Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 100
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 100
Espoo 2006 HUT-TCS-A100

SATU: A SYSTEM FOR DISTRIBUTED PROPOSITIONAL SATISFIABILITY CHECKING IN COMPUTATIONAL GRIDS

Antti E. J. Hyvärinen

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 100
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 100
Espoo 2006 HUT-TCS-A100

SATU: A SYSTEM FOR DISTRIBUTED PROPOSITIONAL SATISFIABILITY CHECKING IN COMPUTATIONAL GRIDS

Antti E. J. Hyvärinen

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FI-02015 TKK
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

© Antti E. J. Hyvärinen

ISBN 951-22-2068-X
ISSN 1457-7615

Multiprint Oy
Helsinki 2006

Abstract: In addition to data storage and indexing systems, computational grids are used for solving computationally demanding tasks. Because of the inherent communication delays and high failure probabilities of a loosely coupled and large computational grid, such an environment poses a great challenge for highly data-dependent algorithms. In this work we present a distribution scheme for solving propositional satisfiability problem (SAT) instances, called scattering. The key advantages of scattering are that it can be used in conjunction with any sequential or parallel satisfiability checker, including industrial black box checkers, and that the distribution heuristic is strictly separated from the heuristic used in sequential solving. We also give an implementation of the scheme and benchmark it using a production-level grid. The benchmarks range from factorization to cryptanalysis and random 3SAT. Some benchmarks not known to have been solved previously are solved with the distribution scheme.

Scattering is analysed with respect to the challenges posed by the grid environment: the long communication delays and the high failure rates of individual jobs. We study different approaches to coping with the communication delays and thus maximizing the effective parallelism in the grid. We give a criterion for the completeness of the scattering in a pure grid environment with failures.

Keywords: Parallel Search Algorithms, Parallelization of DPLL procedure, Propositional Satisfiability, Computational Grid

CONTENTS

1 Introduction **1**
1.1 This Work 3

2 Preliminaries **5**
2.1 Propositional Satisfiability 5
2.2 The Davis-Putnam-Logemann-Loveland -algorithm 6
2.3 DPLL with Clause Learning 7

3 Parallelization **12**
3.1 The Scattering Rule 14
3.2 Implementing the Scattering Rule 15
3.3 The Search 19
3.4 Heuristic for the Scattering 20
3.5 Learning in Scattering 23

4 Architecture **25**
4.1 Scattering Functionality 26
4.1.1 Scatter 26
4.1.2 Search 27
4.2 Interface to the computing environment 28
4.2.1 GRIDJM 28
4.2.2 Filter 31
4.2.3 SATqueue 31
4.3 The Parameters of SATU 32

5 Experimental Results **33**
5.1 The Effect of the GRID Errors 34
5.2 Benchmarking GRIDJM 35
5.3 Benchmarking SATU 38
5.3.1 Unsatisfiable and Satisfiable Random 3SAT 39
5.3.2 Even Bit Composite Number Factorization 49
5.3.3 Prime Number Factorization 49
5.3.4 Four-round One-block DES 54
5.3.5 The Unsolved problems from SAT2005 63

6 Related Work **64**

7 Conclusions **67**
7.1 Further Work 67

Bibliography **73**

List of Figures

2.1 A conflict graph 10
3.1 The interface between the client and the parallel executing environment 13
3.2 A part of a scattering tree for formula (2.4) 14
4.1 The Program Architecture 25
5.1 Parallelism and time 37
5.2 Median and average speedups, and speedups of median and average for all formulas 39
5.3 Median and average run times for all formulas 39
5.4 Unsatisfiable random 3SAT, random and minmax heuristic, duration in seconds 41
5.5 Scalability for some unsatisfiable random 3SAT formula sizes, duration in seconds 42
5.6 Speedup for some unsatisfiable random 3SAT formula sizes, speedup compared to smallest maximum grid jobs 43
5.7 Speedup of minimum, average, median and maximum for some unsatisfiable random 3SAT formula sizes. Speedup compared to minimum, average, median or maximum at smallest maximum simultaneous grid jobs 44
5.8 Satisfiable random 3SAT, random and minmax heuristic, duration in seconds 45
5.9 Scalability for some satisfiable random 3SAT formula sizes, duration in seconds 46
5.10 Speedup for some satisfiable random 3SAT formula sizes, speedup compared to smallest maximum grid jobs 47
5.11 Speedup of minimum, average, median and maximum for some satisfiable random 3SAT formula sizes. Speedup compared to minimum, average, median or maximum at smallest maximum simultaneous grid jobs 48
5.12 Even bit composite number factorization, duration in seconds 50
5.13 Scalability for some even bit composite factorization formula sizes, duration in seconds 51
5.14 Speedup for some even bit composite factorization formula sizes, speedup compared to smallest maximum grid jobs . . . 52
5.15 Speedup of minimum, average, median and maximum for some even bit composite factorization formula sizes. Speedup compared to minimum, average, median or maximum at smallest maximum simultaneous grid jobs 53
5.16 Prime number factorization, duration in seconds 55
5.17 Scalability for some prime factorization formula sizes, duration in seconds 56
5.18 Speedup for some prime factorization formula sizes, speedup compared to smallest maximum grid jobs 57

5.19 Speedup of minimum, average, median and maximum for some prime factorization formula sizes. Speedup compared to minimum, average, median or maximum at smallest maximum simultaneous grid jobs 58
5.20 Scattering tree for random (top) and MINMAX (bottom) heuristics for a four-round one-block DES formula. Darker shades indicate longer run times 59
5.21 Scalability for four-round one-block DES, duration in seconds 60
5.22 Speedup for four-round one-block DES, speedup compared to smallest maximum grid jobs 61
5.23 Speedup of minimum, average, median and maximum for four-round one-block DES. Speedup compared to minimum, average, median or maximum at smallest maximum simultaneous grid jobs 62

List of Tables

3.1 Scattering using over- and under-constraining approximations 23
5.1 Grid nodes used in benchmarks 34
5.2 Heuristic comparison for Four round one block DES 54
5.3 Selected unsolved formulas from SAT2005 solver competition 63

1 INTRODUCTION

After the rise of the Internet and with its ever increasing amount of connected computing and storage resources, new parallel computing models utilizing the unstable but powerful new environment are being widely studied. This new, loosely coupled and heterogeneous computing environment is named after the electric grid, to which it is somewhat analogous, simply as the grid [29].

Many success stories already exist where grid computing has played a major part, SETI@home [53] and Google™ [13] search engine to name a few. Still, there is more to be achieved by distributing computing power in the Internet in the same way as electricity is distributed in a power grid. The construction of the electric grid has created a myriad of applications. The main reason for the success of the electric grid is that it is easy to use, reliable, cost effective and available virtually everywhere. If the computational grid meets these criteria, there is every reason to believe that it will achieve a similar role amidst the general public. An example of an ambitious work-in-progress towards this goal is NorduGrid [25], a network of computer clusters available for general purpose computing to the members of the organization.

The computational grid has also political, economical and ecological aspects. As a large scale project with huge implications, it might be the most interdisciplinary subject ever to have emerged from the field of computer science. As the location of the nodes in the grid is insignificant, the international community is free to place the nodes anywhere in the geo-political map. This includes the developing countries and provinces, wherever the local climate is not prohibitive for computers. The cost of even a large high-end cluster is dramatically smaller than the cost of a traditional mainframe computer and the computational power of the grid is not so much dependent on the power of its discrete components. This enables small companies and universities with limited economical capacity to build computing nodes and join a computational grid. The environmental aspects of the computational grids are analogous to the electric grid: it is clear that producing electricity centrally for a large number of clients is more environmentally friendly than producing it locally assuming that the production is done ecologically in both cases. Same arguments hold for the computational grid. Most computing cycles in the world are wasted on idle processes. By sharing the computing power, higher usage rate can be achieved and thus less computers will be required to satisfy the same need for computing time.

Since computational grids are becoming an inexpensive way of distributing computational power over large user base, the work of designing algorithms suitable for the requirements of grids has recently received interest. A classical computationally demanding problem is the propositional satisfiability problem [42]. The propositional satisfiability problem (SAT), i.e., the problem of determining whether there is a satisfying assignment to a Boolean formula belongs to the set of NP-complete problems, as shown by Cook [19]. No deterministic algorithm yielding a guaranteed polynomial-time solution to a generic NP-complete problem is known; the best current algorithms suffer from worst case exponential run times with respect to the problem size.

As problems emerging from different areas of research turn out to be computationally difficult, some being NP-complete, it is tempting to try to develop reductions from the original problem to a general form and examine the problem with an optimized algorithm. These reductions are typically much simpler to create than writing problem specific optimized algorithms and thus the time and effort spent in creating an efficient generic algorithm is definitely worthwhile. Any NP-complete problem solving algorithm would be fit for the purpose. However, SAT has some advantages, being much studied, simple to represent and for many problems, a natural target language.

This reasoning leads to an idea of a programming language which is suitable for high-level problem descriptions from different fields, such as hardware verification, artificial intelligence and bioinformatics, and which can be converted to SAT. Then an optimized and efficient SAT checker is used as a back-end to solve the actual problem and another filter is used to transform the results to the domain-specific expressions.

A turn-side of the generality of SAT is the unpredictability in computing time. A typical set of SAT instances from an industrial problem might have solving times ranging from seconds to days, even though the instance descriptions might be equally large when measured, say, by the number of variables in the problem. In fact, the mean of the run time for a certain problem class and size does not seem to be well-defined in the algorithms typically used [31]. This empirical observation has to be taken into account when solving any NP-complete problems, such as SAT.

Many interesting practical and theoretical problems have been formulated as propositional formulas and solved successfully. To name a few, probably the most famous application domains of SAT solving are circuit verification (see, for example [9, 8, 54]) and VLSI-routing problems [1]. The transformation of a practical problem to a propositional formula has also been successfully applied to verifying consistency and completeness of an on-line help system [50]. An extension of a SAT solver where it is possible to add constraints concerning the amount of true literals in a clause has been successful in the sports tournament scheduling [57]. An early work shows that SAT solvers are suitable for solving scheduling problems [21], and another one shows how SAT solving can be used in generating test patterns for single stuck-at faults in combinational circuits [36]. An experiment in the AI planning field shows that SAT formulation gives more flexibility compared to traditional deduction method [35]. Transformations from answer set programming (ASP) [40] to SAT are increasingly common mainly because of the increased efficiency of SAT checkers [30]. This opens new application fields, including historical analysis of natural languages and parasite-host systems [14, 26] and many others. Due to the advances on the SAT solving technology, SAT solving has recently been applied to model checking [2], a field previously dominated by Binary Decision Diagram (BDD) [15] based solvers.

SAT solvers are being actively developed both in the industry and in academia. The SAT solver competition [18, 48, 6, 7], organized since 2002, illustrates the progress of the solvers. The winning solver from previous years usually perform only moderately when compared to new solvers. The parallel SAT solving has recently received interest with the increasing importance of SAT-based systems in the industry. The continuous and rapid evolution

of solvers together with the grid, a loosely coupled parallel executing environment, compose an interesting challenge for parallel solvers. The conventional method for parallelization is a solver running in a tightly coupled multiprocessor architecture with shared memory working on a single problem instance. While this path leads to efficient programs and delicate program structures, our fear is that the inherent complexity of parallel computing makes it expensive to modify the solvers as the solving techniques evolve.

1.1 THIS WORK

In this work we study how computational grids can be used in solving of challenging propositional satisfiability problems. For this purpose we develop a distributed search technique for SAT. The technique is based on dividing the SAT problem instance in question into more and more constrained subproblems which are then submitted to grid nodes to be solved there.

We use the Davis-Putnam-Logemann-Loveland (DPLL) SAT solving algorithm in an environment where parallel tasks are only able to communicate through a central authority. We want to use existing and unmodified SAT solvers to be able to benefit from the continuous evolution of sequential solvers without the need to make expensive modifications to their source code. This type of solving fits well computational grids, where direct communication between jobs is costly and sometimes impossible due to network security considerations. Our base model is a master-slave architecture, in which the master acts as the central authority and the slaves run in the grid. We propose an approach called *scattering*, an algorithm for dividing the SAT problem to subproblems forming a *scattering tree* and solving the subproblems in the scattering tree until the solution for the original problem can be deduced from the results of the subproblems by the master. We also give an implementation of the approach, the SATU (SAT Ubiquitous) distributed propositional satisfiability solver. The implementation supports clause learning and implements a limited amount of learning in the master process. Any SAT solver can be used for solving subproblems in the grid. The implementation is benchmarked on a production-level computational grid with benchmarks ranging from random 3SAT to circuit verification and cryptanalysis. The benchmarks are run on NorduGrid, a computational grid consisting of mainly PC clusters running the Linux operating system.

Unlike the SAT solver presented in this work, most existing parallel SAT solvers are either built using shared memory, such as [12, 27], or make heavy modifications to the solving algorithms, such as the solvers presented in [56, 16, 34, 49]. Many of the solvers also expect arbitrary jobs to be able to communicate directly with each other. The existing parallel solvers are discussed in detail in Chapter 6.

The rest of this work is organized as follows: Chapter 2 introduces the SAT problem formally, describes the DPLL algorithm, learning in SAT and non-chronological backtracking. In Chapter 3, we describe the scattering algorithm with pseudo code, give the completeness and soundness proofs for the algorithm, cover the decision heuristic used by the scattering and describe the learning approach. In Chapter 4, we describe our implementation

of scattering with an overview of the architecture and details to some degree. Chapter 5 describes the benchmarking results for random 3SAT, some factorization problems and DES, as well as results for some previously unsolved problems from SAT2006 solver competition. Chapter 6 gives an overview of previous work in distribution of SAT solvers and Chapter 7 concludes the work.

2 PRELIMINARIES

We describe the propositional satisfiability problem and the terminology required for our presentation in Section 2.1. The Davis-Putnam-Logemann-Loveland algorithm, a widely used complete algorithm for solving the propositional satisfiability problem is described in its basic form in Section 2.2. Finally, the concepts and overview of the learning version of the algorithm are given in Section 2.3.

2.1 PROPOSITIONAL SATISFIABILITY

Let V be a finite set of primitive propositions, or variables. The set L of *literals* consists of the variables v and the *negated variables* $\bar{v}$, where $v \in V$. A *propositional formula in conjunctive normal form*, or a *CNF formula*, is a finite set of *clauses* of the form $\{v_1, \ldots, v_n, \bar{v}_{n+1}, \ldots, \bar{v}_m\}$, where $v_1, \ldots, v_m$ are members of V and $v_i \neq v_j$, when $i \neq j$.

A *truth assignment* $\nu : V \to \{true, false\}$ is a mapping from the set of variables V to the set of truth values $\{true, false\}$. It is presented as a set P_L consisting of the literals $\{x \mid \nu(x) = true\} \cup \{\bar{x} \mid \nu(x) = false\}$. Every variable of V *appears* exactly once in the set P_L, either as v or as the negated variable $\bar{v}$. The symbol $\bar{\bar{l}}$ is the same as the literal l. A literal $\bar{l}$ is called the *negation* of the literal l. If a literal l (resp. $\bar{l}$) is a member of the truth assignment P_L, the literal $\bar{l}$ (resp. l) is said to be *false* (resp. *true*) under P_L.

A clause is *satisfied* by the truth assignment P_L, iff (if and only if) it contains a literal l such that $l \in P_L$. A CNF formula F is satisfied by the truth assignment P_L iff all its clauses are satisfied by the truth assignment P_L. In this case we write $P_L \models F$ and call P_L a *satisfying truth assignment*. Otherwise we write $P_L \not\models F$.

A *partial truth assignment* P is a subset of the literals L. The set P is called a *conflicting partial truth assignment* if for some literal l, both $l, \bar{l} \in P$. If a partial truth assignment is not conflicting, it represents a partial function $\nu : V' \to \{true, false\}$, which is a mapping from the subset $V' \subseteq V$ to the values $\{true, false\}$ as described above for the total function ν. A literal is *free* under P if the corresponding variable does not appear in P.

Definition 1 *The* Propositional Satisfiability Problem (SAT) *is: given a CNF formula F over the variables V, determine whether there is a truth assignment P_L such that F is satisfied by P_L. If such P_L exists, then the formula F is* satisfiable. *Otherwise F is* unsatisfiable.

Example 1 *Let $V = \{x_1, x_2, x_3, x_4, x_5\}$ and a CNF formula*

$$F = \{\{x_2, \bar{x}_5\}, \{x_1, \bar{x}_5\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{\bar{x}_2, \bar{x}_3, x_4\}, \{x_3\}, \{x_5\}\}. \quad (2.1)$$

There is no satisfying truth assignment for the formula F, whereas for the formula $F' = F \setminus \{\{x_3\}\}$ there are two satisfying truth assignments, one of which is $P_L = \{x_1, x_2, \bar{x}_3, \bar{x}_4, x_5\}$, and the other is $P_L = \{x_1, x_2, \bar{x}_3, x_4, x_5\}$.

If F is a CNF formula and P is a partial truth assignment, we use the notation $F(P)$ to denote the *simplified CNF formula with respect to* P,

$$F(P) = \{C \in F \mid (\forall l \in P: \{l, \bar{l}\} \cap C = \emptyset)\} \cup \{C \mid (\exists C' \in F) \text{ and } (C = C' \setminus \bigcup_{l \in P}\{\bar{l}\}) \text{ and } (C \cap P = \emptyset)\}. \quad (2.2)$$

Example 2 *For the CNF formula F defined in Equation (2.1), the simplified CNF formula with respect to the partial truth assignment $\{x_5\}$ is*

$$F(\{x_5\}) = \{\{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{\bar{x}_2, \bar{x}_3, x_4\}, \{x_3\}, \{x_2\}, \{x_1\}\} \quad (2.3)$$

A literal l is *implied* under P if $\{l\} \in F(P)$. The *implying clauses* of the implied literal l are the members of the set $\{C \in F \mid C \setminus \bigcup_{q \in P}\{\bar{q}\} = \{l\}\}$. The *unit clauses* under P are the clauses of length 1 in the simplified formula $F(P)$.

Most of the current satisfiability solvers such as those based on the DPLL algorithm described in the next section assume that their input is in the CNF form. However, when discussing the correctness of our distributed algorithm for SAT we use a slightly more general type of a formula, i.e., a disjunction of CNF formulas $M = F_1 \vee \cdots \vee F_n$. The above definitions of satisfiability extend naturally to the disjunctions; given a disjunction $M = F_1 \vee \cdots \vee F_n$ of CNF formulas $F_i, 1 \leq i \leq n$, M is satisfied by the truth assignment P_L iff at least one of the formulas F_i is satisfied by the truth assignment P_L. In this case we write $P_L \models M$. Two disjunctions of CNF formulas are *logically equivalent*, $M_1 \equiv M_2$ if for every truth assignment P_L, $P_L \models M_1$ iff $P_L \models M_2$. If the disjunction is satisfied by all truth assignments P_L, we write $\models M$ and say that M is a *tautology*. The concepts of logical equivalence and tautology can also be extended to other types of propositional formulas, see, for example [42].

2.2 THE DAVIS-PUTNAM-LOGEMANN-LOVELAND -ALGORITHM

The *Davis-Putnam-Logemann-Loveland* -algorithm [23, 22] (DPLL) takes as input a CNF formula F and outputs *satisfiable* if the formula is satisfiable and *unsatisfiable* if the formula is unsatisfiable. The DPLL algorithm is simple to describe recursively, but is usually implemented as the iterative process we here describe. The algorithm is essentially a depth-first search of a satisfying truth assignment to the CNF formula F and is given in Algorithms 1–4. Initially the partial truth assignment P is set to the empty set. The algorithm works in two phases; in a call to *Propagate*(), all unit clauses under P are identified and the corresponding implied literals are added to the partial truth assignment iteratively until no unit clauses under P exist in the formula F. When a literal is added to the partial truth assignment, the *decision level* of that literal is recorded. The initial decision level is 0. When no unit clauses exist, the algorithm calls *ChooseLit*() and starts the second phase. In this *decision point*, given in Algorithm 3, a new literal l is added to P and is marked as a *decision literal*. The implying clause of a decision literal is the unit clause consisting of the decision literal itself. The decision level is

incremented by one and the decision literal will get the new decision level. If the *unit propagation* in *Propagate*() of Algorithm 1 ends in a *conflict*, that is, the partial truth assignment is conflicting, the algorithm backtracks to the previous decision point in which the literal l was inserted to P and replaces it with the literal $\bar{l}$ instead. This is done in a call to *Analyse-Conflict*, given in Algorithm 2. If there is no previous decision point, the algorithm returns *unsatisfiable*. If all the variables appear in P exactly once, the algorithm has found a satisfying truth assignment and returns *satisfiable*.

2.3 DPLL WITH CLAUSE LEARNING

Conflict-driven clause learning and non-chronological backtracking [38] is based on the idea that by analysing the construction of unit clauses leading to a conflict, one can obtain additional information about the structure of the problem. This information, presented as *learned clauses*, can help to prune the search space further. The clauses learned from a CNF formula are *logical consequences* of the formula, i.e., including the clauses to the formula does not affect the set of satisfying truth assignments.

Some theoretical results regarding the effect of clause learning in DPLL are presented in [5]. The speedup provided by clause learning has been investigated, for example, in [3] and [38]. In addition to shrinking the search space by adding new clauses, clause learning is a natural way of guiding the heuristic search [39].

If two literals l and $\bar{l}$ are implied under a partial truth assignment, the situation is called a *conflict*. Assume that the literal l is implied in the current decision level. One of the clauses implying the literal l is selected as a *conflicting clause*. If the literal $\bar{l}$ is not a decision literal, one of the clauses implying the literal $\bar{l}$ is selected as a *reason clause*. The learned clause is constructed from the conflicting clause by replacing the literal l from the conflicting clause by the literals in the reason clause, excluding the literal $\bar{l}$. If $\bar{l}$ is a decision literal, no changes have been made to the conflicting clause to this point, and the current learned clause is the conflict clause. A learned clause can be used to guide the backtracking of a DPLL algorithm if the clause contains a single literal from the current decision level. If the learned clause still contains more than one literal from the current decision level after replacing l, or if the literal $\bar{l}$ was a decision literal, the negations of some of the literals in the learned clause must be implied in the current decision level. Let us assume that one of the implied literals in the current decision level is q. The literal $\bar{q}$ from the learned clause can be replaced by the implying clause of q excluding the literal q. Replacing is applied until the learned clause contains a single literal from the current decision level. All literals in the learned clause are false in the current decision level, and to resolve the conflict, the DPLL algorithm must backtrack to the second highest decision level of the literals in the learned clause. Unit propagation will result in a new assignment for the single literal p previously in the highest decision level in the learned clause as well as the decrease of the decision level of p. A more thorough presentation of learning is given, for example, in [38].

Global vars: P, the partial truth assignment; dl, decision levels of variables; C, variables tried both ways; D, decision variables; d, decision level

```
function IterativeDPLL(F)
  P := {}                /*  Partial truth assignment  */
  dl := {}               /*  Decision levels of variables  */
  C := {}                /*  Variables tried both ways  */
  D := {}                /*  Decision variables  */
  d := 0                 /*  Decision level  */
  while true
    while not Propagate()
      if not ResolveConflict()
        return unsatisfiable
      end if
    end while
    if not ChooseLit()
      return satisfiable
    end if
  end while.
```

```
function Propagate()
  UC := unit clauses in F(P)
  while UC ≠ {}
    while UC ≠ {}
      Select and remove a unit clause {l} from UC
      P := P ∪ {l}
      x := the variable appearing in {l}
      dl := dl ∪ {⟨x, d⟩}
      if P contains both l and l̄
        return false     /*  found conflict  */
      end if
      add the unit clauses in F(P) to UC
    end while
    UC := unit clauses in F(P)
  end while
  return true.
```

Algorithm 1: Iterative DPLL algorithm; main function and propagation

Global vars: dl, decision levels of variables; d, current decision level; D, decision variables; C, variables tried both ways; P, partial truth assignment

function *ResolveConflict*()
  for $d' := d$ **downto** 1
    $x :=$ the variable in D having pair $\langle x, d' \rangle \in dl$
    if not $x \in C$
      $C := C \cup \{x\}$
      $l :=$ the literal corresponding to x in P
      $P := P \setminus \{l\}$
      $P := P \cup \{\bar{l}\}$
      Backtrack(d')
      return true
    end if
  end for
  Backtrack(0)
  return false.

Algorithm 2: Naïve conflict analysis

Global vars: d, current decision level; C, literals tried both ways; dl, decision levels of variables; D, decision variables; P, partial truth assignment

function *ChooseLit*()
  if there are *free* literals under P
    $d := d + 1$
    $l :=$ the best *free* literal according to a heuristic
    $x :=$ the variable corresponding to l
    $dl := dl \cup \langle x, d \rangle$
    $D := D \cup \{x\}$
    $P := P \cup \{l\}$
    return true /* A *free* literal was found */
  end if
  return false. /* All literals have a value */

Algorithm 3: Heuristic for choosing literals

Global vars: P, the partial truth assignment; dl, decision levels of variables; C, variables tried both ways; D, decision variables; d, decision level

function *Backtrack*(d_{new})
  foreach x s.t. $\langle x, d' \rangle \in dl$ **and** $d' \geq d_{\text{new}}$ **and** $(x \notin D$ **or** $d' \neq d_{\text{new}})$
    /* x is set on a higher decision level but is not the decision variable of level d_{new} */
    $dl := dl \setminus \{\langle x, d' \rangle\}$
    $D := D \setminus \{x\}$
    $P := P \setminus \{x, \bar{x}\}$
    $C := C \setminus \{x\}$
  end foreach
  $d := d_{\text{new}}$
  return.

Algorithm 4: Simple backtracking for DPLL

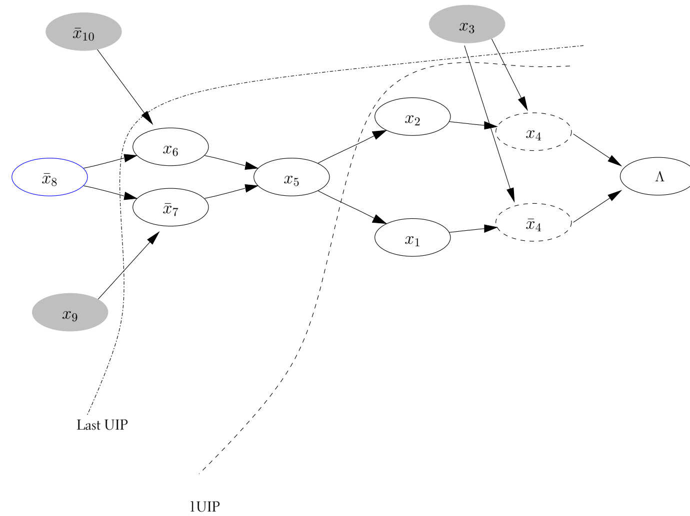

Figure 2.1: A conflict graph

The construction of the clause in *ResolveConflict*() can be alternatively explained with an *implication graph*. Our presentation of the implication graph follows closely that of [5]. The implication graph is a directed acyclic graph in which the nodes are labelled with literals from the partial truth assignment and the edges are directed from the literals of one implying clause towards the implied literal. The choice of the implying clause is part of the heuristic used for constructing the implication graph. Decision literals have in-degree zero. When a partial truth assignment causes a conflict after propagation, the graph will contain some variable twice, negated ($\bar{l}$), and as such (l). In this case, the nodes having a path to either of the variables are of interest to the conflict analysis in constructing the learned clause. This relevant subgraph is the *conflict graph* of the implication graph. For the literals l and $\bar{l}$, a special node Λ is inserted to the graph and directed edges from literals l and $\bar{l}$ to Λ are added to the conflict graph.

The conflict graph is cut into two parts. The other part, called the *conflict side*, contains at least the implied literal l and the node Λ, and the rest of the nodes in the graph having at least one edge to the conflict graph is called the *reason side*. The learned clause is the clause formed by taking the negation of the literals having an edge from the reason side to the conflict side. This cut, or the corresponding clause, is not uniquely defined. The results in [59] suggest that by choosing the cut so that the only literal of the learned clause in the current decision level is nearest to the conflicting literals, yields on average the best speedup. This cut is called the *1UIP-cut*. The 1UIP-cut and another cut used by the *rel_sat* satisfiability solver [4], named Last UIP, are demonstrated in Figure 2.1. The corresponding formula is

$$
\begin{aligned}
F = \{ \quad & \{\bar{x}_2, \bar{x}_3, x_4\}, \\
& \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \\
& \{x_2, \bar{x}_5\}, \\
& \{x_1, \bar{x}_5\}, \\
& \{x_5, \bar{x}_6, x_7\}, \\
& \{x_6, x_8, x_{10}\}, \\
& \{\bar{x}_7, x_8, \bar{x}_9\} \quad \}.
\end{aligned}
\tag{2.4}
$$

In Figure 2.1 the literals in white nodes are from the current decision level and the gray nodes are from previous decision levels. Literal $\bar{x}_8$ is a decision literal and conflicting literals x_4 and $\bar{x}_4$ are marked with dashed circles. The learned clauses are $\{\bar{x}_3, \bar{x}_5\}$ for 1UIP and $\{\bar{x}_3, x_8, \bar{x}_9, x_{10}\}$ for Last UIP.

For a learning implementation of the DPLL algorithm, functions *ResolveConflict*() and *Backtrack*() in Algorithms 2 and 4 are replaced with corresponding functions from Algorithm 5. The call to *Backtrack*() does not need to handle the decision variable of the highest decision level differently because the new search space is discovered by the propagation.

```
function ResolveConflict()
    if d = 0
        return false
    end if
    learned := Construct a clause from the conflict
    C := C ∪ {learned}    /* add the clause to
                             learned clauses */
    d' := Find the second highest decision level from
    learned
    Backtrack(d')
    return true.

function Backtrack(d_new)
    foreach x s.t. ⟨x, d'⟩ ∈ dl and d' > d_new
        dl := dl \ {⟨x, d'⟩}
        D := D \ {x}
        P := P \ {x, x̄}
    end foreach
    d := d_new
    return.
```

Algorithm 5: *ResolveConflict*() and *Backtrack*() in DPLL with learning

3 PARALLELIZATION

Our model of the *parallel execution environment*, or simply the *environment*, offers a simple interface between the client sending *executions*, i.e., executable programs together with their inputs, and the environment receiving and running the executions. The only functionality available to the client are

1. Send, which sends an execution to the environment,
2. Monitor, which reports the state of the execution, and
3. Receive, which returns the result of an execution.

The interface is depicted in Figure 3.1. During the execution, the environment does not support any communication from the client to the environment, apart from the capability of monitoring the state of the executions. We do not either assume that the executions are able to communicate directly with each other. In addition to defining the interface, we also assume that the parallel execution environment has some maximum amount of simultaneous executions it can hold. If this limit is reached, the environment is *saturated* and any new executions sent to the environment when it is saturated might fail without a result.

Any computer program in this environment requires the executions to be autonomous in the sense that once the execution has been constructed by the client and sent to the environment, the execution cannot be further guided.

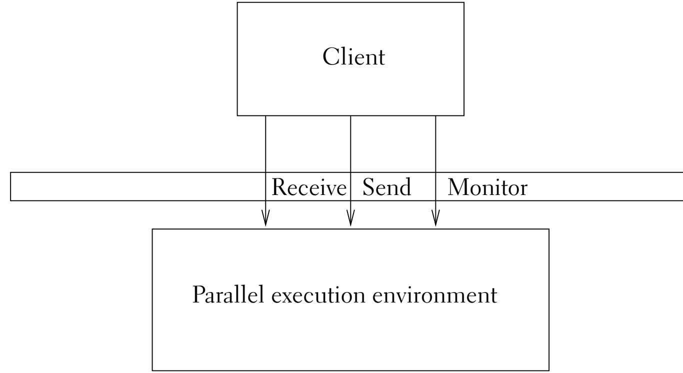

Figure 3.1: The interface between the client and the parallel executing environment

The execution must finish when some condition given at its construction time is triggered.

For this environment, we propose a distribution scheme called *scattering* for solving propositional satisfiability problems. Scattering constructs from a CNF formula F a given number of *scattered CNF formulas* by using a *scattering rule* and applies the scattering rule recursively to these resulting CNF formulas. Conceptually this leads to a *scattering tree* $\mathcal{T}$. In our implementation, scattering aims at adding constraints (new clauses) to the formula so that the propagation and learning of a DPLL algorithm can efficiently prune the search.

The scattering rule constructs a set of scattered CNF formulas from a formula F by conjuncting F with a set of clauses, called *scattering assumptions*. Scattering assumptions of the formula must satisfy three conditions.

(i) The disjunction of the scattered CNF formulas must be logically equivalent to the formula F, so that

$$F \equiv \bigvee_{i=1}^{n} F_i$$

for the n scattered formulas $F_i = F \wedge S_i$.

(ii) Every scattering assumption S_i must have at least one clause that is not in F.

(iii) All variables in scattering assumptions must appear in the formula F.

The condition (i) is related to the *completeness* and *soundness* of the algorithm, whereas the conditions $(ii$–$iii)$ guarantee that the resulting scattering tree is finite.

The number n of scattered CNF formulas in (i) from one application of the scattering rule is the *scattering factor* sf. The scattering factor depends, for example, on the probability of failures in the parallel execution environment as presented in Section 4.2.1.

In a scattering tree $\mathcal{T}$ the nodes are labelled with scattered formulas. The root of the tree is labelled with a CNF formula F_r and given a node labelled

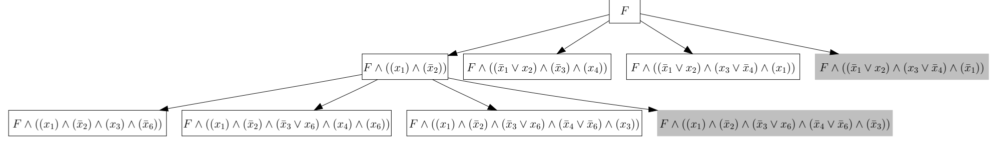

Figure 3.2: A part of a scattering tree for formula (2.4)

with F, the children are the scattered formulas $F_1, \dots F_{sf}$ of the formula F given by the scattering rule. A part of a possible scattering tree for Formula (2.4) is given in Figure 3.2. The tree will extend from the root until no more CNF formulas can be added to any of the nodes by the scattering rule.

Due to the large size of the scattering tree, it is usually not practical to construct the full tree explicitly. A CNF formula F in the root of the scattering tree is unsatisfiable if the path from every leaf to the root of the tree passes through at least one CNF formula which is unsatisfiable. On the other hand, the formula F is satisfiable if at least one of the scattered formulas is satisfiable. These follow directly from the equivalence of the scattered problems in (i). The scattering tree $\mathcal{T}$ can be arbitrarily cut horizontally by removing all edges extending from a set of nodes $\mathcal{F}$ having the property that none of the nodes are in ancestor relation with each other and every path from the root to the leaves of the tree go through exactly one of the nodes from $\mathcal{F}$.[1] Due to the equivalence property of the scattering rule and the associativity of disjunction, it holds for the cut $\mathcal{F}$ that

$$F \equiv \bigvee_{F_i \in \mathcal{F}} F_i. \tag{3.1}$$

Thus showing that every formula in the set $\mathcal{F}$ is unsatisfiable is sufficient to show that the formula F is unsatisfiable, and showing that at least one of the problems is satisfiable is sufficient to show that the formula F is satisfiable.

In Section 3.2 we will give a method for constructing the scattered CNF formulas which complies to the conditions $(i$–$iii)$ given above. In Section 3.3 we give a method for organizing the construction of the scattering tree which will eventually give us a cut $\mathcal{F}$.

3.1 THE SCATTERING RULE

The application of the scattering rule to a CNF formula F results in a set of formulas $F_1, \dots, F_{sf}$, where sf is the scattering factor, a variable depending on the scattering process. The formulas F_i are of form $F \wedge S_i$ where S_i is a scattering assumption and $S_i \not\subseteq F$. Assuming no more than syntactical knowledge of the formula F (e.g., the satisfiability or unsatisfiability of the formula F is not known a priori), it is tempting to ease the condition (i) above by stating for the scattering assumptions S_i that

(i') $\models S_1 \vee \dots \vee S_{sf}$.

[1]This could be expressed as $\forall F_i, F_j \in \mathcal{F}$ it holds that $F_i \not\prec F_j$ and $F_j \not\prec F_i$, and $\forall F_i \in \mathcal{T} \quad \exists F_j \in \mathcal{F}$ such that $F_i \prec F_j$ or $F_j \prec F_i$ where $\prec$ is the usual ancestor relation imposed by the scattering tree

Note that (i') implies (i). To avoid duplicating work for the solving of scattered formulas $F_1, \ldots, F_{sf}$ of a formula F, we add an extra condition,

(iv) $\forall i \neq j :$ for all truth assignments P_L, $P_L \models S_i$ implies $P_L \not\models S_j$.

We can give the following definition for the scattering formulas F_i of the formula F:

$$F_i = \begin{cases} F \wedge T_1 & \text{if } i = 1 \\ F \wedge \neg T_1 \wedge \cdots \wedge \neg T_{i-1} \wedge T_i & \text{if } 1 < i < sf \\ F \wedge \neg T_1 \wedge \cdots \wedge \neg T_{sf-1} & \text{if } i = sf, \end{cases} \tag{3.2}$$

where the CNF formulas T_i are sets of clauses $((l_1^i) \wedge \cdots \wedge (l_{d_i}^i))$, and the negated formulas $\neg T_i$ consist of a single clause, specifically $(\bar{l}_1^i, \vee \cdots \vee \bar{l}_{d_i}^i)$, and $d_i > 0$ for all $i = 1, 2, \ldots, sf - 1$.

Theorem 1 *Equation (3.2) guarantees that* $\models \bigvee_{i=1}^{sf} S_i$, *where* S_i *are the scattering assumptions, and thus complies to condition* (i') *of the scattering rule. Furthermore, the formulas* F_i *comply to condition* (iv) *of the scattering rule.*

Proof In order to prove the condition (i'), we must show that

$$\begin{aligned} \models \quad & T_1 \vee \cdots \vee (\neg T_1 \wedge \cdots \wedge \neg T_{i-1} \wedge T_i) \vee \cdots \vee \\ & \vee(\neg T_1 \wedge \cdots \wedge \neg T_{sf-2} \wedge T_{sf-1}) \vee (\neg T_1 \wedge \cdots \wedge \neg T_{sf-1}), \end{aligned} \tag{3.3}$$

The expression when the two last disjuncts are combined, is logically equivalent to

$$\begin{aligned} & T_1 \vee \cdots \vee (\neg T_1 \wedge \cdots \wedge \neg T_{i-1} \wedge T_i) \vee \cdots \vee \\ & \vee(\neg T_1 \wedge \cdots \wedge \neg T_{sf-2} \wedge (T_{sf-1} \vee \neg T_{sf-1})). \end{aligned} \tag{3.4}$$

The last conjunct of the last disjunct is now a tautology, so we can rewrite the above expression as

$$T_1 \vee \cdots \vee (\neg T_1 \wedge \cdots \wedge \neg T_{i-1} \wedge T_i) \vee \cdots \vee (\neg T_1 \wedge \cdots \wedge \neg T_{sf-2}). \tag{3.5}$$

Continuing as above, removing the terms from the right, we finally get the tautology

$$T_1 \vee \neg T_1 \tag{3.6}$$

which shows that Equation (3.3) does indeed hold.

We can see that the process in Equation (3.2) complies to the condition (iv). Suppose first that $i < j$ in (iv). If T_i is satisfied by P_L, then since $\neg T_i$ must hold in S_j, $P_L \not\models S_j$. If $i > j$ in (iv), we know that $\neg T_j$ is satisfied by P_L. By definition, $P_L \not\models T_j$ and thus $P_L \not\models S_j$. □

Note that the proof does not depend on the formulas T_i to consist of unit literals.

3.2 IMPLEMENTING THE SCATTERING RULE

The method we use to construct the scattered formulas of the form given in (3.2) is based on the DPLL algorithm. The algorithm, given the formula

F and the maximum scattering factor sf_m as input, starts propagating and choosing literals similarly to the DPLL algorithm. After reaching for the first time a certain decision level d_1, the algorithm outputs the first scattered formula F_1. The formula F_1 consists of the clauses of F and the d_1 decision literals as unit clauses. The algorithm modifies the formula F by inserting a clause consisting of the negated decision literals and backtracks to the decision level 0, starting to construct the scattered formula F_2 in the same way. If the formula at some point becomes unsatisfiable on the decision level 0, the algorithm returns the formulas constructed so far. Otherwise the algorithm continues until $sf_m - 1$ scattered formulas have been constructed. Then the final formula is created, which is the formula F with clauses consisting of the negated previous decision literals and no new decision literals.

The scattering algorithm is presented in Algorithms 6, 7, 8 and 9, and it uses the implementations of *Propagate*(), *ChooseLit*(), *ResolveConflict*() and *Backtrack*(), previously given in Algorithms 1, 3 and 5.

The function *ScatterProblem*() in Algorithm 6 is a slight modification of basic DPLL algorithm *IterativeDPLL*() in Algorithm 1. It takes a CNF formula F and a maximum scattering factor *sf* as input and returns a list of scattered CNF formulas constructed from the formula F, or a solution to the formula. First, an initialization of the heuristic function is performed in a call to *Preprocess*(). This is called the *tuning of the heuristic function*. A possible implementation for *Preprocess*() would be to run the DPLL algorithm until some predefined condition is triggered, and then use the heuristic function and learned clauses thus obtained in the rest of the algorithm. We will assume this approach in the rest of the work. The outer loop is repeated until the amount of scattered instances is one less than the scattering factor. If the call to *ResolveConflict*() does not succeed, the function returns the scattered formulas it has constructed to that point. This corresponds to the situation where the working formula, after the insertion of the scattering assumptions, becomes unsatisfiable on decision level 0.

The main difference when compared to *IterativeDPLL*() is the block starting with the call to *ScatterLevel*(). The function *ScatterLevel*() returns true if the current decision level is deep enough for producing a scattered instance. A possible implementation given in Algorithm 7 is explained in Section 3.4. The call to *CreateFormula*() returns a scattered instance F' which is appended to the list of scattered instances. The call to *ExcludeAssumptions*() modifies the formula F to exclude the assumptions of F', after which the DPLL algorithm backtracks to the lowest decision level.

The function *CreateFormula*(), as given in Algorithm 9, creates the scattered instance containing the current decision literals as unit clauses. After the construction of the scattered CNF formula, *ExcludeAssumptions*() in Algorithm 8 inserts the current decision literals negated as a clause.

Theorem 2 *The method of constructing the scattered formulas F_i described above complies to the conditions (ii) and (iii).*

Proof The condition (ii) follows from the fact that the unit clauses inserted to the formula F are inserted after the initial unit propagation, and since the decision level $d_i > 0$, some new unit clauses are always inserted to the

Global vars: P, the partial truth assignment; dl, decision levels of variables; C, learned clauses; D, decision vars; d, decision level; I, instances created so far

Input: F, the problem instance; sf, the maximum number of parallel instances to produce.

Output: I, a list containing at most sf scattering instances; or if a solution was found, a CNF consisting of the literals in P as unit clauses.

```
function ScatterProblem(F, sf)
  P := {}                 /* Partial truth assignment */
  dl := {}                /* Decision levels of vars */
  C := {}                 /* Vars tried both ways */
  D := {}                 /* Decision vars */
  d := 0                  /* Decision level */
  I := []                 /* List of instances cre-
                             ated so far */
  Preprocess()            /* Tune the heuristic */
  while |I| < sf − 1
    while not Propagate()
      if not ResolveConflict()
        return ⟨"scatter", I⟩
      end if
    end while
    if ScatterLevel()
      F′ := CreateFormula()
      append [F′] to the end of I
      ExcludeAssumptions()
      Backtrack(0)
    else if not ChooseLit()
      return ⟨"solution", the literals of P⟩
    end if
  end while
  F′ := CreateFormula()
  append F′ to the end of I
  return ⟨"scatter", I⟩.
```

Algorithm 6: The Scattering algorithm

Global vars: d, the decision level; sf, scattering factor; $|I|$, the number of instances created so far

function *ScatterLevel*()
  if $\frac{1}{2^d} \leq \frac{1}{sf-|I|+1} < \frac{1}{2^{d-1}}$
    return true
  end if
  return false.

Algorithm 7: Determine whether the current decision level is deep enough so that the scattering assumptions lead to equal partition of the solution space.

Global vars: P, the partial truth assignment; D, decision vars; F, the problem instance

function *ExcludeAssumptions*()
  $cl_s := \{\}$
  foreach $l \in P$ appearing in D
    $cl_s := cl_s \cup \{\bar{l}\}$
  end foreach
  $F := F \cup \{cl_s\}$
  return F'.

Algorithm 8: Exclusion of the previous decisions from the formula

Global vars: C, learned clauses; F; the problem instance; D, decision vars; P, the partial truth assignment

function *CreateFormula*()
  foreach $l \in P$ appearing in D
    $F' := F' \cup \{\{l\}\}$
  end foreach
  return F'.

Algorithm 9: Construction of a scattered formula

scattered instance F_i. The formula $\neg T_i$, on the other hand, cannot be already present in the formula F, since this would mean that the unit clauses $\{l_1^i\}, \ldots, \{l_n^i\}$ of T_i would have been added to the instance F which contains the clause $\{\bar{l}_1^i, \ldots, \bar{l}_n^i\}$. This is not possible due to the unit propagation of the DPLL algorithm.

The condition (iii) follows trivially from the fact that the choosing of literals in T_i is done by a DPLL algorithm and the algorithm will never insert literals that do not appear in the clauses of F to the partial truth assignment.
□

The corollary follows now directly from the Theorems 1 and 2.

Corollary 1 *The method described above complies to the conditions* (i–iv) *and is a scattering rule.*

3.3 THE SEARCH

Section 3.2 gives us a method for constructing the scattering tree of a CNF formula F, by using the scattering rule first to the CNF formula F and then recursively to all the resulting scattered formulas. The remaining question of choosing the extent to which the scattering tree is constructed, or equivalently choosing the cut in Equation (3.1), is called the *search* process. One possible approach, constructing the scattering tree as a breadth-first-search guided partially by the results from the previous scattered formulas, is given in this section.

Remembering the assumptions on the parallel execution environment, we know that we are able to send executions to the environment and receive the results after the finishing of an execution. The executions sent to this environment will be SAT solvers with a scattered CNF formula as input. The results from the executions can be used to solve Equation (3.1), and when a sufficient amount of scattered formulas are solved, we can determine the satisfiability of the root CNF formula F. The search will be dynamic in the sense that the results of executions will arrive in an order which is not known in advance, hence the set of formulas sent to the environment will be a superset of the actual set $\mathcal{F}$ used in determining the satisfiability of the formula F in Equation (3.1). A scattered formula F is *known to be* unsatisfiable if the scattering tree rooted at the formula F is shown to be *unsatisfiable* by Equation 3.1, an ancestor of the formula F in the full scattering tree is *unsatisfiable* or the formula F itself is *unsatisfiable*. A factor largely affecting the design of the search algorithm is the assumption that the amount of executions the parallel executing environment can hold before saturating is limited. The time spent in waiting for the environment to become non-saturated should be used in tuning the heuristic of *ScatterProblem*() to yield more balanced subproblems.

The search, given in Algorithm 10, operates as a standard breadth-first-search with a queue of unscattered formulas. Initially the queue contains only the formula F of which satisfiability we are solving. The formulas in branches of the scattering tree known to be *unsatisfiable* are removed from

the queue. When the queue becomes empty, the search terminates indicating an *unsatisfiable* result.

Since the environment is assumed to hold a limited amount of executions, the function *ScatterProblem*() implementing the scattering rule is called to run while the search is waiting for the environment to become non-saturated. As the first step in the while-loop of Algorithm 10, the scattering rule and tuning of the heuristic function is set to run in the background. Then Search waits for the environment to become non-saturated. Finally, an execution is prepared for the environment. This is repeated until all formulas in the queue have been sent, and new scattered formulas must be generated.

The new scattered formulas are collected from the function *ScatterProblem*() and the corresponding formula from which the scattered formulas are scattered is removed from the breadth-first-search queue. In the special case where *ScatterProblem*() has found a satisfying truth assignment for the formula, the search terminates reporting the *satisfiable* result. Otherwise the scattered formulas are appended to the breadth-first-search queue. An *unsatisfiable* result from the function *ScatterProblem*() results in an empty list, which, when appended to the queue effectively removes the corresponding branch from the breadth-first-search. The new scattered formulas are also placed in the scattering tree $\mathcal{T}$.

The last part of the search algorithm updates the scattering tree $\mathcal{T}$ according to the available results from the parallel execution environment. All ancestors of the formulas in the queue are checked for *unsatisfiable* results, and if the formula is thus known to be *unsatisfiable*, it is removed from the breadth-first-search queue. If a *satisfiable* result is found in $\mathcal{T}$, the search terminates and reports *satisfiable*.

3.4 HEURISTIC FOR THE SCATTERING

In solving of any non-trivial CNF formula, a DPLL type algorithm will choose a decision literal in a decision point. While in principle a *legal* variable for the selected literal is any variable appearing in the CNF formula and not appearing in the current partial truth assignment of the algorithm, the selected literal has great impact on the run time of the solver. A *random heuristic* for a DPLL type algorithm selects the literal randomly so that the selection of any literal constructed from a legal variable is equally probable. It is well known that in most cases the random heuristic performs poorly when compared to other heuristics used with DPLL type algorithms.

A widely used heuristic in learning propositional satisfiability solvers based on the DPLL algorithm is the VSIDS-heuristic, originally introduced in [39]. The heuristic maintains a weight VSIDS(l), initially 0, for each literal l. The weight is incremented each time a clause containing the literal is added to the formula. Periodically all the weights are divided by some constant greater than one. The heaviest free literal is chosen in decision points. This heuristic has shown to be effective and its variants are widely used in modern SAT solvers [45, 24].

We propose a variant of the VSIDS heuristic to be used while constructing the scattered instances (in Algorithm 3). The heuristic first calculates the

```
function Search(F, sf)
  queue := [F]
  T := a tree consisting of the root F
  while queue is not empty
    F_scatter := the first formula in queue
    initiate ScatterProblem(F_scatter, sf)
    while there are unsent formulas in queue
      F_send := an unsent formula from queue
      wait for the environment to become non-saturated
      make the formula F_send available for the environment
    end while
    ⟨result, list⟩ := the results from ScatterProblem()
    remove the first formula in queue
    if result = "solution"
      return satisfiable
    else if result = "scatter"
      append list to the end of queue
      make formulas in list as children of F_scatter in T
    end if
    receive any results available from the environment
    update T to contain the new results
    foreach F_s in queue
      if ancestor of F_s in T is unsatisfiable
        remove F_s from queue
      else if ancestor of F_s in T is satisfiable
        return satisfiable
      end if
    end foreach
  end while
  return unsatisfiable.
```

Algorithm 10: Search algorithm

minimum VSIDS() heuristic value $v_{\text{min}} = \min\{\text{VSIDS}(v), \text{VSIDS}(\bar{v})\}$ for every *free* variable v. A variable v with highest v_{min} is then selected as the literal v or $\bar{v}$, depending on which of the two has the highest VSIDS()-score. Ties are broken randomly. The modified heuristic is called MINMAX. The reasoning behind the choice of MINMAX is that we try to find the variables splitting the solution space into equally sized subspaces. Whereas VSIDS always chooses the most promising literal, MINMAX also considers the turn-sides of the choices, with the aim of better balancing of the CNF formulas $F(\{v\})$ and $F(\{\bar{v}\})$.

The heuristic used in the scattering algorithm is very similar to the VSIDS heuristic widely used in contemporary implementations of the DPLL type algorithms. The heuristic gives high scores to literals that appear in the recently learned clauses. The goals of a DPLL splitting heuristic are [58]

1. for satisfiable instances,
   (a) force conflicts quickly to prune search space where solution does not exist and
   (b) satisfy as many clauses as possible to find the satisfying assignment.
2. For unsatisfiable instances
   (a) construct short refutation by quickly propagating conflicts,

whereas the goal for a scattering heuristic is

1. identify the variables dividing the problem space into parts with equally large solution spaces (taking into account the effect of propagation).

The more greedy splitting heuristic of a DPLL type algorithm is not ideal for scattering. A specific heuristic tuned for the needs of the scattering algorithm should concentrate on creating as uniformly difficult problems as possible while still diminishing the solution space.

The set of possible truth assignments for a formula with $|V|$ variables consists of $2^{|V|}$ different truth assignments. A DPLL algorithm will not go through all of them even if the formula is unsatisfiable, due to unit propagation and possibly non-chronological backtracking. We may still approximate the number of searched truth assignments and the run time of a DPLL type algorithm to be on the order $\mathcal{O}(2^{|V|})$. Fixing one literal from all the truth assignment to *true* will half the set of possible truth assignments explored by a DPLL type algorithm to $\mathcal{O}(2^{|V|-1})$.

Returning to the Formula (3.2), where the number of fixed literals in scattered formulas is d_i, the decision levels d_i are selected so that the formula is heuristically divided into scattered formulas for which the DPLL algorithm will explore equally many different truth assignments. In dividing the formula F to sf scattered formulas with the same run time, the run time of a single scattered instance should be $\frac{t(F)}{sf}$, where $t(F)$ is a function from the CNF formula F to the solving time of F. We can write this in the form

$$\frac{t(F)}{sf} = \Big(t(F) - (i-1)\frac{t(F)}{sf}\Big)r_i \quad i \in \{1, 2, \ldots, sf\}, \tag{3.7}$$

Table 3.1: Scattering using over- and under-constraining approximations

Over-constraining			Under-constraining		
Fraction	Numeric	Dec. level	Fraction	Numeric	Dec. level
$\frac{1}{4}$	0.25	2	$\frac{1}{4}$	0.25	2
$\frac{1}{4}$	0.19	2	$\frac{1}{2}$	0.38	1
$\frac{1}{2}$	0.28	1	$\frac{1}{2}$	0.19	1
1	0.28	0	1	0.19	0

since the run times of the already constructed $i-1$ problems must be subtracted from the total run time and r_i is the run time ratio the scattering assumption should lead to. When solved for r_i, we get

$$r_i = \frac{1}{sf - i + 1} \quad i \in \{1, 2, \ldots, sf\}. \tag{3.8}$$

The scattered CNF formulas are constructed by adding literals, and as a result, r_i must be approximated with $\frac{1}{2^{d_i}}$, where d_i is the decision level, or the amount of literals in T_i on Equation (3.2).

Example 3 *When scattering 4 formulas from a formula, Equation (3.8) gives fractions $r_1 = 1/4, r_2 = 1/3, r_3 = 1/2, r_4 = 1$. Assuming that selecting one literal will half the solving time, we get $\frac{1}{2^2} \leq \frac{1}{4} \leq \frac{1}{2^2}, \frac{1}{2^2} \leq \frac{1}{3} \leq \frac{1}{2^1}, \frac{1}{2^1} \leq \frac{1}{2} \leq \frac{1}{2^1}$ and $\frac{1}{2^0} \leq 1 \leq \frac{1}{2^0}$. From this we get that the first decision level (or equivalently, the number of unit clauses in T_1) is 2, the second is 1 or 2, for the the third formula a single literal must be chosen and for the fourth we need no literals. In Table 3.1, two alternative approaches are presented for the given example. On both tables, the left column represents the absolute fraction given by the decision level on the right column. The middle column is the actual fraction of the total problem given by the selections. If the fractioning were optimal, all values should be equal to $\frac{1}{4}$. The rows are ordered by increasing scattered formula index, so that the first scattered formula is on the top row and the last is on the bottom row. The partial scattering tree given in Figure 3.2 is constructed by applying the over-constraining approximation.*

An implementation of an over-constraining *ScatterLevel*() is given in the Algorithm 7.

3.5 LEARNING IN SCATTERING

As a side effect of the tuning of the heuristic function in the call to *Preprocess*() in Algorithm 6, the scattering rule also produces learned clauses. Since learned clauses are known to speed up solving of CNF formulas, it is tempting to include the clauses learned from one application of the scattering rule to other formulas.

Learned clauses are logical consequences of the set of clauses from which they are learned, hence on one application of the scattering rule, previously learned clauses are also logical consequences of the next scattered instance.

However, learned clauses from one application of the scattering rule to a formula F are not necessary logical consequences of another formula F', unless the formula $F \subseteq F'$. Whereas the clauses from which the learned clauses are derived could be recorded, the effort might overweight the profit gained from the learned clauses.

In order to use the learned clauses in other formulas, we propose a scheme based on the scattering tree. The learned clauses from formula F are included to the formula F' iff F is ancestor of F' in the scattering tree.

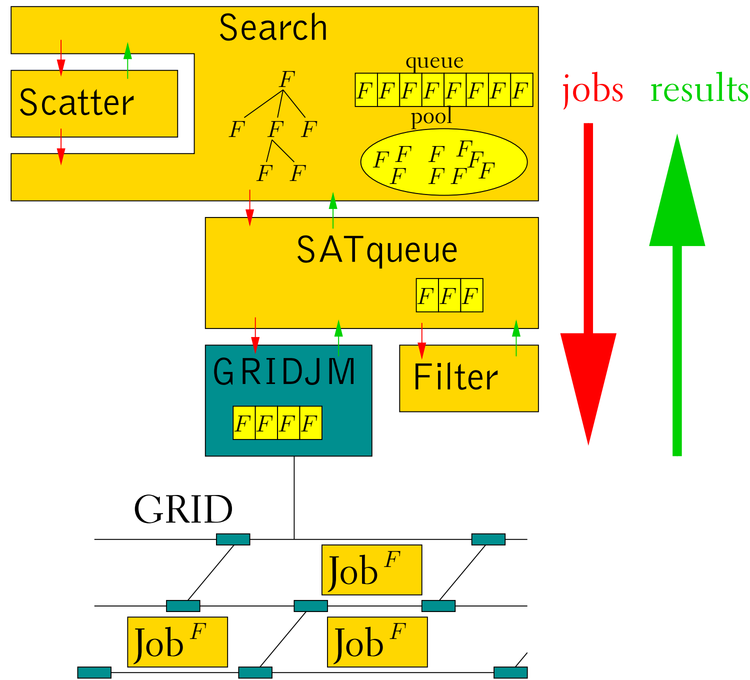

Figure 4.1: The Program Architecture

4 ARCHITECTURE

We have implemented the scattering algorithm presented in the Chapter 3 and the additional components required to apply the parallelization scheme in a computational grid. The implementation is named the *SATU (SAT Ubiquitous) distributed SAT solver*. SATU takes as input a CNF formula F, called the *original formula* and outputs a satisfying truth assignment for F if the formula is shown to be *satisfiable*, an indication that the formula has no satisfying truth assignment if the formula is *unsatisfiable*, or a timeout if the result could not be determined before a given time limit.

The distributed SAT solver SATU consists of five distinct processes communicating with each other through network and Unix domain sockets. The overview of the architecture is presented in Figure 4.1. In the context of the architecture, the executions consisting of a SAT solver and a scattered formula are called *jobs*. The original formula is given to Search, which sends the formula to SATqueue and constructs the scattered formulas from it with Scatter. When a formula arrives at SATqueue, Filter tries to solve it until it is sent to GRIDJM which will finally deliver it to the grid. The process Search receives the results and uses them to guide the further search.

The main functionality of the parallelization scheme described in Chapter 3 is implemented in Search and Scatter. In addition to the functionality described in Chapter 3, the implementation also includes the interface to the grid, GRIDJM, and a local satisfiability solver, Filter, acting as a preprocessor which aims at filtering easy jobs from being sent to the grid.

The processes communicate with a simple protocol, exchanging the CNF formulas, or *jobs* and results, as indicated by the arcs in Figure 4.1. Sockets

are used for communication to obtain portability and to facilitate experimentation with different designs. Implementing the communication via network sockets gives us the additional advantage of being able to run the processes of SATU in distinct computers.

The scattering formulas are passed to Filter where a satisfiability solver gets them as input. The formulas are solved for some time, until GRIDJM initiates a solving process in the grid. If the satisfiability of the formula is determined in Filter, the formula is not sent to the grid, and the result is communicated to Search. Results coming from GRIDJM are also passed to Search, which based on the results determines where the search should continue in the scattering tree, if the result of the original instance is not known so far.

In Section 4.1 we describe the implementation choices related to the construction of the scattering tree. The interface to the computing environment and the optimizations related to the sending of jobs is covered in Section 4.2.

4.1 SCATTERING FUNCTIONALITY

The scattering described in Chapter 3 is implemented in Search and Scatter. The main architectural difference is that in addition to constructing the scattering tree, the implementation must also adapt to the delays of a parallel computing environment.

Search calls Scatter as a subroutine, and expects as a return value a list of scattered instances. We assume that the construction of scattered instances is a heuristic process, which, on average, gives better results in terms of even distribution of the run time when more time is given for the calculations relating to the heuristic. Since it is realistic to assume that the initiation of executions in the parallel computing environment takes some amount of time, this time should be used for heuristical calculations in Scatter. The architecture is designed so that a CNF formula is sent early for scattering and the scattered instances are collected only when resources for parallel execution become available.

4.1.1 Scatter

The process Scatter is an implementation of *ScatterProblem*() (Algorithm 6 in Chapter 3). It receives a CNF formula, constructs *sf* scattered formulas from the formula and sends them back to Search after receiving a signal indicating that Search requires more jobs. The implementation is designed to use the time between receiving the instance and producing the scattered formulas for two intermixed purposes. The most significant part from the point of view of the scattering is the tuning of the heuristic function, which is used to select the most promising literals for the scattering assumptions. The tuning is done in the call to *Preprocess*() in Algorithm 6. In SATU, the tuning is implemented by running a learning DPLL algorithm on the formula with the MINMAX heuristic. The heuristic values of literals are then passed to the actual scattering. As a side effect, Scatter also learns new clauses, and the clauses can be passed on to the scattered instances. If Scatter

finds the formula *satisfiable* or *unsatisfiable* during this preprocessing stage, it immediately reports the result to Search. This is an optimization, since the result would be reported by the time Search queries for the scattered formulas. However, it was noted early on the testing that on some problems, the time lost in waiting for the query was significant.

4.1.2 Search

The process Search corresponds to the *Search*()-function (Algorithm 10 in Chapter 3). It initiates the solving when the original CNF formula is set. The process maintains the scattered formulas in three data structures. The *search queue* is the queue for the breadth-first-search of the scattering tree. The *formula pool* is a buffer from which the computing environment is served the scattered instances. The *scattering tree* is the actual structure from which the satisfiability of the original formula can finally be deduced.

The formulas are organized as a scattering tree, so that every result of an ancestor formula also applies to child formulas. This optimizes the representation of the scattering assumptions as well as implements the learning scheme outlined in Section 3.5. Most importantly, if a formula is in the search queue and an ancestor of the formula is found to be *unsatisfiable*, the formula can be removed from the queue, which prunes the search significantly.

After receiving a CNF formula, Search sends the formula to Scatter for scattering and places it to the *formula pool* of Search. When the computing environment indicates that resources are available, the process Search sends a formula for which the result is not known from the pool to the parallel environment. If the size of the pool goes below a given limit, the *minimum pool threshold*, Scatter is signalled to stop the scattering and new scattered instances are read from it.

In addition to placing the formulas into the formula pool, Search also places them to the search queue. The first formula for which the satisfiability is not known in the search queue is sent to Scatter and the process starts again, working effectively as a breadth first search where the neighbouring nodes are determined by Scatter.

The process Search works as a server listening to results. Possible results are a satisfying truth assignment to the formula, result that a scattered formula has no satisfying truth assignment or that the solving of some formula has timed out. It is also possible that a component below Search has learned new clauses from some formula. The learned clauses can then be inserted to the formula in Search through the server and included in some of the scattered formulas. However, current implementations of the components below Search do not produce learned clauses.

The satisfiability of some formula F in the scattering tree is determined from the satisfiability of the formulas in the tree rooted at the formula F and the ancestors of F. In the implementation, the satisfiability is checked periodically by a depth-first-search of the scattering tree.

4.2 INTERFACE TO THE COMPUTING ENVIRONMENT

The rest of the components, SATqueue, Filter and GRIDJM, are mainly concerned with the interface towards the parallel execution environment. Main problems in the interface are the communication delays related to sending the executions to the environment, and the recovery from the inevitable errors occurring in the environment. The delays make it costly to send executions which only run for a short period of time, and for this purpose the shortest jobs are filtered by the satisfiability solver running in Filter. The errors relating to the parallel environment affect significantly the run time and, if the error rate is sufficiently high with respect to the scattering factor, also the completeness of some possible implementations. The errors are handled in GRIDJM with a resubmission scheme described in Subsection 4.2.1. The communication between GRIDJM, Filter and Search passes through the intermediate process SATqueue.

4.2.1 GRIDJM

SATU is designed to be minimalistic in what comes to the requirements from the underlying computing environment. We expect that the parallel execution environment has some upper limit on how many executions it can simultaneously hold before saturating. The only operations required from the environment are

jobid **send**(*job*) Send a job to the parallel computing environment. The argument *job* is a description of the job to be sent. The return value is *jobid*, a reference uniquely identifying the job in the environment.

state **query**(*jobid*) Query the status of the job specified by *jobid*. The return value *state* is a job state. Possible job states are *running*, the job is running in a node in the environment; *finished*, job has finished correctly and the result can be received; *queued*, the job has been queued because of some congestion in the environment and will possibly enter the *running* state in some time; and *failed*, the job has permanently failed in the environment and there are no results to be received.

output **receive**(*jobid*) Get the *output* of the job from the parallel environment to the local computing environment. The argument *jobid* is the identity of the job as returned by *send*().

The environment does not need to support any kind of interaction during the execution of the job, nor there has to be any direct communication between the jobs. The requirements are designed to suit most of the currently available computational grids.

The jobs sent to the grid consist of a satisfiability solver and the problem instance to be solved. The solver will terminate in the grid after a certain timeout if it has not been able to show the instance *satisfiable* or *unsatisfiable*. The same type of a limit also exists for the memory consumption of the solver.

GRIDJM requests jobs from Search until the number of jobs in the parallel computing environment reaches the saturation limit, *maximum parallel job limit*, given as a parameter to GRIDJM. Jobs arriving as replies to these

requests are first placed in an incoming job queue. This queue is checked periodically for new jobs, which are then sent to the environment one at a time.

The resubmission scheme implemented in SATU tries to minimize the amount of jobs for which no result is eventually received. To detect the failures and finishings of jobs, the status of jobs in the environment is monitored periodically. If the status of a job has not changed for a configurable period of time and the job is not running, or the job has finished reporting an error, it is considered failed and is sent to the environment again. The number of resubmission attempts can be limited. If the job has finished correctly, the results are received and sent upwards through SATqueue.

When the job status update finds a correctly finished job, GRIDJM forks a new process for receiving the results of the finished job. The receiving is done in the background and parallelly. In the beginning of the solving of a difficult formula, it is common that most of the scattered formulas run in the parallel execution environment until the timeout. As a result, the jobs finish approximately at the same time. To avoid this congestion, the timeout value is randomized.

The handling of failed jobs is motivated by the fact that the scattering tree might grow to be very large (infinite by all practical means) if the scattering factor and the failure probability are sufficiently large.

Because of the large number of components in a grid environment, the probability of failure is much higher when compared to a simpler computing environment. Run times of solvers tend to be high, which further increases the amount of failures encountered during the solving of a CNF formula. Such failures could be related to service breaks, network downtimes or any other problems which usually do not affect computer programs with short run times and a single computing node.

If the implementation recovers from failures in the parallel computing environment by simply ignoring the computations that do not finish correctly, the algorithm might not find the solution at all. A scattered formula is further scattered in this case. Since the full scattering tree is usually very large, the scattering will not proceed observably. However, assuming a small enough failure rate and a reasonable scattering factor, the probability of a formula not getting solved eventually is very small.

We may study the behaviour of the scattering in a parallel computing environment in which the probability of the result of a job sent never being received is p. For this purpose we construct an abstract model where the scattered formulas are sent to the environment as oracles immediately replying with an answer to the problem unless the transaction fails. In this abstraction, let us assume that the scattering algorithm in Algorithm 6 (Section 3.2) never gives a *satisfiable* or *unsatisfiable* result and always returns sf scattered instances. This idealization is realistic if the problem is difficult for the scattering algorithm but easy for the satisfiability solvers, which, when the scattering is proceeding normally, should be the case.

Now consider an unsatisfiable problem instance to be solved in a parallel computing environment with the error probability p. The scattering, seen as a stochastic process, can be modelled with a *Galton-Watson Branching Process* [32, pages 91–95]. In a Galton-Watson Branching Process, a population

evolves in generations, and each member of a generation g produces a family of X children of generation $g+1$. The number X is a random variable with the following properties:

1. The family sizes of the individuals of the branching process form a collection of independent random variables, and
2. all family sizes have the same probability measure $\mathbb{P}$ and generating function G.

In our application, we are interested in the probability of the *ultimate extinction*, that is, what is the probability that a generation g is empty, when $g \to \infty$.

The number of scattered instances is the number of children in the branching process. When the formula is successfully sent to the computing environment and the result is received, the number of children is 0. If the result is lost, the number of children for the lost formula is equal to the scattering factor sf. Hence, the probability measure for the number of children X is

$$\mathbb{P}(X=k) = \begin{cases} 1-p & \text{if } k=0, \\ p & \text{if } k=sf, \\ 0 & \text{otherwise.} \end{cases} \tag{4.1}$$

Adhering to the Galton-Watson process, we assume that the number of scattered formulas is independent and the probability measure and the generating functions of the numbers of children are the same for all formulas in the scattering tree. The problem of termination of the scattering algorithm in this setting is essentially the problem of ultimate extinction in the branching process. The probability of extinction is 1 almost surely whenever the expectation for the number of children X is less than or equal to 1, or

$$\mathbb{E}X = p \cdot sf \leq 1. \tag{4.2}$$

If the expected number of children is greater than 1, the probability of extinction is the smallest positive root for the equation $s = G(s)$, where $G(s) = 1 - p + p \cdot s^{sf}$ is the generating function of the probability measure (4.1). For failure probability $p = 0.2$ and scattering factor $sf = 7$, the probability of termination is approximately 0.8851.

Another question of interest, which is related to the efficiency of the scattering algorithm in the presence of grid failures is the height of the scattering tree. The probability of the scattering tree in the above setting to be of height less than or equal to some $i \in \mathbb{N}$ can be expressed as a non-linear recurrence equation. Assume again that the probability of failure of a job is p. Then the probability of a job not failing is $1-p$. For a certain scattered formula, the probability of the corresponding scattering tree being solved is the probability of the root formula being solved together with the probability of the root formula failing to be solved and all its subtrees being solved. Now we can build a recurrence equation for the probability of an unsatisfying formula getting solved at latest on the scattering level i ($i \geq 1$):

$$R[i] = \begin{cases} 1-p & \text{if } i=1, \\ pR[i-1]^{sf} + 1 - p & \text{if } i \geq 1. \end{cases} \tag{4.3}$$

Theoretically the behaviour of the algorithm with respect to completeness is very sensitive to the failure rate but in practice this does not pose a problem. The algorithm will eventually terminate in either Scatter or Filter giving the result, when enough scattering assumptions and restrictions have been inserted to the problem to make it trivial to solve. However, the effect on the run time of the solving is significant.

Due to the sensitivity to transmission errors of the scattering, it is important that GRIDJM minimises the probability of disappearing job results.

4.2.2 Filter

The motivation of the local solving comes from the fact that the communication delays in sending jobs to the parallel computing environment impose certain constraints on whether sending a job to the environment is in fact more efficient than solving the job locally. To justify this argument and to give some weak lower limit for the run times of jobs which should be sent to the grid, we construct a model from our interface to the parallel computing environment.

In our implementation of GRIDJM, the interface to the parallel environment is designed so that sending is always done sequentially and receiving is done parallelly. Assume that the communication delay associated with sending a job to the parallel computing environment is d_1, and the receiving delay is d_2. If we have n jobs of run time r, the jobs can be locally solved in time rn. If the job is sent to a parallel computing environment, however, we expect to get the job solved in time less than this. The additional communication delays make the total solving time when using the parallel environment to be $nd_1 + r + d_2$, since the last job is sent to the environment after n delays, completes in time r and has to be received. We can derive the lower limit for the run time r from the equation $nd_1 + r + d_2 < rn$. When solved for r and letting n tend to infinity, we get

$$r > \lim_{n \to \infty} \left(\frac{n}{n-1} d_1 + \frac{1}{n-1} d_2 \right), \tag{4.4}$$

which gives

$$r > d_1, \tag{4.5}$$

that is, the run time must be greater than the sending delay. Since Filter solves the jobs with low run time locally, the jobs sent to the grid should have run times r that are more likely greater than the sending delay d_1.

4.2.3 SATqueue

The process SATqueue requests jobs from Search. It places the jobs to a queue, from which the first job is always available to GRIDJM. If the environment is currently saturated, a copy of the first job is handed to Filter for solving until the environment is no longer saturated. Only if Filter does not succeed in determining the satisfiability of the formula before the environment becomes non-saturated, Filter is signalled to stop and the formula is sent to GRIDJM. If the instance is solved before it gets sent to GRIDJM, it

is removed from the queue and the result is sent to Search. Solutions arriving from GRIDJM and Filter are collected and sent to Search as a batch periodically.

4.3 THE PARAMETERS OF SATU

As a conclusion, we list here the parameters of SATU.

Scatter *Scattering factor sf*; the number of scattered instances to produce.

Search *Minimum pool threshold*; when the size of the formula pool goes below this limit, Scatter is signalled to produce the scattered formulas. *Scattering tree monitor delay*; the delay between periodical monitoring of the scattering tree.

GRIDJM *Maximum parallel job limit*; the number of executions the environment can simultaneously hold. *Inactivity time*; the time a job can be in a non-running state before it is considered failed. *Resubmission attempts*; the number of times a failed job is resubmitted to the environment. *Monitor delay*; the delay between periodical monitoring of the states of the jobs in the environment. *Job timeout*; the maximum time a job can run in the environment.

Filter No parameters.

SATqueue *Size of formula queue*; the number of formulas in the queue. *Result sending delay*; the delay between periodical result sending to Search. *Result receiving delay*; the delay between periodical result receiving from GRIDJM.

5 EXPERIMENTAL RESULTS

To demonstrate the effectiveness of the presented ideas in checking the satisfiability of propositional formulas, we test SATU in a production-level computational grid. The tests concentrate on two aspects of the solver, the efficiency of MINMAX heuristic in scattering, and the scalability of the solver with respect to the available resources. The MINMAX heuristic described in Chapter 3 is tested against a random heuristic. In MINMAX heuristic we implement the heuristic tuning and learning. When using the random heuristic, the heuristic tuning is meaningless and learning is slow, hence it was decided that in this case scattering is done immediately after receiving the formula. The scalability of the solving is tested by increasing the upper limit of simultaneous jobs in the grid. All the tests are run with the maximum parallel job limits of 4, 8, 16, 32 and 64.

The results are obtained by running Search, Scatter, SATqueue and GRIDJM in a 500 MHz Pentium III with 500 MB of memory, running Linux kernel 2.4.26. To measure the effect of the heuristic in Scatter separately from Filter, Filter is run on a different computer than Scatter. If both processes are run on the same single processor machine, they compete for the same processor resources, which gives advantage to the computationally less demanding random heuristic, compared to MINMAX heuristic. In the tests, we run Filter on an AMD Athlon™ XP 2800+ with 512 KB of L2 cache and 1 GB of memory. As the computational grid, we use the Linux-clusters of NorduGrid [25], a grid running the ARC grid middleware [51] developed by the NorduGrid collaboration [41]. The cluster node types, that is, the different types of computers we were able to submit jobs during the benchmarking are characterized in Table 5.1.[1] Due to the dynamic nature of the grid, the list changes during the testing. To give an overview of the perceivable configuration of the grid, the number of jobs sent to each node type is also provided.

The scattering factor sf of the scattering algorithm is set to 7 after some initial testing. Minimum pool threshold in Search is chosen to be 8 so that it is sufficiently large and is not equal to the scattering factor. The size of the formula queue in SATqueue is 3. The monitor delay of GRIDJM is set to 60 seconds and the result receiving delay of SATqueue is 5 seconds. The result sending delay is 2 seconds. The scattering tree monitor delay is one second. Note that even though the grid state is monitored only every minute, the results might come at a more rapid pace depending on the progress of the receiving processes and results coming from Scatter and Filter. For completeness, we repeat here that the maximum parallel job limit is used for demonstrating the scalability of the solving. Inactivity time is 300 seconds and a single resubmission attempt is made for a failed job. Except in Section 5.2, the job timeout is randomized and gets values from the interval between 50 and 70 minutes.

Learning is implemented so that only the clauses learned in Scatter are passed further to the scattered formulas, as described in Chapter 3. Clauses

[1] If the computing node has more than one processor, the properties of one of them is displayed.

Table 5.1: Grid nodes used in benchmarks

Type	CPUs	MHz	L2 Cache	Jobs
Intel(R) Pentium(R) 4 CPU 2.80GHz	2	2790	512 KB	21740
Pentium III (Coppermine)	2	730	256 KB	5054
Pentium III (Coppermine)	2	800	256 KB	3808
Intel(R) XEON(TM) CPU 2.20GHz	2	2200	512 KB	3459
Pentium III (Katmai)	2	450	512 KB	2077
Pentium III (Katmai)	2	600	512 KB	1771
AMD Athlon(tm) XP 1600+	1	1400	256 KB	937
Intel(R) Xeon(TM) CPU 2.40GHz	2	2390	512 KB	854
Pentium III (Coppermine)	1	1000	256 KB	628
Intel(R) Pentium(R) 4 CPU 3.20GHz	2	3200	1024 KB	620
Intel(R) Pentium(R) 4 CPU 2.80GHz	2	2790	1024 KB	610
Intel(R) Pentium(R) 4 CPU 2.80GHz	1	2790	512 KB	263
Pentium III (Coppermine)	2	870	256 KB	129
Intel(R) Pentium(R) 4 CPU 2.66GHz	1	2660	512 KB	97
Intel(R) Pentium(R) 4 CPU 3.00GHz	1	2990	1024 KB	75
				42122

learned in the grid nodes are used only in solving that particular scattered formula they are learned from, and are not sent back to SATU.

The back-end solver is *zChaff* version *Chaff 2004.11.15 simplified* [39], and when sent to the grid it has a timeout of 60 ± 10 minutes, the distribution being uniform. Memory limit for each job is set to 500 MB in the grid. It is unlikely that the solver uses an amount of memory exceeding that in its running time with the test instances used in the benchmarks. Note that this might not be the case with some other benchmarks. The same solver is used as the locally running Filter, but without the job timeout or memory limit.

5.1 THE EFFECT OF THE GRID ERRORS

Even though the fault tolerance mechanisms in SATU assure the completeness of the algorithm under reasonable assumptions on the scattering factor and failure rate in the grid, the failed jobs can have significant effect on the run time of the solver. Typical failures in the grid are due to jobs not getting to run in a reasonable amount of time or failing to execute as a result of low level operating system problems.[2] The great number of heterogeneous clusters in the grid make these problems to some extent inevitable.

The two fault tolerance mechanisms presented in Section 4.2.1 both have their downsides. The limited resubmission uses resources from the sequential job submission in GRIDJM, lowering the overall degree of parallelism. The scattering, on the other hand, increases the number of jobs to be solved in order to close an unsatisfiable branch of the scattering tree, which in a grid with a high failure rate further increases the probability of future scatterings.

[2] A known problem in current NorduGrid is that program files occasionally fail to get execute permissions as a result of an NFS server bug.

Typical failures manifest themselves as a badly working single cluster, where the failure rate is much higher than normally. The other clusters might be working fine during the single failure or it might happen that the increased load in these clusters cause problems as well. The problems are temporally localized. The grid might provide many days of nearly failure free service and suddenly behave badly for a couple of hours. With SATU run times ranging from tens of minutes to 12 hour, this usually means that a single run from a test suite for a certain CNF formula will experience the faults whereas the rest of the runs will run in a nearly perfectly working grid. The effects of a failed cluster are difficult to tell from the highly non-deterministic run times typical to the propositional satisfiability problem itself. This additional uncertainty needs to be taken into account when considering the error limits in the results.

5.2 BENCHMARKING GRIDJM

The interface to the grid is troublesome due to the relatively high upload and download latencies. The effects of two design choices, the parallel download and the random job timeout, to the parallelism are shown in the four diagrams of Figure 5.1. The diagrams illustrate the parallelism observed by the process SATqueue. Each diagram is constructed from a single run, and should be considered as a typical example of the behaviour of the grid.

The small rectangles represent jobs sent to the grid, starting and stopping at some point in time, which advances towards right in the diagrams. The jobs are added to the imaginary parallel tracks starting from the bottom and advancing upwards in each diagram. Two jobs may overlap for at most 5 seconds on each track. Since the sending and receiving is partially done in parallel, this apparent overlapping of tracks should make the picture to more closely conform to the actual parallelism. If the first track has already a running job, the job is added to the second track above it and so on, until a track with free space is found or a new track has to be added above the top track. The level of parallelism can be approximately read from the diagrams. The time is shown on the bottom of each diagram as a line with a small vertical stick every 60 minutes. The maximum parallel job limit in the grid is set to 64 jobs. The submission, which is done serially, causes the small delays showing at the beginning of the tracks. The sending delay for one job is around 25 seconds. The delay depends on the load on the computing nodes and network delays, the job status checks which take place approximately every minute and the occasional failures of jobs resulting in resubmissions. The receiving of the result also causes a delay which can be clearly seen in the diagrams where the receiving is done serially. All the jobs in the tracks are shown to stop simultaneously at the end of the benchmark on the right of the diagrams, and are run for approximately 6 hours. The individual jobs consist of an idle job not consuming processor time but running in the grid for a period of 50 to 70 minutes, depending on the parameters.

It should be noted that in real-world problems, the solving times of single jobs tend to vary throughout the total solving process and the almost simultaneous ending of all jobs illustrates in some sense the worst-case behaviour of

GRIDJM.

The effect of job timeout of 60 minutes and the serial implementation of the job handling loop in GRIDJM is visualized in upper left diagram of Figure 5.1. The results show a sudden drop of the degree of parallelism in the grid after the timeout, when all jobs sent to the grid finish approximately at the same time. The cumulating download delays together with the upload delays result in an increasing gap after the first timeout. The inherent random delays of the grid blur the gap after the first timeout, but it still shows clearly after the second timeout in the upper tracks. The first jobs sent after the first timeout are sent at the timeout rate, which can be seen as the immediate resubmission after the second timeout. After all the timeouting jobs after the first timeout are collected, the submission again proceeds at a faster pace which can be seen in the increasing gap of the upper tracks after the second timeout. Sending of jobs is faster than receiving and the jobs running until the job timeout cannot be replaced in time to keep the gap small.

The delay caused by the sequential downloading and subsequent loss of parallelism can be diminished by randomizing the job timeout. When the job timeouts occur at different times, the overheads associated with the receiving span a wider time period and the accumulation of the delays is not as dominating. The loss of parallelism can still be seen on upper right diagram of Figure 5.1 however, although the effect of accumulated downloads is noticeable only after enough timeouts have occurred. The effect can be further diminished by adding more random behaviour to the timeouts. After the first timeout, the tracks are completely mixed. The timeout used in Figure 5.1 is 60 ± 10 minutes.

The loss of parallelism may also be tackled by parallelizing the receiving of results in GRIDJM. New jobs can then be sent to grid while the results from old jobs are being downloaded. In this implementation the download delay affects also uploading to some degree and in fact hides some of the delay in the job run time. In practice the downloading parallelizes quite well and although the downloading time of n jobs does not quite drop to n:th fraction, it is approximately $n/2$:th fraction when performed parallelly as opposed to serial downloading. The effects are shown in bottom left diagram in Figure 5.1. The gap is noticeable after the first timeout although much less significant than on the top left diagram.

When the two approaches are combined, we get the implementation used in the following benchmarks. The parallelism remains high from the beginning of the run to the end, as is illustrated in bottom right diagram of Figure 5.1. The gap can still be seen after the first job timeouts, but is small and disappears completely on the second job timeouts.

Another phenomenon clearly visible from Figure 5.1 is the error rates in the grid. The failed jobs are shown as rectangles of run times significantly longer than one hour, starting from some point and advancing right to the end of the benchmark. The fraction of failed jobs range from approximately 2% of the bottom left diagram to the maximum of almost 25%:th of the top right diagram. The failure probability has a positive autocorrelation, so that a recent failure makes further failures more probable. The benchmarks resulting in the rightmost diagrams in Figure 5.1 were run subsequently with little time between the runs, whereas the leftmost diagram in which the grid

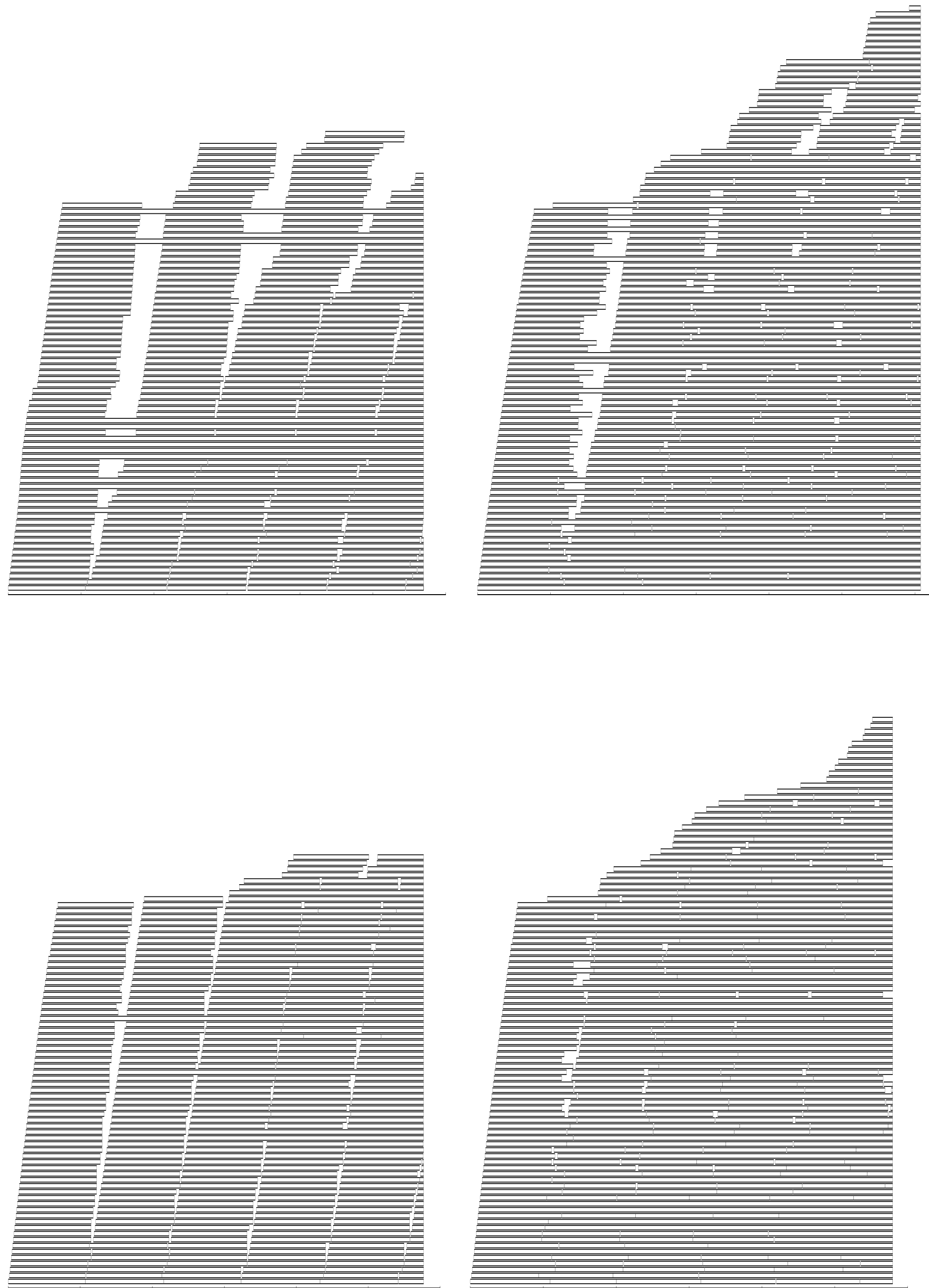

Figure 5.1: Parallelism and time. The horizontal axis shows the time; the scale on the bottom of the diagram shows a small vertical mark every hour. The vertical axis shows the parallelism in the grid. Maximum degree of parallelism is set to 64 and the jobs are placed on imaginary "tracks". On the top diagrams, the implementation of GRIDJM is serial and on the bottom diagrams parallel. On the left column the timeout is fixed to 60 minutes. On right column the timeout is randomized, with uniform distribution in the range 60 ± 10 minutes

behaviour was more reliable were run first and last with some 36 hours between the starts. Since the failed jobs are indicated as jobs with infinite run time in the diagrams, they each occupy one track and the total number of tracks increases as more jobs fail.

5.3 BENCHMARKING SATU

The actual benchmarks were conducted using the implementation of GRIDJM with randomized job timeouts and parallel downloading. This setup corresponds to the bottom right diagram of Figure 5.1.

All run times reported are wall clock times measured from the time the CNF formula was given to SATU to the time SATU reported the result. The speedup is calculated for each individual propositional formula from the wall clock time, using as a reference the time required for solving the formula with maximum parallel job limit of 4 in the grid. The decision of using the reference run time from maximum parallel job limit of 4 was taken due to the high run times of some interesting benchmarks on lower maximum parallel job limits. We also report the speedups of minimum, average, median and maximum jobs for each job type and size. The comparison between the two heuristics (random and MINMAX) is done for the wall clock run times with the maximum parallel job limit of 64 in the grid. Each benchmark runs at most until the timeout of 12 hours, and the benchmarks not finishing until the timeout get the timeout value as the run time. On some occasions, the process GRIDJM runs out of memory due to a large amount of jobs sent to grid. This is assumed to be caused by a memory leak. Also these runs get the maximum run time.

The average speedup over all formulas is given in Figure 5.2. The figure shows that on the benchmarks we used, we get a superlinear speedup when using maximum parallel job limit of 8, when compared to maximum parallel job limit of 4. The average wall clock time available for the process Filter at maximum parallel job limit of 4 is 787.503 seconds, whereas for maximum parallel job limit of 8 it is 309.698 seconds. The numbers are separately calculated from the instrumentation output of Filter. If more time is used in constructing the scattered formulas, the formulas should be more equal in difficulty. However, since *unsatisfiable* results from the grid prune the scattering tree, the aggressive parallelization is more effective. When the maximum parallel job limit is further increased, the speedup increases more slowly.

We can see a slight decrease in median and average speedups when the maximum parallel job limit is raised from 32 to 64. At the same time, the speedup of average increases. The average run times, from which the speedup of average is deduced, are dominated by the high run times of certain formulas. The solving of these difficult formulas can benefit better from the increase of parallelism, whereas the easier formulas suffer from the constant overheads related to communication to the grid. Since easy formulas do not profit from high level of parallelism and more jobs which would have been solvable locally get sent to grid, the run time will slightly increase, as is indicated by the speedup of average.

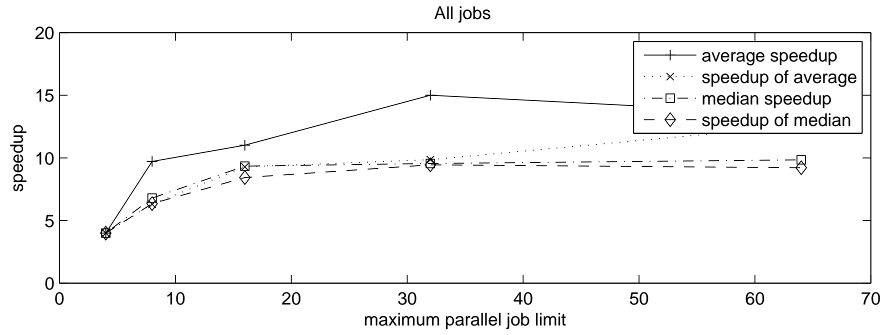

Figure 5.2: Median and average speedups, and speedups of median and average for all formulas

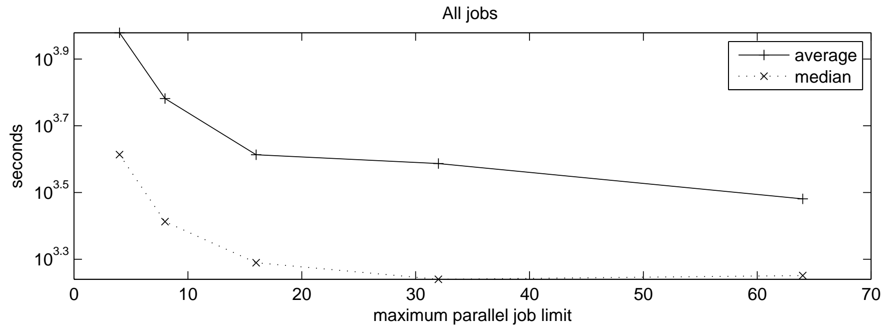

Figure 5.3: Median and average run times for all formulas

The median and average run time of all jobs with respect to the maximum parallel job limit is given in Figure 5.3. The median run time is just slightly more than one hour with maximum parallel job limit of 4, but the average run time is approximately twice the median, more than 2 hours.

5.3.1 Unsatisfiable and Satisfiable Random 3SAT

The random 3SAT instances, CNF formulas with exactly three literals in every clause, are generated with the formula generator *makewff* by Bart Selman [47]. The amount of variables in the instances ranges from 350 to 400 and the clause to variable ratio is set to $4.258 + 58.26|V|^{-5/3}$, where $|V|$ is the number of variables. This is the ratio that produces particularly difficult 3SAT formulas for DPLL type algorithms [20]. The formula size is measured in the number of variables. For each formula size, 10 formulas were generated. Approximately half of them were satisfiable and half were unsatisfiable. The results are shown in Figures 5.4–5.11, where Figures 5.4 and 5.8 compare the two heuristics and Figures 5.5–5.7 and 5.9–5.11 illustrate the speedup of *unsatisfiable* and *satisfiable* instances, respectively.

In Figure 5.4, presenting results for the unsatisfiable formulas, the effect of the heuristic is almost an order of magnitude on the run time, favouring MINMAX heuristic. Since the timeout value of 12 hours stops the growth of the run times for random heuristic, we expect the actual difference in problems having 370 or more variables to be larger than what is suggested by the graph.

Figure 5.5 shows the minimum, average, median and maximum run times for the unsatisfiable problems. The times range at maximum parallel job limit of 4 from over 30 minutes to the timeout value of 12 hours. The general tendency is that the run times decrease as more resources are employed from the grid. Only the maximum run time for formulas with 400 variables does not show a clear decrease. The maximum run time of maximum parallel job limit of 4 is set to the timeout value and as a result we do not get a real reference value for the scalability.

From Figure 5.6 we see that the formula solving scales well up to maximum parallel limit of 16, after which the speedup grows more slowly. The behaviour is explained by the observation that the actual parallelism in the grid does not increase as much as the maximum parallelism. The effect of increasing parallelism is more profound in the more difficult jobs. Especially the minimum speedup at problems with 350 variables is low compared to the speedups of formulas of greater size. This behaviour is expected, given that the longer run times of formulas will give more actual parallelism when the sending delays of the grid are taken into account. The minimum speedups are low both on problems with 350 and 400 variables. This results from the fact that easy formulas with low number of variables do not get the benefit from the grid, or, in fact, might suffer from the penalty of long communication delays. The difficult formulas, on the other hand, do not get a good reference point due to the enforced timeout value of 12 hours. The difficult formulas do not suffer from the long communication delays, as can be seen from the constantly increasing speedup as more resources are employed.

Figure 5.7 is constructed from Figure 5.5 by calculating the speedups of minimum, average, median and maximum. We see that the speedup of maximum for formulas of 400 variables is quite low. This is consistent with the previous observation that the maximum run time is close to the timeout value, and we cannot get a reliable speedup.

The satisfiable random 3SAT problems are characterized by a much more varying run times, which are usually lower than those of same sized unsatisfiable problems. Figure 5.8 shows the difference between the two heuristics. We can see that the random heuristic performs quite well on shorter run times, but loses to the MINMAX heuristic especially when the run times are higher. The overall impression is that the random heuristic does not perform as well as the MINMAX heuristic.

The run time scales better on larger formula sizes, whereas the additional parallel resources are mostly wasted on formulas of smaller size, yielding no speedup in Figure 5.9.

The variation in run times is evident from Figure 5.10. Solving of satisfiable 3SAT formulas can greatly profit from the simultaneous slightly different searches resulting from the scattering, and yields good, but somewhat volatile, speedups. Since the formulas with 400 variables are already quite hard, the solving with 4 parallel jobs is set to the timeout value. As a result, the speedup is not as good as for the formulas with 390 variables.

A slightly more consistent view of the speedups is given by Figure 5.11, where the speedups are collected from the more stable data of Figure 5.9.

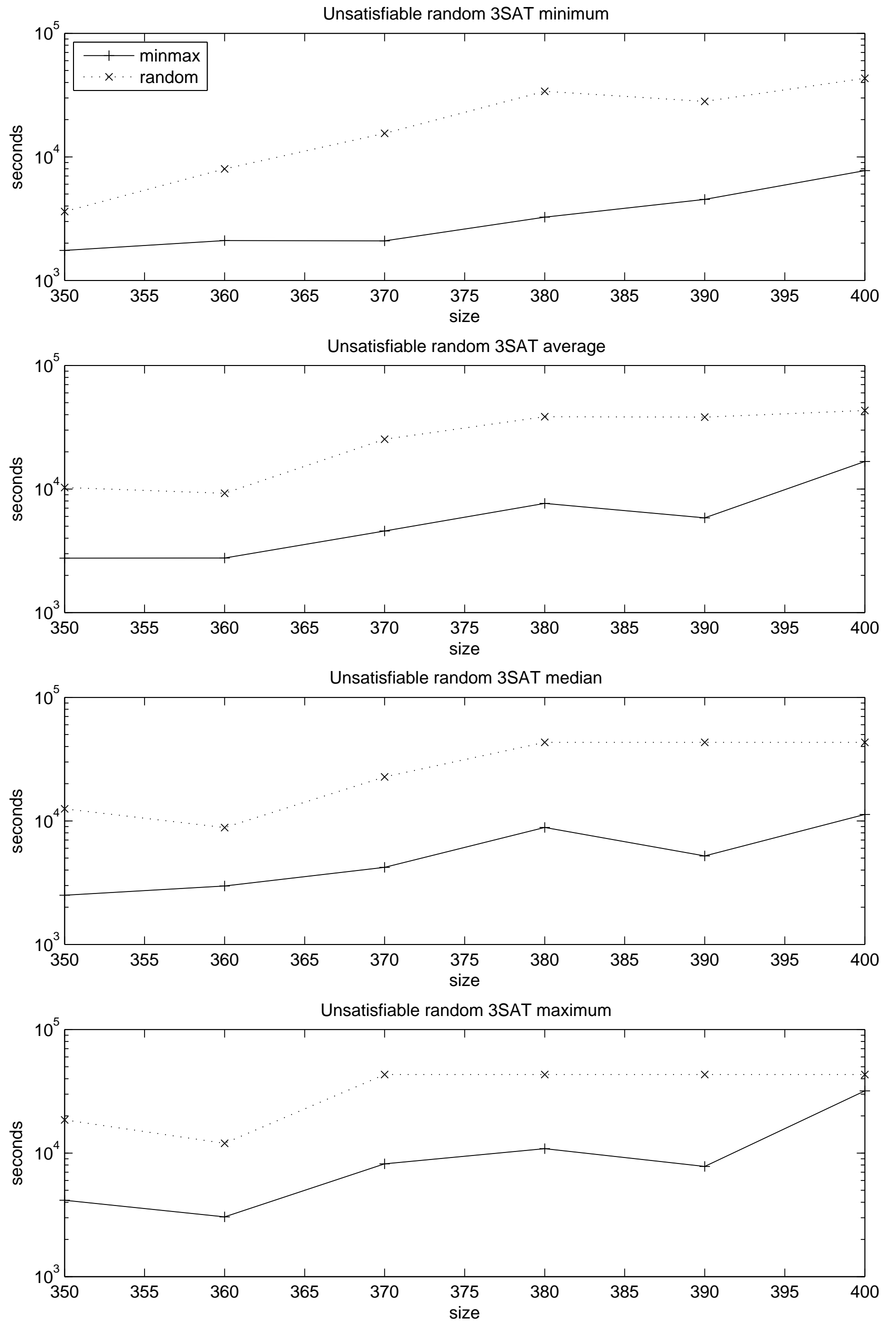

Figure 5.4: Unsatisfiable random 3SAT, random and minmax heuristic, duration in seconds

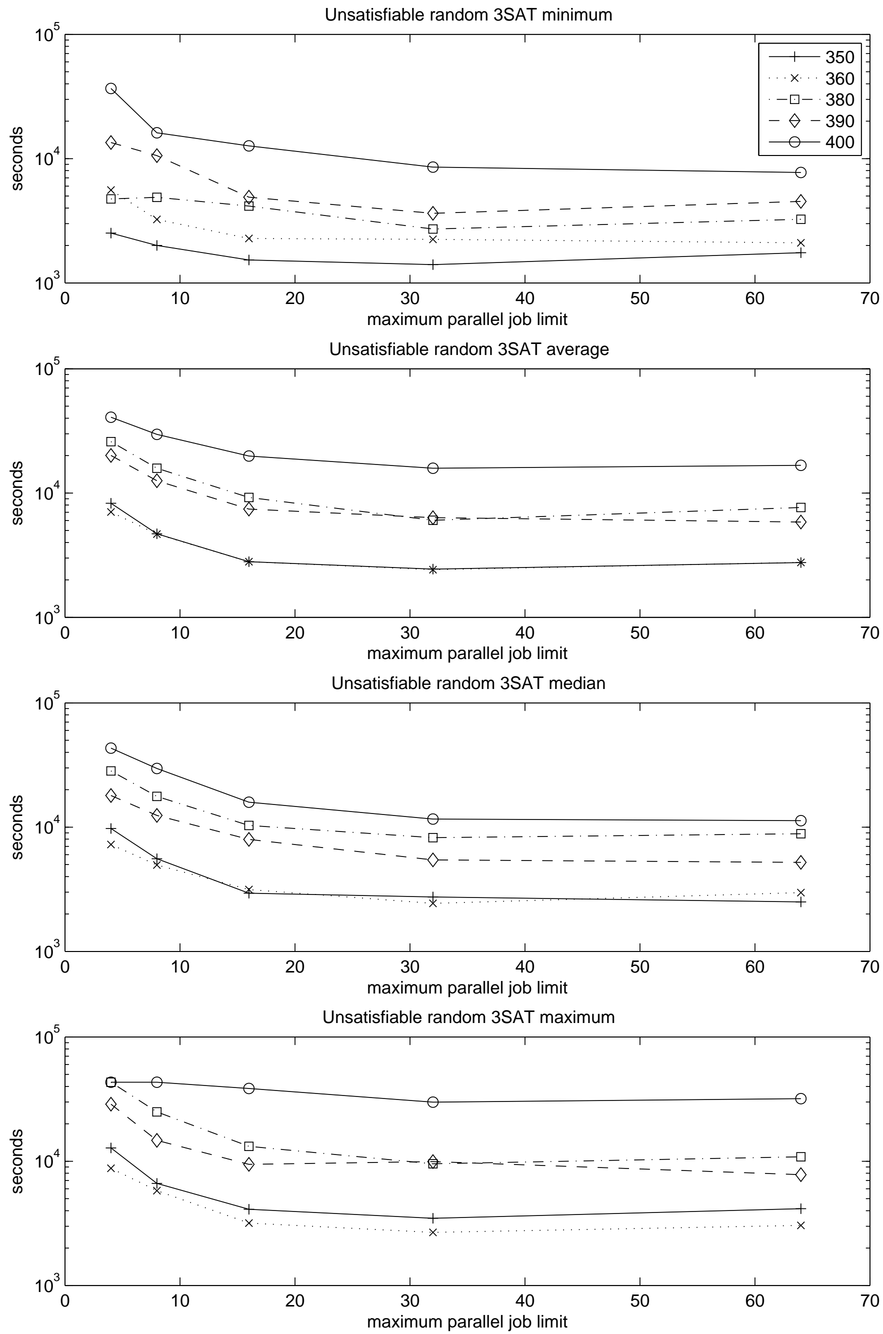

Figure 5.5: Scalability for some unsatisfiable random 3SAT formula sizes, duration in seconds

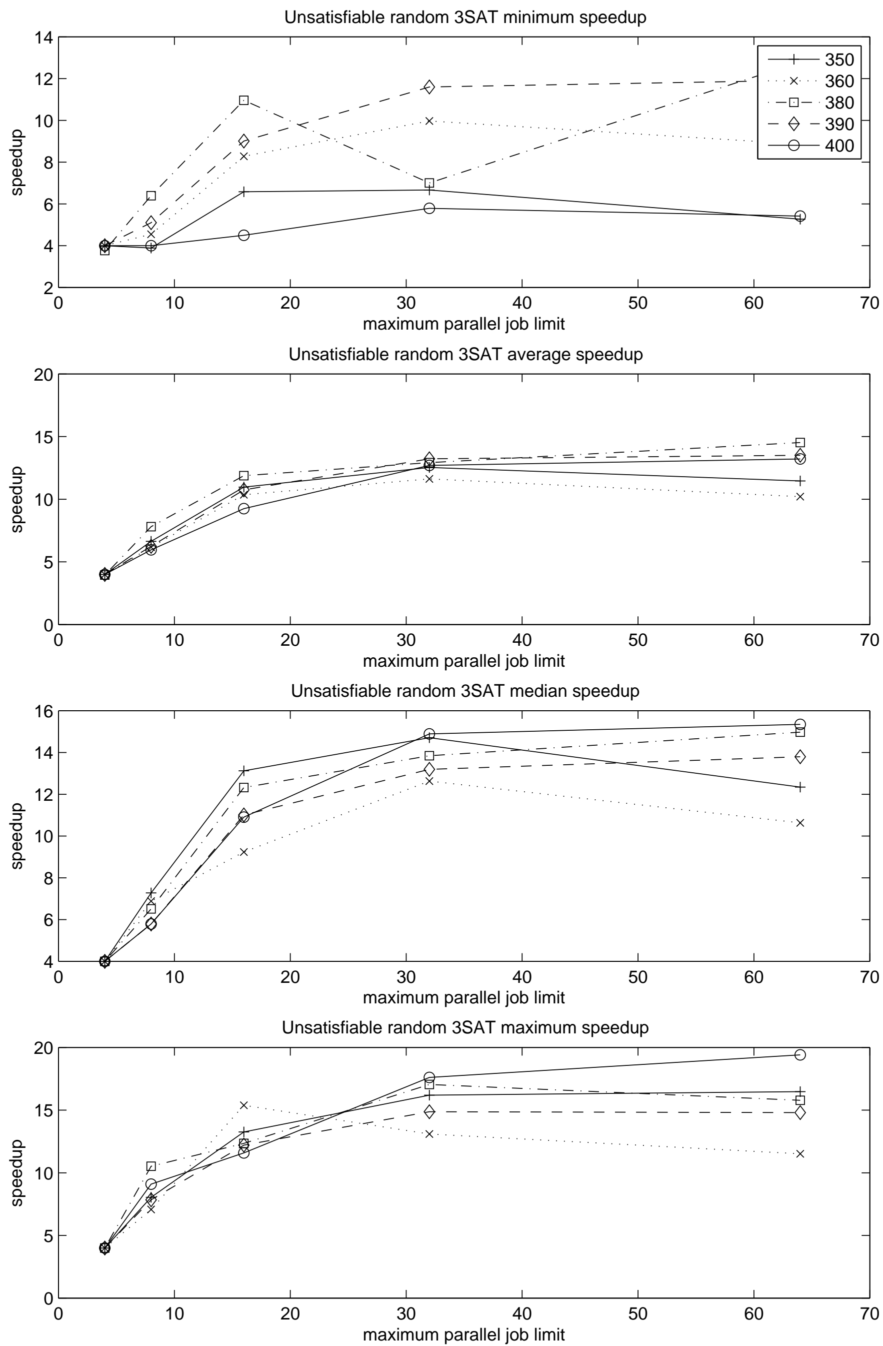

Figure 5.6: Speedup for some unsatisfiable random 3SAT formula sizes, speedup compared to smallest maximum grid jobs

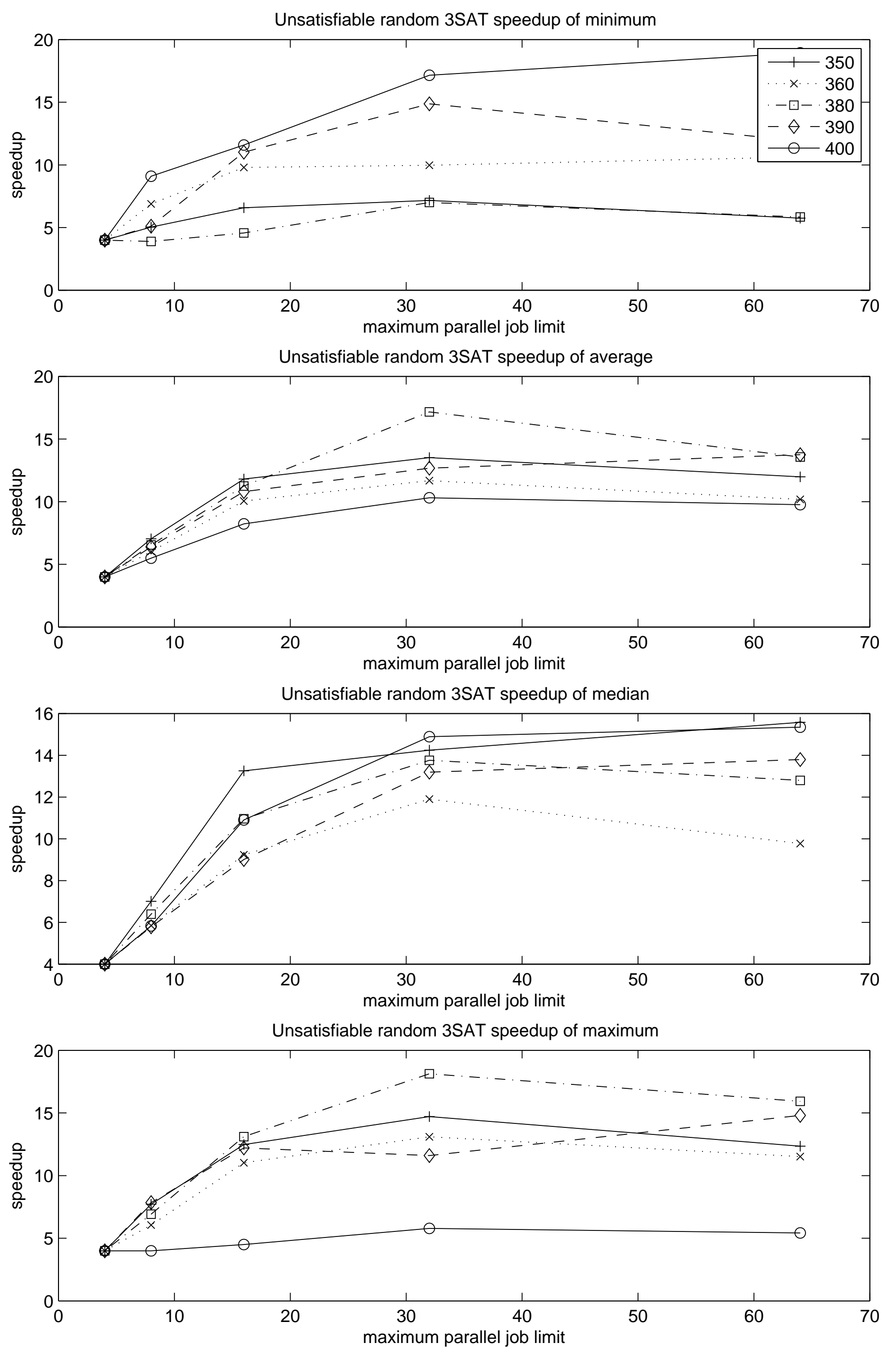

Figure 5.7: Speedup of minimum, average, median and maximum for some unsatisfiable random 3SAT formula sizes. Speedup compared to minimum, average, median or maximum at smallest maximum simultaneous grid jobs

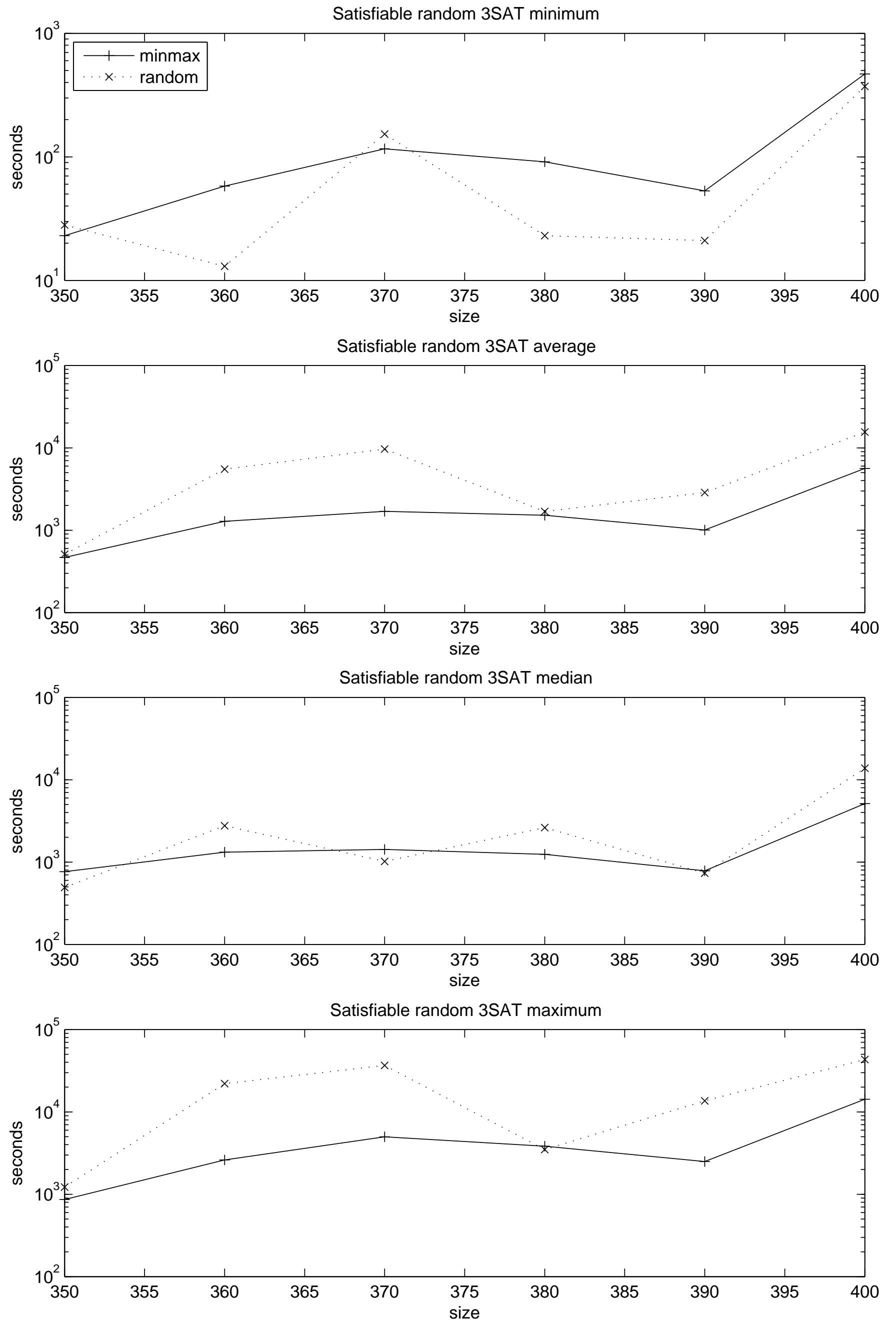

Figure 5.8: Satisfiable random 3SAT, random and minmax heuristic, duration in seconds

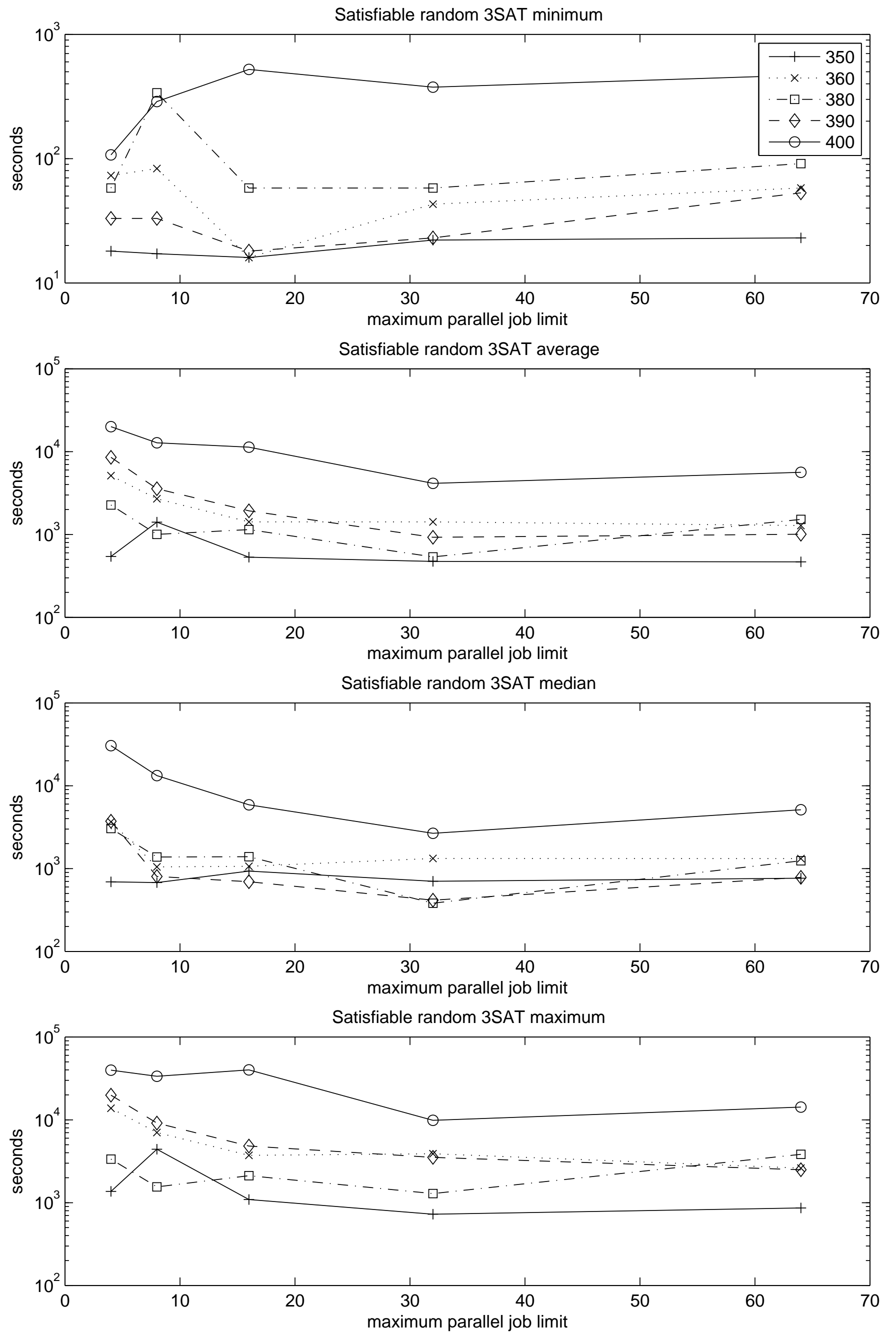

Figure 5.9: Scalability for some satisfiable random 3SAT formula sizes, duration in seconds

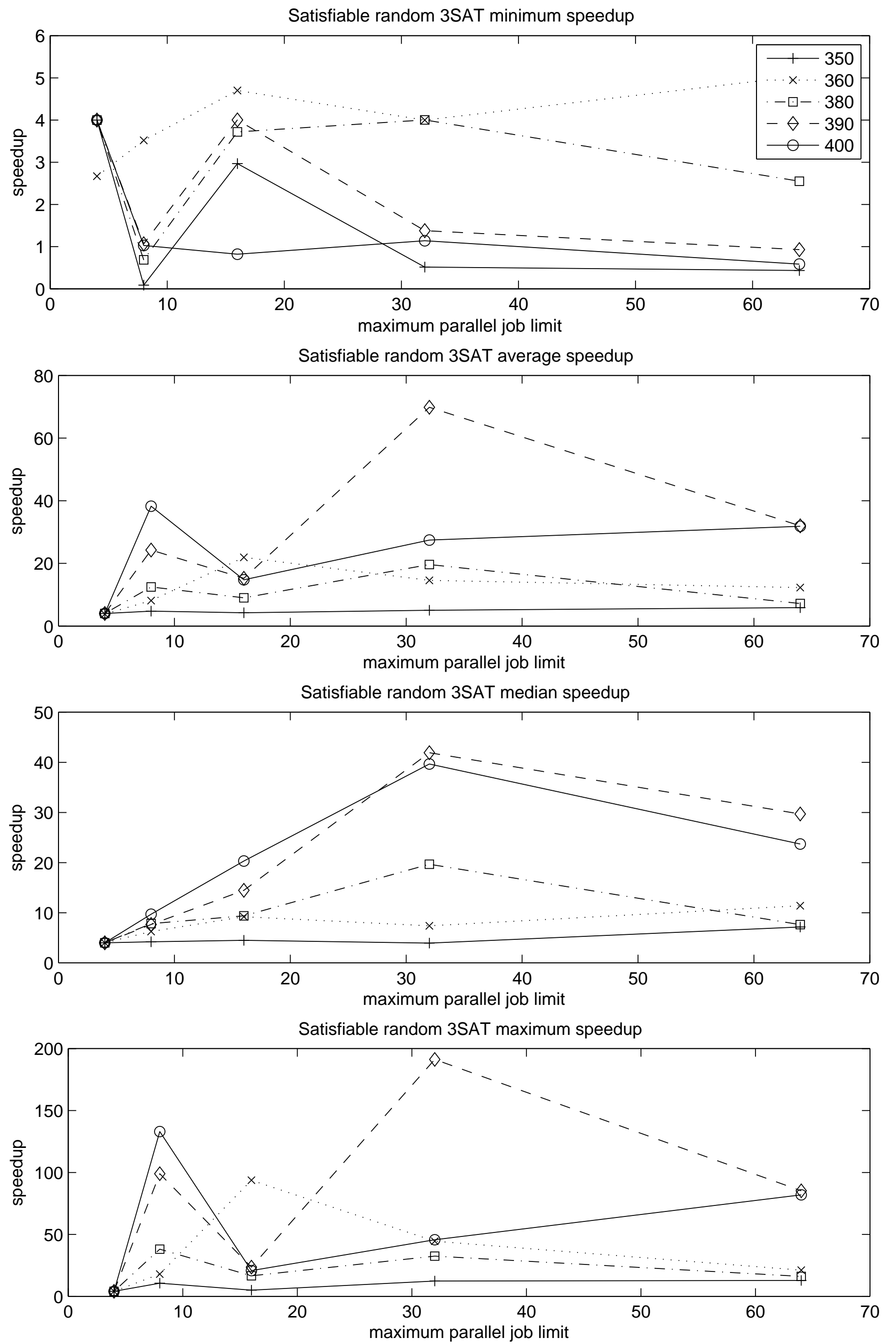

Figure 5.10: Speedup for some satisfiable random 3SAT formula sizes, speedup compared to smallest maximum grid jobs

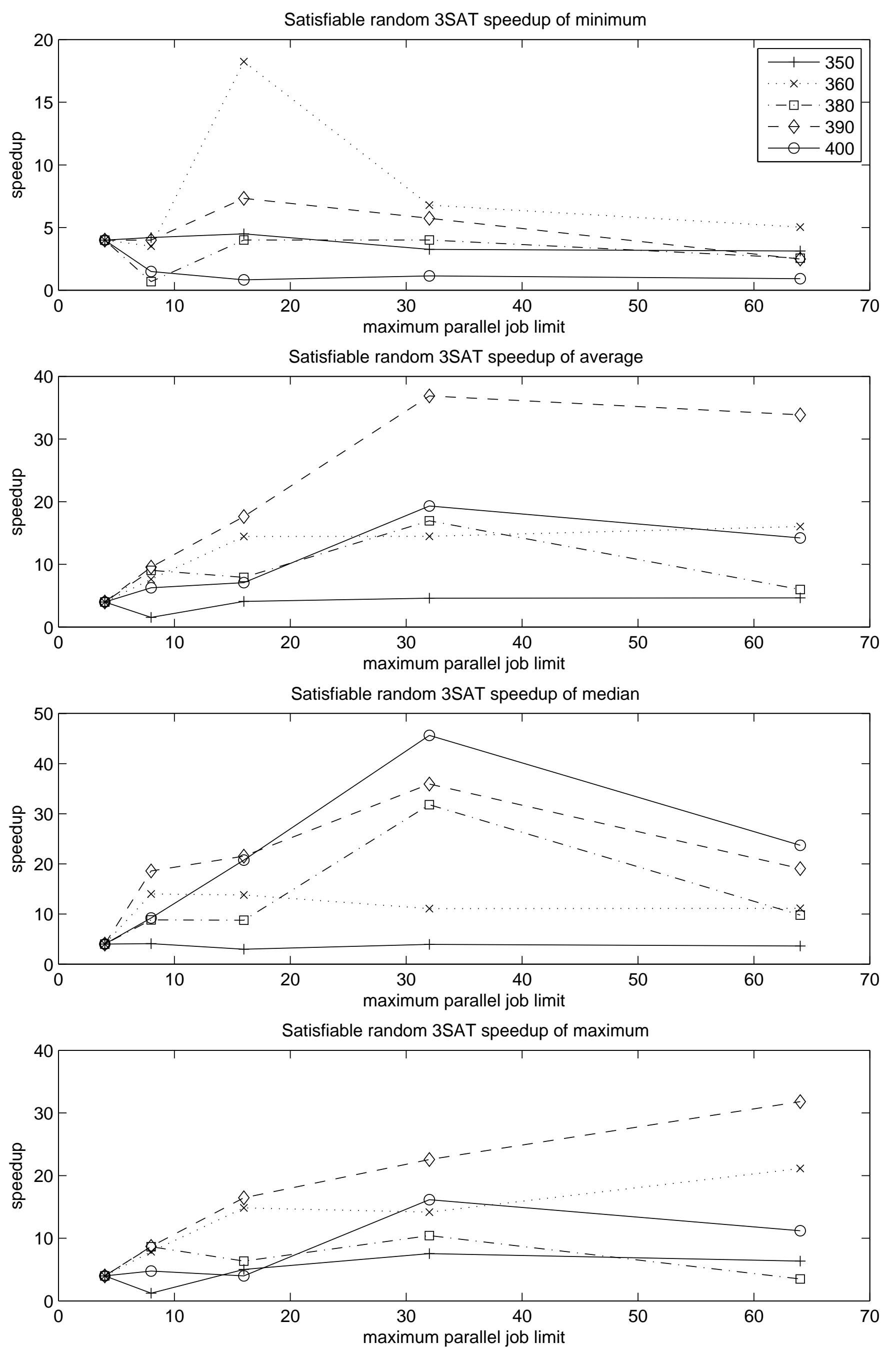

Figure 5.11: Speedup of minimum, average, median and maximum for some satisfiable random 3SAT formula sizes. Speedup compared to minimum, average, median or maximum at smallest maximum simultaneous grid jobs

5.3.2 Even Bit Composite Number Factorization

The satisfiable factorization formulas are generated by the computer program *genfacbm* by Tuomo Pyhälä [43]. An instance consists of a Braun multiplier circuit description in which the multiplicands are left free and the product is fixed to a product of two primes. The satisfying truth assignments to the formula are the valid inputs to the circuit when the output is the fixed product. The truth assignments corresponding to inputs 1 and the product itself are explicitly excluded, resulting in a formula with at most two satisfying truth assignments. This means that finding such a truth assignment amounts to factoring the binary number given as output of the circuit, i.e., finding the two primes whose product the output is. The benchmark simulates a structured and satisfiable circuit verification problem. The reported formula size is the width of the product in bits. For each size, total of nine formulas were created.

In Figure 5.12 the difference between the efficiencies of the two heuristics start to show clearly only at the larger formulas, and it seems to get more significant as the formula size increases. Whereas the run times with random heuristic seem to grow exponentially, the run times with MINMAX heuristic does not show clear signs of growth.

The run time of the formulas is surprisingly constant on the scalability plot of Figure 5.13. Only on the maximum run times can we observe somewhat consistent decrease when more grid jobs are employed.

The run times of the formulas of size 38 and 40 from Figure 5.12 suggest that the selected instances are approximately equally difficult. It is surprising that the speedups of the formulas in Figures 5.14 and 5.15 show such a different behaviour. The averages and maximums show that certain formulas benefit greatly from the added resources, but median and minimum indicate that for most formulas, there is no significant advantage of using maximum parallel job limit greater than four on certain formulas.

5.3.3 Prime Number Factorization

The benchmarks are generated by the same computer program *genfacbm* as the composite number factorizations, but the product is fixed to be a prime. As the only factors of any prime are the prime itself and 1, and these values are prohibited from the constructed circuit, there are no values that would make the propositional formula describing the circuit evaluate to true. We use a set of fixed prime numbers of widths between 35 and 38 in the benchmarks.

Difference between the MINMAX and the random heuristic is clear from Figure 5.16. On the selected formula sizes, the run times with MINMAX heuristic do not noticeably grow when compared to the run times with random heuristic.

As the unsatisfiable benchmarks are usually less prone to run time variations, the speedup can be observed clearly from the prime number factorization formulas. Even though the differences between the run times of the formulas of different sizes are quite small, the run times show a clear difference between the behaviour of the formulas when the maximum parallel job limit is increased, in Figure 5.17.

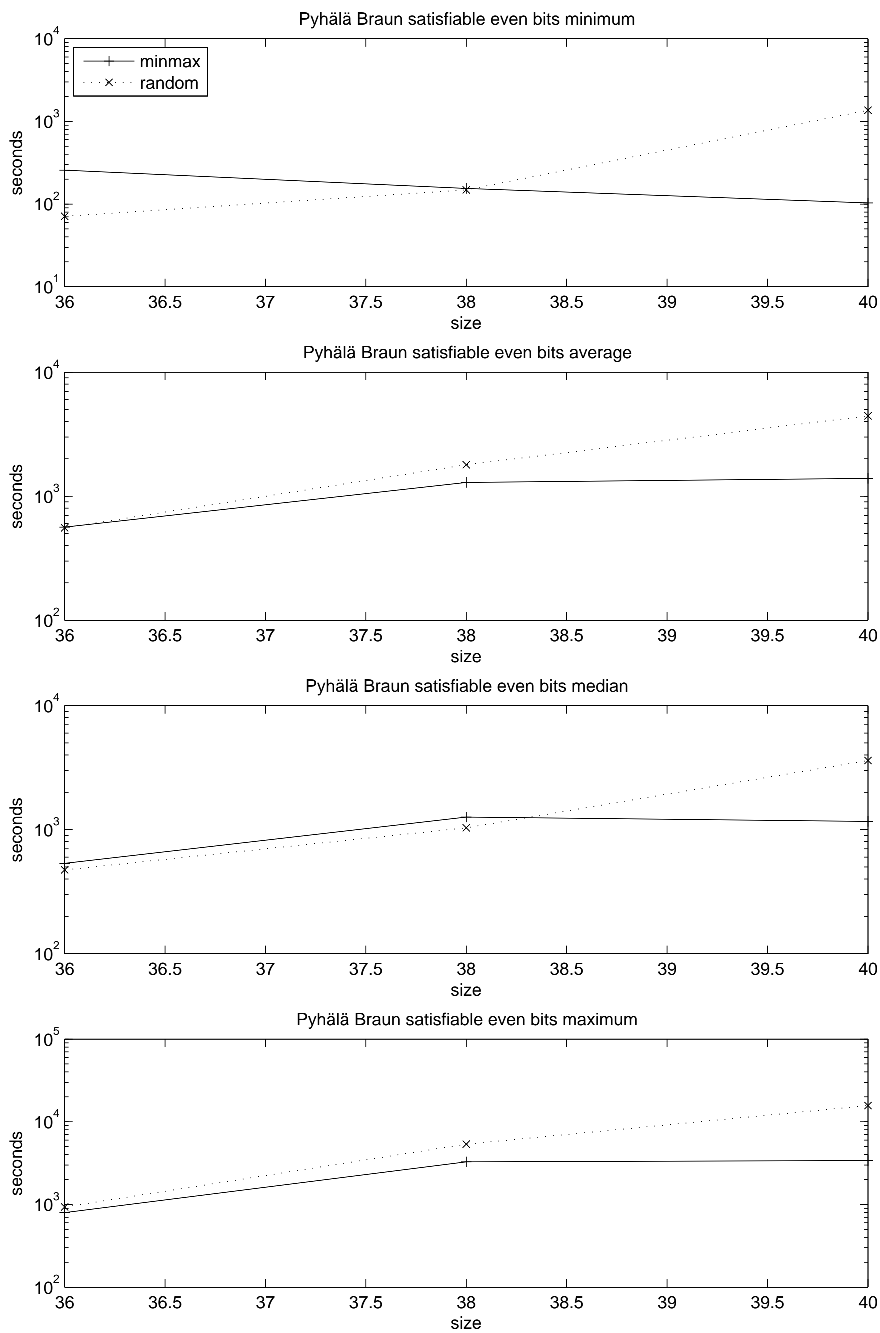

Figure 5.12: Even bit composite number factorization, duration in seconds

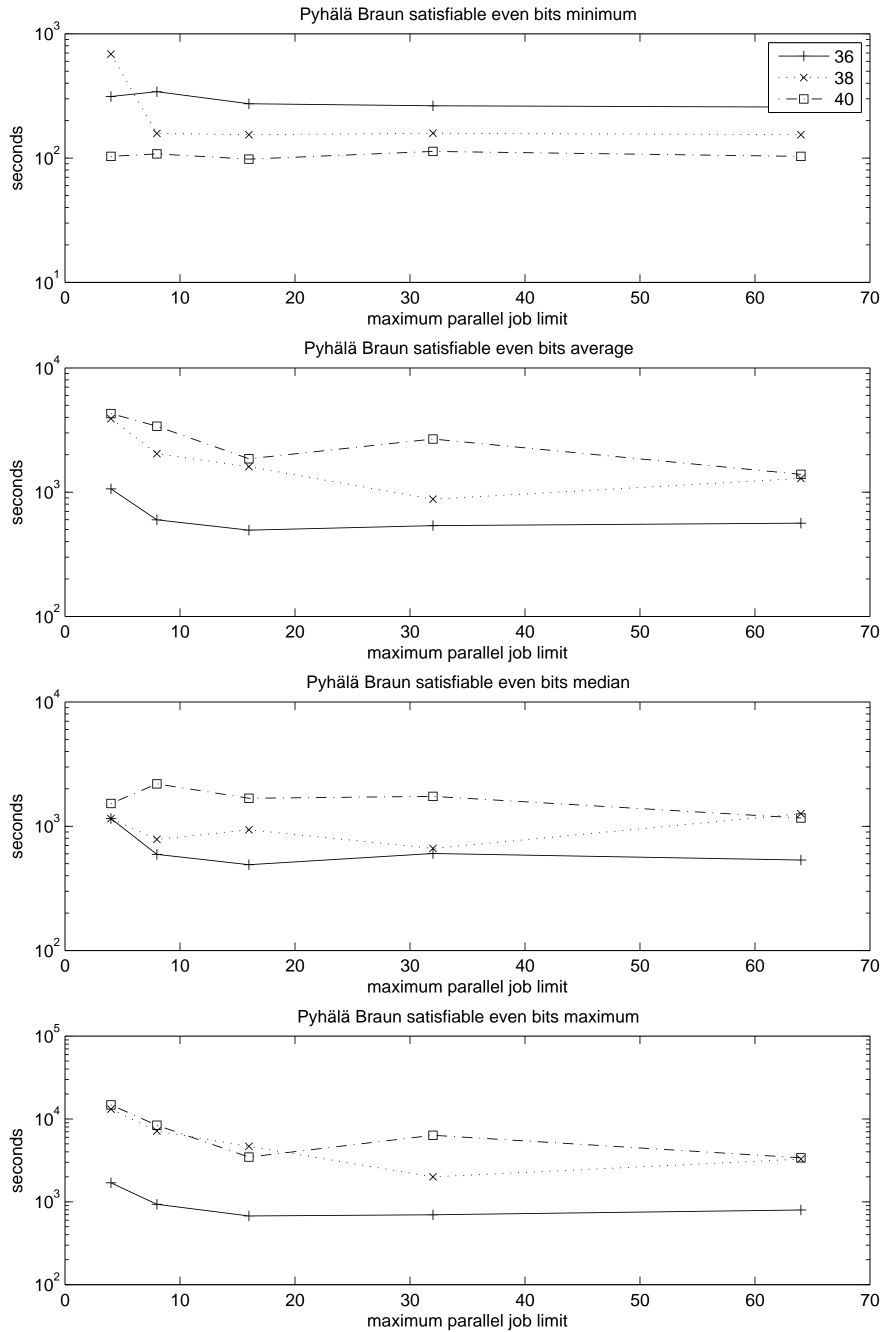

Figure 5.13: Scalability for some even bit composite factorization formula sizes, duration in seconds

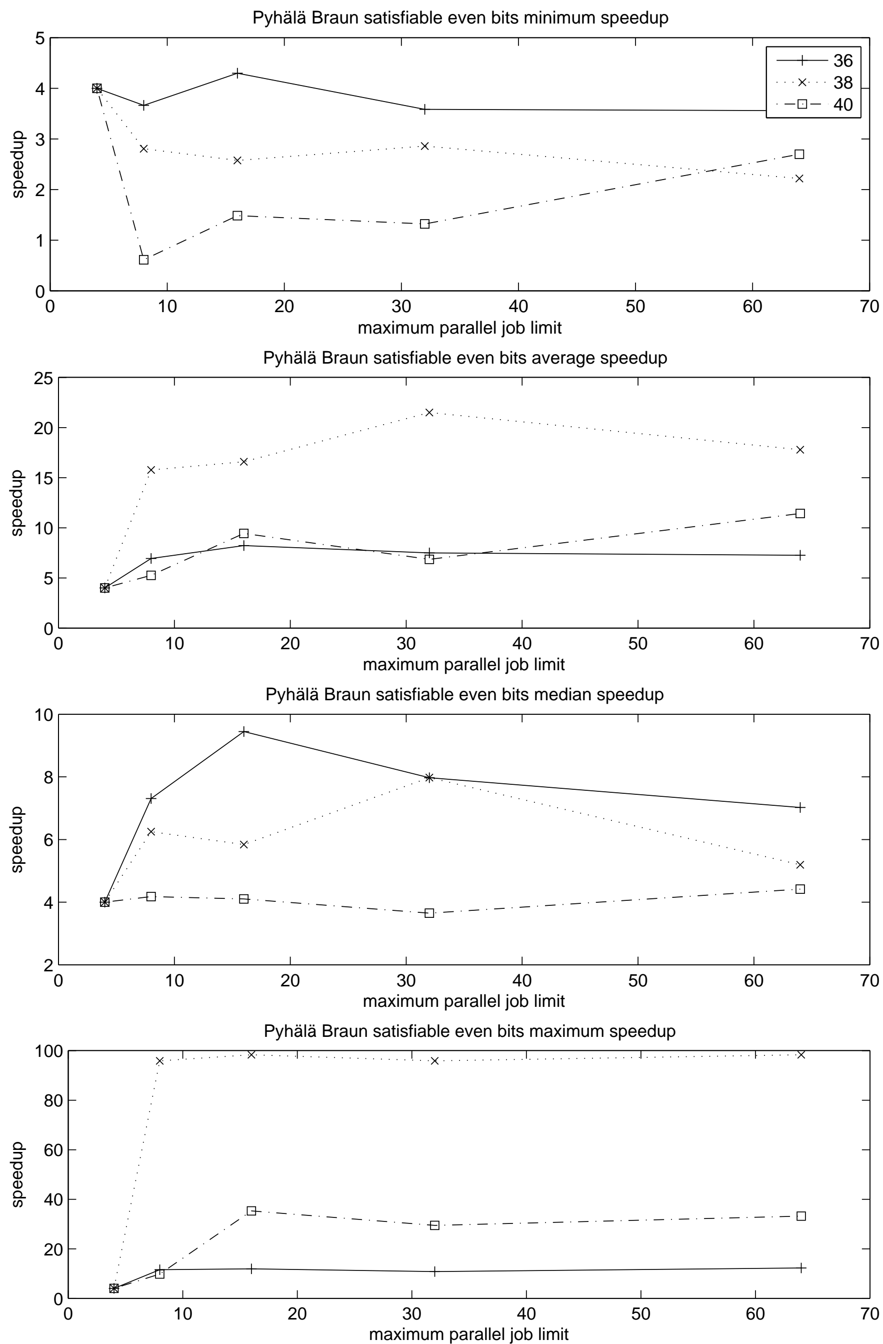

Figure 5.14: Speedup for some even bit composite factorization formula sizes, speedup compared to smallest maximum grid jobs

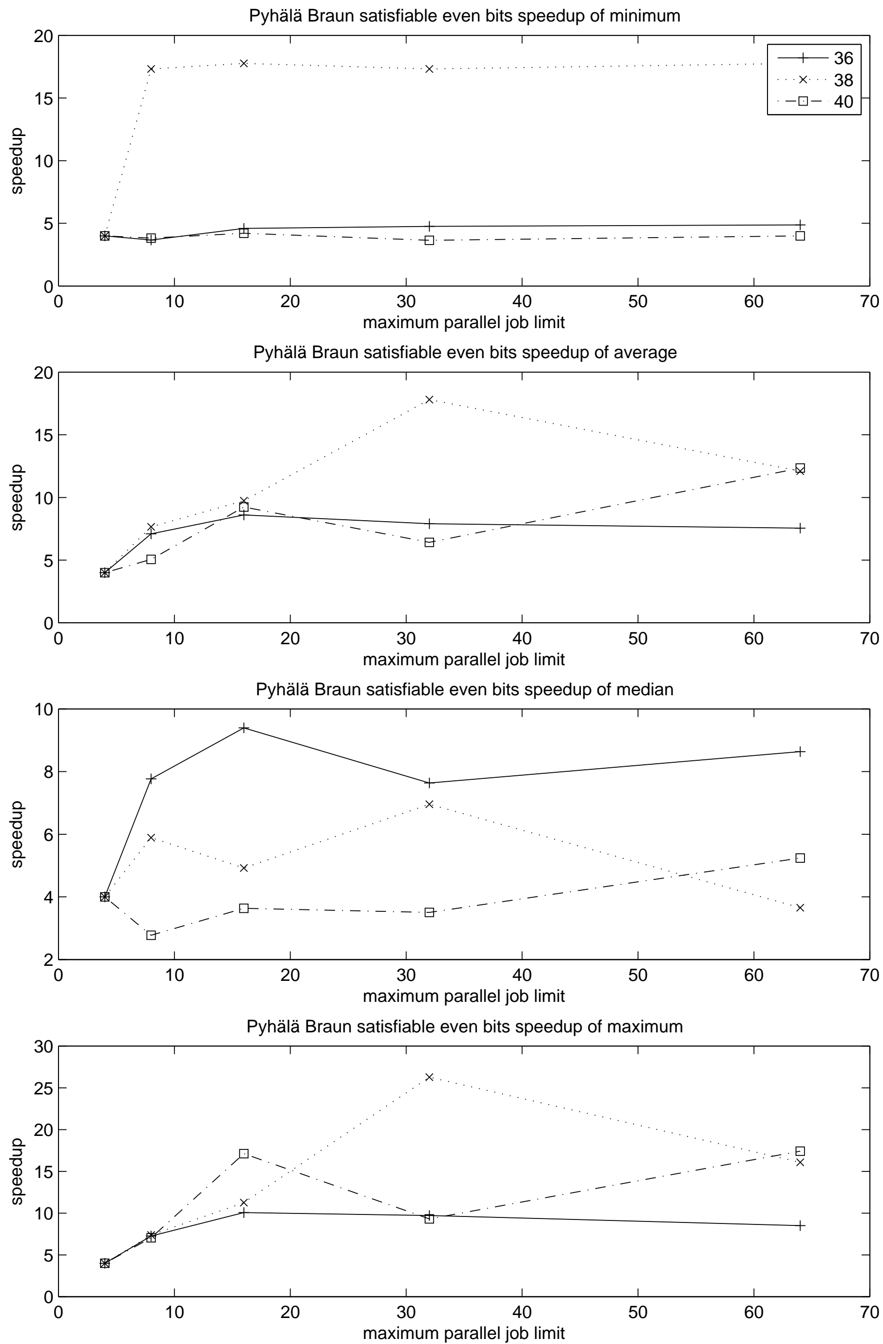

Figure 5.15: Speedup of minimum, average, median and maximum for some even bit composite factorization formula sizes. Speedup compared to minimum, average, median or maximum at smallest maximum simultaneous grid jobs

Table 5.2: Heuristic comparison for Four round one block DES

	minmax	random
min	493.10	423.07
avg	4720.68	17396.40
median	3418.45	10919.36
max	15264.62	43204.03

Since the differences in the problem sizes are rather small especially between the formulas of width 36 and 37, the speedups behave very similarly in Figure 5.18. A clear difference can though be seen when observing the behaviour of the formulas of width 35. On the easy problems, the resources cannot be fully utilized. This results in an almost constant speedup after certain limit. The more difficult formulas can benefit from the parallelism and the saturation point is reached somewhat later. Figure 5.19 is almost identical to Figure 5.18, which is due to the very stable behaviour of the run times in the unsatisfiable factorization formulas.

5.3.4 Four-round One-block DES

We test the scattering with circuits constructed from a known-plain-text attack to a four round DES. The benchmarks are constructed with the computer program *des2bc* by Tommi Junttila [33]. Total of 20 formulas are generated, all of which are *satisfiable*.

The role of the heuristic in solving is again very clear in the DES benchmarks. The solving times for the two heuristics is presented in Table 5.2. As is usual in *satisfiable* formulas, the random heuristic might occasionally find the solution approximately as fast as the MINMAX heuristic. It is noteworthy that whereas all formulas are solved using MINMAX heuristic and maximum parallel job limit of 64, for 6 of them computation reached the timeout of 12 hours when using the random heuristic.

The scattered formulas constructed using the random heuristic are relatively difficult and GRIDJM is able to effectively utilize approximately 40 of the parallel resources available for solving. A comparison between typical scattering trees of random and MINMAX heuristics in Figure 5.20 shows that the run times of scattered formulas are significantly more evenly distributed in the tree formed by the MINMAX heuristic. Only the formulas sent to grid are shown in the figures.

The solving times for the formulas have significant variation from run to run, depending on what kind of formulas are scattered using the MINMAX heuristic. For only 6 formulas, none of the solving attempts were successful in solving the formula in less than one hour. Figure 5.21 shows the scalability of the solving. The minimum run times seem to suffer from the communication delays of the grid. The average run time shows a steady decrease when the maximum parallel job limit is increased, even though the median run time seems to suffer slightly when the maximum parallel job limit is more than 16. The behaviour can be explained by the maximum run time; the solution to a difficult formula cannot be found until the timeout while the maximum

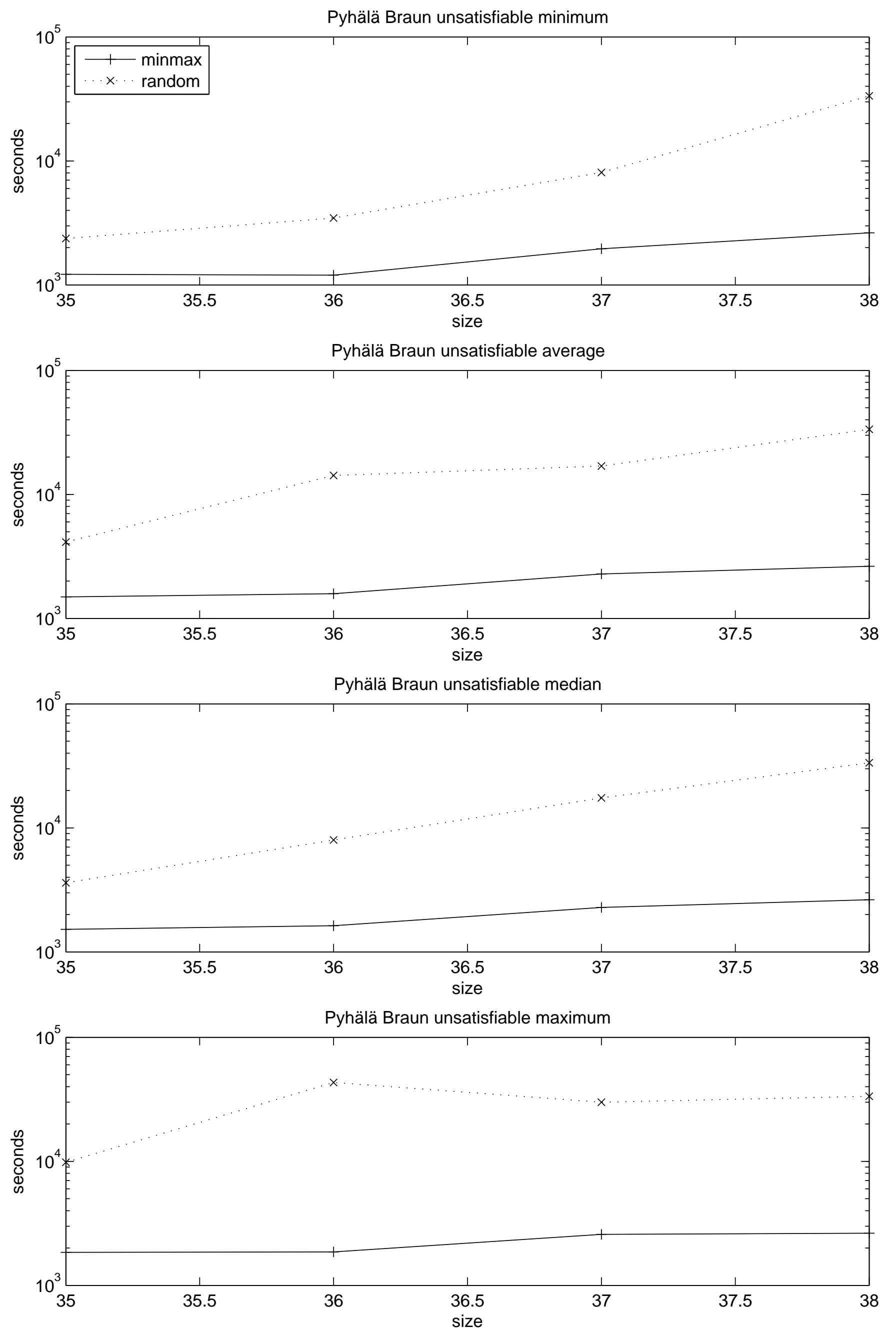

Figure 5.16: Prime number factorization, duration in seconds

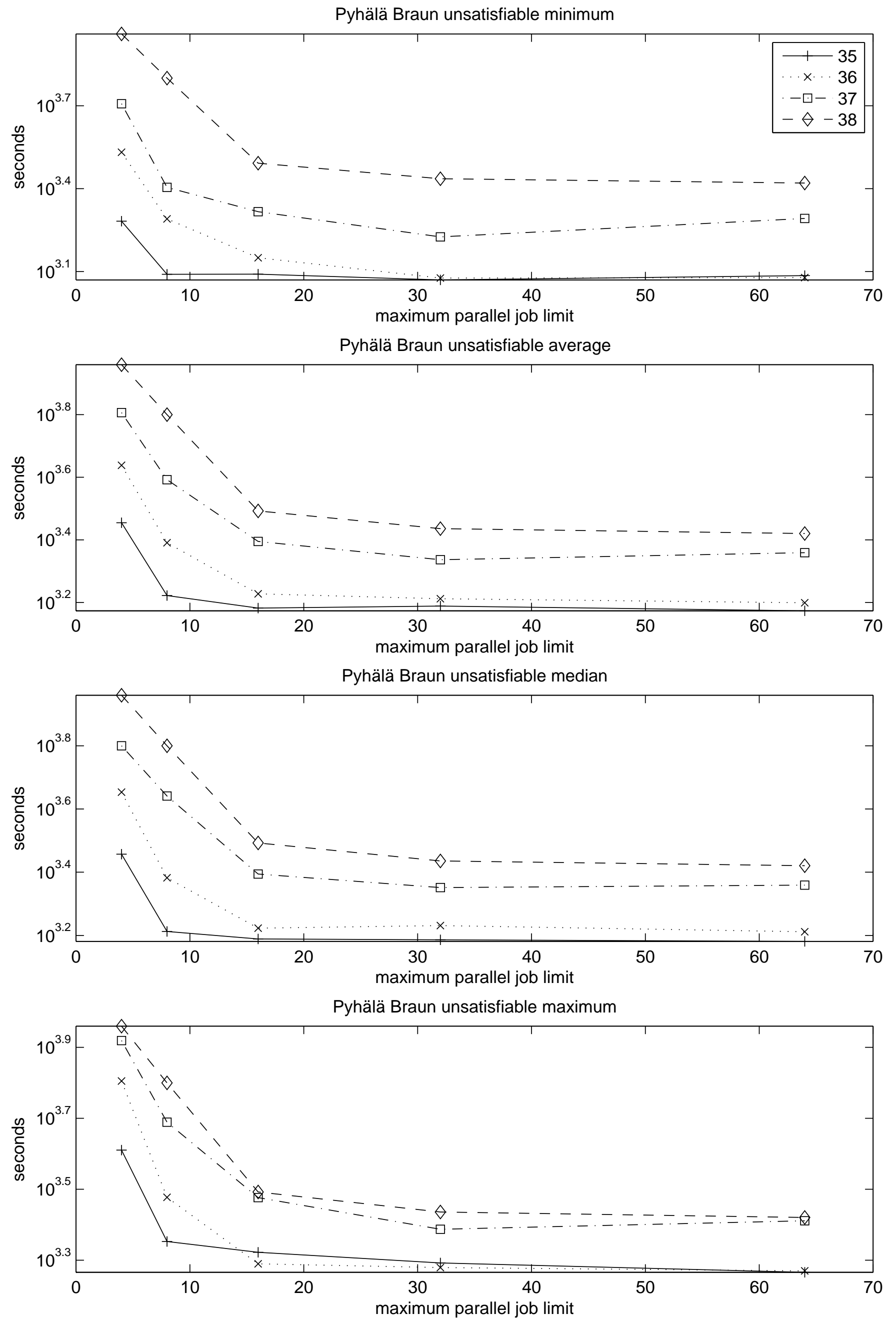

Figure 5.17: Scalability for some prime factorization formula sizes, duration in seconds

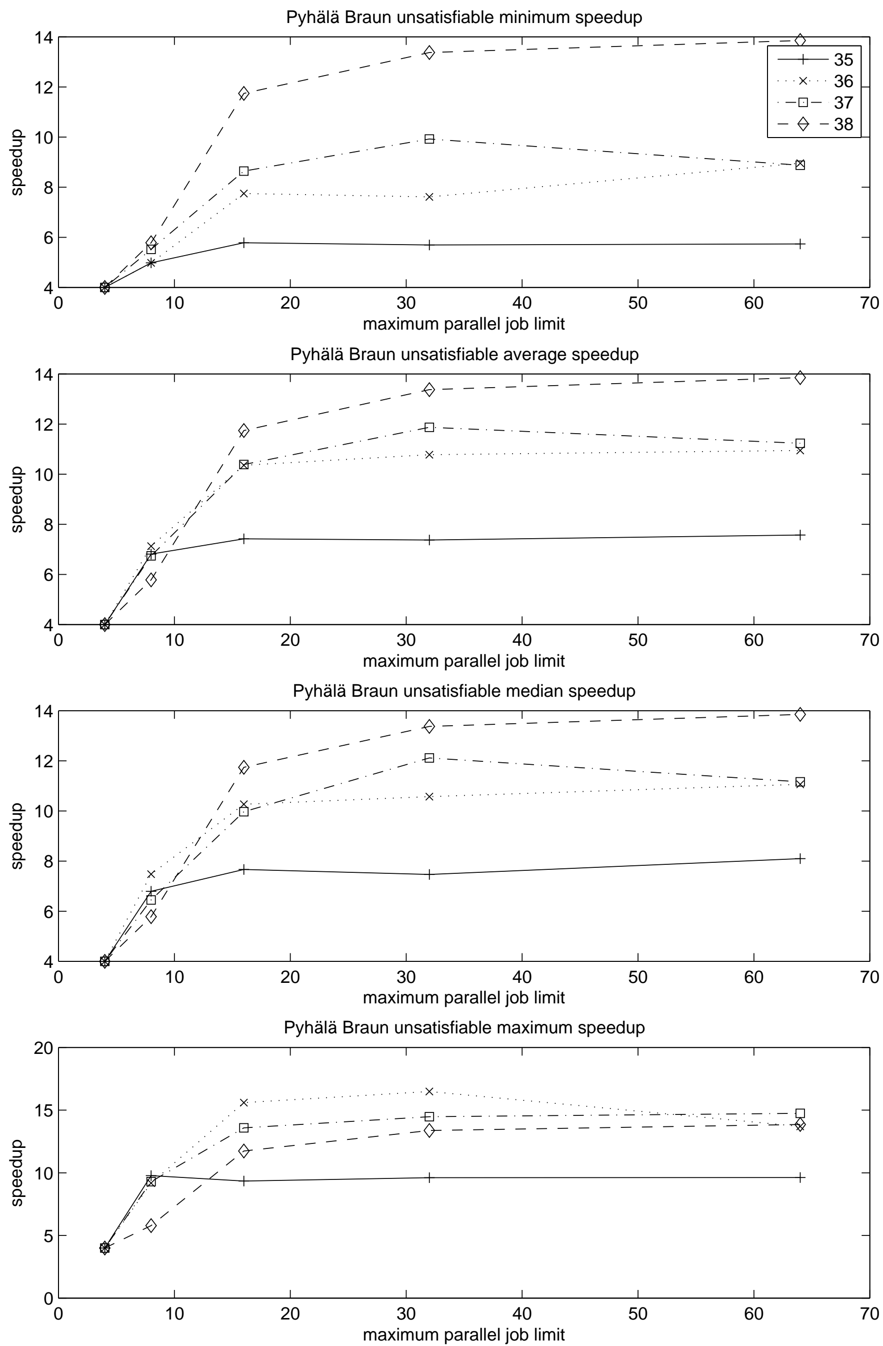

Figure 5.18: Speedup for some prime factorization formula sizes, speedup compared to smallest maximum grid jobs

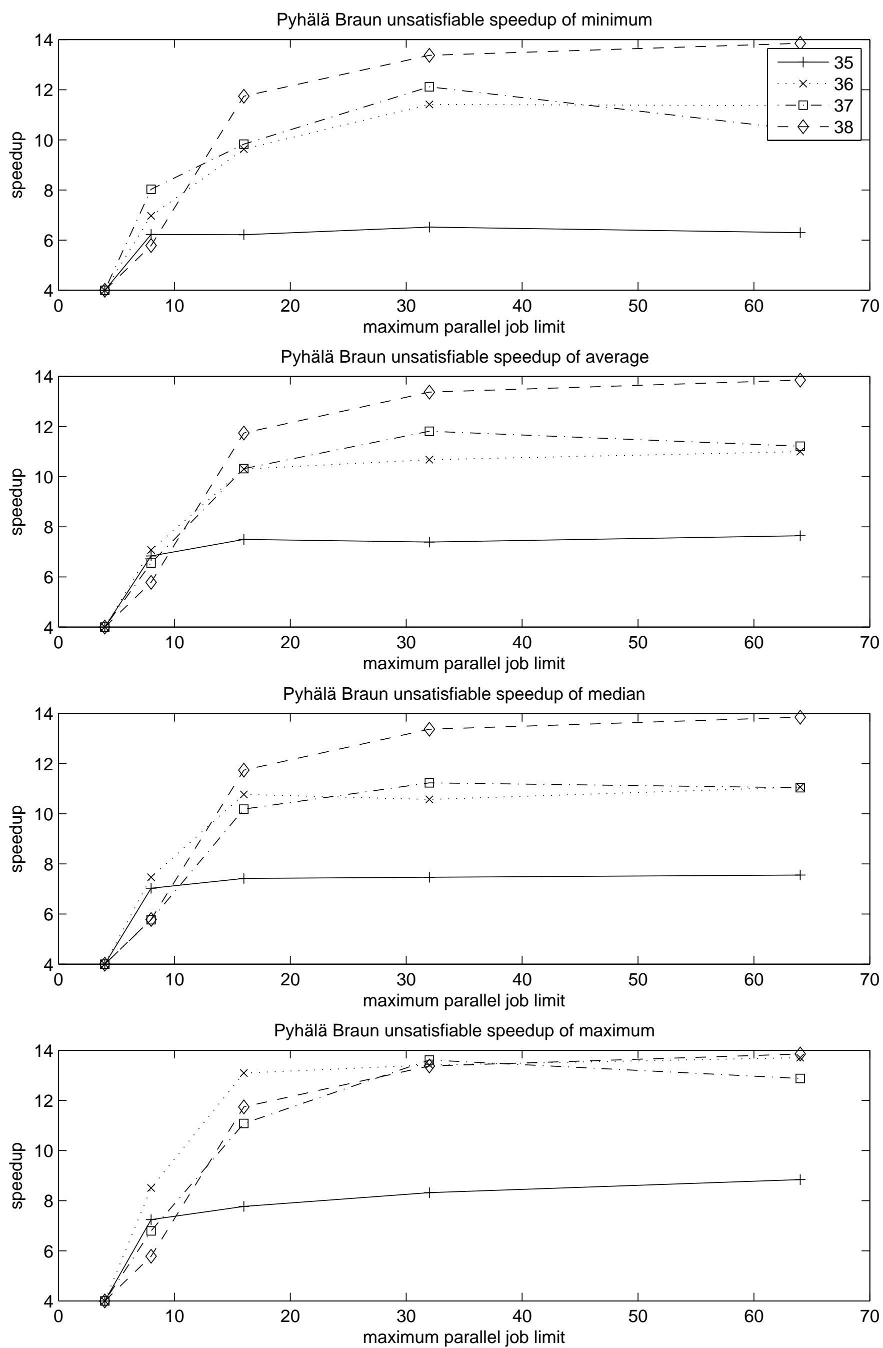

Figure 5.19: Speedup of minimum, average, median and maximum for some prime factorization formula sizes. Speedup compared to minimum, average, median or maximum at smallest maximum simultaneous grid jobs

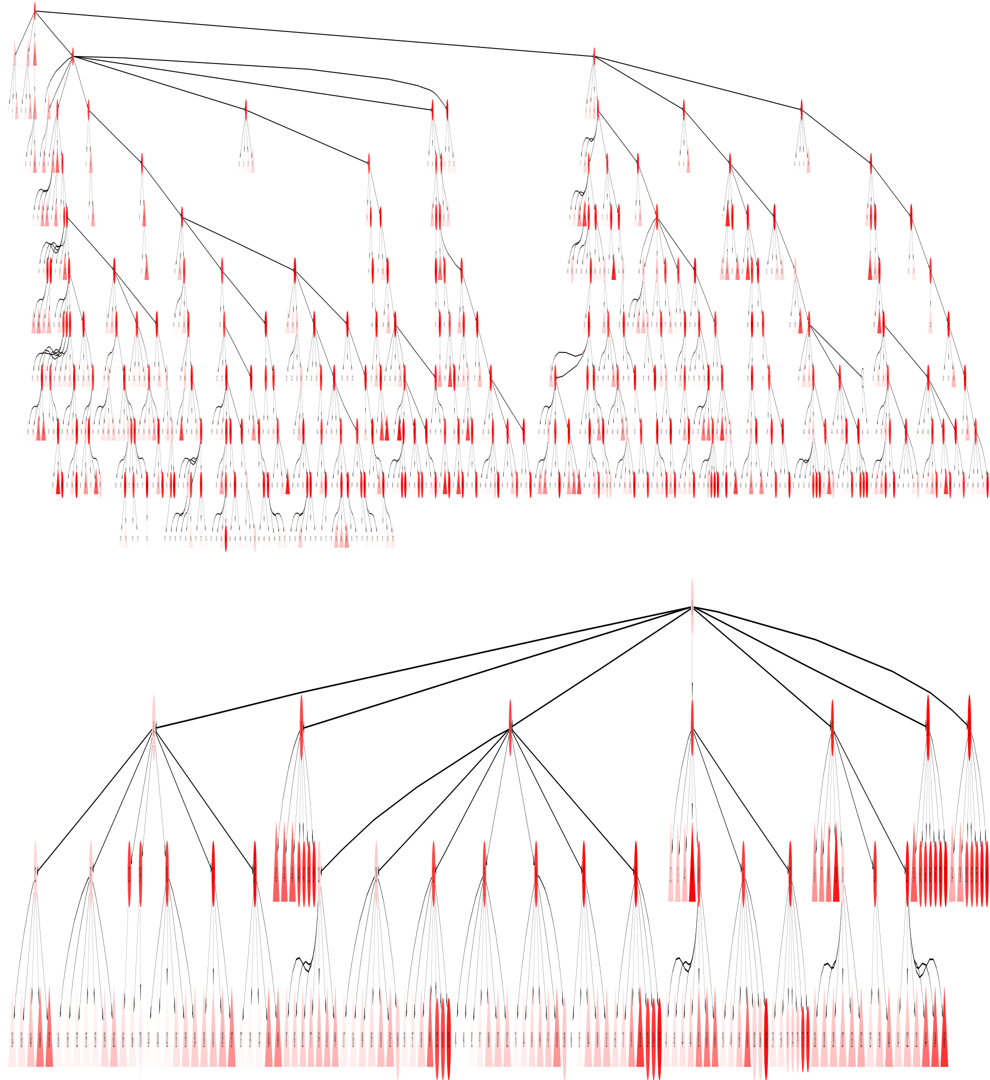

Figure 5.20: Scattering tree for random (top) and MINMAX (bottom) heuristics for a four-round one-block DES formula. Darker shades indicate longer run times

parallel job limit is 32. When the maximum parallel job limit is increased to 64, the solution is finally found in slightly more than 4 hours.

Figure 5.22 shows a superlinear speedup when the maximum parallel job limit is increased from 4 to 8 in the average and median speedups. The growth of speedup decreases dramatically when the maximum parallel job limit is increased from 8 to 16 and the speedup decreases especially in the median when the maximum parallel job limit is increased to 32. Finally, due to the timeouted solving on lower maximum parallel jobs, the speedup again increases when maximum parallel job limit is 64. Somewhat similar behaviour can be seen on Figure 5.23. The average is again dominated by the good speedup with difficult formulas, whereas the easy formulas suffer from the overheads.

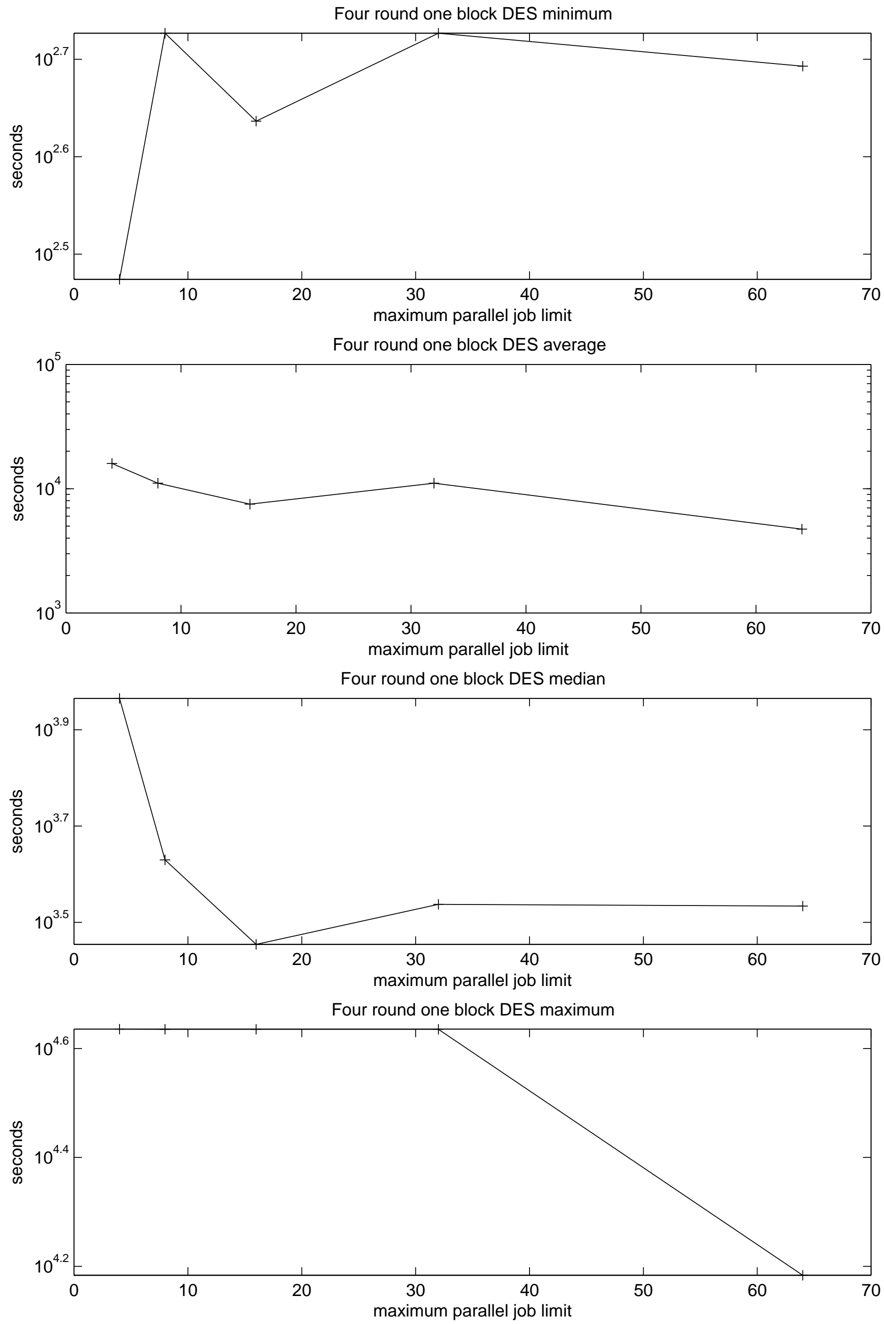

Figure 5.21: Scalability for four-round one-block DES, duration in seconds

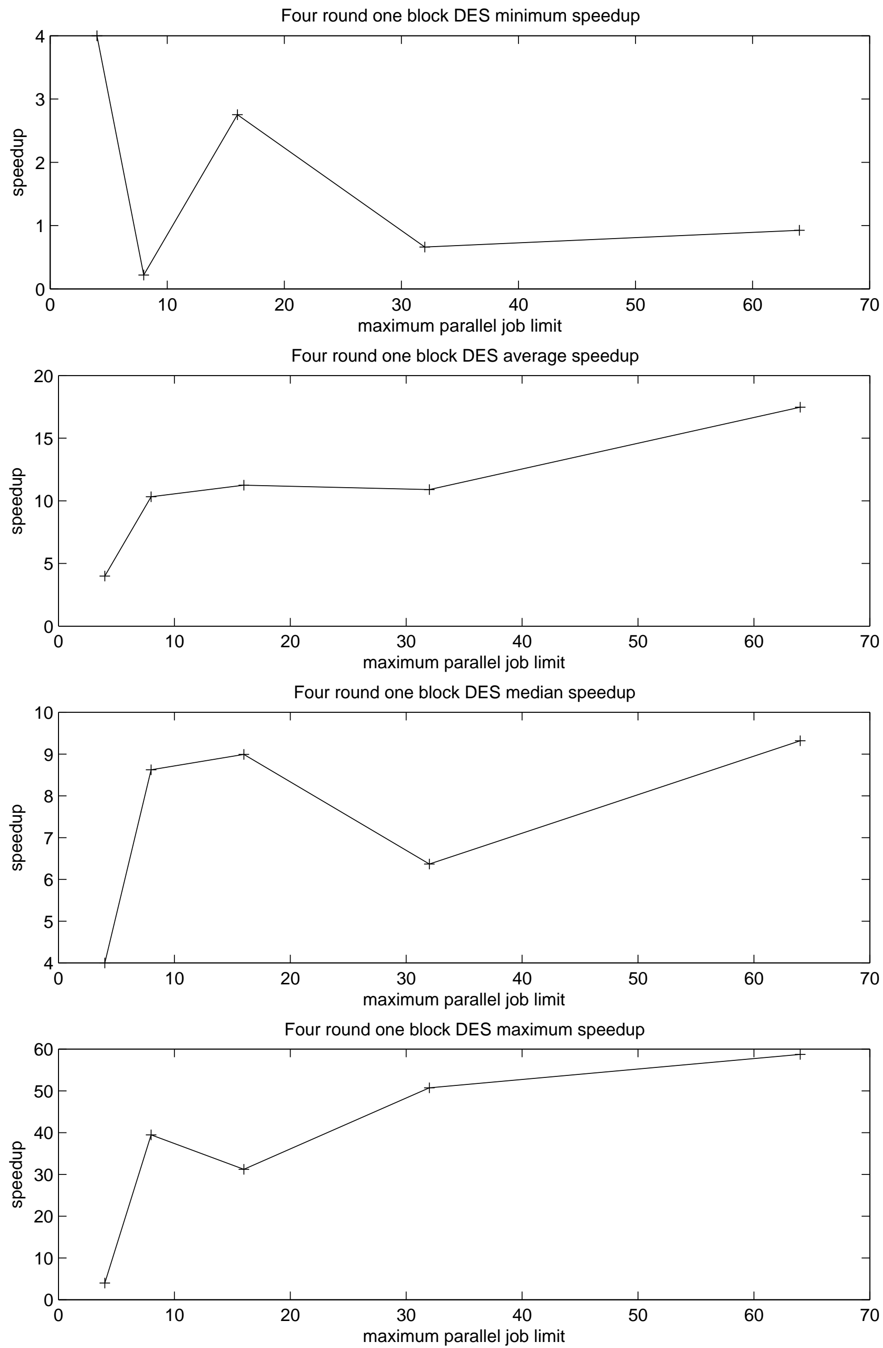

Figure 5.22: Speedup for four-round one-block DES, speedup compared to smallest maximum grid jobs

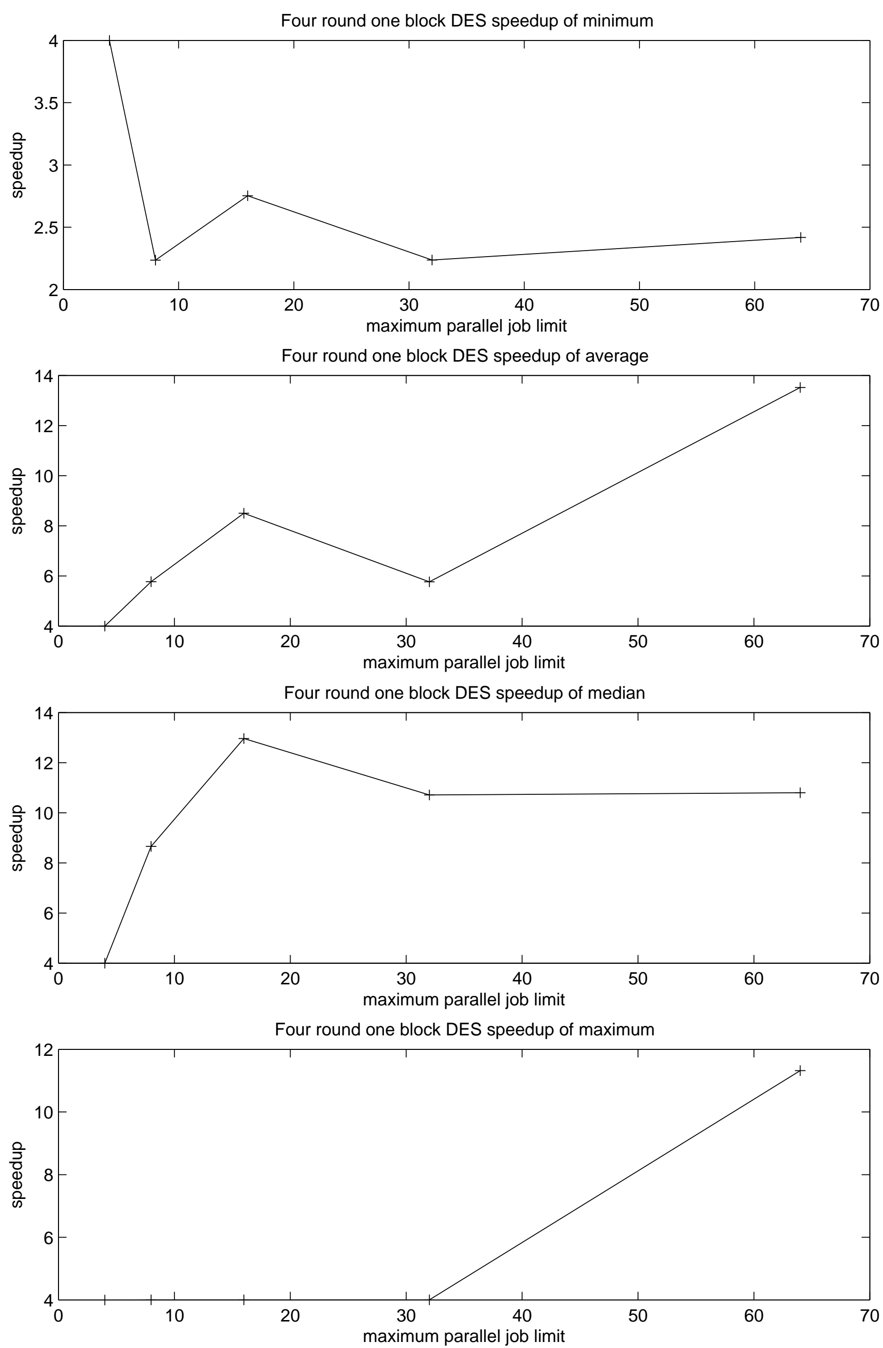

Figure 5.23: Speedup of minimum, average, median and maximum for four-round one-block DES. Speedup compared to minimum, average, median or maximum at smallest maximum simultaneous grid jobs

Table 5.3: Selected unsolved formulas from SAT2005 solver competition

Industrial		
Name	Time (s)	result
vmpc_32	108	*satisfiable*
vmpc_36	43200	timeout
Generated		
Name	Time (s)	result
eulcbip-7-UNSAT	43200	timeout
eulcbip-8-UNSAT	43200	timeout
eulcbip-9-UNSAT	43200	timeout
gensys-ukn007	19192	*unsatisfiable*
gensys-ukn008	13217	*unsatisfiable*
linvrinv6	43200	timeout
linvrinv7	43200	timeout
linvrinv8	43200	timeout
linvrinv9	43200	timeout
mod2c-rand3bip-sat-230-1	3208	*satisfiable*
mod2c-rand3bip-sat-230-3	1302	*satisfiable*
mod2c-rand3bip-sat-240-2	17900	*satisfiable*
mod2c-rand3bip-sat-240-3	43200	timeout
mod2c-rand3bip-sat-250-1	1692	*satisfiable*

5.3.5 The Unsolved problems from SAT2005

The satisfiability solver competition organized in 2005 [18] consists of a set of benchmarks which are solved using state-of-the-art satisfiability solvers. During the competition, certain formulas were not solved by any of the participating solvers. These formulas, which are not known to have been solved by any solver previously, provide a challenging benchmark for new solvers.

From the set of unsolved formulas, we select certain small formulas which during the competition were tried to solve with at least ten of the participating solvers but were solved by none. The results are given in Table 5.3. When solving these problems, the maximum parallel job limit is set to 32. Total of seven problems from the 16 for which the solving is tried are solved before the timeout of 12 hours.

6 RELATED WORK

In this work we present an algorithm for parallel SAT solving and use a computational grid as a platform for experimentations. Several other parallel SAT solvers exist where the benchmarks are either run in a grid or in a conventional multiprocessor environment. We will discuss both grid and multiprocessor solvers, as the distinction is not clear.

The idea of parallelizing the execution of declarative languages, which subsumes the parallelizing of SAT, has been extensively studied [44]. Much of the problems emerging from the parallelization of more expressive languages, such as Prolog [52], come from the preservation of the semantics of the underlying language. The results have mostly been moderate, the expensive co-ordination of variable binding being the key source of overhead, although promising results have lately been achieved [44]. SAT, however, has no such burden, due to its inherent nondeterminism. The underlying mechanism in both SAT solving and logic programming, the backtrack search, is similar. We believe that the interaction of the fields can be very profitable.

To the best of our knowledge, all parallel solvers use a DPLL type algorithm and all other parallel solvers except the GridSAT and SATU use the *guiding path*, introduced in [56], to construct the subproblems. Intuitively, the guiding path is the truth assignment indicating some unsolved branch in the search tree. The branch originally belonging to some solver is given to an idle solver and the initial owner of the branch does not backtrack to that branch. More formally, a *guiding path* is a finite ordered list of pairs $\langle(l_1, b_1), \ldots, (l_n, b_n)\rangle$, where l_i is a literal and b_i is either *true* or *false* for each $1 \leq i \leq n$. When the DPLL algorithm chooses a literal, say l_j, the pair (l_j, false) is added to the guiding path of the subproblem. If the algorithm backtracks to the point where l_j was chosen, the pair (l_j, false) is changed to the pair (l_j, true) in the guiding path. A subproblem can only be generated in the choice point l such that the corresponding entry in the guiding path is the first pair of the form (l, false). The solving of the new subproblem starts with a guiding path $\langle(l_1, \mathit{false}), \ldots, (l_{j-1}, \mathit{false}), (\bar{l}_j, \mathit{true})\rangle$, whereas the original guiding path becomes

$$\langle(l_1, \mathit{false}), \ldots, (l_{j-1}, \mathit{false}), (l_j, \mathit{true}), (l_{j+1}, b_{j+1}), \ldots, (l_n, b_n)\rangle.$$

Example 4 *Given a guiding path* $\langle(l_1, \mathit{true}), (l_2, \mathit{false}), (l_3, \mathit{true})\rangle$*, the single possible split yields a guiding path* $\langle(l_1, \mathit{true}), (\bar{l}_2, \mathit{true})\rangle$*, whereas the original guiding path becomes* $\langle(l_1, \mathit{true}), (l_2, \mathit{true}), (l_3, \mathit{true})\rangle$.

One of the earliest parallel solvers is a solver by Böhm and Speckenmeyer [12]. It is designed for a special parallel machine with up to 256 processors. The architecture is limited so that a processor can communicate with at most 4 other processors. The solver has a workload balancing algorithm that is designed to work efficiently with limited communication. The solver does not implement learning.

PSATO [56] is one of the first gridified SAT solvers. It is based on master-slave -model and is designed for a typical small network of workstations which are idle especially during out-of-office hours. The communication between

master and slaves is implemented with a parallel C-library. PSATO is based on the SAT solver SATO [55]. Each slave runs until it finds a satisfying assignment or discovers that the subproblem is unsatisfiable. If one of the slaves reports a solution to the master or predetermined timeout has occurred, the master sends a halt message to the slaves. Based on the results from slaves, the master manages a list of subproblems which are sent to idle slaves. PSATO has good fault tolerance based on a separate subproblem storage from which the master can recover previous results in case of a client or master crash. PSATO does not implement learning.

GridSAT [16, 17] is a Chaff-based SAT solver running in a computational grid. The architecture supports clause learning and sharing between computing nodes. GridSAT has a client-server architecture but implements direct communication between the clients. The solver splits only when the memory usage of a node exceeds a node-dependent threshold. The shared clause database is typically large, 100 MB on average. The new version of GridSAT implements a checkpoint-based fault recovery system for failures of grid nodes. The GridSAT parallelization model is based on exhausting processes. When a process runs out of resources, it divides the problem in two by copying the literals in the partial truth assignment on the decision level 0 and the negation of the decision literal of the decision level 1 of the exhausting process to the decision level 0 of the new process. The learned clauses are transfered to the new process and of these clauses, the new process stores the clauses not initially satisfied by the decision level 0 to its clause database.

PSatz [34] is a parallel solver implemented with a Remote Procedure Call library and Posix threads. It is based on Satz [37] and thus does not implement clause learning. The amount of computing nodes is fixed for every run. Whenever a node has no job, it *steals* one from a busy node. The computing nodes communicate always through the master node. The computing nodes send their state to the master every hour and every time a load balancing occurs between two nodes. If a computing node fails, the previous state of the failed node is sent by the master to another node, which continues the computation. At most one hour of computing time per node failure is lost.

PaSAT [49, 11] is a parallel SAT-solver designed to run in a grid environment. It is implemented in C++ and uses a platform-independent parallel programming API named DOTS [10]. The solver is based on SATO and it incorporates clause learning and sharing via queues. PaSAT is able to learn clauses and share them using a mobile agent approach. A mobile agent is sent from each computing process to collect short learned clauses that are not already satisfied by the partial truth assignment of the sender. The search forms a tree-like structure of subprocesses. Problem splitting occurs when a computing process runs out of jobs. The parent process in the subprocess arborescent makes a new split of its (sub)problem and shares it with the computing process demanding the new job. In order to avoid the *ping-pong* -effect, that is, the scenario where most of the solving time is spent on constructing easy formulas which do not contribute to the solving of the full problem noticeably, a certain time interval has to elapse before a new split can occur.

NAGSAT [28] is a proof-of-concept parallel satisfiability prover. It uses a parallel programming paradigm called *nagging* [46] in which a master pro-

cess does a normal sequential search. Clients request parts of the search tree and do a possibly redundant search on subproblems. If a client completes a subtree search (finds a solution or proof of unsatisfiability), the result is reported to the master which reports the solution or backtracks accordingly. NAGSAT implements only random branching heuristic in its sequential search and does not include clause learning. Nagging is inherently tolerant to client failures.

A low-level optimized Parallel Multithreaded Satisfiability Solver [27], implemented for Intel processors, shows increased solving time with respect to number of processors on a certain SAT-problem. The implementation runs on a single machine and uses clause learning techniques. Parallelization of the solver is implemented using guiding path. The test problem is solved in relatively short time (from 7 to 195 seconds) and the results show that the positive effect of a cache is not easily ported to parallel implementations.

To summarize the main differences between SATU and other solvers, we emphasize the separate heuristic for constructing the executions, modest requirements for communication, the pruning of the easy formulas before sending them to parallel execution environment, possibility of using third party SAT solvers and the idea of dividing the problem to possibly more than two parts when the executions are constructed.

7 CONCLUSIONS

The main contribution of this work is the introduction of a (to the best of our knowledge) new parallelization scheme for DPLL type solvers. The key advantages of the scheme are the clear separation of the distribution heuristic and the actual solving heuristic, the modest requirements from the underlying parallel architecture and the ability to easily run any propositional satisfiability solvers on the problems, including the industrial black box solvers.

The scheme divides a SAT problem instance in question, a CNF formula, into more and more constrained subproblems using a technique called scattering. The subformulas, called scattered formulas, form a scattering tree from which the formulas are then submitted to grid nodes for solving. The constraints inserted to the subproblems are represented as clauses, which are carefully selected by the scattering heuristic.

We have shown that the proposed parallelization scheme is effective and scalable. The scheme is used to solve SAT problems in NorduGrid, a production level computational grid. Using the implementation of the scheme, we are able to get linear speedup, assuming that run times needed to solve resulting subproblems (scattered formulas) are on average of certain length which depends on the communication delays of the grid.

The new scattering scheme requires significantly less from the underlying grid architecture than the schemes based on the *guiding path* approach (see Chapter 6), where the communication delays between processes play a more central role and a key issue in distributing is to communicate to the running processes the changes in their solution space.

The solving of problems for which DPLL type algorithms do not require much time on average does not suffer from greater overhead than what is imposed by the communication delays of the underlying computing environment, as the subproblem solving proceeds in the same way as the sequential solving. However, the addition of computing resources when solving a problem with short run time does not shorten the run time further.

7.1 FURTHER WORK

The effect of sophisticated clause learning in speeding up propositional satisfiability checkers is undeniable. In this work the learned clauses only come from the basic DPLL algorithm of the scattering. Especially as the implementation of the algorithm is not optimized, the effect of learning in solving time is probably small. The learned clauses from the jobs that have timed out would be especially useful, as they usually appear quite high on the scattering tree and affect many problems.

The development of different scattering heuristics and systematically comparing the performances of such heuristics is a new challenge. A better heuristic is presumably possible to construct when information from the upper layers of the solving process is available, say, from some high level language which is converted to a SAT problem.

The run times of SATU tend to slowly grow after a certain maximum par-

allel job limit has been reached. This might partly results from the shorter CPU time available for Scatter, the implementation of the scattering heuristics. In current implementation of SATU, Scatter does not save its state between the runs, and the heuristic function has to start from scratch with every scattering. In future, this becomes a significant threat to the system scalability as the communication delays are expected to become less significant when the grid technologies become more stable. In addition to optimizing Scatter, a completely different model with a heuristic function storing the heuristic values between the scatterings should be considered.

The test cases are missing a comparison for a commonly used scenario, where no scattering is used but the same formula is submitted to a number of grid nodes to be solved there using a randomized solver. The results from this trivial approach to the parallelization would serve as an interesting reference implementation to scattering.

ACKNOWLEDGEMENTS

This is a reprint of my master's thesis. The work is financially supported by the Academy of Finland under the project Advanced Constraint Programming Techniques (number 211025). Writing the work has been a teaching and rewarding process for me. I want to thank my supervisor Prof. Ilkka Niemelä and my instructor D.Sc. Tommi Junttila for the patient guidance in writing, and for their time, which they have so generously given me in numerous discussions concerning the work. My family for the support in every aspects of life, and my friends for just being my friends.

Bibliography

[1] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Solving difficult instances of boolean satisfiability in the presence of symmetry. *IEEE Transactions on Computer Aided Design*, 22(9):1117–1137, 2003.

[2] N. Amla, X. Du, A. Kuehlmann, R. Kurshan, and K. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In *CHARME*, volume 3725 of *Lecture Notes in Computer Science*, pages 254–268. Springer-Verlag, 2005.

[3] L. Baptista and J. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *Proceedings of the 6th International Conference on Principles and Practices in Constraint Programming (CP'00)*, volume 1894 of *Lecture Notes in Computer Science*, pages 489–494. Springer-Verlag, 2000.

[4] R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208. AAAI Press, 1997.

[5] P. Beame, H. Kautz, and A. Sabharwal. Understanding the power of clause learning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 1194–1201, San Francisco, CA, 2003. Morgan Kaufmann Publishers.

[6] D. Le Berre and L. Simon. The essentials of the SAT 2003 competition. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, volume 2919 of *Lecture Notes in Computer Science*, pages 452–467. Springer-Verlag, 2003.

[7] D. Le Berre and L. Simon. Fifty-five solvers in vancouver: The SAT 2004 competition. In H. H. Hoos and D. G. Mitchell, editors, *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT2004, Vancouver, BC, Canada, May 10–13, 2004, Revised Selected Papers Series*, volume 3542 of *Lecture Notes in Computer Science*, pages 321–344. Springer-Verlag, 2005.

[8] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[9] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *Computer Aided Verification: 11th International Conference, CAV'99*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–71. Springer-Verlag, 1999.

[10] W. Blochinger, W. Küchlin, C. Ludwig, and A. Weber. An object-oriented platform for distributed high-performance symbolic computation. *Mathematics and Computers in Simulation*, 49(3):161–178, 1999.

[11] W. Blochinger, C. Sinz, and W. Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Journal of Parallel Computing*, 29(7):969–994, 2003.

[12] M. Böhm and E. Speckenmeyer. A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17(3–4):381–400, 1996.

[13] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[14] D. Brooks, E. Erdem, J. Minett, and D. Ringe. Character-based cladistics and answer set programming. In *Practical Aspects of Declarative Languages: 7th International Symposium, PADL 2005*, volume 3350 of *Lecture Notes in Computer Science*, pages 37–51. Springer-Verlag, 2005.

[15] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[16] W. Chrabakh and R. Wolski. GridSAT: A chaff-based distributed SAT solver for the grid. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, page 37. IEEE Computer Society, 2003.

[17] W. Chrabakh and R. Wolski. Solving “hard” satisfiability problems using GridSAT, June 2004. Global Grid Forum 2004 Workshop on Grid Applications: from Early Adopters to mainstream users. GGF Informational Document.

[18] Satisfiability Competition. http://www.satcompetition.org.

[19] S. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM Press, 1971.

[20] J. Crawford and L. Auton. Experimental results on the crossover point in random 3-SAT. *Artificial Intelligence*, 81(1–2):31–57, 1996.

[21] J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the American Association for Artificial Intelligence (AAAI-94)*, pages 1092–1097. AAAI Press, 1994.

[22] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[23] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[24] N. Eén and N. Sörensson. An extensible SAT solver. In *Proceedings of the 6th international conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer-Verlag, 2003.

[25] P. Eerola, B. Konya, O. Smirnova, T. Ekelöf, M. Ellert, J. R. Hansen, J. L. Nielsen, A. Wäänänen, A. Konstantinov, and F. Ould-Saada. Building a production grid in Scandinavia. *IEEE Internet Computing*, 7(4):27–35, 2003.

[26] E. Erdem, V. Lifschitz, and R. Ringe. Temporal phylogenetic networks and logic programming. *Theory and Practice of Logic Programming TLPL*. To appear.

[27] Y. Feldman, N. Dershowitz, and Z. Hanna. Parallel multithreaded satisfiability solver: Design and implementation. *Electronic Notes in Theoretical Computer Science*, 128(3):75–90, 2004.

[28] S. Forman and A. Segre. NAGSAT: A randomized, complete, parallel solver for 3-SAT. In *Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*, 2002. Available online at http://gauss.ececs.uc.edu/Conferences/SAT2002/sat2002list.html.

[29] I. Foster and C. Kesselman, editors. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[30] E. Giunchiglia, Y. Lierler, and M. Maratea. SAT-based answer set programming. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI 2004)*, pages 61–66. AAAI Press, 2004.

[31] C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1–2):67–100, 2000.

[32] G. Grimmett and D. Stirzaker. *Probability and random processes*. Oxford University Press, Oxford, UK, 1982.

[33] T. Junttila. des2bc – generate boolean circuits from known plain text attack against bounded round DES, version version 1.0, 2005-12-02, 2005. Computer program, http://www.tcs.hut.fi/ tjunttil/circuits/.

[34] B. Jurkowiak, C. Li, and G. Utard. A parallelization scheme based on work stealing for a class of SAT solvers. *Journal of Automated Reasoning*, 34(1):73–101, 2005.

[35] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1194–1201. AAAI Press, 1996.

[36] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(1):4–15, 1992.

[37] C. Li. A constraint-based approach to narrow search trees for satisfiability. *Information Processing Letters*, 71(2):49–95, 1999.

[38] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[39] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535. ACM Press, 2001.

[40] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

[41] Nordugrid. http://www.nordugrid.org/.

[42] C. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, Inc., Boston, MA, 1994.

[43] T. Pyhälä. genfacbm — a benchmark generator based on factoring for SAT and ASP solvers, version 1.3, 2004. Computer program, http://www.tcs.hut.fi/Software/genfacbm/.

[44] D. Ranjan, E. Pontelli, and G. Gupta. On the complexity of or-parallelism. *New Generation Computing*, 17(3):285–307, 1999.

[45] L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, Burnaby, British Columbia, Canada, February 2004.

[46] A. Segre, S. Forman, G. Resta, and A. Wildenberg. Nagging: A scalable fault-tolerant paradigm for distributed search. *Artificial Intelligence*, 140(1–2):71–106, 2002.

[47] B. Selman. makewff – a random formula generator, 2004. Computer program, http://www.cs.washington.edu/homes/kautz/walksat/.

[48] L. Simon, D. Le Berre, and E. A. Hirsch. The SAT2002 competition report. *Annals of Mathematics and Artificial Intelligence*, 43(1–4):307–342, 2005.

[49] C. Sinz, W. Blochinger, and W. Küchlin. PaSAT — Parallel SAT-checking with lemma exchange: Implementation and applications. In *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, pages 12–13. Elsevier Science Publishers Ltd., 2001.

[50] C. Sinz and W. Küchlin. Verifying the on-line help system of SIEMENS magnetic resonance tomographs. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'2004)*, volume 3308 of *Lecture Notes in Computer Science*, pages 391–402. Springer-Verlag, 2004.

[51] O. Smirnova. The NorduGrid/ARC user guide, advanced resource connector (ARC) usage manual. Available online at http://www.nordugrid.org/documents/userguide.pdf.

[52] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cumberland, RI, 1994.

[53] W. Sullivan III, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A new major SETI project based on Project Serendip data and 100,000 personal computers. In *Proceedings of the Fifth International Conference on Bioastronomy*, number 161 in IAU Colloquium, Bologna, Italy, 1997. Editrice Compositori.

[54] M. Velev and R. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, February 2003.

[55] H. Zhang. SATO: An efficient propositional prover. In *Automated Deduction — CADE-14, 14th International Conference on Automated Deduction, Townsville, North Queensland, Australia, July 13-17, 1997, Proceedings*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275. Springer-Verlag, 1997.

[56] H. Zhang, M. Bonacina, and J. Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4):543–560, 1996.

[57] H. Zhang, D. Li, and H. Shen. A SAT based scheduler for tournament schedules. In *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT2004, Vancouver, BC, Canada, May 10–13, 2004*. Available online at http://www.satisfiability.org/SAT04/programme/index.html.

[58] L. Zhang. SAT-solving: From Davis-Putnam to Zchaff and beyond, lecture notes, 2003. Available online at http://research.microsoft.com/users/lintaoz/SATSolving/satsolving.htm.

[59] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285. IEEE Press, November 2001.

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
RESEARCH REPORTS

HUT-TCS-A87 Harri Haanpää, Patric R. J. Östergård
Sets in Abelian Groups with Distinct Sums of Pairs. February 2004.

HUT-TCS-A88 Harri Haanpää
Minimum Sum and Difference Covers of Abelian Groups. February 2004.

HUT-TCS-A89 Harri Haanpää
Constructing Certain Combinatorial Structures by Computational Methods. February 2004.

HUT-TCS-A90 Matti Järvisalo
Proof Complexity of Cut-Based Tableaux for Boolean Circuit Satisfiability Checking.
March 2004.

HUT-TCS-A91 Mikko Särelä
Measuring the Effects of Mobility on Reactive Ad Hoc Routing Protocols. May 2004.

HUT-TCS-A92 Timo Latvala, Armin Biere, Keijo Heljanko, Tommi Junttila
Simple Bounded LTL Model Checking. July 2004.

HUT-TCS-A93 Tuomo Pyhälä
Specification-Based Test Selection in Formal Conformance Testing. August 2004.

HUT-TCS-A94 Petteri Kaski
Algorithms for Classification of Combinatorial Objects. June 2005.

HUT-TCS-A95 Timo Latvala
Automata-Theoretic and Bounded Model Checking for Linear Temporal Logic. August 2005.

HUT-TCS-A96 Heikki Tauriainen
A Note on the Worst-Case Memory Requirements of Generalized Nested Depth-First Search.
September 2005.

HUT-TCS-A97 Toni Jussila
On Bounded Model Checking of Asynchronous Systems. October 2005.

HUT-TCS-A98 Antti Autere
Extensions and Applications of the A^* Algorithm. November 2005.

HUT-TCS-A99 Misa Keinänen
Solving Boolean Equation Systems. November 2005.

HUT-TCS-A100 Antti E. J. Hyvärinen
SATU: A System for Distributed Propositional Satisfiability Checking in Computational
Grids. February 2006.

ISBN 951-22-2068-X
ISSN 1457-7615