Parallel Resolution of the Satisfiability Problem: A Survey

Daniel SINGER

Publications du LITA, N° 2007-101
Service de reprographie de l'U.F.R. M.I.M.
Université Paul Verlaine - Metz

Parallel Resolution of the Satisfiability Problem: A Survey

Daniel SINGER

Publications du LITA, N° 2007-101
Service de reprographie de l'U.F.R. M.I.M.
Université Paul Verlaine - Metz

Laboratoire d'Informatique Théorique et Appliquée, EA 3097 Université Paul Verlaine - Metz
Directeur de la publication : Maurice Margenstern

Parallel Resolution of the Satisfiability Problem: A Survey

Daniel SINGER
L.I.T.A. Université Paul Verlaine - Metz
daniel.singer@univ-metz.fr

Abstract

The past few years have seen enormous progress in the performance of propositional satisfiability (SAT) solvers, and consequently SAT solvers are widely used in industry for many applications. In spite of this progress, there is strong demand for higher SAT algorithms efficiency to solve harder and larger problems. Unfortunately, most modern solvers are sequential and fewer are parallel. Our intention is to review the work of this last decade on parallel resolution of SAT with DPLL solvers which are the most widely used complete ones.

Keywords

Parallel Resolution, Propositional Satisfiability (SAT).

1 Introduction

The propositional Satisfiability problem (SAT) is one of the most studied in computer science since it was the first problem proven to be NP-complete by S. Cook in 1971. Nowadays, the Satisfiability problem evidences great practical importance in a wide range of disciplines, including hardware verification, artificial intelligence, computer vision and others. Indeed, one survey of Satisfiability in 1996 [28] contains over 200 references of applications. SAT is especially important in the area of Electronic Design Automation (EDA) with a variety of problems such as circuit design, FPGA routing, combinatorial equivalence checking and automatic test and pattern generation. In spite of its computational complexity, there is increasing demand for high performance SAT-solving algorithms in industry. Unfortunately, most modern solvers are sequential and fewer are parallel. Our intention is to review the work on parallel resolution of SAT with DPLL solvers for this last decade from the previous survey paper[27].

The remainder of this paper is organized as follows. Section 2 briefly introduces the SAT problem and major concepts of the field. Section 3 gives an

overview of the main techniques used in the efficient implementation of state-of-the-art sequential DPLL solvers. Section 4 describes the different proposed methods to parallelize the core sequential algorithms. Section 5 presents some of our experimentations in parallel resolution of SAT and it is followed by a brief concluding Section.

2 Preliminaries

Let $V = \{v_1, v_1, \ldots v_n\}$ be a set of n boolean variables. A (partial) truth assignment τ for V is a (partial) function: $V \rightarrow \{True, False\}$. Corresponding to each variable v are two literals: v and $\neg v$ called positive and negative literals. A clause C is a set of literals interpreted as a disjunction, $\Box$ denotes the empty clause and unit clauses have a single literal. A formula F is a set of clauses interpreted as a Conjunctive Normal Form (CNF) of a formula of the propositional calculus. A truth assignment τ satisfies a formula F (τ is *a solution*) *iff* it satisfies every clause in F, and the empty formula $\emptyset$ is always satisfied. A truth assignment τ satisfies a clause C *iff* it satisfies at least one literal in C and the empty clause $\Box$ is never satisfied.

Definition 2.1. *The Satisfiability Problem (SAT):*
-Input: A set of Boolean variables V and a set of clauses $\mathcal{C}$ over V.
-Output: Yes (gives a satisfying truth assignment τ for $\mathcal{C}$ if it exists) or No.

The restriction of SAT to instances where all clauses have at most k literals is denoted k-SAT. Of special interest are 2-SAT and 3-SAT; while 2-SAT is linearly solvable 3-SAT is NP-complete. The Max-SAT problem is the optimization variant problem of SAT to find a truth assignment τ that maximizes the number of satisfied clauses of $\mathcal{C}$. Nevertheless, the Max-2-SAT problem is well known to be NP-hard[66].

2.1 Complete-incomplete algorithms

Current research on propositional satisfiability is focused on two classes of solving methods: complete algorithms mostly based on *Backtrack search* and incomplete ones represented by variations of *Local search* methods. Complete algorithms are guaranteed to find a satisfiable truth assignment (a solution) if the problem is satisfiable, or to terminate with the proof of the problem unsatisfiability. Incomplete algorithms, on the other hand, cannot prove the unsatisfiability of instance even though they may be able to find a solution for certain kinds of satisfiable instances very quickly.

Moreover, SAT (or Max-SAT) can be formulated as a particular Integer Linear Program such that classical Operation Research methods can be applied including Linear Programming by relaxing the integrality constraints. For example this has been the case with linear or non-linear cutting planes, Lagrangian or Semidefinite techniques. Incomplete methods are based on efficient heuristics which help to delay the combinatorial explosion when the size of problem

increases. This category includes Simulated Annealing, Genetic Algorithms and Local Search methods. The most popular ones are GSAT[52] or WSAT[51] with a number of parallel implementations and studies [61, 34]. This survey will not present the work on the parallel resolution of SAT with incomplete algorithms and we refer to [27] for this approach.

Most of the more successful complete SAT solvers are instantiations of the Davis Putnam Logemann Loveland procedure[20] traditionally abbreviated as DPLL (see Figure. 1). Other complete methods include (general) Resolution or Davis Putnam (DP) algorithms for theoretical aspects of SAT or Automatic Reasoning, Ordered Binary Decision Diagrams (OBDD) based solvers and Stälmark's methods in EDA applications. Nowadays, DPLL variants work quite well in practice and are the most widely used SAT solvers.

We may also mention a number of works on the hybridation of incomplete and complete algorithms to solve Boolean Optimization problems, Branch-and-Bound or Tabou with DPLL or DP for example. This survey will report only these DPLL based complete methods for SAT resolution.

Figure 1: DPLL procedure

procedure DPLL ($\mathcal{F}$)
Begin
If $\mathcal{F} = \emptyset$ then return "satisfiable";
Else $\mathcal{F} \leftarrow$ UnitPropagation($\mathcal{F}$);
If $\Box \in \mathcal{F}$ then return "unsatisfiable";
Else /* Branching Rule*/
Choose l a literal according to some heuristic H;
If DPLL($\mathcal{F} \cup \{l\}$)= satisfiable then return "satisfiable" ;
Else DPLL($\mathcal{F} \cup \{\neg l\}$)
End

function UnitPropagation ($\mathcal{F}$)
Begin
While $\Box \notin \mathcal{F}$ and $\exists$ unit clause $l \in \mathcal{F}$
Satisfy l and Simplify $\mathcal{F}$
return($\mathcal{F}$)
End

2.2 SAT-CSP

There is continuing interest in translations betwen CSP (Constraint Satisfaction Problems) and SAT[3, 24, 71]. For example, [3] proposes to translate a SAT problem (F) into a binary CSP (P) as follows: to each clause C_i of F is associated a variable X_i with domain the set of literals of C_i. A constraint is defined between any pair of variables (X_i, X_j) if the clause C_i contains a literal and the clause C_j contains its complement. For each constraint (X_i, X_j) a relation R_{ij} is

defined as the Cartesian product D_ixD_j minus the tuples (t_i, t_j) such that t_i is the complement of t_j. In this approach SAT is reduced to the Path Consistency of the CSP. Moreover it suggests to apply the *Tree Clustering Decomposition* technique for solving the particular CSP associated to 3-SAT instances but we do not know the results. Many other translations are possible that consider the different graph structures of SAT. We mention this aspect because much more work has been done in the parallel resolution of CSP than SAT (for example [30, 31]), but surprisingly, to our knowledge no work takes this trail.

2.3 Decomposition techniques

Decomposition methods have been applied in a variety of problems such as CSP, especially for parallel resolution in[29], but few work exist in this direction for SAT. [1, 2] propose to solve SAT using a decomposition based on some approximation of the most constrained subproblems (those with the least number of solutions) by the c/v ratio parameter[1]. One approach decomposes a formula into a tree of partitions (ie. subformulas) using an algorithm to find *Minimum Vertex Separators*. Then a DPLL algorithm runs with a branching heuristic based on this decompostion solving first the most constrained subformulas. At last it performs a compilation procedure for each of the partitions and *joins* the results to answer the initial SAT problem. Another approach uses a *0-1 fractional programming optimization* algorithm to find the most constrained subformula. Then a *static* or a *dynamic* use of this algorithm may be applied by a DPLL solver. Unfortunately, we are not aware of any experimental results of these two propositions even in the sequential framework. [4] studies the applicability of *Tree Decomposition* method to SAT but only in the sequential framework. Tree decomposition is a graph theoretic concept which captures the topological structure of the CNF formula represented as a hypergraph. Formulas that have bounded *Treewidth* can be checked for satisfiability in linear time by translating them into Decomposable Negation Normal Form (DNNF)[16]. [4] considers the integration of this decomposition method to modern DPLL solvers to achieve better performance in terms of number of decisions (branching steps). The decomposition procedure not only guides the *variable ordering* process, but also the construction of conflict clauses with guaranted bounded size (see below Section 3.3). The reported results show consistent improvements (in terms of number of decisions) compared to traditional solver (zChaff) on benchmark problems with varying treewidth, especially with a static variable ordering approach. It is noticeable that it does not respond as well in terms of runtime because of an unoptimized implementation and overhead of the tree decomposer.

2.4 Experimentation: finding hard problems

Empirical evaluation of SAT solvers on benchmark problems has been of particular interest for both fundamental algorithms and theoretical understanding

[1] c = number of clauses and v = number of variables

of SAT[11]. Web sites[62, 63] are Satisfiability libraries that collect a number of benchmark problems, solvers and tools to provide a uniform test-bed for solvers. SAT seems to have less structure than many other combinatorial problems and, in spite of impressive engineering successes on many difficult problems, there are many *easy* problems with which SAT solvers struggle. These include problems involving parity, the well known "pigeonhole problem", and problems naturally described by first-order formulas. In all of these cases, the underlying structure of the problem is lost with the poor SAT encoding (see below Section 3.4). An anual SAT competition is held as a joint event with the SAT conference to identify new challenging benchmarks and to promote new SAT solvers. Each edition meets a great number of solvers on randomly generated or industrial benchmarks. The 2004 edition[45] pointed out the superiority of incomplete solvers on satisfiable random benchmarks contrary to industrial ones (satisfiable or not). The hardest instances for random benchmarks are generally much smaller in size than the hard structured industrial ones.

SAT has appeared to possess a property of great interest with respect to computational complexity. Random k-SAT formulas exhibit a so-called *Phase Transition Phenomenon*, that is when c clauses with exactly k literals over a set of v variables are chosen at random with a fixed ratio $r = c/v$, the probability of satisfiability falls abruptly from near 1 to near 0 as r passes some critical value τ_k called the *threshold*. Moreover, at the threshold (for $k \geq 3$) a peak of difficulty is observed and it grows exponentially as a function of v. Thus it appears that the *hard* formulas lay in this transition region near the threshold. In the case of $k = 3$ the threshold value for $\tau_k \approx 4.25$ and the best actual DPLL solver for random problems ($kcnfs$) can deal such hard random unsatisfiable 3-SAT instances up to 700 variables in around 25 days, thereby approaching practical feasibility.

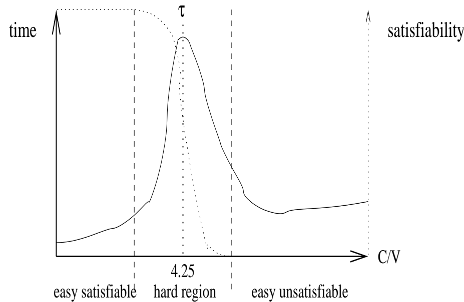

Figure 2: 3-SAT Phase Transition Phenomenon

The SAT phase transition phenomenon has attracted much attention from physicists. In particular, the concept of *Backbone* of a k-SAT formula has turned out to play an important role in theoretical studies to design new hard bench-

mark problems and new branching heuristics[55, 17]. A *backbone variable* of a formula is a variable that is assigned always the same truth value in all assignments that satisfy the maximum number of clauses or in all solutions if the problem is satisfiable.

3 Efficiency of Sequential DPLL SAT Solvers

There has been extensive research effort to develop gradually more efficient SAT solvers (see for example[25, 41, 62]). We will present in this section some of the main components for efficiency of sequential state-of-the-art DPLL solvers.

3.1 Better Branching Heuristics

The first element for efficiency of sequential solvers is determined by the *Branching Rule* heuristic. Two factors have to be considered to define good general purpose decision strategies. The first one is to find a solution if it exists as fast as possible and the second one is to detect a contradiction as early as possible. Moreover, a "good" heuristic is one that does not require too much time to compute and provides a fairly accurate cost estimate. All along the two last decades the branching rule problem has received many attentions and achieved many progresses.

$Moms$ for Maximum number of Occurrences in Minimum Size clauses is one of the most widely used general heuristic. It is a pure refutation based strategy which is simple, easy to implement and problem-independent. It favours the shortest clauses to obtain unit clauses and contradiction by Unit Propagation as a *First Fail Principle*. Many versions of $Moms$ heuristics have been proposed. Let $m_t(l)$ = number of clauses of size t that contains the literal l, $g^+(x) = \sum_t m_t(x)2^{n-t}$ and $g^-(x) = \sum_t m_t(\neg x)2^{n-t}$ for a variable x. Let $h(x) = g^+(x) + g^-(x)$, $Moms$ chooses the variable x with Maximum($h(x)$) and the literal with Maximum($g^+(x), g^-(x)$) to be the first explored branch. The 2-sided Jeroslow-Wang heuristic (*2s-JW*) for example, combine $g^+(x)$ and $g^-(x)$ in some fashion balancing the two branches, with $h(x) = g^+(x) + g^-(x) + g^+(x) * g^-(x)$.

The Böehm and Speckenmeyer (*BS*) heuristic[6] chooses the literal l with maximal vector $(M_2(l), M_3(l), \ldots, M_t(l))$ under the lexicographic order where $M_i(l) = \alpha Max(m_i(l), m_i(\neg l) + \beta Min(m_i(l), m_i(\neg l)$. Experiments have shown best results with $\alpha = 1$ and $\beta = 2$ and it calculates two elements of the vector only, $(M_s(l), M_{s+1}(l))$ where s is the size of the shortest clause of the formula (generally $s = 2$).

UP heuristic, such the one used in Satz[43] exploits the power of Unit Propagation by choosing the literal that would produce the maximal number of unit clauses. Let $w(l)$ = number of new binary clauses obtained by running UnitPropagation($\mathcal{F} \cup$ {l}). Satz will branch on the variable with Maximum($h(x)$) with $h(x) = w(x) + w(\neg x) + 1024 * w(x) * w(\neg x)$.

Recent works on dynamic learning and conflict analysis (see below Section 3.3) define new heuristics such as *VSIDS* for Variable State Independent

Decaying Strategy in Chaff[49] that are the best actual ones for very large industrial problems. Note that heuristics may be designed to suit particular classes of instances, for example $kcnfs$ [17, 18] is one of the best actual DPLL solver dedicated to solve hard random k-SAT problems. It uses a completely different heuristic based on backbone variables as branching nodes (see Section 2.4).

3.2 Careful Unit Propagation Implementation

DPLL solvers spend the bulk of their effort (greater than 90% in most cases) searching for clauses implied in Unit Propagation, sometimes called *Boolean Constraint Propagation*. Therefore, an efficient Unit Propagation procedure implementation is the key for efficiency. GRASP[53] and Satz[43] for example, keep two counters for each clause, one for the number of value *true* literals and one for the number of value $false$ literals. Each variable has two lists containing the clauses where this variable appears positively and negatively. In SATO[76] the *head/tail list* structure is introduced defining two pointers for each clause; one pointing to the first literal and the other to the last literal of the clause stored in an array. Each variable has four lists containing pointer to clauses that have their head/tail literal appearing positively and negatively respectively. The head/tail list method has been argued faster than the counters-based one because when a variable v is assigned value *true*, the clauses containing positively v will not be visited at all and vice-versa. One of the best actual solver using this structure is BerkMin[26] a closed-source solver. Unfortunately for both methods undoing a variable's assignment during backtrack has about the same complexity as assigning the variable. Chaff[49] proposed the *2-literal watching* structure that is similar to the head/tail list associating two special *watched literals* to each clause and two lists of pointers to clauses having their positive and negative watched literals corresponding to each variable. With this structure undoing an assignment takes constant time.

[77] gives an interesting deep case study on cache performance of SAT solvers showing that "cache friendly data structures is one of the key elements for an efficient implementation". It gives comparative results of the three different data structures on various application domains in term of run times, data access and cache miss rates. The counters-based solvers perform poorly due to the high cache misses. The best head/tail-based solver (BerkMin) and recent watched literals-based solver (zChaff) have similar good cache performance. It finds that there are still a lot of space for improvements because in current and future generations of microprocessors, the speed difference of main memory, L1 and L2 caches tends to be larger (see Section 4.3 for parallel efficiency).

Another important but basic remark for implementation efficiency is that a solver designed to handle a large number of variables should be quite different than a solver designed to handle fewer variables. Clearly, this explains why the best solvers for industrial benchmarks are not the best for hard random problems as mentioned in Section 2.4.

3.3 Dynamic learning based on conflict analysis and non-chronological backtracking

In 1996, these CSP look-back techniques were simultaneously introduced in DPLL algorithms by Marques-Silva, Sakallah[54] and by Bayardo, Schrag [7]. It has now become a standard and is implemented in most of recent SAT solvers[78]. At each decison step for a branching variable choice is associated a *decision level*. All variables assigned as a consequence of implications of a certain decision will have the same decision level as the decision variable. If a conflict is encountered the DPLL algorithm analyses it for backtracking to a level so as to resolve this conflict and a 0-level backtracking means that the problem is unsatisfiable. A clause is called *conflicting clause* if it has all its literals assigned to *False*. Advanced conflict analysis relies on an *implication graph* to determine the actual reasons for the conflict. This permits to backtrack up more than one level of the decision stack and, at the same time, to add some clauses called *conflict clauses* to a database. This last operation is the base for the *learning process* which plays a very important role in pruning the search space of SAT problems and this will be of particular interest in the parallel resolution framework. Clause learning and variable branching strategies have traditionally been studied separately but there is great promise in developing branching strategies that explicitely take into account the order in which clauses are learned. This is the case of Chaff[49] which developed domain specific strategies using such a learning preprocessing step.

3.4 Specific SAT problems processing

DPLL solvers typically suppose CNF encoded problems but this is seldom the natural formulation of “real world” applications. Indeed, the CNF representation provides conceptual simplicity and implementational efficiency, it also entails considerable loss of information about the problem’s structure that could be exploited in the search. There is a new interest in studying the CNF conversion for DPLL solving in different domains such as Planning, Bounded Model Checking (BMC) or Automatic Reasoning to improve search efficiency.

One approach[70] argues that this conversion is both unnecessary and undesirable. It presents a non-CNF DPLL like solver able to process any propositional formula represented as DAGs (Directed Acyclic Graphs) such as in OBDD-based solvers. Experimental results show performance better or close to that of the highly optimized CNF solver zChaff. It seems to be promising because many other potential ways of exploiting the structure have not been experimented. One opposit approach is to “optimize” the CNF conversion with respect to the number of generated clauses such as in[37], to the number of variables or both such as in SATPLAN04[38], which takes first place for optimal deterministic planning at the 2004 International Planning Competition. Another important feature to be mentioned is that for a number of real applications such as BMC, solving time may be largely dominated by encoding time thus “optimizing” may also refer to the conversion time.

Other different approaches[72, 5, 47, 32] tackle the general problem of equivalency reasoning. A lot of SAT applications contain a large number of equivalences (or Xor) and this results in poor efficiency of DPLL solvers because they produce very few unit clauses. For example, the notorious *parity-32 bits* DIMACS benchmark remained unsolved by general purpose SAT solvers for a considerable time and first solved by these approaches.

4 Parallel Resolution of SAT

Sequential computer performance improvements are the most significant factor in explaining the few existing works on Parallel Algorithms for Satisfiability compared to sequential ones. Indeed, the challenge to parallel processing is substantial in this area because there are still many problems considered out of reach for the best currently available solvers. There have been several parallel computation models implemented on modern parallel or distributed computers reflecting advances in new computational devices and environments. The two computational models used in this study are *Shared Memory* and *Message Passing*. In the shared memory model, each processor has access to a single, shared address space. In practice it is difficult to build true shared memory machines with more than a few tens of processors. A variation of the pure shared memory model is to let the processors have local memory and share only a part of the main memory such as in SMP clusters. In the message passing model, the processors have only local memory but are able to communicate with each other by sending and receiving messages. There are different network topologies for the connections between the processors. The message passing model is highly portable since it matches the hardware of most modern supercomputers as well as network of workstations. It is not unheard of for both models to be applied simultaneously -threads on shared memory for "node computations" and message passing among them. Such an hybrid approach could become standard and will be used in future works on Parallel Resolution of SAT for further progress.

Incomplete methods based on local search such as GSat, WalkSat or TSat are much more easily implemented on parallel machines because of their inherent parallel nature[27, 59]. There have been also some parallel implementations of the Davis Putnam (DP) complete algorithm[10, 56]. In the following, we will restrict our review to the complete but DPLL based parallel versions.

4.1 Randomized parallel Backtrack search

In 1992, [73] proposed a Monte Carlo type randomized algorithm for SAT which always gives an answer but not necessarily right. A number k of instantiations are randomly generated, where k is related to the probability ε of expected failure. It concludes the input formula to be satisfiable if any instantiation makes it *true* but perhaps concludes wrongly unsatisfiable otherwise. The random generation step may be parallelize but only the theoretical analysis of the polynomial average time is presented. [42] introduces a generic parallel *Backtrack* search

algorithm with a deterministic and a randomized method for the message passing model. Most of this work is dedicated to the *Branch and Bound* algorithm to prove its efficiency with a multinode-donation strategy and [79] analyses a single-node donation strategy. In 1994, [33] studies the behavior of parallel search for a complete backtrack based algorithm for graph coloring which is a particular CSP. Independent parallel searches are easily obtained with different heuristics. It concludes saying that such concurrent parallelization gives fairly limited improvement on hard problem instances.

More recently, *Nagging*[23] is a distributed search paradigm that exploits the speedup anomaly (see below Section 4.2) by playing multiple reformulations of the same problem – or portions of the problem – against each other. Originally developed within the relatively narrow context of distributed automated reasoning, it has been generalized and used to parallelize DPLL algorithm. Partial results on hard random 3-SAT instances empirically show in this case the possible performance advantage of nagging over partitioning. Moreoever and aside from performance considerations, Nagging holds several additional practical advantages over partitioning; it is intrinsically fault tolerant, naturally load-balancing, requires relatively brief and infrequent interprocessor communication, and is robust in the presence of reasonably large message latencies. These properties may contribute directly to Nagging's demonstrated scalability, making it particularly well suited for use on the Grid. Unfortunately, we are not aware of any new results in this direction.

4.2 Search space partitioning

An important characteristic of the SAT search space is that it is hard to predict the time needed to complete a specific branch. Consequently, it is difficult (or impossible) to statically partition the search space at the beginning of the algorithm. To cope with this problem, most of the parallel algorithms dynamically partition the search space assigning work to the available threads during runtime. The most difficult part consists of balancing the workload in such a way that on the one side idle time should be limited, and on the other side the workload balancing process should consume as few computing and communication time as possible.

In 1985, the seminal works of Speckenmeyer et al.[50, 67] introduce the important notion of *autarky* to prove the worst case complexity of 3-SAT to be $\mathcal{O}(1.6181^n)$, and it shows the fact that in average the solutions of k-SAT problems are non-uniformly distributed. This was the first explanation of the experimental *superlinear speedup* obtained by a parallel backtrack search with a fixed number of processors. Nowadays, this is a well known phenomenon sometimes called *anomaly*, corresponding to the non-deterministic treatment of the search tree by a parallel execution. Moreover, this is the reason to dissociate satisfiable from unsatisfiable problems for parallel resolution evaluation. The other one reason is that for satisfiable instances only a portion of the tree is explored, thus parallel execution highly depends on the order of the parts to be searched.

Let F be a formula as a set of clauses and $\mathcal{L}it(F)$ be the set of literals defined by the variables of F. An *autarky* A is a subset of $\mathcal{L}it(F)$ such that it defines a partition of $F = autsat(A) \oplus autrem(A)$ with $autsat(A) = \{C \in F/\exists l \in A : l \in C\}$ and $autrem(A) = \{C \in F/\forall l \in A\text{: } l \notin C \text{ and } \neg l \notin C\}$. Its main property is that, after assigning *true* the literals of an autarky A if it exists, the satisfiability of F is reduced to that of $autrem(A)$ for which all the variables associated to A are eliminated. There are a number of works investigating this major concept (eg.[39]), but to our best knowledge only [56] used it for parallel resolution in a *Model Elimination* algorithm which is not DPLL based.

In 1994, Böehm and Speckenmeyer[6] gives the first real parallel implementation of the DPLL procedure on a message based MIMD (Multiple Instructions Multiple Data) machine, and it is the reference work of the domain. Excellent efficiencies have been obtained on a Transputer system with up to 256 T800 processors and with two different connexion topologies, the linear array and the 2-dimensional grid. The authors suggest the grid topology for much more than 256 processors! The sequential version of the solver was the fastest program at the first SAT competition in 1993. Its quality heavily depends on the branching heuristic (see 3.1) and the associated optimized dynamic data structures. Doubly chained lists with ascending order are used to assign and unassign variables. The sophisticated dynamic workload balancing strategy is the key for parallel efficiency. Even if no reliable measure of workload for a subformula is known, a simple estimation function α^n for a partial truth assignment with n unset variables and α varying between 1.04 and 1.42 is used (remember the previous upper bound of 1.6181). If the estimated workload for some processor goes down some limit then the workload redistribution phase is activated. Each processor runs two processes, the *worker* and the *balancer*. In the initialization phase the input formula F is sent to all processors p and a list L_p of partial assignments representing subformulas. The worker process tries to solve or split subformulas of its list by assigning *true* resp. $false$ to a literal x chosen according to the branching heuristic. If its list is empty the worker process waits for either new work or a termination message of the balancer process. The balancer process of each processor p estimates its workload $\lambda(p)$ and a precomputation phase calculates the amount of workload to be sent or received by each processor. The balancing phase is performed only if at least one processor holds less than s problems (actually 3) in its list to reduce communication overhead. Sampling method to adjust the α value is presented for hard random k-SAT formulas and for hard unsatisfiable Tseitin's graph formulas. To our best knowledge, there is no recent work on modern parallel machines which report so good results with hundreds of processors.

In 1994, Zhang et al.[74, 75] present PSATO the first DPLL solver for distributed architectures. It is based on the sequential DPLL solver SATO [76] which was at that time one of the most efficient. The constant need of more computing power and the networked workstations underused justified this approach. The other major motivation of this work was to attack open problems in the quasi-groups algebraic domain. The next important concept of *guiding path* was for the first time introduced to define non-overlapping portions of the

search space to be examined. It does so by recording the list of variables to which the algorithm assigned a value up until the given point of execution (the current state of the process). For each variable of the guiding path is associated its currently assigned truth value, as well as a boolean flag saying that whether the variable is *closed* (value 0): both truth values have been tested or *open* (value 1): one truth value remains to be tested (see Figure 3 for an example).

This notion provides a simple and effective way to define "cumulating search" allowing to suspend and continue the run without repeating the work already done. It is also the mean to deal with the fault-tolerance problem and splitting guiding path is the workload balancing method. Surprisingly, PSATO is organized according to the centralized *master-slave model* for workload balancing, opposite to the completely distributed preceding proposition. All communications take place between the master and the slaves, but implementation was done in an obsolete language called P4. The reported experimental results show good speedup with 20 workstations but are not significant for a number of reasons. However, PSATO made new discoveries in the quasi-group theory.

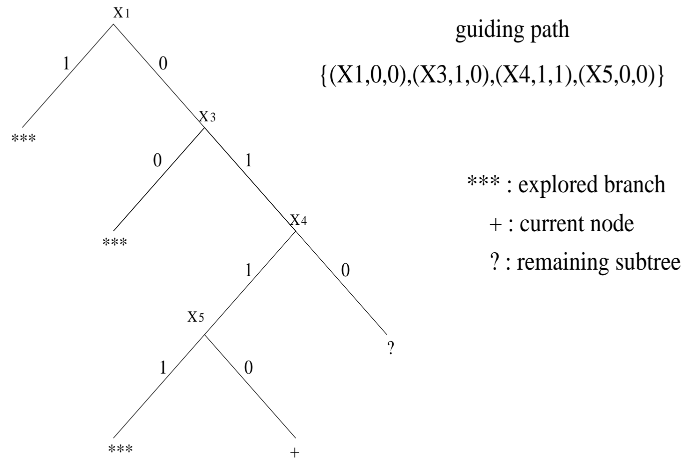

Figure 3: guiding path example

More recently, Jurkowiak et al. present //Satz[35], a parallel-distributed DPLL solver based on the *master-slave* communication model and *work stealing* for workload balancing. The guiding path concept is used to define the tasks associated to parts of the search tree. The sequential solver Satz which is based on, is equipped with a sophisticated experimental Unit-Propagation branching heuristic (see Section 3.1). The master evaluates the workload of a slave processor by the depth of the first remaining subtree to be explored in its guiding path: the smallest corresponds to the most loaded one. If two slaves have the same value then the second remaining subtrees are compared. When a slave becomes idle it sends a work request to the master which sends back the first remaining subtree of the most loaded slave. Every process has two threads:

the working thread and the communicating one. Moreover, the communication strategy used a semaphore mechanism implemented in standard RPC (Remote Procedure Call) to be easily portable on all Unix-like networks and on the Grid. This work emphasizes the *ping-pong phenomenon* which may occur in workload balancing. A process A sends its right subtree to the idle process B but quickly finds a contradiction in its left subtree and becomes idle. The process B then sends its right subtree to process A but quickly becomes idle for the same reason. It is argued that experimental UP branching rule prevents //Satz from this phenomenon. [36] presents experimantal results on a cluster of 216 PIII PCs interconnected by a Fast-Ethernet network. Significant speedup is obtained for hard random unsatisfiable 3-SAT instances with more than 500 variables up to 128 processors. For easier problems the communication overhead and the Satz preprocessing done by all slaves penalize the overall parallel performance. For structured real world SAT problems the results are less convincing.

4.3 Intelligent backtracking and lemma exchange

In 2001, Blochinger et al.[8, 9] propose PaSAT the first parallel DPLL solver with *intelligent backtracking* and lemma exchange for *learning*. As mentioned in Section 3.3, learning has now become a standard and is implemented in most of recent sequential SAT solvers. The guiding path approach for dynamic problem decomposition is used as well and the lemmas to be exchanged between the nodes of the distributed environment are the *conflict clauses*. Since at every leaf in the search tree a conflict analysis is carried out, a vast number of lemmas are generated at each node thus a selection rule has to be applied at the source. The clause size defines this criterion which is consistent with the sequential algorithm. Experiments to determine an appropriate value for this parameter are reported. Moreover, when inserting "foreign lemmas" into a task's clause set, these are filtered by the receiver node. Only those lemmas that are not *subsumed* by the task's initial guiding path are incorporated, to prevent insertion of superfluous lemmas for the receiver. A randomized *work stealing* strategy for workload balancing is applied in a *master-slave* communication model. The algorithm is implemented on top of the DOTS (Distributed Object-Oriented Threads System) platform. This system is designed for the prototyping and the parallelization of highly irregular symbolic applications such as SAT. It is natively object-oriented (C++) and it has been deployed on a number of (heterogeneous) clusters. It gives a strict multithreading programming tool in a fork-join style, located at a medium level of abstraction (compared to MPI), providing transparent synchronization and communication. A detailed experimental study discuss the different parameters involved by the algorithm on a cluster of 24 SUN workstations. It used the *quasi-group* and BMC *longmult* for unsatisfiable instances, and *DES* (3 rounds) logical cryptanalysis for satisfiable ones. This includes the *split-threshold* parameter which defines when a search space split is performed to produce an additional thread, *steal-threshold* parameter which defines when to steal a thread from a randomly chosen "victim node" and other *timing* parameters. It is not a trivial work to understand the

lemma exchange effect and benefit of learning on the overall performance of the program, but it is argued in conclusion that distributed learning can result in considerable speedup. Definitively in our opinion, thorough theoretical investigations and additional experimentations have to be managed in this direction.

In 2004, Feldman et al.[22] present a parallel multithreaded SAT solver on a single multiprocessor workstation with a *shared memory architecture*. It provides experimental results for the execution of the parallel algorithm on a variety of single multiprocessor machines with a shared memory. The emphasis has been on providing an efficient portable implementation that improves runtime by distributing the workload over several processors on the single machine. To understand low-level implementation details the solver was designed and coded from scratch contrary to other implementations based on *zChaff*[13] or other existing publicly available source code of solvers such as *Satz*[35]. The performance of the solver in a single-threaded configuration on a benchmark suite of about 500 industrial tests is comparable to the *zChaff* one[49]. The main contribution of this paper is that it shows the general disadvantageousness of parallel execution of a backtrack-search algorithm on a single multiprocessor workstation, due to increased cache misses. More precisely, it observes the negative effect on otherwise highly optimized cache performance of the sequential algorithm (see Section 3.2). It incorporates most of the precedent technologies for parallel execution as well as state-of-the-art sequential ones presented in Section 3. Guiding path for search space partitioning and a fork-join style for multithreading programming are used. The search algorithm is parallelized by letting threads explore the search space defined by the open variables on the guiding path of the current thread. The execution of the parallel algorithm starts with one thread that is assigned the whole problem instance, thus defining a number of open variables on its guiding path. If all threads are in suspension while waiting for an available task, this indicates that the search space has been fully explored and the formula is unsatisfiable. For dynamic workload balancing, to minimize the thread waiting times and the time to find a new task, a list of available tasks is maintained. When a thread introduces a new open variable it adds its description to the list. The number of open variables is usually much larger than the number of working threads so that they add new tasks to the list only until a ceratin threshold on the list size is reached. To reduce the overhead of task switches, the tasks are chosen as candidates for entering into the list the ones with open variable closest to the beginning of the guiding paths such that its expected running time is higher. Moreover the tasks list is similarly sorted according to the length of the guiding path to its open variables. As in the previous work, the parallel algorithm performs special treatment of the *conflict clause* lemmas produced by each thread for distributed learning. This is done by maintaining a list of conflict clauses that is accessible from all threads. These two lists of available tasks and conflict clauses together with the initial clauses formula are the only shared data structures. Their synchronization overhead is claimed insignificant compared to the overall performance but experimental results show the contrary. This becomes worse when the number of working threads is increased. Specially severe performance degradation are reported for

(physical or logical) multiprocessors systems. In the worst case, with only four threads 10% of the total running time is spent in waiting on synchronization locks for shared data structures and the authors argue that this could not explain the observed degradation. With the help of the $Intel\ VTune^{TM}\ Performance$ *Analyser* they present a deep investigation on the reasons of this degradation mainly the amount of cache misses. Their interesting conclusion is that optimized cache performance for modern sequential SAT solvers contadicts the parallel execution efficiency on a single multiprocessor workstation!

It is worthly to notice that in 1996, [64] already studied a multithreaded SAT solver with a dynamic load balancing strategy similar to the one described in[6] (see Section 4.2). It reports good performance on randomly generated 3-SAT formulas up to 32 threads but, and this perhaps makes the difference, the threads were simulated on a single processor SUN workstation with a simple round robin schedule.

4.4 Grid Computing

In 2003, Chrabakh et al.[12, 13] present *GridSAT* the first DPLL SAT solver designed to solve real "hard" and previously unsolved satisfiability problems on a large number of widely distributed and heterogeneous resources (the Grid). Its philosophy is to keep the execution as sequential as possible and to use parallelism only when it is needed. It is based on *zChaff*[49] as sequential core solver but it implements a distributed learning clause database system (as discribed in precedent Section 4.3) that can acquire and release memory from a Grid resource pool on demand. It is developed from "first-principles" for the Grid and not a legacy parallel application with modifications for Grid execution. Two different types of resources are managed by this system: the time-shared and the batch controlled ones making an important difference with all the previous parallel or distributed propositions. This initial GridSAT implementation uses the *Every-Ware* development toolkit in a master-slave style communication system, to prove its feasibility. The baseline Grid infrastructure is provided by the *Globus* MDS and NWS (Network Weather Service) systems. The *GridSAT* scheduler located in the master node, is the focal point and is responsible for coordinating the execution and launching the clients (slaves). It uses a progressive scheme for acquiring resources and adding them to the pool because of the variability and unpredictability of resource need for a particular SAT instance. Typically, the master requests the resource list available from deployed Grid services or simply from a configuration file. The scheduler submits any batch jobs to their respective queues. When a remote client starts running, it contacts the client manager also located in the master node and registers with it. The scheduler ranks the set of available clients based on their processing power and available memory. It uses the first available client to start solving the problem. A client notifies the master that it wants to split its assigned subproblem when its memory usage exceeds a certain limit or after running for a certain period of time. The splitting process is performed by the cooperation of the splitting client, the master and the idle client and it uses a *sender-initiated strategy* for

load balancing with five messages in a peer-to-peer fashion to communicate one splitting. In addition, the master can direct a client to migrate its current problem instead of splitting it. *GridSAT* implements a limited form of resilience in the presence of failures based on check-pointing, except for machine crash or “out-of-memory killer” process termination.

The experimental results are obtained on different but non-dedicated nationally distributed Grids, and all the resources were in continuous use by various other applications. A number of various challenge problems of the SAT’2002 conference is presented as test applications, including industrial, hand-made and randomly generated instances already solved by a sequential solver or open ones. One experimentation used a few dozen of machines distributed among three or four sites in California including small clusters. For the hardest problems, 100 nodes of the Blue Horizon batch system with each node having 8 CPUs and 4 gigabytes of memory were requested moreover! Because *GridSAT* is not a traditional parallel application the results are not very good in terms of speedup for a number of reasons, but the authors conclusion is that “parallel solver is more efficient than a sequential one”. Their argument is that it solved open problems such as one *parity-32 bits* instance with 8 hours of 800 CPUs of Blue Horizon plus 33 hours of other resources. But this is not true, as noticed in Section 3.4 it was first solved in 1998, and also in 2000 with 20 minutes of one processor running time! In [14] improvements over the previous implementation and new experimental results are presented. In particular, it defines a new method for lemma exchange which is too complex to detail here, and new problems first solved. But as mentionned by the authors themselves, it remains an open question to decide when using more resources increases the solver’s performance, especially in this case of Grid computing.

4.5 Miscellaneous

In 1998 Kokotov at MIT, proposes PSolver[40] the first brick of an ambitious distributed SAT solving framework. In the spirit of Grid computing, it considers a large network of workstations which are underused or idle for significant periods of time. It allows these resources to run a client that solves portions of SAT instances in a master-slave fashion as well. A server maintains a list of subproblems to be solved (by their guiding paths), doles out these tasks to clients which connect and disconnect asynchronously, then aggregates their results. The main features of the project were the following: solver-independence, any sequential SAT solver can be used at the client end; high scalability, a network (a cluster) can run a bunch of clients and a server which in turn acts as a client to a higher-level network; volontary computing environment, similar to the SETI@home project, support for asynchronously connecting and disconnecting clients; fault tolerance, crashing clients do not affect the server; hardware/OS independence, PSolver is written in C++ with Solaris, Linux and Windows distributions, and it relies on the basic TCP/IP stack for its communication. Unfortunately, this ambitious but precursory work has not been carried on, and we are not aware of any experimental results.

To our best knowledge, Cope et al.[15] is the unique published work to investigate the parallel functional programming for DPLL implementation, inspite of its natural recursive expression. This proposition relies on the parallel evaluation of both left and right branches (see Figure 1) when the splitting rule is applied and it is implemented in a parallel dialect of *Haskell*. It incorporates the conflict-directed backjumping (CBJ) technique but only for non-chronological backtracking. A poor experimentation is reported and the early results obtained by simulation show "modest speedup". Unfortunately, and perhaps because of the impressive progress of the iterative implementation of DPLL, there is no new proposition in this direction.

Unlike all the presented works that may be qualified as *Macro Parallelism*, a completely different approach relies on specific hardware circuitry conception to parallelize SAT solving, and it may be qualified as *Micro Parallelism*. Among others, the *FPGA* (Field Programmable Gates Array) technology gives another way of using the inherent parallelism inside the circuitry (pipelines eg.). Its first main characteristic is that it permits to adapt (by reconfigurating) the hardware to a specific problem instance, taking into account the inherent structure of computed functions and data structures. Ultimately, this allows one to find a solution heeding the specific instance complexity rather than the general worst case one. A great deal of research effort has been done in this area from the initial works of Suyama et al.[69] and Platzner et al.[58] in 1998 (see [65] for a recent review). FPGAs are composed of programmable gates embedded in a programmable interconnection routing network and the programmable gates are implemented using LookUp Tables (LUTs). While significant speedup has been reported over software implementations for a number of applications, a few fundamental hurdles remain before it can be widely applied.

5 Our Experimentation

We will present in the following our experimentation in parallel SAT solving. It is a complementary approach of all the recent propositions for dynamic workload balancing, in the sense that it only explores an initial decomposition for workload repartition. The two computational models of *Shared Memory* and *Message Passing* are compared, using OpenMP for Shared Memory and MPI for Message Passing implementations. OpenMP[57] is a complete API for directive-driven parallel programming of shared memory computers. Fortran, C and C++ compilers supporting OpenMP are available for Unix and Windows workstations. OpenMP becomes the de facto standard for writing portable, shared memory, parallel programs. MPI[68] is a portable standard for message passing applications and can be used on distributed-memory or shared-memory multiprocessors, as well as on homogeneous or heterogeneous workstations networks or clusters. It provides a large library of functions including point-to-point and collective communications. While message passing reigns as the practical choice for large-scale parallel computing, threads are the medium of choice for SMP multiprocessors.

5.1 Sequential SAT solver choice

Our approach is to be as possible independent of the sequential solver running on all the processors for parallel execution. We only put a parallel layer upon the sequential solver that is viewed as a blackbox. Among all the recent freeware DPLL implementations we have experimented versions of *zChaff*, *Sato*, *kcnfs* and *Satz*. Here, for lack of place we will only present partial results obtained with *Satz* to illustrate this simple approach, however actually the best absolute execution times have been obtained with *zChaff*. The target sequential solver *Satz* developped by Chu Min Li [43, 46] is written in C which enables the use of OpenMP and MPI as well without any extra processing. One important feature of *Satz* compared to other DPLL implementations is that it explores independently left and right subtrees making easier parallel implementation. It has no intelligent backtracking and sophisticated conflict analysis for learning, thus reducing the posible communications for lemma exchange between processors. As previously presented in Section 3 it uses experimental Unit Propagation has branching rule heuristic to prune as much as possible the search tree. One may notice that for the other solvers intelligent backtracking and learning are reduced to one processor.

5.2 Initial decomposition strategy

The first step of our parallel proposition aims to obtain at most 2^k independent subproblems assigning both possible values $true$ and $false$ to some k " well chosen" variables. At each variable choice, the simplifications obtained by Unit Propagation are achieved. All the subproblems are placed in a stack that can be then dynamically allocated to processors all along the parallel execution. In OpenMP implementation this is obtained by a simple parallel $forloop$ with a dynamic allocation strategy directive. In MPI implementation, a classical master-slaves communication protocol has to be writen. The value for the k parameter may be adapted to the available processors number such that $2^k >> Nbproc$. This k parameter reflects the parallel *granularity* of our application and is studied in subsequel Section 5.4 giving the effective number of subproblems to be proceed. In practice, it never goes beyond 10 giving at most 1024 potential tasks, sufficient for the maximum of 32 processors architecture we used for this reported experience. The strategy for the choice of these k variables used to initially partition the problem is of particular importance. We will consider the three following strategies.

1. *Satz* : the branching heuristic of *Satz* to give comparative results with the sequential resolution.

2. *Rand* : the random choice of variables.

3. *Moms* : the classical Maximum number of Occurrences in Minimum Size clauses heuristic (see Section 3.1).

5.3 SAT applications

The first SAT application we used for this experimentation is called the *longmult* family, and it comes from the BMC (*Bounded Model Checking*) domain. Each of the 16 instances of the family is associated with one output bit of a 16x16 shift and add multiplier. They are reputed difficult for the OBDD (Ordered Binary Decision Diagrams) approach and are classical benchmark for DPLL SAT solvers. They can be found at Armin Biere web page[2], and all the instances are unsatisfiable. We will present only the hardest ones *longmult14* and *longmult15*. The second SAT application we used is called the *DES* family, and it comes from the logical cryptanalysis domain initiated by Fabio Massacci et al. [48] in 1999. It is a new way to generate hard and structured SAT problems by light encoding a few numbers of tours of DES encrypton system. It can be found at SATLIB site [62], and all these instances are satisfiable. We will present only the hardest ones *b1-k1.1* and *b1-k1.2* for three tours corresponding to the presentation of one block of the plaintext and one or two blocks of the cyphertext.

5.4 Experimental results

We conducted the experimentation on a SGI Origin 3800 machine thanks to the CINES (*Centre Informatique National de l'Enseignement Supérieur*). The machine configuration is 768 R14000/500MHz processors, and 384Go of memory. Its architecture is of ccNUMA type, made of a number of interconnected blocks of processors giving 1.6GB/s data transfer rate and latency lower than 1.4μs. The computing environment is LSF for batch jobs submission, moreover the system guarantees the exclusive use of the requested processors number for all the job predefined duration.

The following tables present for each problem instance: its reference, the number of variables v, the number of clauses c, the sequential CPU time (in secs) of *Satz*, the respective parallel time obtained with 4, 8, 16 and 32 processorrs, and their respective efficiency. Parallel efficiency is computed by :

$$Ef_{nb\ of\ proc} = \frac{(Satz\ sequential\ time)}{(parallel\ time) \times (nb\ of\ proc)}$$

Table 1 gives comparative results of OpenMP and MPI implementations on the *longmult* family with the same *Satz* initial decomposition strategy and the same granularity giving about 120 subproblems for *longmult14* and 200 subproblems for *longmult15*. Actually, there was a contradiction between the use of OpenMP for this application and our basic choice not to go inside the solver code. State-of-the-art DPLL solvers such as *Satz* make intensive use of dynamic data structures, but unfortunately OpenMP does not yet permit the dynamic private (not shared) memory allocation, thus leading to mandatory memory management overhead (see [31]). It is specially the case for more than 8 processors, and when OpenMP encounters an out-of-memory to give up the task for this reason a "?" is put in the table.

[2] http://www.cs.cmu.edu/ modelcheck/bmc.html

In all the cases the MPI implementation overcomes the OpenMP one, but both provide noticeable linear speedup until 8 processors, then a regular decreasing efficiency for MPI and out-of-memory for OpenMP with more processors.

Table 1: OpenMP-MPI comparison with *Satz* decoposition strategy

Pb.	Strat.	Seq	4 Pr.	Ef_4	8 Pr.	Ef_8	16 Pr.	Ef_{16}	32 Pr.	Ef_{32}
Lm14	Satz	4053	t4	e4	t8	e8	t16	e16	t32	e32
7.176v	OMP		1060	0.95	540	0.93	384	0.66	?	?
22.389c	MPI		971	1.04	503	1.00	327	0.77	284	0.44
Lm15	Satz	4865	t4	e4	t8	e8	t16	e16	t32	e32
7.807v	OMP		1251	0.97	665	0.91	?	?	?	?
24.351c	MPI		1211	1.00	622	0.97	361	0.84	274	0.55

Table 2 gives the comparative results of the three decomposition strategies *Satz*, *Rand* and *Moms* with MPI implementation on both *longmult* unsatisfiable family and *DES* satisfiable one. We may notice the *Satz* strategy to be always near the best one in this experiment. It shows superlinear speedup for the satisfiable instances. A non surprisingly random behaviour of the *Rand* strategy is observed too. It is wortwhile to notice that both *Rand* and *Moms* strategies are much more easily implemented compared to the *Satz* one which runs the solver up to a depth bound. Moreover we have not reported the CPU time needed for the decomposition step which is not insignificant especially in the *Satz* case.

Table 2: Decomposition heuristics comparison with MPI

Pb.	Strat.	Seq	4 Pr.	Ef_4	8 Pr.	Ef_8	16 Pr.	Ef_{16}	32 Pr.	Ef_{32}
Lm14	Satz	4053	971	1.04	503	1.00	327	0.77	284	0.44
7.176v	Moms		1750	0.58	891	0.57	552	0.46	291	0.43
22.389c	Rand		2050	0.49	1286	0.39	748	0.34	410	0.31
Lm15	Satz	4865	1211	1.00	622	0.97	361	0.84	274	0.55
7.807v	Moms		2027	0.60	1102	0.55	568	0.53	350	0.43
24.351c	Rand		2448	0.50	1505	0.33	1108	0.27	554	0.27
b1-k1.1	Satz	6352	1319	1.20	660	1.20	336	1.18	173	1.15
307v	Moms		1562	1.01	797	0.99	423	0.94	150	1.32
1731c	Rand		1823	0.87	702	1.13	428	1.08	187	1.06
b1-k1.2	Satz	7709	1595	1.20	839	1.14	430	1.12	220	1.10
398v	Moms		1994	0.96	921	1.04	401	1.20	154	1.56
2124c	Rand		65	29	114	8.45	105	4.58	1461	0.16

The last Table 3 presents the comparative results obtained with different granularity values for the *longmult* unsatisfiable family. It gives for each number of generated subproblems, its efficiency obtained with the *Satz* decomposition strategy in MPI implementation. It shows the important effect of this parameter on the overall efficiency especially for scalability.

Table 3: Granularity study

Pb.	NB-Pbs.	Seq	4 Pr.	Ef_4	8 Pr.	Ef_8	16 Pr.	Ef_{16}	32 Pr.	Ef_{32}
Lm14	32	4053	1308	0.77	1182	0.43	1070	0.23	1079	0.12
7.176v	60		1099	0.92	721	0.70	610	0.41	605	0.20
22.389c	112		990	1.02	513	0.99	344	0.73	292	0.43
Lm15	46	4865	1384	0.88	1009	0.60	832	0.36	786	0.19
7.807v	92		1273	0.95	728	0.83	568	0.53	456	0.33
24.351c	184		1247	0.97	628	0.97	364	0.83	277	0.55

6 Conclusion

We have mainly review the work on parallel resolution of the Satisfiability problem by a DPLL solver. After briefly introduce the different components for sequential efficiency of the state-of-the-art solvers, we give an almost chronological presentation of the essential steps towards the parallel framework progress of this last decade. However significant parallel efficiencies are still obtained by a number of recent propositions, more detailed research in all the directions are needed to improve the parallel performance. Specific works on decomposition techniques, hybridation of complete-incomplete algorithms, learning ability for example have to be started up, as well as general study for parallel application to Satisfiability problem. Indeed new improvements can be done in the sequential resolution, the next step for a widespread use of Satisfiabiliy technique in real world applications remains the parallel efficiency challenge.

References

[1] E. Amir, S. McIlraith, Solving Satisfiability using Decomposition and the Most Constrained Subproblem, In *Proc. of SAT'2001*[41], 2001.

[2] E. Amir, S. McIlraith, Partition-Based Logical Reasoning for First Order and Propositional Theories, *Artificial Intelligence Journal*, 2004.

[3] H. Bennaceur, The satisfiability problem regarded as a constraint satisfaction problem, In *Proc. of ECAI'96*, pp.155-159, 1996.

[4] P. Bjesse, J. Kukula, R. Damiano, T. Stanion, Y. Zhu, Guiding SAT Diagnosis with Tree Decompositions, In *Proc. of SAT'2003*, 2003.

[5] P. Baumgartner, F. Massacci, The Taming of the (X)OR, In *Computational Logic - CL 2000*, LNAI Vol.1861, pp.508-522, 2000.

[6] M. Böehm, E. Speckenmeyer, A fast Parallel Sat Solver - efficient work load balancing, In *Third Int. Symp. on AI and Maths.* AIMSA, Fort Lauderdale, Florida USA, 1994.

[7] R. Bayardo Jr., R. Schrag, Using CSP look-back techniques to solve real-world SAT instances, In *Proc. of CP'96*, LNCS 1118, pp.46-60, 1996.

[8] W. Blochinger, C. Sinz, W. Küchlin, PaSAT-Parallel SAT-Checking with Lemma Exchange: Implementation and Applications, In *Proc. of SAT'2001*[41], 2001.

[9] W. Blochinger, C. Sinz, W. Küchlin, Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 29(7):969-994, 2003.

[10] W.T. Chen, L.L. Liu, A Parallel Approach for Theorem Proving in Propositional Logic, *Information Sciences*, Vol.41, pp.61-76, 1987.

[11] S. A. Cook, D. G. Mitchell, Finding Hard Instance of the Satisfiability Problem: a survey, In *DIMACS Series in Dicrete Maths. and TCS.*, AMS, Vol.35, pp.1-17, 1997.

[12] W. Chrabakh, R. Wolski, GrADSAT: a Parallel SAT Solver for the Grid, TR.2003-05, CS. University of California, Santa Barbara, 2003. (http: www.cs.ucsb.edu/research/trcs/docs/2003-05.pdf)

[13] W. Chrabakh, R. Wolski, GridSAT: a Chaff-based Distributed SAT Solver for the Grid, *Super Computing Conference, SC'2003*, Phoenix Arizona, USA 2003.

[14] W. Chrabakh, R. Wolski, Solving "hard" stisfiability problems using GridSAT, 2004. (http:: www.cs.ucsb.edu/∼chrabakh/papers/gridsat-hp.pdf)

[15] M. Cope, I. Gent, K. Hammond, Parallel Heuristic Search in Haskell, In*Trends in Functional Programming*, Vol.2, pp.65-73, S. Gilmore ed., Intellect Books, Bristol, UK, 2000.

[16] A. Darwiche, Compiling knowledge into decomposable negation normal form, In*Proc. 15th.IJCAI*, pp.284-289, 1999.

[17] G. Dequen, O. Dubois, A backbone-search heuristic for efficient solving of hard random 3-SAT formulae, In*Proc.17th. IJCAI*, pp.248-253, 2001.

[18] G. Dequen, O. Dubois, $kcnfs$: an efficient solver for random k-SAT formulae, In*Proc. SAT'2003*, LNCS series. 2919, pp. 486-501, 2003.

[19] E. Dantsin, A. Goerdt, E. A. Hirsch, et al. , A Deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search, *TCS*, Vol.289, pp.69-83, 2002.

[20] M. Davis, G. Logeman, D. Loveland, A machine program for Theorem Proving, CACM, Vol.5 (7), 1962.

[21] R. Finkel, U. Manber, DIB, A Distributed Implementation of Backtracking, *(ACM) Transactions on Programming Languages and Systems*, Vol.9-2, pp.235-256, 1987.

[22] Y. Feldman, N. Derschowitz, Z. Hanna, Parallel Multithreaded Satisfiability Solver: Design and Implementation. *PDMC 2004*, 2004.

[23] S. Forman, A. Segre, NAGSAT: a randomized complet, parallel solver for 3-SAT, In*Proc. SAT'2002*, pp.236-243, 2002.

[24] R. Génisson, P. Jégou, Davis-Putnam were already checking forward, In *Proc. of ECAI'96*, pp.180-184, 1996.

[25] I. Gent, H, van Maaren, T. Walsh eds., *SAT 2000, Highlights of Satisfiability Research in the Year 2000*, Frontiers in AI and Applications, Vol. 63, Kluwer AC. Publ. ,2000.

[26] E. Goldberg, Y. Novikov, BerkMin: a fast and robust SAT-solver, In *Design, Automation and Test in Europe (DATE'02)*, pp.142-149, 2002

[27] J. Gu, Parallel Algorithms for Satisfiability problem, In *DIMACS Series in Dicrete Maths. and TCS.*, AMS, Vol.22, pp.105-161, 1995.

[28] J. Gu, P.W. Purdom, J. Franco, B.W. Wah, Algorithms for Satisfiability (SAT) problem: a Survey, In *DIMACS Series in Dicrete Maths. and TCS.*, AMS, Vol.35, pp. 19-152 1996.

[29] Z. Habbas, M. Krajecki, D. Singer, Decomposition Techniques for Parallel Resolution of Constraint Satisfaction Problems in Shared Memory: a Comparative Study. Special issue of ICPP-HPSECA01, Int. Jour. of Computational Science and Engineering (IJCSE), to be published, 2001.

[30] Z. Habbas, M. Krajecki, D. Singer, Parallel resolution of CSP with OpenMP. In *Proc. of the second European Workshop on OpenMP, EWOMP'00*, Edinburgh, Scotland, pp.1-8, 2000.

[31] Z. Habbas, M. Krajecki, D. Singer, Shared memory implementation of CSP resolution. In *Proc. of HLPP'2001*, Orléans, France, 2001.

[32] M. Heule, H. van Maaren, Aligning CNF and equivalence reasoning, In *Proc. of SAT'2004*, LNCS series, to appear 2005.

[33] T. Hogg, C.P. Williams, Expected gains from parallelizing constraint solving for hard problems, In *Proc. of AAAI'94*, pp.331-336, 1994.

[34] K. Iwama, D. Kawai, S. Miyazaki, Y. Okabe, J. Umemoto, Parallelizing local search for cnf satisfiability using vectorization and pvm, *ACM Journal of Experimental Algorithms*, 7 (2), 2002.

[35] B. Jurkowiak, C.M. Li, G. Utard, Parallelizing Satz Using Dynamic Workload Balancing. In *Proc. of SAT'2001*[41], 2001.

[36] B. Jurkowiak, Programmation Haute Performance pour la Résolution des problèmes SAT et CSP, *Thèse de l'Université de Picardie*, Amiens, 2004.

[37] P. Jackson, D. Sheridan, The optimality of a fast CNF conversion and its use with SAT, In *Proc. of SAT'2004*, LNCS series,to appear, 2005.

[38] H. Kautz, SATPLAN04: Planning as Satisfiability,
http: //www.cs.washington.edu/homes/kautz/satplan/.

[39] O. Kullmann, Investigations on autark assignments, *Discrete Applied Mathematics*, 107:99-137, 2000.

[40] D. Kokotov, PSolver: Distributed SAT Solver Framework,
http: //sdg.lcs.mit.edu/satsolvers/PSolver/index.html.

[41] H. Kautz, B. Selman: *Proc. of the Workshop on Theory and Applications of Satisfiability Testing, (SAT'2001)*, Elsevier Science Publishers, Electronic Notes in Discrete Mathematics Vol.9. 2001.
http://www.elsevier.nl/gej-ng/31/29/24/42/show/Products/notes/cover.htt

[42] R.M. Karp, Y. Zhang, Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation, *Journal of the ACM.*, Vol.40, No.3, pp.765-789, 1993.

[43] C. M. Li, Anbulagan, Heuristics based on unit propagation for satisfiability problems. In *15th Int. Joint Conference on AI, IJCAI'97*, Morgan Kaufmann Pub., Nagoya Japon, pp.366-371, 1997.

[44] D. Leberre, Exploiting the real power of unit propagation lookahed, In *Proc. of SAT'2001*[41], 2001.

[45] D. Leberre, L. Simon, Fifty-five solvers in Vancouver: the SAT'2004 competition, In *Proc. of SAT'2004*, LNCS series, to appear 2005.

[46] C. M. Li, S. Gérard, On the Limit of Branching Rules for Hard Random Unsatisfiable 3-SAT. In *14th. European Conf. on AI, ECAI 2000*, Berlin, 2000.

[47] C. M. Li, Integrating Equivalency reasoning into Davis-Putnam procedure. In *the proceedings of AAAI-2000*, Austin Texas USA, pp.291-296, 2000.

[48] F. Massacci, L. Marraro, Logical cryptanalysis as a SAT-problem: Encoding and analysis of the U.S. Data Encryption Standard, *Journal of Automated Reasoning*, Vol.24(1-2), pp.165-203, 2000, (http://www.dis.uniroma1.it/ massacci/papers/).

[49] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an Efficient SAT Solver, *Proc. of the 39th. DAC*, Las Vegas USA, 2001.

[50] B. Monien, E. Speckenmeyer, Solving Satisfiability in less than 2^n steps, *Discrete Applied Maths.*, Vol.10, pp.287-295, 1985.

[51] D. McAllester, B. Selman, A. Kautz, Evidence for invariant in local search, In *Proc. of 14th. Nat. Conf. on AI, AAAI 1997*, pp.321-326, MIT Press, 1997.

[52] D. Mitchell, B. Selman, H. Leveque, A new method for solving hard satisfiability problems, In *Proc. of 10th. Nat. Conf. on AI, AAAI 1992*, pp.440-446, MIT Press, 1992.

[53] J.P. Marques-Silva, K.A. Sakallah, GRASP- A New Search Algorithm for Satisfiability, *Proc. of ICCAD'96*, pp.220-227, 1996.

[54] J.P. Marques-Silva, K.A. Sakallah, Conflict Analysis in Search Algorithms for Propositional Satisfiability, *Proc. of the IEEE. ICTAI* , 1996.

[55] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, L. Troyansky, Determining computational complexity from characteristic "phase transitions", *Nature*, 400, pp.133-137, 1999.

[56] F. Okushi, Parallel Cooperative Propositional Theorem Proving, In *5th. Int. Symp. on AI and Maths.* AIMSA, Fort Lauderdale, USA, 1998.

[57] OpenMP Architecture Review Board, *OpenMP C and C++ Application Program Interface*, http://www.openmp.org.

[58] M.Platzner, G. De Micheli, Acceleration of Satifiability Algorithms by Reconfigurable Hardware, In *Proc. of the 8th. Int. Workshop FPL'98*, LNCS-1482, pp.69–78, 1998.

[59] P.M. Pardalos, L.S. Pitsoulis, M.G.C. Resende, A parallel GRASP for MAX-SAT problems, In *Proc.PARA'96*, LNCS 1180, pp.575-585, 1996.

[60] V.N. Rao, V. Kumar,, On the Efficiency of Parallel Backtracking, In *IEEE Trans.on Parallel and Distributed Systems*,Vol.4, No.4, 1993.

[61] A. Roli, Criticality and Parallelism in GSAT, In *Proc. of SAT'2001*,[41], 2001.

[62] SATLIB - The Satisfiability Library,
http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/.

[63] SATLive: http://www.satlive.org.

[64] E. Speckenmeyer, M. Böhm, P. Heusch, On the imbalance of distributions of solutions of CNF-formulas and its impact on satisfiability solvers, In *DIMACS Series in Dicrete Maths. and TCS.*, AMS, Vol.35, pp.669-676, 1996.

[65] I. Skliarova, A.B. Ferrari, Reconfigurable hardware SAT solvers: a survey of systems, In *Field-Programmable Logic and Applications*, P. Cheung, G. Constantinides, J. de Sousa eds., LNCS 2778, pp. 468-477, 2003.

[66] T. Stützle, H. Hoos, A. Roli, A review of the literature on local search algorithms for MAX-SAT, TR. AIDA.01.02, Darmstadt University of Technology, 2001.

[67] E. Speckenmeyer, B. Monien, O. Vornberger, Superlinear speedup for Parallel Backtracking, *Proc. SUPERCOMPUTING 87*, LNCS 297, pp. 985-995, Springer, 1987.

[68] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra, MPI: the complete reference, the MIT Press, 1996.

[69] T. Suyama, M. Yokoo, H. Sawada, Solving Satifiability Problems Using Logic Synthesis and Reconfigurable Hardware, *Proc.of the 31th. Hawaian Internationnal Conference on System Sciences HICSS-31*, 1998.

[70] C. Thiffault, F. Bacchus, T. Walsh, Solving non-clausal formulas with DPLL search, *Proc. SAT'2004*, LNCS series, to appear, 2005.

[71] T. Walsh, SAT versus CSP, *Proc. CP'2000*, LNCS 1894, pp. 441-456, Springer, 2000.

[72] J.P. Warners, H. van Maaren, A two phase algorithm for solving a class of hard satisfiability problems, *Operation Research Letter*, 23 (3-5), pp.81-88, 1998.

[73] L.C. Wu, C.Y. Tang,, Solving the satisfiability problem by using randomized approach, *Information Processing Letters*, 41, 187-190, 1992.

[74] H. Zang, M.P. Bonacina, Cumulating search in a distributed computing environment: a case study in parallel satisfiability, In *Proc. PASCO'94*, 1994.

[75] H. Zang, M.P. Bonacina, J. Hsiang, PSATO: a distributed propositional prover and its applications to Quasigroup problems, *Journal of Symbolic Computation*, Vol.21, pp.543-560, 1996.

[76] H. Zang, SATO: An Efficient Propositional Prover, *Proc. of Int. Conf. on Automated Deduction, CADE-97*, 1997.

[77] L. Zang, S. Malik Cache Performance of SAT Solvers: a Case Study for Efficient Implementation of Algorithms, *Proc. of SAT'2003*, 2003.

[78] L. Zhang, C. Madigan, M. Moskewicz, S. Malik, Efficient Conflict Driven Learning in a Boolean Satisfiability Solver, *Proc. of ICCAD 2001*, San Jose USA, 2001.

[79] Y. Zhang, A.Ortynski, Efficiency of Randomized Parrallel Backtrack Search, *Algorithmica*,Vol. 24, pp.14-28, 1999.