Deterministic Parallel (DP)2LL

Youssef Hamadi, Said Jabbour, Cédric Piette, Lakhdar Saïs

Deterministic Parallel DPLL (DP^2LL)

MSR-TR-2011-47

Youssef Hamadi[1,2], Said Jabbour[3], Cédric Piette[3], and Lakhdar Saïs[3]

[1] Microsoft Research, Cambridge United Kingdom
[2] LIX École Polytechnique, F91128 Palaiseau, France
youssefh@microsoft.com
[3] Université Lille-Nord de France, Artois,
CRIL, CNRS UMR 8188, F-62307 Lens
{jabbour,piette,sais}@cril.fr

Abstract. Parallel Satisfiability is now recognized as an important research area. The wide deployment of multicore platforms combined with the availability of open and challenging SAT instances are behind this recognition. However, the current parallel SAT solvers suffer from a non-deterministic behavior. This is the consequence of their architectures which rely on weak synchronizing in an attempt to maximize performance. This behavior is a clear downside for practitioners, who are used to both runtime and solution reproducibility. In this paper, we propose the first Deterministic Parallel DPLL engine. It is based on a state-of-the-art parallel portfolio architecture and relies on a controlled synchronizing of the different threads. Our experimental results clearly show that our approach preserves the performance of the parallel portfolio approach while ensuring full reproducibility of the results.

Keywords: SAT solving, Parallelism

1 Introduction

Parallel SAT solving has received a lot of attention in the last three years. This comes from several factors like the wide availability of cheap multicore platforms combined with the relative performance stall of sequential solvers. Unfortunately, the demonstrated superiority of parallel SAT solvers comes at the price of non reproducible results in both runtime and reported solutions. This behavior is the consequence of their architectures which rely on weak synchronizing in an attempt to maximize performance. Today SAT is the workhorse of important application domains like software and hardware verification, theorem proving, computational biology, and automated planning. These domains need reproducibility, and therefore cannot take advantage of the improved efficiency coming from parallel engines. For instance, a software verification tool based on a parallel SAT engine can report different bugs within different runtimes when executed against the same piece of code; an unacceptable situation for a software verification engineer. More broadly in science and engineering, reproducibility of the

results is fundamental, and in order to benefit to a growing number of application domains parallel solvers have to be made deterministic. More specifically, evaluating or comparing highly non-deterministic SAT solvers is clearly problematic. This problem led the organizers of the recent SAT race and competitions to explicitly mention their preferences for deterministic solvers and to introduce specific rules for the evaluation of parallel SAT solvers. For example, we can find the following statement at the SAT race 2008 "*In order to obtain reproducible results, SAT solvers should refrain from using non-deterministic program constructs as far as possible. For many parallel solver implementations it is very hard to achieve reproducible runtimes. Therefore, this requirement does not apply to the parallel track. As runtimes are highly deviating for parallel SAT solvers, each solver was run three times on each instance. An instance was considered solved, if it could be solved in at least one of the three runs of a solver*". For the parallel track, different evaluation rules are used in the SAT 2009 competition and SAT race 2010, illustrating the recurring problems of the evaluation of parallel SAT solvers.

In this work, we propose a deterministic parallel SAT solver. Its results are fully reproducible both in terms of reported solution and runtime, and its efficiency is equivalent to a state-of-the-art non-deterministic parallel solver. It defines a controlled environment based on a total ordering of solvers' interactions through synchronization barriers. To maximize efficiency, information exchange (conflict-clauses) and check for termination are performed on a regular basis. The frequency of these exchanges greatly influences the performance of our solver. The paper explores the trade off between frequent synchronizing which allows the fast integration of foreign conflict-clauses at the cost of more synchronizing steps, and infrequent synchronizing which avoids costly synchronizing at the cost of delayed foreign conflict-clauses integration.

The paper is organized as follows. We detail the main features of modern sequential and parallel SAT solvers in Section 2 and provides motivation of this work in Section 3. Our main contribution is depicted in Section 4. In Section 5, our implementation is fully evaluated. Before the conclusive discussion in Section 7, a comprehensive overview of previous works is given in Section 6.

2 Technical Background

2.1 Modern SAT solvers

The SAT problem consists in finding an assignment to each variable of a propositional formula expressed in conjunctive normal form (CNF). Most of the state-of-the-art SAT solvers are based on a reincarnation of the historical Davis, Putnam, Logemann, and Loveland procedure, commonly called DPLL [1]. A modern sequential DPLL solver performs a backtrack search; selecting at each node of the search tree, a decision variable which is set to a Boolean value. This assignment is followed by an inference step that deduces and propagates some forced unit literal assignments. This is recorded in the implication graph, a central data-structure, which records the partial assignment together with its implications. This branching process is repeated until finding a feasible assignment (model), or reaching a conflict. In the first case, the formula is answered to be satisfiable, and the model is reported, whereas in the second case, a conflict clause

(called asserting clause) is generated by resolution following a bottom-up traversal of the implication graph [2,3]. The learning process stops when a conflict clause containing only one literal from the current decision level is generated. Such a conflict clause (or learnt clause) expresses that such a literal is implied at a previous level. The solver backtracks to the implication level and assigns that literal to *true*. When an empty conflict clause is generated, the literal is implied at level 0, and the original formula can be reported unsatisfiable.

In addition to this basic scheme, modern solvers use additional components such as an activity based heuristics, and a restart policy. Let us give some details on these last two important features. The activity of each variable encountered during each resolution process is increased. The variable with greatest activity is selected to be assigned next. This corresponds to the so-called VSIDS variable branching heuristic [3]. During branching after a certain amount of conflicts, a cutoff limit is reached and the search is restarted. The evolution of the cutoff value is controlled by a restart policy (e.g. [4]). Thanks to the activity of the variables cumulated during the previous runs, the objective of the restarts is to focus early (top of the tree) on important areas of the search space. This differs from the original purposes behind randomization and restarts introduced in [5]. For an extensive overview of current techniques to solve SAT, the reader is refered to [6].

2.2 Parallel SAT solvers

There are two approaches to parallel SAT solving. The first one implements the historical divide-and-conquer idea, which incrementally divides the search space into subspaces, successively allocated to sequential DPLL workers. These workers cooperate through some load balancing strategy which performs the dynamic transfer of subspaces to idle workers, and through the exchange of interesting learnt clauses [7,8].

The parallel portfolio approach was introduced in 2008. It exploits the complementarity between different sequential DPLL strategies to let them compete and cooperate on the same formula. Since each worker deals with the whole formula, there is no need to introduce load balancing overheads, and cooperation is only achieved through the exchange of learnt clauses. With this approach, the crafting of the strategies is important, especially with a small number of workers. In general, the objective is to cover the space of good search strategies in the best possible way.

3 Motivations

As mentioned above, all the current parallel SAT solvers suffer from a non-deterministic behavior. This is the consequence of their architectures which rely on weak synchronizing in an attempt to maximize performance. Conflict-clauses sharing usually exploited in parallel SAT solvers to improve their performance accentuates even more their non-deterministic behavior i.e., non reproducibility in terms of runtime, reported solutions or unsatisfiability proofs.

To illustrate the variations in terms of solutions and runtimes, we ran several times the same parallel solver `ManySAT` (version 1.1) [9] on all the satisfiable instances used

in the SAT Race 2010 and report several measures. Each instance was tested 10 times and the time limit for each run was set to 40 minutes. The obtained results are depicted in Table 3. For each instance, we give the number of variables ($nbVars$), the number of successful runs ($nbModels$) and the number of different models (*diff*) obtained by those runs. Two models are different if their hamming distance is not equal to 0. We also report the normalized average hamming distance $n\bar{H} = \frac{\bar{H}}{nbVars}$ where $\bar{H}$ is the average of the pairwise hamming distance between the different models.

Instance	**nbVars**	**nbModels** (**diff**)	**n**$\bar{H}$	**avgTime** (σ)
12pipe_bug8	117526	10 (1)	0	2.63 (53.32)
ACG-20-10p1	381708	10 (10)	1.42	1452.24 (40.61)
AProVE09-20	33054	10 (10)	33.84	19.5 (9.03)
dated-10-13-s	181082	10 (10)	0.67	6.25 (9.30)
gss-16-s100	31248	10 (1)	0	38.77 (18.75)
gss-19-s100	31435	10 (1)	0	441.75 (35.78)
gss-20-s100	31503	10 (1)	0	681 (58.27)
itox_vc1138	150680	10 (10)	26.62	0.65 (22.99)
md5_47_4	65604	10 (10)	34.8	173.9 (31.03)
md5_48_1	66892	10 (10)	34.76	704.74 (74.65)
md5_48_3	66892	10 (10)	34.16	489.02 (68.96)
safe-30-h30-sat	135786	10 (10)	22.32	0.37 (0.79)
sha0_35_1	48689	10 (10)	33.18	45.4 (21.88)
sha0_35_2	48689	10 (10)	33.25	61.65 (29.93)
sha0_35_3	48689	10 (10)	32.76	72.21 (21.93)
sha0_35_4	48689	10 (10)	33.2	105.8 (35.22)
sha0_36_5	50073	10 (10)	34.19	488.16 (58.58)
sortnet-8-ipc5-h19-sat	361125	4 (4)	15.86	2058.39 (47.5)
total-10-19-s	331631	10 (10)	0.5	5.31 (6.75)
UCG-20-10p1	259258	10 (10)	2.12	768.17 (31.63)
vmpc_27	729	10 (2)	2.53	11.95 (32.62)
vmpc_28	784	10 (2)	3.67	34.61 (25.92)
vmpc_31	961	8 (1)	0	583.36 (88.65)

Table 1. Non-deterministic behavior of `ManySAT`

As expected, the solver exhibits a non-deterministic behavior in term of reported models. Indeed, on many instances, one can see that each run leads to a different model. These differences are illustrated in many cases by a large $n\bar{H}$. For example, on the $sha0\_*\_*$ familly, the number of different models is 10 and the $n\bar{H}$ is around 30% i.e., the models differ in the truth value of about 30% of the variables. Finally, the average time ($avgTime$) and the standard deviation (σ) illustrate the variability of the parallel solver in term of solving time. The $12pipe\_bug8$ instance illustrates an extreme case. Indeed, to find the same model (*diff* =1) the standard deviation of the runtime is about

Algorithm 1: Deterministic Parallel DPLL

```
   Data: A CNF formula F;
   Result: true if F is satisfiable; false otherwise
1  begin
2      <inParallel, 0 ≤ i < nbCores>
3          answer[i] = search(core_i) ;
4      for (i = 0; i < nbCores; i++) do
5          if (answer[i]! = unknown) then
6              return answer[i];
7  end
```

53.32 seconds while the average runtime value is 2.63 seconds. This first experiment clearly illustrates to which extent the non-deterministic behavior of parallel SAT solvers can influence the non reproducibility of the results.

4 Deterministic Parallel DPLL

In this section, we present the first deterministic portfolio based parallel SAT solver. As sharing clauses is proven to be important for the efficiency of parallel SAT solving, our goal is to design a deterministic approach while maintaining at the same time clause sharing. To this end, our determinization approach is first based on the introduction of a barrier directive (<barrier>) that represents a synchronization point at which a given thread will wait until all the other threads reach the same point. This barrier is introduced to synchronize both clauses sharing between the different computing units and termination detection (Section 4.1). Secondly, to enter the barrier region, a synchronization period for clause sharing is introduced and dynamically adjusted (Section 4.2).

4.1 Static Determinization

Let us now describe our determinization approach of non-deterministic portfolio based parallel SAT solvers. Let us recall that a portfolio based parallel SAT solver runs different incarnations of a DPLL-engine on the same instance. Lines 2 and 3 of the algorithm 1 illustrate this portfolio aspect by running in parallel these different search engines on the available cores. To avoid non determinism in term of a reported solution or an unsatisfiability proof, a global data structure called $answer$ is used to record the satisfiability answer of these different cores. The different threads or cores are ordered according to their threads ID (from 0 to nbCores-1). Algorithm 1 returns the result obtained by the first core in this ordering who answered the satisfiability of the formula (lines 4-6). This is a necessary but not a sufficient condition for the reproducibility of the results. To achieve a complete determinization of the parallel solver, let us take a closer look to the DPLL search engine associated to each core (Algorithm 2). In addition to the usual description of the main component of DPLL based SAT solvers, we can see that two

Algorithm 2: search(core$_i$)

```
   Data: A CNF formula F;
   Result: answer[i] = true if F is satisfiable; false if F is unsatisfiable, unknown
           otherwise
 1 begin
 2     nbConflicts=0;
 3     while (true) do
 4         if (!propagate()) then
 5             nbConflicts++;
 6             if (topLevel) then
 7                 answer[i]= false;
 8                 goto barrier_1;
 9             learntClause=analyze();
10             exportExtraClause(learntClause);
11             backtrack();
12             if (nbConflicts % period == 0) then
13                 barrier_1: <barrier>
14                 if (∃j|answer[j]! = unknown) then
15                     return answer[i];
16                 updatePeriod();
17                 importExtraClauses();
18                 <barrier>
19         else
20             if (!decide()) then
21                 answer[i]= true;
22                 goto barrier_1;
23 end
```

successive synchronization barriers (<*barrier*>, lines 13 and 18) are added to the algorithm. To understand the role of these synchronizing points, we need to note both their placement inside the algorithm and the content of the region circumscribed by these two barriers. First, the barrier labeled $barrier_1$ (line 13) is placed just before any thread can return a final statement about the satisfiability of the tested CNF. This barrier prevents cores from returning their solution (i.e. model or refutation proof) in an anarchic way, and forces them to wait for each other before stating the satisfiability of the formula (line 14 and 15). This is why the search engine of each core goes to the first barrier (labeled $barrier_1$) when the unsatisfiability is proved (backtrack to the top level of the search tree, lines 6-8), or when a model is found (lines 20-22). At line 14, if the satisfiability of the formula is answered by one of the cores ($\exists j|answer[j]! = unknown$), the algorithm returns its own $answer[i]$. If no thread can return a definitive answer, they all share information by importing conflict clauses generated by the other cores during the last period (line 17). After each one of them has finished to import clauses (second barrier, line 18), they continue to explore the search space looking for a solution. This second synchronization barrier is integrated to prevent each core from leaving the synchronization region before the others. In other words, when a given core enter this

second barrier, it waits for all other cores until all of them have finished importing the foreign clauses. As different clauses ordering will induce different unit propagation ordering and consequently different search trees, the clauses learnt by the other cores are imported (line 17) while following a fixed order of the cores w.r.t. their thread ID.

To complete this detailed description, let us just specify the usual functions of the search engine. First, the $propagate()$ function (line 4) applies classical unit propagation and returns *false* if a conflict occurs, and *true* otherwise. In the first case, a clause is learnt by the function $analyze()$ (line 9), such a clause is added to the formula and exported to the other cores (line 10, $exportExtraClause()$ function). These learned clauses are periodically imported in the synchronization region (line 17). In the second case, the $decide()$ function choses the next decision variable, assigns it and returns *true*, otherwise it returns *false* as all the variable are assigned i.e., a model is found.

Note that to maximize the dynamics of information exchange, each core can be synchronized with the other ones after each conflict, importing each learnt clause right after it has been generated. Unfortunately, this solution proves empirically inefficient, since a lot of time is wasted by the thread waiting. To avoid this problem, we propose to only synchronize the threads after some fixed number of conflicts $period$ (line 10). This approach, called $(DP)^2LL\_static(period)$, does not update the period during search (no call to the function $updatePeriod()$, line 16). However, even if we have the "optimal" value of the parameter $period$, the problem of thread waiting at the synchronization barrier can not be completely eliminated. Indeed, as the different cores usually present different search behaviors (different search strategies, different size of the learnt databases, etc.), using the same value of the $period$ for all cores, leads inevitably to wasted waiting time at the first barrier.

In the next section, we propose a speed-based dynamic synchronization approach that exploits the current size of the formula to deduce a specific value of the $period$ for each core (call to $updatePeriod()$ function, line 16).

4.2 Speed-based Dynamic Synchronization

As mentioned above, the DPLL search engines used by the different cores develop different search trees, hence thread waiting can not be avoided even if the static value is optimally fine tuned. In this section, we propose a speed-based dynamic synchronization of the value of the period. Our goal is to reduce as much as possible the time wasted by the different cores at the synchronization barrier. The time needed by each core to perform the same number of conflicts is difficult to estimate in advance; however we propose an interesting approximation measure that exploits the current state of the search engine. As decisions and unit propagations are two fundamental operations that dominate the SAT solver run time, estimating their cost might lead us to a better approximation of the progression speed of each solver. More precisely, as the different cores are run on the same formula, to measure the speedup of each core, we exploit the size of its current learnt set of clauses as an estimation of the cost of these basic operations. Consequently, our speed-based dynamic synchronization of the period is a function of this important information.

Let us formally describe our approach. In our dynamic synchronization strategy, for each core or computing unit u_i, we consider a synchronization-time sequence as a

set of steps t_i^k with $t_i^0 = 0$ and $t_i^k = t_i^{k-1} + period_i^k$ where $period_i^k$ represents the time window defined in term of number of conflicts between t_i^{k-1} and t_i^k. Obviously, this synchronization-time sequence is different for all the computing units u_i ($0 \leq i < nbCores$). Let us define Δ_i^k as the set of clauses currently in the leant database of u_i at step t_i^k. In the sequel, when there is no ambiguity, we sometimes note t_i^k simply k.

Let us now formally describe the dynamic computation of these synchronization-time sequences. Let $m = max_{\forall u_i}(|\Delta_i^k|)$, where $0 \leq i < nbCores$, be the size of the largest learnt clauses database and $S_i^k = \frac{|\Delta_i^k|}{m}$ the ratio between the size of the learnt clauses database of u_i and m. This ratio S_i^k represents the speedup of u_i. When this ratio tends to one, the progression of the core u_i is closer to the slowest core, while when it tends to 0, the core u_i progresses more quickly than the slowest one. For $k = 0$ and for each u_i, we set $period_i^0$ to α, where α is a natural number. Then, at a given time step $k > 0$, and for each u_i, the next value of the period is computed as follows: $period_i^{k+1} = \alpha + (1 - S_i^k) \times \alpha$, where $0 \leq i < nbCores$. Intuitively, the core with the highest speedup S_i^k (tending to 1) should have the lowest period. On the contrary, the core with the lowest speedup S_i^k (tending to 0) should have the highest value of the period.

5 Evaluation

period	**solving time**	**waiting time**	**waiting/solving time (in %)**
static(1)	10,276	4,208	40.9
static(100)	9,715	2,559	26.3
static(10000)	9,124	1,605	17.6

Table 2. Waiting time w.r.t. period synchronizing

All the experimentations have been conducted on Intel Xeon 3GHz under Linux CentOS 4.1. (kernel 2.6.9) with a RAM limit of 2GB. Our deterministic DPLL algorithm has been implemented on top of the portfolio-based parallel solver `ManySAT` (version 1.1). The timeout was set to 900 seconds for each instance, and if no answer was delivered within this amount of time, the instance was said unsolved. We used the 100 instances proposed during the recent `SAT Race 2010`, and we report for each experiment the number of solved instances (x-axis) together with the total needed time (y-axis) to solve them. Each parallelized solver is running using 4 threads. Note that in the following experiments, we consider the real time used by the solvers, instead of the classic CPU time. Indeed, in most architectures, the CPU time is not increased when the threads are asleep (e.g. waiting time at the barriers), so taking the CPU time into account would give an illegitimate substantial advantage to our deterministic solvers.

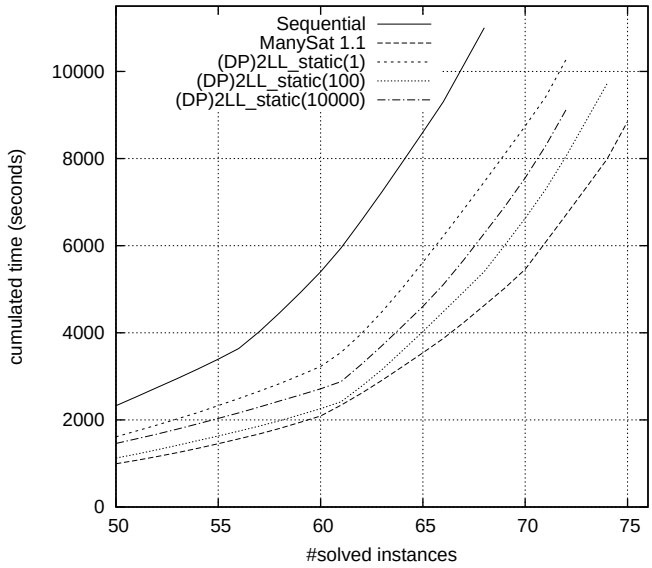

Fig. 1. Performances using static synchronizing

5.1 Static Period

In a first experiment, we have evaluated the performance of our Deterministic Parallel DPLL ($(DP)^2$LL) with various static periods. Figure 1 presents the obtained results. First, a sequential version of the solver has been used. Unsurprisingly, this version obtains the worst global results by only solving 68 instances in more than 11,000 seconds. This result enables to show the improvement obtained by the use of parallelized engines. We also report the results obtained by the non-deterministic solver `ManySAT 1.1`. Note that as shown in Section 3, executing several times this version may lead to different results. This non-deterministic solver has been able to solve 75 instances within 8,850 seconds. Next, we ran a deterministic version of `ManySAT` where each core synchronizes with the other ones after each clause generation ((DP)2LL_static(1)). We can observe that the synchronization barrier is computationally heavy. Indeed, the deterministic version is clearly less efficient than the non-deterministic one, by only solving 72 instances out of 100 in more than 10,000 seconds.

This negative result is mainly due to the time wasted by the cores waiting for each others on a (very) regular basis. To overcome this issue, we also tried to synchronize the different threads only after a given number of conflicts (100, 10000). Figure 1 shows that those versions outperform the "period=1" one, but they stay far from the results obtained by the non-deterministic version.

The barriers that enable our algorithm to be deterministic are clearly a drawback to its efficiency. Therefore, we evaluated the amount of time spent by the threads waiting for each others. In Table 2, we report the rate of time spent waiting for synchronization with respect to the total solving time for the different versions of (DP)2LL. As the results show, this waiting time can be really significant when waiting after each conflict.

Indeed, in this case, more than 40% of the time is spent by the threads waiting for each other. As expected, this rate is reduced when the synchronization is achieved less often, but is never insignificant. Note that even if the "period=10000" version wastes less time than "period=100" waiting, it obtains worst global results by only solving 72 instances. The "period=100" version obtains the best results (discarding the non-deterministic version). Based on extensive experiments, this parameter seems to be a good tradeoff between the time spent at the barrier, and the dynamics of information exchange. Indeed, we believe that information is exchanged too late in the "period=10000" version, which explains its less satisfying behavior.

Accordingly, we tried to reduce the waiting time of the threads by adopting a Speed-based dynamic strategy (presented Section 4.2). Empirical results about this dynamic technique are presented in the next section.

5.2 Dynamic Period

In a second experiment, we tried to empirically evaluate our dynamic strategy. We compare the results of this version with the ones obtained by `ManySAT 1.1`, and with the results of the best static version (100), and of the sequential one too. The results are reported in Figure 2. The dynamic version is run with parameter $\alpha = 300$.

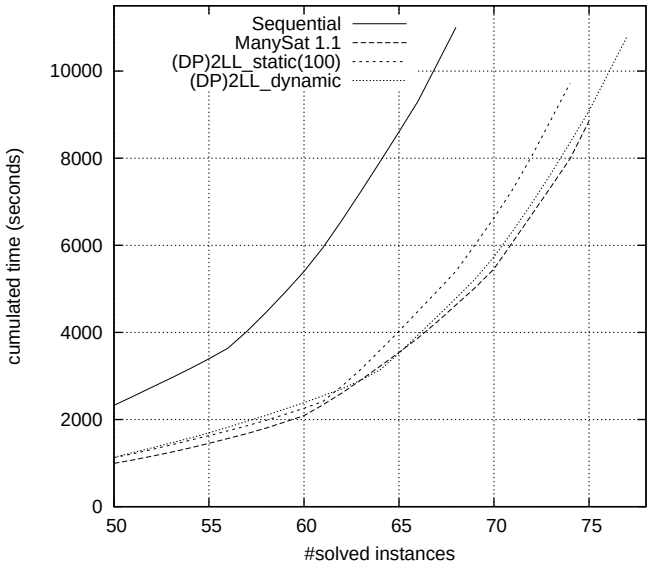

Fig. 2. Performance using dynamic synchronizing

This experiment empirically confirms the intuition that each core should have a different period value, w.r.t. the size of its own learnt clauses database, which heuristically indicates its unit propagation speed. Indeed, we can observe in Figure 2 that the "solving curve" of this dynamic version is really close to the one of `ManySat 1.1`. This

means that the 2 solvers are able to solve about the same amount of instances within about the same time. Moreover, this adaptive version is able to solve 2 more instances than the non-deterministic one, which makes it the most efficient version tested during our experiments.

6 Previous Works

In the last two years, portfolio based parallel solvers became prominent, and we are not aware of a recently developed divide-and-conquer approach (the latest being [8]). We describe here all the parallel solvers qualified during the 2010 SAT Race[4]. We believe that these parallel portfolio approaches represent the current state-of-the-art in parallel SAT.

In `plingeling`, [10] the original SAT instance is duplicated by a boss thread and allocated to worker threads. The strategies used by these workers are mainly differentiated by the amount of pre-processing, random seeds, and variables branching. Conflict clause sharing is restricted to units which are exchanged through the boss thread. This solver won the parallel track of the 2010 SAT Race.

`ManySAT` [9] was the first parallel SAT portfolio. It duplicates the instance of the SAT problem to solve, and runs independent SAT solvers differentiated on their restart policies, branching heuristics, random seeds, conflict clause learning, etc. It exchanges clauses through various policies. Two versions of this solver were presented at the 2010 SAT Race, they finished respectively second and third.

In `SArTagnan`, [11] different SAT algorithms are allocated to different threads, and differentiated with respect to, restarts policies, and VSIDS heuristics. Some threads apply a dynamic resolution process [12], or exploit reference points [13]. Some others try to simplify a shared clauses database by performing dynamic variable elimination or replacement. This solver finished fourth.

In `Antom`, [14] the SAT algorithms are differentiated on decision heuristic, restart strategy, conflict clause detection, lazy hyper binary resolution [12], and dynamic unit propagation lookahead. Conflict clause sharing is implemented. This solver finished fifth.

7 Discussion

In this paper, we tackle the important issue of non determinism in parallel SAT solvers by proposing $(DP)^2$LL, a first implementation of a deterministic parallelized procedure for SAT. To this purpose, a simple but efficient idea is presented; it mainly consists in introducing two synchronization barriers to the algorithm. We propose different synchronizing strategies, and show that this deterministic version proves empirically very efficient; indeed, it can compete against a recent non-deterministic algorithm.

This first implementation opens many interesting research perspectives. First, the barrier added to the main loop of the parallelized CDCL for making the algorithm deterministic can be seen as a drawback of the procedure. Indeed, every thread that has

[4] `http://baldur.iti.uka.de/sat-race-2010`

terminated its partial computation has to wait for the all other threads to finish. Nevertheless, our synchronization policy proves effective while keeping the heart of the parallelized architecture: clause exchanges. Moreover, we think that it is possible to take even better advantage of this synchronization step. New ways for the threads of execution to interact can be imagined at this particular point; we think that this is a promising path for future research.

References

1. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Communications of the ACM **5**(7) (1962) 394–397
2. Marques-Silva, J., Sakallah, K.: GRASP - A New Search Algorithm for Satisfiability. In: ICCAD. (1996) 220–227
3. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: ICCAD. (2001) 279–285
4. Biere, A.: Adaptive restart strategies for conflict driven SAT solvers. In: SAT. (2008) 28–33
5. Gomes, C., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: AAAI. (1998) 431–437
6. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (2009)
7. Chrabakh, W., Wolski, R.: GrADSAT: A parallel SAT solver for the grid. Technical report, UCSB Computer Science (2003)
8. Chu, G., Stuckey, P.J.: Pminisat: a parallelization of minisat 2.0. Technical report, SAT Race (2008)
9. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. Journal on Satisfiability, Boolean Modeling and Computation **6** (2009) 245–262
10. Biere, A.: Lingeling, plingeling, picosat and precosat at SAT race 2010. Technical Report 10/1, FMV Reports Series (2010)
11. Kottler, S.: SArTagnan: solver description. Technical report, SAT Race 2010 (July 2010)
12. Biere, A.: Lazy hyper binary resolution. Technical report, Dagstuhl Seminar 09461 (2009)
13. Kottler, S.: SAT solving with reference points. In: SAT. (2010) 143–157
14. Schubert, T., Lewis, M., Becker, B.: Antom: solver description. Technical report, SAT Race (2010)