A Multi-Processor Approach to SAT-Problems

Christian Posthoff
The University of The West Indies
St. Augustine Campus
Trinidad & Tobago
email:cposthoff@fsa.uwi.tt

Bernd Steinbach
Freiberg University of Mining and Technology
Institute of Computer Science
D-09596 Freiberg, Germany
email:steinb@informatik.tu-freiberg.de

Abstract

The 3-SAT problem is one of the most important and interesting NP-complete problems with many applications in different areas. In several previous papers and in [5] we showed the use of ternary vectors and set-theoretic considerations as well as binary codings and bit-parallel vector operations in order to solve this problem. After the parallelism of the solution process has been established on the register level, i.e. related to the existing hardware, this article shows the extension to the use of several processors working in parallel. Based on the previous considerations this extension is easy to understand. Simultaneously it should be used as an inspiration to implement such a multi-processor solution and to apply it to many different problems.

1 Introduction

Boolean problems combine practical and theoretical aspects in a special manner. Konrad Zuse [8] revealed the practical importance that two different values of a binary variable can be easily distinguished in a technical machine. This understanding led to the breakthrough of modern computer science. The first fully functional electromechanical digital computer in the world (the Z3) was completed by Zuse in 1941. Recently, the binary representation and processing of information is used in more and more areas of our life. Computers can process 64-bit numbers directly in a single step. Programs, data, music and movies are stored on CDs or DVDs using binary values. Phone calls, faxes, files and pictures are transmitted as sequences of binary values using ISDN or the Internet, and the traditional analogous techniques of radio or TV will be substituted by digital systems having a much higher quality and requiring less resources.

In addition to the theories of Boolean Algebras and the Boolean Differential Calculus [5], certain Boolean problems appear as a key in complexity theory [7]. The *Boolean satisfiability problem (SAT)* was the first known *NP-complete problem.* A decision problem is in the complexity class NP if a non-deterministic Turing machine can solve it in polynomial time. A decision problem is NP-complete if it is in NP and if every problem in NP can be reduced [3] to it by a polynomial-time reduction. Stephen Cook proved that the Boolean satisfiability problem is NP-complete [1]. This theorem was independently proven by Leonid Levin [4] at about the same time, so that it is called Cook-Levin theorem. NP-complete problems are the most interesting problems in NP. In the context of the practical importance mentioned above we suggest in this paper an approach to solve large SAT problems in acceptable time.

2 Preliminaries

In order to make this note independently readable, we summarize the concepts that have been used previously. Let $\mathbf{x} = (x_1, \cdots, x_n), x_i \in \{0, 1, -\}, i = 1, \ldots, n$. Then $\mathbf{x}$ is called a *ternary vector* which can be understood as an abbreviation of a set of binary vectors. When we replace each $-$ by 0 or 1, then we get several binary vectors generated by this ternary vector. In this way, the vector $(0-1-)$ represents four binary vectors (0010), (0011), (0110) and (0111). A list (matrix) of ternary vectors can be understood as the union of the corresponding sets of binary vectors.

A Boolean expression is said to be *satisfiable* if binary values 0 or 1 can be assigned to its variables in a way that makes the expression equal to 1 (true). Both a variable and its negation are called *literals*. A disjunction of literals is called *clause*. A Boolean expression is given in *conjunctive form* if it only consists of clauses connected by AND-operators. The check for satisfiability (SAT) is NP-complete if the Boolean expression is given in conjunctive form with three or more variables in its clauses. In the special case of the 3-SAT problem each clause includes three literals.

If we consider now one clause of a 3-SAT problem, such as $C = x_2 \vee \overline{x}_3 \vee x_4$, then we find all solutions of $C = 1$ in the following way:

- If $x_2 = 1$, then $C = 1$ independent on the other variables. Hence, $(-1--)$ describes a set of solutions.
- If $x_2 = 0$, then $\overline{x}_3 = 1$, i.e. $x_3 = 0$ gives more solutions that are not covered by the previous case and described by $(-00-)$.
- Finally, if $x_2 = 0$ and $x_3 = 1$, then x_4 must be equal to 1. Thus, (-011) is another set of solutions.

If we consider now two clauses $C_1C_2 = 1$, then we build the solution sets of $C_1 = 1$ and $C_2 = 1$ and find the solution of $C_1C_2 = 1$ as intersection of the respective solution sets.

The intersection will be computed according to Table 1 which has to be applied in each component. The $\emptyset$ indicates that the intersection is empty and can be omitted.

Table 1: Intersection of Ternary Values

x_i	0	0	0	1	1	1	$-$	$-$	$-$
y_i	0	1	$-$	0	1	$-$	0	1	$-$
$x_i \cap y_i$	0	$\emptyset$	0	$\emptyset$	1	1	0	1	$-$

A sophisticated coding of the three values 0, 1 and $-$ allows the introduction of binary vector operations that can be executed on the level of registers (32, 64 or even 128 bits in parallel). We use the coding of Table 2.

Table 2: Binary Code of Ternary Values

ternary value	bit1	bit2
0	1	0
1	1	1
$-$	0	0

When the three-valued operations for the intersection are transferred to these binary vectors, then the empty intersection can be determined by

$$bit1(\mathbf{x}) \wedge bit1(\mathbf{y}) \wedge (bit2(\mathbf{x}) \oplus bit2(\mathbf{y})) \neq 0.$$

If the intersection is not empty, then it can be determined by the following bit vector operations:

$$bit1(\mathbf{x} \cap \mathbf{y}) = bit1(\mathbf{x}) \vee bit1(\mathbf{y}), \quad bit2(\mathbf{x} \cap \mathbf{y}) = bit2(\mathbf{x}) \vee bit2(\mathbf{y}).$$

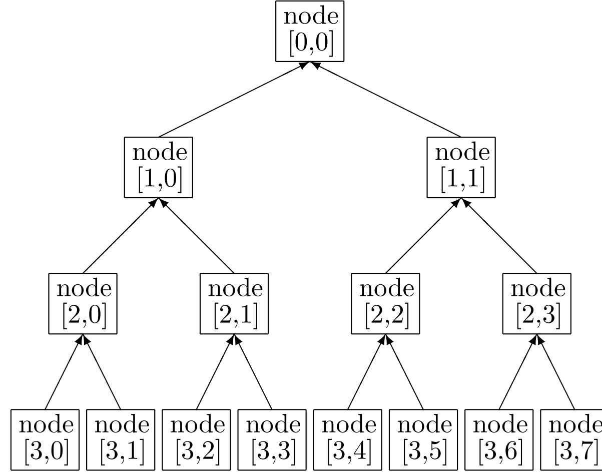

Figure 1: 4-Level Tree Structure of Processor Nodes

Hence, by using some very fast and very simple bit vector operations, we can find the solution sets of any SAT-equations, and especially of 3-SAT-equations. In [2] this could be done for a maximum of up to 280 variables in about 10 minutes.

3 Using Several Processors

3.1 Divide the Set of Clauses

There are several approaches to distribute the check for satisfiability to more than one processor. First we try to divide the given set of clauses. In order to do this we assume that a given equation $F(x_1, \ldots, x_n) = 1$ is represented in 3-SAT-format, i.e. by $C_1C_2 \ldots C_k = 1$, and each C_i is a disjunction of three literals. The only information that has to be known to all processors is the number n of variables, because this number defines the length of the vectors that has to be used by all processors.

Now we can assume that we have a given number of processors. These processors are arranged in a tree structure, where each 'father processor' is connected to two 'son processors'. A complete tree of k levels requires $2^k - 1$ processors. Figure 1 shows an example of 15 processor nodes. This number is only taken as an illustration, any other (larger) number will be even better. To simplify matters we assume first a complete tree. The same program runs on each processor node.

We use one processor as the root of the tree in level 0. In an initial phase this root processor splits the original expression in the middle into two sets of clauses. The first set of clauses is forwarded as task to the son processor on the left side, and the second set will be submitted as task to the son processor on the right side, respectively. The recursive procedure of splitting and transmitting subtasks will stop at the processors associated to the leaf nodes of the tree.

At the topmost *Level 0* the original equation will be split into two parts:

$$f(x_1, \ldots, x_n) = f(\mathbf{x}) = f_1(\mathbf{x})f_2(\mathbf{x}) = 1.$$

It is well known that the single equation $f_1(\mathbf{x})f_2(\mathbf{x}) = 1$ can be replaced by an equivalent system of equations:

$$f_1(\mathbf{x}) = 1, \; f_2(\mathbf{x}) = 1.$$

Each partial equation will be allocated to one dedicated processor of the next level. The processors of the leaf nodes in the tree cannot divide and forward the task to be solved anymore. Hence, the leaf processors solve the received partial equation according to the principles described above. For each clause a list of ternary vectors is created which describes the solution set of the associated clause. Subsequently the intersection of all solution sets of different clauses will be calculated iteratively. This processing of the partial equations and the generation of

the partial solutions can be done completely in parallel, no information exchange whatsoever between different processors is required.

The calculation of the intersection in the leaf processors is organized in such a way that, as soon as solutions of the partial equation of one processor have been found, these partial solutions are immediately forwarded to the next higher level for further processing. Empty intersections help to speed up the calculations. Each solution vector of a partial task will be transmitted from the son processor to its father processor through a piped stream. In Figure 1 each pair of processors of Level 3 submits its partial solutions to one processor of Level 2. Here another very important parallelism can be observed. The processors at Level 2 can start working immediately after the respective processors of Level 3 have found their first partial solution(s).

The processors on the intermediate levels calculate the intersection of all pairs of vectors found by their son processors. All partial solution vectors created by one son processor are orthogonal to each other. If the intersection between one partial solution vector of the left son processor and one partial solution vector of the right son processor is not empty, then this partial solution vector is transmitted immediately to the father processor using a piped stream. If such a solution vector is found in the root processor, then the satisfiability has been proven, and the calculation process can stop. Another reason for stopping all calculations is that one of the processors does not find any solution. In that case there is no vector which satisfies the given Boolean equation. If a son processor finishes its work, it sends a signal to its father processor such that the father processor can decide about the end of its own work. In Algorithm 1 the transmission of an empty set is taken as such a signal.

In order to simplify the algorithmic structure, Algorithm 1 assumes a complete binary tree of processors. Generally any partial binary tree of processors can be adapted such that a father processor with only one son processor emulates the missing son processor by an additional thread.

3.2 Divide the Set of Potential Solutions

In this approach we change our point of view slightly. Using the well-known negation of De Morgan (NDM), the given Boolean equation $CNF(f) = 1$ can be transformed into $DNF(f) = 0$ in constant time. Unfortunately, the solution of this equation finds all vectors that are not an element of the desired solution. Thus we need the complement (CPL) of the solution set which creates the required solution set of $CNF(f) = 1$. All solution vectors S that satisfy $CNF(f) = 1$ can be calculated by

$$S \leftarrow \texttt{CPL}(\texttt{NDM}(CNF(f))) \quad (1)$$

The main part of the work is required for the CPL-operation. In the previous approach equation (1) was applied to the two parts $(CNF(f_1), CNF(f_2)) = CNF(f)$, therefore an additional intersection ISC must be executed:

$$S \leftarrow \texttt{ISC}(\texttt{CPL}(\texttt{NDM}(CNF(f_1))), \texttt{CPL}(\texttt{NDM}(CNF(f_2)))) \quad (2)$$

Equation (2) shows that in the previous approach two large intermediate solution sets are calculated, and the final solution set must be built in a time-consuming step by means of the ISC-operation.

Alternatively, equation (1) offers an iterative solution method. A first partial solution set S_1 can be calculated from the first clause C_1. Note: no solution vector can exist outside the partial solution set S_1. The next clause C_2 restricts this set S_1 - the set of vectors $\texttt{NDM}(C_2)$ that are not a solution must be removed using a difference operation DIF. At the end of such an iteration the final solution S_n is calculated using the previous partial solution S_{n-1} and the last clause C_n:

$$\begin{aligned}
&S_1 \leftarrow \texttt{CPL}(\texttt{NDM}(C_1)) \\
&S_2 \leftarrow \texttt{DIF}(S_1, \texttt{NDM}(C_2)) \\
&\ldots \\
&S_n \leftarrow \texttt{DIF}(S_{n-1}, \texttt{NDM}(C_n)).
\end{aligned}$$

The set of binary vectors that can be generated by using S_i is a superset of the set of binary vectors that can be generated by using S_{i+1}, i.e. this sequence of sets is monotone with regard

Algorithm 1 Satisfiability of CNF be: `TreeSat`$(BooleanExpression\ be, Level\ l)$

Require: complete tree of processors; Boolean expression be, given in CNF; Level $l \geq 0$

Ensure: satisfiability of $be = f(\mathbf{x})$

```
1: if there are two son processors then {not a leaf processor}
2:   {initial phase}
3:   (be_1, be_2) ← split(be)
4:   left.TreeSat(be_1, l + 1)
5:   right.TreeSat(be_2, l + 1)
6:   {final phase}
7:   SL ← ∅
8:   SR ← ∅
9:   repeat
10:    SLN ← RecieveFromLeftSon()
11:    SL ← SL ⋃ SLN
12:    S ← SLN ⋂ SR
13:    if S ≠ ∅ then {there is a solution}
14:      if l = 0 then {root processor}
15:        return true {the given expression is satisfiable}
16:      else
17:        father.Transmit(S)
18:      end if
19:    end if
20:    SRN ← RecieveFromRightSon()
21:    SR ← SR ⋃ SRN
22:    S ← SRN ⋂ SL
23:    if S ≠ ∅ then {there is a solution}
24:      if l = 0 then {root processor}
25:        return true {the given expression is satisfiable}
26:      else
27:        father.Transmit(S)
28:      end if
29:    end if
30:  until (SLN = ∅) ∧ (SRN = ∅)
31:  if l = 0 then {root processor}
32:    return false {the given expression is not satisfiable}
33:  else
34:    father.Transmit(S)
35:  end if
36: else {calculate basic partial solution}
37:  for all clause C[i] of be do
38:    S[i] ← ClauseSolutionSet(C[i])
39:  end for
40:  sv ← ClauseIntersetion(NEW)
41:  while sv ≠ ∅ do
42:    father.Transmit(sv)
43:    sv ← ClauseIntersetion(CONTINUE)
44:  end while
45:  father.Transmit(sv)
46: end if
```

to $\supseteq$. The number of ternary vectors, however, which are required to describe these partial solution sets can grow from S_i to S_{i+1}.

Such a large number of orthogonal vectors in a partial solution set suggests another new approach: allocate the check for satisfiability to more than one processor. We can split a given partial solution set S_i into to parts $S1_i$ and $S2_i$. Both parts of the next partial solution set $S_{i+1} = (S1_{i+1}, S2_{i+1})$ can be calculated in parallel on different processors:

$$S1_{i+1} \leftarrow \texttt{DIF}(S1_i, \texttt{NDM}(C_{i+1})),$$
$$S2_{i+1} \leftarrow \texttt{DIF}(S2_i, \texttt{NDM}(C_{i+1})).$$

This approach has the advantage that the parts of each pair of partial solution sets are orthogonal to each other, and this means that the solution vectors can be combined by `CON` in constant time:

$$S_{i+1} \leftarrow \texttt{CON}(S1_{i+1}, S2_{i+1}).$$

Algorithm 2 applies this approach recursively. The partial solution ps of the initial invocation of `RecSat` algorithm is the set $(- - - - -...-)$ that covers all possible solutions. The limit is an empirical integer value used to decide whether a partial solution set will be split or not. The created subtasks can run in separate threads. The allocation of the threads to the available processors can be organized by the operation system. Thus any number of processors can be used in this approach. Algorithm 2 is written in such a way that the complete solution set is calculated. It is easy to modify it for checking satisfiability only.

Algorithm 2 Set that satisfies CNF be: `RecSat`$(PotSolution\ ps, BooleanExpr.\ be, Limit\ l)$

Require: any set of processors; nonempty set of potential solutions ps, given in DNF; Boolean expression be, given in CNF; Limit $l > 0$

Ensure: solution set that satisfies $be = f(\mathbf{x})$

1: {initial phase}
2: **while** $(l > \texttt{NTV}(ps)) \wedge (be \neq \emptyset)$ **do**
3: $\quad clause \leftarrow \texttt{SelectFirstClause}(be)$
4: $\quad be \leftarrow \texttt{DeleteFirstClause}(be)$
5: $\quad ps \leftarrow \texttt{DIF}(ps, \texttt{NDM}(clause))$
6: **end while**
7: **if** $be = \emptyset$ **then** {all clauses evaluated}
8: $\quad$ **return** ps {the given expression be is satisfiable if $ps \neq \emptyset$}
9: **end if**
10: {recursive phase}
11: $(ps_1, ps_2) \leftarrow \texttt{split}(ps)$
12: $s_1 \leftarrow \texttt{RecSat}(ps_1, be, l)$ {run in thread 1}
13: $s_2 \leftarrow \texttt{RecSat}(ps_2, be, l)$ {run in thread 2}
14: $s \leftarrow \texttt{CON}(s_1, s_2)$ {solved in constant time}
15: **return** s {the given expression be is satisfiable if $s \neq \emptyset$}

---

4 Order of Clauses

4.1 Lexicographic Order

The order of clauses in the given expression does not change the solution, but has a strong influence on the calculation efforts. Thus a useful introductory step is a sorting of the clauses. We study first the effect of sorting the clauses according to a lexicographic order. We use in a first step $\overline{x}_1 < x_1 < -$. This means that we write down first all clauses with $\overline{x}_1$ in the first component, followed by all clauses with x_1 in the first component, thereafter all clauses with $-$ in the first component. If two clauses have the same first component, then the second component is considered in the same way etc. This ordering will keep the number of intermediate ternary vectors to be considered rather "small".

If we have two clauses of three literals with x_1 in the first component, then the corresponding ternary matrices will have 100 in the first column, and we see the following situation for the intersection:

$$\begin{pmatrix} 1 & - & \cdots \\ 0 & - & \cdots \\ 0 & - & \cdots \end{pmatrix} \cap \begin{pmatrix} 1 & - & \cdots \\ 0 & - & \cdots \\ 0 & - & \cdots \end{pmatrix} \Rightarrow \cdots$$

If we have two clauses of three literals, one has x_1, the second has $\overline{x}_1$ in the first component, then we can see the following situation:

$$\begin{pmatrix} 1 & - & \cdots \\ 0 & - & \cdots \\ 0 & - & \cdots \end{pmatrix} \cap \begin{pmatrix} 0 & - & \cdots \\ 1 & - & \cdots \\ 1 & - & \cdots \end{pmatrix} \Rightarrow \cdots$$

In the first situation the intersection will have at most five ternary vectors, in the second case even only four, instead of nine that would be possible in principle.

4.2 Rearrange the Order of Clauses according to the Influence of Variables

The lexicographic order is once again improved in Algorithm 3. The larger the number of literals of one variable in the whole intermediate matrix, the stronger is the improvement based on its lexicographic order. Thus Algorithm 3 counts first the number of literals for each variable of the Boolean expression and implements the lexicographic sorting according to these values starting with the high values.

Algorithm 3 Rearrange order of CNF *be*: `InfVarOrder`(*BooleanExpression be*)

Require: Boolean expression $be = f(\mathbf{x})$, given in CNF
Ensure: $ben = f(\mathbf{x})$ arranged in such a way that the first clause includes the variable of the highest occurrence in the CNF *be* and the last clause includes the variable of the lowest occurrence in the CNF *be*

1: **for all** *var* in *be* **do**
2: $NL[var] \leftarrow$ `NumberOfLiterals`(be, var)
3: **end for**
4: $ben \leftarrow \emptyset$
5: **while** $be \neq \emptyset$ **do**
6: $var_{max} \leftarrow (var|NL[var] \geq max_{allvar}(NL[var])$
7: **for all** *clause* in *be* **do**
8: **if** var_{max} in *clause* **then** {move clause}
9: $ben \leftarrow$ `CON`$(ben, clause)$
10: $be \leftarrow$ `DTV`$(be, clause)$
11: **end if**
12: **end for**
13: $NL[var_{max}] \leftarrow 0$
14: **end while**
15: **return** *ben*

4.3 Rearrange Order of Clauses by Number of Used Variables

It follows from the structure of clauses in 3-SAT that some calculations can be done in subspaces. The intersection of two disjoint sets of variables leads to a solution set, where the calculated number of ternary vectors is equal to the product of the numbers of ternary vectors in the given sets. If the two sets of variables are equal to each other, the intersection creates a smaller number of ternary vectors.

Algorithm 4 is based on this fact. In the same way as Algorithm 3, Algorithm 4 counts first the number of literals for each variable of the Boolean expression. A second step calculates a weight for each clause. This weight is the sum of values calculated in the first step associated

to the variables of the clause. In a third step the clauses will be rearranged according to these weights. The first clause will have the highest weight, and the next clauses are selected so that they have the same set of variables, thereafter two variables in common, thereafter one variable in common, and the clauses with disjoint sets of variables are taken last.

Algorithm 4 Rearrange order of CNF *be*: `UsedVarOrder`(*BooleanExpression be*)

Require: Boolean expression $be = f(\mathbf{x})$, given in CNF
Ensure: $ben = f(\mathbf{x})$ arranged so that the growth of the set of variables used from the first to the i-th clause is as small as possible.

1: **for all** *var* in *be* **do**
2: $NL[var] \leftarrow$ `NumberOfLiterals`(be, var)
3: **end for**
4: $w_{max} \leftarrow 0$
5: **for all** *clause* in *be* **do**
6: $WC[clause] \leftarrow 0$
7: **for all** *var* in *clause* **do**
8: $WC[clause] \leftarrow WC[clause] + NL[var]$
9: **end for**
10: **if** $w_{max} < WC[clause]$ **then** {memorize new best clause}
11: $w_{max} \leftarrow WC[clause]$
12: $clause_{max} \leftarrow clause)$
13: **end if**
14: **end for**
15: $ben \leftarrow clause_{max}$
16: $be \leftarrow$ `DTV`$(be, clause_{max})$
17: **while** $be \neq \emptyset$ **do**
18: $soc_{best} \leftarrow 0$
19: **for all** *clause* in *be* **do**
20: $cover \leftarrow$ `SVISC`$(be, clause)$
21: $soc \leftarrow$ `SVSIZE`$(cover)$
22: **if** $soc = 3$ **then** {clause completely covered}
23: $ben \leftarrow$ `CON`$(ben, clause)$
24: $be \leftarrow$ `DTV`$(be, clause)$
25: **else if** $soc > soc_{best}$ **then**
26: $soc_{best} \leftarrow soc$
27: $clause_{best} \leftarrow clause$
28: **end if**
29: **end for**
30: $ben \leftarrow$ `CON`$(ben, clause_{best})$
31: $be \leftarrow$ `DTV`$(be, clause_{best})$
32: **end while**
33: **return** *ben*

5 Small Detailed Example

We will show a small example that will simply illustrate the behaviour to be expected. For larger numbers of processors the improvements to be achieved in comparison to a one-processor system will still be much more remarkable. However, since the processors on one level can work fully in parallel, it can be expected that the time t required on each level is reduced to $\frac{t}{k}$ if k processors are available on the level.

Let us see the following equation:

$$(x_1 \vee x_2 \vee x_3)(\overline{x_2} \vee x_4 \vee x_5)(\overline{x_3} \vee \overline{x_4} \vee x_5)(\overline{x_1} \vee \overline{x_3} \vee \overline{x_5}) = 1.$$

We try to find one solution as fast as possible using one processor. After four steps the first vectors of the respective ternary matrices are available:

$$(1----)(-0---)(--0--)(0----) \rightarrow 4 \ steps.$$

Three intersections are required to combine these vectors from the left to the right:

$$\rightarrow (10---)(--0--)(0----) \rightarrow (100--)(0----) \rightarrow \emptyset \rightarrow 7 \textit{ steps}.$$

Since the last intersection is empty, in the fourth matrix the second vector must be generated and used which results in two more operations:

$$(100--)(1-0--) \rightarrow (100--) \rightarrow 9 \textit{ steps}.$$

Thus this depth-first method finds the first solution after 9 steps if one processor is used.

For finding all solutions the four ternary matrices of the four disjunctions will be used.

$$\begin{pmatrix} 1 & - & - & - & - \\ 0 & 1 & - & - & - \\ 0 & 0 & 1 & - & - \end{pmatrix} \begin{pmatrix} - & 0 & - & - & - \\ - & 1 & - & 1 & - \\ - & 1 & - & 0 & 1 \end{pmatrix} \begin{pmatrix} - & - & 0 & - & - \\ - & - & 1 & 0 & - \\ - & - & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & - & - & - & - \\ 1 & - & 0 & - & - \\ 1 & - & 1 & - & 0 \end{pmatrix}$$

This requires twelve steps. After nine steps we get the intersection of the first and the second matrix which is equal to

$$\begin{pmatrix} 1 & 0 & - & - & - \\ 1 & 1 & - & 1 & - \\ 1 & 1 & - & 0 & 1 \\ 0 & 1 & - & 1 & - \\ 0 & 1 & - & 0 & 1 \\ 0 & 0 & 1 & - & - \end{pmatrix}.$$

After another 18 intersections the first three matrices have been considered, and after 39 intersections the final solution has been found. We show here these two next matrices:

$$\begin{pmatrix} 1 & 0 & 0 & - & - \\ 1 & 0 & 1 & 0 & - \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & - \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & - \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & - \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 0 & 1 & - \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & - \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & - & - \\ 1 & 1 & 0 & 1 & - \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Thus after 78 steps all the solutions are available.

Now we will use three processors, two in parallel on the lowest level, and one to combine the results of these two on the second level.

Two steps are now required for generating the first vectors of the four matrices, because the equation is split into two parts which are processed simultaneously: $(1----)(-0---)$ and $(--0--)(0----)$ are available after these two steps, and the two intersections $(10---)$ and $(0-0--)$ require two steps on the lowest level. The processor on the higher level finds out that the intersection is empty (after four steps altogether), the generation of the second vector $(1-0--)$ requires another step, and after two more steps the first solution has been found after seven steps. Because of the size of the example the reduction from nine to seven is not so remarkable ($\approx 77.7\,\%$).

It will be much more remarkable when all solutions are required. The generation of the original four matrices requires now six steps:

$$\begin{pmatrix} 1 & - & - & - & - \\ 0 & 1 & - & - & - \\ 0 & 0 & 1 & - & - \end{pmatrix} \begin{pmatrix} - & 0 & - & - & - \\ - & 1 & - & 1 & - \\ - & 1 & - & 0 & 1 \end{pmatrix}$$

Table 3: Search Space of the Solved 3-SAT Problems

number of variables n	search space of the size 2^n
50	1,125,899,906,842,624
75	37,778,931,862,957,161,709,568
100	1,267,650,600,228,229,401,496,703,205,376

Table 4: Partial Results of Split Tasks

	number of		time in seconds				
Benchmark	variables	clauses	a0	a1l1	a1l2	a1s	a1p
uf50-218-01	50	218	134	82	27	109	82

and

$$\begin{pmatrix} - & - & 0 & - & - \\ - & - & 1 & 0 & - \\ - & - & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & - & - & - & - \\ 1 & - & 0 & - & - \\ 1 & - & 1 & - & 0 \end{pmatrix}.$$

After nine steps we have the two intermediate matrices

$$\begin{pmatrix} 1 & 0 & - & - & - \\ 1 & 1 & - & 1 & - \\ 1 & 1 & - & 0 & 1 \\ 0 & 1 & - & 1 & - \\ 0 & 1 & - & 0 & 1 \\ 0 & 0 & 1 & - & - \end{pmatrix} \text{ and } \begin{pmatrix} 0 & - & 0 & - & - \\ 1 & - & 0 & - & - \\ 0 & - & 1 & 0 & - \\ 1 & - & 1 & - & 0 \\ 0 & - & 1 & 1 & 1 \end{pmatrix}.$$

And the computation of this final intersection requires 30 steps. By adding these values we see that all solutions are available after 45 steps. However, this does not yet take into consideration that the calculation on the higher level does not have to wait until the operations on the lower level are fully finished. The reduction to 45 steps (or less) means a reduction to $\approx 57.6\%$ as expected.

It will be very interesting to see the implementation of this system in different multi-processor environments. First results are included into this paper.

Such a structure of processors can work in parallel horizontally as well as vertically, without serious restrictions by communication needs, fully in parallel. This is rather surprising because many algorithms cannot be parallelized, neither fully nor partially.

6 Experimental Results

The experimental work based on the suggested algorithms is not yet finished. The results achieved so far, however, give already useful hints for future research. We used 3-SAT problems from the SATLIB benchmark suite [9] depending on 50, 75 or 100 Boolean variables. Table 3 gives an impression of the size of the search spaces for these problems.

In a first experiment we compare the basic algorithm defined in (1) (called *a0* in Table 4) with the subtasks in two leaf processors of a simple tree. Based on the algorithm defined in (2) these subtasks are called *a1l1* and *a1l2* in Table 4. The set of clauses was cut in the middle. Table 4 shows, as expected, that the subtasks were solved faster than the complete task. When we assume that the subtasks are solved sequentially (a1s), then the required time is shorter than the time for the complete solution (a0). In the parallel approach the time for one subtask, in this example a1l1, determines the required time. Restricted to this part of the task, we get a reduction of 61%.

Unfortunately one additional task must be solved in the root processor. Because of the size of the calculated two subsolutions, the calculation of the intersection (`ISC`) in the root processor is so time-consuming that it seems to be helpful to reduce the number of rows in both subsolutions

Table 5: Complete Results of Split Tasks (Benchmark: uf50-218-01)

	number of rows			time in seconds				
Operation	a1l1	a1l2	a1r	a1l1	a1l2	a1r	a1s	a1p
`NDM` 1	109			0				
`CPL` 1	1929870			82				
`OBB` 1	602786			13660				
`NDM` 2		109			0			
`CPL` 2		3165473			26			
`OBB` 2		1014440			36962			
`ISC`			4			6168		
all together			4	13742	36988	6168	56900	43158

Table 6: Complete Results of Sorted Split Tasks using OBB (Benchmark: uf50-218-01)

	number of rows			time in seconds				
Operation	a1l1	a1l2	a1r	a1l1	a1l2	a1r	a1s	a1p
sort()			218			0		
`NDM` 1	117			0				
`CPL` 1	15128			0				
`OBB` 1	5777			1				
`NDM` 2		101			0			
`CPL` 2		363220			9			
`OBB` 2		124457			490			
`ISC`			10			7		
all together			10	1	499	7	507	506

using the algorithm (`OBB` - orthogonal block building) which combines two ternary vectors that differs only in one column by '01'. Table 5 shows in detail the number of rows calculated by the operations `NDM`, `CPL`, `OBB`, and `ISC` distributed in a three-processor tree with the root a1r and the leafs a1l1 and a1l2, respectively. Most of the time was required to reduce the number of rows approximately to one third of the previous size. It is an important advantage that the `OBB`-operations can be calculated in parallel on the leaf processors. The final `ISC`-operation benefits from the reduced size of the TVLs to be intersected and needs much less time. The comparison of the sequential (a1s) and parallel (a1p) approach of the subtasks enumerated in Table 5 offers an reduction of 75%. Admittedly this distributed calculation of the solution is a strong deterioration. The comparison of the time for a0 of Table 4 and a1p of Table 5 offers that the parallel approach needs 322 times more time than the direct solution using one processor only.

One of the reasons for this extremely bad result might be the allocation of the clauses to the processors. In order to determine this influence, we used a very simple greedy algorithm that sorts the rows before their allocation to the leaf processors. As can be seen in Table 6, the rearrangement of the rows has a very strong influence on both the number of rows in the created TVLs and the required time for the calculation. The sorting reduced the time required for solving the same task by a factor of approximately 100. The detailed study of Table 6 shows that the times required for the subtasks differ strongly. The second `OBB`-operation dominates the total time, even when this subtask solves the smaller set of clauses. It is expected that in case of *small* solution sets the `OBB`-operations can be omitted. Note: the different numbers of rows in the solutions of Table 5 (4) and Table 6 (10) comes from a different transformation of Boolean vectors to ternary vectors. In each case 24 Boolean solution vectors are covered.

Table 7 confirms this conjecture. Without the `OBB`-operations the parallel approach a1p of Table 7 requires only 34% of the basic time for a0 of Table 4 in order to solve the same task uf-50-218-01. Table 7 shows a bad load-balancing between the three processors. The main work must be done in the root processor for the `ISC`-operation. As mentioned above Algorithm 2 does

Table 7: Complete Results of Sorted Split Tasks without OBB (Benchmark: uf50-218-01)

	number of rows			time in seconds				
Operation	a1l1	a1l2	a1r	a1l1	a1l2	a1r	a1s	a1p
sort()			218			0		
NDM 1	117			0				
CPL 1	15128			0				
NDM 2		101			0			
CPL 2		363220			9			
ISC			10			37		
all together			10	0	9	37	46	46

Table 8: Second Method of Sorted Split Tasks (Benchmark: uf50-218-01)

	number of rows			time in seconds				
Operation	a2l1	a2l2	a2r	a2l1	a2l2	a2r	a2s	a2p
sort()			218			0.05		
DIF 1	10			0.03				
DIF 2		0			0.01			
CON			10			0.00		
all together			10	0.03	0.01	0.05	0.09	0.08

not need this intersection.

In order to have a good comparison we organized the next experiment as follows. After the same sorting of the clauses we

1. calculate the first partial solution $S_1 \leftarrow \texttt{CPL}(\texttt{NDM}(C_1))$,
2. split S_1 into $S1_1$ (first row) and $S2_1$ (next two rows),
3. calculate in the leaf processors the complete partial solutions

$$S1_1 \leftarrow \texttt{DIF}(S1_1, \texttt{NDM}(C_2, ..., C_n))$$
$$S2_1 \leftarrow \texttt{DIF}(S2_1, \texttt{NDM}(C_2, ..., C_n)),$$

   and

4. combine the partial solutions in the root processor: $S \leftarrow \texttt{CON}(S1_1, S2_1)$.

The comparison of the time for a0 in Table4 and a2p of Table 8 shows a speeding up of the parallel solution of more than 1000. Because of the very short periods of time to be measured we take for the following experiments larger examples.

Algorithm 2 allows to control the splitting into subtasks depending on the number of rows in the partial solution. Table 9 shows that the increase of the cut limit from $l = 200$ to $l = 500$ reduces the required solution time. The reason for that is the decrease of the time for creating the subtasks. If the cut limit is further increased from $l = 500$ to $l = 1000$, the required solution time increases too because the calculation of larger lists is more time-consuming. All subtasks in Algorithm 2 need approximately the same small time. Thus the total solution time for k processors is nearly t/k if t is the time which one processor needs to solve the task.

Finally we evaluate the influence of the rearrangement algorithms suggested in section 4. As shown in Table 10, the runtime of Algorithm 3 (called a3 in Table 10) is slightly shorter than the time for Algorithm 4 (called a4 in Table 10). The small additional effort of 0.02 seconds for the rearrangement of rows in Algorithm 3 decreases the required time in the main task of solving the 3-SAT problem by a factor of 175.

Using the combination Algorithm 4 - Algorithm 2, next larger benchmark examples of 100 variables and 430 clauses were solved. All solution vectors of the example uf100-430-01 were calculated in 38192 seconds. After 1496 seconds it was found that the example uuf100-430-01 is not solvable.

Table 9: Evaluation of the Parameter Limit l in Algorithm 2

	number of		time in seconds		
Benchmark	variables	clauses	$l = 200$	$l = 500$	$l = 1000$
uf75-325-01	75	325	34	26	27
uuf75-325-01	75	325	24	18	19

Table 10: Evaluation of the Rearrangement Algorithms

	number of		time in seconds			
Benchmark	variables	clauses	a3	a3-a2	a4	a4-a2
uf75-325-01	75	325	0.07	4569	0.09	26

7 Conclusions

The investigations in this paper have resulted in the expected observation that the required time to solve a 3-SAT problem can be reduced if several processors work together.

Several approaches for parallelisms were offered and experimentally verified. On the register level all variables up to the size of the register can be considered in parallel. The SAT problem can be split in different ways into subproblems. One of them consists in splitting the set of clauses, and another one in splitting the solution space. The subproblems can be solved independent on each other on different processors. The communication effort is small.

It has also been shown that splitting and distributing subtasks to different processors do not ensure to find the solution in shorter time. Large intermediate partial solution sets extend the total solution time enormously. Optimal local algorithms are necessary to speed up the solution process of the 3-SAT problem using several processors.

It was verified that the order of the clauses has a strong influence on the solution efforts. Algorithm 4 leads to an arrangement of clauses that restricts the partial solution set in two manners, the number of used variables and the remaining subspace.

It should also not be forgotten that 3-SAT is a NP-Complete problem. Each other NP-problem can be transformed into 3-SAT and, thereafter, be solved using this parallelization principle. It will also be very interesting to see the results after this transformation. Up to now these transformations have 'only' been used to show that different problems have the same complexity. Based on this and further implementations, however, it can (will) be the case that other NP-problems will be transformed into a SAT-problem or particularly into the 3-SAT problem in order to get an efficient solution method. This will be opening a whole new area for research and experiments as well as applications.

An extension of the 3-SAT problems to general SAT-problems is also easy to implement since this requires only a more general approach to the coding of the initial solution candidates.

References

[1] St. Cook: The Complexity of Theorem Proving Procedures. Proceedings of the third annual ACM symposium on Theory of computing, Shaker Heights, Ohio, United States, 1971, pp. 151-158.

[2] M. Johnson, Ch. Posthoff: TRISAT - A SAT - solver using ternary-valued logics. 14th International Workshop on Post-Binary ULSI Systems, Calgary, Canada, 2005.

[3] R.M. Karp. Complexity of computer computations. In R.E. Miller and J.W. Thatcher (editors): Reducibility Among Combinatorial Problems, New York, Plenum Press. 1972, pages 85–103.

[4] L. Levin: Universal'nye perebornye zadachi. Problemy Peredachi Informatsii 9 (3), 1973, pp. 265-266. English translation: Universal Search Problems. in B.A. Trakhtenbrot: A

Survey of Russian Approaches to Perebor (Brute-Force Searches) Algorithms. Annals of the History of Computing 6 (4), 1984, pp. 384–400.

[5] Ch. Posthoff, B. Steinbach: Logic Functions and Equations - Binary Models for Computer Science. Springer, Dordrecht, The Netherlands, 2004.

[6] B. Steinbach, N. Kmmling: Effiziente Lsung hochdimensionaler Boolescher Probleme mittels XBOOLE auf Transputer. Transputeranwendertreffen TAT'90, Proceedings, Aachen, Germany, 1990.

[7] I. Wegener: Complexity Theory - Exploring the Limits of Efficient Algorithms. Springer, Dordrecht, The Netherlands, 2005.

[8] K. Zuse: The Computer My Life. Springer-Verlag, Berlin/Heidelberg, Germany, 1993. (translated from the original German edition: Der Computer Mein Lebenswerk. Springer, 1984)

[9] SATLIB - Benchmark Problems. http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html