2010 22nd International Conference on Tools with Artificial Intelligence

Improving Search Space Splitting for Parallel SAT Solving

Ruben Martins, Vasco Manquinho and Inês Lynce
Technical University of Lisbon
IST/INESC-ID
Lisbon, Portugal
{ruben,vmm,ines}@sat.inesc-id.pt

***Abstract*—The last two decades progresses have led Propositional Satisfiability (SAT) to be a competitive practical approach to solve a wide range of industrial and academic problems. Thanks to these advances, the size and difficulty of the SAT instances have grown significantly. 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 machine, like clusters and grids. However, extra computing power is not coming anymore from higher processor frequencies, but rather from a growing number of computing cores and processors. It becomes clear that exploiting this new architecture is essential for the evolution of SAT solvers.**

Search space splitting is probably the most commonly used strategy to explore the parallelism provided by the search space. However, it is not clear how to find the relevant set of variables to divide the search space. This paper extends a method based on the VSIDS heuristic to find the initial set of partition variables. A drawback of search space splitting is load balancing. To overcome this problem, we propose the use of a hybrid approach between search space splitting and portfolio. Preliminary results show that both these techniques improve the performance of the solver and reveal that combining search space splitting and portfolio approaches can lead to better results.

I. Introduction

In addition to the traditional hardware and software verification domains, Propositional Satisfiability (SAT) is becoming popular in new domains. For instance, SAT solvers are used for general theorem proving and computational biology [22], [10]. This widespread use is the result of the efficiency gains made during the last two decades. Many industrial problems with hundreds of thousands of variables and millions of clauses can now be solved within a few minutes. This impressive progress can be related to both the algorithmic improvements and the ability of SAT solvers to exploit the hidden structure of a practical problem.

However, many new applications with instances of increasing size and complexity are challenging modern solvers, while at the same time, it comes clear that the gains traditionally given by low level algorithmic adjustments are scarce. As a result, a large number of industrial instances from the last competitions remain challenging for state of the art SAT solvers. Nowadays, multicore processors are becoming prevalent. While in general it is important for existing applications to exploit this new architecture, for SAT solvers this becomes crucial. As a result, work has been done on parallelizing SAT solvers for its use on asynchronous distributed systems, using some form of message passing. Examples are `PaMiraXT` [27], `c-SAT` [23] and others [31], [17], [29], [7], [15]. However, message passing is slow and requires an additional overhead when compared to a well designed shared memory system. Recent works address multithreaded shared memory solvers, such as `ySAT` [13], `MiraXT` [18], `PMiniSAT` [8] and `ManySAT` [16]. Despite search space splitting being one of the most used strategies for parallel SAT solving, lately there has been an increasing interest in portfolio approaches, which seem to have a better performance [16], [23].

Two of the major challenges for search space splitting are (1) *choosing the partition variables*: for a given large SAT instance with hundreds of thousands of variables it is difficult to find the most relevant set of variables to divide the search space, and (2) *load balancing*: for some subproblems it is easier to prove (un)satisfiability than others. Since the time needed to prove (un)satisfiability for the subproblems cannot be predicted, the work cannot be balanced prior to search. Therefore, dynamic work stealing is expected to balance the work between all processors. Without such procedure, some processors might be quickly idle while others can take a long time to solve their subproblem. To overcome these difficulties, this paper proposes to extend a method based on the VSIDS heuristic [21] for choosing the initial set of partition variables [25] and to use a hybrid solver that includes a heuristic to switch from search space splitting to a portfolio approach when load balancing becomes an issue or when a cutoff is reached.

To evaluate these techniques, they were implemented in a parallel solver called `SAT4J//` which was developed to be run in a multicore architecture. This parallel solver is built upon the sequential SAT solver `SAT4J` [1]. `SAT4J` is implemented in Java and allows the configuration of solvers through the use of different heuristics and learning schemes. Moreover, Java is a multithreaded programming language that makes programming with threads easier, by providing built-in language support for threads. Even though `SAT4J` is not as efficient as other SAT solvers, it is probably one of the most used solvers. Due to its implementation in Java, it can be easily incorporated into other Java applications that require a SAT solver. Since multicore machines with a small number of cores are now common, it is useful to have a parallel version of `SAT4J` for a multicore architecture.

Preliminary results show that the techniques proposed in this paper improve the performance of a parallel solver and reveal that combining search space splitting and portfolio approaches can lead to better results.

1082-3409/10 $26.00 © 2010 IEEE
DOI 10.1109/ICTAI.2010.56

The main contributions of this paper are the following: (1) extension of a method based on the VSIDS heuristic for choosing the initial set of partition variables in a multicore environment, (2) inclusion of a transaction heuristic that allows a parallel solver to change from search space splitting into a portfolio approach, and (3) building of the first parallel SAT solver implemented in Java and designed for a multicore architecture.

This paper is organized as follows. The next section introduces some background on Propositional Satisfiability as well as the framework of a modern SAT solver. Section III briefly presents the most noticeable parallel SAT solvers. Next, section IV describes the techniques proposed in the paper to improve the search space splitting for parallel SAT solving. Section V describes the implementation of the different versions of the new parallel solver `SAT4J//`. Section VI presents experimental results that show the improvements due to choosing the partition variables and using a hybrid approach. Finally, section VII concludes the paper and suggests future work.

II. Preliminaries

Propositional formulas are commonly represented in Conjunctive Normal Form (CNF). A CNF formula is represented using n Boolean variables $x_1, x_2, \ldots, x_n$, each of which can be assigned truth values 0 (false) or 1 (true). A literal l is either a variable x_i (i.e., a positive literal) or its complement $\neg x_i$ (i.e., a negative literal). A clause ω is a disjunction of literals and a CNF formula φ is a conjunction of clauses.

A literal l_j of a clause ω that is assigned truth value 1 satisfies the clause, and the clause is said to be satisfied. A literal that is assigned truth value 0 can be removed from a clause. A clause with a single literal is said to be *unit*. Given a unit clause, the *unit clause rule* [11] may be applied: the unassigned literal has to be assigned value 1 for the clause to be satisfied. The derivation of an empty clause indicates that the formula is unsatisfied for the given assignment. The formula is satisfied if all its clauses are satisfied.

The *SAT problem* consists of deciding whether there exists a truth assignment to the variables such that the formula becomes satisfied.

A. Conflict-Driven Clause Learning SAT Solvers

One of the main reasons for the widespread use of SAT in many applications is that *Conflict-Driven Clause Learning* (CDCL) SAT solvers are so effective in practice. Since their inception in the mid-90s, CDCL SAT solvers have been applied, in many cases with remarkable success, to a number of practical applications.

Algorithm 1 shows the standard organization of a CDCL SAT solver first proposed by Marques-Silva and Sakallah [20]. In Algorithm 1, the following auxiliary functions are used:

- `UnitPropagation` consists of the iterated application of the unit clause rule. If an unsatisfied clause is identified, then **CONFLICT** is returned.
- `AssignBranchingVariable` consists of choosing a variable to assign and its respective value. Different heuristics have been explored in the past, including the VSIDS (Variable State Independent Decaying Sum) heuristic [21] which is considered to be one of the most effective. This heuristic associates an activity counter with each literal. Whenever a learnt clause is created after a conflict, each literal that occurs in the clause has its activity incremented by a certain value. Periodically, all activity counters are divided by some experimentally tuned number. When `AssignBranchingVariable` is called, the highest-value unassigned literal is chosen.
- `ConflictAnalysis` consists of analysing the most recent conflict and learning a new clause from the conflict. This is done by analysing the structure of unit propagation and deciding which literals to include in the learnt clause [20], [21]. If there is no way to overcome this conflict, then **UNSATISFIABLE** is returned.
- `Backtrack` procedure backtracks the search to the decision level computed by `ConflictAnalysis`. Backtracking can be non-chronological, i.e. it can be done to an earlier decision level. A decision level of a variable denotes the depth of the decision tree at which the variable is assigned a value. While backtracking, all variables assignments at decision levels higher than the one computed by `ConflictAnalysis` become unassigned.
- `AllVariablesAssigned` tests whether all variables have been assigned, in which case the algorithm terminates indicating that the CNF formula is satisfiable.

Algorithm 1 CDCL SAT Solver(φ)

```
1:  dl ← 0
2:  if (UnitPropagation(φ) == CONFLICT) then
3:    return UNSATISFIABLE
4:  while (not AllVariablesAssigned(φ)) do
5:    dl ← dl + 1
6:    AssignBranchingVariable(φ)
7:    while (UnitPropagation(φ) == CONFLICT) do
8:      β ← ConflictAnalysis(φ)
9:      if (β < 0) then
10:       return UNSATISFIABLE
11:     else
12:       Backtrack(φ, β)
13:       dl ← β
14: return SATISFIABLE
```

Algorithm 1 illustrates how modern SAT solvers work. It receives a propositional formula φ and applies unit propagation (line 2). Next, if a conflict is found then the formula is trivially unsatisfiable. Otherwise, the search process begins. The search is done until all variables are assigned a value or until unsatisfiability is proved. At each step, the solver increases the decision level (dl) (line 5). Next, it chooses a variable and its respective value (line 6) and applies unit propagation (line 7). If no conflict is found, the three steps shown in lines 5 to 7 are repeated. Otherwise, if a conflict

is found, conflict analysis is performed in order to learn a clause that will prevent similar conflicts from occurring in the future and to determine the decision level to which the solver can safely backtrack to (β in line 8). Notice that if β is less than zero, then the conflict does not depend on any variable assignment. Therefore, backtracking cannot be performed and so the formula is unsatisfiable. Otherwise, backtracking is performed and all assignments made at decision levels higher than β are undone (line 12). The decision level is updated to the current backtrack level and the solver returns to line 7 to repeat this process.

III. Parallel SAT Solving

Parallel computing has become an affordable reality, forcing a shift in the programming paradigm from sequential to concurrent applications, in particular those which demand significant computation power or large search spaces. There are a few parallel solving approaches dedicated to the SAT problem. Most of them use the message passing paradigm and search space partitioning to assign work to the available processors during the runtime. This often leads to use a master-slave scheme where the most difficult part consists of balancing the workload. Among the parallel SAT solvers mentioned in the literature, we enumerate the following most noticeable ones:

- `PSATO` [31] (1996): This solver introduces the important notion of guiding paths, which is the most common form of search space splitting for parallel SAT solving. The guiding path describes the current state of the search process. It does so by keeping track of the assigned variables until the current point of execution. For each one of these variables, the guiding path associates the currently assigned truth value, as well as a Boolean flag that represents whether there has been an attempt to assign both values to the given variable. A variable for which there was an attempt to assign both Boolean values is said to be *closed*, while one for which there was an attempt to assign only one value is *open*. Open variables represent junctions on the guiding path that lead to a yet unexplored search space.
- Solver by Böhm et al. [5] (1996): It dynamically divides the input formula into disjoint subformulas. Each subformula is solved by a sequential SAT solver running on a particular processor. The algorithm uses optimized data structures to modify the formulas. Additionally, workload balancing algorithms are used to achieve a uniform distribution of workload among the processors.
- `PSatz` [17] (2001): An important characteristic of a SAT instance search-space is its unbalanced distribution. The use of a guiding path accentuates the unbalanced phenomenon, which is counterbalanced with dynamic workload balancing. This solver follows a master-slave model, where the master is responsible for distributing the work among the slaves.
- `PaSAT` [29] (2001): This solver presents clause sharing between the different processors following a master-slave model, where the master is also responsible for sharing the conflict clauses recorded by each slave. Since adding all conflict clauses to the formula can result in an exponential blow-up, only clauses that have less than a fixed number of literals are exchanged.
- `NAGSAT` [14] (2002): It implements *nagging*, which involves a master and a set of slaves called *naggers*. In `NAGSAT`, the master runs a standard SAT algorithm with a static variable ordering. When a nagger becomes idle, it requests a *nagpoint* which corresponds to the current state of the master. Upon receiving a nagpoint, it applies a transformation (e.g., a change in the ordering of the remaining variables), and begins its own search on the corresponding subproblem. While the search of the master and naggers is semantically equivalent, they will be searched differently.
- `GridSAT` [7] (2003): This distributed solver is especially dedicated to grid computing. Its philosophy is to keep the execution as sequential as possible and to parallelize tasks when it is advantageous. The master maintains a distributed learning clause database and schedules the jobs requesting the available resources list.
- Solver by Blochinger et al. [4] (2003): It uses an architecture similar to `GridSAT`. However, a client learns a shared clause only if it is not subsumed by the current guiding path constraints. In practice, clause sharing is implemented by *mobile-agents*.
- `PaMira` [26] (2005): It uses a master-slave architecture with a dynamic search space partitioning through guiding paths. Dynamic work stealing and several clause selecting strategies are implemented.
- Solver by Plaza et al. [25] (2006): It proposes the use of the VSIDS heuristic to determine the partition variables. Its learning strategy is based on clause activity instead of the traditional clause size. Additionally, it implements a parallel preprocessing based on recursive learning.
- `PMSat` [15] (2008): It uses a master-slave scenario to implement a classical divide-and-conquer search. The user can select among several partitioning heuristics. Learnt clauses are shared between the processors, and can also be used to stop efforts related to search spaces that have been shown to be irrelevant. `PMSat` runs on networks of computers through a Message Passing Interface (MPI) implementation and is based on `MiniSAT` [12].
- `JaCk-SAT` [28] (2008): It presents an approach that does not use a search-space decomposition but rather a cut and join scheme of the variables set. The idea is to decompose the input problem into simpler subproblems with less variables that can be independently solved. Within a parallel framework, the list of variable solutions of each subproblem are computed. Finally, these partial interpretations are joined and checked to exhibit a global solution, if one exists.
- `PaMiraXT` [27] (2009): It follows a master-slave model based on message passing, making it suitable for any kind of workstation cluster. For the slaves, `MiraXT` is used,

which itself is a thread-based parallel solver designed to take advantage of current and future shared memory multiprocessor systems.

- `c-SAT` [23] (2009): A parallelization of `MiniSAT` using a MPI. It employs a layered master-slave architecture, where the master handles clause sharing, deletion of redundant clauses and the dynamic partitioning of search trees, while the slaves perform search using different heuristics and random number seeds. Therefore, it combines search splitting with a portfolio of algorithms for each subspace of the search tree.

Recently, the SAT community is looking at multicore approaches. The SAT-Race 2008[1] had a special track for parallel SAT solvers, where each of them could use four cores. Next, we present multithreaded SAT solvers.

- `ySAT` [13] (2005): It shares the original clauses from the CNF formula, while each thread maintains its own local conflict clause database. As for the master-slave model, it synchronizes a list of available tasks to minimize the idle threads. This solver shows a detrimental effect on cache performance, thus degrading the overall performance of the entire solver.
- `MiraXT` [18] (2007): It uses a divide-and-conquer approach where threads share a unique clause database which represents the original and the learnt clauses. Currently, this is the only multicore solver that shares its entire clause database. When a new clause is learnt by a thread, it uses a lock to safely update the common database. Read access can be done in parallel.
- `PMiniSat` [8] (2008): It is a multithreaded version of the sequential solver `MiniSAT` that uses a standard divide-and-conquer approach based on guiding paths. It further exploits the knowledge on these paths to improve clause sharing. It also improves unit propagation by using cache conscious data structures [9]. To reduce the idle time, a central queue of work is kept which is topped up by the longer running threads. The idle threads can then steal from this central queue without having to wait for other threads to respond.
- `ManySAT` [16] (2008): This solver is the winner of the parallel track of the SAT-Race 2008. `ManySAT` uses a portfolio of complementary sequential algorithms based on `MiniSAT`. Additionally, each sequential algorithm shares clauses to improve the overall performance of the whole system. This contrasts with most of the parallel SAT solvers generally designed using the divide-and-conquer paradigm.
- `MTSS` [30] (2009): The MultiThreaded SAT Solver (`MTSS`) 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 heuristic information to simplify the rich task. This approach is based on collaborative solving instead of the traditional search space splitting strategies or the portfolio approaches.

[1]Results available at `http://baldur.iti.uka.de/sat-race-2008/`.

IV. Improving Search Space Splitting

Search space splitting is commonly done through the use of guiding paths. However, it is not clear which variables should be chosen to construct the initial guiding path. Gil et al. [15] propose to choose the partition variables as those variables that occur more frequently or the variables that occur in larger clauses. Alternatively, Plaza et al. [25] propose to sequentially run the master for a small amount of time in order to allow the VSIDS heuristic to produce information that can be used to choose the partition variables.

We propose to extend the method introduced by Plaza et al. into a multicore environment. Instead of running only the master to gather the VSIDS information, we propose to use all threads. Since the VSIDS heuristic converges with each conflict, the number of conflicts is used as a measure instead of time. Each thread runs a sequential SAT algorithm until a certain number of conflicts is reached. In order to increase the information that is gathered, each thread searches a different initial search space. This is done by randomizing the initial position of the variables in the heap of the VSIDS heuristic. With this approach we have an initial stage of *weak portfolio*, since each thread searches in a different part of the search space. Our motivation with this approach is two-fold: first, we can solve easy instances with the weak portfolio approach, i.e. without even splitting the search tree; second, in this first stage more information is collected about the variables that will allow to choose better partition variables.

Algorithm 2 PartitionVariables()

```
1:  if (conflicts ≥ k) then
2:    for pos = 1 to n do
3:      var ← VSIDS[pos − 1]
4:      infoVars[var] ← infoVars[var] + pos
5:    threadInfo ← threadInfo + 1
6:    if (threadInfo == nrThreads) then
7:      workQueue ← ConstructInitialGPs(nPart)
8:      currentGP ← workQueue.pop()
9:      notifyAll()
10:   else
11:     wait()
12:     currentGP ← workQueue.pop()
```

Algorithm 2 describes the proposed procedure for choosing the partition variables. Consider n variables. After k conflicts (line 1), the variables that have the highest activity are considered the most interesting ones to branch on. The VSIDS score of each thread is analysed and for each variable is given a score from 1 to n according to its position in the heap of the VSIDS heuristic (lines 2 to 4). With this counting, method the score of each variable in *infoVars* will not be dominated by the VSIDS score of a thread. If the current number of threads that gathered information (*threadInfo*) is less than the total number of threads (*nrThreads*), then the current thread waits for the remaining threads to gather information before continuing (line 11). Otherwise, if it is the last thread that gathered

information (line 6), then it builds the initial guiding paths (line 7) based on the sum of all the scorings for each variable. The chosen partition variables are the ones with the lowest score in *infoVars*, which means that according to the VSIDS heuristic of all threads these variables are the best candidates for being partition variables. `ConstructInitialGPs` builds the initial guiding paths. Note that *nPart* partition variables produce 2^{nPart} guiding paths. Afterwards, the thread gets its current guiding path (*currentGP*) from the work queue (*workQueue*) (line 8). Finally, it notifies all threads (line 9) that are waiting to continue their execution. The threads that are awaken (line 11) also get their current guiding path from the work queue (line 12).

A clear drawback of search space splitting is load balancing, since the branches of a search tree are typically extremely imbalanced and may require a non-negligible overhead of communication for work stealing. To overcome this difficulty, we propose to switch from search space splitting to portfolio when load balancing becomes an issue or when a cutoff is reached.

In the recent past, have been proposed a few hybrid approaches between search space splitting and portfolio, which will be described next.

Blochinger [3] proposes to use an adaptive competition by starting with a search space splitting strategy and switching into a portfolio approach when a particularly hard region of the search space is encountered. In order to steer the transition from search space splitting to portfolio, a transition heuristic is employed to evaluate the progress of the solving process of an individual subproblem. Next, it decides when to switch to a portfolio approach. When a transition is initiated, the respective subproblem is additionally treated using different search heuristics. Further search splitting is disabled and if this subproblem is proved to be satisfiable, then the search finishes. Otherwise, if it is proved to be unsatisfiable, then the solver returns to a search space splitting strategy. This process is iterated until a solution is found or the problem is proved to be unsatisfiable.

Alternatively, `c-SAT` [23] uses search space splitting to begin the search and in each subspace of the search tree it uses a portfolio of SAT algorithms. Since this parallel SAT solver is designed for a workstation environment, it tries to take advantage of a high number of machines by combining search space splitting with a portfolio approach. Similarly, Bordeaux et al. [6] use a hybrid approach between search space splitting and portfolio in their experiments with massively parallel SAT solving. Initially, a partial fixing of the variable ordering is used. Three variables are chosen randomly and the search process is biased so that these variables are always branched on first; the solver then continues branching with its normal strategy.

We propose to begin with an initial stage of search space splitting, switching to a portfolio approach when load balancing becomes an issue or when a cutoff is reached. After this switch, the solver does the remaining search in a portfolio mode. Our motivation is to use search space splitting when this approach is more efficient and then to change to a portfolio approach when difficulties arise. This is done using the following heuristic:

1) If a given thread is searching for more than k conflicts and has created work that occupies the majority of the work queue, it means that the thread is currently searching in an hard subspace of the search tree and the work queue is being filled with guiding paths corresponding to this area. At this point, we switch into a portfolio mode, by changing the heuristics of each SAT algorithm on each thread. Additionally, the existing guiding paths are merged in order to find variable assignments that were already found to be necessary. Clauses learnt previously by each thread are also kept when changing into portfolio mode.
2) Alternatively, it could be the case that most threads are searching in hard subspaces of the search tree, i.e. the solving process of most subproblems is not making sufficient progress. In practice, a cutoff is used to prevent reaching hard subspaces. This happens when a given thread is searching for more than z conflicts ($z >> k$) and condition 1) was yet not verified. Although this is not the case that a subspace of a thread is dominating the others, the diversification of the search through the use of a portfolio approach tends to lead to better results. Therefore, in this case we also change into a portfolio mode as described above.

Algorithm 3 HybridHeuristic()

```
1:  if (conflicts ≥ z) OR
       (conflicts ≥ k AND
       createdWork > workQueue.size()/2) then
2:    workQueue.add(currentGP)
3:    threadPort ← threadPort + 1
4:    if (threadPort == nrThreads) then
5:      portfolioGP ← mergeGP(workQueue)
6:      changeToPortfolio()
7:      notifyAll()
8:    else
9:      wait()
10:     changeToPortfolio()
```

Algorithm 3 shows the hybrid heuristic. When conditions 1) or 2) occur (line 1), the solver enters into portfolio mode. *createdWork* is a local variable for each thread that is initialized to 0 each time the thread starts searching in a new subproblem, and increased by 1 each time it puts work in the form of guiding paths into the work queue (*workQueue*). Each thread starts by adding its own guiding path to the work queue (line 2). If the current number of threads that entered the portfolio mode (*threadPort*) is less than the total number of threads (*nrThreads*), then it waits for the remaining threads to enter into portfolio mode (line 9). Otherwise, if the current thread is the last one to enter into portfolio mode (line 4), it then merges all guiding paths

TABLE I
Pfolio: DIFFERENT STRATEGIES.

Strategies	Core 0	Core 1	Core 2	Core 3
Restart	PicoSAT restarts	MiniSAT restarts	Luby restarts	MiniSAT restarts
Polarity	Progress saving	Phase in latest learnt clause	Progress saving	Negative phase
Learning Simplification	Expensive simplification	Expensive simplification	Simple simplification	Simple simplification

in the work queue into the portfolio guiding path (line 5). The merge operation builds the *portfolioGP* with the literals that are common to all the guiding paths in the work queue. Afterwards, it changes the heuristics of the SAT algorithm (line 6). `changeToPortfolio` changes the restart heuristics, the polarity of variables and the simplification heuristic of learnt clauses. Finally, it notifies all threads (line 7) that were waiting (line 9) to continue their execution and to change their heuristics (line 10).

V. IMPLEMENTATION

`SAT4J//` is a parallelization of `SAT4J 2.1`. Each thread maintains its own local clause database, containing all clauses of the original CNF formula as well as conflict clauses deduced by the thread. At the moment, the current version of `SAT4J//` does not share clauses between the different threads. Although clause sharing can dramatically increase the performance of a solver, without clause sharing we can have a better understanding of the impact of our techniques.

`SAT4J//` features standard parallelization techniques like dynamic work stealing using guiding paths. To reduce the idle time, a central queue of work is kept which is topped up by the longer running thread (similar to [8]). Motivated by the fact that a short guiding path (GP) (with respect to the number of literals) correlates to a large fraction of the search tree, while a long GP indicates a smaller part (that might be also easier and faster to solve), the longer running thread (measured by the number of conflicts) will have the shortest guiding path and therefore should be used to divide the current search tree. This dynamic work stealing is based on stealing from the longest running thread, and from as high in that search tree as possible. With a central work queue, the idle threads can steal from it without having to wait for other threads to respond, except when the work queue becomes empty. If the work queue size is less than a given value, the longest running thread generates work by splitting its own guiding path into two. One guiding path is put on the work queue, while the other is used to continue the search of the current thread.

Four versions of `SAT4J//` were implemented to test the different techniques proposed in this paper: *no-Info*, *Info*, *Pfolio* and *Hybrid*.

In the *no-Info* version the partition variables are chosen using the VSIDS heuristic applied to the sequential SAT algorithm. However, the VSIDS heuristic in `SAT4J` is initialized with activity 0 for all variables. Therefore, using the sequential VSIDS, at the start of the SAT algorithm, is equivalent to randomly choosing the partition variables. In fact, the variables that are chosen are the ones that were last read from the input file.

The *Info* version uses the `PartitionVariables` algorithm presented in the previous section. This method allows the solver to choose the partition variables taking into consideration the VSIDS information of all threads, rather than just using a random initialization like in *no-Info*.

Pfolio version uses a portfolio of SAT algorithms designed to be run with 4 threads. Similarly to `ManySAT` [16], in *Pfolio* each thread has it own restart policy, polarity heuristic and simplification heuristic of learnt clauses. `SAT4J` allows the configuration of solvers through the use of different heuristics and learning schemes which makes it easier to build a portfolio approach. *Pfolio* was built using the configuration of solvers of `SAT4J` and it was inspired in the portfolio of `ManySAT`. For example, *Pfolio* uses a combination of rapid restart strategies with progress saving, similarly to `ManySAT`.

Table I summarizes the choices made for the different solvers of *Pfolio*. It uses different restart strategies: *MiniSAT restarts* [12] uses a classical geometric policy; *Luby restarts* [19] and *PicoSAT restarts* [2] are different rapid restart strategies. The polarity heuristic uses *progress saving* [24] that records the polarity of the assigned variables between the conflict and the back-jumping level, preferring this polarity if one of these variables is chosen again. It also uses the *phase appearing in the latest learnt clause* or the standard policy of choosing the *negative phase* first. *Expensive* or *simple reason simplification of learnt clauses* is also implemented in the different solvers.

The *Hybrid* version is based on the *Info* and *Pfolio* versions. It starts with the search space splitting strategy present in *Info* with the additional `HybridHeuristic` algorithm described in the previous section. When this method switches to a portfolio mode, the solver switches to the *Pfolio* version. Each SAT algorithm on each core changes its restart policy, polarity heuristic and simplification heuristic of learnt clauses as described above. Additionally, when transiting to the portfolio mode, the guiding paths are merged and the learnt clauses are kept.

VI. EXPERIMENTAL RESULTS

This section evaluates the techniques for improving search space splitting that were proposed. For the experiments reported, we used 82 instances from the applications category of the SAT competition 2009 (available from `http://www.satcompetition.org/2009/`). These instances were selected from the initial set of 292 instances and correspond to the instances that `SAT4J 2.1` was able to solve in more than 180 seconds or that `SAT4J 2.1` was unable to solve but `MiniSAT 2.1` was able to solve within the first phase competition timeout (within 1200 seconds). `SAT4J` is a Java

implementation of MiniSAT, but while MiniSAT 2.1 was ranked 2nd in the first phase of the SAT competition of 2009, SAT4J 2.1 was ranked 29th. The selected set of instances is therefore challenging for SAT4J and interesting for parallel testing.

The results were obtained on an Intel Core i7 CPU 930 (2.80 GHz with 6GB of RAM) running Ubuntu 10.04 LTS with a timeout of 3,600 seconds (wall clock time). All versions of SAT4J// were run with 4 threads. As runtimes are highly deviating for parallel SAT solvers, each version of SAT4J// was run three times on each instance. The runtimes shown in this section are the median of the successful runs for each instance. An instance was considered solved if it could be solved in at least one of the three runs of a solver. Note that this procedure corresponds to the one used in the SAT race of 2008 where ManySAT, PMiniSAT and MiraXT were evaluated.

TABLE II
NUMBER OF INSTANCES SOLVED BY EACH APPROACH.

	# Inst	Seq	no-Info	Info	Pfolio	Hybrid
SAT	25	16	17	19	19	20
UNSAT	57	43	42	42	45	45
Total	82	59	59	61	64	65

Table II shows the number of solutions found by each approach. We should note that SAT4J// suffers from high memory contention and a corresponding increase of cache misses, which may explain why the results are not better. To study this effect, we have run 4 SAT4J solvers simultaneously and analyzed the average solving time. This is equivalent to run SAT4J// with 4 threads without splitting the search tree. If this was done in 4 different machines, the solving time should be always the same as running SAT4J in a single machine. However, since they are all running in the same machine, memory is shared along the 4 solvers, thus increasing the memory access and the cache misses. Indeed, using 4 threads is on average 25% slower than a single thread. The same experience with MiniSAT 2.0 reveals an average slowdown of 15%. This also shows that SAT4J, that is implemented in Java, has more issues with memory contention than MiniSAT, that is implemented in C++.

Such a slowdown represents a major problem of SAT4J// since memory contention deteriorates the solver performance. Cache deterioration in parallel solvers has already been reported in the past by the authors of ySAT [13]. Recently, Chu et al. [9] proposed cache conscious data structures that reduce cache misses. This new data structures are implemented in PMiniSAT, and according to the authors increase the parallel performance of their solver by 140% with 8 threads.

Table II shows that all versions of SAT4J// can solve either the same number (*no-Info*) or more instances (*Info, Pfolio* and *Hybrid*) than the sequential solver (*Seq*). However, our goal is to compare the different version of SAT4J// and to show that our techniques can improve the overall performance of search space splitting for parallel SAT solving.

Figure 1 provides a scatter plot with the runtimes for

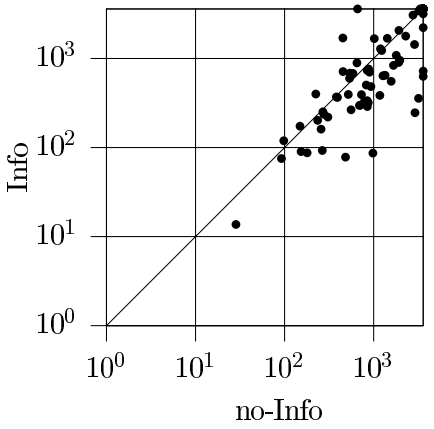

Fig. 1. Runtimes for *no-Info* and *Info*.

no-Info and *Info*. Each point in the plot corresponds to a problem instance, where the x-axis corresponds to the runtime required by *no-Info* and the y-axis corresponds to the runtime required by *Info*. Clearly, *Info* outperforms *no-Info* since for most instances it was able to prove (un)satisfiability faster than *no-Info*. Additionally, table II shows that *Info* was able to solve 2 more instances than *no-Info*. In our experiments, 4 initial partition variables were chosen to build the initial guiding paths. With 4 partition variables, we construct 16 guiding paths that were used to start the search and fill the work queue. Even though we are only choosing 4 initial partition variables, these results clearly show that the way they are chosen can dramatically affect the performance of the parallel solver.

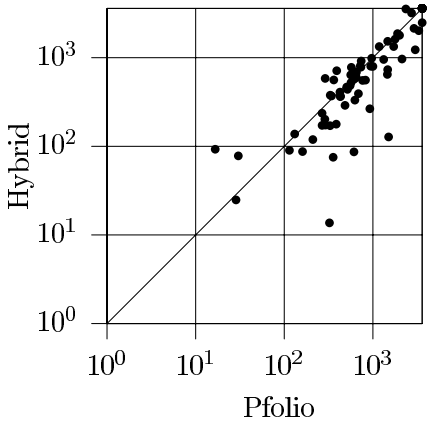

Fig. 2. Runtimes for *Pfolio* and *Hybrid*.

Despite the fact that *Info* improves *no-Info*, it still solves less 3 instances than the *Pfolio* version. However, our *Hybrid* version is able to solve more instances than the *Pfolio* version and at the same time to outperform on most instances. Figure 2 provides a scatter plot with the runtimes for *Pfolio* and *Hybrid*. From the 65 instances that were solved by *Hybrid*, 54 of them switched from search space splitting to a portfolio approach. This also shows that some instances are solved using only search space splitting. For those instances, *Hybrid* has the

same performance as *Info*. Even though our portfolio version might not be as tuned as the one presented in ManySAT, there is a clear improvement in the number of solutions found by *Hybrid* in comparison to *Info*. Therefore, even if the set of solvers that compose our portfolio approach could be improved, consequently improving the results of *Pfolio*, it should also improve the results obtained by *Hybrid*. This shows that by using both our techniques we can improve the performance of split space search for parallel SAT solving and outperform a pure portfolio approach.

VII. Conclusions and Future Work

Parallel approaches will likely be the future in the field of SAT solving as many chip manufactures are turning from single core to multicore processors. Utilizing the extra power of these CPUs, regardless a single workstation or a cluster of workstations is used, will therefore be a fundamental issue.

This paper proposes to extend a method based on the VSIDS heuristic to find the initial set of partition variables. This technique allows to choose better initial partition variables. Additionally, this paper also introduces a hybrid approach between search space splitting and portfolio. A hybrid approach allows to reduce load balancing issues of search space splitting as well as to change to a portfolio approach when a cutoff is reached. Experimental results show that the proposed method to find the initial set of partition variables can improve the performance of the solver. The results also show that a hybrid approach can lead to better results than search space splitting or portfolio. This provides a strong stimulus for further exploration of hybrid solutions since they can improve the current state-of-the-art parallel SAT solvers.

SAT4J// is the first parallel SAT solver for a multicore architecture implemented in Java. Due to the popularity of SAT4J and to multicore machines being widespread, it is useful to have a parallel version of SAT4J since it can improve SAT4J performance.

As future work, we propose to extend the use of the VSIDS heuristic of all threads to guide the search during runtime. Additionally, the hybrid heuristic should be further developed. In order to reduce the cache misses of SAT4J//, we will implement the cache conscious data structures proposed by Chu et al. [9]. Finally, clause sharing between threads will be implemented. Clause sharing should further boost the performance of our hybrid approach.

Acknowledgment

This work was partially supported by FCT under research projects PTDC/EIA/76572/2006 and PTDC/EIA-CCO/102077/2008 and INESC-ID multiannual funding through the PIDDAC program funds.

References

[1] D. L. Berre, "SAT4J Library," http://www.sat4j.org.

[2] A. Biere, "PicoSAT Essentials," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 4, pp. 75–97, 2008.

[3] W. Blochinger, "Towards robustness in parallel SAT solving," in *ParCo Conference*, 2005, pp. 301–308.

[4] W. Blochinger, C. Sinz, and W. Küchlin, "Parallel Propositional Satisfiability Checking with Distributed Dynamic Learning," *Journal of Parallel Computing*, vol. 29, no. 7, pp. 969–994, 2003.

[5] M. Böhm and E. Speckenmeyer, "A Fast Parallel SAT-Solver - Efficient Workload Balancing," *Annals of Mathematics and Artificial Intelligence*, vol. 17, no. 3-4, pp. 381–400, 1996.

[6] L. Bordeaux, Y. Hamadi, and H. Samulowitz, "Experiments with Massively Parallel Constraint Solving," in *IJCAI*, 2009, pp. 443–448.

[7] W. Chrabakh and R. Wolski, "GridSAT: A Chaff-based Distributed SAT Solver for the Grid," in *Supercomputing Conference*, 2003, pp. 37–49.

[8] G. Chu and P. J. Stuckey, "PMiniSAT: A parallelization of MiniSAT 2.0," *SAT Race, Solver Description*, 2008.

[9] G. Chu, A. Harwood, and P. J. Stuckey, "Cache Conscious Data Structures for Boolean Satisfiability Solvers," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 99–120, 2009.

[10] F. Corblin, L. Bordeaux, Y. Hamadi, E. Fanchon, and L. Trilling, "A SAT-based Approach to Decipher Gene Regulatory Networks," in *Integrative Post-Genomics Conference*, 2007.

[11] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *Journal of the ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[12] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *SAT Conference*, 2003, pp. 502–518.

[13] Y. Feldman, N. Dershowitz, and Z. Hanna, "Parallel Multithreaded Satisfiability Solver: Design and Implementation," *ENTCS*, vol. 128, no. 3, pp. 75–90, 2005.

[14] S. L. Forman and A. M. Segre, "NAGSAT: A Randomized, Complete, Parallel Solver for 3-SAT," in *SAT Conference*, 2002, pp. 236–243.

[15] L. Gil, P. Flores, and L. M. Silveira, "PMSat: a parallel version of MiniSAT," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 71–98, 2008.

[16] Y. Hamadi, S. Jabbour, and L. Sais, "ManySAT: a Parallel SAT Solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 245–262, 2009.

[17] B. Jurkowiak, C. M. Li, and G. Utard, "Parallelizing Satz Using Dynamic Workload Balancing," in *SAT Conference*, 2001, pp. 205–211.

[18] M. Lewis, T. Schubert, and B. Becker, "Multithreaded SAT Solving," in *ASP-DAC Conference*, 2007, pp. 926–931.

[19] M. Luby, A. Sinclair, and D. Zuckerman, "Optimal Speedup of Las Vegas Algorithms," *Journal of Information Processing Letters*, vol. 47, no. 4, pp. 173–180, 1993.

[20] J. P. Marques-silva and K. A. Sakallah, "GRASP - A Search Algorithm for Propositional Satisfiability," *IEEE Transactions on Computers*, vol. 48, pp. 506–521, 1999.

[21] M. W. Moskewicz and C. F. Madigan, "Chaff: Engineering an Efficient SAT Solver," in *Design Automation Conference*, 2001, pp. 530–535.

[22] L. D. Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS Conference*, 2008, pp. 337–340.

[23] K. Ohmura and K. Ueda, "c-SAT: A Parallel SAT Solver for Clusters," in *SAT Conference*, 2009, pp. 524–537.

[24] K. Pipatsrisawat and A. Darwiche, "A Lightweight Component Caching Scheme for Satisfiability Solvers," in *SAT Conference*, 2007, pp. 294–299.

[25] S. Plaza, I. Kountanis, Z. Andraus, V. Bertacco, and T. Mudge, "Advances and Insights into Parallel SAT Solving," in *Internacional Workshop on Logic & Synthesis*, 2006, pp. 188–194.

[26] T. Schubert, M. Lewis, and B. Becker, "PaMira - A Parallel SAT Solver with Knowledge Sharing," in *Workshop on Microprocessor Test and Verification*, 2005, pp. 29–36.

[27] ——, "PaMiraXT: Parallel SAT Solving with Threads and Message Passing," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 203–222, 2009.

[28] D. Singer and A. Monnet, "JaCk-SAT: A New Parallel Scheme to Solve the Satisfiability Problem (SAT) Based on Join-and-Check," in *Parallel Processing and Applied Mathematics*, 2008, pp. 249–258.

[29] C. Sinz, W. Blochinger, and W. Küchlin, "PaSAT - Parallel SAT-Checking with Lemma Exchange: Implementation and applications," *Electronic Notes in Discrete Mathematics*, vol. 9, pp. 205–216, 2001.

[30] P. Vander-Swalmen, G. Dequen, and M. Krajecki, "A Collaborative Approach for Multi-Threaded SAT Solving," *International Journal of Parallel Programming*, vol. 37, no. 3, pp. 324–342, 2009.

[31] H. Zhang, M. P. Bonacina, M. Paola, Bonacina, and J. Hsiang, "PSATO: a Distributed Propositional Prover and Its Application to Quasigroup Problems," *Journal of Symbolic Computation*, vol. 21, pp. 543–560, 1996.