Cumulating Search in a Distributed Computing Environment: A Case Study in Parallel Satisfiability

Hantao Zhang* **Maria Paola Bonacina** †
Department of Computer Science
The University of Iowa
Iowa City, IA 52242-1419, USA
{hzhang,bonacina}@cs.uiowa.edu

Abstract: We present a parallel propositional satisfiability (SAT) prover called PSATO for networks of workstations. PSATO is based on the sequential SAT prover SATO, which is an efficient implementation of the Davis-Putnam algorithm. The master-slave model is used for communication. A simple and effective workload balancing method distributes the workload among workstations. A key property of our method is that the concurrent processes explore disjoint portions of the search space. In this way, we use parallelism without introducing redundant search. Our approach provides solutions to the problems of (i) cumulating intermediate results of separated runs of reasoning programs; (ii) designing high scalable parallel algorithms and (iii) supporting "fault-tolerant" distributed computing. Several open problems in the study of quasigroups have been solved using PSATO.

Keywords: Distributed and parallel computing, propositional satisfiability, constraint satisfaction, fault-tolerant computing.

1 Introduction

We are concerned with distributed implementations of constraint satisfaction algorithms on networked environments. Nowadays, there are dozens of workstations in a typical industrial or academic research laboratory. Such networked workstations represent great computational power that is often underused, especially after hours. This paper shows a way of using such resources to solve hard problems.

Constraint satisfaction has many important applications in many areas of Computer Science and Artificial Intelligence. In recent years, there has been considerable renewed interest in a special case of constraint satisfaction: propositional satisfiability (SAT) problems. For instance, many open quasigroup problems in algebra have been reportedly solved by SAT provers [11, 13, 9].

The difficulty with using a SAT prover as a tool in artificial intelligence is that SAT is NP-complete, and this intractability is a significant obstacle in real problems. No matter how good the algorithms are, we are in constant need of large computing power, in order to solve a significative range of SAT problems in tolerable time. This justifies the importance of exploiting underused networked workstations.

Our goal is to provide a convenient and effective way of using networked workstations. To achieve this goal, we will answer the following questions:

- Are there simple and effective ways to cumulate the effort of reasoning programs which have to run in separated slots of time?

*Partially supported by the National Science Foundation under Grants CCR-9202838 and CCR-9357851.

†Supported in part by the GE Foundation Faculty Fellowship to the University of Iowa and by the National Science Foundation with grant CCR-94-08667.

- Are there highly scalable parallel implementation on networked workstations, such that the more are the workstations, the faster a problem can be solved?

- Are there "fault-tolerant" parallel algorithms such that the failure of one workstation or the interruption of networks causes minimal damage?

Most existing reasoning systems require continuous running time. If a solution is not found within the allotted time, the execution is aborted and all the effort is wasted. There exist several parallel implementations of reasoning algorithms, but they do not address the problem of using underused networked workstations. For instance, an efficient parallel SAT prover is reported in [2] but the machine they used is a parallel computer with continuous run time. We feel that our solutions can be used not only for solving SAT problems, but also other constraint satisfaction problems. This is a realistic expectation, since many constraint satisfaction problems can be easily converted into SAT problems.

We chose the Davis-Putnam algorithm [4] as our basic tool, because this method has been a major practical method for solving SAT problems. The method is based on unit-propagation (i.e., unit-resolution, unit-subsumption) and case-splitting. We already had an efficient sequential implementation of the Davis-Putnam algorithm, called SATO (SAtisfiability Testing Optimized) ([12], [13]). SATO is written in C and is available by anonymous ftp. This paper describes a parallel implementation of SATO, called PSATO.

We have been experimenting with our department's workstations (HPs, Sparcs, SIGs and IBM RS6000s) connected by the Ethernet. A public domain distributed language called P4 is currently installed on this network. P4, developed at Argonne National Laboratory, primarily by Butler and Lusk, provides a C library for programming a variety of parallel machines [3]. Although public domain languages other than P4 are available, it appears that a different choice would not make much difference, since our implementation intends to have the least dependency on the network communication.

One of the major motivation for developing our programs is to attack some open quasigroup problems in algebra [1]. Quasigroup problems raise many hard computational problems. The usefulness of advanced automated reasoning techniques in attacking these quasigroup problems was successfully demonstrated in [14, 5, 11, 13]. Several open problems were solved using the approach described in this paper; other reasoning systems have not been able to reproduce these results.

This paper is organized as follows: the next section introduces SATO and provides a solution on how to cumulate the search effort of SATO based on the notion of *guiding paths*. Section 3 presents the master-slave model used for PSATO. Section 4 demonstrates how PSATO is used to solve open quasigroup problems. The last section summarizes the paper.

2 SATO: A SAT Prover

Our parallel implementation of a SAT prover, PSATO, is based on a sequential SAT prover called SATO (SAtisfiability Testing Optimized) ([12]), which is an efficient implementation of the Davis-Putnam algorithm [4]. In SATO, the trie data structure (i.e., discrimination trees) is used for representing clauses, and a sublinear decidability algorithm for Horn theory is used for unit propagation [13]. In order to understand the parallel implementation, first, we review the Davis-Putnam algorithm, then, we show how to cumulate search based on the notion of guiding paths of the Davis-Putnam algorithm.

2.1 The Davis-Putnam Algorithm

We assume that the reader is familiar with propositional logic and related concepts, such as clause, literal, model, etc. The Davis-Putnam algorithm [4] accepts a set of propositional clauses and returns true (or models) if and only if the input clauses are satisfiable. The Davis-Putnam algorithm is based on three simple facts about truth table logic. First, the conjunction $A \wedge (\overline{A} \vee B)$, where A and B are any formulae, is equivalent to $A \wedge B$ and the conjunction $A \wedge (A \vee B)$ is equivalent to A. It follows that the application of unit resolution and subsumption to any set of ground clauses results in an equivalent set. Second, if X is any set of formulae and A any ground formula, X has a model if and only if either $X \cup \{A\}$ has a model or $X \cup \{\overline{A}\}$ has a model. Third, and equally obviously, if X is any set of ground clauses and A any ground atomic formula, such that A does not occur at least once positively in [some clause in] X and at least once negatively, then the result of deleting from X all clauses where A occurs is a set which has a model if and only if X has a model.

A simple algorithm based on the first two of these facts[1] is shown in Figure 1. The algorithm

[1] Eliminating variables that occur only positively or

```
function Satisfiable ( set S ) return boolean
  /* unit propagation */
  repeat
    for each unit clause L in S do
      /* unit-subsumption */
      delete (L ∨ Q) from S
      /* unit-resolution */
      delete ¬L from (¬L ∨ Q) ∈ S
    od
    if S is empty then
      return TRUE
    else if the null clause is in S then
      return FALSE
    fi
  until no further changes result
  /* case-splitting */
  choose a literal L occurring in S
  if Satisfiable ( S ∪ {L} ) then
    return TRUE
  else if Satisfiable ( S ∪ {¬L} ) then
    return TRUE
  else
    return FALSE
  fi
end function
```

Figure 1: A Simple Davis-Putnam Algorithm

returns *TRUE*, if a model is found, *FALSE*, if the problem is unsatisfiable. It is well-known that this algorithm is sound and complete for propositional clause problems .

In SATO, `unit-propagation` is done by a new algorithm for the satisfiability of Horn theory which avoids unit-subsumption. It has been proven in [13] that the complexity of the new algorithm is linear in the number of literals which are false at the end of `unit-propagation`, thus it is sublinear to the size of the input clauses.

Note that in `case-splitting`, the clause set S is used twice in the two recursive calls of `Satisfiable`. In SATO, the trie data structure is used in such a way that `Satisfiable` is implemented without requiring new memory, because frequently allocating memory is relatively slow.

Naturally, one important place where heuristics may be inserted is in the choice of a literal for splitting. It is well-known that different splitting rules make the performance of the Davis-Putnam algorithm different by a magnitude of several orders. Splitting rules have been extensively studied in a recent paper by Hooker and Vinay [8]. SATO provides several popular splitting rules. However, in the study of quasigroup problems, one rule seems better than the others: *choose one literal in one of the shortest positive clauses* (all the literals are positive).

2.2 Cumulating Search in SATO

By "cumulating search" in solving a SAT problem, we mean that we may visit the search space of the SAT problem one portion at a time without overlapping. There is no standard definition of the search space of a SAT problem. For example, one may consider the set of all assignments of the propositional variables appearing in the input as its search space. In this paper, we consider the depth-first search tree of `Satisfiable` for a given SAT problem as its search space. The search tree of `Satisfiable` is a binary tree, which represents the relation of recursive calls of `Satisfiable` in the usual way. That is, each node represents a call to `Satisfiable`. Given any two nodes a and b, there is a link from a to b with label $\langle L, \delta\rangle$ if and only if a calls b with input $S \cup \{L\}$, where S is the input to a, δ is 1, if b is the first child of a, is 0, if it is the second one. In this way, a path from the root to any node can be represented by

$$(\langle L_1, \delta_1\rangle\langle L_2, \delta_2\rangle ... \langle L_k, \delta_k\rangle)$$

where $\langle L_i, \delta_i\rangle$, $1 \le i \le k$, are the labels of the edges in the path. We say $\langle L, \delta\rangle$ is *open*, if δ is 1, *closed*, if it is 0. The algorithm returns *TRUE*, if at least a leaf of the search tree returns *TRUE*, *FALSE*, if all the leaves of the search tree (which is finite) return *FALSE*.

Obviously, the search tree of the Davis-Putnam algorithm for a given input depends on the splitting rule used by the algorithm. Once the splitting rule is fixed, the search tree is fixed. Besides some general bounds, we cannot know exactly beforehand how large a search tree is.

`Satisfiable` may be forced to stop prematurely at a node of the search tree. This may happen for different reasons, including that the allotted time has expired or the program ran out of memory. Whenever this happens, the path from the root to that node, called *guiding path*, provides very valuable information about the nested calls of `Satisfiable`, which we may use to avoid repeated search when `Satisfiable` is called the next time with the same input. For instance, if `Satisfiable` halts with the guiding path $(\langle x_1, 1\rangle$

only negatively is not necessary for completeness. Moreover, in many types of problems, such as the quasigroup problems that we are especially interested in, the condition never occurs.

$\langle x_5, 0\rangle\langle x_3, 0\rangle$), then when `Satisfiable` is called again with the same input, there is no need to go through paths like ($\langle x_1, 1\rangle\langle x_5, 1\rangle$...) in the search tree.

This idea of using guiding paths is easy to implement in the Davis-Putnam algorithm: we design a Davis-Putnam algorithm that accepts in input, together with the input clauses, a guiding path of the search tree. If the given path is empty, the Davis-Putnam algorithm is unchanged. If the path is not empty, when choosing a literal for splitting, instead of choosing one according to any given splitting rule, we simply extract one pair from the guiding path. If the pair is closed, only the recursive call prescribed by the pair is made, with no backtracking at this point. We call this version of the Davis-Putnam algorithm, which has been implemented in SATO, `Satisfiable-guided`.

Clearly, `Satisfiable-guided` allows us to cumulate the search of the Davis-Putnam algorithm done in discrete times. Our experiments show that for a hard typical SAT problem, which requires several days of running, the length of the guiding path after eight hours of running is between 70 and 200, but the number of branches (or leaves) of its search tree visited during the same period of time is over a million. In other words, the cost of revisiting a guiding path is so small that it can be ignored. Furthermore, it is much easier to pop a pair from the path in `Satisfiable-guided` than to choose a literal by the splitting rule. `Satisfiable-guided` can be also used to simplify the implementation of the incremental Davis-Putnam algorithm described in [7], where the input clause set grows each time the Davis-Putnam algorithm is called.

In short, the notion of guiding paths provides us a simple and effective way to cumulate the search of the Davis-Putnam algorithm, so that a hard SAT problem can be solved in discrete times. Later, we shall show how to use guiding paths to split jobs effectively in a parallel SAT prover.

3 The Master-Slave Model

Before implementing a parallel/distributed program on a network of workstations or on a parallel computer, we have to decide the net-topology for communication among the processors. Even though the P4 language allows us to implement any net-topology on the Ethernet, we simply chose the master-slave model (i.e., the star graph) for PSATO. That is, one processor is designed as *master* and the remaining processors are *slaves*. The master takes care of job distribution and workload balancing. Each slave runs an identical copy of `Satisfiable-guided`. All communication happens between the master and a slave. The reasons to choose this model are:

- Balancing the workload between processors is considered a very difficult part of implementing a SAT prover on parallel machines. However, for the special computing environment we consider here, that is, a set of loosely connected workstations available at discrete times and of uneven computing power, we have a simple and effective solution which only involves the master.

- We wish that our program has the least dependency on the Ethernet, which may be slow, crowded or unreliable, especially when the communication is through a telephone line. Symmetrically, we do not want our program to deteriorate the performance of the Ethernet. The master-slave model allows us to reduce the communication to a minimum. Indeed, once a slave has received its assignment, it can continue its work even if the network is broken.

- The master-slave model is simple, thus easy to implement. It is flexible, allowing the user to add any number of slaves (i.e., workstations) during the experiments. Moreover, this model allows very high scalability, that is, the more processors are used, the faster a solution can be found. This is because we are able to divide the search space of a SAT problem into mutually disjoint portions and each portion is visited only once.

In the following, we discuss these points in detail, with supporting experimental results.

3.1 Master's Work

As stated above, the master's responsibility is to manage all the slaves:

(a) broadcast general information, such choice of splitting rule, allotted time, etc., to each slave;

(b) prepare and balance workloads and distribute them to slaves;

(c) receive reports from slaves and

(d) tell slaves to halt.

Among these duties, (b) is the most important; the others are straightforward. Balancing workload between slaves is important, because, on one hand, idle times of slaves should be avoided, and on the other hand, the workload balancing should take as little computing time as possible. This is not a trivial task, because (i) there is no reliable estimate of the complexity of a SAT problem and (ii) the power of each slave is not known exactly beforehand (because other users may be using the same workstation).

In PSATO, a workload is represented by a pair (S, P), where S is the set of input clauses and P is a guiding path for `Satisfiable-guided` (see section 2.2). Since a hard SAT problem may contain up to several millions of clauses, in our implementation, S is either a file name (containing the input clauses) or a list of parameters for a clause generator.

There are many ways to generate multiple guiding paths for a given SAT problem, in such a way that the guiding paths lead the concurrent instances of `Satisfiable-guided` to search mutually disjoint portions of the search tree. For instance, we may randomly choose two propositional variables, x and y, and generate four guiding paths:

$$\{(\langle x,0\rangle\langle y,0\rangle), (\langle \overline{x},0\rangle\langle y,0\rangle), (\langle x,0\rangle\langle \overline{y},0\rangle), (\langle \overline{x},0\rangle\langle \overline{y},0\rangle)\}$$

In general, if we choose k variables, we may obtain 2^k guiding paths. However, this way of dividing the search space is similar to choosing randomly a variable for splitting in a sequential SAT prover. This random splitting rule is known to have poor performance for many SAT problems. Hence, this is not a good way to balance workloads among slaves.

In the following, we assume that we have a good splitting rule for a sequential SAT prover. We describe below a simple way of balancing workloads, with the property that the search space of a SAT problem is divided into the same portions as by the splitting rule for a sequential SAT prover. Suppose we have a guiding path P, either obtained by a sequential SAT prover or left from the previous run.

Definition 1 Given a guiding path P

$$P = (\langle L_1, 0\rangle ... \langle L_i, 1\rangle ... \langle L_k, \delta_k\rangle),$$

where $\langle L_i, 1\rangle$ is the first open pair in P (counting from left to right), the two *splits* of P are P_1 and P_2:

$$\begin{aligned} P_1 &= (\langle L_1, 0\rangle, .., \langle \overline{L_i}, 0\rangle), \\ P_2 &= (\langle L_1, 0\rangle, .., \langle L_i, 0\rangle, ..., \langle L_k, \delta_k\rangle) \end{aligned}$$

Lemma 2 *Let P_1 and P_2 be the two splits of P, and S be a set of input clauses. The search space of* `Satisfiableguided`(S, P) *is the union of those of* `Satisfiable-guided`(S, P_1) *and* `Satisfiable-guided`(S, P_2).

In our implementation, the master always maintains a list of guiding paths, according to the rule that the number of guiding paths be about 10% higher than the number of slaves. If it is smaller than that, we split one path into two according to definition 1[2]. We control the number of paths in this way, because we do not want to spend unnecessary resources to maintain a large number of them. The list is sorted in such a way that

1. shorter paths go first;
2. among paths of the same length, paths with fewer open pairs go first;
3. the tie is broken arbitraryly between two paths of the same length with the same number of pairs.

The master assigns the next unassigned path in the list to the next idle slave. If a slave has found a model, the master tells all the slaves to halt and then stops itself. If a slave has found that its assignment is unsatisfiable, the master assigns another path to that slave. When the allowed time expires, the master sends the halt signal to all the slaves and, at the same time, collects new paths from each slave. The new paths are saved together with the unassigned paths for the next run.

3.2 Slave's Work

Upon receiving a workload, $\langle S, P\rangle$, from the master, each slave runs `Satisfiable-guided`(S, P), until one of the following occurs:

- `Satisfiable-guided`(S, P) has found a model (return *TRUE*) or has completed the search (return *FALSE*). In this case, the slave reports the result to the master.
- The slave has received a halt signal from the master, or the allotted time (the wall clock) has run out. The slave interrupts `Satisfiable-guided`(S, P) and reports the new guiding path (i.e., the information about the nested calls of `Satisfiable-guided`) to the master.

[2] Of course, if there are no open pairs in any path, then splitting is not possible and one slave will be idle. Our experiences show that this is often the case when a SAT problem is near to be solved.

Each slave needs to check its allotted run time, because the master may die or hang. If that happens, the master will not be able to halt the slaves. Thus, the slaves must be able to halt by themselves by checking their allotted run time. We introduced this provision because we observed the following scenario using P4: the master tries to communicate with a dead slave (there are many factors to bring a slave to death, such as physical failure, or insufficient memory, etc.); as a consequence, the master hangs, whereas all the other slaves continue working. We called this phenomenon "runaway slaves". Such "runaway slaves" may happen quite often if the chosen communication language is not sufficiently "fault-tolerant", e.g. it causes the master to hang if the master tries to communicate with a dead slave. This is another reason why we wish that our program has the least dependency on the Ethernet: communication may propagate "death" of processors.

Our implementation achieves some additional fault-tolerance by establishing that each slave saves in a file or sends by electronic mail to some address a copy of all the information that the slaves sends to the master. In this way, even when the master is dead, the effort of all the "healthy" slaves will not be wasted, because all the information generated by those slaves can be retrieved.

3.3 Performance Evaluation

The master-slave model described above has been implemented in PSATO using P4 as the communication tool and SATO as a sequential SAT prover. Because a fine-grained algorithmic complexity analysis of PSATO appears impossible, we conducted some testing in the hope that it may shed some light on the issue. A more extensive empirical study would be more desirable but very expensive, since it would require to find a representative set of domain examples and to run each parameter scheme several times (once for each setting) on each problem.

Because of the simplicity of the workload balancing method – splitting guiding paths – we do not expect our method to perform regularly well on tests made of a single run. On the other hand, since the computing environment under discussion only allows separated run times, it is more realistic to test the model in this environment. In fact, we do not expect a fancy balancing algorithm to do better, because the powers of the workstations are different and cannot be known a priori (because other programs may be running at the same time as ours).

Table 1: Experiment of the balancing algorithm in PSATO on random 3-SAT unsatisfiable problems. The 20 workstations consists of 10 HP 700 series workstations, 6 IBM RS6000 and 4 Sparcs. The RS6000 was chosen for the experiments with one and five machines, because it has an average computing power. #V = number of variables. #P = number of processors (or workstations). The times are measured in seconds.

#V	#P	Wall Clock	Total Time	Speed-up	Over-head
100	1	22.2	22.2	–	–
	5	7.9	24.4	2.81	0.10
	20	3.6	26.0	6.17	0.17
150	1	1082.5	1082.5	–	–
	5	237.9	1169.1	4.55	0.08
	20	60.7	1212.4	17.83	0.12
200	1	53346.7	53346.7	–	–
	5	10777.0	54947.1	4.95	0.05
	20	2899.3	58793.4	18.40	0.07

We have chosen as test cases random unsatisfiable 3-SAT problems; for satisfiable 3-SAT problems, the performance of PSATO varies dramatically from case to case. Each problem consists of m clauses of length 3 over n variables with ratio $m/n = 4.25$. These have been considered as hard SAT problems [6]. For $n = 100$ (and $m = 425$), we tested a sample of 50 cases on 1, 5 and 20 workstations, respectively. We did the same for $n = 150$. For $n = 200$, we tested only a sample of 20 cases which can be finished in 24 hours on a single machine.

In these experiments of PSATO, the parallel execution is done as follows: first, the master assigns the whole job to one slave with one second of allotted time. After this time expires, the master splits the guiding path obtained from that slave into more paths and distributes them among the slaves, with one minute of allotted time for each slave. After that, the allotted time will be one hour for the rest of the execution. We gave slaves small allotted times, because the test problems are easy in comparison with real application problems, which require many days to solve. For hard SAT problems from real applications, we can use each workstation for about eight hours a day (from midnight to 8am).

The experimental data are listed in table 1. In this table, the "wall clock" is the average time (in seconds) to solve each test case. The "total time" is the sum of the CPU times of all the involved

machines[3]. The "speed-up" is the ratio between the wall-clock time of the sequential prover and the current wall-clock time. It is clear from the table that the harder are the test cases, the higher are the speed-up's, because for hard problems, the master has more chances to manage guiding paths and balance workloads.

We remark that speed-up is an interesting measure, but it is not the best or the only indicator for the computing environment under discussion, because a crowded or slow workstation may contribute very little but it is counted as one machine. The "overhead" of parallel execution may provide more useful information. The "overhead" is the ratio of the extra CPU time (i.e., the current total CPU time minus the sequential CPU time) incurred by the parallel implementation versus the sequential CPU time. This includes the time spent on communication, balancing workload and re-creating data structures. Again, we found that the "overhead" becomes smaller as the difficulty of the problems increases. The data also confirm that our method has very good scalability.

4 Quasigroup Problems

In our opinion, quasigroup problems are much better benchmarks than randomly generated SAT problems for testing constraint solving methods. Quasigroup problems are real problems, have fixed solutions and simple descriptions that are easy to communicate. Most important, some open cases of these problems attracted the interest of several researchers and became a challenge for friendly competition.

The usefulness of general automated reasoning techniques to attack these quasigroup problems was successfully demonstrated in [14, 5, 11, 13, 9] To our knowledge, Zhang [14] was the first to use a general reasoning program to solve an open case of the quasigroup problems. Subsequently, Fujita used MGTP, a model-generation based first-order theorem prover, and Slaney used FINDER, a program based on constraint solving, to solve several open cases [5]. Later, Stickel, McCune, and the first author used their propositional provers to attack these problems and reported very competitive results [11, 12, 13, 9]. In particular, PSATO is able to reproduce all the results obtained by other systems [12, 13].

[3]For the one workstation case, the wall-clock time is in general slightly larger than the cpu time. For simplicity, we assume they are the same.

4.1 Specification of Quasigroups

A quasigroup is a cancellative finite groupoid $\langle S, * \rangle$, where S is a set and $*$ a binary operation on S. The cardinality of S, $|S|$, is called the *order* of the quasigroup. The "multiplication table" for the operation $*$ forms a Latin square, where each row and each column is a permutation of S. Many classes of Latin squares are interesting, partly because they are very natural objects in their own right and partly because of their relationships to design theory [1].

Without loss of generality, let $S = \{0, 1, ..., (v-1)\}$, where v is the order of $\langle S, * \rangle$. The following clauses specify a Latin square: for all elements $x, y, u, w \in S$,

$$x * u = y, x * w = y \Rightarrow u = w \quad (1)$$

$$u * x = y, w * x = y \Rightarrow u = w \quad (2)$$

$$x * y = u, x * y = w \Rightarrow u = w \quad (3)$$

$$(x * y = 0) \vee \cdots \vee (x * y = (v-1)) \quad (4)$$

It was shown in [11] that the following two clauses are valid consequences of the above clauses and adding them into a prover can reduce the search space.

$$(x * 0 = y) \vee \cdots \vee (x * (v-1) = y) \quad (5)$$

$$(0 * x = y) \vee \cdots \vee ((v-1) * x = y) \quad (6)$$

In the next table, we list the quasigroup problems given by Fujita, Slaney and Bennett in their award-winning IJCAI paper [5]. For any x, y, z in $\{0, ..., (v-1)\}$, the following constraints are given in [5][4]:

Name	Constraint
QG1	$x * y = u, z * w = u, v * y = x,$ $v * w = z \Rightarrow x = z,\ y = w$
QG2	$x * y = u, z * w = u, y * v = x,$ $w * v = z \Rightarrow x = z,\ y = w$
QG3	$(x * y) * (y * x) = x$
QG4	$(x * y) * (y * x) = y$
QG5	$((x * y) * x) * x = y$
QG6	$(x * y) * y = x * (x * y)$
QG7	$((x * y) * x) * y = x$

In the following, problem QGi.v denotes the problem represented by clauses (1)–(6) plus QGi for $S = \{0, ..., (v-1)\}$. In addition, the idempotency law, $x * x = x$ and the isomorphism cutting clause, $1 + x < y \Rightarrow x * 0 \neq y$, are assumed in every problem unless otherwise stated. The reader is referred to [5] and [11] for the details of these problems.

Propositional clauses are obtained by simply instantiating the variables in clauses (1)–(6) by

[4]The QG7 constraint is the one used in [11], not [5].

values in S and replacing each equality $x * y = z$ by a propositional variable $p_{x,y,z}$. The number of the propositional clauses is determined by the order of the quasigroup (i.e., v) and the number of distinct variables in a clause. Constrains QG1 and QG2 can be handled in the same way. For QG3–QG7, we have to transform them into "flat" form. For example, the flat form of QG5 is

$$(x * y = z), (z * x = w) \Rightarrow (w * x = y).$$

It can be shown that the two "transposes" of the above clause are also valid consequences of QG5:

$$(w * x = y), (x * y = z) \Rightarrow (z * x = w),$$
$$(z * x = w), (w * x = y) \Rightarrow (x * y = z).$$

It has been confirmed by experiments that adding these "transposes" to the input can reduce the search space. This is true for QG3–QG7.

4.2 Experimental Results

For a quasigroup of order v, the number of propositional clauses obtained from clauses like QG1 and QG2 above is $O(v^6)$, because there are 6 distinct variables in QG1 and QG2. For large v, in addition to the large of number of clauses, the search space is huge. For instance, SATO, or any known computer programs, including those in [14, 5, 11, 13, 9], could not complete an exhaustive search for QG2.10.

Table 2 shows the performance of SATO (on a Sparc2 workstation) on quasigroup problems of small orders. It is clear that the complexity is exponential in the order of quasigroups. QG5.9, the first open quasigroup problem solved by Zhang [14], can be considered as trivial now.

Despite of the difficulty of quasigroup problems, many of them are solved by a computer. Table 3 shows some recently solved problems. PSATO is able to reproduce all the results in this table. In particular, PSATO was the first to solve QG2.14, QG2.15, QG5.14 (without the idempotency), QG6.15, QG7.15, and QG6.17. Except for QG2.14, QG2.15 and QG6.17, we have not heard of any other programs which reproduce these results.

One of the reasons why problems like QG5.14 (non-idem), QG6.15 and QG7.15 are so hard is that they are unsatisfiable SAT problems. An exhaustive search is necessary to establish unsatisfiability. Since few heuristic methods can help to cut the search space, this requires an enormous amount of computing time. From Table 4, it is clear that the use of distributed programs on networked workstations is indispensable to our

Table 2: Performance of SATO on some quasigroup problems. The time (in seconds) was collected on a Sparc2 workstation. "Branches" is the number of branches (or leaves) of the corresponding search tree (which is half of the size of the search tree). "Create" is the time spent on creating the internal data structure, thus it is part of the overhead of parallelism. "Search" is the time spent on search.

Prob.	Models	Branches	Create (sec.)	Search (sec.)
QG1.7	8	376	1	1
.8	16	102610	3	379
QG2.7	14	340	1	1
.8	2	80245	3	341
QG3.8	18	1072	.1	3
.9	0	48545	.3	157
QG4.8	0	925	.1	2
.9	178	52826	.2	168
QG5.9	0	19	.1	.2
.10	0	62	.3	.5
.11	5	111	1	2
.12	0	369	2	7
.13	0	10764	3	224
QG6.9	4	24	.2	.2
.10	0	150	.4	.7
.11	0	519	1	6
.12	0	5728	1	92
QG7.9	4	7	.2	.2
.10	0	54	.4	.4
.11	0	254	1	3
.12	0	1281	2	22
.13	64	27988	2	592

Table 3: Quasigroup problems recently solved for the first time by a computer: Y means that a model is found; N means an exhaustive search confirms that no models exist. O means that case remains open. Some other open cases are: QG5.18, QG6.20, and QG7.33.

v :	9	10	11	12	13	14	15	16	17
QG1				O					
QG2		O		Y		Y	Y		
QG3				Y					
QG4				Y					
QG5	N	N	Y	N	N	N	N	N	O
QG6		N	N	N		N	N		Y
QG7		N	N	N		N	N		

Table 4: Performance data of PSATO on hard quasigroup problems. #P = number of machines. A workday equals 8 hours.

Prob.	#P	Workdays	P-Measure
QG5.14(*)	20	35	11
QG6.15	20	8	8
QG7.15	20	5	6
QG5.16	20	4	5
QG6.17	8	2	–

(*): non-idempotency

success. For instance, it would require approximately 240 days of continuous running on a single workstation to solve QG5.14 (non-idempotency).

We observed in our experiments on quasigroups that, if a problem is unsatisfiable, guiding paths provide a good empirical measure of the complexity of the problem. Assume that the guiding path is obtained after the first workday, and n is the number of consecutive open pairs in the beginning of this path, then this problem will need approximately $O(2^n)$ workdays to complete (on a single workstation). This empirical rule does not apply to satisfiable SAT problems. Let us call this n the *P-measure* of the given problem. For instance, the P-measure of QG6.15 is 8 and we needed 160 workdays to solve it. P-measures of all the problems in Table 4 are given in the last column. Note that the P-measure of QG2.10 is 25 and that of QG1.12 is 89. If both problems are unsatisfiable, then it is not practical to confirm this by our program, because it would take approximately 2^{80} workdays to solve QG1.12.

5 Conclusions

We presented PSATO, a master-slave distributed SAT prover, for networked workstations. Each slave process executes a modified version of the sequential SAT prover SATO. The master process decomposes the given problem among the slaves, in such a way that the slaves explore non-overlapping portions of the search space in discrete times. In this way, we are able to exploit parallelism, without incurring in the overhead of redundant search, e.g. parallel processes searching the same portion of the search space.

Our parallel program is as easy to use as a sequential program. A typical scenario is the following: in the morning, we check the output of the previous run and prepare the task for the night.[5] Our work shows that it is possible to utilize the unused computing power of networked workstations after hours to attack computationally intensive problems.

We were able to use our program to solve several open problems in the study of quasigroups. We believe that the same approach can be generalized at least to constraint satisfaction techniques. A similar approach may be adopted in general theorem proving as well, although partioning the search space in non-overlapping portions is a much more difficult problem in general theorem proving.

Acknowledgements

We thank Bill McCune for providing us with the P4 scheme code of the master-slave model. We thank Rusty Lusk for helping us to use P4.

References

[1] F. Bennett, L. Zhu. Conjugate-orthogonal Latin squares and related structures. J. Dinitz & D. Stinson (eds), *Contemporary Design Theory: A Collection of Surveys*, 1992.

[2] M. Boehm, E. Speckenmeyer. A fast parallel SAT-solver - efficient workload balancing. Presented at Third International Symposium on Artificial Intelligence and Mathematics. Fort Lauderdale, Florida, 1994.

[3] R. Butler and E.L. Lusk. User's Guide to the p4 Programming System, Technical Report ANL-92/17, Mathematics and Computer Science Division, Argonne National Laboratory, October 1992.

[4] M. Davis, H. Putnam. (1960) A computing procedure for quantification theory. J. of ACM, 7, 201-215.

[5] M. Fujita, J. Slaney, F. Bennett. Automatic Generation of Some Results in Finite Algebra, *Proc. International Joint Conference on Artificial Intelligence*, 1993.

[6] A. Goldberg. Average case complexity of the satisfiability problem. *Proc. Fourth Workshop on Automated Deduction*, pp. 1-6 (1979).

[5] We have been using the "at" facility of Unix so that we do not have to start personally the program late in the night.

[7] J.N. Hooker, V. Vinay. Solving the incremental satisfiability problem. *J. of Logic Programming* 15:177–186, 1993.

[8] J.N. Hooker, V. Vinay. Branching rules for satisfiability. Presented at Third International Symposium on Artificial Intelligence and Mathematics. Fort Lauderdale, Florida, 1994.

[9] W. McCune. A Davis-Putnam program and its application to finite first-order model search: quasigroup existence problems. Preprint, Division of MCS, Argonne National Laboratory, 1994.

[10] J. Slaney. FINDER, Finite Domain Enumerator: Version 2.0 Notes and Guide, technical report TR-ARP-1/92, Automated Reasoning Project, Australian National University, 1992.

[11] J. Slaney,, M. Fujita, M. Stickel. Automated reasoning and exhaustive search: Quasigroup existence problems. To appear in *Computers and Mathematics with Applications*.

[12] H. Zhang. SATO: A decision procedure for propositional logic. Association for Automated Reasoning Newsletter, No 22, March 1993, pp. 1-3.

[13] H. Zhang, M. Stickel. Implementing the Davis-Putnam algorithm by tries. Technical Report, Dept. of Computer Science, The University of Iowa, 1994.

[14] J. Zhang. Search for idempotent models of quasigroup identities. Typescript, Institute of Software, Academia Sinica, Beijing, 1990.