A Parallelization Scheme Based on Work Stealing for a class of SAT solvers

Bernard Jurkowiak, Chu Min Li and Gil Utard
LaRIA, Université de Picardie Jules Verne,
33 rue Saint Leu, 80039 Amiens, FRANCE
e-mail: {jurkowiak, cli, utard}@laria.u-picardie.fr

Abstract. Due to the inherent NP-completeness of SAT, many SAT problems currently cannot be solved in a reasonable time. Usually, to tackle a new class of SAT problems, new ad-hoc algorithm must be designed. Another way to solve a new problem is to use a generic solver and employ parallelism to reduce the solve time.

In this paper we propose a parallelization scheme for a class of SAT solvers based on the DPLL procedure. The scheme uses dynamic load balancing mechanism based on the work stealing techniques to deal with the irregularity of SAT problems. We parallelize Satz, one of the best generic SAT solvers, with the scheme to obtain a parallel solver called PSatz. The first experimental results on random 3-SAT and a set of well-known structured problems show the efficiency of PSatz. PSatz is freely available and runs on any networked workstations under Unix/Linux.

Keywords: automated theorem proving, SAT problem, parallelism

1. Introduction

Consider a propositional formula $\mathcal{F}$ in *Conjunctive Normal Form (CNF)* on a set of n boolean variables $\{x_1, x_2, \cdots, x_n\}$. A literal l is a variable x_i or its negation $\bar{x}_i$, a clause c is a logical *or* of some literals (e.g. $(x_1 \vee \bar{x}_2 \vee x_3)$) and a formula $\mathcal{F}$ is a logical *and* of clauses. A k-SAT problem is a problem where there are k literals per clause. The *satisfiability (SAT) problem* consists in finding an assignment of truth value (1 for $true$ and 0 for $false$) to variables such that $\mathcal{F}$ is true. If such an assignment exists, then $\mathcal{F}$ is said *satisfiable*, otherwise, $\mathcal{F}$ is said *unsatisfiable*.

SAT is a specific kind of finite-domain *Constraint Satisfaction Problem (CSP)* and it is the first known NP*-complete problem* [4] (3-SAT is its smallest NP-complete sub-problem).

Many problems in various domains can naturally be encoded into SAT then solved by a SAT solver. Encoding a real-world problem into SAT is generally easier than writing a specialized algorithm for the problem. Moreover, some problems can be solved more efficiently by a SAT solver (after encoding) than by the specialized algorithm. For example, by using propositional reasoning, many problems like quasi-group completion problems [33, 35], planning [22, 21] and hardware

© 2005 *Kluwer Academic Publishers. Printed in the Netherlands.*

modeling [10, 32] have already been solved with SAT solvers better than other methods.

As is pointed out by Selman, Kautz and McAllester in [30], several other factors also contributed to the growing interest in SAT. First, new algorithms were developed such as WalkSAT [29], UnitWalk [15], SP [3], Sato [34], Satz [25], Chaff [28] and Kcnfs [9]. Second, continuous progress in computer performance extends the range of the algorithms. Third, better encodings in SAT are found for interesting real-world problems.

However, because of the *NP-completeness* of SAT, we are in constant need of more computing power to solve a significant range of SAT problems in reasonable time. Moreover, despite of the power of state-of-the-art SAT solvers, the proof of the satisfiability or the unsatisfiability of many interesting problems remains challenging (see e.g. [33]). On the other hand, single computers such as PC become so cheap that it is easy to find dozens of networked machines, providing considerable computing resources.

Thus, several distributed/parallel systematic SAT solvers have been developed such as Böhm and Speckenmeyer Parallel Solver [2], PSATO [35], PaSAT [31] and PSolver [16] to exploit the power of networked computers. In all these approaches, a formula is dynamically divided into subformulas which are then distributed among processors. Relatively small subformulas are sequentially solved by a processor. When a processor begins to sequentially solve a subformula, it does not give part of load of this solving to another processor.

The approaches work well in case subformulas of the same size require roughly the same amount of time to be solved. Unfortunately, because of its NP-completeness, SAT is very irregular, because a subformula may require much more time to be solved than another subformula of the same size and this fact is currently unpredictable, especially when solving real-world problems. If the load balancing is not preemptive and processors busy in (sequentially) solving hard subformulas cannot efficiently give part of their load to idle processors, idle processors cannot help busy ones and will remain idle after solving all easy subformula.

In order to avoid the processor idleness, we propose in this paper a new approach to parallelize a class of SAT solvers. The load balancing is preemptive and idle processors steal parts of load of busy processors for load balancing. The approach is applied to parallelize Satz, one of the best generic solvers, to obtain an efficient systematic parallel SAT solver called *PSatz* (previously called //Satz in [20]).

The paper is organized as follows. We first present preliminaries about the general arborescent search parallelization and SAT problems

with their search space in Section 2. Then we present Satz and some other DPLL procedures in section 3. In Section 4, we present our dynamic load balancing scheme based on work stealing and its application to parallelize Satz. In Section 5, we show the performances of PSatz for random 3-SAT problems and a set of well-known structured SAT problems. Finally we discuss related work and compare PSatz with other parallel solvers in Section 6 before concluding in Section 7.

2. Preliminaries

Our approach belongs to a large field of parallelization of arborescent search. In this section, we present some preliminaries in the field related to SAT problems and their search space.

2.1. Parallelization of an arborescent search

Many finite search spaces can be structured in form of a tree, of which each node represents a sub-space. An arborescent search consists in visiting every node of the tree in a specific order. For simplicity, we only consider here depth first search in a finite binary tree, which is recursively defined by

- visit the root
- search in the left subtree
- search in the right subtree

The tree may be known before the search or should dynamically be constructed as the search proceeds.

Parallelization of an arborescent search consists in dividing the whole task among p available processors, so that each processor carries out roughly $1/p$ of the whole load. Ideally, the computation is p times faster. However, parallelization introduces some overhead (communication, synchronization), so the speedup is generally smaller. Sometimes, we search for a single solution in the search space and the solution is in a subtree directly searched by a processor. In this case the speedup may be larger than p. A speedup larger than p is said *super-linear*.

When all subtrees of a node are independent and roughly have the same size, it is straightforward to parallelize the depth-first search, since search(left-subtree) and search(right-subtree) can be simultaneously executed on two processors and each of the two searches can be in turn divided. The division continues until all processors are loaded. If

the subtrees have different size but the size of each subtree is known or predictable, only larger subtrees are divided so that the load of each processor is balanced.

When the subtrees of a node are of very different size but their size cannot be predicted, we say the corresponding search problem is irregular. Perfect load balancing cannot be obtained by a static partition of the tree among processors. In this case, a usual approach is to introduce a dynamic load balancing scheme where the subtrees are redistributed during the computation according to the load of different processors.

A dynamic load balancing scheme should allow to achieve two goals: the communication overhead between processors is as small as possible and the load of processors is as balanced as possible. In other words, the overhead of redistribution should not be greater than the load balancing gain: redistributing too often may lead to a deterioration of the efficiency.

When designing such a scheme, one essentially deals with the following questions:

- When a redistribution must be fired?
- Which work must be redistributed?
- Which processors are elected to receive additional work?

These decisions are usually based on an estimation of the workload and the cost of redistribution. Firing a redistribution may be due to an overload of a processor which then sends part of this work to a less loaded processor. This technique is called Sender Initiative Decision (SID). Another technique is the Receiver Inititiative Decision (RID) where a less loaded processor asks task to a more loaded one. For heavy computation, it was observed that RID strategy is usually better than an SID one because fewer requests are initiated [36].

The search space of SAT problems can be naturally structured by a binary tree, the two subtrees of a node are separated by the truth value 1 and 0 (See subsection 2.4 below). However because of its NP-completeness, SAT is very irregular. We propose a RID approach in this paper to parallelize a class of SAT solvers.

2.2. Depth cutoff and Ping-Pong phenomenon

When parallelizing an arborescent search (such as a depth-first search), a depth cutoff [14] is generally used to restrict the number of messages exchanged between the processors, i.e., a processor does not give out a subtree at a depth greater than a fixed threshold. Search after this

threshold is then sequential. Adjusting the cutoff value is a challenging task, since a small cutoff value limits the number of communications but may increase processor idleness, and vice versa.

Adjusting the cutoff value for an irregular search problem such as SAT is still harder, because one cannot predict the size of subtrees after the cutoff threshold. Since a sequential search after the cutoff threshold may be very long, other processors may become idle after finishing all other tasks but cannot help the busy processor.

Standard depth cutoff is based on the number of workload partitions already done (i.e. partitions before the current node is reached) to decide whether the processor should divide the current subtree. A variant of depth cutoff technique uses an estimator to estimate the workload of the current subtree, which is divided only if the estimated workload is larger than a threshold.

Another difficulty when dealing with an irregular search problem is the so-called "Ping-Pong phenomenon": the processors spend more time in communication than in the search itself. In the extreme case, a task makes a round trip between two (or more) processors over and over as a ping-pong ball, processors spend no or little time on the search itself.

2.3. SAT problems

SAT problems can be divided into two classes: random problems and structured problems.

The most interesting random problems for us are random k-SAT. A random k-SAT problem of n variables and m clauses is generated as follows. Each clause is created by randomly picking k variables and negating each of them with probability 0.5. The case $k = 3$ is well studied. It is found in [27] that when m/n is near 4.25, the 3-SAT problems are the most difficult to solve and are satisfiable with probability 0.5. Random k-SAT problems are very suitable for studying the properties of SAT and for evaluating SAT solvers. We use hard random 3-SAT problems with m/n=4.25 in our experimentation.

While there is no intended relation between variables (and clauses) in a random problem, there may be different intended relations between variables (and clauses) in a structured problem. Examples of these relations include symmetries (two variables are symmetric if their permutation in the formula gives the same formula) and equivalences (two literals are equivalent if their value is equal in all satisfying assignments). Structured problems may come from the encoding of a real-world problem into SAT or may be created by hand according to some properties.

For example, in order to encode into SAT the famous unsatisfiable pigeon hole problem of placing $n+1$ pigeons in n holes so that every hole exactly contains one pigeon [5], we number the pigeons from 1 to $n+1$ and holes from 1 to n and define $(n+1) \times n$ boolean variables as follows:

$$x_{ij} = \begin{cases} 1 \text{ if pigeon } i \text{ is placed in hole } j \\ 0 \text{ otherwise} \end{cases}$$

We define a group of $(n^2+n+2)/2$ clauses for each hole j ($1 \leq j \leq n$). The first clause of the group is:

$$x_{1j} \vee x_{2j} \vee ... \vee x_{nj} \vee x_{n+1j}$$

saying that hole j contains at least a pigeon. Then for every couple of pigeons i_1 and i_2 ($1 \leq i_1 < i_2 \leq n+1$), a clause $\bar{x}_{i_1 j} \vee \bar{x}_{i_2 j}$ is defined to say that these two pigeons cannot be both in hole j. The intended meaning of this group of clauses is that there is one and only one pigeon in hole j. We recall that the clauses in a random problem don't have any intended meaning.

The total number of clauses for this problem is $(n^3+n^2+2n)/2$ since there are n holes.

The boolean variables vector $x_{1j}x_{2j}...x_{n+1j}$ is symmetric to the boolean variables of $x_{1k}x_{2k}...x_{n+1k}$ for $j \neq k$, since if we simultaneously permute $x_{1j}x_{2j}...x_{n+1j}$ with $x_{1k}x_{2k}...x_{n+1k}$ in the formula, we obtain the same formula.

2.4. Search space of a SAT problem

There are 2^n truth assignments in the search space for a SAT problem of n variables. Given any variable x, the search space can be divided into two sub-spaces. One sub-space contains all assignments in which x=1, the other contains all assignments in which x=0. Each of the two sub-spaces can be in turn divided using another variable. So the whole search space can be structured in a binary tree, of which each leaf represents a disjoint sub-space specified by the partial assignment from the root to the leaf.

For example, the binary tree in Figure 2 in section 4.1 structures a search space into 5 sub-spaces respectively represented by 5 partial assignments (from left to right):

1. $x_1 = 1$,
2. $x_1 = 0, x_3 = 1$,
3. $x_1 = 0, x_3 = 0, x_4 = 1, x_5 = 1$,

4. $x_1 = 0, x_3 = 0, x_4 = 1, x_5 = 0$,

5. $x_1 = 0, x_3 = 0, x_4 = 0$

Each sub-space can be in turn structured by a (sub) binary tree.

The DPLL procedure presented in the next section dynamically structures the search space in a binary tree to solve SAT problems.

3. Satz and some other DPLL-based SAT solvers

SAT solvers can be divided into two classes:

- incomplete solvers that try to find a consistent assignment to a fixed problem but are currently not designed to prove unsatisfiability;

- complete solvers that are able to find a consistent assignment and to prove unsatisfiability.

Most complete SAT solvers are based on the DPLL algorithm developed by Davis, Putnam, Logemann and Loveland [7, 6]. In this section after a description of the DPLL procedure, we present some efficient complete DPLL-based solvers and justify our choice of parallelizing Satz.

3.1. The DPLL procedure

The *DPLL* procedure [7, 6] provides the basis of most complete algorithms for SAT. It is sketched in Algorithm 1.

In order to solve a SAT problem, the *DPLL* procedure dynamically structures a search space of all possible truth assignments into a binary search tree until it either finds a satisfying truth assignment or concludes that no such assignment exists. The recursive call in line 10 of Algorithm 1 represents the left subtree ($x = 1$) in the binary search tree and the recursive call in line 13 represents the right subtree ($x = 0$). x is called *branching variable.* A leaf corresponds to an empty clause (i.e. a dead-end) or to a satisfying assignment.

The DPLL algorithm is mainly based on the *UnitPropagation* procedure (see lines 16-21), illustrated in the following example. Let $\mathcal{F}$ be the following propositional formula:

$$\mathcal{F} = (\bar{x}_1) \wedge (x_1 \vee x_2) \wedge (x_1 \vee x_3 \vee x_4) \wedge (\bar{x}_2 \vee \bar{x}_5 \vee x_6) \wedge (\bar{x}_1 \vee x_6)$$

The first clause is unit since it contains only one literal. The *UnitPropagation* procedure satisfies it by assigning $x_1 = 0$, then it removes

all clauses containing $\bar{x}_1$ (which are satisfied) and reduces the clauses containing x_1 by removing x_1 from them. The second clause becoming unit, the unit propagation continues by satisfying this new unit clause and reducing the fourth clause. Finally, $\mathcal{F}$ becomes:

$$\mathcal{F} = (x_3 \vee x_4) \wedge (\bar{x}_5 \vee x_6)$$

Dividing a formula into two subformulas means that for some x, we get two formulas $\mathcal{F}_1{=}UnitPropagation(\mathcal{F} \wedge (x))$ and $\mathcal{F}_2{=}UnitPropagation(\mathcal{F} \wedge (\bar{x}))$.

Algorithm 1 DPLL procedure

1: **DPLL**($\mathcal{F}$)
2: **if** $\mathcal{F} = \emptyset$ **then**
3:    **return** "satisfiable";
4: **end if**
5: $\mathcal{F} \leftarrow$ **UnitPropagation(**$\mathcal{F}$**)**;
6: **if** $\mathcal{F}$ contains an empty clause **then**
7:    **return** "unsatisfiable";
8: **end if**
9: **choose** x in $\mathcal{F}$ according to a heuristic H;
10: **if DPLL**($\mathcal{F} \wedge (x)$) = "satisfiable" **then**
11:    **return** "satisfiable";
12: **else**
13:    **return DPLL**($\mathcal{F} \wedge (\bar{x})$);
14: **end if**
15:
16: **UnitPropagation**($\mathcal{F}$)
17: **while** there is no empty clause and a unit clause with a literal l exists in $\mathcal{F}$ **do**
18:    **satisfy** clauses containing l (remove them from $\mathcal{F}$);
19:    **reduce** clauses containing $\bar{l}$ (remove $\bar{l}$ from them);
20: **end while**
21: **return** $\mathcal{F}$;

3.2. Satz

Satz is a DPLL procedure using a heuristic based on unit propagation to select the next branching variable [26]. Given a variable x, the Unit Propagation heuristic examines x by adding, respectively, the unit clauses x and $\bar{x}$ to $\mathcal{F}$ and making independently two unit propagations. Let $w(\bar{x})$ (resp. $w(x)$) be the number of clauses reduced by unit propagation when $\bar{x}$ (resp. x) is added into $\mathcal{F}$. Then the following equation

suggested by Freeman [11] is used to give the weight of the variable x:

$$H(x) = w(\bar{x}) * w(x) * K + w(\bar{x}) + w(x)$$

where K is a constant fixed experimentally.

Satz branches on x such that $H(x)$ is the largest. Intuitively, the larger $w(x)$ is, the more quickly DPLL($\mathcal{F} \wedge (x)$) may probably encounter a contradiction. In this case DPLL($\mathcal{F} \wedge (x)$) produces a smaller tree. The equation $H(x)$ favors those variables x such that $w(\bar{x})$ and $w(x)$ do not differ much. A consequence is that Satz constructs a very balancing binary search tree for random problems [26].

Satz also includes the so-called experimental unit propagation of the second level [24] to eliminate small subtrees. The technique is based on the following observation. If the satisfaction of a literal l introduces many strong constraints (e.g. binary clauses) by unit propagation, it probably leads to an imminent dead-end which can be detected by further experimental unit propagations.

Recall that Satz examines a variable x by calling $UnitPropagation(\mathcal{F} \wedge (\bar{x}))$ and $UnitPropagation(\mathcal{F} \wedge (x))$. Let l be x or $\bar{x}$. If $UnitPropagation(\mathcal{F} \wedge (l))$ reduces more than T clauses (T is empirically fixed), then for every variable y in the newly produced binary clauses occurring both positively and negatively in binary clauses, Satz executes $UnitPropagation(\mathcal{F} \wedge (l) \wedge (\bar{y}))$ and $UnitPropagation(\mathcal{F} \wedge (l) \wedge (y))$. If both propagations reach a dead-end, $\bar{l}$ should be satisfied.

We illustrate the experimental unit propagations of the second level by Figure 1.

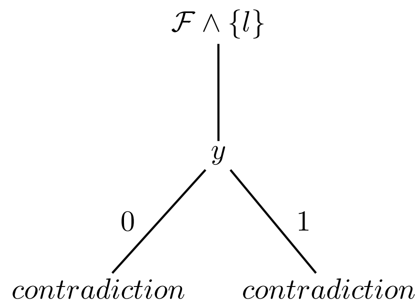


Figure 1. Experimental unit propagation of the second level.

Another technique in Satz is the preprocessing of the input formula by adding some resolvents of length ≤ 3 [25, 12]. For example, if $\mathcal{F}$ contains the following two clauses:

$$(x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_4)$$

then the clause $(x_2 \vee x_3 \vee \bar{x}_4)$ is added to $\mathcal{F}$. The new clauses can in turn be used to produce other resolvents. This process is performed until saturation. This preprocessing adds constraints in the input formula to exhibit contradiction more easily.

Many improved versions of Satz have been developed (e.g. [23]). In this paper, we parallelize Satz215 which includes the implied literal propagation mechanism suggested by D. Le Berre [1]. Precisely, if both $\mathcal{F} \wedge (x)$ and $\mathcal{F} \wedge (\bar{x})$ implies a literal l, then l is satisfied and propagated in $\mathcal{F}$. This enables Satz to solve some hard real world problems [1, 18].

Thanks to these techniques, Satz is a generic SAT solver very efficient for hard random problems and many hard structured problems (see e.g. [1, 18, 26]). The source code of Satz is freely available [17].

3.3. Some other SAT solvers

There are many other efficient SAT solvers based on the DPLL procedure. Among the most representative SAT solvers, we find Sato [34], Chaff [28] and Kcnfs [8].

3.3.1. *Sato and Chaff*

Sato has been first developed to solve quasigroup problems. Then it has been modified to solve structured problems such as DIMACS problems. Chaff is also developed to solve structured problems. One common feature of Sato and Chaff is the use of the so-called look-back search techniques.

Roughly speaking, when a DPLL-based SAT solver encounters a dead-end in the left subtree, the look-back techniques analyze the reason of the dead-end. One or more clauses are added into the formula to avoid the same dead-end in the right subtree.

For example, suppose that the solver encounters an empty clause which is originally $x_1 \vee \bar{x}_2 \vee x_3$ (the clause becomes empty because x_1, x_2 and x_3 are respectively assigned 0, 1 and 0) and that x_1 is the last assigned variable. If x_1 is assigned 0 because $\bar{x}_1$ was in a unit clause which is originally $\bar{x}_1 \vee x_4 \vee x_5$ (x_4 and x_5 were assigned 0 before), the falsification of x_4 and x_5 is the reason of x_1=0. We replace x_1 by $x_4 \vee x_5$ and obtain a new empty clause $x_4 \vee x_5 \vee \bar{x}_2 \vee x_3$. If the last assigned variable in this new empty clause was in a unit clause, it is replaced by its reason to obtain another empty clause. The process continues until the last assigned variable is a branching variable. The DPLL procedure branches on the other value of the branching variable.

Note that the empty clauses successively obtained become non empty after backtracking. Look-back techniques add some of them into the formula to prevent the solver from reaching the same dead-end in

the right subtree, since these clauses would become empty before the clause $x_1 \vee \bar{x}_2 \vee x_3$ in the right subtree.

Obviously, the left and right subtrees are not independent for solvers employing look-back techniques, which makes it harder to parallelize them.

3.3.2. *Kcnfs*

Kcnfs is specially developed to solve random k-SAT problems. It uses a powerful heuristic exploiting the random properties of k-SAT to choose the next branching variable. It is very efficient for k-SAT problems. No look-back techniques are used in Kcnfs.

3.4. Comparison of Satz, Kcnfs, Sato and Chaff

We compare the performance of Satz, Kcnfs, Sato and Chaff for random 3-SAT problems and several representative structured SAT problems on a PC with an Athlon XP2000+ CPU and 256Mb of RAM under Linux.

Table I shows the performance of the 4 solvers for hard random 3-SAT problems with 200 and 300 variables. In the table, #V is the number of variables, #P is the number of problems for each #V, #S is the number of problems solved by each solver among #P problems within 7200 seconds, *Time* is the average run time in seconds over #S of each solver to solve a problem. Satz has comparable performance with Kcnfs but is substantially better than Sato and Chaff.

Table I. *Run time in seconds for random problems.*

#V	#P	*Satz215*		*Kcnfs*		*Sato321*		*Chaff*	
		#S	*Time*	#S	*Time*	#S	*Time*	#S	*Time*
200	100	100	0.089s	100	0.064s	100	2.80s	100	6.55s
300	100	100	2.25s	100	1.73s	75	1868.27s	77	224.93s

Tables II and III compare the performance of the four solvers for planning problems, Beijing challenge suite and Bounded Model Checking, all available in SATLIB [19]. All times are in seconds. Kcnfs is not able to tackle some formulas due to some internal limitations, which is indicated by "?"in the table. It is clear that Satz has comparable performance with Sato and Chaff to solve structured problems although without look-back techniques, and is significantly better than Kcnfs for these problems.

Table II. *Run time in seconds by using planning and Beijing problems.*

Problem Class	*Satz215*	*Kcnfs*	*Sato321*	*Chaff*
logistics.a.cnf *(sat)*	2.840s	51.08s	<1s	<1s
logistics.b.cnf *(sat)*	<1s	<1s	37.57s	<1s
logistics.c.cnf *(sat)*	<1s	>7200s	18.09s	<1s
logistics.d.cnf *(sat)*	<1s	>7200s	>7200s	<1s
2bitadd_10.cnf *(unsat)*	>7200s	>7200s	493.5s	219.4s

Table III. *Run time in seconds by using Bounded Model Checking problems.*

Problem Class	*Satz215*	*Kcnfs*	*Sato321*	*Chaff*
barrel5.dimacs *(unsat)*	24,99s	?	4,100s	<1s
barrel6.dimacs *(unsat)*	291,5s	?	37,81s	2,690s
barrel7.dimacs *(unsat)*	1,740s	?	160,9s	11,74s
barrel8.dimacs *(unsat)*	<1s	?	450,3s	32,56s
barrel9.dimacs *(unsat)*	1563s	?		<1s
longmult6.dimacs *(insat)*	15,07s	6,190s	39,51s	1,510s
longmult7.dimacs *(insat)*	73,39s	29,83s	83,31s	13,40s
longmult8.dimacs *(insat)*	233,1s	82,67s	105,1s	69,60s
longmult9.dimacs *(insat)*	462,1s	142,9s	271,8s	134,0s
longmult10.dimacs *(insat)*	743,5s	218,0s	273,9s	260,6s
longmult11.dimacs *(insat)*	1009s	280,2s	529,2s	355,8s
longmult12.dimacs *(insat)*	1181s	331,2s	499,9s	317,4s
longmult13.dimacs *(insat)*	1364s	375,7s	673,8s	265,2s
longmult14.dimacs *(insat)*	1552s	417,0s	644,5s	275,5s
longmult15.dimacs *(insat)*	1944s	472,9s	850,4s	215,0s

In summary, Satz is one of the best generic SAT solver. Note that many years of effort were needed to obtain Sato, Chaff, Kcnfs and Satz. The parallelization of a generic solver is generally easier and allows to obtain better results in many cases.

4. A parallelization scheme based on work stealing

We present a parallelization scheme based on work stealing and apply this scheme to parallelize Satz. The proposed approach can be

directly used to parallelize any DPLL-based solver without look-back techniques.

4.1. Representation of working context of the DPLL procedure

We recall that the DPLL procedure dynamically structures the whole search space into a binary search tree to solve a SAT problem. At any moment of the search, the DPLL procedure is at a tree node and is going to construct the two subtrees rooted at this node. We represent a node by a context in which we want to find three things:

- the part of the tree that has been constructed and explored,
- the current partial variable assignment,
- the part of the tree remaining to construct.

For this purpose we define the context by an ordered list of triplets:

$$(variable,\ current_value,\ remaining_value)$$

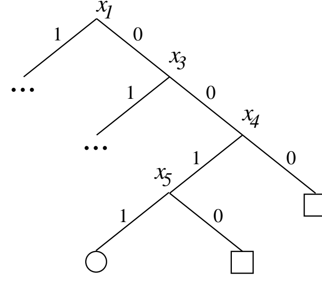
where:

$$current_value = \begin{cases} 1 & (true) \\ 0 & (false) \end{cases}$$

and:

$$remaining_value = \begin{cases} 1 & (true) \\ 0 & (false) \\ -1 & (no\ more\ remaining\ value\ to\ try) \end{cases}$$

For example, let T be a binary search tree represented by Figure 2.



... : Explored subtree

○ : Current node

□ : Remaining subtree root

Figure 2. A binary search tree T and its contexts.

T divides the whole search space in 5 disjoint subspaces (see section 2.4). The current node (or subspace) of T is represented by the following list of triplets:

$$\{(x_1, 0, -1), (x_3, 0, -1), (x_4, 1, 0), (x_5, 1, 0)\}.$$

From the context, we know that the DPLL procedure has successively explored the subspaces (or subtrees) respectively represented by the following partial assignments:

$$\{(x_1 = 1)\}$$
$$\{(x_1 = 0), (x_3 = 1)\}$$

since the remaining-value -1 in the context of the current node means that the truth value opposit to the current one has been tried out (unsucessfully) already. The subspaces (or subtrees) remaining to explore are respectively represented by the partial assignments:

$$x_1 = 0, x_3 = 0, x_4 = 1, x_5 = 0$$

and:

$$x_1 = 0, x_3 = 0, x_4 = 0$$

and correspond to the contexts (assume that the first subspace is explored before the second one):

$$\{(x_1, 0, -1), (x_3, 0, -1), (x_4, 1, 0), (x_5, 0, -1)\}$$
$$\{(x_1, 0, -1), (x_3, 0, -1), (x_4, 0, -1)\}$$

Since a context represents the current partial assignment, the variables fixed (i.e. assigned) by unit propagation are also included (in

order) with remaining-value -1. In a context, we do not distinguish a branching variable with remaining-value -1 and a variable fixed by unit propagation. For example, x_3 in the above context might be a variable fixed by unit propagation.

A natural way to parallelize the DPLL procedure is to exploit the inherent parallelism exposed by the tree. Indeed, each subtree can be explored independently. On a parallel computer with P processors, it is straightforward to divide the binary search tree in P subtrees and assigns each subtree to a processor. However, since the workloads of these subtrees are very different but are currently unpredictable, the hard issue is to design a dynamic load balancing so that the communication overhead is as small as possible and the load of each processor is as balanced as possible.

We present below our approach to parallelize Satz. The basic idea is that an idle processor steals one of the remaining subtrees of a busy processor.

4.2. General communication model

We choose the master/slave architecture for communication. As is illustrated in Figure 3, every processes (master or slave) has two threads: working thread and communicating thread. The communicating thread receives messages and puts them into its letter-box.

The working thread of a slave is a normal Satz procedure, but modified for communication. By a semaphore mechanism, it detects and reads its messages from the letter-box before every splitting. Note that all slaves use the same heuristic to select branching variable for splitting.

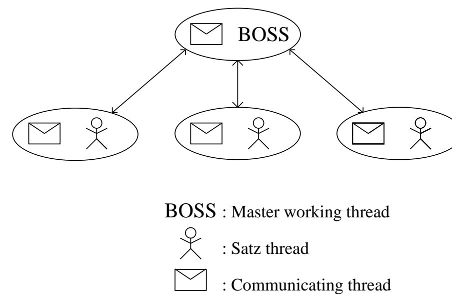


Figure 3. Master-Slave model.

4.3. Initialization

The master initiates the computation by launching all slaves. Each slave gets the same input formula and pre-processes it before being ready. Then the master choose an arbitrary ready processor and makes its working thread start the whole computation. At this stage, there is a busy slave and other ready processors are idle.

4.4. Dynamic load balancing

When one or several slaves are idle after the preprocessing or after finishing their subtree, they send a task request to the master. The master asks the load of each busy slave and removes the first remaining subtree of the most loaded slave and sends it to an idle slave. The load evaluation is carried out in the following manner.

Consider the context of a slave:

$$\{(x_1, cv_1, rv_1), (x_2, cv_2, rv_2), ..., (x_k, cv_k, rv_k)\}$$

where cv_k is the (current) value of x_k in the current partial assignment, rv_k is the remaining value of x_k to be tried, the position of the first remaining subtree is the smallest j $(1 \leq j \leq k)$ such that rv_j is different from -1. Intuitively, the larger j is, the more variables are fixed in the context:

$$\{(x_1, cv_1, -1), (x_2, cv_2, -1), ..., (x_{j-1}, cv_{j-1}, -1), (x_j, cv_j, rv_j)\}$$

In this case, the corresponding remaining subtree will probably be smaller. Figure 4 shows an example. The slave with the smallest position j of the first remaining subtree is said *the most loaded*. If there are at least two most loaded slaves, the positions of the second remaining subtrees of these slaves are compared. The remaining ties are broken arbitrarily.

The master begins load balancing by asking all busy slaves to send the positions of their first and second remaining subtrees. A special value is sent for the case where there are no remaining subtrees in the context and the case where there is only one remaining subtree. Then the master selects the most loaded slave and sends it a stealing message to get its first remaining tree.

After sending the positions of its first and second remaining subtrees, a slave continues its search. If slave i is selected and its first remaining subtree is at position j, it modifies its context into:

$$\{(x_1, cv_1, -1), ..., (x_j, cv_j, -1),\\ (x_{j+1}, cv_{j+1}, rv_{j+1}), ..., (x_k, cv_k, rv_k)\}$$

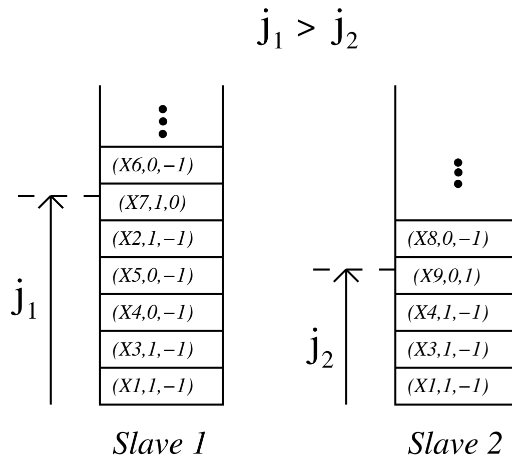


Figure 4. *Two slave contexts. Slave 2 is probably more loaded since* $j_2 < j_1$

and sends the context:

$$\{(x_1, cv_1, -1), (x_2, cv_2, -1), ..., (x_{j-1}, cv_{j-1}, -1), (x_j, rv_j, -1)\}$$

to the master before continuing its search from the modified context. The master sends the received context to an idle slave which then begins to work from the new context.

Since a slave continues its search after sending to the master the positions of its first and second subtrees for load evaluation, the search of the slave may enter the first remaining subtree. In this very rare case, either the slave has a different first remaining tree (because the old first remaining tree has been split) which will be sent to the master as in the usual case if it is selected to send a part of its load, or the slave may not have any remaining tree at all (because the slave has finished its work). In the last case, the slave sends a special message to the master for it to select another busy slave.

The load balancing is continued until there is no more idle slave.

Ping pong phenomenon

A possible ping pong phenomenon when parallelizing a DPLL procedure is illustrated in Figure 5.

In this Figure, (1) processor A sends the right subtree to processor B but quickly finds a contradiction in the left subtree and becomes idle; (2) processor B then sends its right subtree to processor A but quickly becomes idle for the same reason; ... The parallel procedure would spend more time in communication than in resolution, although the search indeed proceeds.

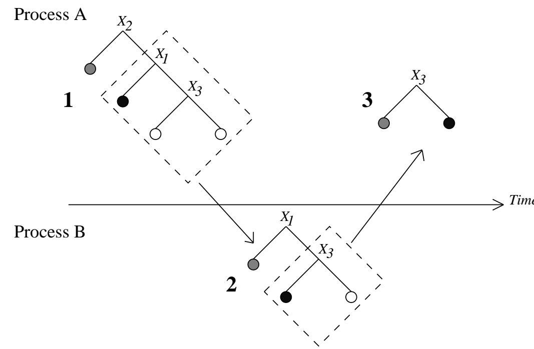


Figure 5. Ping pong phenomenon in load balancing

The ping pong phenomenon is very improbable in PSatz, thanks to the powerful branching rule of Satz [25], which examines every possible branching variable x by two experimental unit propagations followed by several experimental unit propagations of the second level as we saw in section 3.2.

For example, the left subtree 1 in Figure 5 means that for some y both $x_2 = 1 \wedge y = 1$ and $x_2 = 1 \wedge y = 0$ lead to a contradiction. However this contradiction would be detected by experimental unit propagations of the second level and x_2 would be directly assigned 0 without branching. So the subtrees shown in Figure 5 are very improbable in Satz and PSatz.

Cutoff
PSatz doesn't use a depth cutoff at all, which avoids the difficulty of adjusting a cutoff value. When a busy slave receives a work request, it always gives its first remaining tree whatever its context is. The communication overhead of PSatz in our experimentation is small even when it runs with 200 processors.

4.5. Termination

When a slave finds a solution (when the problem is satisfiable) or all slaves finish their subtrees (when the problem is unsatisfiable), the master collects all running information such as subtree sizes and tells all slaves to halt.

4.6. Fault tolerance

Because of the NP-completeness of SAT, PSatz may require a long time to solve some problems. During this time a processor may die

and/or the network may be interrupted. Fault tolerance means that the resolution should be continued if one or more slaves die, and that the resolution can be stopped and re-started from the interruption point.

In our approach, all slaves send their current context to the master to be saved every hour. After a load balancing, the master also saves the new context of the two concerned slaves. If a slave dies, the master sends its saved context to the idle slave at the time of load balancing so that the work of the died slave can be continued.

It seems that one hour is a good time interval in our experimentation, since it only produces negligible communication overhead and the death of a slave causes at most the loss of one hour computation.

Our fault tolerance mechanism is similar to that implemented in PSATO [35].

4.7. Implementation

We implemented PSatz in the C language using RPC (Remote Procedure Call) and Posix Threads, which are standard. Therefore, PSatz runs on any networked machines under Linux (Unix) without specific toolkit. PSatz is available at the following url:

`http://www.laria.u-picardie.fr/∼cli/englishpage.html`

5. Performance evaluation

We use random 3-SAT problems and a set of well-known structured real-world problems to evaluate the performances of PSatz. Our experimental results are obtained on a cluster of 216 computers (HP e-vectra nodes with Pentium III 733 MHZ processors and 256 Mb of RAM) under Linux (Linux kernel 2.2.15-4mdk or Linux kernel 2.4.2) and networked by Fast-Ethernet (100 Mbit/second) and 5 switches (HP procurve 4000 with 1 Gbits/second switch mesh) called Icluster[1].

All communication times are in seconds and represent the total time the master spends for sending messages to slaves (including work requests and contexts). The time for the master to send one message is measured from the point the master sends the message to a slave to the point the master receives the acknowledgment of receipt from the slave.

All run times are also in seconds, unless otherwise stated, and represent real times of the master starting from the reading of the input formula to the end of the resolution. This time includes the communication time of the master, load evaluation times and computation

[1] http://www-id.imag.fr/Grappes/icluster/description.html

time of the slave that terminates last for unsatisfiable problem or the computation time of the slave that finds a model first plus the time to halt all other slaves and the master for satisfiable problem. PSatz uses NFS (Network File System): at the beginning all processors read and pre-process simultaneously the input formula. Note that, Satz and PSatz develop the same tree for unsatisfiable problems because they use the same heuristic for branching.

5.1. Random 3-SAT problems

We show the performance of PSatz for hard random 3-SAT problems. We generate 100 problems with 400, 450, 500 variables, respectively, 25 problems with 550 variables and 20 problems with 600 variables, and separate results for satisfiable and unsatisfiable problems.

In the following Tables, $\#V$ is the number of variables, $\#P$ is the number of solved problems, T_1 is the average real time for sequential Satz, T_{128} is the average real time for PSatz on 128 processors, Sp is the Speedup T_1/T_{128}, $\#WB$ is the average number of load balancings (after the initialization) and T_{com} is the average total communication time spent by PSatz (note that T_{com} is included in T_{128}). Standard deviation is noted by *(std dev.)*.

Note also that, the average is taken over #P. For example, the average real time of Satz is $T_{total}/\#\mathrm{P}$ where T_{total} is the total real time for all the #P instances.

5.1.1. *Unsatisfiable random 3-SAT problems*

Tables IV and V show the performance of PSatz on unsatisfiable random 3-SAT problems on 128 processors for problems with 400, 450 and 500 variables.

The speedup of PSatz is smaller for easier problems, because the communication overhead for load balancing is relatively more important for them. For example, the average communication time of PSatz represents 18.21 % of the real time for problems with 400 variables, but the part of the communication is reduced to 2.18 % for problems with 500 variables. Despite the number of load balance operations is more important for larger problems, the communication time measured for these problems is negligible compared to the real time.

The reading and the pre-processing of the input formula by all slaves is not negligible compared to the search time for small problems, which is the second reason for the smaller speedup of PSatz for these problems.

Figure 6 shows the scalability of PSatz. Usually when the number of processors increases, the overhead of communications increases too. To keep a good efficiency, it is necessary to compensate this overhead

Table IV. *Run time of PSatz for unsatisfiable random 3-SAT problems.*

#V	#P	T_1 *(std dev.)*	T_{128} *(std dev.)*	Sp
400	49	256.24s *(111.0)*	12.08s *(3.39)*	21.21
450	41	1511.03s *(682.0)*	30.17s *(9.35)*	50.08
500	46	9784.83s *(4677)*	107.15s *(38.09)*	91.31

Table V. *Number of load balancings and communication time of PSatz.*

#V	#P	#WB *(std dev.)*	T_{com} *(std dev.)*
400	49	48.36 *(13.43)*	2.20s *(0.57)*
450	41	111.87 *(24.98)*	2.48s *(0.54)*
500	46	255.00 *(50.78)*	3.02s *(0.57)*

by increasing the size of the problem. Thus, a parallel algorithm is said “scalable” if the efficiency remains the same when the number of processors and the size of problems are increased.

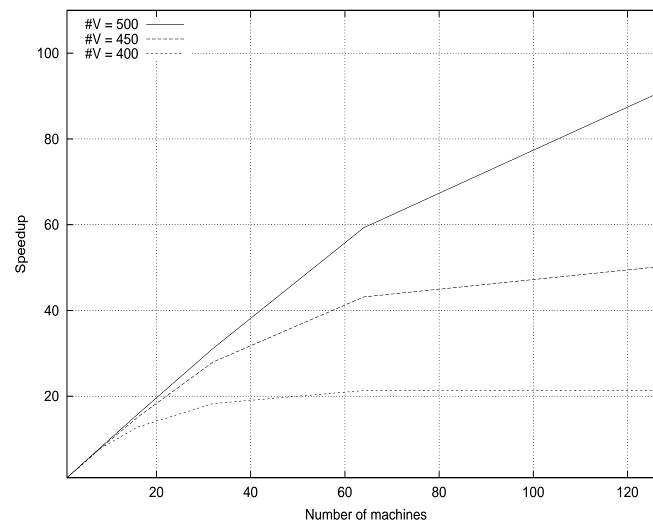


Figure 6. The average speedup of PSatz on 1, 2, 4, 8, 16, 32, 64 and 128 processors for unsatisfiable 3-SAT problems

We observe that the speedup of PSatz for a problem with 400 variables is always about 21 on more than 64 processors from Figure 6, the

communication overhead cancels the benefit of the increase in processor number for these easy problems. The phenomenon suggests a limit of the performance of the parallelization, to be studied in the future. However we observe in Figure 6 that PSatz is indeed scalable, since for a fixed number of processors, the efficiency of the parallelization increases with the size of problems.

5.1.2. *Satisfiable random 3-SAT problems*

Tables VI and VII show the performance of PSatz on satisfiable random 3-SAT problems. There is a large variation in speedup: for some problems the gain is super-linear. However we do not observe the expected super-linear gain in the average.

The performance of PSatz is comparable for satisfiable and unsatisfiable random 3-SAT problems and is scalable in both cases.

Table VI. *Run time of PSatz for satisfiable random 3-SAT problems.*

#V	#P	T_1 *(std dev.)*	T_{128} *(std dev.)*	Sp
400	51	84.06s *(73.03)*	4.11s *(1.87)*	20.45
450	59	483.35s *(496.4)*	9.79s *(7.59)*	49.37
500	54	2633.53s *(3181)*	24.62s *(31.12)*	106.96

Table VII. *Number of load balancings and communication time of PSatz.*

#V	#P	#WB *(std dev.)*	T_{com} *(std dev.)*
400	51	2.41 *(5.60)*	2.76s *(1.09)*
450	59	13.44 *(19.65)*	2.32s *(0.46)*
500	54	27.35 *(48.82)*	2.42s *(0.59)*

As for the unsatisfiable problems, the performance of PSatz is also stagnated for 400 variable random satisfiable 3-SAT problems on more than 64 processors, suggesting a limit of the parallelization. Another limit (if any) to be studied in the future is the number of the processors from which PSatz is slowed down. While the tree constructed by PSatz is the same as that by Satz for unsatisfiable problems, the solution found for a satisfiable problem depends on the number of processors. The solution found by PSatz on a number i of processors is not necessarily the same as the solution found on a number j of processors when $i \neq j$. The reason is that when PSatz runs on i (resp. j) processors,

i (resp. j) disjoint subspaces are simultaneously searched, we do not know how to predict which subspace contains a solution and when the solution is reached.

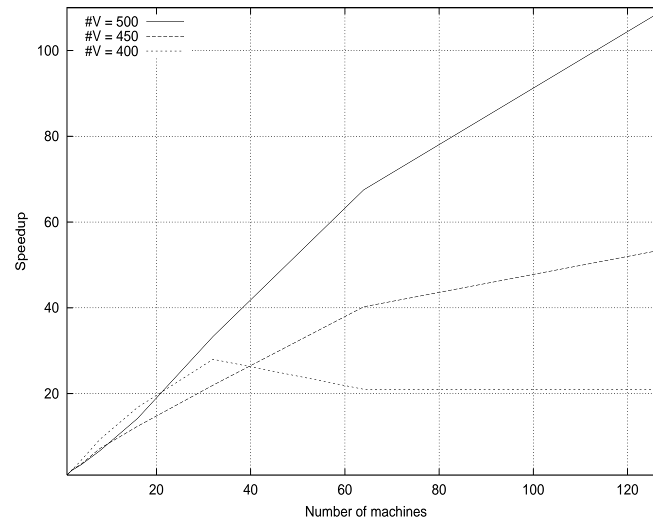


Figure 7. The average speedup of PSatz on 1, 2, 4, 8, 16, 32, 64 and 128 processors for satisfiable random 3-SAT problems

5.1.3. *Hard random 3-SAT problems with 550 and 600 variables*

We compare the performances of PSatz on 16 and 128 processors for problems with 550 variables (32 and 128 processors for problems with 600 variables). These problems are too hard to solve for Satz in sequential mode on Icluster, which is a shared cluster and where jobs running for over 6 hours are killed by default.

Table VIII and IX show the performance of PSatz for problems of 550 variables on 16 and 128 processors. T_{com16} (resp. T_{com128}) is the communication time of the master when PSatz runs on 16 (resp. 128) processors.

Table VIII. *Run time of PSatz for problems of 550 variables.*

	#V	#P	T_{16}	T_{128}	T_{16}/T_{128}	$\#WB_{128}$
Unsat	550	12	2743.50s	377.75s	7.26	411.75
Sat	550	13	839.76s	145.53s	5.77	41.53

Table IX. *Communication time of PSatz for problems of 550 variables.*

	#V	#P	T_{com16}	T_{com128}
Unsat	550	12	2.41s	3.66s
Sat	550	13	1.61s	2.53s

Figure 8 shows the efficiency of PSatz for problems of 550 variables. It appears that PSatz has a linear speedup on 16, 32, 64 and 128 processors for these problems.

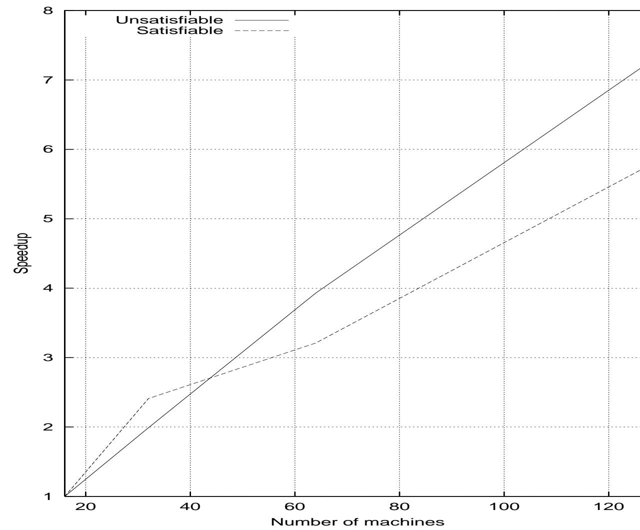


Figure 8. The average speedup of PSatz compared with 16 processors for hard random 3-SAT problems of 550 variables on 16, 32, 64 and 128 processors.

Table X and XI show the performance of PSatz for problems with 600 variables. T_{com32} is the communication time of the master when PSatz runs on 32 processors.

Table X. *Communication time of PSatz for problems with 600 variables.*

	#V	#P	T_{32}	T_{128}	T_{32}/T_{128}	$\#WB_{128}$
Unsat	600	9	7964.11s	2018.55s	3.94	596.77
Sat	600	11	4287.18s	1296.90s	3.30	90.18

Table XI. *Run time of PSatz for problems with 600 variables.*

	#V	#P	T_{com32}	T_{com128}
Unsat	600	9	3.77s	4.44s
Sat	600	11	2.18s	3.18s

As is shown in Figure 9, PSatz scales well for these hard problems, especially when the problems are unsatisfiable. Note that on 128 processors for unsatisfiable random problems with 550 variables, only 0.96% of real time is used for communications (0.21% for problems with 600 variables).

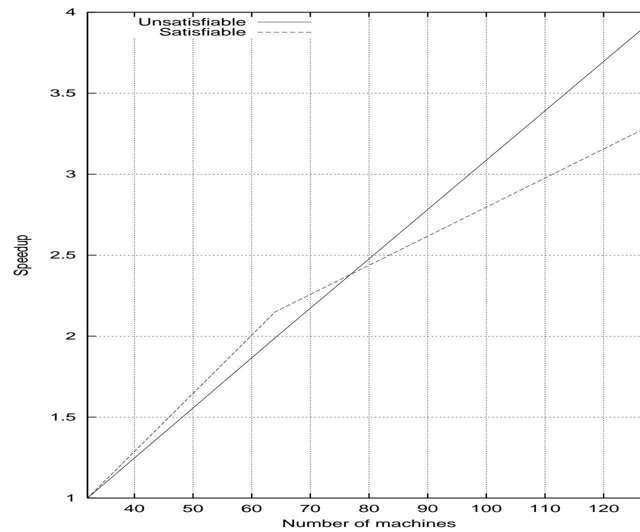


Figure 9. The average gain of PSatz compared with 32 processors for hard random 3-SAT problems with 600 variables on 32, 64 and 128 processors

5.2. Structured problems

We compare the performance of PSatz with Satz only for problems that Satz does not instantaneously solve and PSatz solves in reasonable time. These are problems of Bounded Model Checking (BMC), ii32 in DIMACS, the Beijing challenge suite and 3-Round DES problems, all available in SATLIB [19]. In the following Tables, T_1 is the real run time on one processor, T_{32} is the real run time on 32 processors, T_{64} is the real run time on 64 processors, Sp is the speedup, $\#WB$ is the number of load balancing and T_{com} is the communication time.

PSatz inherits a preprocessing of the input formula from Satz, consisting in adding all resolvents of length less than 4. The preprocessing is repeated by each slave, resulting in an overhead for the parallelization. For large structured problems, the overhead may not be negligible, especially when the search itself is fast.

Tables XII and XIII show the performance of PSatz for structured problems on 64 and 32 processors. We observe a super-linear speedup for 2 problems *des-r3-b4-k1.2* and *ii32d3*. For these problems, PSatz finds the solution in the right subtree by a processor when it runs on many processors. In the sequential case, the left subtree is developed before the right subtree but does not contain any solution.

Table XII. *Run time of PSatz for some structured problems on 64 processors.*

Problem	#V	#C	T_1	T_{64}	Sp	#WB	T_{com}
des-r3-b1-k1.1 *(sat)*	1461	8966	3599s	81s	44.43	2	5s
des-r3-b1-k1.2 *(sat)*	1450	8891	5171s	137s	37.74	83	5s
des-r3-b4-k1.2 *(sat)*	5721	35963	> 3 days	35s	> 7405.71	0	7s
2bitadd_10 *(unsat)*	590	1422	> 3 hours	400s	> 36	185	2s

Note that pSatz performs better than Sato and Chaff on many hard structured SAT problems such as “longmult” instances.

5.3. Load balancing for structured problems

Figure 10 shows the number of load balancings of PSatz done during each 20 seconds when it solves the *2bitadd_10* instance in the Beijing challenge suite on 64 processors. Solving a structured problem with PSatz can generate many load balancings at the beginning and at the end of the resolution. In any case, the dynamic load balancing approach used in PSatz avoids processor idleness.

Table XIII. *Run time of PSatz for some structured problems on 32 processors.*

Problem	#V	#C	T_1	T_{32}	Sp	#WB	T_{com}
barrel5 *(unsat)*	1407	5383	56s	3s	18.66	5	1s
barrel6 *(unsat)*	2306	8931	652s	29s	22.48	7	2s
barrel9 *(unsat)*	8903	36606	2690s	126s	21.34	7	3s
longmult6 *(unsat)*	2848	8853	33s	11s	3	10	3s
longmult7 *(unsat)*	3319	10335	162s	20s	8.1	28	4s
longmult8 *(unsat)*	3810	11877	506s	38s	13.31	45	6s
longmult9 *(unsat)*	4321	13479	996s	51s	19.52	62	5s
longmult10 *(unsat)*	4852	15141	1558s	72s	21.63	63	5s
longmult11 *(unsat)*	5403	16863	2148s	89s	24.13	68	4s
longmult12 *(unsat)*	5974	18645	2480s	108s	22.96	101	5s
longmult13 *(unsat)*	6565	20487	2782s	126s	22.07	98	5s
longmult14 *(unsat)*	7176	22389	3122s	148s	21.09	118	4s
longmult15 *(unsat)*	7807	24351	3864s	182s	21.23	112	6s
ii32d3 *(sat)*	824	19478	$>$ 3 days	18s	$>$ 14400	0	4s

6. Related work

Our work belongs to a large field of arborescent search parallelization. We find three parallel DPLL procedures in the literature to solve SAT problems, all dividing the input formula into subformulas in the usual way and answering the three main questions of a dynamic load balancing in a simple and elegant way. The common issue of these systems is how to decide when a subformula is no longer divided and is sequentially solved by a processor.

Böhm and Speckenmeyer proposed a general fully distributed load balancing framework on a message-based MIMD processor for a Transputer system built up from up to 256 processors (INMOS T800/4MB) [2]. This approach is applied to solve SAT problems. Concretly, the input formula is divided into subformulas. The generated subformulas represent workloads distributed among the processors such that all processors have the same amount of workload (if possible). Small subformulas are solved by the sequential SAT-solver. Note here that a subformula is not divided after the sequential SAT-solver begins to solve it.

Each processor keeps a list of subformulas. The balancing is performed by sending subformulas among processors as soon as a processor has less than s subformulas to solve.

A subformula $\mathcal{F}_1$ is solved by a sequential solver on a processor p if its estimated load $\lambda(\mathcal{F}_1) < \beta * \lambda(p)$, where $\lambda(p)$ is the load sum of all

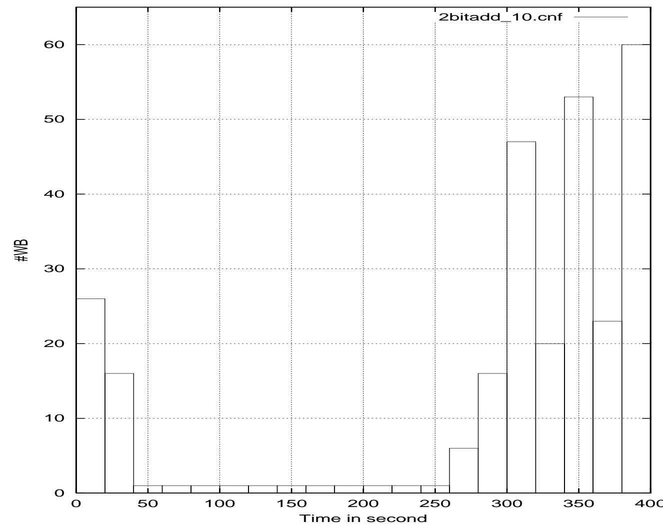


Figure 10. The number of load balancings of PSatz every 20 seconds when the search proceeds for solving the 2bitadd_10 instance

subformulas kept by p and β is a parameter. Otherwise $\mathcal{F}_1$ continues to be divided. By adjusting parameters such as s and β for a particular class of SAT problems, the efficiency of workload balancing can be optimized.

The approach of Böhm and Speckenmeyer can be considered as RID (Receiver Initiative Decision) technique, since it is the processor having less than s subformulas that fires a task redistribution.

Based on an earlier version of sequential SATO, Zhang, Bonacina and Hsiang developed a parallel SAT solver, called PSATO [35]. PSATO also supports accumulation of work over time and fault tolerance. However it seems that PSATO doesn't involve look-back techniques, contrarily to the recent version of SATO.

PSATO uses master-slave model for load balancing. The master keeps a list of m guiding paths. A guiding path contains the successive branching variables chosen by SATO and their current value. When the current value is 1 the remaining value is 0, and when the current value is 0 there is no remaining value (corresponding the remaining value -1 in PSatz). A guiding path uniquely determines a subformula (or a sub search space) through unit propagation, since each branching variable with its current value can be considered as a unit clause added into the input formula.

The number m of the guiding paths kept by the master is 10% or 4 more than the number of processors. If m is or becomes smaller

than that, the largest subformula (or equivalently the shortest guiding path) in the list is divided. Otherwise the smallest subformula (or equivalently the longest guiding path) in the list is no longer divided and is sequentially solved in an idle processor (if any).

PSATO has been used to solve random 3-SAT problems and quasigroup problems [35].

PaSAT [31] is a parallel DPLL procedure exploiting look-back techniques. It uses the same load balancing technique as PSATO but exchanges lemmas learned during the search among processors. So the main originality of PaSAT is the exploitation of the lemmas (or clauses) learned in a processor by all other processors.

The approach of PSATO and PaSAT can be considered as SID (Sender Initiative Decision) technique, since it is the master which generates tasks and sends a task to an idle slave. This is different from PSatz which uses a RID approach.

The common feature of Böhm and Speckenmeyer's system, PSATO and PaSAT is that an implicit depth cutoff technique is used to decide whether a subformula should be divided or should be sequentially solved in a processor and when a processor begins to sequentially solve a subformula, it is not stopped or suspended for load balancing, i.e., it does not give a part of its task to an idle processor. The risk is that when this subformula is very long to solve, other processors may remain idle after finishing all other tasks, which is possible since SAT is very irregular. Nevertheless, the fault tolerance of PSATO allows to partly remedy this situation, because when the busy slave dies or its allotted time runs out, the corresponding guiding path will be split and redistributed.

The notion of context in PSatz can be considered as an extension of the notion of guiding path in PSATO, since a guiding path only includes branching variables but a context also includes variables fixed by unit propagation. The advantage of including variables fixed by unit propagation is that we have a more precise load estimation for a subformula.

Recall that the slave such that the position of the first remaining subtree is the smallest is considered as the most loaded in our approach. This position is obtained by counting the number of all variables fixed before the first remaining tree and whose remaining value is -1, whereas in PSATO only the number of branching variables (i.e. the length of the guiding path) is taken into account to estimate the load of a subformula. Intuitively, more variables do we fix in a formula, simpler does it probably become. However more branching variables do not necessarily fix more variables in a formula (via unit propagation).

Of course, the load estimation used in PSatz may be still imprecise, as any other load estimation today for SAT problems. The work stealing mechanism used in PSatz allows to correct the possible imprecise estimation, since if an estimated easy task is discovered hard in a processor, a part of it will be stolen at a later time. In fact, as soon as a processor becomes idle, the first remaining subtree of the most loaded processor is stolen for load balancing. So our approach completely eliminates the idle slave phenomenon with little communication overhead, at least for the large sample of SAT problems we have tested in this paper.

We do not implement lemma exchanges in PSatz as in PaSAT, since Satz does not include look-back techniques. Thanks to the work stealing mechanism, we do not need depth cutoff technique.

While PSATO is implemented using a special communication library called P4 and PaSAT uses the Distributed Object-Oriented Threads System [31], PSatz uses RPC and Posix Threads, which are standard. Recall that the code sources of PSatz is freely available and PSatz runs on any networked Linux or Unix workstations.

7. Conclusion

We propose a dynamic load balancing scheme to parallelize a class of SAT solver. The scheme uses master/slave model for communication. If one or several slaves are idle during the initialization or after finishing their work, the master evaluates the load of each busy slave and steals the first remaining subtree of the most loaded slave for an idle slave. Since the load of each slave is dynamically adapted as the search proceeds, we overcome the difficulty in predicting a search tree size when solving a subformula. The scheme can be used to parallelize any DPLL-based solver without look-back techniques.

We apply the scheme to parallelize Satz, one of the best generic SAT solvers, to obtain the parallel solver called PSatz. Experimental results show the good performance of PSatz. Using this efficient generique solver, we can obtain better results than ad-hoc solvers such as Kcnfs for random problems or Chaff and Sato for structured problems.

PSatz also supports fault-tolerance computing and accumulation of work over time by checkpointing. As all communications are realized using standard RPC (Remote Procedure Call), PSatz works on all networked Unix/Linux machines and is easily portable.

The communication overhead of PSatz is small even when it runs with 200 processors. If the overhead becomes important for a larger number of machines (e.g. 1000 machines), a possible way is to use a hierarchy of masters to reduce the communication overhead, in which

each sub-master manages a number of different slaves and the general master manages the sub-masters according to the same principle. We did not find a cluster sufficiently large to test this approach.

We plan to develop PSatz in global computing, intending to use idle machines networked by Internet in the world (e.g. SETI@home, XtremWeb [13]). In such a framework, the whole search is partitioned and contexts are sent to idle machines. After some time of execution new contexts are sent back, from which new computations can be performed later or on other machines.

References

1. D. Le Berre. Exploiting the real power of unit propagation lookahead. In *Proceeding of SAT'2001, Electronic Notes in Discrete Mathematics (Henry Kautz and Bart Selman, eds.)*, volume 9. Elsevier Science Publishers, Boston, USA, 2001.
2. M. Böhm and E Speckenmeyer. A fast parallel SAT-solver with Efficient Workload Balancing. In *Third International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdate, USA, 1994.
3. A. Braunstein, M. Mezard, and R. Zecchina. Survey propagation: an algorithm for satisfiablility, http://www.ictp.trieste.it/∼zecchina/sp.
4. S.A. Cook. The Complexity of Theorem Proving Procedures. In *3rd ACM Symp. on Theory of Computing*, pages 151–158, 1971.
5. S.A. Cook. A short proof of the pigeon hole principle using extended resolution. In *SIGACT News*, pages 28–32, 1976.
6. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communication of ACM*, pages 394–397, 1962.
7. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of ACM*, pages 201–215, 1960.
8. O. Dubois and G. Dequen. A backbone-search heuristic for efficient solving of hard 3-sat formulae. In *Proceeding of IJCAI-01, 17th International Joint Conference on Artificial Intelligence (Bernhard Nebel, eds.)*, volume 1, pages 248–253, Seattle, USA, 2001.
9. O. Dubois and G. Dequen. Renormalization as a function of clause lengths for solving random k-sat formulae. In *Proceedings of SAT'2002, 5th International Symposium on Theory and Applications of Satisfiability Testing, (John Franco, eds.), http://gauss.ececs.uc.edu/Workshops/SAT/sat2002list.html*, pages 130–132, Cincinnati, USA, 2002.
10. Luis Guerra e Silva, Joao P.Marques Silva, Luis M.Silveira, and Karem A.Sakallah. Timing analysis using propositional satisfiability. In *Proceedings of the ICECS, 5th IEEE International Conference on Electronics, Circuits and Systems*. IEEE pubisher, Lisboa, Portugal, 1998.
11. J.W. Freeman. *Improvement to Propositional Satisfiability Search Algorithms*. Ph.d. thesis, Univ. of Pennsylvania, Philadelphia, http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/biblio.html, 1995.

12. A.V. Gelder and Y.K. Tsuji. Satisfiability testing with more reasoning and less guessing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26, 1996.
13. C. Germain, V. Néri, G. Fedak, and F. Cappello. Xtremweb: Building an Experimental Platform for Global Computing. In *Grid 2000, 1st IEEE/ACM International Workshop on Grid Computing, (Rajkumar Buyya and Mark Baker Eds.)*. Springer-Verlag, LNCS 1971, Bangalore, India, 2000.
14. Y. Grama and V. Kumar. A Survey of Parallel Search Algorithms for Discrete Optimization Problems. – Personnal communication, http://citeseer.nj.nec.com/et93survey.html. University of Minnesota (1993).
15. Edward A. Hirsch and Arist Kojevnikov. UnitWalk: a new SAT solver that uses local search guided by unit clause elimination. Technical Report PDMI preprint 9/2001, Steklov Inst. of Math. at St.Petersburg, 2001.
16. http://sdg.lcs.mit.edu/satsolvers/psolver.
17. http://www.laria.u-picardie.fr/∼cli/englishpage.html.
18. http://www.lri.fr/ simon/satex/satex.php3.
19. Holger H. Hoos and T. Sttzle. SATLIB: An Online Resource for Research on SAT. In *I.P.Gent, H.v.Maaren, T.Walsh, editors, SAT 2000*, pages 283–292, SATLIB is available online at www.satlib.org, IOS Press, 2000.
20. B. Jurkowiak, C.M. Li, and G. Utard. Parallelizing Satz Using Dynamic Workload Balancing. In *Proceeding of SAT'2001, Electronic Notes in Discrete Mathematics (Henry Kautz and Bart Selman, eds.)*, volume 9. Elsevier Science Publishers, Boston, USA, 2001.
21. H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *Proceedings of IJCAI'99, 16th International Joint Conference on Aritificial Intelligence, Inc. (Thomas Dean, Eds.)*, volume 1, pages 318–325, 1999.
22. H. Kautz and B. Selman. Pushing the envelope : Planing, propositional logic, and stochastic search. In *Proceedings of AAAI'96, 13th National Conference on Artificial Intelligence*, pages 1194–1201. AAAI Press, Portland, USA, 1996.
23. C.M. Li. Integrating equivalency reasoning into davis-putnam procedure. In *Proceedings of AAAI'2000, 17th National Conference on Artificial Intelligence*, pages 291–296. AAAI Press, Austin, USA, 2000.
24. C.M. Li. A constraint-based approach to narrow search trees for satisfiability. *Information Processing Letters*, 71:75–80, Elsevier Science Publishers, 1999.
25. C.M. Li and Anbulagan. Heuristic Based on Unit Propagation for Satisfiability Problems. In *Proceedings of IJCAI'97, 15th International Joint Conference on Aritificial Intelligence, Inc. (Thomas DeanMartha E. Pollack, Eds.)*, volume 1, pages 366–371, Nagoya, Japan, 1997.
26. C.M. Li and Anbulagan. Look-Ahead Versus Look-Back for Satisfiability Problems. In *Proceedings of CP-97, Third International Conference on Principles and Practice of Constraint Programming*, pages 342–356. Springer-Verlag, LNCS 1330, Shloss Hagenberg, Austria, 1997.
27. D. Mitchell, B. Selman, and H. Levesque. Hard and Easy Distributions of SAT Problems. In *Proceedings of AAAI'92, 10th National Conference on Artificial Intelligence*, pages 459–465. AAAI Press, San Jose, USA, 1992.
28. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceeding of SAT'2001, Electronic Notes in Discrete Mathematics (Henry Kautz and Bart Selman, eds.)*, volume 9. Elsevier Science Publishers, Boston, USA, 2001.

29. B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of AAAI-94, 12th National Conference on Artificial Intelligence*, pages 337–343. AAAI Press, Seattle, USA, 1994.
30. B. Selman, H. Kautz, and D. McAllester. Ten Challenges in Propositional Reasoning and Search. In *Proceedings of IJCAI-97, 15th International Joint Conference on Aritificial Intelligence, Inc. (Thomas DeanMartha E. Pollack, Eds.)*, volume 1, pages 50–54, Nagoya, Japan, 1997.
31. C. Sinz, W. Blochinger, and W. Kchlin. Parallel SAT-Checking with Lemma Exchange: Implementation and Applications. In *Proceeding of SAT'2001, Electronic Notes in Discrete Mathematics (Henry Kautz and Bart Selman, eds.)*, volume 9. Elsevier Science Publishers, Boston, USA, 2001.
32. M.N Velev. Formal Verification of VLIW Microprocessors with Speculative Execution. In *CAV '00, 12th Conference on Computer-Aided Verification*, pages 296–311. Springer-Verlag, LNCS 1855, Chicago, USA, 2000.
33. H. Zhang. Specifying latin squares in propositional logic. In *in R. Veroff(ed.): Automated Reasoning and Its Applications, Essays in honor of Larry Wos, Chapter 6, MIT Press*, pages 1–30, 1997.
34. H. Zhang. An efficient propositional prover. In *Proceedings of CADE'97, 14th International Conference on Automated Deduction*, pages 272–275. Springer-Verlag, LNCS 1249, Townsville, Australia, 1997.
35. H. Zhang, M.P. Bonacina, and J. Hsiang. PSATO: a Distributed Propositional Prover and Its Applications to Quasigroup Problems. *Journal of Symbolic Computation, Academic Press, London*, 21:543–560, 1996.
36. S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Trans. on Software Engineering*, 14(9):1327–1341, September 1988.