Int J Parallel Prog (2009) 37:324–342
DOI 10.1007/s10766-009-0097-6

A Collaborative Approach for Multi-Threaded SAT Solving

Pascal Vander-Swalmen · Gilles Dequen · Michaël Krajecki

Received: 6 February 2009 / Accepted: 31 March 2009 / Published online: 23 April 2009

Abstract The last decade progresses have led the Satisfiability Problem (SAT) to be a great and competitive practical approach to solve a wide range of industrial and academic problems. Thanks to these progresses, the size and difficulty of the SAT instances has grown significantly. Among the recent solvers, a few are parallel and most of them use the message passing paradigm. In a previous work by Vander-Swalmen et al. (IWOMP, 146–157, 2008), we presented a fine grain parallel SAT solver designed for shared memory using OPENMP and named MTSS, for *Multi Threaded Sat Solver*. MTSS extends the "guiding path" notion and uses a collaborative approach where a *rich* thread is in charge of the search-tree evaluation and where a set of *poor* threads yield logical or heuristics information to simplify the rich task. In this paper, we extend the poor thread abilities of MTSS and present extensive comparative results on random 3-SAT problems. These new experimentations show that fine grained techniques associated to poor tasks within the framework of MTSS can achieve very interesting speedup on multi-core processors.

Keywords OPENMP · Parallel combinatorial optimization · Satisfiability · DLL

P. Vander-Swalmen (✉) · M. Krajecki
University of Reims Champagne-Ardenne, Reims, France
e-mail: pascal.vander-swalmen@u-picardie.fr

M. Krajecki
e-mail: michael.krajecki@univ-reims.fr

P. Vander-Swalmen · G. Dequen
University of Picardie Jules Verne, Amiens, France

G. Dequen
e-mail: gilles.dequen@u-picardie.fr

1 Introduction

The Satisfiability Problem (SAT) is a well-known NP-Complete problem [1] and is a core problem in mathematical logic and computing theory. The interest in studying SAT has grown significantly over the last years because of its conceptual simplicity and ability to express a large set of various problems. To date, it remains a central problem in artificial intelligence, logic and computational complexity theory. Thus, it was recently used as a guide to show the convergence between combinatorial optimization and the statistical physics of disordered systems and to propose a new class of Algorithms [2]. Within a more practical framework, a lot of works highlight SAT implications in "real world" problems as diverse as Planning [3], Model Checking [4], Cryptography [5], VLSI design, ... In recent years, several improvements dedicated on the one hand, to the original backtrack-search DLL procedure [6], and on the other hand to the logical simplification techniques [7] have allowed SAT solvers to be very efficient in solving huge problems from industrial areas [8].[1] All these improvements have been proposed within a sequential framework.

In spite of the actual trend in processor development which is from single-core to multi-core CPU, there are few parallel solving approaches dedicated to the SAT problem and more generally to the solving of combinatorial problems. The parallel solvers available in the literature are, for most of them, generally designed for the message passing paradigm. Even if some CSP (Constraint Satisfaction Problems) solvers in shared memory exist [9], they mainly distribute the search-tree among the available processors.

In a previous work [10] we presented a collaborative approach dealing with the resolution of combinatorial problems. This parallel solver takes advantage from shared memory architecture using the OpenMP Application Program Interface. The parallel solving scheme we proposed was mainly focused on an exhaustive search-space enumeration. Thus, one thread has to implicitly enumerate the search-space of the problem and consequently can guarantee an answer in a finite, but exponential, time. This thread is named *rich thread*. All the other threads of our approach, named *poor threads*, aim at providing partial or definitive information about the nodes of the search-tree which are not visited by the rich thread. One of the key advantage of this solution is to manage the parallel search using a shared memory data structure, the *guiding tree*, which is interesting when considering multi-core CPU. Moreover, the data locality involved by this approach is a good propriety when considering the cache memory management: it can be useful to avoid cache misses.

This paper is organized as follows. In Sect. 2 we briefly describe the SAT problem and provide an overview of the main techniques used to efficiently solve it within a sequential framework. Section 3 presents the different works dedicated to the parallel implementation of the DLL procedure. We describe our approach using the OpenMP API in Sect. 4. Finally, we provide some experimental results in Sect. 5.

[1] http://www.satcompetition.org.

2 Preliminaries

Let $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ be a set of boolean variables. A literal is the signed form of a boolean variable. We note respectively v and $\bar{v}$ be the positive and negative literals associated to the variable v. A *CNF-Formula* (Conjunctive Normal Form) $\mathcal{F}$ is a set (interpreted as a conjunction) of *clauses*, where a clause is a set (interpreted as a disjunction) of literals. An *interpretation* of $\mathcal{F}$ is an assignment of truth values $\{true, false\}$ to $\mathcal{V}$. It is a partial interpretation if a subset of the variables of $\mathcal{F}$ are assigned. A positive (resp. negative) literal is satisfied if the corresponding variable has the value TRUE (resp. FALSE). A clause is satisfied if at least one of its literals is satisfied. Finally, $\mathcal{F}$ is satisfied according to an interpretation if all its clauses are satisfied. The SAT problem is to decide if there exists an interpretation of $\mathcal{F}$ in such a way as to make the formula evaluate to TRUE. When no such an assignment exists, $\mathcal{F}$ is FALSE. In this latter case, we would say that $\mathcal{F}$ is *unsatisfiable*; otherwise it is *satisfiable* and each interpretation satisfying $\mathcal{F}$ is a *solution*.

2.1 The SAT Solving

The SAT Problem is formulated as a decision problem and is proved NP-Complete [1]. However, we distinguish two related problems. The first one is *to find an interpretation that satisfies* $\mathcal{F}$. Local search methods [11] are useful in this case. Nevertheless, there is no guarantee that such an algorithm will find a solution when it exits. Hence, these approaches are *incomplete*. The second problem related to SAT is *to provide a certificate of the non-existence of a solution of* $\mathcal{F}$. To date, only the enumerative methods which mostly are based on Backtrack-Search process are able to prove efficiently the unsatisfiability [12]. Thus, these methods scan the search-space systematically and find a solution to the problem if it exists. If they cannot find a solution, they certify that $\mathcal{F}$ has no solution. These methods are *complete*.

2.2 The DLL Procedure

Even if our approach can be applied within an incomplete process, we mainly focus this paper on the parallelization of the complete algorithms where most of them are based on the DLL procedure [6]. The DLL procedure is described in the Algorithm 1. Assigning a truth value to a variable v helps to simplify some clauses of $\mathcal{F}$. If the TRUE literal associated to v belongs to a clause C, then C cannot participate to contradict $\mathcal{F}$. On the other hand, if it is the FALSE literal then C cannot be satisfied by it. Hence, when assigning a variable v to TRUE (resp. FALSE), we can delete the clauses containing v (resp. $\bar{v}$) and the occurrences of $\bar{v}$ (resp. v). This is denoted $\mathcal{F}\backslash v$ in the Algorithm 1. A literal l is *monotonic* (see "Monotonic" label in Algorithm 1) when its opposite does not belong to $\mathcal{F}$. In this case, we can deduce that if it exists at least one solution of $\mathcal{F}$, we will find it when l is TRUE. A *unit clause* is a clause which consists of exactly one literal. Each unit clause will be satisfied by its unique literal (see "Unit Propagation" label in Algorithm 1) unless an empty clause is encountered. The DLL procedure recursively enumerates the search-space by constructing a tree whose paths correspond to variable assignments. At each node of this search-tree, a variable v is chosen and the formula $\mathcal{F}$ is split into two simpler sub-problems $\mathcal{F}\backslash v$ and $\mathcal{F}\backslash \bar{v}$. If at least one of them contains at least one empty clause, DLL backtracks to the nearest (in term of hamming distance) unvisited assignment (see "Backtrack" label). A solution is found when no clause belongs to at least one of them (see "Solution" label).

Algorithm 1 The DLL procedure

```
Require: $\mathcal{F}$: a propositional formula
  DLL($\mathcal{F}$)
  if $\mathcal{F}$ contains one monotonic literal $l$ then
    return DLL($\mathcal{F}\backslash l$) (Monotonic)
  else if $\mathcal{F}$ contains one unit clause containing $l$ then
    return DLL($\mathcal{F}\backslash l$ ) (Unit Propagation)
  else if $\mathcal{F}$ contains at least one empty clause then
    return FALSE(Backtrack)
  else if $\mathcal{F}$ is empty then
    return TRUE(Solution)
  else
    $v \leftarrow$ one unassigned variable of $\mathcal{F}$ (Split)
    if DLL($\mathcal{F}\backslash v$) = TRUE then
      return TRUE
    else
      return DLL($\mathcal{F}\backslash \bar{v}$)
    end if
  end if
```

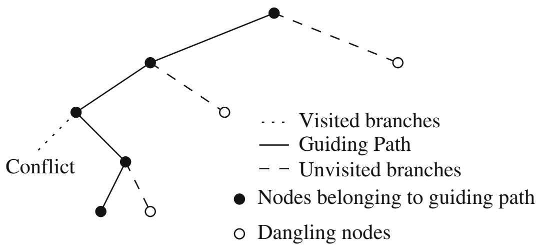
In order to improve the DLL procedure, the literature proposes some research fields which the mains are:

- *The choice of the splitting variable* (see "Split" label of the Algorithm 1) determines the variables ordering in which search is executed. It is an essential key to minimize the size of the search-tree. To date, we mainly distinguish splitting policies dedicated for randomly generated problems [12] and for industrial problems [8]. In this latter case, some works propose "restart strategies" with aim to use the clauses learning to improve the branching variable choice.
- *The pruning techniques for* DLL are related to all techniques which are able to reduce the domain of the variables. The most known of them is Unit Propagation described above. We also refer to the *look-ahead*, *equivalency reasoning*, and more recently the *clause recording* and *non-chronological backtracking*.
- *The preprocessing of the formula* refers to all techniques which simplify the formula before applying DLL. For instance we can find restrictive resolution or hyper-resolution [7] techniques.

3 SAT Parallel Solving

During the last decade, a lot of works to improve the sequential resolution runtime of the SAT problem have been proposed and have allowed SAT solvers to be very efficient in solving formulas from which the size and the solving difficulty increase. Nevertheless, there is to date few parallel solving approaches dedicated to the SAT problem. Moreover, the most of them are dedicated to the message passing paradigm and use the search-space partitioning to assign work to the available processors during the

Fig. 1 Guiding path



runtime. This often leads to use a master-slave scheme where the most difficult part consists in balancing the workload. Among the parallel SAT solvers from the literature, we can remark:

- `PSATO` [13]: This solver is based on the sequential solver `SATO` which introduces the important notion of *guiding path*. The guiding path is a dynamic object which represents the partial ordered interpretation of the splitting variables from the root to the current leaf of the search-tree during the backtrack-search process. Thus, it defines disjoint search-spaces respectively assigned to the parallel tasks. The Fig. 1 provides a sample illustration of it. Thus, each CPU executes the sequential solver on each associated subtree rooted at each node of the guiding path.
- `//satz` [14]: An important characteristic of the SAT search-space is its unbalanced distribution. Hence, it is hard to predict the time needed to achieve the enumerative process of a branch. The use of the guiding path accentuates the unbalanced phenomenon. To limit this phenomenon and following the same *master/slave* model, the parallel distributed solver `//satz` uses a dynamic workload balancing. This solver is essentially dedicated to solve Random k-SAT formula [15].
- `GridSAT` [16]: This distributed solver is especially dedicated to the grid computing. Its philosophy is to mainly keep the execution as sequential as possible and to parallelize the task when it is advantageous. `zchaff` [8] is the core sequential solver launched by the "clients". The "master" maintains a distributed learning clause database and schedules the jobs requesting the available resources list.
- `JackSAT` [17]: It presents a new approach that does not use a search-space decomposition but a cut and join scheme of the variables set. The idea is to decompose the input problem in simpler sub-problems with less variables and thus that can be independently solved. Within a parallel framework, the list of available solutions of each sub-problem are computed. Finally, these partial interpretations are join and check to exhibit, if it exists, a global solution.

Recently, the SAT community is looking at multi-cores approaches. The SAT-Race 2008[2] had a special track for parallel SAT solvers, each of them could use four cores. Three SAT solvers only participated in this competition. Here are some multi-threaded SAT solvers.

- `ySAT` [18]: It is a multi-threaded solver that proposes a lemma exchange protocols within a shared memory framework. As for the master/slave model, it synchronizes a list of available tasks to minimize the idle threads. This global

[2] http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/.

synchronization leads to drastically decrease the performance when the number of processors increases. This solver was not present at SAT-Race 2008.

- `MiraXT` [19]: This multi-threaded SAT solver is based on the Conflict Driven Learning Clause scheme and integrates a *clause learning* [20] technique in order to share logical informations between tasks. Moreover, it implements a shared *lazy structure* [8] which consists on *Watched Literal Reference List*. It was ranked third at SAT-Race 2008 (parallel track).
- `PMiniSat` [21]: `PMiniSat` is a multi-threaded version of the sequential solver `MiniSat v2.0`. This solver does not broadcast learnt clauses it they are too long but only to close threads which have a long common part of guiding path. It was ranked second at SAT-Race 2008 (parallel track).
- `ManySat` [22]: This solver is the winner of SAT-Race 2008 (parallel track). `ManySat` is a multi-threaded version of the sequential solver `MiniSat v2.02` where is grafted some extensions of conflict-analysis. The parallel approach makes unfair use of the weakness to tune the parameters of `MiniSat` for designing a more robust system.

4 Our Collaborative Approach

In this paper we propose a new parallel scheme of the DLL procedure with a fine grain parallel process. The first of our contribution is to enhance the guiding path notion. The guiding path (see Fig. 1) consists in the dynamic partition of the search-space provided by the current assignment of the splitting variables. Each dangling node of this binary depth-first search enumeration corresponds to a disjoint sub-space that can be assigned to a parallel task. As mentioned above, one of the major problem of the guiding path is the unbalanced distribution of the parallel tasks. Moreover, a sequential SAT solver has to limit the mean depth of its search-tree to be efficient. This means that a parallel solver which uses the guiding path cannot be divided in a number of tasks greater than the maximum depth of the search-tree. Otherwise, it leads to have idle processors. To avoid these two problems, we propose to extend the guiding path to the notion of *guiding tree*. The guiding tree remains a dynamic object which includes the guiding path where we dynamically open nodes of the unvisited search tree. The Fig. 2 presents this object. This notion is strongly coupled with the two others concepts of our approach: *the rich thread* and *the poor thread*.

The rich thread is able to conclude the logical value of the problem in a finite (and exponential) time. One poor thread is a task which provides a partial or definitive information about the formula (e.g.: unit propagation, choice heuristic of the splitting variable, local search algorithm, look-ahead, preprocessing technique, clause learning, …). This approach leads to have a lot of exchanges between threads through the search-tree structure. These exchanges will be small but numerous and quite unpredictable, *in extenso* depending of both the instance of the SAT problem and the SAT solver used. This is the main reason conducting us to use OpenMP rather than MPI. In fact, using MPI will lead to a complicated program involving a large set of synchronization barriers to guarantee the global state of the guiding tree. Considering this point of view, OpenMP provides a high level programming model oriented for multi-cores architecture. Moreover, an MPI implementation will need more memory, due to buffer

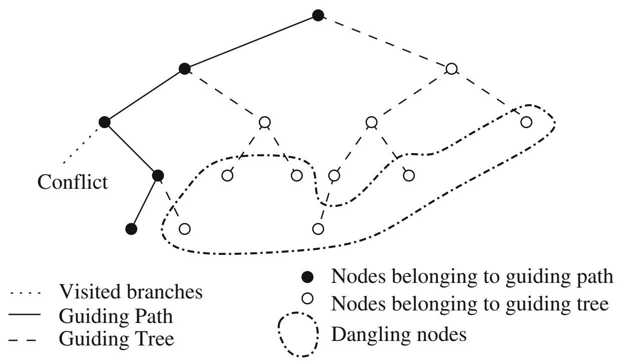

Conflict
Visited branches
Guiding Path
Guiding Tree
Nodes belonging to guiding path
Nodes belonging to guiding tree
Dangling nodes

Fig. 2 Guiding tree sample

management for exchanged messages, than an OpenMP one. We further describe the OpenMP features we use.

4.1 Rich Thread

MTSS is not a classical partitioning search-tree algorithm with similar solving threads. It consists in exactly one rich thread (i.e. *a* DLL*-like procedure*) which is helped by several poor threads. When none of poor thread contributes to solve the formula, the rich thread is equivalent to a usual sequential SAT solver. As mentioned in the Algorithm 2, the cooperation between rich and poor threads is done when rich thread backtracks and considers the nearest dangling node (see "Poor Task" label in Algorithm 2). At this point, the rich thread is able to use logical or structural informations computed by poor threads. Moreover, the rich thread maintains the context associated to all nodes of the guiding path and then allows poor threads to compute additional informations.

Algorithm 2 The Rich Thread procedure

```
Require: $\mathcal{F}$: a propositional formula
  RichThread($\mathcal{F}$)
  if $\mathcal{F}$ contains one monotonic literal $l$ then
    return RichThread($\mathcal{F}\backslash l$) (Monotonic)
  else if $\mathcal{F}$ contains one unit clause containing $l$ then
    return RichThread($\mathcal{F}\backslash l$ ) (Unit Propagation)
  else if $\mathcal{F}$ contains at least one empty clause then
    return FALSE(Backtrack)
  else if $\mathcal{F}$ is empty then
    return TRUE(Solution)
  else
    $v \leftarrow$ one unassigned variable of $\mathcal{F}$ (Split)
    if RichThread($\mathcal{F}\backslash v$) = TRUE then
      return TRUE
    else if Information from Poor Threads computations is available then
      replace current calculus context by the Poor Thread's one (Poor Task)
    else
      return RichThread($\mathcal{F}\backslash \bar{v}$)
    end if
  end if
```

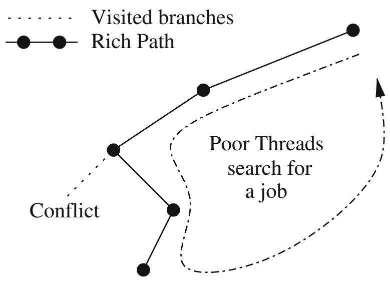


Fig. 3 Poor threads search themselves job (guiding path)

4.2 Poor Tasks

The poor threads are entities which browse the guiding tree resulting from rich and poor computations (see Fig. 3). They manage themselves their jobs and are autonomous. This induces a natural workload balancing scheme based on a server initiated principle [23]: idle poor thread will open new nodes extending the guiding tree. The algorithm of the poor thread is presented in Algorithm 3.

Algorithm 3 PoorThread procedure

```
Require: $\mathcal{F}$: a propositional formula
Require: T : a task
  PoorThread($\mathcal{F}$, T)
  $n \leftarrow$ Root of $\mathcal{F}$-search-tree
  while $\mathcal{F}$ has no solution do
    if T can be applied on $n$ then
      Apply T on $n$
    end if
    if $n$ is the last node of the guiding path or the threshold is reached then
      $n \leftarrow$ Root of $\mathcal{F}$-search-tree
    else
      $n \leftarrow$ next node of the guiding tree from $n$
    end if
  end while
```

The poor threads are looking for job browsing the guiding tree. When a task can be computed, the poor thread follows this outline, divided in three steps:

(1) *Get computation ready:* During this step, the poor thread does not lock the node it chooses. This preserves the ability of the rich to visit without any latency this node. Hence, in this latter case, the poor thread spending time during this step is lost. At the end of the preparation, the poor thread locks the node to indicate it is working on it, then the lock is removed.
(2) *Compute:* During the computation, the poor thread can find three kinds of result: a global (e.g. the formula is satisfiable) or a local (e.g. the rooted sub-tree is unsatisfiable) information or sometimes none of these. Each global information is given to the rich by a global variable. In the second case, the result is stored in one of the shared contexts of the search-tree.

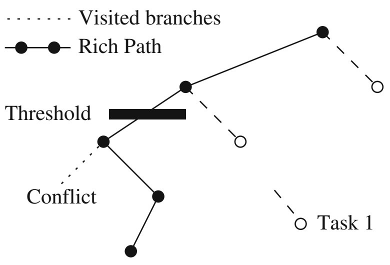


Fig. 4 Poor threads's first task and threshold system

(3) *Give result:* In this final step, the poor thread leaves its calculus context on the search-tree for a future use by the rich or a poor thread. The node is locked during this step.

This collaborative approach allows us to devise many poor tasks in the future. To date, our solver named MTSS implements two essential tasks:

- *Open Guiding Sub-Tree:* The first task assigned to a poor thread is rooted in the guiding path: the computation of the right branch of a node belonging to the guiding path (if the rich thread is computing the left one). The poor thread computes the formula with the opposite truth value chosen by the rich thread on the left branch. It then computes the next branching variable of the sub-tree if no solution was found. At the end of this task, either a root of a guiding sub-tree is created, or a global or local solution is found. The computation of the context of a new node in the guiding tree is very expensive. Each new node will have the smallest rank as possible for the first flipped split variable from the root node unless to waste time if the rich thread backtracks before the end of the poor thread. That is why we define an empirical threshold value which corresponds to an upper bound of the depth in the search-tree. Beyond this depth limit the poor threads do not work (see Fig. 4). The default value is empirically chosen from experimental results and is around the middle depth of the search-tree according to the splitting variable selection rule used (i.e. BSH [12]).
- *Develop Guiding Sub-Tree:* This task consists in computing a node and the next splitting variable in a guiding sub-tree. The implementation is the same as the previous task except that it can be computed independently for a left or a right node. In this case, the context of the previous calculus is copied. Since we are far enough in the search-tree from the guiding path of the rich thread, the threshold value mentioned above is ignored in this case. Thus, the poor threads simultaneously deploy several guiding sub-trees rooted in the guiding path (i.e. the guiding tree). This task is shown in Fig. 5. In order to maximize the helpful future work of the rich thread and the future nodes to develop (i.e. the right dangling nodes), a poor thread chooses to first open left branch from a node belonging to the guiding tree.

4.3 Context Swap and Role Swap

The threads take information from others through the shared search-tree. This induces significant data exchanges between threads and a lot of context swaps. In order to

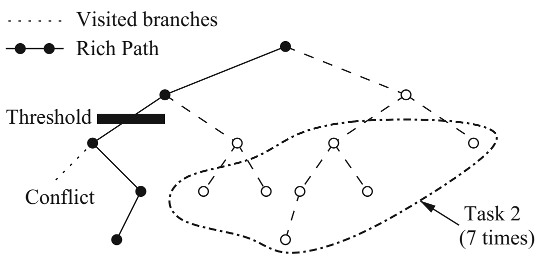


Fig. 5 Example of execution with the second task

Table 1 Context swap cost between one poor thread and one rich thread

Execution time:	0.6 s. (240 vars)		9.7 s. (300 vars)		40 s. (350 vars)	
Thread:	Rich	Poor	Rich	Poor	Rich	Poor
# Context swaps	327	3,764	2,756	53,383	8,188	170,944
% / Time	0.32%	3.63%	0.17%	3.77%	0.13%	3.74%
Time spent (s)	0.00192	0.02178	0.01649	0.36569	0.052	1.496

estimate the swap context spending time, we run MTSS on Itanium Montecito dual core processor (see Sect. 5.1) using only two threads: one dedicated to the rich thread, and the second to a single poor thread.

Based on the initial version of MTSS, some benchmarks have been conducted to evaluate the spending time of the threads to make context swaps. Table 1 shows the time spent to swap the context calculus from poor to rich (see the “rich” columns) and from poor to poor (see the “poor” columns). Readers can notice that this information exchange cost for rich thread decreases as the formula size increases. A poor thread spends much more time swapping context than the rich one because it always begins its second task by copying the current context of a node in its local memory, and always copies the new computed context in the shared node at the end of each task. The rich thread will change its context if and only if it can take valuable informations from a poor thread.

To date, MTSS operates less context swaps than in its previous version because the parallel solver is now able to directly swap a poor thread and the rich thread. This role swap happens when the rich thread needs a node on which one poor task is in computation. The rich thread becomes a poor thread when it finds a such node.

The swap is possible because the poor thread unlocks the node during the computation (task’s second step), hence, the rich thread can lock it to read the information about the task and the associated poor thread. A global variable containing the identity of the rich thread is changed by the rich thread before it becomes a poor thread. Once the poor thread has finished its computation, it reads the identity of the new rich thread and becomes it if the number of the global variable is equal to itself. In a such case, the poor thread does not copy context into the search-tree but stops its function and calls the rich function. Moreover, this principle will permit us to add long tasks. Indeed, now the rich thread never waits for any poor thread to finish its task. A mean number of this function swap is given in Table 2. It is interesting to notice we had to develop

Table 2 Number of role swaps between three poor threads and one rich thread

Formula size	350 vars	400 vars	450 vars
Execution time (s)	8.04	48.01	306.42
# Role swaps	1731	7697	24686

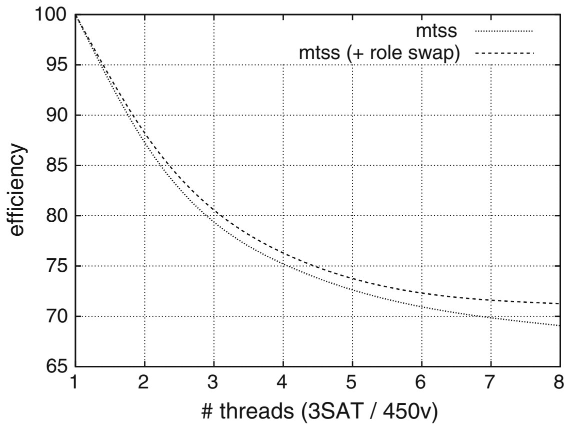
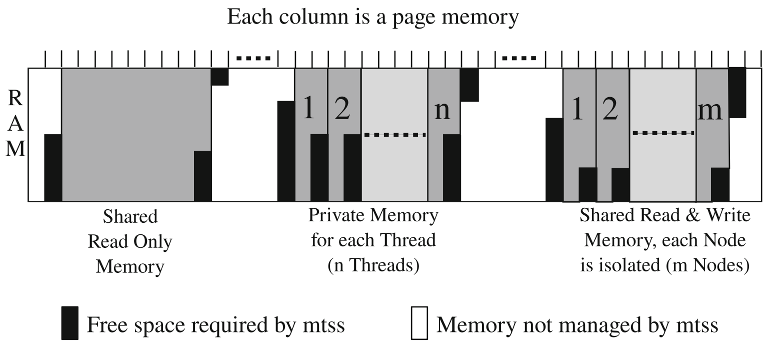


Fig. 6 Impact of role swap on th efficiency

an iterative version of this swap principle to keep good efficiency. The efficiency gap (Fig. 6) is quite reasonable but the most interesting by this improvement is the ability to devise long poor tasks in future works.

4.4 Memory Management

Our technique leads to irregular time and space memory accesses. To insure good performances despite that difficulty, we have implemented a specific memory management. It consists in isolating memory as mentioned in [24] (see Fig. 7). The memory allocations are grouped by usage so that each part will be contiguous. Thus, cache-faults (cache-misses not required) are restricted. Some free areas are needed to avoid



Each column is a page memory
RAM
1
2
n
1
2
m
Shared Read Only Memory
Private Memory for each Thread (n Threads)
Shared Read & Write Memory, each Node is isolated (m Nodes)
Free space required by mtss
Memory not managed by mtss

Fig. 7 Memory management

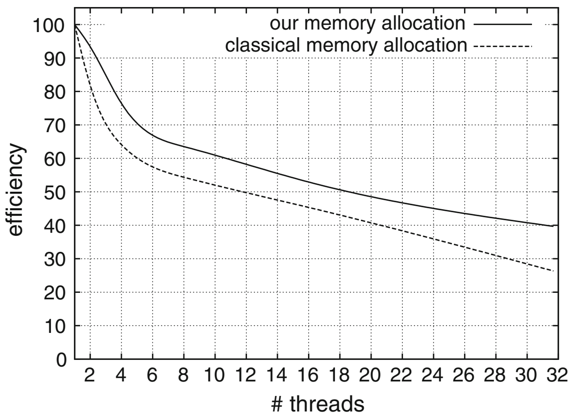


Fig. 8 Efficiency of our approach using two memory managing policies according to the number of threads

the allocation of a memory page dedicated to two areas with different usages. Each area is aligned on the beginning of a memory page. Three memory types are distinguished:

- Private memory for each thread (arrays of some datas in functions, …), moreover each thread memory is isolated from others
- Shared and read-only memory (invariant datas as clauses or number of variables, …)
- Read and write shared memory (search-tree), each node is isolated from the others.

To estimate improvements due to this memory locality, a benchmark is shown in Fig. 8. We can note a difference between efficiency: with four threads, our solver is 78% efficient versus 63% for a modified solver using classical mallocs. This difference of efficiency increases with the number of threads.

4.5 New Features

On top of parallel improvements, we added some enhancements in the heuristic BSH which are picking, pickdepth and pickback. This is a brief explanation of these enhancements. First of all, we must explain what the look ahead technique is. This technique is used by our heuristic to determine the best variable to choose: a set of variables are propagated and we analyse the formula after each propagation. This analysis helps us to determine the best choice.

- *Picking*: if during the look-ahead of the variable x, a variable y is always forced at the same value v, the variable y can be assigned to v even if x is not choosed.
- *Pickback*: during the look-ahead of the variable x, we try to increase the number of propagated variables by assigning variables which can propagate x.
- *Pickdepth*: during a look-ahead of the variable x, some more look-aheads are launched on a subset of variables belonging to reduced clauses.

The new sequential version of MTSS, thanks to these features, is nearly four times faster than the previous one. For example, you can see computation time for the two

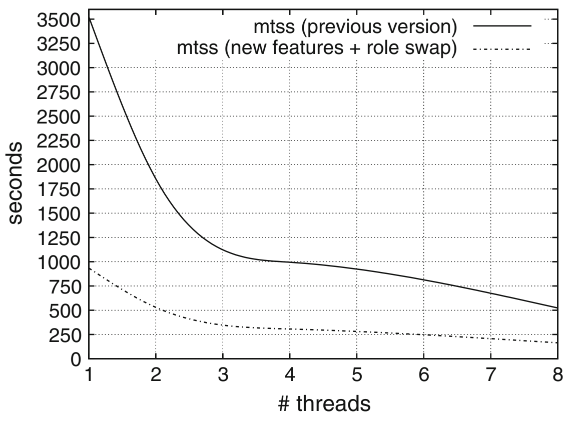


Fig. 9 Running time on a 3-SAT formula of 450 variables

versions in Fig. 9. To take best advantage from these features, we had to change some data structures, that is the reason why you can observe a light fall of efficiency. In future works, we will try to conserve speed and regain efficiency.

4.6 Implementing Our Approach with OPENMP

Thanks to OPENMP, the parallel section was very easy to design (see Annex). Actually, the main function remains smart: it defines a parallel section and creates the rich thread and the set of poor ones. To launch threads, we use these following OPENMP directives and functions: *#pragma omp parallel*, *omp_set_num_threads*, *omp_get_thread_num*.

Unfortunately, it was impossible to use automatic OPENMP parallel loops (like "*#pragma omp parallel for*") in MTSS due to the unpredictable path followed by threads during the computation and the unpredictable size of the search-tree. That is why we had to explicitly manage ourself the threads of MTSS. Synchronization of our threads implies the use of locks on nodes each time the threads modify them (description in Sect. 4.2). Hence, we use these following OPENMP directives and functions: *#pragma omp flush*, *omp_set_lock*, *omp_unset_lock*, *omp_test_lock* and *omp_init_lock*.

5 Experimental Results

MTSS is developed in C language with OpenMP primitives and functions. It has been compiled with the Intel compiler ICC 10.1.

5.1 Protocol

The cluster of SMP used for benchmarks is ROMEO II[3] from the University of Reims. 48 dual Core Itanium 2 (Montecito 4M 1.6Ghz) are dedicated to computation. The

[3] http://www.romeo2.fr.

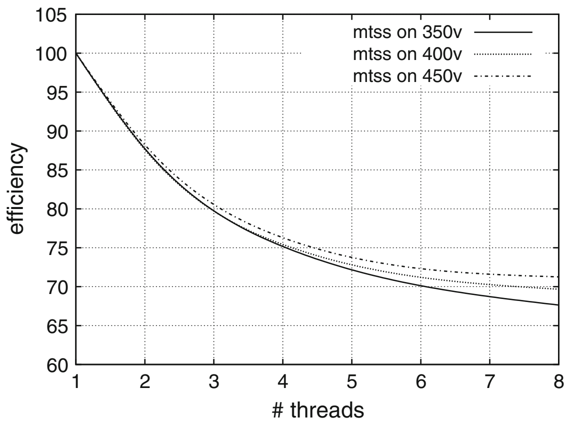


Fig. 10 Efficiency graph of random 3-SAT unsatisfiable formulas solving with a ratio $\frac{\#vars}{\#clauses} = 4.25$ (pick of difficulty) for 350, 400 and 450 variables

Table 3 Informations on sequential running time

#vars	#Instances	Sequential time (s)		
		Min	Max	Mean
350	10	17.33	32.39	24.20
400	10	46.2	248.18	144.86
450	10	451.46	1,568.30	933.97

cluster is made-up of six SMP servers of 8 cores, one of 16 cores and the last SMP node offers 32 cores. Each core of the cluster has at least 2 Gbytes of main memory.

5.2 Formulas

The formulas tested are random 3-SAT at the pick of difficulty [15] for each size. Each curve is generated from several computations and for different number of threads on 10 formulas for each size. The benchmarks were on three different sizes of formulas: 350, 400 and 450 variables. Results are shown in Fig. 10.

The studied formulas are relatively small: the sequential solving time is around 25, 150 and 1000 s respectively for the formulas of tiny, medium and large size (see Table 3).

5.3 Results

As mentioned earlier, the objective is to have an efficient SAT solver for multi-core CPU, this is the reason why the experiments are limited to 8 cores.

For each size of the problem, 10 formulas have been generated. Each formula has been two times computed with 1, 2, 4 and 8 processors. So, 240 runs have been conducted to obtain results in Fig. 10.

One can observe that MTSS achieves good efficiencies until eight processors even for the smaller problems (with 350 variables, the efficiency is greater than 65%). When using only 4 cores, the efficiency measured is close to 75%. One may notice that we have a fine grain application since MTSS threads visit about 2,000 nodes per second.

Second, when the size of the problem increases, the efficiency is more and more higher. The efficiency observed with 450 variables is near from 72% for 8 processors which is five points better than with 350 variables.

6 Conclusion and Future Works

In this paper, we presented a new parallel scheme to improve the main state-of-the-art enumerative SAT solving approaches and provided an easy way to use and to parallelize the existing sequential deduction techniques. Our solution has been implemented in a new parallel solver named MTSS. Readers should also notice that SAT is an original, and difficult, application for OPENMP since it is not a computation-intensive application and, in the same time, it is an irregular application in terms of data structures and memory access. So, to reach good efficiency, one cannot assume that the computation overlap will balance the time wasted in cache missings.

The MTSS key features can be summarized in the following way:

- MTSS defines a parallel exploration of the search space without any *a priori* informations about the instance of the problem to solve or about the SAT solver used;
- it induces a dynamic load balancing scheme which is fully distributed and server initiated;
- with the general concept of poor thread, any sequential SAT optimization can be easily effective in the parallel solver;
- MTSS is particularly well adapted to multi-core processors by having reasonable needs in terms of memory allocation and by offering reasonably memory access predictions for an irregular application;

The current version of MTSS is an efficient parallel solver, and is faster than the first version introduced in [10]. Nevertheless, several improvements can be implemented to make MTSS a real concurrent to the actual best SAT solvers.

In the short run, MTSS will include a larger set of poor tasks. Each new poor task is a new hope to improve it and we plan to study some of the existing ones such as the preprocessing techniques, subsumption deduction, clause learning, stochastic local search, ... Thanks to the architecture of MTSS and its dynamic load balancing strategy, the implementation of these new tasks will not impact the rich thread and the efficiency of the parallel solver.

An other way of improving the parallel solver would consist in developing a more effective collaboration between the threads, not only based on the guiding tree, but also taking into account information learned by threads during the resolution, like good or no good recording for example. To achieve this goal, MTSS should introduce, and manage, a new parallel shared learning space that can be read and modified by threads.

In the long run, MTSS should be able to take advantage from a cluster of SMP nodes. To address this goal, the parallel evaluation of the search space should used both the guiding tree approach introduced in this paper and the classical search tree decomposition. In the same time, MTSS should be implemented using an hybrid MPI/OPENMP model to manage the exchange between the nodes, using MPI and the threads inside a node, using OPENMP.

Annex: Implementing MTSS with OPENMP

Thanks to OPENMP, the parallel section was very easy to design. In main function:

```
nb_all_threads = number of poor threads required by user + 1;
omp_set_num_threads(nb_all_threads);
#pragma omp parallel
{
  thread_handler(omp_get_thread_num());
}

function thread_handler(me) {

  #pragma omp flush(formula_unsolved, rich_id)
  while(formula_unsolved) {
    if (me == rich_id) rich_thread(me);
    else poor_thread(me);
    #pragma omp flush(formula_unsolved, rich_id)
  }
}
```

The *rich_thread* function is in charge of the guiding-path management.

```
function rich_thread(me) {
  node = my_node[me];

  #pragma omp flush(formula_unsolved)
  while(formula_unsolved) {
    result = propagation & BSH (node);

    switch(result) {
      case UNDEF:
        node = new_node_rooted_in(node);
        break;
      case SAT:
        omp_set_lock(solution);
        formula_solution = SAT;
        formula_unsolved = FALSE;
        #pragma omp flush(formula_unsolved)
        omp_unset_lock(solution);
        break;
      case UNSAT:
        node = last_node_with_a_side_not_finished_from
          (node);
        if (node == ROOT && right_branch(node) == UNSAT)
          {omp_set_lock(solution);
          formula_solution = UNSAT;
          formula_unsolved = FALSE;
          #pragma omp flush(formula_unsolved)
          omp_unset_lock(solution);
        }
        else {
          omp_set_lock(node);
          if (a poor is working on 'node')
          {rich_id = the poor's id which is working on
              'node';
            #pragma omp flush(rich_id)
            omp_unset_lock(node);
            return;
          }
          else {
            if (informations from poor threads are
              available) {
              omp_unset_lock(node);
              copy_context from 'node' to my_context;
            }
            else {
              omp_unset_lock(node);
              update_current_context;//normal backtrack
            }
          }
        }
    }
    #pragma omp flush(formula_unsolved)
  }
}
```

The *poor_thread* function browses the guiding-tree constructed by the *rich_thread* function and the *poor_thread* functions.

```
function poor_thread(me) {
  node = ROOT;

  while(node != NULL && node.depth < threshold) {
    switch (node.poor_task_done) {
      case NONE:
        prepare_task_1(node);
        omp_set_lock(node);
        'me' is computing task 1 on 'node';
        omp_unset_lock(node);
        compute_task_1(node);
        omp_set_lock(node);
        #pragma omp flush(rich_id)
        if (me == rich_id) {
          my_node[me] = node;
          omp_unset_lock(node);
          return;
        }
        else give_infos_and_results_from_task_1(node);
        node.poor_task_done = OPEN_GUIDING_SUBTREE;
        omp_unset_lock(node);
        break;
      case OPEN_GUIDING_SUBTREE:
      case DEVELOP_GUIDING_SUBTREE:
        temp_node = browse_guiding_subtree_from(node);
        prepare_task_2(temp_node);
        omp_set_lock(temp_node);
        'me' is computing task 2 on 'temp_node';
        omp_unset_lock(temp_node);
        compute_task_2(temp_node);
        omp_set_lock(temp_node);
        #pragma omp flush(rich_id)
        if (me == rich_id) {
          my_node[me] = temp_node;
          omp_unset_lock(temp_node);
          return;
        }
        else give_infos_and_results_from_task_2
         (temp_node);
        node.poor_task_done = DEVELOP_GUIDING_SUBTREE;
        omp_unset_lock(temp_node);
    }
    node = next_node_of_guiding_path(node);
  }
}
```

Since the way followed by a thread is unpredictable, it was impossible to use OpenMP parallel loops in MTSS. That is why we had to use locks on nodes to do it by ourself.

References

1. Cook, S.A.: The complexity of theorem proving procedures. In: 3rd ACM Symposium on Theory of Computing, pp. 151–158. Ohio (1971)
2. Braunstein, A., Mézard, M., Zecchina, R.: Survey propagation: an algorithm for satisfiability. Random Struct. Algorithms **27**(2), 201–226 (2005)
3. Kautz, H., Selman, B.: Pushing the envelope: planning, propositional logic and stochastic search. In: Proceedings of the 30th National Conference on Artificial Intelligence and the 8th Innovative

Applications of Artificial Intelligence Conference, pp. 1194–1201. AAAI Press / MIT Press, Menlo Park, 4–8 August 1996
4. Biere, A., Heljanko, K., Junttila, T., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Logic. Method Computer Sci. **2**, (2006)
5. Potlapally, N.R., Raghunathan, A., Ravi, S., Jha, N.K., Lee, R.B.: Aiding side-channel attacks on cryptographic software with satisfiability-based analysis. IEEE Trans. VLSI Syst. **15**(4), 465–470 (2007)
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. J. Assoc. Comput. Mach. **5**, 394–397 (1962)
7. Bacchus, F., Winter, J.: Effective preprocessing with hyper-resolution and equality reduction. (Online). Available: citeseer.ist.psu.edu/bacchus03effective.html (2003)
8. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in a boolean satisfiability solver. In: Proceedings of ICCAD, San Jose, Nov 2001
9. Habbas, Z., Krajecki, M., Singer, D.: Decomposition techniques for parallel resolution of constraint satisfaction problems in shared memory: a comparative study. Intern. J. Comput. Sci. Eng. (IJCSE). **1**(2/3/4):192–206, inderscience Publishers, ISSN : 1742–7185 (2005)
10. Vander-Swalmen, P., Dequen, G., Krajecki, M.: On multi-threaded satisfiability solving with openmp. In: IWOMP, pp. 146–157 (2008)
11. Hoos, H.H., Stützle, T.: Stochastic local search : foundations and applications (The Morgan Kaufmann Series in Artificial Intelligence). Morgan Kaufmann, September 2004
12. Dequen, G., Dubois, O.: An efficient approach to solving random-satproblems. J. Autom. Reasoning. **37**(4), 261–276 (2006)
13. Zhang, H., Bonacina, M.P., Hsiang, J.: Psato: a distributed propositional prover and its application to quasigroup problems. J. Symb. Comput. **21**(4–6), 543–560 (1996)
14. Jurkowiak, B., Li, C.M., Utard, G.: Parallelizing Satz using dynamic workload balancing. In: Proceedings of Workshop on Theory and Application of Satisfiability Testing (Sat'2001), pp. 205–211. Boston, June 2001
15. Mitchell, D., Selman, B., Levesque, H.J.: Hard and easy distribution of SAT problems. In: Proceedings of 10th National Conference on Artificial Intelligence, pp. 459–465. AAAI (1992)
16. Chrabakh, W., Wolski, R.: Gridsat: design and implementation of a computational grid application. J. Grid Comput. **4**(2), 177–193 (2006)
17. Singer, D., Monnet, A.: JaCk-SAT: A new parallel scheme to solve the satisfiability problem (SAT) based on join-and-check. In: Proceedings of Parallel Processing and Applied Mathematics, Gdansk, (2007)
18. Feldman, Y., Dershowitz, N., Hanna, Z.: Parallel multithreaded satisfiability solver: design and implementation (2004)
19. Lewis, M., Schubert, T., Becker, B.: Multithreaded sat solving. In: ASP-DAC '07: Proceedings of the 2007 Conference on Asia South Pacific Design Automation, pp. 926–931. Washington, DC, USA: IEEE Computer Society (2007)
20. Silva, J.P.M., Sakallah, K.A.: Grasp a new search algorithm for satisfiability. In: ICCAD '96: Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided design, pp. 220–227. Washington, DC, USA: IEEE Computer Society (1996)
21. Chu, G., Stuckey, P.J.: Pminisat: a parallelization of minisat 2.0. Tech. Rep. (Online). Available: http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/descriptions/solver_32.pdf (2008)
22. Hamadi, Y., Jabbour, S., Sais, L.: Manysat a multicore sat solver (first rank at the sat race 2008 competition). Tech. Rep. (Online). Available: http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/descriptions/solver_24.pdf (2008)
23. Wand, Y.T., Morris, R.J.: Load sharing in distributed systems. IEEE Trans. Computers 202–217 (1985)
24. Jaillet, C., Krajecki, M.: Parallel programming with openmp: a new memory allocation model avoiding cache faults. In: International Workshop on OpenMP 2007 (IWOMP2007). Tsinghua University, Beijing, China, jun 2007, short paper