Journal on Satisfiability, Boolean Modeling and Computation 6 (2008) 71-98

PMSat: a parallel version of MiniSAT

Luís Gil luis.filipe.gil@gmail.com
Instituto Superior Técnico
Technical University of Lisbon
Portugal
Paulo Flores pff@inesc-id.pt
INESC–ID / Instituto Superior Técnico
Technical University of Lisbon
Portugal
Luís Miguel Silveira lms@inesc-id.pt
Cadence Research Laboratories
INESC–ID / Instituto Superior Técnico, Technical University of Lisbon
Portugal

Abstract

Parallel computing has become an affordable reality forcing a shift in the programming paradigm from sequential to concurrent applications, specially those who demand much computational power or with large search spaces like SAT-solvers. In this context we present the research, planning and implementation of PMSat: a parallel version of MiniSAT with MPI (*Message Passing Interface*) technology, to be executed in clusters or grids of computers. The main features of the program are described: search modes, search space pruning and share of learnt clauses. An analysis of its performance and load balance is also presented.

KEYWORDS: *parallel computing, SAT-solver, satisfiability, message passing interface*

Submitted December 2007; revised August 2008; published November 2008

1. Introduction

The SAT problem is concerned with the task of finding satisfying assignments to boolean formulas of propositional logic. It is associated to computational complexity by being the first NP-Complete problem ever found [5]. Its applications range from industry to science where many problems like EDA (Electronic Design Automation), search, planning and formal verification, are converted into boolean formulas and solved by specialized programs called SAT-solvers. This is done because it is easier to solve the problem in SAT than in its original domain with a specialized algorithm. By being at the same time a useful and hard to solve problem, much research has been devoted to develop methods and algorithms to process larger problems faster and with limited resources. Algorithmic advances over the last ten or so years have seen enourmous progress in the ability to handle large scale, real-life problems by current day sophisticated and efficient solvers.

The widespread availability of increasingly faster computational power, has also re-sparked the interest in new distributed paradigms potentially able to improve the capacity or efficiency of heavy computations such as those required by modern day SAT solvers. Researchers in parallel and grid computing have therefore proceeded to develop their own SAT-solver implementations in order to investigate the performance of these programs in a distributed environment and what would be the potential gains with the increase of computational power.

In this work, we present the research, development and performance analysis of PMSat: a parallel SAT-solver based on MiniSAT [7] that uses MPI technology to be executed in a cluster or grid of computers. The architecture of PMSat is that of a master-slave hierarchy, where the master generates tasks that the solvers work on. Tasks generated by the master correspond to subsets of the search space where certain assignments or assumptions were made with respect to some of the problem's variables. The main features that set PMSat apart from existing work in the area, are related to the variables selection and assumptions generation process used to partition the search space, as well as the assumptions pruning and the sharing of learnt clauses and automatic settings. Several modes of operation are available for partitioning the search space between the various computing units. Sharing of learnt clauses is also enabled in order to improve prunning of the space in subsequent searches. All of these features contribute to determine to what extent such architectures can be advantageous to SAT solution of large problems, which could be construed as another goal of this work. The usage of MPI as the underlying technology for distributed computation is also an important aspect of this work. MPI has long become the de facto standard for parallel computation and applications based on MPI are portable to a large set of environments, possibly with very distinct characteristics and structure.

This paper is organized as follows: Section 2 presents the basic background in Logic, describes the gest of the algorithms used by modern SAT-solvers, describes briefly some parallel SAT-solvers and explains PMSat's main specifications. Section 3 presents the parallel algorithm of PMSat, Section 4 shows its implementation details, Section 5 presents the performance achievements. Finally Section 6 discusses the conclusion of the work.

2. Background and previous work

This section reviews, in a very general way, the basic algorithms used by a sequential SAT-solver. Then some parallel SAT-solvers and their features are presented and finally the technological choices for PMSat are discussed.

2.1 Logic and SAT-solvers

The SAT problem is about to determine whether a propositional formula can be evaluated as *true*. There are programs called SAT-solvers to make this test automatically. Propositional formulas are built over a set of boolean variables (denoted here by characters of greek alphabet) and related by the operators $\vee$, $\wedge$ and $\neg$ named conjunction, disjunction and complement, respectively. A literal is a variable or its complement. A clause is a disjunction of literals. A formula in Conjunctive Normal Form (CNF) is a conjunction of clauses, for

instance $(\varphi \vee \neg\psi \vee \gamma) \wedge (\varphi \vee \neg\delta)$. An assignment is a mapping from a set of variables to $\{true, false\}$. For n variables there are 2^n different combinations of assignments. A formula is *satisfiable* if there exists an assignment that makes it *true*. That assignment is known as a *solution* to the problem. Otherwise the formula is *unsatisfiable*. In the worst case one needs to test an exponential number of assignments to determine whether a solution exists.

Most SAT-solvers search for satisfiability using variations of the Davis-Putnam algorithm [6] combined with *backtracking*, *learning* and *propagation* procedures. Figure 1 presents the main loop of a generic SAT-solver.

```
while(true){
        if(!decide()) return SAT;
        while(!BCP()){
                if(!resolveConflict()) return UNSAT;
        }
}
```

Figure 1. Davis-Putnam algorithm

The *decide()* function selects an unassigned variable, sets it to *true* or *false* by some, usually heuristic, method and evaluates the formula. If all clauses are satisfied and no conflicts exist the formula is satisfiable. An *unit clause* is a clause that has one unassigned literal and all the others assigned as *false*. An *implication* is to assign as *true* the remaining literal in an unit clause. A *conflict* happens when the same variable is set to *true* in one implication and *false* in another. The *BCP()* function is responsible for carrying the *Boolean Constraint Propagation*: to identify unit clauses and create implications until there are no more implications or a conflict arises.

The decisions are saved in a stack called *decision stack*. Each decision is associated to an integer tag called *decision level* that corresponds to the height of the decision in the stack. Every implication is related to the corresponding decision by the decision level, which makes it easy to find the decision responsible for an implication. The *resolveConflict()* is responsible for undoing all the implications of the current decision level and flip the value of the decision. If both values of the decision have already been tried, a *backtrack* is made in the decision stack (canceling decisions and implications) until one finds a decision not tried both ways. If no such decision can be found, and we reach the decision level zero, then the problem is UNSAT. The set of decisions responsible by the conflict are grouped and complemented to form a conflict clause that is added in a database to forbid such sets of assignments in the future. This process is called *learning*.

Some of the most popular SAT-solvers that implement these techniques are Chaff [14], GRASP [13], MiniSAT [7], SATO [17] and Satz [12].

2.2 Parallel SAT-solvers

The demand for more computational power led to the creation of new computer architectures and paradigms composed by multiple machines connected by a network to act as one, like clusters and grids. Another recent technology with potential impact on SAT solvers involves processors with multiple cores, but we will refrain from discussing this topic in this paper.

To ensure that programs make full use of the available computing power provided by distributed architectures composed of multiple processing units, most software applications must be first reengineered to partition either its data or its functionality, or both. Then the so-modified application must be coded using some special programming language and libraries to control communication and synchronization between the machines/processors.

Since the last decade there have been several implementations of SAT-solvers to be executed in clusters and grids, using distinct technologies. Some of them are GrADSAT [4], NAGSAT [8], PSATO [18] and //Satz [10]. For the most part, all these solvers have a master/slave (or *Task Farm*) architecture where a master task sends out work and collects the results, while the clients run a sequential SAT-solver. They also split the search space, using different methods, and analyze each subspace in parallel, in separate clients. Their execution is time constrained and all of them are, to some extent, fault tolerant.

GrADSAT is a solver that is based on Chaff and runs over a grid of widely distributed computers that are dynamically acquired and released. Each client in the GrADSAT environment searches in some specific subspace, but learnt clauses are shared between clients (immediately after being generated), a characteristic that can improve performance as the sharing of clauses can potentially improve search space prunning, albeit at the cost of additional communication. PSATO, based on solver SATO, runs in clusters and it is implemented in C and uses the parallel language P4 [1] to manage concurrency and communication. PSATO can save checkpoints of the search allowing a temporary suspension of the program. Clients in PSATO also search in non-overlapping subspaces of the original problem. //SATz is based on SATz, also run in clusters, is implemented in C, communicates through RPC (Remote Procedure Call) and makes load balance using the work stealing technique, which amounts to repartitioning of subspaces that have been found to be too large for a single client to search in. PaSAT uses the DOTS parallel programing toolkit [3] to build a parallel SAT solver in C++ that is able to partition the search space dynamically and share learnt clauses among the parallel tasks (*lemma exchange*). However, this technique was implemented using shared memory which limits its usage to multiprocessors machines. More recently, the same authors have extended this solver for distributed environment systems, where mobile agents were used for lemma exchange between different nodes [2]. NAGSAT exhibits a slightly different behavior as it uses a technique called nagging, where the master task runs a sequential solver while the clients explore subspaces of the search space and transmit to the master important informations. However, unlike all the other implementations, the master task is the sole responsible for the final decision and is free to use the information provided by the clients or not. NAGSAT runs on large local networks of heterogeneous desktops and it is aimed at solving 3Sat problems. It was written in C and communicates through BSD sockets.

An important thing to point out is that all these programs were developed in or assuming different parallel environments, therefore, the results achieved were computed under non-comparable conditions and using distinct sets of benchmarks. However, all the parallel solvers achieved some speedup relatively to the sequential program that each one was based on. In general the number of instances in which a super-linear speedup is achieved reduces when the number of used processors increase (probably due to the communication overhead). For the GrADSAT solver the speedup is sub-linear in most instances but, due to the parallel

implementation, solutions have, in some cases, been found for a set of instances that were never solved before.

2.3 Choices and features for PMSat

Therefore PMSat shares some common characteristics with other parallel versions of SAT solvers, such as GrADSAT, PSATO and //SATz. For once it is based on partitioning of the domain or search space, an assignment undertaken by the master, which controls the scheduling of the clients and distributes the various tasks between them. The individual clients then perform the actual search corresponding to the task given. Unlike other solvers, however, more than one partitioning heuristic is available for the user as we will see in the following Section. Like GrADSAT and PaSAT sharing of learnt clauses is allowed, albeit using a slightly different mechanism and heuristics (discussed in Section 4). Conflict learning is also used to prune the outstanding tasks and potentially to stop running clients whose search space has been proven irrelevant.

PMSat was written in C++ and uses MPI (Message Passing Interface) technology to control its execution in several computing nodes and the communication between them. MPI was chosen because, as mentioned previously, it has become industry's de-facto standard with versions for several technological platforms, providing a high degree of portability. Usage of MPI and the resulting portability and generality is one of the main features of PMSat. PMSat was also implemented based on a *Task Farm* architecture. The developed solver should be executed in a local dedicated cluster to maximize data input/output and communication speed, although it can be run on any computational platform that supports MPI. Each run of PMSat creates a fixed number of clients (called workers) and indicates to them parts of the search space to be explored, but without time constraints and assuming the absence of infrastructure faults. These last two features allowed a simpler design and a reduction in the complexity of the implementation.

In terms of inner SAT solver, we chose MiniSAT to be PMSat's core solver because it is efficient and well documented. However, its most important feature, common to some other solvers, that influenced our choice is the possibility to give to the solver a particular set of literals to be assumed as *true* and search for satisfiability based on that information. This is the procedure used by the master to instruct the clients on the subdomain to work in. When that search ends, the assumptions can be undone and the solver returns to a usable state, even when it returns UNSAT (being the result interpreted as UNSAT under assumptions). In that case the solver may fill a vector of conflicts with some of the assumed literals responsible for the contradiction. The database of learnt clauses is preserved by the solver between different runs. These features enable the possibility to share learnt clauses, prune the search space and reuse the solver without reloading the formula. In this work, different modes for search space partitioning were developed and tested, providing greater flexibility.

3. Parallel Algorithm

This section presents the partition, architecture and work flow of the parallel algorithm, as well as the details of some interesting features.

3.1 Algorithm partition

There are two basic ways in which modifications can be brought to an algorithm in order to develop a parallel version. Both of these involve a partitioning procedure: functional partition or domain partition.

A functional partition consists in splitting the algorithm in independent procedures that can be executed in parallel. In essence, the parallel version is in fact a new algorithm although it may share some similarity with the original one. A domain partition is a procedure aimed at partitioning the dataset involved in the algorithm execution, typically in order to process concurrently independent sets of input, internal or output data. In the case of SAT solvers, a partition of the problem cannot easily be made at the functional level due to the architecture of the DPLL algorithm (Figure 1). Its correct execution depends on the coherence in the data structures, that must be sequentially updated by the various functions.

Domain partitioning is however, conceptually much easier to perform. Essentially a SAT solver searches through a combinational space for an assignment that satisfies all clauses. Once such an assignment is determined, the search is halted and a result of SAT is returned. If no such assignment exist, the solver implicitly tries all combinations finally returning a result of UNSAT (in practice, conflict detection and clause learning implies that this search is never exhaustive). The multiple combinations of assignments that compose the search can be organized as a binary tree (*assignments tree*) where all branches and subtrees are independent of each other, allowing their exploration in parallel. The strategy used makes *assumptions* (sets of literals assumed as true) over a predefined, small set, of variables and searches for satisfiability on subtrees of the assignments tree using a standard solver. Note that, in the case of MiniSAT, there is no need to change the underlying solver because, as was said before, it can receive a vector with assumptions and determine whether the problem is SAT or UNSAT under those assumptions. In other words, it can be directed to a specific subspace of the original space. Once the search is finished, control returns to the main solver, the assumptions are undone and the solver is ready to be reused. By making this type of domain/space partition the search space can be split into subtrees which can be searched concurrently.

3.2 Architecture and work flow

The architectural structure chosen for PMSat is the so-called *Task Farm* approach, with a master and several workers, as presented in Figure 2. Each worker communicates only with the master and receives from him a set of assumptions that need to be tested for satisfiability. This test is performed by the worker which explores the subspace resulting from those assumptions and returns back a result. A single executable file is shared by master and workers and the work flow is described as follows:

1. upon starting, the program detects whether it is the master or a worker and executes the correspondent code;
2. the master partitions the search space according to the mode of operation configured; in practice this corresponds to generating the set of assumptions which are then sent to the workers; after this, the master waits for results from the workers;

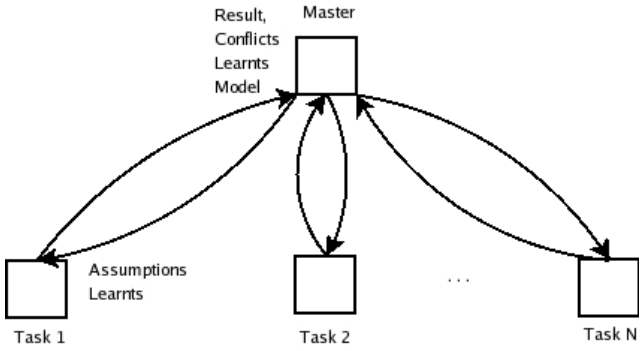


Figure 2. Architecture of the program

3. the workers use the assumptions to search for satisfiability, essentially working in the implicitly defined corresponding subspace;
4. if a worker returns UNSAT, it may send a vector with a set of learnt clauses and/or a vector of conflicts with some (or all) of the assumed literals responsible for that result; after this, the worker waits for further instructions from the master;
5. the master, after receiving the vector of conflicts, use it to remove all untested assumptions that will result in UNSAT;
6. the master, upon receiving an UNSAT result, and after updating the remaining set of assumptions, sends another assumption, possibly with some extra learnt clauses, to the idle worker;
7. when a worker returns SAT, it may optionally send back the solution found;
8. when the master receives SAT or all the assumptions are reported as UNSAT, it ends the execution (if necessary stopping all workers still active).

When a worker determines an UNSAT result, it must signal this condition to the master but also return the corresponding vector of conflicts. Since the size of this vector is unknown, one possibility would be to send an initial message with the UNSAT result and the size of the vector which would subsequently be transmitted. However this implies always sending two message, which introduces unnecessary delay. Instead we chose to include the vector in the result message, perhaps breaking the message into multiple messages, but only when necessary. In this case the vector of conflicts is sent to the master directly in the result message. This message is composed of an array of fixed, programmable, size. To enable the communication of vectors with large size, multiple messages can be sent back and a protocol has been setup to break a vector into several arrays that can be sent in the various messages and rebuilt by the master. In our tests we used an array size of twenty slots, which proved to be enough for most of the vectors tested, hence ensuring that just one message will be sent back to the master in most cases.

The pseudocode of the master's management function is shown in Figure 3. It is composed by an initialization stage where all assumptions are created and one is sent to each

```
test4SAT(){
  generate all assumptions;
  send one assumption for each worker;
  do{
    receive a result from a worker;
    if(result is SAT) return SAT;

    if(sharing learnt clauses)
      receive learnt clauses and add them into the database;

    if(conflicts enabled)
      erase the assumptions containing the conflicts;

    if(more assumptions to try){
      if(sharing learnt clauses)
        retrieve from database a set of learnt clauses and send it;
      send another assumption;
    }
  }while(not tested all assumptions);
return UNSAT;
}
```

Figure 3. Master's management function

worker. Typically more assumptions are created than the number of available workers, in order to account for workers that finish very quickly and can receive additional assumptions. After sending assumptions to all workers, the program enters into a loop waiting for the answers from any of the workers and sending more assumptions to those that report UNSAT. Inside this loop, the master may also receive learnt clauses that are stored and sent to other workers, or remove assumptions that contain conflicts. The loop ends when a solution is found or all the assumptions are reported as UNSAT. The complete algorithm, with the master and the worker code, is shown in Figure 4. After a common initialization to read the input formula, the program differentiates itself into the master code and the worker code. The master runs the function previously described to manage the workers, while these run the solver trying to find a solution constrained by the assumption received. The workers may also send and receive learnt clauses. There is an additional option to execute the sequential MiniSAT in one machine, without parallelism, which is relevant only for comparisons purposes.

4. PMSat implementation choices

In this section we describe in detail some procedures and features of the application such as: variables selection, assumptions generation, assumptions pruning, sharing of learnt clauses and automatic settings. The variables selection and assumptions generation is directly related to how the search state is partitioned into subspaces or tasks that the various workers will process in turn. As a result of the search in some subspace, a conflict may be detected that will allow prunning of outstanding assumptions and thus elimination of

```
main(){
Solver S;
  parse parameters;
  read input file into S;
  automatically calculate the missing parameters;

  if(local mode) S.solve() and output result;
  else{ //parallel mode

       if(master){ //master code
         choose the most popular variables;
         result = test4SAT();
         output result;
         if(should write the solution)
           receive and write the solution;
         abort computation;
       } //end master

       else{ //worker code
         while(true){
           receive assumption;

           if(sharing learnt clauses){
             receive a set of learnt clauses;
             insert the learnt clauses into S;
           }

           result = S.solve(assumptions);
           if(result is SAT){
             send result;
             if(should write the solution) send the solution;
           }

           if(sharing learnt clauses)
             get a set of learnt clauses from the solver and send them;

           if(removing all learnt clauses)
             delete all learnt clauses from S;

           if(conflicts enabled)
             get the set of conflicts and insert them in the result;

           send result;
         } //while
       } //end else worker
     } //end else parallel
}
```

Figure 4. Program's main with master and worker code

irrelevant portions of the search space. Finally, learnt clauses may, in some settings help speed up the search process itself.

4.1 Variables selection

As selection criteria for sets of variables to use in the assumptions we considered two options: we either choose the variables that occur more frequently or the variables that occur in bigger clauses. The goal of the first option is that by choosing the more frequent variables, our assumptions will simplify more clauses and thus, hopefully, reduce the problem to a greater extent. The second option is aimed at simplifying the biggest clauses. The number of occurrences of each literal is also registered because it is used to create the assumptions. In the following we will refer to the variables with more occurrences, and to their literals, as "the most popular".

4.2 Assumptions generation

Given a set of selected variables, specific values are assigned to them and the resulting search subspace is ready to be analyzed by a worker. This assignment procedure is termed the assumptions generation process. The assumptions generation process is akin to the generation of the *guiding path*, a concept introduced in PSATO [18]. However here the master generates at once all the paths associated with all search spaces to be used. Considering the selected variables, there is more than one way to generate assumptions and to explore the assignments tree. It was decided to allow different assumptions generation and work assignment methods. This provides a freedom of choice to the user and the possibility to compare their performance.

We propose two major methods to create assumptions, subdividing each one in two search modes. Note that irrespective of the method chosen for breaking up the assumptions tree and the search mode, all possible combinations of the selected variables are, if necessary, investigated.

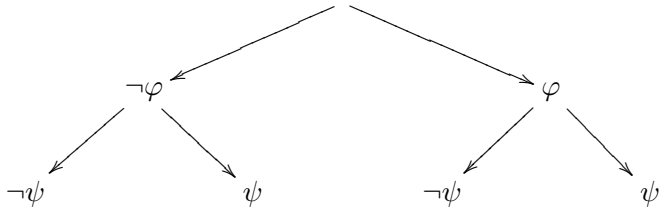
- *Equal*: all assumptions have the same number of literals.
    1. *Random*: assumptions are chosen randomly.
    2. *Sequential*: assumptions are chosen in a sequential way.
- *Progressive*: the number of literals in the assumptions changes.
    1. *Few first*: we start from assumptions with few literals and proceed to those with many literals.
    2. *Many first*: we start from assumptions with many literals and proceed to those with few literals.

In the following subsections we shall analyze and describe each method in more detail. We will assume that the k most popular variables were already chosen and we will refer to them just as the k variables for which an assignment is warranted.

4.2.1 Equal method

In the *Equal* method each assumption is a branch of the assignments tree over the k variables. All possible combinations are generated and tested, making for a total of 2^k different assumptions.

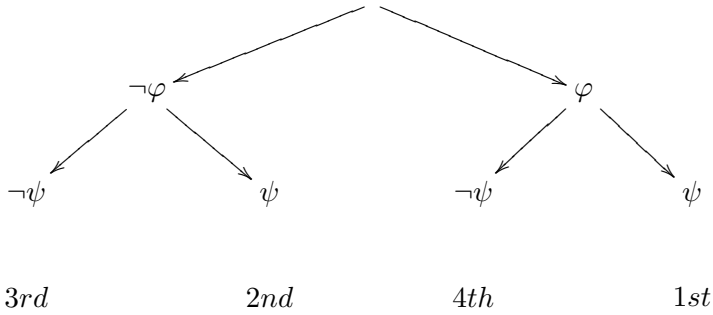
Example 4.1. *If the set of most popular variables is $\{\varphi, \psi\}$, the four assumptions to test are: $\{\neg\varphi, \neg\psi\}$, $\{\neg\varphi, \psi\}$, $\{\varphi, \neg\psi\}$ and $\{\varphi, \psi\}$, arranged in the following tree:*



Note that the union of these four assumptions constitutes the whole search space.

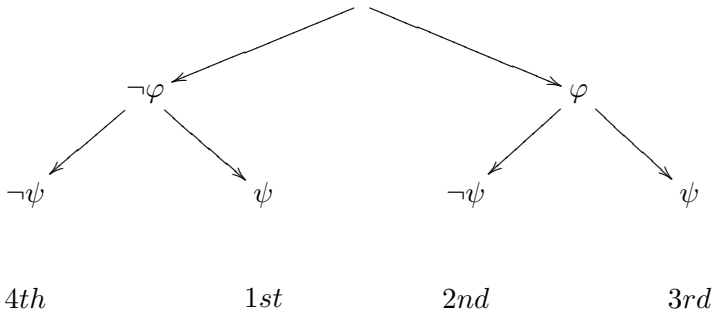
The two modes refer to the testing sequence of the assumptions. In *Random* mode the assumptions are chosen randomly to be tested.

Example 4.2. *Choosing randomly the branches:*



In *Sequential* mode the assumptions are chosen sequentially, starting by the one with the most popular literals, traversing the assignments tree from left to right.

Example 4.3. *Sequential mode, having $\{\neg\varphi, \psi\}$ as the most popular literals, where $\neg\varphi$ has more occurrences than ψ. Thus, $\{\neg\varphi, \psi\}$ is chosen first and then assignments are made left to right in the order shown (think of the leaf nodes shown as having codes $0, 1, 2, 3$ and 4. The first leaf is defined by the most popular sequence of literals (in this case leaf 1). From there on leafs are picked in natural order (2, 3, and then 0).*



4.2.2 Progressive method

The *Progressive* method is a different approach to explore the assignments tree. While in *Equal* method all the explored subtrees have the same size and the number of assumptions to test grows exponentially when the amount of their literals increase, in the *Progressive* method the intention is to make the number of assumptions grow polynomially as their literals increase but having a variation on the size of the searched subtrees. The main objective is to provide a way to have assumptions with many literals, without having an exponential growth of combinations, to explore more deeply the assignments tree. Consequently, a polynomial number of clients is sufficient to generate a response to the underlying SAT problem. For k variables this method will create $2 \times k$ assumptions. This is accomplished by generating the assumptions using the following algorithm:

Proposition 4.4. *The progressive algorithm provides a set of* $2 \times k$ *assumptions, with an amount of literals ranging from* 2 *to* k*, that covers the entire assignments tree.*

This is done in the following manner:

1. Dispose the most popular literals in a list indexed from 1 to k, with increasing popularity;
2. for each $i - th$ literal from the list, $1 < i \leq k$, create an assumption containing:
   (a) all the previous $j - th$ literals, $1 \leq j < i$;
   (b) the complement of the $i - th$ literal;
3. make an assumption with all the k literals;
4. for each of the previous assumptions, make new ones by complementing their first literal.

Proof: we will prove that steps 2 and 3 of the algorithm generate assumptions that cover exactly half of the assignments tree and that phase 4 generates the assumptions of the other half.

Proof by induction on the number of literals k in the list.

Base: $k = 1$

1. the list has one literal: $\{\varphi_1\}$.
2. the cycle does not create any assumption because there is no 2nd literal.
3. creates the assumption $h_1 = \{\varphi_1\}$. h_1 covers one branch (half) of the assignments tree for φ_1.
4. creates the assumption $h_2 = \{\neg\varphi_1\}$ that covers the other branch of the tree.

The induction hypothesis asserts that for $k = n$, with a list of n literals $\{\varphi_1, \ldots, \varphi_n\}$, the steps 2 and 3 generate the assumptions $h_1, \ldots h_n$ that cover half of the assignments tree. Complementing the first literal (φ_1) that appears in all $h_1, \ldots, h_n$ by construction, we cover the other half of the tree.

Step: $k = n + 1$

1. the list has $n + 1$ literals $\{\varphi_1, \ldots, \varphi_{n+1}\}$.
2. generates the assumptions $\{h'_1, h'_2, \ldots, h'_n\}$ where $h'_1 = h_1$, $h'_2 = h_2$, $\ldots$, $h'_{n-1} = h_{n-1}$ by construction and $h'_n = \{\varphi_1, \ldots, \varphi_n, \neg\varphi_{n+1}\} = h_n \cup \{\neg\varphi_{n+1}\}$.
3. creates the assumption $h'_{n+1} = \{\varphi_1, \ldots, \varphi_n, \varphi_{n+1}\} = h_n \cup \{\varphi_{n+1}\}$.
   The assumptions h'_n and h'_{n+1} extend the assumption h_n that covered the longest branch of that half of the tree by including the two combinations for φ_{n+1}. For induction hypothesis as $h_1, \ldots, h_n$ cover half of the assignments tree then $h'_1, \ldots, h'_{n+1}$ also cover half of the assignments tree because h'_n and h'_{n+1} extend the longest branch h_n.
4. generating $n+1$ assumptions by complementing the literal φ_1 of the previous assumptions, the other half of the tree is also covered.

QED

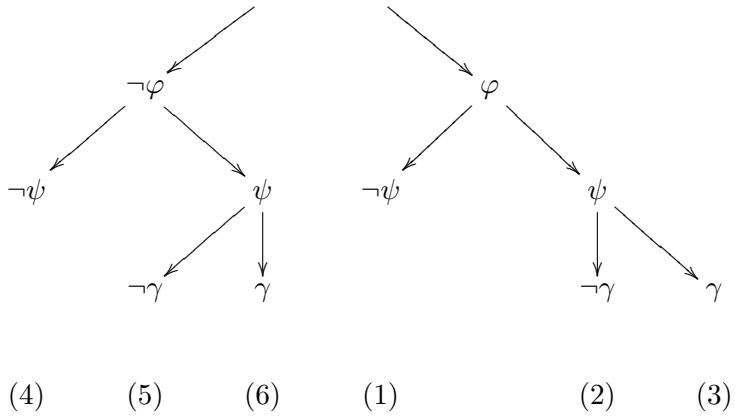
Example 4.5. *Consider the most popular literals φ, ψ and γ. Applying the steps of the method:*

1. *Get the list of popular literals $\{\varphi, \psi, \gamma\}$;*
2. *the assumption for $i = 2$ is $\{\varphi, \neg\psi\}$ and for $i = 3$ is $\{\varphi, \psi, \neg\gamma\}$;*
3. *create the assumption $\{\varphi, \psi, \gamma\}$;*
4. *create the assumptions $\{\neg\varphi, \neg\psi\}$, $\{\neg\varphi, \psi, \neg\gamma\}$ and $\{\neg\varphi, \psi, \gamma\}$.*

Six assumptions were created:

1. $\{\varphi, \neg\psi\}$;
2. $\{\varphi, \psi, \neg\gamma\}$;
3. $\{\varphi, \psi, \gamma\}$;
4. $\{\neg\varphi, \neg\psi\}$;
5. $\{\neg\varphi, \psi, \neg\gamma\}$;
6. $\{\neg\varphi, \psi, \gamma\}$.

As we can see, the assumptions (with the correspondent number under the branches) cover all the assignments tree:



The first and fourth assumptions explore a larger subtree.

The two search modes implemented allow to decide whether to start from the assumptions with two literals (*Few first*) or from those with k literals (*Many first*).

4.3 Assumptions pruning

Suppose a given worker is searching on some subspace of the variables given the assumptions assumed and it finds a contradiction under the those assumptions. In this case the solver may fill the vector of conflicts with a subset (or the entire set) of the assumed literals responsible for that contradiction. They are added to the result message sent to the master, that proceeds to erase all the assumptions not yet tested that contain them. The objective is to invalidate branches from the search tree even without evaluating them in order to save time. The effectiveness of this procedure depends on the quality and pervasiveness of the conflict learned, which is unfortunately problem-dependent.

4.4 Learnt clauses

We implemented a mechanism to optionally share learnt clauses between the workers. If this option is enabled, each worker sends to the master a set of learnt clauses upon finishing its task, i.e., after finishing its search. A balance must be striken here since on one hand, sharing of learnt clauses implies communication, which is costly, while on the other hand it may help prune the search space, avoiding the scheduling of unnecessary future tasks. In our approach, each set may have a maximum number of clauses, each one with a limited size (amount of literals). These restrictions put a threshold on the size of the message sent, and limits the size of the clauses database (there may exist clauses with hundreds of literals).

When the sharing of learnt clauses is active (by default it is off) the user can control the "sharing intensity". Two additional options are available for this purpose: one limits, for each worker, the maximum number of clauses to be shared (default value is 50), the other limits the maximum allowed size, in number of literals, of shared clauses (default

value is 20). These options impact directly the size of the database of learnt clauses and the communication overhead between master and workers. The defined default values seem to result in an adequate balance for the set of benchmarks we used.

The selection of the learnt clauses for sharing is made after sorting them by activity. Half of the most active clauses are analyzed (to share only the most active ones) and only those with size smaller than the maximum allowed are selected. Note that it is counterproductive to send a clause that contains many variables. Those lead to smaller cuts in the search space and are also less likely to occur in other subspaces.

The master has a database for each worker where it stores the most recent set of learnt clauses sent by each worker, up to the maximum defined value. The idea is to always have the most recent ones at hand, so the old ones are replaced by new ones that arrive. When the master has to send a set of learnt clauses to a worker he chooses one set that has not been sent to that worker yet. So the receiver will always get different and updated information. Although there is an activity associated to each learnt clause, when a worker sends a set of learnt clauses to the master their activities are not included. Each worker has the responsibility for assigning activities to that set of clauses it receives in order to respect the scale of values of other existing learnt clauses that it kept locally. The activities of the new learnt clauses are set to the maximum value of those stored in the worker to give them some relevance and to avoid being removed in the next solver's learnts cleanup.

Besides sharing, the solver, by default, saves the learnt clauses on each worker, but can also remove them all after each execution. Saving them is useful because they guide the next search.

4.5 Automatic settings

When the search mode or the number of variables to assume are not specified, they are determined automatically by the program, based on the number of workers, the assumptions-CPUs ratio (*acr*) and the search mode. The *acr* (which has a default value of 3, but may be redefined by the user) is a measure for the granularity that we want in the division of the search space and indicates the intended relation between the amount of assumptions and workers. Given w workers and an *acr*, we have a total of $acr \times w$ assumptions that are used to determine the amount V of variables for the *Progressive* and *Equal* methods, by the following formulas:

$$V_{progressive} = \lceil w \times acr/2 \rceil \tag{1}$$

$$V_{equal} = \lceil \log_2 (w \times acr) \rceil \tag{2}$$

In the *Progressive* method the number of assumptions is $2 \times V_{progressive} \approx acr \times w$.

In the *Equal* method the number of assumptions is $2^{V_{equal}} = 2^{\lceil \log_2 (acr \times w) \rceil} \approx acr \times w$.

When only the number of variables is given, the search mode is set to *Random* if $2^V \leq acr \times w$, otherwise it is set to *Many first*. When none of the parameters are given, the search mode is set to *Random* and $V = V_{equal}$.

5. Results

In this section we compare the performance (computation time) taken by the sequential MiniSAT (version 1.14) and our solver, PMSat[1], on several benchmarks instances.

5.1 Performance metrics

The sequential MiniSAT measures computation time as the CPU time from the beginning to the end of the execution. However, measuring the computation time in a distributed environment is a difficult task because several computational units (clients) are working in parallel and some communication is involved. Therefore, an investigation was started to determine a fair method for measuring the computation time in a distributed environment. As result, and after running some tests, we decided to measure the computation time of PMSat as the CPU time spent by the most occupied worker plus the CPU time spent by the master attending its requests, initializing and finalizing. All times presented in this section are in seconds.

Regarding the performance measures related with time, we present the relative speedup, relative efficiency and serial fraction. These values are calculated using the following formulas:

Notation 5.1. *We will denote T_1 as the execution time of the sequential program, T_p as the execution time of the parallel program, speedup as s_p, efficiency as e_p and serial fraction as f_p on p processors, defined respectively as*

$$s_p = \frac{T_1}{T_p} \tag{3}$$

$$e_p = \frac{s_p}{p} = \frac{T_1}{T_p \times p} \tag{4}$$

$$f_p = \frac{\frac{1}{s_p} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{5}$$

Relative speedup (s_p) can be classified as sub-linear ($s_p < p$), linear ($s_p = p$) or super-linear ($s_p > p$). Perhaps contrary to expected behavior, super-linear behavior can be seen and is in fact not that unusual. This happens because by breaking the search space into tasks that are searched independently, if a solution exists in one of these spaces, the potential speedup is limited by how quickly that task is initiated. If it is initiated quickly, then a solution is rapidly encountered and the process stopped, possibly much faster that what would happen

1. PMSat can be downloaded from http://algos.inesc-id.pt/software/pmsat

Table 1. Benchmark instances

Instance	Abbreviation	Solution	Variables	Clauses
fpga10_11_uns_rcr.cnf	fpga10_11	UNSAT	220	1122
fpga10_12_uns_rcr.cnf	fpga10_12	UNSAT	240	1344
fpga10_13_uns_rcr.cnf	fpga10_13	UNSAT	260	1586
hole11.cnf	hole11	UNSAT	132	738
mod2-3cage-unsat-9-11.cnf	mod2-9-11	UNSAT	87	232
mod2-3cage-unsat-9-4.cnf	mod2-9-4	UNSAT	87	232
unif-r4.cnf	unif-r4	UNSAT	400	1700
frb40-19-1.cnf	frb40-19-1	SAT	760	43780
frb40-19-2.cnf	frb40-19-2	SAT	760	43780
frb40-19-3.cnf	frb40-19-3	SAT	760	43780
frb40-19-4.cnf	frb40-19-4	SAT	760	43780
frb40-19-5.cnf	frb40-19-5	SAT	760	43780
mod2-3g14-sat.cnf	mod2-3g14	SAT	192	768
mod2c-rand3bip-sat-150-11.cnf	mod2c-11	SAT	212	1520
mod2c-rand3bip-sat-150-15.cnf	mod2c-15	SAT	213	1528
sat2.cnf	sat2	SAT	283	1358
unif-r5.cnf	unif-r5	SAT	251	323
vmpc_21.renamed-as.sat05-1923.cnf	vmpc_21.rn	SAT	441	45339
vmpc_23.renamed-as.sat05-1927.cnf	vmpc_23.rn	SAT	529	59685
vmpc_25.renamed-as.sat05-1913.cnf	vmpc_25.rn	SAT	625	76775
vmpc_25.shuffled-as.sat05-1945.cnf	vmpc_25.sf	SAT	625	76775
vmpc_26.renamed-as.sat05-1914.cnf	vmpc_26.rn	SAT	676	86424
vmpc_26.shuffled-as.sat05-1946.cnf	vmpc_26.sf	SAT	676	86424
vmpc_27.renamed-as.sat05-1915.cnf	vmpc_27.rn	SAT	729	96849
vmpc_27.shuffled-as.sat05-1947.cnf	vmpc_27.sf	SAT	729	96849

in the sequential code, where perhaps, that space could take a long time to be investigated. This is quite likely to occur in SAT problems, in our experience, and less likely, although not impossible, in UNSAT problems. Efficiency (e_p)is the relative speedup achieved divided by the number of processors. Note that super-linear speedups occurs when $e_p > 1$. The serial fraction (f_p) [11] determines the fraction of the program that is spent performing sequential computing. When it is negative, it indicates that a super-linear speedup has occurred.

5.2 Benchmarks and test methodology

The 25 instances used as benchmarks, their abbreviated name, solution type, number of variables and clauses are presented in Table 1.

The UNSAT instances were separated from the SAT. We selected 7 UNSAT instances and 18 SAT instances from benchmarks available in the Internet and from the SAT 2005 Competition [15].

The tests were made in the Grid Infrastructure available at INESC–ID. This grid has 11 available computers with an homogeneous architecture consisting of Pentium 4 processors at 3.2 GHz, 1 GB of RAM, Linux operating system and MPICH 1.2.6. All computers have access to the same hard drive shared via the Network File System (NFS).

To measure the performance of PMSat and gauge influence of its two specific options (removal of assumptions with conflicts and sharing of learnt clauses), two sets of tests were defined according to the number of selected variables:

1. Fixed number of selected variables:
   - assuming a selection of 6 variables for assumptions generation;
   - number of workers varied between 2 and 10;
   - for all search modes (*Random*, *Sequential*, *Few first*, *Many first*);
   - three settings, respectively, (a) with no specific options (i.e. options disabled), (b) allowing the removal of assumptions with conflicts and (c) the capability of sharing learnt clauses.
   - all other parameters assuming their default values;
2. Variable number of selected variables:
   - automatically calculating the number of variables to assume (according to (1) and (2));
   - number of workers and assumptions-CPUs ratio with equal values between 2 to 10 (2 workers, $acr = 2$; 3 workers, $acr = 3$; ...);
   - for all search modes (*Random*, *Sequential*, *Few first*, *Many first*);
   - without specific options selected (i.e. no removal of assumptions due to detected conflicts and no sharing of learnt clauses).
   - all other parameters assuming their default values;

The first set of tests enables a comparison of the time spent searching with the various search modes, considering separately the removal of assumptions with conflicts, the sharing of learnt clauses or with both options disabled (no options). We decided to select a set of 6 variables for generating the assumptions because they provide an adequate amount of assumptions generated by the search modes (in other words, they lead to a number of partitions of the search space that is sufficient to keep all workers busy). This is a very empiric choice and probably very dependent on the instances chosen. However, from the experiences we made, there does not seem to exist a particular value that is more accurate than others.

The second set of tests enables the determination of the impact of the number of assumptions to solve in the performance of the SAT solver. Note that, in the proposed setup, the number of assumptions grows quadratically with the number of workers (since we have chosen to keep the number of workers and the *acr* equal).

We could have tested the multiple combinations of options, but the interactions between them would not allow us to determine which one has a major influence on the result. So we just made tests with one option enabled each time.

Table 2. Average time taken by MiniSAT on all, just SAT and just UNSAT instances

Instances	Average solving time
All instances	839.723
Just SAT instances	916.618
Just UNSAT instances	641.994

5.3 Performance tests

In this section we compare the performance between MiniSAT and PMSat. First of all, we present the average times that MiniSAT took to solve the instances. Then, the results from the two sets of tests are presented. These results are organized in tables with the average, best and worst times determined from the total amount of measurements made.

The reader should be aware that some functions inside MiniSAT are purposedly built to exhibit random behavior in order to unbias certain choices that the solver is required to make. The end effect of that behavior, however, is that each time an instance is solved the time spent may differ by a couple of seconds. These small differences are nevertheless generally irrelevant for the type of results we seek.

Comparing our results with other developed parallel SAT solvers (such as the ones mentioned in Section 2.2) is difficult and could be misleading. This is mainly because the set of heterogeneous distributed environments used by each solver and the selected benchmarks are very different, that making a fair comparison is impossible. But, in general, we observed that our solver is able to achieve a higher percentage of super-linear speedups, when compared with the results presented by the other solvers. This may result from the lower communication overhead introduced by our parallel implementation of the PMSat solver.

5.3.1 Sequential times

Table 2 presents the average time taken by MiniSAT to solve all, just the SAT and just the UNSAT instances. The time to solve each instance is displayed in the tables of best and worst times presented in the following subsections.

5.3.2 Performance tests with 6 variables

This section presents the average, best and worst times that PMSat took to solve all instances under the conditions of the first set of tests.

The tables with average times for all instances (Table 3), just SAT instances (Table 4) and just UNSAT instances (Table 5) present the average time spent by PMSat to solve them with 3, 6 and 9 workers.

The following information is shown in tables of average times: the number of workers used (**#W**), the selected search mode (**Mode**), and the average times spent with the following settings: with options disabled (i.e. without removing assumptions with conflicts or sharing learnt clauses) (**Avg-no-opts**), allowing removal of assumptions with conflicts (**Avg-confs**) and allowing sharing learnt clauses (**Avg-learnts**).

Table 3. Average time by PMSat of all instances for the first set of tests

#W	Modes	Avg-no-opts	Avg-confs	Avg-learnts
3	*Few first*	583.171	583.062	354.246
3	*Many first*	515.157	515.325	507.609
3	*Random*	233.955	325.698	402.908
3	*Sequential*	491.947	481.000	487.951
6	*Few first*	263.225	248.453	297.568
6	*Many first*	487.311	487.295	490.225
6	*Random*	442.357	435.161	462.267
6	*Sequential*	455.172	456.049	452.549
9	*Few first*	204.162	203.460	202.118
9	*Many first*	494.989	494.613	503.160
9	*Random*	437.607	404.607	395.692
9	*Sequential*	440.508	442.092	442.603

Table 4. Average time by PMSat of SAT instances for the first set of tests

#W	Modes	Avg-no-opts	Avg-confs	Avg-learnts
3	*Few first*	667.143	667.597	344.804
3	*Many first*	570.140	569.682	567.213
3	*Random*	232.513	378.467	470.377
3	*Sequential*	593.173	589.025	587.246
6	*Few first*	234.601	218.502	280.733
6	*Many first*	558.833	558.679	559.042
6	*Random*	549.986	535.787	580.246
6	*Sequential*	570.729	569.105	564.815
9	*Few first*	148.731	148.865	149.395
9	*Many first*	556.352	555.212	556.074
9	*Random*	541.893	505.616	497.348
9	*Sequential*	562.493	561.593	565.651

Comparing the average time per instance taken by PMSat and MiniSAT presented in Table 3 and Table 2 respectively, we notice that it decreased from more than 800 seconds taken by the sequential solver to about 500 seconds taken by PMSat. We can also see that there were registered few gains due to the sharing of learnt clauses or the removal of assumptions with conflicts. To some extent this could be attributed to the cost of communication incurred with transmitting the associated information, which may balance out the potential benefit of using such information.

Table 4 presents the average time per SAT instance varying between 150 and 650 seconds, smaller than the 916 seconds taken by MiniSAT in Table 2. The search mode *Few first* achieved the best average times with values near 250 seconds for 6 workers and 150 for 9 workers.

Table 5. Average time by PMSat of UNSAT instances for the first set of tests

#W	Modes	Avg-no-opts	Avg-confs	Avg-learnts
3	*Few first*	367.242	365.687	378.527
3	*Many first*	373.769	375.551	354.343
3	*Random*	237.662	190.008	229.415
3	*Sequential*	231.650	203.219	232.620
6	*Few first*	336.829	325.469	340.857
6	*Many first*	303.398	303.737	313.267
6	*Random*	165.595	176.409	158.891
6	*Sequential*	158.026	165.335	163.864
9	*Few first*	346.699	343.846	337.689
9	*Many first*	337.199	338.787	367.097
9	*Random*	169.442	144.868	134.292
9	*Sequential*	126.830	134.803	126.195

Results in Table 5 show that the average time per UNSAT instance varied between 125 and 370 seconds, clearly smaller than the 640 seconds taken by MiniSAT, in Table 2. Once again, the best average time (126 seconds) was achieved with 9 workers, but this time with the *Sequential* search mode.

The tables with the best and worst times present for each instance the configuration that made PMSat spend more time to solve it.

The tables of best and worst times display the following information: instance name (**Instance**), time spent by the sequential MiniSAT (**MiniSAT**), time of the parallel algorithm (**PMSat**), search mode (**Mode**), number of workers (**#W**), speedup (**Spd.**), efficiency (**Eff.**) and serial fraction (**S. F.**).

Table 6 shows the best performances achieved by PMSat for the first set of tests (6 variables) without any defined options (without removing assumptions with conflicts or sharing learnt clauses). All but one of the runtimes were smaller than those taken by MiniSAT and about half or more corresponded to super-linear speedups, specially visible in the SAT instances. The modes with more occurrences were *Random* and *Few first*.

However, PMSat did not always solve the problems faster, as indicated in Table 7 (the worst case performances). In some situations PMSat took more time than MiniSAT. Most of the worst times were registered with less than 4 workers due to the long computations that occupied them excessively. This means that using many workers is typically a good option since it tends to avoid worst case settings.

5.3.3 Performance tests with different number of variables

This section presents the average, best and worst times that PMSat took to solve all instances under the conditions of the second set of tests. These times are compared with those achieved by MiniSAT.

Many of the best performances were obtained in this second set of tests.

Table 6. PMSat's best performances for the first set of tests

Sol	Instance	MiniSAT	PMSat	Mode	#W	Spd.	Eff.	S. F.
UNSAT	fpga10_11	44.999	31.563	*Few first*	8	1.426	0.178	0.659
	fpga10_12	158.474	71.475	*Many first*	9	2.217	0.246	0.382
	fpga10_13	161.586	165.130	*Random*	5	0.979	0.196	1.027
	hole11	723.157	218.260	*Few first*	10	3.313	0.331	0.224
	mod2-9-11	78.573	29.017	*Many first*	9	2.708	0.301	0.290
	mod2-9-4	80.753	28.197	*Random*	9	2.864	0.318	0.268
	unif-r4	3246.920	198.606	*Random*	10	16.349	1.635	-0.043
SAT	frb40-19-1	287.402	15.303	*Random*	6	18.781	3.130	-0.136
	frb40-19-2	540.394	0.233	*Random*	9	2319.228	257.692	-0.125
	frb40-19-3	6340.410	8.459	*Few first*	8	749.589	93.699	-0.141
	frb40-19-4	901.448	236.657	*Random*	3	3.809	1.270	-0.106
	frb40-19-5	4528.720	43.460	*Random*	5	104.205	20.841	-0.238
	mod2-3g14	877.347	37.102	*Many first*	6	23.647	3.941	-0.149
	mod2c-11	75.333	4.713	*Few first*	3	15.983	5.328	-0.406
	mod2c-15	26.554	0.996	*Random*	7	26.659	3.808	-0.123
	sat2	66.552	6.372	*Random*	10	10.444	1.044	-0.005
	unif-r5	360.687	1.423	*Random*	10	253.454	25.345	-0.107
	vmpc_21.rn	51.191	0.375	*Random*	10	136.505	13.651	-0.103
	vmpc_23.rn	151.893	5.560	*Few first*	8	27.317	3.415	-0.101
	vmpc_25.rn	472.618	0.732	*Random*	10	645.627	64.563	-0.109
	vmpc_25.sf	25.826	5.107	*Random*	6	5.057	0.843	0.037
	vmpc_26.rn	142.121	116.604	*Few first*	5	1.219	0.244	0.776
	vmpc_26.sf	301.119	11.303	*Few first*	10	26.641	2.664	-0.069
	vmpc_27.rn	896.080	57.139	*Random*	3	15.683	5.228	-0.404
	vmpc_27.sf	453.440	0.465	*Few first*	5	975.123	195.025	-0.249

In first place we present Table 8 with the average times per instance using 3, 6 and 9 workers respectively, for all, just the SAT and just the UNSAT instances.

The following information is shown in the Table 8: the number of workers used (**#W**), the selected search mode (**Mode**) and the average times spent for all (**All instances**), just SAT (**SAT instances**) and just UNSAT instances (**UNSAT instances**).

Once again the times spent by PMSat were smaller than the ones spent by MiniSAT. For all instances, the best average time taken by the parallel solver was 10 times inferior when were used 9 workers and the search mode *Few first*. We also notice that the search modes from the *Progressive* method had average times smaller than the ones from the *Equal* method.

Now we present the tables with the best and worst times achieved by the second set of tests.

The tables of best and worst times display the following information: instance name (**Instance**), time spent by the sequential MiniSAT (**MiniSAT**), time of PMSat (**PMSat**),

Table 7. PMSat's worst performances for the first set of tests

Sol	Instance	MiniSAT	PMSat	Mode	#W	Spd.	Eff.	S. F.
UNSAT	fpga10_11	44.999	95.217	*Sequential*	2	0.473	0.236	3.232
	fpga10_12	158.474	221.705	*Random*	3	0.715	0.238	1.598
	fpga10_13	161.586	415.011	*Sequential*	3	0.389	0.130	3.353
	hole11	723.157	432.901	*Few first*	8	1.670	0.209	0.541
	mod2-9-11	78.573	123.353	*Sequential*	2	0.637	0.318	2.140
	mod2-9-4	80.753	113.850	*Sequential*	2	0.709	0.355	1.820
	unif-r4	3246.920	2091.194	*Many first*	4	1.553	0.388	0.525
SAT	frb40-19-1	287.402	271.148	*Few first*	3	1.060	0.353	0.915
	frb40-19-2	540.394	571.121	*Few first*	7	0.946	0.135	1.066
	frb40-19-3	6340.410	7540.098	*Few first*	7	0.841	0.120	1.221
	frb40-19-4	901.448	3548.995	*Few first*	2	0.254	0.127	6.874
	frb40-19-5	4528.720	6313.364	*Few first*	2	0.717	0.359	1.788
	mod2-3g14	877.347	1970.777	*Few first*	2	0.445	0.223	3.493
	mod2c-11	75.333	237.513	*Random*	2	0.317	0.159	5.306
	mod2c-15	26.554	113.870	*Many first*	2	0.233	0.117	7.577
	sat2	66.552	149.822	*Many first*	2	0.444	0.222	3.502
	unif-r5	360.687	130.581	*Few first*	2	2.762	1.381	-0.276
	vmpc_21.rn	51.191	32.319	*Few first*	5	1.584	0.317	0.539
	vmpc_23.rn	151.893	197.157	*Few first*	5	0.770	0.154	1.373
	vmpc_25.rn	472.618	46.515	*Many first*	2	10.161	5.080	-0.803
	vmpc_25.sf	25.826	384.740	*Few first*	10	0.067	0.007	16.442
	vmpc_26.rn	142.121	1480.721	*Few first*	2	0.096	0.048	19.837
	vmpc_26.sf	301.119	551.501	*Few first*	8	0.546	0.068	1.950
	vmpc_27.rn	896.080	2904.091	*Random*	6	0.309	0.051	3.689
	vmpc_27.sf	453.440	35.820	*Random*	2	12.659	6.329	-0.842

search mode (**Mode**), number of variables assumed (**#V**), number of workers (**#W**), speedup (**Spd.**), efficiency (**Eff.**) and serial fraction (**S. F.**).

Table 9, displays the best times for the second set of tests. These times were inferior to the best times of the first set of tests shown in Table 6. This demonstrates that the number of variables assumed is as important as the search mode and plays a major role in the effort to find the solution. The fastest search modes were *Few first* and *Many first*, which also expresses the effectiveness of the *Progressive* method in the partition of the search space.

Super-linear speedups occurred frequently, most of them with SAT instances. The most incredible speedups were achieved when instances that took more than 500, 1000 or even 5000 seconds in MiniSAT were solved in few seconds by PMSat, meaning that the partition holding the solution was quickly (and one of the firsts being) explored.

About the worst times, presented in Table 10, many of them are 2 to 10 times higher than the time spent by the sequential algorithm. This result confirms the same worst times presented in the previous subsection, that the parallel search may also degradate performance, specially when few worker tasks are used.

Table 8. Average time by PMSat of all, just SAT and just UNSAT instances for the second set of tests

#W	Modes	All instances	SAT instances	UNSAT instances
3	*Few first*	360.804	396.191	269.810
3	*Many first*	514.925	576.319	357.054
3	*Random*	479.217	572.181	240.165
3	*Sequential*	383.589	429.391	265.811
6	*Few first*	219.419	216.414	227.144
6	*Many first*	192.874	151.095	300.308
6	*Random*	448.592	560.982	159.590
6	*Sequential*	452.214	568.754	152.541
9	*Few first*	95.589	42.726	231.520
9	*Many first*	116.089	32.758	330.369
9	*Random*	472.791	598.701	149.021
9	*Sequential*	537.086	696.683	126.693

5.4 Load balance

The load distribution, or the time that each worker spends on computing, depends on the time to solve the generated assumptions. There is no mechanism for automatic load balance or on the fly subdivision of the search space and distribution through the idle workers (akin to work stealing procedures). Since each assumption takes a different time to be solved, it leads to potential load imbalances between the workers. But during our tests we found all types of load balance. We will now report several situations that occurred, although they should be generalized with care (or not at all). A work stealing approach is currently being investigated to improve load balancing.

With UNSAT instances we found that with few workers, the time taken by them was similar although they had solved a different number of assumptions. When the number of workers increased, the load distribution remained similar with the *Random* and *Sequential* modes or sometimes diverged with the *Few first* and *Many first* modes (some workers ended quickly and others spent a long time with few assumptions). This is not necessarily unexpected and is a consequence of the different sizes of the search space partitions when progressive method is used. These are situations where stealing techniques that we are currently investigating may prove useful. With the SAT instances, we found three situations: the solution was in the hardest partition and one worker spent much time to find it while the others finished after just a few seconds; there was a balance and all the workers took similar time; or one worker found the solution while all (or some part of) the others were still solving their first assumption. In this last case, the master aborts all running workers.

Table 9. PMSat's best performances for the second set of tests

Sol	Instance	MiniSAT	PMSat	Mode	#v	#w	Spd.	Eff.	S. F.
UNSAT	fpga10_11	44.999	13.331	*Random*	2	2	3.376	1.688	-0.408
	fpga10_12	158.474	52.116	*Few first*	5	3	3.041	1.014	-0.007
	fpga10_13	161.586	181.370	*Few first*	5	3	0.891	0.297	1.184
	hole11	723.157	64.229	*Few first*	41	9	11.259	1.251	-0.025
	mod2-9-11	78.573	26.452	*Few first*	50	10	2.970	0.297	0.263
	mod2-9-4	80.753	24.190	*Sequential*	7	10	3.338	0.334	0.222
	unif-r4	3246.920	180.548	*Random*	7	9	17.984	1.998	-0.062
SAT	frb40-19-1	287.402	1.919	*Many first*	50	10	149.758	14.976	-0.104
	frb40-19-2	540.394	5.743	*Many first*	50	10	94.090	9.409	-0.099
	frb40-19-3	6340.410	9.868	*Many first*	50	10	642.548	64.255	-0.109
	frb40-19-4	901.448	25.892	*Few first*	41	9	34.816	3.868	-0.093
	frb40-19-5	4528.720	1.804	*Few first*	41	9	2510.234	278.915	-0.125
	mod2-3g14	877.347	2.433	*Few first*	25	7	360.581	51.512	-0.163
	mod2c-11	75.333	4.444	*Few first*	25	7	16.951	2.422	-0.098
	mod2c-15	26.554	0.847	*Random*	5	5	31.348	6.270	-0.210
	sat2	66.552	0.508	*Few first*	18	6	131.001	21.833	-0.191
	unif-r5	360.687	0.810	*Many first*	41	9	445.267	49.474	-0.122
	vmpc_21.rn	51.191	0.102	*Many first*	18	6	501.899	83.650	-0.198
	vmpc_23.rn	151.893	0.133	*Few first*	41	9	1142.113	126.901	-0.124
	vmpc_25.rn	472.618	0.529	*Random*	6	7	893.391	127.627	-0.165
	vmpc_25.sf	25.826	3.833	*Few first*	41	9	6.737	0.749	0.042
	vmpc_26.rn	142.121	2.334	*Few first*	13	5	60.888	12.178	-0.229
	vmpc_26.sf	301.119	4.476	*Many first*	41	9	67.270	7.474	-0.108
	vmpc_27.rn	896.080	9.311	*Few first*	41	9	96.243	10.694	-0.113
	vmpc_27.sf	453.440	0.787	*Sequential*	6	6	576.142	96.024	-0.198

6. Conclusion

We have presented a new parallel SAT-solver, called PMSat [19], that is based on MiniSAT and uses MPI technology, to be executed in clusters or in a grid of computers. It contains several features including a high degree of portability, different search modes, sharing of learnt clauses and pruning of the search space.

The development of PMSat gave us an indication of the potential contribution of parallel computing in this field. It showed how a simple idea like domain decomposition together with several search modes can introduce improvements into the search and decrease the time spent. The differences in the performances obtained indicate that they depend on the boolean formula, the search mode and the amount of variables used in assumptions. Unfortunately it seems to be impossible to predict the program's behavior as it seems to be very dependent upon the structure of the search space of the particular problem under analysis.

Table 10. PMSat's worst performances for the second set of tests

Sol	Instance	MiniSAT	PMSat	Mode	#V	#W	Spd.	Eff.	S. F.
UNSAT	fpga10_11	44.999	138.720	*Many first*	41	9	0.324	0.036	3.343
	fpga10_12	158.474	263.936	*Many first*	13	5	0.600	0.120	1.832
	fpga10_13	161.586	515.525	*Many first*	8	4	0.313	0.078	3.921
	hole11	723.157	970.758	*Few first*	2	2	0.745	0.372	1.685
	mod2-9-11	78.573	75.082	*Random*	2	2	1.046	0.523	0.911
	mod2-9-4	80.753	78.703	*Random*	2	2	1.026	0.513	0.949
	unif-r4	3246.920	2818.094	*Few first*	32	8	1.152	0.144	0.849
SAT	frb40-19-1	287.402	315.537	*Sequential*	2	2	0.911	0.455	1.196
	frb40-19-2	540.394	445.817	*Many first*	5	3	1.212	0.404	0.737
	frb40-19-3	6340.410	5347.412	*Many first*	5	3	1.186	0.395	0.765
	frb40-19-4	901.448	2218.451	*Sequential*	6	6	0.406	0.068	2.753
	frb40-19-5	4528.720	4980.721	*Random*	6	8	0.909	0.114	1.114
	mod2-3g14	877.347	1757.525	*Sequential*	2	2	0.499	0.250	3.006
	mod2c-11	75.333	119.166	*Many first*	2	2	0.632	0.316	2.164
	mod2c-15	26.554	60.001	*Many first*	8	4	0.443	0.111	2.679
	sat2	66.552	139.505	*Sequential*	2	2	0.477	0.239	3.192
	unif-r5	360.687	250.100	*Many first*	2	2	1.442	0.721	0.387
	vmpc_21.rn	51.191	55.452	*Random*	2	2	0.923	0.462	1.166
	vmpc_23.rn	151.893	116.994	*Sequential*	6	6	1.298	0.216	0.724
	vmpc_25.rn	472.618	444.469	*Random*	2	2	1.063	0.532	0.881
	vmpc_25.sf	25.826	469.870	*Sequential*	4	3	0.055	0.018	26.791
	vmpc_26.rn	142.121	1957.178	*Random*	4	4	0.073	0.018	18.028
	vmpc_26.sf	301.119	730.186	*Random*	4	3	0.412	0.137	3.137
	vmpc_27.rn	896.080	2770.463	*Few first*	2	2	0.323	0.162	5.184
	vmpc_27.sf	453.440	42.710	*Many first*	50	10	10.617	1.062	-0.006

The parallel search allowed to solve some instances in seconds if the right partition was explored. As a consequence, super-linear speedup is an achievable reality. But there does not seem to be a best search mode, because the results show that with 6 variables the predominant modes were *Random* and *Few first*, but with different granularity the modes *Few first* and *Many first* emerged as the fastest, indicating that the *Progressive* method, proposed in this paper, should be considered as a serious alternative to *Equal*.

In the tests of granularity the majority of the better times were achieved with a large number of variables and workers. This seems to be a good combination to get significant performance improvements as well as better load balancing. So fine-grained granularity with a large number of processors appears to hold better promise.

Features such as removing assumptions with conflicts or sharing learned clauses, presented few good results and did not influence the performance as expected, because we were hoping that they could improve the search and reduce the time more often. We do not believe that this is necessarily always the case, so this issue should be revisited in future work and perhaps more elaborate techniques pursued.

Additionally for future research we might suggest several decisions and features to be implemented and studied, such as different heuristics to select the variables, new methods to partition the search space, learnt clause sharing, or a system of load balancing.

Acknowledgments

The authors would like to thank Niklas Eén from Cadence Research Laboratories in Berkeley, CA, USA, for his help in understanding certain aspects of MiniSAT.

References

[1] R. Butler, E. L. Lusk, *User's guide to the P4 programming system*, Technical Report ANL-92/17, Mathematics and Computer Science Division, Argonne National Laboratory.

[2] W. Blochinger, C. Sinz, W. Küchlin, *A Universal Parallel SAT Checking Kernel*, In Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA), 2003.

[3] W. Blochinger, W. Küchlin, C. Ludwig, A. Weber, *An object-oriented plat- form for distributed high-performance symbolic computation*. Mathematics and Computers in Simulation, **49**:161178, 1999.

[4] W. Chrabakh and R. Wolski, *GrADSAT: A Parallel SAT Solver for the Grid*, Proceedings of IEEE SC03, November 2003.

[5] S. Cook, *The complexity of theorem proving procedures* in Proceedings of the third annual ACM Symposium of Theory of Computing, 1971.

[6] M. Davis, G. Logemann, and D. Loveland, *A machine program for theorem proving*, Communications of the ACM, (5):394-397, 1962.

[7] N. Eén and N. Sörensson, *An extensible SAT-solver* in SAT 2003 Volume **2919** of LNCS, Springer (2004) 502–518.

[8] S. L. Forman and A. M. Segre, *NAGSAT: A Randomized, Complete, Parallel Solver for 3-SAT*, Fifth International Symposium on the Theory and Applications of Satisfiability Testing, May 2002.

[9] I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995. Online version at http://www-unix.mcs.anl.gov/dbpp/.

[10] B. Jurkowiak, C. M. Li, and G. Utard, *A Parallelization Scheme Based on Work Stealing for a class of SAT solvers*, in Journal of Automated Reasoning (2005) **34**:73-101.

[11] A. H. Karp and H. P. Flatt, *Measuring parallel processor performance*, Comm. ACM **33** (5) (1990), pp 539-543.

[12] C. M. Li, *A constrained-based approach to narrow search trees for satisfiability*, Information processing letters **71**, 1999.

[13] J. P. Marques Silva and K. A. Sakallah, *GRASP - A New Search Algorithm for Satisfiability* in ICCAD. IEEE Computer Society Press, 1996.

[14] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang and S. Malik, *Chaff: Engineering an Efficient SAT Solver*, in Proc. of the 38th Design Automation Conference, 2001.

[15] Benchmarks from SAT 2005 Competition available at http://www.satcompetition.org.

[16] C. Sinz, W. Blochinger, W. Küchlin, *PaSAT - Parallel SAT-Checking with Lemma Exchange: Implementation and Applications*, Electronic Notes in Discrete Mathematics (LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT)), pp 205-216, 2001.

[17] H. Zhang, *SATO: A decision procedure for propositional logic*, Association for Automated Reasoning Newsletter, **22**, 1-3, March 1993.

[18] H. Zhang, M. P. Bonacina, J. Hsiang, *PSATO: a distributed propositional prover and its applications to quasigroup problems.* Journal of Symbolic Computation, 1996.

[19] PMSat solver is available for download at http://algos.inesc-id.pt/~pff/pmsat