.7.1 startSlave............................................................................................. 26
      5.7.2 preAssignment..................................................................................... 26
      5.7.3 getNextBranchingVariable............................................................... 26
      5.7.4 booleanConstraintPropagation......................................................... 26
      5.7.5 setVar................................................................................................... 27
      5.7.6 unsetVar............................................................................................... 27
      5.7.7 analyzeConflict................................................................................... 27
      5.7.8 learnConflictClause............................................................................ 27
      5.7.9 calculateBacktrackingLevel.............................................................. 29
      5.7.10 saveLearnedClause.......................................................................... 29
5.7.11 nonChronologicalBacktracking .................................................... 29
5.7.12 deleteNonAssertingBlackClauses .................................................... 29
5.7.13 reset ................................................................................................ 29

6 Performance evaluation and comparison .............................................. 29
  6.1 Effect of Each Portfolio, Clause Database Size, and Core Count .... 30
  6.2 Performance Comparison with Other Solvers ............................... 31

7 Future work ......................................................................................... 33

8 Related work ....................................................................................... 34

9 Conclusion ........................................................................................... 35

10 References .......................................................................................... 36
### List of Images and Tables

<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>CDCL solver</td>
</tr>
<tr>
<td>13</td>
<td>Non-Chronological Backtracking</td>
</tr>
<tr>
<td>16</td>
<td>HyParSAT Architecture</td>
</tr>
<tr>
<td>17</td>
<td>Diversity of All Four Portfolios</td>
</tr>
<tr>
<td>18</td>
<td>Creation of Two Asynchronous Portfolios Using PJ2</td>
</tr>
<tr>
<td>18</td>
<td>Creation of Slave Processes and Load Balancing Using PJ2</td>
</tr>
<tr>
<td>19</td>
<td>Different Scheduling schemes available in PJ2</td>
</tr>
<tr>
<td>20</td>
<td>Literal Class</td>
</tr>
<tr>
<td>21</td>
<td>Clause Class</td>
</tr>
<tr>
<td>22</td>
<td>AssignmentVector Class</td>
</tr>
<tr>
<td>22</td>
<td>CNFFormula Class</td>
</tr>
<tr>
<td>23</td>
<td>SolverConfiguration Class</td>
</tr>
<tr>
<td>23</td>
<td>Attributes of config.txt</td>
</tr>
<tr>
<td>23</td>
<td>RANDBranchingHeuristic Class</td>
</tr>
<tr>
<td>24</td>
<td>VSIDSPriorityQueue Class</td>
</tr>
<tr>
<td>24</td>
<td>Attributes of Slave Classes</td>
</tr>
<tr>
<td>25</td>
<td>CDCL Algorithm Used in HyParSAT</td>
</tr>
<tr>
<td>26</td>
<td>Pre-assignment of Variables</td>
</tr>
<tr>
<td>28</td>
<td>Implication Graph and 1-UIP Scheme</td>
</tr>
<tr>
<td>31</td>
<td>Effect of Each Portfolio on Overall Performance of HyParSAT</td>
</tr>
<tr>
<td>32</td>
<td>Performance Comparison of HyParSAT with Other Solvers</td>
</tr>
</tbody>
</table>
1. Introduction

The Boolean Satisfiability Problem (SAT) was the first known NP-complete problem and is one of the most-researched problems in Computer Science. A wide range of other decision and optimization NP problems can be transformed into SAT instances so that SAT solvers can efficiently solve many SAT instances which are useful in various areas such as circuit design and automatic theorem proving. SAT solvers have also been directly useful in various areas of computer science including theoretical computer science, artificial intelligence, hard-ware design, electronic design automation, and verification.

The Davis-Putnam-Logemann-Loveland (DPLL) [1] algorithm is one of the oldest complete backtracking-based search algorithms for solving SAT problems, and it has been the basis for most of the modern more efficient sequential SAT solvers [2]. In the last two decades, many improvements on all levels such as logic over calculi, heuristics, data structures, and low-level processes have been realized in various existing SAT solvers [18]. Despite these several advancements introduced in DPLL by recent algorithms many important SAT problems are still beyond the reach of present sequential SAT solvers. The inherent complexity of the SAT limits the possibility of improvements in sequential algorithms. Parallelization of the SAT solvers seems to be the only potential approach to break the performance barrier. In last two decades, parallelization of SAT solvers has made it possible to solve SAT instances with 1,000,000+ variables and 5,000,000+ constraints within a few minutes [3].

CDCL is a DPLL based highly efficient backtracking algorithm that is used for solving SAT problems. Most of the existing parallel SAT solvers for SMP computers use the standard CDCL algorithm as a starting point followed by partitioning the search space effectively using some divide-and-conquer technique. The parallel solvers in another category use a portfolio of sequential SAT solvers, each of which would run CDCL based independent sequential solver on the same SAT instance. A solver in an individual thread could perform knowledge exchange with others in order to improve the performance globally [4].

In this report, a Hybrid Parallel SAT (HyParSAT) solver has been proposed and thoroughly discussed that combines the portfolio approach with a guiding path divide and conquer technique. HyParSAT implements most of the modern SAT solver features such as 1-UIP [5] clause learning, shared clause database, smart clause sharing, random restart, optimized Boolean constraint propagation (BCP) scheme using two watch literals [5] data-structure, Variable State Independent Decaying Sum (VSIDS) [5] branching heuristics, non-chronological backtracking [12], and local clause deletion. It is highly configurable to adapt the available number of CPU cores and dynamically load balanced to achieve the highest degree of parallelism. The objectives of this project are the following:

a. Design and implement HyParSAT in 100% Java using PJ2 [9] API, which is a hybrid approach of CDCL based parallel SAT solver.

b. Test its functional correctness on the industrial benchmark SAT instances selected from SAT Race 2010 [19].
c. Observe the effects of each portfolio of HyParSAT on the overall performance of the solver.

Section-2 discusses some preliminaries about SAT problem and solvers. Section-3 briefly explains the design and implementation aspects of modern CDCL solvers. In section-4 report discusses various approaches of parallelism implemented by modern parallel complete SAT solvers. Section-5 presents the HyParSAT architecture and design choices I made. Section-6 covers the performance evaluation and comparison of HyParSAT with MiniSat and ManySAT solvers. Section-7 describes the future work to enhance the performance of HyParSAT. Section-8 provides the overview of few state of the art SAT solvers. Section-9 concludes this report and points out some significant perspectives on SAT solvers. In rest of the report SAT solver is referred to as simply “solver”.

2. SAT Problem and SAT Solvers

Boolean or propositional satisfiability, also commonly known as SATISFIABILITY (abbreviated as SAT) was the first known NP-complete problem [7]. Since it is strongly believed (but not proven) that $P \neq NP$ [8], it would not be an overstatement to say that an efficient algorithm which solves all the SAT problems in polynomial time simply does not exist. Before formally defining the SAT problem, one must first understand certain terminologies.

**Boolean Expression:** It is an expression built from variables, operators AND/conjunction ($\land$), OR/disjunction ($\lor$), NOT/negation ($\neg$), and parentheses.

**Literal:** It is either a variable (positive literal) or the negation of a variable (negative literal). For example, “$x_1$” is a positive literal whereas “$\neg x_1$” is a negative literal.

**Clause:** It is a disjunction of one or more literals. For example; “$x_1$”, “$\neg x_2$”, “$x_1 \lor \neg x_2$”, and “$x_1 \lor \neg x_2 \lor \neg x_3$” are clauses of different sizes.

**Conjunctive Normal Form (CNF) formula:** It is a Boolean expression containing conjunction of clauses. For example; $(x_1 \lor \neg x_2) \land (x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_3)$ is a CNF Boolean formula. Every propositional logic formula can be transformed into an equivalent CNF using Boolean algebra laws. It has been proven that every decision problem in the complexity class NP can be transformed into an instance of SAT problem for CNF [7]. The SAT problem with the most $k$ literals in each of its clause is called a $k$-SAT problem. Surprisingly the 2-SAT problem belongs to complexity class NL-complete [10] hence can be solved in polynomial time. 3-SAT problem is NP-complete and this can be generalized theoretically for any SAT problem with $k \geq 3$. HyParSAT is designed to solve SAT problems with arbitrary $k$ values in CNF form.

**Truth Assignment:** It is a mapping $f$ that assigns 1 (TRUE) or 0 (FALSE) to each variable in its domain; it shall enumerate all the variables in the domain as $(x_1, x_2, ..., x_n)$. Truth assignment satisfies a clause if and only if it maps at least one literal in the clause to 1. Truth assignment satisfies a CNF formula if and only if it satisfies all of its clauses.

**Unit Literal Rule:** If an unsatisfied clause contains all but one literal set to FALSE, the one remaining literal must be set to TRUE in order to satisfy this clause.
**Conflict Rule:** If all the literals in a clause have been set to FALSE, this clause is unsatisfied and so the entire CNF cannot be satisfied along with the current assignment path.

### 2.1 SAT Problem

The *Boolean satisfiability* (SAT) problem refers to the question that: “Given a Boolean formula, determine if there exists a truth assignment that satisfies the entire formula.” Find such an assignment if it does exist, and only then can the problem be considered *satisfiable*. Otherwise, this formula can be considered *unsatisfiable*.

### 2.2 SAT Solvers

The algorithms used to solve the SAT problems are called SAT solvers. Since SAT is NP-complete problem the algorithms with only exponential worst-case running time complexity are known. Despite of the inherent complexity of SAT problems various highly efficient and scalable solvers have been developed over the last two decades and have increased our ability to solve SAT problems with a few hundred variables and millions of clauses. SAT solvers are categorized in mainly two classes discussed in the following two subsections.

#### 2.2.1 Complete SAT Solvers

The solvers in this category solve SAT problems with 100% certainty. Given enough computing resources, complete solvers are guaranteed to terminate with the correct decision on the satisfiability or unsatisfiability of the given SAT problem. Almost all of the modern solvers in this category are inherited from the DPLL solver which employs a systematic backtracking search procedure that explores the exponentially sized space of variable assignments looking for satisfying assignments. Today, complete solvers are designed based on either “Conflict-driven” or “Look-ahead” approach [11].

The conflict-driven solvers implement DPLL along with conflict analysis, and that is why they are more commonly known as Conflict Driven Clause Learning (CDCL) SAT solvers. It has been theoretically proven that clause learning with conflict analysis does not affect the soundness and completeness of the solver. The details of conflict analysis technique with other CDCL features have been discussed in the section-3.

The Look-ahead architecture implements DPLL too along with clause learning, however, it deploys different mechanism for unit propagation directed by a look-ahead procedure. In each step the solver selects a decision variable $x$ from the CNF formula $F$. The look-ahead procedure measures the importance of $x$ and detects the possible reduction of the formula $F$ followed by backtracking to finish the look-ahead. Look-ahead is performed in a hope that an evaluation of the effect of the actual truth
assignment of variables is more reliable than only guessing based on some statistics derived from the original formula. More details about look-ahead based solvers can be found in [13].

There are two concepts associated with the CNF formula. The first one is the cost of unit propagation known as “Density” whereas the second is the global connectivity of clauses known as “Diameter” (larger diameter means more local clusters). In general CDCL solvers perform well on SAT problems with high density and large diameters whereas Look-ahead based solvers perform well on SAT problems with low density and small diameters. This is because look-ahead solvers have especially strengthened reductions and heuristics; therefore they are generally stronger than conflict-driven solvers on hard instances. Conflict-driven solvers can be much better on large instances which actually have an easy instance inside [14].

2.2.2 Incomplete SAT Solvers

The solvers in this category do not guarantee that they will report the satisfying truth assignment or proves the unsatisfiability of the problem. These solvers are usually biased on either satisfiable or unsatisfiable SAT instances. More elaborately the solvers biased on satisfiable problems either find the satisfying truth assignment or terminate with failure, but never report unsatisfiability. Alternatively, the solvers biased on unsatisfiable problems either provide a proof of unsatisfiability or terminate with failure, but never report the satisfying truth assignment. Theoretically such solvers can be incomplete with respect to both satisfiable and unsatisfiable instances [15].

These solvers are based on one of the techniques such as Stochastic Local Search (SLS), Evolutionary Algorithms (EAs) such as Genetic Algorithms (GAs) [32], translation to Integer Programming [33], and Finite Learning Automata [34]. The tremendous amount of research of such solvers has been done and many solvers have been proposed thus far. It is not surprising that incomplete solvers often perform much better than their counterpart complete solvers on very specific classes of SAT problems such as randomized SAT instances. Since this report discusses the CDCL parallel complete solver, the detailed discussion of all these incomplete solvers is out of the scope of this report. Please refer to [35] for more details about incomplete solvers.

3. Implementation Aspects of CDCL Algorithm

As we briefly discussed before, the CDCL solver is just an extension of a DPLL solver but with more sophisticated features. This section discusses important design and implementation aspects of the modern CDCL solver. Figure-1 shows the standard structure of the CDCL algorithm. One must first understand certain terminologies before going into the details of CDCL algorithm.

a. **Branching operation**: Picks a variable $x$ from the pool of unassigned variables.
b. **Decision or Branching variable**: A variable $x$ that is chosen and assigned by branching operation.
c. **Implied variable**: A variable $y$ that is forcefully assigned a truth value as a result of the unit clause rule after the assignment of some other variable $x$.

d. **Antecedent clause**: The unit clause $w$ used to imply variable $y$ is said to be the antecedent clause (or simply antecedent) of $y$. Antecedents are defined only for implied variables. Therefore, antecedent of a branching or unassigned variable is NIL.

e. **Decision stack or assignment stack**: It’s a stack of assigned variables in the order they were assigned.

f. **Decision level**: The decision level of some variable $x$ is the depth of the decision tree at which $x$ is assigned a truth value. The search procedure specifies the decision level of a branching variable which is the current depth of the binary search tree. The decision level of an implied variable is the same as of the most recently branched variable. The decision level for an unassigned variable is $-1$.

```
CDCL(ϕ, ν)
1   if (UnitPropagation(ϕ, ν) == CONFLICT)
2      then return UNSAT
3   dl ← 0 ▷ Decision level
4   while not ALLVariablesAssigned(ϕ, ν))
5      do (x, ν) = PickBranchingVariable(ϕ, ν) ▷ Decide stage
6         dl ← dl + 1 ▷ Increment decision level due to new decision
7         ν ← ν ∪ {x, ν})
8      if (UnitPropagation(ϕ, ν) == CONFLICT) ▷ Deduce stage
9         then β = ConflictAnalysis(ϕ, ν) ▷ Diagnose stage
10            if (β < 0)
11               then return UNSAT
12            else Backtrack(ϕ, ν, β)
13               dl ← β ▷ Decrement decision level due to backtracking
14 return SAT
```

**Figure-1: CDCL solver [12]**

### 3.1 Boolean Constraint Propagation (BCP)

BCP operation iteratively applies unit clause rule to identify variables which needs to be assigned a specific truth value. It continues until no more literals are implied or the conflict is identified. In Figure-1, BCP is performed by `UnitPropagation` function in line-1 and line-8. Every branching step is followed by the BCP operation to derive the logical consequences. The overall performance of the CDCL algorithms significantly depends on the efficiency of BCP since solvers spend almost 90% of their running time in BCP operation for most of the SAT problems [5]. That is why it’s very crucial for the solver’s good performance to have highly optimized BCP technique.

### 3.2 Branching Heuristic

A function `AllVariablesAssigned` checks if all the variables of CNF have been assigned a truth value. If yes, CNF is declared satisfiable. If not, a function `PickBranchingVariable` uses some heuristic to select an unassigned variable and assigns
it some truth value. Over the years numerous heuristics have been proposed, all of which have tried to improve the tradeoff between computation/memory required for the heuristics, and its ability to improve the efficiency of search.

Random (RAND) selection of branching variable is the simplest and efficient in terms of time and space complexity. There are some heuristics such as Böhm [37] and Maximum Occurrences in clauses of Minimum Size (MOMS) [38] that selects branching variable based on a maximization function of some statistics related to current variable and clause database state. Dynamic Largest Individual Sum (DLIS) [39] selects the variable with the largest frequency in unsatisfied clauses. There are a few more similar heuristics have been proposed in last two decades. For example; Dynamic Largest Combined Sum (DLCS), Jeroslow-Wang (JW) rule [40], 2-sided Jeroslow-Wang rule [41], D.S. Johnson’s (DSJ) [42] maximum satisfiability approximation algorithm, Approximate Unit-Propagation Count (AUPC) rule [43], and Circuit-Sat (CSat) rule [17].

The performance of the branching heuristics may be evaluated using statistics such as the number of branching decisions made by the solver while solving a particular problem, or the number of conflicts, however, the shorter sequence of branching decisions may actually result in more BCP operations or conflicts. Moreover, all branching heuristics have different running time and memory requirements. Apparently, all that matters is how fast the heuristic can solve the given problem. For more details about their performance comparison, I ask readers to refer to [5] [17].

3.3 Clause Learning

The purpose of clause learning is to avoid repeating the same mistakes. As soon as the BCP operation identifies a conflicting clause, the CDCL algorithm of Figure-1 calls ConflictAnalysis to deduce the reason of that conflict, and identifies the responsible variable assignments. The conjunction of these assignments always leads to the same conflict so the complement of that conjunction represents the newly learned clause. Adding such newly learned clauses to the clause database ensures that subsequent search procedure will not encounter the same conflict situations.

3.4 Random Restart

SAT solvers use a restart policy to compact the assignment stack and improve the order of assumptions. The performance of various restart policies widely depends on the type of considered SAT instances. Based on the observation of early works [4] [22], usually fast restart policy drastically improves the solver performance on industrial instances, however, slow restarts is preferred for hard instances. Usually the number of conflicts is used as the triggering event for restart so this may result in incompleteness, as there might not be sufficient enough search space (binary tree depth) available to find the solution. That is why solver needs an additional technique to guarantee its completeness. The simplest solution is to iteratively increase the cutoff value of triggering event after every restart. One thing to be noted here is that clauses learned
during the previous runs are kept in the database to avoid the same conflicts during the next runs.

3.5 Restricted Clause Learning and Clause Deletion Strategies

The CDCL solver learns and saves at least one clause after each conflict. These learned clauses consume memory. The possible number of conflicts and eventually the number of learned clauses is exponential just like the whole search space. It is experimentally observed that the average size of learned clauses tends to increase over time during the solver’s search procedure [36]. Moreover, large learned clauses do not really help to prune as much search space as shorter clauses do. Therefore, large learned clauses bring additional memory overhead without much performance enhancement. That is why solver needs to restrict the clause recording to limit the worst case growth of learned clauses to be polynomial and avoid memory overflow errors. There are various restricted clause learning techniques such as size-bound strategy, relevance-based strategy, and heuristic activity-based strategy [12]. Most of the modern solvers such as ManySAT and zChaff implement the combination of more than one of those strategies. HyParSAT implements the combination of size-bounded and relevance-based learning strategy.

3.6 Non-Chronological Backtracking

Clause learning is closely coupled with non-chronological backtracking and that is why it is also called conflict-driven backtracking. In case of classical chronological backtracking scheme solver always backtracks to the most recent decision level and flip the decision variable. Alternatively, a non-chronological scheme mostly backtracks further levels. zChaff solver proposed to always backtrack to the decision level calculated using the available learned clause information. This clause contains exactly one literal assigned at the current level. The highest decision level of the remaining other literals is considered the backtrack level. This is because the one literal assigned at the current level becomes implied literal after backtracking. In figure-2, each vertex is labeled as x=t@d which is interpreted as this: Variable x has been assigned a truth value t at the decision level d. All the edges incoming to the vertex associated with variable x are labeled with its antecedent clause w.

Figure-2: Non-Chronological Backtracking [20]
As shown in figure-2, chronological backtracking would back track to level 4 and flip the decision variable $x1$, however, this will lead to the same conflict with variable $x5$. On the other side non chronological back tracking scheme uses recently learned clause $(\neg x4 \lor x8 \lor x9)$ to decide the back track level so it backtracks to level 3 where $x4$ becomes implied literal.

4. Parallel Complete SAT Solvers

This section discusses various design aspects and approaches of parallelism that have been implemented by existing parallel complete SAT solvers so far. The architecture of parallel complete SAT solvers can be designed based on one of the two approaches explained below.

**Divide and Conquer Approach:** Almost all of the SAT solvers proposed during the early era of SAT solver parallelism were based on master-slave architecture, and implemented cooperative parallelism by splitting the problem search space. To do so they used some divide and conquer approach such as classical heuristic based partitioning, guiding-paths, or dividing the Boolean formula itself [47]. In guiding-paths strategy, the master selects a set of $k$ variables that are used for pre-assignment in order to create $2^k$ sub-problems which are distributed among the slave processes. The main problem with these approaches is the complexity of balancing the work load between multiple slaves. These solvers usually enjoy the luxury of scalability and load balancing but lack diversity.

**Portfolio Approach:** A few recently proposed solvers such as ManySAT [4] and SArTagnan. In [24] they have suggested a new competitive parallelism based portfolio approach in which the main solver runs a portfolio of multiple independent CDCL solvers which may share learned clauses with each other to improve the overall performance. Each solver in the portfolio would exploit a particular parameter set so that each solver searches different part of the entire binary tree search space. They implement different branching heuristics, clause learning scheme, and clause sharing heuristics to provide huge diversity. Generally the solvers based on portfolio approach enjoy the benefit of having diversified multiple SAT solvers; however, suffer from the lack of true parallelism and scalability.

Before the era of multi-core CPUs architectures, parallel SAT solvers were implemented using the network communication between multiple single-core architectures. Since the advent of multi-core CPUs, many solvers based on such architecture have been proposed.

**Network communication based approach:** The solvers in this category are implemented using master-slave architecture, and are designed to run on a grid of single-core processing units. These solvers can be implemented using either of the previously discussed strategies of parallelism. The solvers running in the slave processes may communicate over the network to share learned clauses. Moreover, these solvers need to deal with the tradeoff between the number of learned clauses shared and the overall
performance [47]. Therefore, the slave processes must share their learned clauses with others very wisely.

**Share-memory based approach:** The solvers in this category are designed to run on share-memory multi CPUs processing systems, and they can be implemented using either portfolio or divide and conquer approach. Since all the parallel processes of a solver share the same limited memory, clause sharing has to be controlled very wisely in order to achieve efficiency and scalability. That is why this solvers need to address questions such as which learned clauses should be shared and when shall they be deleted. Furthermore, theoretically memory access is uniform and can be done asynchronously; however, the possible cache coherence may slow down the memory access due to the multi-level cache mechanism of modern multi-core processing units. While selecting the data structures and heuristics for these solvers, one must consider the effect of multi-level memory architecture of modern computers, and how the low level processes efficiently use the non-uniform memory access latencies.

**Hybrid Approach:** The performance of parallel solvers can potentially be improved furthermore by combining above two approaches where many multi-core computing systems are connected in network. PaSAT [48] implemented this approach and shown that clause learning and sharing technique dominantly affect the runtime work distribution because the different parts of the search space would be searched by different runs.

The solvers that implemented above mentioned approaches have been briefly discussed in section-8.

5. **HyParSAT: A Hybrid Parallel Complete SAT Solver**

After studying various existing sequential and parallel solvers, I came up with a new idea of combining portfolio approach with divide and conquer approach together in HyParSAT to remedy the tradeoff between scalability and diversity. The following subsections discuss the design and implementation details of HyParSAT.

5.1 **System Architecture**

HyParSAT is designed using three level master-slave architecture as depicted in figure-3. At level-1 the only object *Solver* is created which acts as a grand-master and performs three tasks in its *main*.

a. It creates an object of *SolverConfiguration* which reads the configuration attributes from config.txt. The detail of the config.txt contents is explained in section 5.5.5.

b. It creates an object of *CNFFormula* which reads CNF from the input file.

c. It calls *selectPortfolio* method to start one/more/all of the four asynchronous portfolios: *ManySATportfolio1*, *ManySATportfolio2*, *zChaffportfolio1* and *zChaffPortfolio2*. These portfolios are started by calling their *startPortfolio*
method. Whether to run one/more/all of the portfolios is decided based on the value of the portfolioSelection attribute of config.txt.

Let us assume that the Solver object started all four portfolios. The startPortfolio method of each portfolio performs following two tasks.

a. It calls the selectPreAssignedLits method to pre-select \( k \) literals for pre-assignment. Each literal can be assigned TRUE/FALSE, two possible values that create \( 2^k \) sub-problems.

b. Each portfolio creates \( n \) slave processes and distributes \( 2^k \) sub-problems among them using a dynamic scheduling mechanism (explained in section 5.3). Each slave process creates an object of corresponding slave class which implements sequential CDCL solver. The slave process uses an object of slave class to process assigned sub-problems. The CDCL solver implemented by the slave object is started by calling its startSlave method. The value of \( k \) and \( n \) may be
different for all portfolios and can be specified in the config.txt file. The startSlave method returns an appropriate message to the portfolio depending on whether it found the solution, it did not find the solution, or its execution was interrupted by some other process that found the solution.

### 5.2 Diversity of Portfolios

<table>
<thead>
<tr>
<th>Branching Heuristic</th>
<th>ManySATPortfolio1</th>
<th>ManySATPortfolio2</th>
<th>zChaffPortfolio1</th>
<th>zChaffPortfolio2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restart Policy</td>
<td>Geometric Policy</td>
<td>Arithmetic Policy</td>
<td>Geometric Policy</td>
<td>Arithmetic Policy</td>
</tr>
<tr>
<td></td>
<td>(x_{i+1} = 1.5 \times x_i), with (x_1 = 100)</td>
<td>(x_{i+1} = x_i + 16000), with (x_1 = 16000)</td>
<td>(x_{i+1} = 1.5 \times x_i), with (x_1 = 100)</td>
<td>(x_{i+1} = x_i + 16000), with (x_1 = 16000)</td>
</tr>
<tr>
<td>Literal Selection for Preassignment</td>
<td>Random Selection</td>
<td>Random Selection</td>
<td>Literal Frequency based Selection</td>
<td>Literal Frequency based Selection</td>
</tr>
</tbody>
</table>

**Figure-4: Diversity of All Four Portfolios**

Figure-4 explains the diversity of all four portfolios. All portfolios are similar in a way that they all implement the same 1-UIP clause learning scheme, non-chronological backtracking, two watch literals method based BCP procedure, and a combination of clause size and relevance based clause deletion policy. To provide diversity, each portfolio differs from others by three factors explained below.

**Selection of Variables for Pre-Assignment:** The zChaffPortfolios select variables with the highest frequency in the original CNF formula. It is hoped that assigning such variables at the beginning helps to solve the given instance quickly; however, such greedy selection does not always result in expected fast performance. To complement such greedy variable selection and introduce some randomness, ManySATPortfolios select variables for pre-assignment randomly.

**Branching Heuristic:** As explained in section 3.2, the nature of branching heuristic used by the solver performs a key role in the overall performance of the backtracking search procedure. RAND [44] chooses the next branching variable randomly whereas VSIDS chooses the variable with highest counter as explained in [5]. Experiment results in [17] have shown that despite the simplest stochastic nature of RAND, it can perform very well on randomly generated SAT problems. To favor such instances, HyParSAT implements RAND in ManySATPortfolios. Since hard (not necessarily large) SAT instances tend to generate many conflicts, the conflict clauses dominantly drive solver’s search process. VSIDS try to satisfy the most recent conflicts first and that’s why it specifically outperforms other heuristics while solving such hard instances. VSIDS has been implemented and its high performance has been proven by many modern solvers. This is why I decided to implement it in zChaffPortfolios.

**Restart Policy:** As explained in section 3.4, the fast restart policy performs well on structured industrial instances whereas the slow restart policy does well on hard instances. ManySATPortfolio1 and zChaffPortfolio1 implement fast geometric restart policy \(x_{i+1} = 1.5 \times x_i\), with \(x_1 = 100\) [22] where \(x\) is the number of conflicts after which the restart should take place. ManySATPortfolio2 and zChaffPortfolio2 implement
the slow arithmetic restart policy \( x_{i+1} = x_i + 16000 \), with \( x_1 = 16000 \) [4], where \( x \) is the number of conflicts after which the restart should take place.

### 5.3 Master-Slave Design Using Parallel Java 2 (PJ2)

The master-slave architecture explained in section-5.1 is realized using PJ2 [9] API. Solver class inherits Task class to be able to create asynchronous portfolios. Here the \( parallelDo \) method inherited from Task creates four parallel sections, each of which creates and starts one portfolio in a separate new thread as shown in the code snippet of figure-5.

The Solver object is passed to all portfolios so they can create parallel slave processes and distribute sub-problems to them dynamically as shown below. One thing to be noted here is each slave process creates an object of slave class only once and uses it to process all the sub-problems assigned to it; however, the object of slave class needs to be reset after processing every sub-problem to make the object ready to process the next sub-problem. The code snippet shown in figure-6 is taken from ManySATPortfolio1 but the code for other portfolios is similar as this.

```java
solver.parallelDo(
    new Section(){
        public void run() throws Exception{
            new ManySATPortfolio1(solver)).startPortfolio();
        }
    },
    new Section(){
        public void run() throws Exception{
            (new ManySATPortfolio2(solver)).startPortfolio();
        }
    },
    new Section(){
        public void run() throws Exception{
            (new zChaffPortfolio1(solver)).startPortfolio();
        }
    },
    new Section(){
        public void run() throws Exception{
            (new zChaffPortfolio2(solver)).startPortfolio();
        }
    }
);
```

**Figure-5: Creation of Two Asynchronous Portfolios Using PJ2**

```java
public void startPortfolio() throws Exception{
    //preselecting some variables
    selectPreAssignedLits();
    long lb =0, ub = (1<<preAssignedCount)-1;
    ManySATSlave1.setMasterAndSolver(this,solver);
}
```
// Creating Chunks, processes and schedule tasks using dynamic scheduling
solver.parallelFor(lb,ub).threads(slaveCount).schedule(Solver.dynamic).
exec(new LongLoop(){
    // Thread-local variable declarations
    ManySATSlave1 slave;
    public void start(){
        slave = new ManySATSlave1();
    }
    public void run (long i) throws Exception{
        if(!solver.isSolved()){
            System.out.format("|%-70s |n","ManySATPortfolio1: Slave["+rank()+"]:chunk["+i+ "]: Started");
            String message = slave.startSlave(i,rank());
            if(!solver.isSolved())
                slave.reset();
            System.out.format("|%-70s |n","ManySATPortfolio1: Slave["+rank()+"]:chunk["+i+ "]: Stopped" +message);
        }
    }
    if (!solver.isSolved())
        solver.setSolved(false);
}

Figure-6: Creation of Slave Processes and Load Balancing Using PJ2

5.4 Load Balancing

Figure-7: Different Scheduling schemes available in PJ2 [9]
As shown in the figure-6 code snippet, the `parallelFor` method of `Task` class creates `slaveCount` number of slave processes, sets the scheduling scheme, creates chunks, and distributes these chunks among the slave processes. The execution time required to solve a sub-problem depends on various factors such as the number of conflicts faced, the nature of the problem (easy, hard, structured, or random), or the size of back-jumps. In short, there is a huge variation between the execution time required to solve different sub-problems of the given SAT instance. Therefore from the load balancing perspective uniform distribution of the chunks among all the slave processes is not a good idea. That is why HyParSAT uses a dynamic scheduling mechanism to improve the load balancing between all the slave processes.

As shown in the figure-7 it is very obvious that a fixed schedule is recommended only if the execution time of all the iterations of a loop is theoretically the same, otherwise dynamic or guided scheduling scheme is recommended. I have tested both later schemes for HyParSAT; however, I have used only a dynamic scheduling scheme for HyParSAT because their performance difference is too arbitrary and negligible.

### 5.5 Data Structures

HyParSAT implements its own utility wrapper classes using data-structures provided in Java Collection Framework in order to make the its implementation easy. The following subsections provide the overview of all the utility classes and other data-structures used in the CDCL solver of slave classes. Please refer to the actual code/Javadocs for the description of all methods.

#### 5.5.1 Literal

```
<table>
<thead>
<tr>
<th>Literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>- id : int</td>
</tr>
<tr>
<td>- depth : int</td>
</tr>
<tr>
<td>- antecedent : int</td>
</tr>
<tr>
<td>- isSatisfied : boolean</td>
</tr>
<tr>
<td>- impliedValue : boolean</td>
</tr>
<tr>
<td>- watchingClausesPos : LinkedList&lt;Integer&gt;</td>
</tr>
<tr>
<td>- watchingClausesNeg : LinkedList&lt;Integer&gt;</td>
</tr>
</tbody>
</table>

+Literal() |
+reset() : void |
+getId() : int |
+setAntecedent() : void |
+getAntecedent() : int |
+setDepth() : void |
+getDepth() : int |
+addWatchingClause() : void |
+removeWatchingClause() : boolean |
+getWatchingClausesIterator() : Iterator<Integer> |
+isSatisfied() : boolean |
+getImpliedValue() : boolean |
+setImplied() : void |
+unsetImplied() : void |
+toString() : String |
```

**Figure-8: Literal Class**
The data structure shown in figure-8 holds the information specific to individual literals while running CDCL solver in each slave process.

a. **id**: A unique id number.
b. **depth**: The decision level at which this literal was assigned/implied.
c. **antecedent**: The antecedent clause id for implied literals.
d. **isImplied**: The Boolean value indicating if this literal has been implied.
e. **impliedValue**: The Boolean value representing the implied value of the literal.
f. **watchingClausesPos**: A linked list of clause ids watching the positive literal.
g. **watchingClausesNeg**: A linked list of clause ids watching the negative literal. I decided to use a linked list because add/remove operations are performed frequently whereas a look-up operation is not performed at all.

5.5.2 Clause

<table>
<thead>
<tr>
<th>Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>-id : int</td>
</tr>
<tr>
<td>-WL : int[]</td>
</tr>
<tr>
<td>-clause : ArrayList&lt;Integer&gt;</td>
</tr>
<tr>
<td>+Clause()</td>
</tr>
<tr>
<td>+reset() : void</td>
</tr>
<tr>
<td>+getID() : int</td>
</tr>
<tr>
<td>+getClause() : ArrayList&lt;Integer&gt;</td>
</tr>
<tr>
<td>+getWL() : int</td>
</tr>
<tr>
<td>+setWL() : void</td>
</tr>
<tr>
<td>+getLit() : int</td>
</tr>
<tr>
<td>+isWL() : boolean</td>
</tr>
<tr>
<td>+getWLIndex() : int</td>
</tr>
<tr>
<td>+size() : int</td>
</tr>
<tr>
<td>+toString() : String</td>
</tr>
</tbody>
</table>

**Figure-9: Clause Class**

The data structure shown in figure-9 holds the information specific to individual clauses while running CDCL solver in each slave process.

a. **id**: A unique id number.
b. **WL**: The references for two Watch literals.
c. **clause**: The list of literals of the given clause.

5.5.3 Assignment Vector

The assignment vector shown in the figure-10 is the wrapper class around the `Java.util.BitSet` class. Its index range is from 1 to n, where n is the number of variables in CNF. Assignment Vector holds mainly two kinds of information. Each CDCL solver running in individual slave processes maintains exactly one instance of assignment vector.

a. **Odd bits**: The Boolean value that indicates if a particular variable is assigned or not.
b. **Even Bits**: The truth assignment of variables.
5.5.4 CNFFormula: The Global Clause Database

As shown in the figure-11, CNFFormula is the wrapper class around an `ArrayList<ArrayList<Integer>>` combined with `HashSet<ArrayList<Integer>>`. `ArrayList` stores all the clauses in original CNF in addition to the learned clauses reported by individual CDCL solvers running in each of the slave processes. However, the `HashSet` is used to efficiently check the redundancy of learned clauses. `HashSet<Integer>` is used to store literals from the unit clauses. These literals are assigned their truth values before the CDCL solver starts its search procedure.

5.5.5 Solver Configuration

`SolverConfiguration` is a wrapper class around a `HashMap<String>` which holds the information related to the configuration of the entire HyParSAT solver. This information affects the factors such as which portfolios should be run, the number of
slaves for each portfolios, input CNF file, etc. Table in the figure-13 lists of all the attributes used in config.txt file. Please refer to the config.txt file for details about each attributes and what their specific value represents.

<table>
<thead>
<tr>
<th><strong>SolverConfiguration</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>-config : HashMap&lt;String,String&gt;</td>
</tr>
<tr>
<td>+SolverConfiguration()</td>
</tr>
<tr>
<td>+toString() : String</td>
</tr>
<tr>
<td>+getAttrValue() : String</td>
</tr>
</tbody>
</table>

**Figure-12: SolverConfiguration Class**

<table>
<thead>
<tr>
<th><strong>cnfFileName</strong></th>
<th>Input CNF file name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>portfolioSelection</strong></td>
<td>Decides which portfolios need to be created</td>
</tr>
<tr>
<td><strong>ManySATPRNGSeed</strong></td>
<td>PRNG seed for ManySATPortfolio</td>
</tr>
<tr>
<td><strong>zChaffPRNGSeed</strong></td>
<td>PRNG seed for zChaffPortfolio</td>
</tr>
<tr>
<td><strong>ManySATSlaveCount</strong></td>
<td>Slave processes count for ManySATPortfolio</td>
</tr>
<tr>
<td><strong>zChaffSlaveCount</strong></td>
<td>Slave processes count for zChaffPortfolio</td>
</tr>
<tr>
<td><strong>ManySATPreAssignedCount</strong></td>
<td>Preassignment literal count for ManySATPortfolio</td>
</tr>
<tr>
<td><strong>zChaffPreAssignedCount</strong></td>
<td>Preassignment literal count for zChaffPortfolio</td>
</tr>
<tr>
<td><strong>LearnedClauseSizeCutoff</strong></td>
<td>Cutoff value for learned clauses’ size</td>
</tr>
<tr>
<td><strong>ManySATRestartTrigger</strong></td>
<td>Starting value for restart policy of ManySATPortfolio</td>
</tr>
<tr>
<td><strong>zChaffRestartTrigger</strong></td>
<td>Starting value for restart policy of zChaffPortfolio</td>
</tr>
<tr>
<td><strong>zChaffDecayRate</strong></td>
<td>VSIDS decaying rate for zChaffPortfolio</td>
</tr>
<tr>
<td><strong>TotalLearnedClauseDBSize</strong></td>
<td>Size of the Learned Clauses global database</td>
</tr>
</tbody>
</table>

**Figure-13: Attributes of config.txt**

5.5.6 RANDBranchingHeuristic

It implements RAND branching heuristic that selects the next branching variable randomly. This heuristic can be calculated in $O(n)$ space complexity and $O(1)$ running time complexity. Experiment results in [17] have shown that despite the simplest stochastic nature of RAND, it can perform very well on certain SAT problems.

<table>
<thead>
<tr>
<th><strong>RANDBranchingHeuristic</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+unassignedLitCount : int</td>
</tr>
<tr>
<td>+litArray : int[]</td>
</tr>
<tr>
<td>+litIndex : int[]</td>
</tr>
<tr>
<td>-prng : Random</td>
</tr>
<tr>
<td>+RANDBranchingHeuristic()</td>
</tr>
<tr>
<td>+reset() : void</td>
</tr>
<tr>
<td>+markUnset() : void</td>
</tr>
<tr>
<td>+markSet() : void</td>
</tr>
<tr>
<td>+getRANDLit() : String</td>
</tr>
<tr>
<td>+swapLits() : void</td>
</tr>
</tbody>
</table>

**Figure-14: RANDBranchingHeuristic Class**
5.5.7 VSIDSPriorityQueue

It implements Variable State Independent Decaying Sum (VSIDS) [5] branching heuristic. A priority queue can be maintained for unassigned literals based on their counters in order to quickly (log n time) select the decision variable. According to our observation, every time a variable is assigned/unassigned, it needs to be removed/added (log n time each) to the priority queue. In most of the SAT problems, operations of assigning and unassigning variables are performed way more frequently than branching therefore maintenance of priority queue is the overhead needed to deal with. This is why I have used a TreeSet data structure in which case literals needs to be removed and added back to it only when a new clause is added to the database. The next branching variable is the first unassigned variable from the set. It derives the next branching variable in $O(n)$ running time; however, practically in much less time because of the nature of the VSIDS heuristic.

<table>
<thead>
<tr>
<th>VSIDSPriorityQueue</th>
</tr>
</thead>
<tbody>
<tr>
<td>-queue : TreeSet</td>
</tr>
<tr>
<td>+VSIDSPriorityQueue()</td>
</tr>
<tr>
<td>+reBuildVSIDSPriorityQueue() : void</td>
</tr>
<tr>
<td>+updateAll() : void</td>
</tr>
<tr>
<td>+update() : void</td>
</tr>
<tr>
<td>+getNext() : String</td>
</tr>
</tbody>
</table>

Figure-15: VSIDSPriorityQueue Class

5.5.8 Global Variables

The attributes shown in figure-16 are the class variables maintained by the sequential CDCL solver running in each slave process.

<table>
<thead>
<tr>
<th>master</th>
<th>A reference to its master object</th>
</tr>
</thead>
<tbody>
<tr>
<td>solver</td>
<td>A reference to the Solver object</td>
</tr>
<tr>
<td>litList</td>
<td>A list of Literal objects</td>
</tr>
<tr>
<td>clsList</td>
<td>An list of Clause objects</td>
</tr>
<tr>
<td>assignment</td>
<td>An object of the AssignmentVector</td>
</tr>
<tr>
<td>assignedLits</td>
<td>A stack of assigned literals</td>
</tr>
<tr>
<td>globalUclQueue</td>
<td>A queue of unit clause literals</td>
</tr>
<tr>
<td>blackClauses</td>
<td>A list of references</td>
</tr>
<tr>
<td>contradictoryUC</td>
<td>A flag to indicate a contradiction</td>
</tr>
<tr>
<td>conflictingClause</td>
<td>Conflicting clause id</td>
</tr>
<tr>
<td>impliedLit</td>
<td>Implied literal as a result of recently learned clause</td>
</tr>
<tr>
<td>antecedentCls</td>
<td>A clause id of recently learned clause</td>
</tr>
<tr>
<td>depth</td>
<td>Current decision level</td>
</tr>
<tr>
<td>unassignedLitCount</td>
<td>A count of unassigned literals</td>
</tr>
<tr>
<td>clauseCount</td>
<td>The current clause count</td>
</tr>
<tr>
<td>prevClauseCount</td>
<td>The previous clause count before the last reset()</td>
</tr>
<tr>
<td>cutoff</td>
<td>The cutoff value for learned clause size</td>
</tr>
</tbody>
</table>
### 5.6 CDCL Algorithm of HyParSAT

The CDCL algorithm shown in the figure-17 is implemented by both of the slave classes. It is similar to the one shown in the figure-1 but with some additional functionality. In line-5 it checks the `isSolved` flag of the `Solver` object to see if solution has been found by some other slave process. In line 15-17 random restart takes place. Lines 18-19 are part of only `zChaffPortfolio` because it implements the step-5 of VSIDS branching heuristic. Details of rest of the functions used in CDCL algorithm are explained in the following subsections. Please refer to the source code for detailed implementation of each function to get a better idea about it.

#### 5.7 Important Functions

This section discusses about all the functions used by each slave CDCL solver.
5.7.1 startSlave

This method is invoked by the portfolio object to start the slave object’s CDCL solver to solve the current sub-problem \( i \) which is passed to it as a parameter. To do so, this method invokes \( \text{preAssignment} \) and \( \text{CDCL} \) methods.

5.7.2 preAssignment

This method performs the pre-assignment of the variables selected by the portfolio object, and it also assigns unit clause literals at depth 0. It converts the integer value \( i \) into its binary representation with a length equal to the number of variables selected for pre-assignment. These variables are pre-assigned to TRUE/FALSE according to their corresponding binary bit values (TRUE for 1 and FALSE for 0). An example shown in the figure-18 explains this well.

<table>
<thead>
<tr>
<th>( i=100 ) and ( \text{preAssignmentCount}=10 )</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary representation</td>
<td>0</td>
</tr>
<tr>
<td>variable</td>
<td>70</td>
</tr>
<tr>
<td>Truth Assignment</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>24</td>
</tr>
</tbody>
</table>

**Figure-18: Pre-assignment of Variables**

5.7.3 getNextBranchingVariable

This function chooses the next decision variable from the remaining unassigned variables using branching heuristics RAND and VSIDS for \( \text{ManySATSlave} \) and \( \text{zChaffSlave} \), respectively.

5.7.4 booleanConstraintPropagation

This method performs BCP operation during preprocessing of CNF as well as after branching and backtracking. \( \text{zChaff} \) solver suggested a “Two watch literal method” [5] based highly optimized BCP engine and since then almost all of the modern CDCL solvers have used it and proven that two watch literal based BCP technique is the most efficient one known so far [5]. That is why I decided to use it for HyParSAT.

**Two Watch Literals Method:** Each clause maintains references to two of its literals as watched literals. Each literal maintains a list of clauses watching it. A clause is visited only when one of the two watched literals is assigned to FALSE. Let’s assumes a Clause \( C \) is watching two literals \( u \) and \( v \). When \( u \) is assigned to FALSE, \( C \) is visited and it holds one of the four following conditions.

1. If some literal \( w \) of \( C \) has been assigned TRUE, \( C \) is identified as satisfied.
2. If a non-watched literal \( w \) is still unassigned, \( w \) replaces \( u \) and becomes a new watch literal.
3. If all non-watched literals have been assigned to FALSE, there are two options.
3.1. If \( v \) has not been implied, \( v \) is implied now. This is the unit literal rule of BCP.

3.2. If \( v \) has been implied TRUE, it satisfies the clause and we move to next clause.

3.3. If \( v \) has been implied to FALSE while visiting some other clause, then now we have all the literals of \( C \) are assigned to False. This is the conflict rule of BCP; hence \( C \) is a conflicting clause. This step leads to conflict analysis and backtracking.

A key benefit of two watch literals method is while backtracking there is no need to modify the watched literals as they could be any two literals of the clause. Therefore while backtracking; unassigning a variable can be performed in a constant time. Moreover, watched literals would be watched by fewer clauses than they were right before backtracking so reassigning this variable would tend to be faster. Eventually, this fact reduces the number of memory accesses and cache miss rate as well [5].

5.7.5 setVar

This function accepts the literal \( v \) as a parameter and assigns its truth value. It traverses the list of watching clauses to find the unit or contradicting clauses resulted due to this truth assignment. It keeps adding all the implied literals in **globalUclQueue** until no more implied literals exist or a contradiction is found in which case it sets the **contradictoryUC** and **conflictingClause** variables. In order to identify the conflict as soon as possible, **setVar** checks if the implied variable has already been implied before adding it to the queue. If the current implied variable has already been implied the same value before, it does not need to be added in the queue, however, if it has been implied opposite polarity previously, it indicates the contradiction.

5.7.6 unsetVar

This function accepts the literal \( v \) as the parameter and declares it unassigned. Due to the use of the lazy data structure two-watched literal for BCP operation, this **unsetVar** can be performed in \( O(1) \).

5.7.7 analyzeConflict

This method is invoked every time the conflict is identified. This method performs conflict analysis by invoking mainly three methods: **learnConflictClause**, **calculateBacktrackingLevel**, and **saveLearnedClause**. It returns the backtrack level which is used to update the variable **depth**.

5.7.8 learnConflictClause

This method implements the 1-UIP [12] clause learning scheme to learn conflict clause using Implication graph also known as I-Graph [12] maintained by **Literal** and
Clause data structures. Please refer [12] [20] for the details of I-Graph and to understand the step-by-step procedure of how to create it.

Before discussing 1-UIP scheme, one needs to first understand a few terminologies.

**Unique Implication Point:** A *Unique Implication Point* (UIP) is any node at the current decision level such that any path from the decision variable to the conflict node must pass through it. By definition the decision variable at the current decision level is always an UIP.

**Cut:** When an I-Graph contains a conflict it can be split into two sides using bipartition called a cut. The partition side containing the conflict node is called the conflict side, whereas the other side is called the reason side. The conflict side may contain nodes other than conflict nodes. There are many ways to get different cuts and they are associated with different clause learning schemes.

The first-UIP or 1-UIP scheme corresponds to a 1-UIP cut which partition the I-Graph right before the 1-UIP on the path from the conflict to decision variable as shown in the Figure-19. The rationale behind this is that the learned clause is more relevant to the conflict as the 1-UIP is the closest UIP to the conflict nodes. The conflict clause is created using all of the variables on reason side with edges going to conflict side through 1-UIP cut. In our example these variables are $x4$, $x8$, and $x9$. Conflict clauses are learned to avoid the same conflict in the future, and since the truth assignments of these three variables altogether causes the conflict in the I-Graph, the conflict clause is derived by negating these three variables with respect to the I-Graph. So, in our example, resulted learned conflict clause is $(\neg x4 \lor x8 \lor x9)$. Please refer to the [12] for more detailed description of 1-UIP with an example.
5.7.9 calculateBacktrachingLevel

As explained in section 3.4, this method calculates the backtrack level using the conflict clause returned by the learnConflictClause method. Given a conflict clause, backtrack level is the 2nd highest depth of all the literals. If the size of the conflict clause is one, the back-track level is 0.

5.7.10 saveLearnedClause

This method performs mainly two operations as explained below:

a. The learned clauses with a size less than or equal to cutoff k value are reported to the global clause database where as larger clauses are marked as black clauses. This is called k-ordered learning technique [12].

b. A Clause object is created using the learned clause and inserted into the clsList.

5.7.11 nonChronologicalBacktracking

It backtracks non-chronologically by performing the following three tasks.

a. It removes all the implied literals from the globalUclQueue and marks them as not implied.

b. From the assignedLits stack, it pops out all the literals that were assigned at the level greater than current depth and unset them. Remember that the depth was already updated by the backtrack level returned by the analyzeConflict method.

c. It sets the literal that was implied due to the recently learned clause.

5.7.12 deleteNonAssertingBlackClauses

The references to the black clauses are local to the CDCL solver running in each slave process because they are never reported to the global clause database. Because their sizes are greater than the LearnedClauseCutoff, they need to be removed according to the k-bounded learning scheme [12].

5.7.13 reset

This function reads recently learned clauses from global database and updates the local clause database of CDCL algorithm. It is called when the solver needs to be restarted as well as when slave objects needs to be reset for next chunk processing.

6. Performance evaluation and comparison

This section discusses the results of our experiments and HyParSAT’s performance comparison with few existing SAT solvers. I used champ.cs.rit.edu and nessie.cs.rit.edu [28] which are two parallel computers with identical configuration as
mentioned below. During all the experiments, HyParSAT was executed on *champ* whereas ManSAT and MiniSat were executed on *nessie*.

- One Intel Xeon E5-2690 processor
- Eight dual-hyperthreaded CPU cores
- 16 threads
- 2.9 GHz clock
- 32 GB main memory
- One NVIDIA Tesla C2075 GPU accelerator, 448 cores
- Ubuntu 12.04 LTS 64-bit Linux

### 6.1 Effect of Each Portfolio, Clause Database Size, and Core Count

Table shown in figure-20 demonstrates how the different combinations of portfolios, the number of cores used, restart policies, and the learned clause database size affect the overall performance of HyParSAT. In this table M1, M2, Z1, and Z2 represent *ManySATPortfolio1*, *ManySATPortfolio2*, *zChaffPortfolio1*, and *zChaffPortfolio2*, respectively. The function `currentTimeMillis` available in `System` class of Java is used to retrieve the current system time in milliseconds. The running time of each execution is the absolute difference between the current times measured right before and after calling the `selectPortfolio` method in `Solver` class. This running time includes only the time spent while solving the problem. It does not include the time spent while reading the original CNF from the input file.

Experiment in case-1 is designed to show the effect of the number of portfolios and the size of global clause database on the overall performance while using only 1 core per portfolio. Running all the portfolios with higher global database size for relatively small and easy SAT instance actually adds up the overhead of thread creation, thread synchronization, and processing of more learned clauses. It has been observed that running only Z2 portfolio with database size of 500 takes the minimum CPU time to solve smaller instances. This experiment suggests using small global clause database and appropriate portfolio while trying to solve comparatively small instances.

Experiment in Case-2 demonstrates the side effects of using unnecessary number of cores while solving small and easy instances. It uses only one portfolio Z2 and same database size for all the runs to observe the effect of the number of cores. Running Z2 portfolio with 8 cores for small instance takes more CPU time than it takes with only 4 cores. This is because of the overhead associated with the more numbers of threads is more than actual processing time. It shows that the number of slave processes should be chosen based on the relative size and hardness of the considered instance.

In Case-3 I have chosen relatively large instance while running only Z2 portfolio to show the effect of database size and the number of cores. This experiment shows that the use of more cores and larger clause database significantly lowers the required CPU time. Large instances tend to generate many conflicts so it is recommended to use larger database to save many clauses, and many number of cores to increase parallelism.
Case-4 shows the effect of the fast restart versus slow restart policy. Here I have performed all the experiments using fix number of core per portfolio and the database size. While solving larger and harder instances, M1 and Z1 are portfolios with fast geometric restart policy which significantly slows down the performance; however, use of portfolios M2 and Z2 which are equipped with the slow arithmetic restart policy skyrocket the solver performance.

<table>
<thead>
<tr>
<th>Case No.</th>
<th>Instance</th>
<th>Variables</th>
<th>Clauses</th>
<th>Portfolio</th>
<th>Cores per portfolio</th>
<th>DB size</th>
<th>CPU Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>qg4-08</td>
<td>512</td>
<td>9685</td>
<td>M1M2Z1Z2</td>
<td>1</td>
<td>1000</td>
<td>8.196</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M2Z2</td>
<td>1</td>
<td>1000</td>
<td>5.803</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Z2</td>
<td>1</td>
<td>500</td>
<td>1.027</td>
</tr>
<tr>
<td>2</td>
<td>qg7-11</td>
<td>1331</td>
<td>49534</td>
<td>Z2</td>
<td>8</td>
<td>500</td>
<td>8.065</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Z2</td>
<td>4</td>
<td>500</td>
<td>6.801</td>
</tr>
<tr>
<td>3</td>
<td>qg7-12</td>
<td>1728</td>
<td>70327</td>
<td>Z2</td>
<td>8</td>
<td>500</td>
<td>93.890</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Z2</td>
<td>16</td>
<td>500</td>
<td>43.580</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Z2</td>
<td>16</td>
<td>2000</td>
<td>37.981</td>
</tr>
<tr>
<td>4</td>
<td>Qg7-13</td>
<td>2197</td>
<td>97072</td>
<td>M1M2Z1Z2</td>
<td>4</td>
<td>2000</td>
<td>3.054</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M1Z1</td>
<td>4</td>
<td>2000</td>
<td>80.495</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M2Z2</td>
<td>4</td>
<td>2000</td>
<td>3.03</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Z1Z2</td>
<td>4</td>
<td>2000</td>
<td>3.797</td>
</tr>
</tbody>
</table>

**Figure-20: Effect of Each Portfolio on Overall Performance of HyParSAT**

### 6.2 Performance Comparison with Other Solvers

This section demonstrates the performance of HyParSAT as opposed to MiniSat, and ManySAT. Table in figure-21 shows the CPU time taken by these three solvers while solving various satisfiable and unsatisfiable SAT instances selected from SAT Race 2010 and [31]. It can be observed that HyParSAT took much longer than other solvers while solving 11 satisfiable and 9 unsatisfiable instances. I believe that this was caused because of one or more of the below explained factors:

a. HyParSAT uses branching heuristic RAND in two portfolios whereas VSIDS in other two portfolios. This decision was made hoping that it would be better to provide huge diversity among the branching heuristics of all four portfolios. ManySAT uses VSIDS in its all four portfolios with up to 3% randomness in each. It has already been proven by [5] that VSIDS performs much better than any other existing heuristics. Apparently, RAND branching heuristics did not really boost the performance of HyParSAT.

b. ManySAT implements clause learning scheme 1-UlP in its two portfolios whereas extended clause learning [29] in other two portfolios. According to ManySAT experiments, this new advanced clause learning scheme increases the size of back-jumps, and eventually the overall performance. HyParSAT uses
only 1-UIP scheme in all of its four portfolios which would potentially limit its overall performance.

c. ManySAT implements the polarity saving technique [4] to speed up the BCP operation taking place right after backtracking. HyParSAT does not implement any such technique.

d. ManySAT has implemented Luby [49] and Dynamic restart policies in addition to geometric and arithmetic policies. HyParSAT has implemented only the later two policies. ManySAT experiments indicate that dynamic restart policy works well on industrial SAT instances. This might have slightly boosted of ManySAT’s performance here as I have used industrial instances for my experimentations.

HyParSAT could not solve 5 satisfied and 4 unsatisfied instances whereas the other two solvers could. There are 6 satisfiable instances which could not be solved by any of the three solvers within 2 hours cutoff time. One thing to be noted that MiniSat showed an error while processing dubois100.cnf instance. In case of some SAT instances, MiniSat surprisingly outperforms ManySAT.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Variable</th>
<th>Clauses</th>
<th>HyParSAT (seconds)</th>
<th>MiniSat (seconds)</th>
<th>ManySAT (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hanoi4.cnf</td>
<td>718</td>
<td>4934</td>
<td>1.710 s</td>
<td>0.040</td>
<td>0.056</td>
</tr>
<tr>
<td>aim-200-1_6-yes1-1</td>
<td>200</td>
<td>320</td>
<td>0.056</td>
<td>0.001</td>
<td>0.004</td>
</tr>
<tr>
<td>aim-200-1_6-yes1-2</td>
<td>200</td>
<td>320</td>
<td>0.093</td>
<td>0.001</td>
<td>0.004</td>
</tr>
<tr>
<td>bw_large.a</td>
<td>459</td>
<td>4675</td>
<td>0.112</td>
<td>0.001</td>
<td>0.008</td>
</tr>
<tr>
<td>bw_large.b</td>
<td>1087</td>
<td>13772</td>
<td>0.241</td>
<td>0.008</td>
<td>0.016</td>
</tr>
<tr>
<td>bw_large.c</td>
<td>3016</td>
<td>50457</td>
<td>0.826</td>
<td>0.040</td>
<td>0.060</td>
</tr>
<tr>
<td>bw_large.d</td>
<td>6325</td>
<td>131973</td>
<td>400.019</td>
<td>0.200</td>
<td>0.200</td>
</tr>
<tr>
<td>qg2-08</td>
<td>512</td>
<td>148597</td>
<td>480</td>
<td>1.480</td>
<td>2.408</td>
</tr>
<tr>
<td>qg7-09</td>
<td>729</td>
<td>22060</td>
<td>0.11</td>
<td>0.008</td>
<td>0.012</td>
</tr>
<tr>
<td>qg2-07</td>
<td>343</td>
<td>68083</td>
<td>0.354</td>
<td>0.012</td>
<td>0.020</td>
</tr>
<tr>
<td>qg7-13</td>
<td>2197</td>
<td>97072</td>
<td>3.03</td>
<td>0.228</td>
<td>0.076</td>
</tr>
<tr>
<td>hanoi5</td>
<td>1931</td>
<td>14468</td>
<td>Stopped (2hr)</td>
<td>3.1802</td>
<td>0.836</td>
</tr>
<tr>
<td>vmpec_27</td>
<td>729</td>
<td>96849</td>
<td>Stopped (2hr)</td>
<td>413.93</td>
<td>1033.69</td>
</tr>
<tr>
<td>vmpec_28</td>
<td>784</td>
<td>108080</td>
<td>Stopped (2hr)</td>
<td>141.057</td>
<td>8095.92</td>
</tr>
<tr>
<td>vmpec_31</td>
<td>961</td>
<td>146909</td>
<td>Stopped (2hr)</td>
<td>413.93</td>
<td>1033.69</td>
</tr>
<tr>
<td>f600</td>
<td>600</td>
<td>2550</td>
<td>Stopped (12hr)</td>
<td>Stopped (2hr)</td>
<td>Stopped (2hr)</td>
</tr>
<tr>
<td>g125.17</td>
<td>2125</td>
<td>66272</td>
<td>Stopped (2hr)</td>
<td>Stopped (2hr)</td>
<td>Stopped (2hr)</td>
</tr>
<tr>
<td>g125.18</td>
<td>2250</td>
<td>70163</td>
<td>Stopped (2hr)</td>
<td>Stopped (2hr)</td>
<td>Stopped (2hr)</td>
</tr>
<tr>
<td>g250.15</td>
<td>3750</td>
<td>233965</td>
<td>Stopped (2hr)</td>
<td>Stopped (2hr)</td>
<td>Stopped (2hr)</td>
</tr>
<tr>
<td>g250.29</td>
<td>7250</td>
<td>454622</td>
<td>Stopped (2hr)</td>
<td>Stopped (2hr)</td>
<td>Stopped (2hr)</td>
</tr>
<tr>
<td>md5_47_4</td>
<td>65604</td>
<td>273506</td>
<td>Stopped (3hr)</td>
<td>418.218</td>
<td>1118.48</td>
</tr>
<tr>
<td>md5_48_1</td>
<td>66892</td>
<td>279248</td>
<td>Stopped (3hr)</td>
<td>556.619</td>
<td>2685.73</td>
</tr>
<tr>
<td>Instance</td>
<td>Variable</td>
<td>Clauses</td>
<td>HyParSAT (seconds)</td>
<td>MiniSat (seconds)</td>
<td>ManySAT (seconds)</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
<td>---------</td>
<td>--------------------</td>
<td>-------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>aim-200-1_6-no-3</td>
<td>200</td>
<td>320</td>
<td>0.064</td>
<td>0.001</td>
<td>0.004</td>
</tr>
<tr>
<td>aim-200-1_6-no-4</td>
<td>200</td>
<td>320</td>
<td>0.09</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>dubois100</td>
<td>300</td>
<td>800</td>
<td>0.11</td>
<td>Error</td>
<td>0.004</td>
</tr>
<tr>
<td>qg4-08</td>
<td>512</td>
<td>9685</td>
<td>1.027</td>
<td>0.016</td>
<td>0.063</td>
</tr>
<tr>
<td>qg3-09</td>
<td>729</td>
<td>16732</td>
<td>85.4713</td>
<td>1.404</td>
<td>4.668</td>
</tr>
<tr>
<td>qg7-10</td>
<td>1000</td>
<td>33736</td>
<td>0.217</td>
<td>0.004</td>
<td>0.16</td>
</tr>
<tr>
<td>qg7-11</td>
<td>1331</td>
<td>49534</td>
<td>7.629</td>
<td>0.016</td>
<td>0.056</td>
</tr>
<tr>
<td>qg7-12</td>
<td>1728</td>
<td>70327</td>
<td>37.981</td>
<td>0.120</td>
<td>0.332</td>
</tr>
<tr>
<td>ssa6288-047</td>
<td>10410</td>
<td>34238</td>
<td>0.096</td>
<td>0.004</td>
<td>0.016</td>
</tr>
<tr>
<td>ndhf_xits_13_UNSAT</td>
<td>2730</td>
<td>271285</td>
<td>Stopped (3hr)</td>
<td>Stopped (3hr)</td>
<td>390.944</td>
</tr>
<tr>
<td>q_query_3_l48_lambda</td>
<td>34656</td>
<td>174528</td>
<td>Stopped (3hr)</td>
<td>76.112</td>
<td>393.573</td>
</tr>
</tbody>
</table>

Figure-21: Performance Comparison of HyParSAT with Other Solvers

7. Future work

In HyParSAT I have incorporated most of the latest proven features of modern CDCL SAT solvers; however, I could not implement certain features due to the limited time availability. Moreover, there are few more recently suggested but unproven features which we have not implemented in HyParSAT. Below is the overview of features that I have planned to incorporate in HyParSAT in the future.

a. HyParSAT implements the restricted clause learning using the combination of \( n \)-order learning and \( k \)-bounded learning schemes but it never deletes any clause from the global database. For better performance, I plan to introduce heuristic based clause deletion policy in HyParSAT.

b. It has been proven that preprocessing of the original CNF helps to reduce the number of learned clauses and eventually improves the overall performance. Currently, HyParSAT performs very basic preprocessing of the given CNF; however, this may be significantly improved.

c. Currently, the LearnedClauseSizeCutoff is static and does not change over the time during solver execution; however, it can be configured as dynamic since the average size of learned clauses tends to increase over the time.

d. HyParSAT has been designed by adhering object oriented design aspects, but during the final stages of its implementation I realized that its design could have been even more standardized in such a way that the new portfolios can be easily introduced in HyParSAT.

e. Currently HyParSAT is not designed to deal with memory overflow errors. This is one of the most important but difficult to implement feature. Memory usage awareness can be introduced in HyParSAT such that the solver can keep track of its memory usage, and react before hitting the memory overflow errors.

f. ManySAT implemented the extended clause learning scheme in two of the four solvers in its portfolio. This technique often increases the size of back-jumps. I plan to implement this new technique in two of the portfolios of HyParSAT.
HyParSAT has been designed very thoughtfully using efficient data structures and algorithms; however, I believe that the performance of HyParSAT can be tuned up by using cache aware data structures and algorithms.

8. Related work

I came up with the design of HyParSAT after exhaustive study of many complete sequential and parallel solvers. Here, I present some state of the art modern sequential and parallel complete SAT solvers.

zChaff [5] is a sequential SAT solver which implemented new approach of BCP based on two-watched literal data structure, highly optimized branching heuristic VSIDS focusing on recently added learned clauses, and the sophisticated 1-UIP clause learning scheme. These key features of zChaff’s efficient engineering make it much faster than most of the public domain sequential SAT solvers on hard instances. That’s why we have incorporated these three key features of it in my solver HyParSAT.

ManySAT [4] is a portfolio-based parallel solver that runs a portfolio of four complementary sequential solvers with dynamic clause sharing policy. This new portfolio-based approach with four diversified solvers along with a new efficient dynamic restart policy, and a new polarity strategy for literal assignment help ManySAT to perform well on industrial SAT instances. This is why it could secure the first rank in the parallel track of 2008 SAT-Race.

SArTagnan [24] is another promising portfolio-based parallel solver which introduced highly efficient clause sharing and inter-thread communication scheme that benefits every thread from the application of simplification and clause minimization techniques of other threads. Its combination of logical and physical clause sharing technique compensates the ill effects of slow BCP operation. Moreover, they suggested an XOR value based approach, an alternative of two-watched literals to improve their BCP operation.

GrADSAT [25] is a zChaff based parallel solver aimed for hard SAT instances and designed to run on a grid of distributed commodity computation resources. It implements the master-slave model based on the classical guiding path divide-and-conquer strategy used to dynamically balance the workload between clients. Their clause sharing scheme is very similar to the one implemented in HyParSAT. All clients exchange the learned clauses if their size is smaller than cutoff value.

PMSat [26] is a MiniSat based parallel solver designed to run on a grid of computing resources. It uses master slave approach and implements guiding path divide and conquer approach that is very similar to GrADSAT, however, it provides the user several partitioning heuristics options to select from. The slave process reports 50 most active clauses to the master if it finds the sub-task unsatisfiable. The master process shares these clauses with running processes dynamically. It uses Message Passing Interface (MPI) to implement clause sharing between slave processes.
PaMira [27] is the master-slave architecture based parallel solver for multi-core computers. Its design is based on Mira, a highly optimized sequential SAT solver engine. Its clause sharing scheme is similar to HyParSAT, however, PaMira implemented an efficient work stealing method for load balancing. For this, it uses dynamic search partitioning scheme which divides the whole binary tree search space into disjoint parts which could be processed in parallel.

The grid-based solver proposed in [45] uses portfolio approach and runs multiple asynchronous solvers with different search strategies that do not communicate with each other because they work on their own task independently; however, the grid-based solver proposed in [46] the currently running tasks report their learned clauses to the master to share them with the future tasks. Such kind of clause sharing improves overall performance in most of the cases, however, the heuristic to guide clause sharing that guarantees the performance improvement has not been found yet.

9. Conclusion

In this project I have demonstrated and implemented HyParSAT, a new hybrid approach of complete parallel SAT solver which is developed in 100% java and designed to run on shared memory multiprocessor computers. It combines a portfolio-based approach with the classical divide and conquer strategy which makes HyParSAT one its kind. It is highly configurable and adaptable to the available CPU cores. Its three level master-slave architecture with four complimentary portfolios provides diversity and parallelism together.

The inherent complexity and NP-Completeness of SAT problem makes it extremely difficult to achieve true parallelism in SAT solvers. Even though guiding path divide and conquer strategy divides the binary tree search space into theoretically non-overlapping chunks, all the CDCL solvers running in slave processes tend to eventually end up searching in overlapped search spaces due to the nature of the CDCL algorithm. During my experimentation, I noticed that sequential solver MiniSat sometimes performs much better than parallel solvers HyParSAT and ManySAT on certain instances.

Even though HyParSAT implements most of the features of modern CDCL solvers, the performance of HyParSAT is still not up to the mark as opposed to sequential solver MiniSat and parallel solver ManySAT which are two of the best available solvers in the market. I believe this is because of MiniSat and ManySAT solvers have implemented some techniques that I could not incorporate in HyParSAT due to the limited available time. I believe that HyParSAT can perform much better if it could be provided with all the features discussed in section-6.
10. References


