ANGEWANDTE MATHEMATIK UND INFORMATIK
UNIVERSITÄT ZU KÖLN

Report No. 94.159

**A Fast Parallel SAT–Solver —
Efficient Workload Balancing**

by

Max Böhm, Ewald Speckenmeyer

1994

to appear in:
Annals of Mathematics and Artificial Intelligence

Institut für Informatik
Universität zu Köln
Pohligstr. 1
D–50969 Köln

A Fast Parallel SAT–Solver — Efficient Workload Balancing

Max Böhm[†], Ewald Speckenmeyer
Institut für Informatik
Universität zu Köln
Pohligstr. 1, D–50969 Köln
e–mail: boehm@informatik.uni–koeln.de, esp@informatik.uni–koeln.de

Abstract: We present a fast parallel SAT–solver on a message based MIMD machine. The input formula is dynamically divided into disjoint subformulas. Small subformulas are solved by a fast sequential SAT–solver running on every processor, which is based on the Davis–Putnam procedure with a special heuristic for variable selection. The algorithm uses optimized data structures to modify boolean formulas. Additionally efficient workload balancing algorithms are used, to achieve a uniform distribution of workload among the processors. We consider the communication network topologies d–dimensional processor grid and linear processor array. Tests with up to 256 processors have shown very good efficiency–values (> 0.95).

1 Introduction

The satisfiability problem of boolean formulas in conjunctive normal form (SAT–problem) was the first problem to be shown to be NP–complete, see [2]. Since that time the SAT–problem has attracted the interest of many researchers. This is due to its simple structure on the one side. On the other side the SAT–problem has several important applications in the area of logic–programming, fault testing of switching circuits, etc. Therefore it is important to have algorithms, which are able for solving a wide range of instances of the SAT–problem in tolerable time. We know classes of instances of the SAT–Problem which can be solved in linear time. The class of Horn formulas, [4], and the 2–SAT formulas, [2], e.g., or the formulas with the implication as the only operator and with every variable occurring twice, [7]. Classes of instances of the SAT–problem have been studied in order to show proof systems like resolution to be exponential time provers for these classes, as the pigeonhole formulas, [6], or Tseitin's graph formulas, [11]. The satisfiability test of these instances is hard for certain proof systems only, but not for a human solver, who knows in advance — due to an understanding of the idea behind the construction principle of the formulas — whether they are satisfiable or not. Thus random formulas form a really hard class of formulas when asking whether they are satisfiable or not.

We now have a lot of knowledge about the average time complexity or the probabilistic behavior of algorithms, when solving certain parametrized classes of random instances, see [9], [5]. Despite of many attempts in this direction, for a wide range of instances the

[†]This research was supported by the Federal State of Nordrhein–Westfalen in the *Forschungsverbund Paralleles Rechnen*, Az.: IV A 3 - 107 021 91 -

SAT–problem remains intractable from an experimental point of view. We have no idea how to test these formulas efficiently for satisfiability.

Here we are essentially interested in solving hard random instances. We first have developed a Davis–Putnam based satisfiability solver. The quality of such a solver heavily depends on the ability of predicting, which of the unset literals should be set true next in order to hold the search space as small as possible. By experiments with k–SAT formulas (all clauses have a fixed length k) with varying ratios of clauses and variables we have singled out a strategy essentially computing for every unset literal a vector weighting the occurrences of the literal and its complement in clauses of length $i = 2, 3, \ldots, k$. A literal with highest vector under the lexicographic order will be chosen next, see chapter 2. Each step of truth setting a literal, and consequently of its complementary literal, causes an update of the current subformula. This update should be as efficient as possible, i.e. a good data structure for representing formulas is needed. In order to be able to represent any boolean formula in conjunctive normal form, we have used a purely pointer based data structure. This data structure needs linear storage space and allows for an optimal execution of nearly all data structure operations, which have to be performed by our algorithm, see chapter 3. The implementation of this algorithm turned out to be the fastest program among 35 programs in a SAT competition organized by Hans Kleine Büning from the University of Paderborn, see [1].

In order to be able to further speed up our algorithm, we have implemented it on a parallel computer, a Transputersystem built up from up to 256 processors (INMOS T800/4MB). Transputersystems enable the programmer to realize every net–topology with the restriction that every processor is connected with at most four other processors. All experiments, which are reported here, where run on 2-dimensional grids of up to 16×16 processors and on linear processor arrays. We want to mention here in advance, that a SPARC 10 processor works at least 10 times faster than a T800 processor. What we are interested in and what we have achieved however, is to speed up our sequential SAT–solver by a factor of nearly $1/N$ by running a copy of it on each of the N processors and distributing the workload between the processors in a convenient way. Thus, if we were able to substitute each T800 processor by a SPARC 10 processor, the reported runtimes for the N processor Transputernetwork would be at least 10 times faster for a SPARC 10–network.

As mentioned above, our parallel SAT–solver runs a copy of the sequential SAT–solver on every processor, and the N processors cooperate, when searching for a solution in the space of partial truth assignments, by partitioning this space. That way each processor will be provided with a certain amount of workload represented by its subspace of partial truth assignments. The most difficult part, which has to be solved when running the SAT–solver on a parallel computer, consists of balancing the workload between the processors in such a way that on the one side idle times of processors should almost always be avoided, on the other side the workload balancing phase should consume as few computing time as possible. This is a nontrivial task. Our parallel algorithm redistributes workload between the processors at certain points of time. This is necessary, because no reliable estimation of workload represented by a partial truth assignment, which has not yet been singled out not to lead to a satisfying truth assignment, is known. We have used as workload

estimation the function α^n for a partial truth assignment with n unset variables, for varying values of α between 1.04 and 1.42, depending on the parameters of the class of formulas from which the input formula is chosen.

The workload redistribution phase is activated, if the estimated workload for some processor goes down some limit. Then, in case of a linear processor array, the following steps are performed. In the first step the prefix sums of the workload estimations for all processors are determined for the linear array of N processors. Then in a second step the last processor, which knows the total amount L of estimated workload and the number N of processors, broadcasts the ratio $\mu = L/N$, i.e. the amount of workload for each processor in case of a uniform workload distribution among the N processors, to all processors. Finally in a third step, each processor p knows due to the knowledge of μ and its rank in the linear processor array, i.e. how many processors are to the left of p, whether workload has to be sent from p to its left (right) neighbor, or whether p has to receive workload from its left (right) neighbor. These three steps can be performed in time $O(N)$. The workload redistribution phase for rectangular grids is achieved by first performing the above workload balancing phase for all linear arrays of processors linked in the first dimension and then for all linear arrays of processors linked in the second dimension. I.e., in case of square grids with N processors the workload distribution phase can be performed in time $O(\sqrt{N})$.

We have run many experiments with our parallel SAT–solver with N=1,16,32,64,128, and 256 processors, see chapter 6. We want to stress the point, that we have obtained an efficiency close to 1. Obtaining these results is not a trivial task, and it requires a lot of tuning.

2 Sequential SAT–Solver

We consider boolean formulas in conjunctive normal form (CNF). The following notations are used:

- Let V be a set of n boolean *variables*.
- $X = \{x, \overline{x} \mid x \in V\}$ is called the set of *literals*.
- $c = x_1 \vee \ldots \vee x_k$ with $x_i \in X$ is called a *clause*. A clause is a disjunction of literals. The clause length $|c|$ of this clause is k. A clause is not allowed to contain both, a literal and its complement.
- $F = c_1 \wedge \ldots \wedge c_m$, with clauses c_i, is called a *formula* in conjunctive normal form. We define $|F| := \sum_{c \in F} |c|$ to be the total number of literals in F.
- A *truth assignment* $A \subset X$ is a set of literals with $\forall x \in A : \overline{x} \notin A$. F_A denotes the formula we obtain from F by assigning the value $true$ to all literals $x \in A$ and the value $false$ to their complements. We write F_x as a shorthand for $F_{\{x\}}$.

The satisfiability test of a CNF–formula F is performed by the following algorithm $Solve$, which is a variant of the Davis–Putnam procedure, see [3], with a special heuristic for variable selection. We already described this algorithm briefly in [1]. The input formula F_I is satisfiable iff $Solve(F_I)$ returns $true$.

function $Solve(F)$

1. if F is empty then return $true$
2. if F contains the empty clause then return $false$
3. if F contains a unit clause (x) then return $Solve(F_x)$
4. select a literal x for branching according to the lexicographic heuristic
   if $Solve(F_x)$ then return $true$
   else return $Solve(F_{\overline{x}})$

A formula F without any clauses is satisfied by definition and if F contains the empty clause it is unsatisfiable thus justifying steps 1 and 2 of $Solve$.

For reasons of efficiency the formula should be simplified as much as possible before branching. A powerful simplification is the *unit clause rule* or unit resolution which is done in step 3. A clause of length 1 (unit clause) containing the literal x forces the assignment $x = true$. The simplified formula is solved recursively. This strategy is implemented with the following slight modification to speed up the program: Before an assignment based on the unit clause rule is made, the formula is checked for two complementary unit clauses (y), $(\overline{y})$. In this case the formula is determined to be unsatisfiable.

We want to mention that we have not included the *pure literal rule* into our SAT–solver, because it slowed down the run time of the algorithm in our experiments.

Step 4 is the branching step. A literal x is chosen according to a special heuristic, which is described below. First the value $true$ is assigned to x and the resulting subformula F_x is solved recursively. If no solution is found the value $false$ is assigned to x and the resulting formula $F_{\overline{x}}$ is solved recursively. F is satisfiable iff at least one of the two subformulas F_x and $F_{\overline{x}}$ is satisfiable.

The idea behind the *lexicographic heuristic* is to assign a value to a literal occurring as often as possible in the shortest clauses of the formula. This way the length of the shortest clauses is often reduced by one, which will result in clauses of length 1 after a few steps. So the formula collapses fast.

A literal x with maximal vector $(H_1(x), H_2(x), \ldots, H_n(x))$ under the lexicographic order is chosen, where

$$H_i(x) = \alpha\, max(h_i(x), h_i(\overline{x})) + \beta\, min(h_i(x), h_i(\overline{x})),$$

and $h_i(x)$ is the number of clauses of length i, containing x.

If $\alpha = \beta = 1$ the function simplifies to $H_i(x) = h_i(x) + h_i(\overline{x})$, which is equal to the number of positively and negatively occurrences of literal x in clauses of length i. In order

to get subproblems of about the same size, we want to prefer literals which do not differ too much in $h_i(x)$ and $h_i(\overline{x})$. Experiments have shown that the choice of $\alpha = 1$ and $\beta = 2$ often leads to shorter run times.

Note that $H_i(x) = H_i(\overline{x})$. After having determined the literal x we proceed first with that subformula of F_x and $F_{\overline{x}}$ which has the fewest number of clauses.

In our implementation we calculate and compare two elements $H_s(x)$ and $H_{s+1}(x)$ of the vector only, where s is the length of the shortest clauses of the formula. This improves the efficiency of the calculation. The size of the search trees thus generated does not change significantly due to this simplification.

3 Data Structures

Beside a good branching heuristic, which keeps the search tree small, an efficient data structure for representing formulas is important. The run time of a SAT–algorithm will be slowed down orders of magnitude due to a poor implementation of the underlying data structures. This is also reflected by the results of a SAT competition described in [1].

The design of suitable data structures depends on the operations, which should be supported efficiently. Our data structure stores a boolean formula F, which is initially the input formula. The basic operation consists of modifying F according to a truth assignment of a variable. This operation is executed at each edge of the search tree. Another operation, which is executed at each node of the search tree consists of looking for unit clauses. The data structure should be chosen in order to support efficient execution of these operations.

Having a look at the shape of the search tree, we see that most nodes (especially the leaves) represent subformulas of small size compared to the size of the input formula. Therefore any subformula should be stored and accessed as efficiently as the input formula, i.e. the run time for a 'linear time' operation should be linear in the size of the current subformula represented by the data structure and not in the size of the input formula.

A backtracking algorithm has to remember subproblems not yet evaluated (i.e. subformulas). If this is done by copying the whole subproblem the run time of this step will be at least linear in the size of the subproblem. Additionally, for each subproblem, which is copied new memory space for storing the problem is needed. This typically leads to a quadratic space requirement. In order to avoid this inefficiency we use the following approach: The data structure represents only one formula which is modified in situ. The operation $assign(x)$ modifies F to F_x. The removed parts of the formula (i.e. satisfied clauses, removed literals) are linked on a stack. In case of a backtrack the formula will be reconstructed by the reverse operation $unassign(x)$ which modifies F_x to F using the stack.

Our implementation of *assign* and *unassign* performs both operations in time $O(|F| - |F_x|)$. Direct access to all clauses containing x and all clauses containing $\overline{x}$ is therefore supported. Unit clauses are detected in time $O(1)$. We have implemented the following

forward and backward chained list structures:

- The formula is stored as a list of clauses (ordered by clause length). Direct access to parts of the formula with constant clause length k is supported.
- A clause is represented by a clause head and a list of its literals.
- For each literal x a list of clauses containing x exists (literal occurrence list).

An example of the data structure is shown in figure 1.

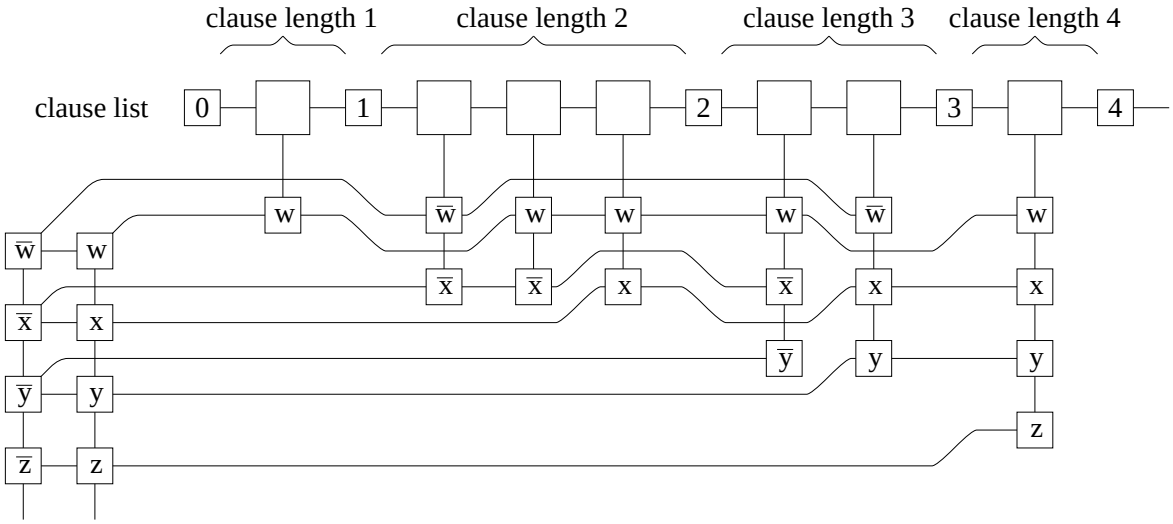

Figure 1: Data structure representing $F = (w) \wedge (\overline{w} \vee \overline{x}) \wedge (w \vee \overline{x}) \wedge (w \vee x) \wedge (w \vee \overline{x} \vee \overline{y}) \wedge (\overline{w} \vee x \vee y) \wedge (w \vee x \vee y \vee z)$

The operation of calculating the lexicographic heuristic is executed for *branching nodes* (nodes of out–degree 2) of the search tree only. Experiments have shown that the number of branching nodes is very small compared to the total number of nodes in the search tree (about 3% for hard random 3–SAT formulas, i.e. for formulas with a clause–variable ratio of 4.3). This operation needs time $O(|F|)$ in our implementation. We accepted this time, although some bookkeeping could speed this operation, but this would increase the running time of the *assign* and *unassign* operations.

A literal x and its complement $\overline{x}$ are treated as inverse elements only. The knowledge which of them is positive and which is negative is not important, because simple renaming of literals should not influence the behavior of the program.

For random k–SAT formulas with a fixed ratio r of clauses and variables the average number of occurrences of a literal x is equal to kr. For these formulas the operations *assign* and *unassign* need constant time in the average.

Table 1 summarizes the execution times of the operations performed on the data structure.

operation	running time
$assign(x)$, $unassign(x)$	$O(\|F\| - \|F_x\|)$
find clause of length k	$O(1)$
find clause c with $x \in c$	$O(1)$
find literal of clause c	$O(1)$
calculate lexicographic heuristic	$O(\|F\|)$

Table 1: Time complexity of operations performed on the data structure

We have implemented the operations $assign(x)$ and $unassign(x)$ as described below: When assigning $x = true$, we have to remove all clauses $c = (\cdots x \cdots)$ containing x. These clauses can be found immediately by looking up the *literal occurrence list* for x. Each literal knows its clause head. The clause c and all literals of its *literal list* with exception of x are unlinked as shown in figure 2. Finally we unlink literal x from the list of unassigned literals.

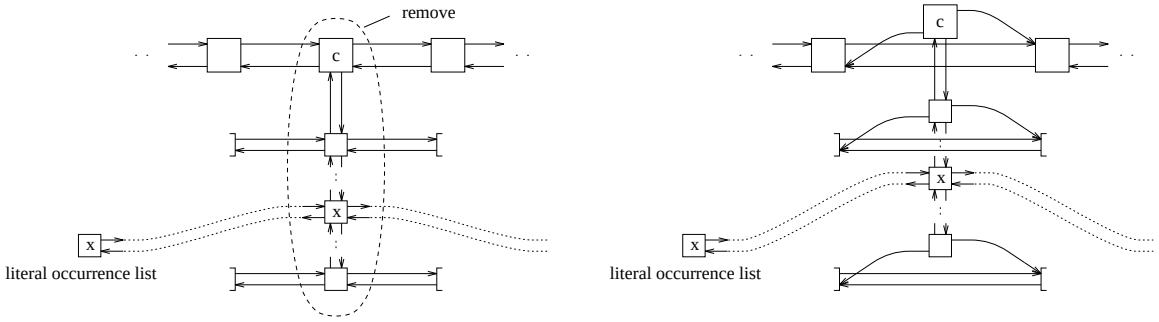

Figure 2: Remove clauses $c = (\cdots x \cdots)$ in time $O(|c|)$

In the second step we shorten all clauses $c = (\cdots \overline{x} \cdots)$ containing literal $\overline{x}$. Using the *literal occurrence list* for $\overline{x}$ we find all clauses containing $\overline{x}$. The literal $\overline{x}$ is unlinked and its clause head is moved to the sublist of clauses of length $i-1$ if the clause length of c was i before this operation. This is shown in figure 3. Finally we unlink literal $\overline{x}$ from the list of unassigned literals.

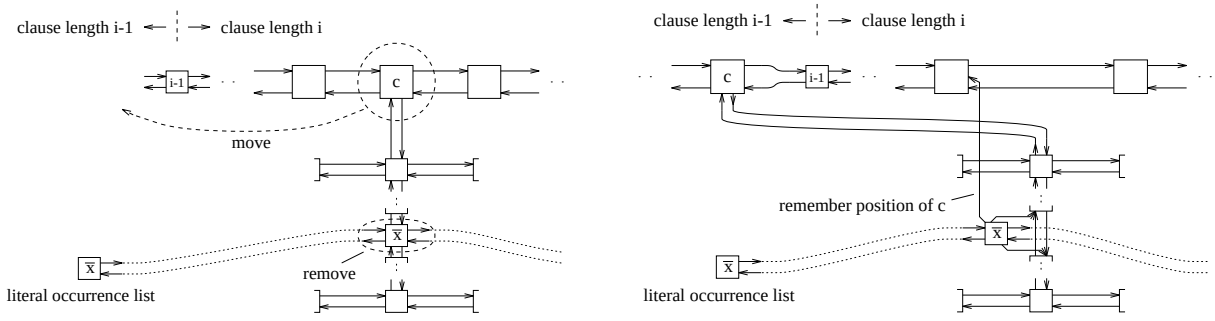

Figure 3: Remove literal $\overline{x}$ from clauses $c = (\cdots \overline{x} \cdots)$ in time $O(1)$

To reverse these operations, we follow the *literal occurrence list* of x and $\overline{x}$ and link the removed clauses and literals into the current formula by using the old pointers of those elements, which are kept valid to find their original locations in the formula. This is done exactly in reverse order to the unlinking operation as described above and it leads to the original formula.

4 Workload Balancing

Given a set P of N processors and a communication network. At some fixed point in time every processor $p \in P$ holds a workload (WL) $\lambda(p) \in \mathbb{R}^+$, which is an estimation for the time needed to solve the problems placed on p. In the following we assume, that WL is divisible in infinitely small pieces, which can be exchanged between processors. A processor p is allowed to send some of its WL to a neighbored processor q if p and q are linked by the network.

The workload balancing problem (WLB) consists of exchanging WL between processors resulting in a *uniformly distributed* WL, i.e. $\forall p, q \in P : \lambda(p) = \lambda(q)$. The following algorithm solves the problem for a linear array of N processors $p_1, \ldots, p_N$ in time $O(N)$. The main task of the algorithm is to determine the amount of WL $l(p_i)$, which has to be exchanged between a processor p_i and its right neighbor p_{i+1} to achieve a uniform distribution of WL.

WLB algorithm for linear arrays

Every processor $p_i \in P$ performs the following steps:

1. calculate *prefixsum* $\hat{\lambda}(p_i) := \lambda(p_1) + \cdots + \lambda(p_i)$
2. p_N calculates *optimal* WL $\mu := \frac{\hat{\lambda}(p_N)}{N}$; broadcast μ to all processors
3. calculate overload $l(p_i) := \hat{\lambda}(p_i) - i\mu$ and $l(p_{i-1}) := l(p_i) - (\lambda(p_i) - \mu)$
4. if $l(p_i) > 0$ then send WL $l(p_i)$ to p_{i+1} else receive WL $|l(p_i)|$ from p_{i+1}
   if $l(p_{i-1}) > 0$ then receive WL $l(p_{i-1})$ from p_{i-1} else send WL $|l(p_{i-1})|$ to p_{i-1}

The algorithm starts with a precomputation phase. Every processor p_i determines $\hat{\lambda}(p_i)$ which is the total WL of the processors $p_1, \ldots, p_i$. This is done in $N-1$ steps from p_1 to p_N. Processor p_N calculates the optimal workload $\mu := \frac{\hat{\lambda}(p_N)}{N}$ and broadcasts it to all processors in $N-1$ steps.

In step 3 every processor p_i calculates the overload or underload $l(p_i)$ of the processors $p_1, \ldots, p_i$, which has to be balanced over the link between p_i and p_{i+1} and the overload or underload $l(p_{i-1})$ of the processors $p_1, \ldots, p_{i-1}$, which has to be balanced over the link between p_{i-1} and p_i.

Some send/receive steps of step 4 need a special sequence of executions, because a processor p, which has to send a WL greater than $\lambda(p)$ may has to wait for receipt of

sufficient WL from other processors first. Nevertheless the above WLB–algorithm achieves a uniform distribution of WL among the processors within $N-1$ steps of moving WL, which can easily be seen to be optimal. Particularly we want to stress the point that the algorithm achieves a uniform distribution of WL by moving the smallest possible amount of WL among the processors, which is important in case of WL consisting of huge data–packets.

WLB algorithm for d–dimensional m–sided grids

The algorithm can be easily extended to d–dimensional m–sided grids ($N = m^d$) using the above WLB algorithm for linear arrays for each dimension. This leads to a run time $O(d \cdot m)$:

for $k := 1$ to d do
  $\forall i_j \in \{1, \ldots, m\}$, $1 \leq j \leq d$, $j \neq k$ do in parallel
    execute the WLB algorithm for the linear array with
    processor set $\{p_{i_1 \cdots i_k \cdots i_d} \mid i_k \in \{1, \ldots, m\}\}$

Note that d–dimensional hypercubes are d–dimensional 2-sided grids. Therefore the above algorithm solves the WLB problem for d–dimensional hypercubes in time $O(d)$.

5 Parallel Implementation

The input formula is divided into subformulas in the same way as in the sequential case. The generated subformulas represent workload, which has to be distributed among the processors s.t. all processors have the same amount of workload if possible. Small subformulas are solved by the sequential SAT–solver, which is running as a process on all processors.

To determine an estimation of the workload function λ we have measured the run time of our sequential SAT–solver using samples of random 3–SAT formulas with a clause and variable ratio of 5, for varying numbers of variables n. Each sample consisted of 50 unsatisfiable instances. The average run times of the experiments are indicated by the solid line in figure 4, which grows approximately like the function 1.04^n. Therefore we defined the workload of a subformula F_A to be $\lambda(F_A) := \alpha^{n-|A|}$ with $\alpha = 1.04$ for the input formulas F with n variables.

We did similar experiments with random k–SAT formulas, for $k = 3, \ldots, 6$, and a clause–variable ratio of r_k, where r_k was chosen such that experimentally about 50% of all formulas from a sample were unsatisfiable. Considering the unsatisfiable formulas of the samples, only, we determined the growth α_k^n of the average search trees corresponding to these formulas. The results are summarized in table 2.

A subformula F_A is represented by a string A' of literals, which is a subset of the partial truth assignment A. The string A' only contains those literals of A, which are chosen in the branching step of the algorithm. Variables whose truth assignments are forced by the unit clause rule will not be represented. This is sufficient for achieving a

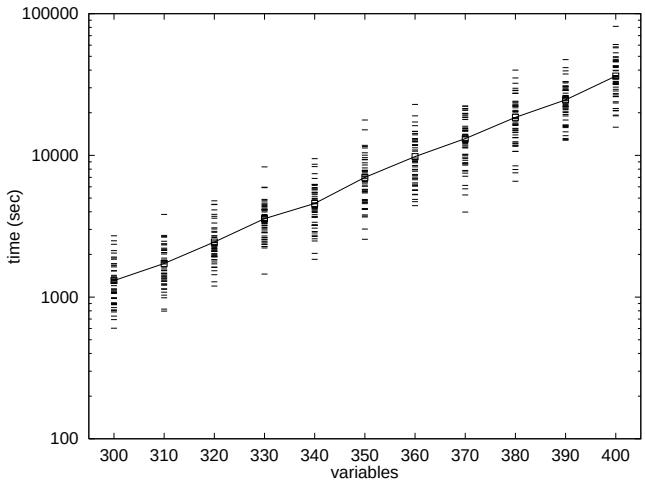

Figure 4: Sequential run times (T800 processor) for random 3–SAT formulas, clauses/variables=5.0, each sample contains 50 instances

	r_k	α_k	b_k
3–SAT	4.3	1.04	3%
4–SAT	9.9	1.11	6%
5–SAT	21.1	1.18	10%
6–SAT	43.7	1.25	14%

Table 2: Workload estimation parameter α_k for random k–SAT formulas, b_k is percentage of branching nodes with respect to the size of the search tree

unique reconstruction of F_A from A'. A processor is able to send A' to a neighbored processor consuming little communication time, only.

In the initialization phase the input formula F is sent to all processors in the network. Each processor p holds a list L_p of strings A' representing subformulas F_A to be solved by p. At the beginning one list contains the empty string representing the input formula F, all other lists are empty.

Each processor runs two processes in parallel, the worker and the balancer.

The *worker process* tries to split or solve subformulas of the list. It takes a string A' representing the subformula F_A from the top of the list. If the list is empty the worker process waits for either new work or a termination message of the balancer process. If the workload $\lambda(F_A)$ goes down a certain limit, the subformula is solved by the sequential SAT–Solver. Otherwise F_A is split into two disjoint subformulas assigning $true$ resp. $false$ to a literal x which is chosen according to the lexicographic heuristic. The strings $A'x$ and $A'\overline{x}$ representing these subformulas are inserted at the top of the list. This processing is repeated until it is terminated by the balancer process.

A subformula F_A is solved by the fast sequential SAT–Solver if $\lambda(F_A) < \beta\lambda(p)$, where $\lambda(p) := \sum_{A' \in L_p} \lambda(F_A)$. We have chosen $\beta = 0.05$ in order to hold a sufficient number of subformulas in the list. This is especially useful during the ending phase of the computa-

tion when only small subformulas are residing in the lists.

The *balancer process* of each processor p performs the following steps periodically: First the estimated workload $\lambda(p)$ of p is sampled. Based upon this information the precomputation phase of the WLB algorithm is performed, which calculates the amount of WL to be sent or received by each processor. Additionally the following information is broadcasted:

- All lists are empty (termination)?
- A solution was found?
- At least one list contains less than s subformulas?

The balancing of WL between processors is actually performed only if at least one processor holds less than s problems in its list to reduce communication overhead. We have chosen $s = 3$. In case of a balancing activity the processors try to resolve their send and receive requests of workload in parallel. If a processor has a send request, but its list contains not enough workload it may have to wait for receipt of WL from other processors first. To execute a send request of WL of some size l the balancer process takes problems from the bottom of the list, sends them to the neighbored destination processor, and reduces l appropriately. This is repeated until $l < \delta\mu$. Workload needs not to be balanced exactly. Therefore we have chosen $\delta = 0.5$ to reduce communication activity.

6 Experimental Results

We implemented the parallel SAT–solver in "C" on a parallel MIMD machine of the University of Paderborn, called Supercluster SC320. It consists of 320 T800 transputers each of 4 Mbyte local memory. A T800 transputer is a 32 bit processor from Inmos of a peak performance of 20 MIPS. It includes 4 serial links of 20 Mbits/sec for communication. The Supercluster SC320 allows to configure any network topolgy of maximal degree 4 by software. The connections between processors are established physically by crossbar switches. Each transputer runs its program code in its local memory.

The T800 this chip exists since 1987, therefore the performance of a single transputer is slow. A SUN Sparc Station 10 is about 10 times faster. Nevertheless the measured speedup values indicate the usefulness of a parallel approach.

6.1 Random k–SAT Formulas

The first class of test formulas we used were randomly generated k–SAT formulas of n variables and m clauses. A clause is generated by randomly selecting k different variables. Each of these variables is then negated with probability 0.5. m clauses are generated independently. This generation scheme ensures no double or complementary occurring literals in a clause, but it allows double clauses in the formula. It is not guaranteed that all n variables occur in the formula.

We first have generated a sample of 50 random 3–SAT formulas of 400 variables and 2000 clauses. All test formulas were unsatisfiable. Table 3 shows the average run times and standard deviation in seconds for the network topologies *linear array* and *2–dimensional grid*. Average *speedup*– and *efficiency*–values are shown in figure 5. The average speedup–value is calculated by dividing the sum of run times of the 1–processor algorithm by the sum of run times of the N–processor algorithm. The efficiency–value is the speedup–value divided by the number of processors. The speedup of the network topology *grid* is nearly linear. Even for the network topology *linear array* with large diameter an acceptable speedup was measured. This indicates the usefulness of the grid algorithm for much more than 256 processors.

linear array					2–dimensional grid				
proc.	time	std. dev.	speedup	eff.	proc.	time	std. dev.	speedup	eff.
1	36295	12479			1	36295	12479		
16	2296	785	15.8	0.99	4×4	2307	789	15.7	0.98
32	1152	392	31.5	0.98	4×8	1154	394	31.5	0.98
64	583	197	62.3	0.97	8×8	579	197	62.7	0.98
128	301	99	120.6	0.94	8×16	292	99	124.3	0.97
256	167	52	217.3	0.85	16×16	150	50	242.0	0.95

Table 3: Sample: 50 unsatisfiable random 3–SAT formulas, 400 variables, 2000 clauses

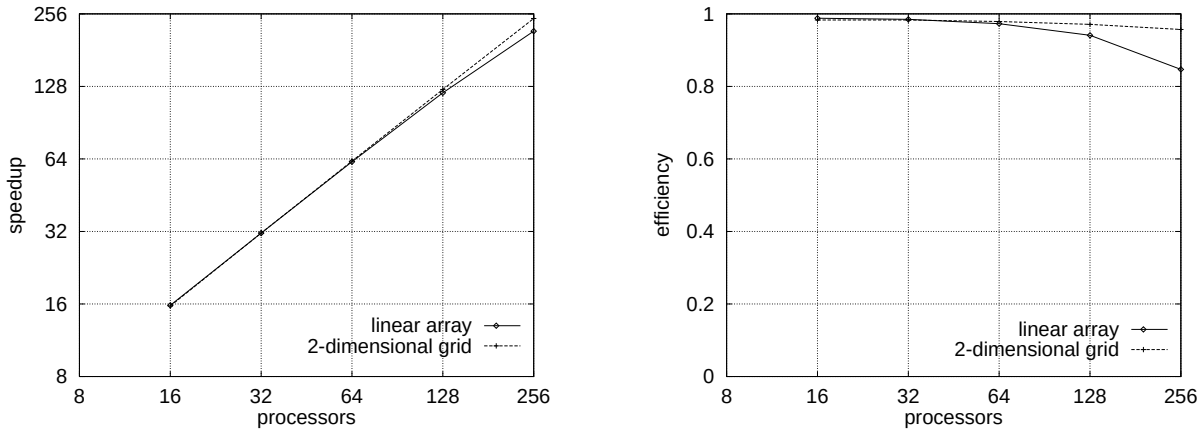

Figure 5: average speedup– and efficiency values

We also generated a sample of 50 hard random 3–SAT formulas of 350 variables and 1505 clauses (ratio 4.3 of clauses and variables). 19 formulas were satisfiable and 31 formulas were unsatisfiable. The results are shown in table 4. Each field contains a value for the unsatisfiable formulas atop a value for the satisfiable formulas. The network topology *2–dimensional grid* was chosen. Figure 6 shows the run times in seconds and speedup–values for each instance by a small dash. The dash is drawn left from the vertical grid if the formula is satisfiable and right from the vertical grid, if the formula is unsatisfiable. The speedup is nearly linear for unsatisfiable formulas. For satisfiable formulas the speedup–values varies very much. Note that the average speedup may be more than linear which indicates, that the sequential algorithm can be improved, see [10].

proc.	sat?	time	std. dev.	speedup	eff.	branching nodes
1	31 no	126139	52703			6092071
	19 yes	39009	33119			1904555
16	31 no	7981	3326	15.8	0.99	6092071
	19 yes	1503	1544	26.0	1.62	1152119
32	31 no	3990	1661	31.6	0.99	6092071
	19 yes	846	1147	46.1	1.44	1295830
64	31 no	1997	830	63.2	0.99	6092071
	19 yes	520	779	75.0	1.17	1593679
128	31 no	1002	415	125.9	0.98	6092071
	19 yes	251	468	155.4	1.21	1542045
256	31 no	504	208	250.3	0.98	6092071
	19 yes	113	147	345.2	1.35	1378098

Table 4: Sample of 50 random 3–SAT formulas, 350 variables, 1505 clauses

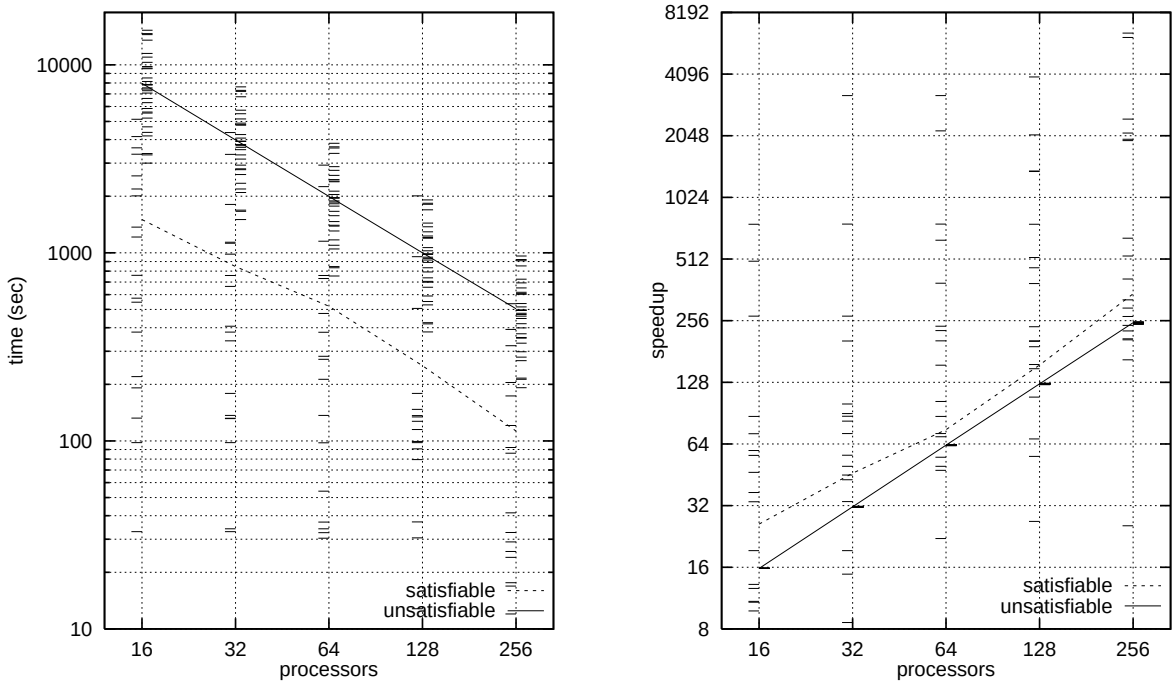

Figure 6: average run times and speedup–values

Next we generated samples of 50 hard random k–SAT formulas for $k \in \{3, 4, 5, 6\}$. We experimentally determined ratios between clauses and variables which result in about 50% satisfiable instances. The average run times for a 16×16 processor grid in contrast to the run times of one processor are shown in table 5. Additionally the number of branching nodes of the search tree and the standard deviation are shown.

k	n	m	$\frac{m}{n}$	sat?	$time_{256}$	$time_1$	speedup	eff.	br. nodes	std. dev.
3	350	1505	4.3	31 no	504.1	126139	250.2	0.98	6092071	2606870
				19 yes	112.8	39009	346.0	1.35	1378098	1801701
4	130	1285	9.9	25 no	214.8	53287	248.1	0.97	2967599	479921
				25 yes	47.3	14755	312.0	1.22	674080	583775
5	80	1690	21.1	28 no	166.4	40781	245.1	0.96	1866775	126520
				22 yes	68.5	15998	233.7	0.91	798736	491517
6	60	2620	43.7	27 no	183.3	43644	238.1	0.93	1546258	64021
				23 yes	56.9	16164	284.1	1.11	503170	511303

Table 5: Hard random k–SAT formulas, efficiency of 256–processor algorithm

6.2 Graph Formulas

The second class of test formulas consists of Tseitin's graph formulas as defined in [11]. These unsatisfiable formulas seem to be very hard to be solved by Davis–Putnam based SAT–solvers. We briefly describe how to generate these formulas. We randomly choose an undirected graph $G = (V, E)$ of f nodes and each node of degree d. Each node $v \in V$ is randomly labeled by $p(v) \in \{0, 1\}$ so that $\sum_{v \in V} p(v) \equiv 1 \ (mod\ 2)$. A unique literal is assigned to each edge of G. For each node $v \in V$ we generate all 2^{d-1} possible clauses c involving literals incident with v such that the number of complemented literals in c is opposite in parity to $p(v)$. The obtained d–SAT formulas consists of $n = df/2$ variables and $m = 2^{d-1}f$ clauses.

We have generated 4 samples of 10 unsatisfiable random graph formulas each. The parameter α of the workload approximation has to be adjusted for this class of formulas. The test results are shown in table 6.

f	d	n	m	$\frac{m}{n}$	$time_{256}$	$time_1$	speedup	eff.	branching nodes	α
48	3	72	192	2.67	172.1	42301	245.8	0.96	38803864	1.11
24	4	48	192	4	129.2	31324	242.4	0.95	34393291	1.18
16	5	40	256	6.4	122.8	29868	243.3	0.95	33554431	1.25
12	6	36	384	10.67	137.7	33581	243.8	0.95	33973861	1.42

Table 6: unsatisfiable graph formulas, efficiency of 256–processor algorithm

Acknowledgement: We would like to thank H. Stamm–Wilbrandt for helpful discussions on the branching heuristic and F. Meisgen for implementing parts of the parallel algorithm.

References

[1] Buro M., Kleine Büning H.: *Report on a SAT competition*, EATCS Bulletin, No 49, Feb 1993, 143–151

[2] Cook, S.A.: *The Complexity of Theorem–Proving Procedures* , Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, 1971, 151–158

[3] Davis, M. and Putnam, H.: *A Computing Procedure for Quantification Theory*, J. Assoc. Comput. Mach., 7, 1960, 201–215.

[4] Dowling, W. and Gallier, J.: *Linear–Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae*, J. Logic Programming, 3, 1984, 267–284.

[5] Franco J.: *Elimination of Infrequent Variables Improves Average Case Performance of Satisfiability*, SIAM J. Comput. 20, 1991, 1119–1127

[6] Haken A.: *The Intractability of Resolution*, Theor. Comput. Sci., 39, 1985, 297–308

[7] Heusch P.: *Implikationen der Implikation*, Dissertation, Mathematisches Institut, Heinrich–Heine–Universität–Düsseldorf, 1993

[8] Monien, B. and Speckenmeyer, E.: *Solving Satisfiability in less than* 2^n *Steps*, Discrete Applied Mathematics, 10, 1985, 287–295.

[9] Purdom P.W. Jr., Haven N.G.: *Backtracking and Probing*, Indiana University Computer Science Technical Report No. 387, 1993

[10] Speckenmeyer, E.: *Is Average Superlinear Speedup Possible?*, Proc. CSL'88, Springer–Verlag (LNCS 385), 1989, 301–312.

[11] Tseitin G.S.: *On the Complexity of Derivation in Propositional Calculus*, A.O. Slisenko, ed., Studies in Constructive Mathematics and Mathematical Logic, Part II (translated from Russian) (Consultants Bureau, New York, 1970), 115–125