From: AAAI-00 Proceedings. Copyright © 2000, AAAI (www.aaai.org). All rights reserved.

A Distributed Algorithm to Evaluate Quantified Boolean Formulae

Rainer Feldmann, Burkhard Monien, Stefan Schamberger
Department of Computer Science
University of Paderborn
Fürstenallee 11, 33102 Paderborn, Germany
(obelix|bm|schaum)@uni-paderborn.de

Abstract

In this paper, we present PQSOLVE, a distributed theorem-prover for Quantified Boolean Formulae. First, we introduce our sequential algorithm QSOLVE, which uses new heuristics and improves the use of known heuristics to prune the search tree. As a result, QSOLVE is more efficient than the QSAT-solvers previously known. We have parallelized QSOLVE. The resulting distributed QSAT-solver PQSOLVE uses parallel search techniques, which we have developed for distributed game tree search. PQSOLVE runs efficiently on distributed systems, i. e. parallel systems without any shared memory. We briefly present experiments that show a speedup of about 114 on 128 processors. To the best of our knowledge we are the first to introduce an efficient parallel QSAT-solver.

Introduction

QSAT generalizes propositional satisfiability (SAT) which has been thoroughly analyzed, see e.g. (Gu et al. 1997), since it is the prototype of an NP-complete problem and has applications in automated reasoning, computer-aided design, computer architecture design, planning, and VLSI. QSAT is the problem to decide the satisfiability of propositional formulae, in which the variables may either be universally ($\forall$) or existentially ($\exists$) quantified. Thus, the inputs of a QSAT-solver look like the following:

$$f(X) = Q_N X_N Q_{N-1} X_{N-1} \dots Q_0 X_0 : f',$$

with $Q_i \in \{\forall, \exists\}$. The X_i are disjoint sets of boolean variables, $X = \cup_{i=0}^{N} X_i$, and f' is a propositional formula over X. $f(X) = \exists X_N \phi$ is satisfiable iff there is a truth assignment for the variables in X_N such that ϕ is true, and $f(X) = \forall X_N \phi$ is satisfiable iff ϕ is true for all possible truth assignments of the variables in X_N.

According to the increasing interest in problems not in NP, QSAT has been studied as a prototype of a PSPACE-complete problem. Furthermore, by restricting the number of quantifiers to some fixed c, it has been analyzed as a family of prototypical problems for the polynomial hierarchy. (Gent and Walsh 1999) study QSAT and show that a phase transition similar to that observed for SAT does occur for QSAT, too. They show that some models of random instances are "flawed" and propose a model, in which each clause contains at least two existentials.

Copyright © 2000, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

The first QSAT-solver, published in (Kleine-Büning, Karpinski, and Flögel 1995), is based on resolution. (Cadoli, Giovanardi, and Schaerf 1998) propose EVALUATE, an algorithm based on the Davis-Putnam procedure for SAT. EVALUATE contains a heuristic to detect trivial truth of a QSAT instance. (Rintanen 1999) presents an algorithm based on the Davis-Putnam procedure and introduces a heuristic called Inverting Quantifiers (IQ). He shows that the use of IQ before the entire evaluation process speeds up the computation of several QSAT instances which originate from conditional planning.

First, we describe QSOLVE, a sequential QSAT-solver. QSOLVE is based on the Davis-Putnam procedure for SAT. We make use of a generalization of the data structures of a SAT-solver (Böhm and Speckenmeyer 1996). The data structure supports the operations to delete a clause, to delete a variable, and to undo these deletions. We implemented most of the heuristics that were introduced by (Cadoli, Giovanardi, and Schaerf 1998) and (Rintanen 1999). Moreover, we have developed an approximation algorithm for the IQ-heuristic of (Rintanen 1999) which allows the use of IQ during the evaluation process. We developed a simple history heuristic to determine on whether or not to apply the heuristic to detect trivial truth (TTH) of (Cadoli, Giovanardi, and Schaerf 1998) at a node in the search tree. In addition, we have developed a heuristic to detect trivial falsity (TFH) of a QSAT instance. TFH is controlled by a history heuristic, too. We show experimentally, that with the help of these additional heuristics our algorithm is faster than the ones mentioned above.

Then we present our parallelization of QSOLVE. The parallelization is similar to the parallelization used in the chess program ZUGZWANG (Feldmann, Monien, and Mysliwietz 1994). For chess programs this parallelization is still the best known (Feldmann 1997).

Since QSAT can be regarded as a two-person zero-sum game with complete information, it is not surprising that techniques from parallel chess programs are applicable to QSAT-solvers. We briefly explain the general concepts of our parallelization and then concentrate on the heuristics that are used in order to schedule subproblems. Finally, we show

experimentally that the resulting parallel QSAT-solver PQSOLVE works efficiently. On a set of randomly generated formulae the speedup of the 128-processor version is about 114. To the best of our knowledge, PQSOLVE is the first parallel QSAT-solver. Moreover, it is very efficient. Its efficiency of about 90 % is partly due to the fact that the dynamic load balancing together with our scheduling heuristics often result in a "superlinear" speedup. The effect has already been observed for SAT (Speckenmeyer, Monien, and Vornberger 1987) and indicates that the sequential depth-search algorithm is not optimal. However, as the sequential program is best among the known algorithms for QSAT, we define speedup on the basis of our sequential implementation.

QSOLVE: The Sequential Algorithm

The skeleton of QSOLVE is presented below:

```
boolean Qsolve(f)        /* f is "call by value" parameter */
  /* let f = Q_N X_N Q_{N-1} X_{N-1} ... Q_0 X_0 : f', */
  /* let s ∈ {true, false, unknown}, x a literal */

  s ← simplify(f);                       /* f may be altered */
  if (s ≠ unknown) return s;                       /* prune */

  if (Q_N = ∃)
    if (N ≥ 2)            /* f has ≥ 3 blocks Q_N ... Q_0 */
      if (x ← InvertQuantifiers(f))                   /* IQ */
        s ← reduce(f,x = true);                /* f is altered */
        if (s ≠ unknown) return s;                 /* prune */
        return Qsolve(f);                      /* recursion */

    x ← SelectLiteral(f);

    s ← reduce(f,x = true);                  /* f is altered */
    if (s ≠ unknown) return s;                     /* prune */
    if (Qsolve(f) = true)                         /* branch */
      return true;                                /* cutoff */

    undo(f,x = true);                        /* f is altered */
    s ← reduce(f,x = false);                 /* f is altered */
    if (s ≠ unknown) return s;                     /* prune */
    return Qsolve(f);                             /* branch */

  else /* Q_N = ∀ */
    if (TrivialTruth(f) = true)               /* TTH: SAT */
      return true;                                 /* prune */
    if (TrivialFalsity(f) = false)            /* TFH: SAT */
      return false;                                /* prune */

    x ← complement(SelectLiteral(f));         /* complement */

    ...
    if (Qsolve(f) = false)                        /* branch */
      return false;                               /* cutoff */
    ...
end
```

When called for a formula f, f is simplified first. This is done by setting the truth values of monotone and unit existential variables. The formula is checked for an empty set of clauses or the empty clause, etc.

We measure the length of a clause in terms of $\exists$-quantified literals of the clause. Thus, a variable $x \in X_N$ is unit existential iff $Q_N = \exists$ and there is a clause that contains x as the only $\exists$-quantified variable. The simplification is then performed according to (Cadoli, Giovanardi, and Schaerf 1998). The function "simplify" may deliver the result that the formula is satisfiable or unsatisfiable.

Then, in the first case ($Q_N = \exists$) the IQ-heuristic may determine a literal that must be set to true in order to satisfy f.

If the IQ-heuristic does not deliver the desired literal a branching literal is determined in such a way that x is set to true in the first branch. Unless a cutoff occurs, both branches are tested recursively.

In the second case ($Q_N = \forall$), a test for trivial truth and trivial falsity is performed first. The literal that is selected for the branching is negated. The cutoff condition is changed according to the $\forall$-quantor. The rest of the case $Q_N = \forall$ is similar to the first case.

In the next sections, we will describe the features of QSOLVE mentioned above in more detail.

Literal Selection

The function SelectLiteral selects a literal $x \in X_N$. SelectLiteral first determines the variable v to branch with. Then the literal $x \in \{v, \bar{v}\}$ is selected in such a way that it is set to true in the first branch and to false in the second one. If the literal is $\exists$-quantified, the selection is done like in the SAT-solver by (Böhm and Speckenmeyer 1996): For every literal x let $n_i(x)$ ($p_i(x)$) be the number of negative (positive) occurrences of literal x in clauses of length i, and let $h_i(x) = max(n_i(x), p_i(x)) + 2 \cdot min(n_i(x), p_i(x))$. A literal x with lexicographically largest vector $(h_0(x), \ldots h_k(x))$ is chosen for the next branching step. The idea is to choose literals that occur as often as possible in short clauses of the formula and to prove satisfiability in the first branch already. The setting of these literals often reduces the length of the shortest clause by one. Often, the result is a unit clause or even an empty clause. If $Q_N = \forall$, the literal is negated, i. e. the branching variable is the same but the branches are searched in a different order. This often helps to prove unsatisfiability in the first branch already.

Inverting Quantifiers

The technique to invert quantifiers is based on (Rintanen 1999). We have developed the approximation algorithm presented below. The function InvertQuantifiers (IQ) tries to compute an $\exists$-quantified literal of X_N which must be set to true in order to satisfy f. IQ starts checking on whether there is a unit existential $x \in X_N$. If this is not the case, it tries to create unit existentials by setting monotone $\forall$-literals in unit clauses. Note that a unit clause is a clause with one existential. If none of the $\forall$-literals is monotone, it keeps track of the $\forall$-literal h with a maximum occurrence in unit clauses.

IQ continues recursively by testing h = true and h = false. In our implementation, we stop searching for a literal, if the number of recursive calls of InvertQuantifiers exceeds four. This constant has been the result of a simple program optimization on a small benchmark.

```
literal InvertQuantifiers(f)            /* f is "call by value" */
  /* let f = Q_N X_N Q_{N-1} X_{N-1} ... Q_0 X_0 : f', */
  /* let s ∈ {true,false,unknown}, x, h literals */

  do                                    /* create unit existentials */
    forall (∃-units x ∈ f)
      if (x ∈ X_N) return x;
      s ← reduce(f,x = true);           /* f is altered */
      if (s = false) return NULL;

    forall (∀-literals x)
      if (#_{|C|=1}(x) > 0 and #_{|C|=1}(x̄) = 0)
        /* x is monotone */
        s ← reduce(f,x = false);        /* f is altered */
        if (s = false) return NULL;
      if (#_{|C|=1}(x) = 0 and #_{|C|=1}(x̄) > 0)
        /* x̄ is monotone */
        s ← reduce(f,x = true);         /* f is altered */
        if (s = false) return NULL;
      if (#_{|C|=1}(x) > 0 and #_{|C|=1}(x̄) > 0)
        /* neither x nor x̄ are monotone */
        h ← max_{#_{|C|=1}}(h, x, x̄);
  while (there are ∃-units ∈ f);

  if (h = NULL) return NULL;    /* no recursive search */

  reduce(f,h = true);                   /* recursive search */
  x ← InvertQuantifiers(f);
  if (x ≠ NULL) return x;

  undo(f,h = true);                     /* recursive search */
  reduce(f,h = false);
  return InvertQuantifiers(f);
end
```

Lemma: Let f be a Quantified Boolean Formula, and let x = InvertQuantifiers(f) be a literal of f. f is satisfiable iff $f_{[x=true]}$ is satisfiable.

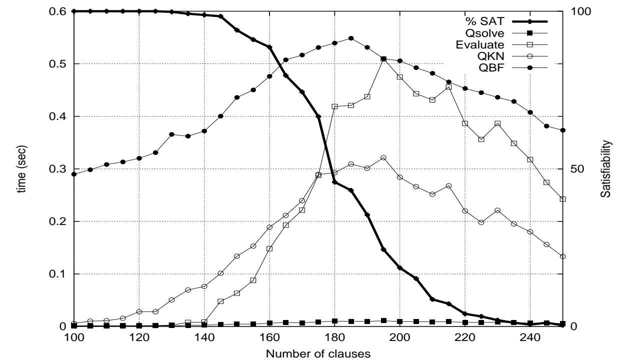
The proof is an easy induction on the recursion depth of InvertQuantifiers. Note that f does not contain unit existentials before the initial call to InvertQuantifiers.

Trivial Truth

(Cadoli, Giovanardi, and Schaerf 1998) present a test for trivial truth: Given a QSAT instance f, delete all $\forall$-quantified literals from the clauses. This results in a set of clauses of $\exists$-quantified literals. Solve the corresponding SAT instance f'. If f' is satisfiable then f is satisfiable. SAT is NP-complete, but algorithmically QSOLVE can be used to solve the SAT-problem. The QSAT-solver does benefit from this heuristic, only if f' is satisfiable.

Adaptive Trivial Truth

Since a considerable amount of time may be wasted for the solving of SAT instances (one at every $\forall$-node of the search tree) we have developed a simple adaptive history heuristic to determine on when to apply this test: At every node of the search tree two global variables s and v are changed: Initially, v=1 and s=2. TrivialTruth is executed if $v \geq s$. If TrivialTruth is executed v is set to 1 and s is set as follows: if the execution proves satisfiability s is set to 2, otherwise $s = max(2 \cdot s, 16)$. If TrivialTruth is not executed then $v = 2 \cdot v$. The idea is that in the parts of the search tree where the function TrivialTruth is successfully applied, the heuristic is used at every second node of the search tree ($s = 2$). In the parts of the tree, where TrivialTruth is unsuccessful, s increases to 16 and thus, the use of TrivialTruth is restricted to every fifth node of the search tree. Again, the constants of 2 and 16 have been the result of a program optimization on some benchmark instances.



The diagram above presents the average running times of QSOLVE, EVALUATE (Cadoli, Giovanardi, and Schaerf 1998), QKN (Kleine-Büning, Karpinski, and Flögel 1995), and QBF (Rintanen 1999) on formulae with 50 variables (15 $\forall$-quantified ones), clauses of length four and three blocks. The formulae have been generated randomly according to Model A by (Gent and Walsh 1999). The average is taken over 500 formulae. QSOLVE and QKN are C programs, EVALUATE is a C++ program, to run QBF we used a binary from Rintanens home page. Note that the running times are arithmetic means of unnormalized data (left y-axis). E.g. from the 500 QSAT instances with 180 clauses we obtain the following running times rounded to 4 decimal digits:

time(sec)	QSOLVE	EVALUATE	QKN	QBF
minimum	0.0004	0.0000	0.0050	0.0700
average	**0.0100**	**0.4188**	**0.2927**	**0.5394**
maximum	0.1225	6.7700	4.3910	1.2200
variance	0.0119	0.6726	0.3969	0.1699

QSOLVE uses considerably less time than any of the other QSAT-solvers. This has been observed for other classes of randomly generated formulae, too. However, it should be pointed out that QBF needs less recursions than QSOLVE.

Adaptive Trivial Falsity

Let $f(X) = Q_N X_N Q_{N-1} X_{N-1} \ldots Q_0 X_0 : \bigwedge_{i=1}^{m} C_i$ be a QSAT instance, let $L_k = \{x, \bar{x} \mid x \in X_k\}$, $L_\Sigma = \bigcup_{Q_i=\exists} L_i$, $L_\Pi = \bigcup_{Q_i=\forall} L_i$. For $x \in L_k$ let $block(x) := k$.

Definition: For a formula f and a set $I \subset \{1, \ldots, m\}$ we define $f_I := \exists\Sigma : \bigwedge_{i \in I} C_i \cap \Sigma$.

$f_{\{1,\ldots,m\}}$ is the SAT instance f' of the test for trivial truth.

Definition: Two clauses C_i, C_j are conflict free, if for all $x \in L_\Pi$

$$x \in C_i \Rightarrow \begin{cases} \bar{x} \notin C_j \text{ or} \\ block(y) > block(x) \forall y \in (C_i \cup C_j) \cap L_\Sigma \end{cases}$$

I is conflict free if for all $i, j \in I$ C_i and C_j are conflict free.

Lemma: Let f be a QSAT instance, let $I \subset \{1, \ldots, m\}$ be conflict free. If f_I is not satisfiable, then f is not satisfiable.

The lemma above can be proven by induction on the number of variables of f. Simplify f without deleting or reordering the clauses. Then, if I is conflict free for f, I is conflict free for $f_{[x=0]}$ and $f_{[x=1]}$ for the outermost variable x of f. Furthermore, $(f_{[x=0]})_I = (f_I)_{[x=0]}$ and $(f_{[x=1]})_I = (f_I)_{[x=1]}$.

The generating of a conflict free clause set I with a maximum number of clauses can be shown to be computationally equivalent to the Maximum Independent Set problem, which is NP-complete in general. The function TrivialFalsity first determines a conflict free set of clauses I by a greedy approximation and then evaluates f_I by using QSOLVE. The use of TrivialFalsity is controlled by a history heuristic similar to the one described for TrivialTruth. Moreover, the test which has been used successfully most recently in the search process is performed first.

We tested a version of QSOLVE using TrivialFalsity on a set of 9500 randomly generated formulae of the form $\exists\forall\exists - 150 - L4$, i. e. formulae with 150 variables (50 $\forall$-quantified) and clause length four. The number of clauses varied from 300 to 650 in steps of 2. For each number of clauses we evaluated 50-100 formulae. The table below presents us with the average, minimum, and maximum savings in terms of recursions and running time. The net decrease of the running time is 11.57 %. The minimal savings occur at formulae with 648 or 302 clauses resp., whereas the maximum savings occur at formulae with 324 clauses.

Savings	Rec %	(#cls)	Time %	(#cls)
Average	35.26 %		11.57 %	
Minimum	16.36 %	648	-7.00 %	302
Maximum	59.98 %	324	31.44 %	324

PQSOLVE: The Distributed Algorithm

We first describe a general framework to search trees in parallel.

The Basic Algorithm

The basic idea of our distributed QSAT-solver is to decompose the search tree and search parts of it in parallel. This is organized as follows: Initially, processor 0 starts its work on the input formula. All other processors are idle. Idle processors send requests for work to a randomly selected processor. If a busy processor P gets a request for work, it checks on whether or not there are unexplored parts of its search tree waiting for evaluation. The unexplored parts are rooted in the right siblings of the nodes of P's current search stack. On certain conditions, which will be described later, P sends one of these siblings (a formula) to the requesting processor Q. P is now the master of Q, and Q the slave of P. Upon receiving a node v of the search tree, Q starts a search below v. After having finished its work, Q sends the result back to P. The master-slave relationship is released and Q is idle again. The result is used by P as if P had computed it by itself, i. e. the stack is updated and the search below the father of v is stopped, if a cutoff occurs (see the conditions for a cut in QSOLVE). The message which contains the result is interpreted by P as a new request for work. If, upon receiving a request for work, a processor is not allowed to send a subproblem, it passes the request to a randomly selected processor. Whenever P notices that a subproblem sent to another processor Q may be cut off, it sends a cutoff message to Q, and Q becomes idle.

In distributed systems messages may be delayed by other messages. It may happen, that messages refer to nodes that are no longer active on the search stack. Therefore, for every node v a processor generates a locally unique ID. This ID is added to every message concerned with v. All messages received are checked for validity. Messages that are no longer valid are discarded.

The load balancing is completely dynamic: a slave Q of a master P may itself become master of a processor R. However, if a processor P has evaluated the result for a node v, but a sibling of v is still under evaluation at a processor Q, P has to wait until Q finishes its search, since the result of the father of v depends on the result of Q.

The nodes searched for the solution of the SAT (TTH, TFH) instances are not distinguished from the nodes searched for the solution of the original QSAT instance. Therefore, the tests are done in parallel too.

The above is a general framework for parallel tree search. In the next section we will describe in detail our scheduling methods in order to cope with the problems that arise when searching QSAT trees:

- In general, a busy processor has a search stack with more than one right sibling available for a parallel evaluation. Upon receiving a request for work, it has to decide on which subproblem (if any) to send to the requesting processor.
- A processor that waits for the result of a slave is doing nothing useful. We describe a method for getting it busy while it is waiting.
- At nodes that correspond to $\exists$-quantified ($\forall$-quantified) variables a cutoff occurs, if the left branch evaluates to true (false). In this case the right branch is not evaluated by the sequential algorithm. However, in the parallel version, both branches may be evaluated at the same time. In this case the parallel version may do considerably more work than the sequential one. We describe a method for delaying parallelism, in order to reduce the amount of useless work.

The scheduling heuristics presented in the next sections are not needed to prove the correctness of the parallel algorithm, but rather to improve its efficiency.

Scheduling

The **selection of subproblems** that are to be sent upon receiving a request is supposed to fulfill the following requirements:

- The subproblem is supposed to be large enough to keep the slave busy for a while. Otherwise, the communication overhead increases since at least two messages (the subproblem itself and the request for work / result) have to be sent for every subproblem. In general, the size of a search tree below a node v is unpredictable. However, the subtrees rooted at nodes higher in the tree are typically larger than the ones rooted at the nodes deeper in the tree.
- The heuristic to select a literal for the branching process of QSOLVE selects a variable to branch with and then decides on which branch is searched first. The intention is to prove (un)satisfiability first at nodes that correspond to $\exists$- ($\forall$-) quantified variables. A perfect heuristic selects subproblems such that both siblings must be evaluated to get the final result. Our heuristic to select subproblems prefers the variables x such that both literals $x, \bar{x}$ appear equally often in the formula.

Formally, let $v_0, \ldots, v_m$ be the nodes of the search stack of a processor P. Let $x_0, \ldots, x_m$ be the boolean variables that correspond to $v_0, \ldots, v_m$. Upon receiving a request, P sends the highest right son of v_i such that $3 \cdot |N(x_i) - P(x_i)| + i$ is minimized, where $P(x_i) = \sum_j p_j(x_i)$ and $N(x_i) = \sum_j n_j(x_i)$ (see section "Literal Selection").

In order to avoid masters having to wait for slaves we have proposed the **Helpful Master Scheduling** (HMS) for a distributed chess program (Feldmann et al. 1990): Whenever a processor P waits for its slave Q to send a result, P sends a special request for work to Q. Q handles this request like a regular request. If Q does not send a subproblem, P will keep waiting for the result of Q. If Q sends a subproblem to P, it will be guaranteed that the root of the subproblem is deeper in the tree than the node where P is waiting. P then behaves like a regular slave of Q. Later, if Q waits for its slave P, the protocol is repeated with Q as the master and P as the slave. The termination of this protocol is guaranteed since the depth of the overall search tree is limited by the number of variables. While supporting its slave a processor P handles messages concerned with upper parts of the search tree as P would do while waiting for Q. A cutoff message requires the deletion of several HMS-shells from the work stack.

Since the search trees of QSOLVE are binary, the avoidance of waiting times is crucial for the efficiency of our parallel implementation.

Another problem arises when two processors P and Q search two siblings v_0, v_1 of a node v in parallel, but the result of v_0 cuts off the search below v. In this case the work done for the search below v_1 is wasted. Since the load balancing is fully dynamic a considerable amount of work which is avoided by the sequential program is done by the parallel one. For a distributed chess program we use the **Young Brothers Wait Scheduling** (YBWS) (Feldmann et al. 1990) to avoid irrelevant work. YBWS states that the parallel evaluation of a right ("younger") sibling may start only after the evaluation of the leftmost sibling has been finished. With the help of the YBWS the parallelism at a node v is delayed until at least one successor of v is completely evaluated. By the use of the YBWS the parallel search performs all cutoffs produced by the result of the evaluation of the leftmost son. However, in binary trees such as the ones searched for QSAT, this would lead to a sequential run. Therefore, in PQSOLVE we apply YBWS to blocks of variables. The subtrees that correspond to a block X have $2^{|X|+1} - 1$ nodes. For each of these subtrees, the parallel evaluation of these nodes is delayed until the leftmost leaf is evaluated.

Results

QSAT instances: The results are taken from a set of 48 QSAT instances. These instances have been generated randomly according to the model A by (Gent and Walsh 1999). The number of variables is about 120, the clause length is four, the number of blocks range from two to five. The fraction of $\forall$-variables is 25 %, the number of clauses varies from 416 to 736. The instances are hard in the sense that all sequential QSAT-solvers mentioned in this paper need considerable running times to solve them.

Hardware: Experiments with PQSOLVE are performed on the PSC2-cluster at the Paderborn Center for Parallel Computing. Every processor is a Pentium II/450 MHz running the Solaris operating system. The processors are connected as a 2D-Torus by a Scali/Dolphin CluStar network.

The effects of HM-scheduling have been studied by running PQSOLVE on the PSC-cluster, a machine with Pentium II/300 MHz processors and Fast-Ethernet communication. The communication is implemented on the basis of MPI.

Efficiency:

P	time(s)	SPE	work %
1	1594.60	1.00	0.00
32	43.29	36.84	-32.13
64	24.44	65.25	-30.30
128	13.99	114.02	-29.55

The table above presents us with the data from the parallel evaluations (averaged over 48 QSAT instances). As can be seen, the overall speedup is about 114 on 128 processors. The high efficiency is due to the fact that PQSOLVE needs about 30 % less recursions than QSOLVE (fourth column), i.e. the parallel version does less work than the sequential one. The result is a "superlinear" speedup $(SPE(P) > P)$ on several instances. This effect has already been observed for SAT (Speckenmeyer, Monien, and Vornberger 1987) but is surprising for QSAT.

The main reason for this effect is the fact that the trees searched by QSOLVE are highly irregular due to the tests for trivial truth and trivial falsity. The load balancing supports the parallelism in the upper parts of the tree. A considerable amount of work can be saved by searching two sons of a node in parallel: The one that would have been searched second by QSOLVE delivers a result that cuts off the first branch, or, both branches would deliver a result cutting off the other one, but the branch considered first by QSOLVE is harder to evaluate than the second one.

Load balancing: The table below presents us with the percentage of the running time the processors spend in the states $BUSY$ (evaluating a subtree), $WAIT$ (waiting for the result of a slave), COM (sending or responding to messages), and $IDLE$ (not having any subproblem at all).

P	forks	$BUSY$	$WAIT$	COM	$IDLE$
1	0.0	100.00	0.00	0.00	0.00
32	2107.5	83.86	8.87	3.75	0.80
64	4017.6	77.29	9.23	8.71	3.65
128	7413.2	69.00	10.16	16.58	2.08

The second column reveals the average number of subproblems that are sent during an evaluation process. The work load of the sequential version is 100 % by definition. The scheduling works well, resulting in an average work load of 69 % for 128 processors.

HM-scheduling: A crucial point for the evaluation of binary QSAT-trees is the HM-scheduling. Although HM-scheduling increases the number of subproblems that are sent by a factor of about four, it reduces the waiting times from 31.38% to less than 9 % for 32 processors. The tables below present us with results obtained from running our 48 QSAT instances on the PSC-cluster. Note that the communication of the PSC-cluster is significantly slower than the one of the PSC2-cluster used for the experiments above.

	time(s)	SPE	work
¬HM	92.64	22.61	-27.72
HM	70.72	29.61	-29.20

	forks	$BUSY$	$WAIT$	COM	$IDLE$
¬HM	566.1	52.03	31.38	13.23	1.52
HM	2185.6	69.42	8.75	18.73	0.71

YBW-scheduling: The YBW-scheduling has two effects: Firstly, as intended, the number of recursions is frequently decreased on instances with sublinear speedup. Secondly, the number of recursions is increased on many instances with superlinear speedup. In total YBW-scheduling has nearly no effect.

Conclusions and Future Work

We presented QSOLVE, a QSAT-solver that uses most of the techniques published for other QSAT-solvers before. In addition, we have implemented an adaptive heuristic to decide on when to use the expensive tests for trivial truth and trivial falsity. Moreover, QSOLVE benefits from a new test for trivial falsity.

We parallelized QSOLVE. The result is the parallel QSAT-solver PQSOLVE which runs efficiently on even more than 100 processors.

These encouraging results were obtained from random formulae. We are going to run PQSOLVE on structured instances in the near future. We are currently analyzing the test for trivial falsity. It may be improved by the way the conflict free set I is determined or by the use of more than one set. Moreover, this test for trivial falsity may lead to new insights into the theory of randomly generated QSAT instances.

Acknowledgment

We would like to thank Marco Cadoli for providing us with a binary of EVALUATE, Theo Lettmann for many helpful discussions, and the referees of AAAI for their constructive comments. This work has been supported by the DFG research project “Selektive Suchverfahren” under grant Mo 285/12-3.

References

Böhm, M.; and Speckenmeyer, E. 1996. A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence* 17:381–400.

Cadoli, M.; Giovanardi, A.; and Schaerf, M. 1998. An Algorithm to Evaluate Quantified Boolean Formulae. *Proc. of the 15th National Conference on Artificial Intelligence (AAAI-98)* 262–267. AAAI Press.

Feldmann, R.; Monien, B.; Mysliwietz, P.; and Vornberger O. 1990. Distributed Game Tree Search. In *Parallel Algorithms for Machine Intelligence and Pattern Recognition,* Kumar, V., Kanal, L.N., and Gopalakrishnan, P.S. eds., 66–101, Springer-Verlag.

Feldmann, R.; Monien, B.; and Mysliwietz, P. 1994. Studying Overheads in Massively Parallel MIN/MAX-Tree Evaluation. *Proc. of the 6th ACM Symp. on Parallel Algorithms and Architectures (SPAA-94)* 94–103. ACM.

Feldmann, R. 1997. Computer Chess: Algorithms and Heuristics for a Deep Look into the Future. *Proc. of the 24th Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM-97)* LNCS 1338, 511–522, Springer Verlag.

Gent, I.P.; and Walsh, T. 1999. Beyond NP: The QSAT Phase Transition. *Proc. of the 16th National Conference on Artificial Intelligence (AAAI-99)* 648–653. AAAI Press.

Gu, J.; Purdom, P.W.; Franco, J.; and Wah, B.W. 1997. Algorithms for the Satisfiability (SAT) Problem: A Survey. In *Satisfiability Problem: Theory and Applications,* Du, D., Gu, J., and Pardalos, P.M. eds. DIMACS 35, 19–151. American Mathematical Society.

Kleine-Büning, H.; Karpinski, M.; and Flögel, A. 1995. Resolution for quantified boolean formulas. *Information and Computation,* 117:12–18.

Rintanen, J. 1999. Improvements to the Evaluation of Quantified Boolean Formulae. *Proc. of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99),* 1192–1197. Morgan Kaufman.

Speckenmeyer, E; Monien, B; and Vornberger, O. 1987. Superlinear speedup for parallel backtracking. *Proc. of the International Conference on Supercomputing (ICS-87)* LNCS 385, 985–993, Springer-Verlag.