HAL
archives-ouvertes.fr

Improving Parallel Local Search for SAT

Alejandro Arbelaez, Youssef Hamadi

▶ **To cite this version:**

Alejandro Arbelaez, Youssef Hamadi. Improving Parallel Local Search for SAT. Learning and Intelligent OptimizatioN Conference LION5, Jan 2011, Rome, Italy. inria-00563775

HAL Id: inria-00563775
https://hal.inria.fr/inria-00563775

Submitted on 7 Feb 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Improving Parallel Local Search for SAT

Alejandro Arbelaez[1], Youssef Hamadi[2,3]

[1] Microsoft-INRIA joint-lab, Orsay France
alejandro.arbelaez@inria.fr
[2] Microsoft Research, Cambridge United Kingdom
[3] LIX École Polytechnique, F91128 Palaiseau, France
youssefh@microsoft.com

Abstract. In this work, our objective is to study the impact of knowledge sharing on the performance of portfolio-based parallel local search algorithms. Our work is motivated by the demonstrated importance of clause-sharing in the performance of complete parallel SAT solvers. Unlike complete solvers, state-of-the-art local search algorithms for SAT are not able to generate redundant clauses during their execution. In our settings, each member of the portfolio shares its best configuration (i.e., one which minimizes conflicting clauses) in a common structure. At each restart point, instead of classically generating a random configuration to start with, each algorithm aggregates the shared knowledge to carefully craft a new starting point. We present several aggregation strategies and evaluate them on a large set of problems.

Keywords: local search, SAT solving, parallelism.

1 Introduction

Complete parallel solvers for the propositional satisfiability problem have received significant attention recently. These solvers can be divided into two main categories the classical divide-and-conquer model and the portfolio-based approach. The first one, typically divides the search space into several sub-spaces while the second one lets algorithms compete on the original formula [1]. Both take advantage of the modern SAT solving architecture [2], to exchange the conflict-clauses generated in the system and improve the overall performance.

This push towards parallelism in complete SAT solvers has been motivated by their practical applicability. Indeed, many domains, from software verification to computational biology and automated planning rely on their performance. On the contrary, since local search techniques only outperform complete ones on random SAT instances, their parallelizing has not received much attention so far. The main contribution on the parallelization of local search algorithms for SAT solving basically executes a portfolio of independent algorithms which compete without any communication between them. In our settings, each member of the portfolio shares its best configuration (i.e., one which minimizes the number of conflicting clauses) in a common structure. At each restart point, instead of

classically generating a random configuration to start with, each algorithm aggregates the shared knowledge to carefully craft a new starting point. We present several aggregation strategies and evaluate them on a large set of instances.

This paper is organized as follows: background material is presented in section 2. Section 3 describes previous work on parallel SAT and cooperative algorithms. Section 4 presents our methodology and our aggregation strategies, section 5 evaluates them, and section 6 presents some concluding remarks and future directions of research.

2 Background

2.1 The Propositional Satisfiability Problem

The Propositional Satisfiability Problem (SAT) can be represented by a pair $\langle \mathcal{V}, \mathcal{C} \rangle$ where, $\mathcal{V}$ indicates a set of boolean variables and $\mathcal{C}$ a set of clauses representing a propositional *conjunctive-normal form* (CNF).

Solving a SAT problem involves finding a solution i.e., a truth assignment for each variable such that all clauses are satisfied, or demonstrating that no such assignment can be found. If a solution exist the problem is stated as satisfied and unsatisfied otherwise. Currently, there are two well established techniques for solving SAT problems, complete and incomplete techniques [3], the former is developed on top of the DPLL algorithm. It combines a tree-based search with constraint propagation, conflict-clause learning, and intelligent backtracking while the latter is based on local search algorithms to quickly find a truth assignment for a given satisfiable instance [4].

2.2 Local Search for SAT

Algorithm 1 describes a traditional local search algorithm for SAT solving, it starts with a random truth assignment for each variable in the formula F (*initial-configuration* line 2), and the key point of local search algorithms is depicted in lines (3-9) here the algorithm flips the most appropriate variable candidate until a solution is found or a given number of flips is reached (MaxFlips), after this process the algorithm restarts itself with a new (fresh) random configuration.

As one may expect, a critical part of the algorithm is the variable selection function (*select-variable*) which indicates the next variable to be flipped in the current iteration of the algorithm. Broadly speaking, there are two main categories of variable selection functions, the first one motivated by the GSAT algorithm [5] is based on the following score function:

$$score(x) = make(x) - break(x)$$

Intuitively $make(x)$ indicates the number of clauses that are currently satisfied but flipping x become unsatisfied, and $break(x)$ indicates the number of clauses that are unsatisfied but flipping x become satisfied. In this way, local search algorithms select the variable with minimal score value (preferably with

negative value), because flipping this variable would most likely increase the chances of solving the instance.

The second category of variable selection functions is the Walksat-based one [6] which includes a diversification strategy in order to avoid local minimums, this extension selects, at random, an unsatisfied clause and then picks a variable from that clause. The variable that is generally picked will result in the fewest previously satisfied clauses becoming unsatisfied, with some probability of picking one of the variables at random.

Algorithm 1 Local Search For SAT (CNF formula F, Max-Flips, Max-Tries)

```
1: for try := 1 to Max-Tries do
2:    A := initial-configuration(F).
3:    for flip := 1 to Max-Flips do
4:       if  A satisfies F then
5:          return A
6:       end if
7:       x := select-variable(A)
8:       A := A with x flipped
9:    end for
10: end for
11: return 'No solution found'
```

2.3 Refinements

This section briefly reviews the main characteristics of state-of-the-art local search solvers for SAT solving. As pointed out above these algorithms are developed to deal with the variable selection function and are mainly devoted to avoid getting trapped in a local minima. This way, the following list describes several well-known mechanisms for selecting the most appropriate variable to flip at a given state of the search.

- *Novelty* [7] firstly selects an unsatisfied clause c and from c selects the best v_{best} and second best v_{2best} variable candidates, if v_{best} is not the latest flipped variable in c then Novelty flips this variable, otherwise v_{2best} is flipped with a given probability p and v_{best} with probability $1-p$. Important extensions to this algorithm can be found in *Novelty+*, *Novelty++* and *Novelty+p*.
- G^2WSAT [8] (G2) uses a list of promising decreasing variables to determine the next variable to be flipped and if the list of decreasing variables is empty the algorithm uses *Novelty++* as a backup heuristic. $G^2WSAT+p$ (G2+p) uses a similar strategy that G^2WSAT however in this case the backup solver is *Novelty+p*.
- *Scaling and Probabilistic Smoothing* (SAPS) [9] implements a multiplicative increase rule to dynamically modify the penalty for unsatisfied clauses and with a given probability P_{smooth} this penalty value is adjusted according to a given smoothing factor ρ.

– *Pure Additive Weighting Scheme* (PAWS) [10] implements an additive increase rule to dynamically modify the penalty for unsatisfied clauses and if a given clause penalty has been changed a given number of times this penalty value is adjusted.
– *Reactive SAPS* (RSAPS) [9] extends SAPS by adding an automatic tuning mechanism to identify suitable values for the smoothing factor ρ.
– *Adaptive Novelty+* (AN+) [11] uses an adaptive mechanism to properly tune the noise parameter of Walksat-like algorithms (e.g, *Novelty+*)
– *Adaptive* G^2 *WSAT* (AG2) [12] aims to integrate an adaptive noise mechanism into the G^2 *WSAT* algorithm. Similarly, *Adaptive* G^2 *WSAT+p* (AG2+p) also uses an adaptive noise mechanism into the G^2 *WSAT+p* algorithm.

3 Previous Work

In this section, we review the most important contributions devoted to parallel SAT solving and cooperative algorithms.

3.1 Complete Methods for Parallel SAT

GrADSAT [13] is a parallel SAT solver based on the zChaff solver and equipped with a master-slave architecture in which the problem space is divided into subspaces, these sub-spaces are solved by independent zChaff clients and learnt clauses whose size (i.e., number of literals) is less or equal to a given limit are exchanged between clients. The technique organizes load-balancing through a work stealing technique which allows the master to push work to idle clients.

Unlike other parallel solvers for SAT which divide the initial problem space into sub-spaces, ManySAT [1] is a portfolio-based parallel solver where independent DPLL algorithms are launched in parallel to solve a given problem instance. Each algorithm in the portfolio implements a different and complementary restart strategy, polarity heuristic and learning scheme. In addition, the first version of the algorithm exchanges learnt clauses whose size is less or equal to a given limit. It is worth mentioning that ManySAT won the 2008 SAT Race, the 2009 SAT Competition and was placed second in the 2010 SAT Race (all these in the parallel track). Interestingly all the algorithms successfully qualified in the 2010 parallel track were based on a Portfolio architecture.

In [14] the authors proposed a hybrid algorithm which starts with a traditional DPLL algorithm to divide the problem space into sub-spaces. Each sub-space is then allocated to a given local search algorithm (Walksat).

3.2 Incomplete Methods for Parallel SAT

PGSAT [15] is a parallel version of the GSAT algorithm. The entire set of variables is randomly divided into τ subsets and allocated to different processors. In this way at each iteration, if no global solution has been obtained, the i^{th}

processor uses the GSAT score function (see section 2) to select and flip the best variable for the i^{th} subset. Another contribution to this parallelization architecture is described in [16] where the authors aim to combine PGSAT and random walk, therefore at each iteration, with a given probability wp an unsatisfiable clause c is selected and a random variable from c is flipped and with probability 1-wp. PGSAT is used to flip τ variables in parallel at a cost of reconciling partial configurations to test if a solution has been found.

gNovelty+ (v.2) [17], belongs to the portfolio approach, this algorithm executes n independent copies of the *gNovelty+ (v.2)* algorithm in parallel, until at least one of them finds a solution or a given timeout is reached. This algorithm was the only parallel local search solver presented in the *random* category of the 2009 SAT Competition[4]

In [18], Kroc et al., studied the application of a parallel hybrid algorithm to deal with the max-SAT problem. This algorithm combines a complete solver (minisat) and an incomplete one (Walksat). Broadly speaking both solvers are launched in parallel and minisat is used to guide Walksat to promising regions of the search space by means of suggesting values for the selected variables.

3.3 Cooperative Algorithms

In [19] a set of algorithms running in parallel exchange hints (i.e., partial valid solutions) to solve hard graph coloring instances. To this end, they share a blackboard where they can write a hint with a given probability q and read a hint with a given probability p.

In [20] the authors studied a sequential cooperative algorithm to deal with the office-space-allocation problem. In this paper cooperation takes place when a given algorithm is not able to improve its own best solution, at this point a cooperative mechanism is used to explore suitable partial solutions stored by individual heuristics. This algorithm is also equipped with a diversification strategy to explore different regions of the search space.

Although *Averaging in Previous Near Solutions* [21] is not a cooperative algorithm by itself, this method is used to determine the initial configuration for the i^{th} restart in the GSAT algorithm. Broadly speaking, the initial configuration is computed by performing a bitwise average between variables of the best solution found during the previous restart ($restart_{i-1}$) and two restarts before ($restart_{i-2}$). That is, variables with same values in both configurations are reused, and the extra set of variables are initialized with random values. Since overtime, configurations with a few conflicting clauses tend to become similar, all the variables are randomly initialized after a given number of restarts.

4 Knowledge Sharing in Parallel Local Search for SAT

Our objective is to extend a parallel portfolio of state-of-the-art local search solvers for SAT with knowledge sharing or cooperation. Each algorithm is going

[4] http://www.satcompetition.org/2009/

to share with others the best configuration it has found so far with its respective cost (number of unsatisfied clauses) in a shared pair $\langle M, C \rangle$.

$$M = \begin{pmatrix} X_{11} & X_{12} & \dots & X_{1n} \\ X_{21} & X_{22} & \dots & X_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ X_{c1} & X_{c2} & \dots & X_{cn} \end{pmatrix} \qquad C = [C_1, C_2, \dots, C_c]$$

Where n indicates the total number of variables of the problem and c indicates the number of local search algorithms in the portfolio. In the following we are associating local search algorithms and processing cores. Each element X_{ji} in the matrix indicates the i^{th} variable of the best configuration found so far by the j^{th} core. Similarly, the j^{th} element in C indicates the cost for the respective configuration in M.

These best configurations can be exploited by each local search to build a new initial configuration. In the following, we propose seven strategies to determine the initial configuration (cf. function *initial-configuration* in algorithm 1).

4.1 Using Best Known Configurations

In this section, we propose three methods to build the new initial configuration *init* by aggregating best known configurations. In this way, we define $init_i$ for all the variables $X_i, i \in [1..n]$ as follows:

1. *Agree*: if there exists a value v such that $v{=}X_{ji}$ for all $j \in [1..c]$ then $init_i{=}v$, otherwise a random value is used.
2. *Majority*: if there exists two values v and v' such that $|\{X_{ji} = v | j \in [1..c]\}| > |\{X_{ji} = v' | j \in [1..c]\}|$ then $init_i{=}v$, otherwise a random value is used.
3. *Prob*: $init_i{=}1$ with probability $p_{ones}{=}\frac{ones}{c}$ and $init_i{=}0$ with probability $1 - p_{ones}$, where $ones = |\{X_{ji} = 1 | j \in [1..c]\}|$.

4.2 Weighting Best Known Configurations

In contrast with our previous methods where all best known solutions are treated equally important, the methods proposed in this section use a weighting mechanism to consider the cost of best known configurations. The computation of the initial configuration *init* uses one of the following two weighting systems: *Ranking* and *Normalized Performance*, where values from better configurations are most likely to be used.

Ranking This method sorts the configurations of the shared matrix from worst to best according to their cost. The worst ranked one gets weight of 1 (i.e., $RankW_1{=}1$), and the best ranked c (i.e., $RankW_c{=}c$).

Normalized Performance This method assigns weights ($NormW$) considering a normalized value of the number of unsatisfied clauses of the configuration:

$$NormW_j = \frac{|\mathcal{C}| - C_j}{|\mathcal{C}|}$$

Using the previous two weighting mechanisms, we define the following four extra methods to determine initial configurations.

To this end, we define $\Phi(val, Weight) = \sum_{k \in \{j | X_{ji} = val\}} Weight_k$.

1. *Majority RankW*: if there exists two values v and v' such that $\Phi(v, RankW) > \Phi(v', RankW)$ then $init_i$=v, otherwise a random value is used.
2. *Majority NormalizedW*: if there exists two values v and v' such that $\Phi(v, NormW) > \Phi(v', NormW)$ then $init_i$=v, otherwise a random value is used.
3. *Prob RankW*: $init_i$=1 with probability P_{Rones}=$\frac{Rones}{Rones+Rzeros}$ and $init_i$=0 with probability 1-P_{Rones}, where $Rones$=$\Phi(1, RankW)$ and $Rzeros$=$\Phi(0, RankW)$.
4. *Prob NormalizedW*: $init_i$=1 with probability P_{Nones}=$\frac{Nones}{Nones+Nzeros}$ and $init_i$=0 with probability 1-P_{Nones}, where $Nones$=$\Phi(1, NormW)$ and $Nzeros$=$\Phi(0, NormW)$

4.3 Restart Policy

As mentioned earlier on, shared knowledge is exploited when a given algorithm is restarted. At this point the current working configuration of a given algorithm is re-initialized according to a given aggregation strategy. However, it is important to restrict cooperation since it adds overheads and more importantly tend to generate similar configurations. In this context, we propose a new restart policy to avoid re-initializing the working configuration again and again. This new policy re-initializes the working configuration for a given restart (i.e., every MaxFlips) if and only if, performance improvements in best known solutions have been observed during the latest restart window. This new restart policy is formally described in the following definition, where we assume that bc_{ki} is the cost of the best known configuration for a given algorithm i up to the $(k-1)^{th}$ restart.

Definition 1 *At a given restart k for a given algorithm i the working configuration is reinitialized iff there exists an algorithm q such that $bc_{kq} \neq bc_{(k-1)q}$ and $q \neq i$.*

5 Experiments

5.1 Experimental Settings

We conducted experiments using instances from the RANDOM category of the 2009 SAT competition. Since state-of-the-art local search solvers are unable to

solve UNSAT instances, we filtered out these instances. We also removed instances whose status was reported as UNKNOWN in the competition. This way, we collected 359 satisfiable instances.

We decided to build our parallel portfolio on UBCSAT-1.1, a well known local search library which provides efficient implementation of the latest local search for SAT algorithms [22]. We did preliminary experiments to extract from this library the 8 algorithms which perform best on our set of problems. From that, we defined the following three baseline portfolio constructions where algorithms are independent searches without cooperation. The first one *pcores-PAWS* uses p copies of the best single algorithm (PAWS), the second portfolio *4cores-No sharing* uses the best subset of 4 algorithms (PAWS, G2+p, AG2, AG2+p) and the last one *8cores-No sharing* uses all the 8 algorithms (PAWS, G2+p, AG2, AG2+p, G2, SAPS, RSAPS, AN+). All the algorithms were used with their default parameters, and without any restart. Indeed these techniques are equipped with important diversification strategies and usually perform better when the restart flag is switched off (i.e., MaxFlips=∞).

On the other hand, the previous knowledge aggregation mechanisms were built on top of a portfolio with 4 algorithms (same algorithms as *4cores-No sharing*) and a portfolio with 8 algorithms (same algorithms as *8cores-No sharing*). There, we used the modified restart policy described in section 4.3 with $MaxFlips$ set to 10^6.

All tests were conducted on a cluster of 8 Linux Mandriva machines with 8 GB of RAM and two quad-core (8 cores) 2.33 Ghz Intel Processors. In all the experiments, we used a timeout of 5 minutes (300 seconds) for each algorithm in the portfolio, so that for each experiment the total CPU time was set to $c \times 300$ seconds, where c indicates the number of algorithms in the portfolio.

We executed each instance 10 times (each time with a different random seed) and reported two metrics, the *Penalized Average Runtime* (PAR) [23] which computes the average runtime overall instances, but where unsolved instances are considered as 10$\times$ the cutoff time, and the runtime for each instance which is calculated as the median across the 10 runs. Overall, our experiments for these 359 SAT instances took 187 days of CPU time.

5.2 Practical Performances with 4 Cores

Fig. 1 shows the results of each aggregation strategy using a portfolio with 4 cores, comparatively to the 4 cores baseline portfolios. The x-axis gives the number of problems solved and the y-axis presents the cumulated runtime.

As expected, the portfolio with the top 4 best algorithms (*4cores-No Sharing*) performs better (309) that the one with 4 copies of the best algorithms (*4cores-PAWS*) (275).

The performance of the portfolios with knowledge sharing is quite good. Overall, it seems that adding a weighting mechanism can often hurt the performance of the underlying aggregation strategy. Among the weighting options, it seems that the Normalized Performance performs better. The best portfolio

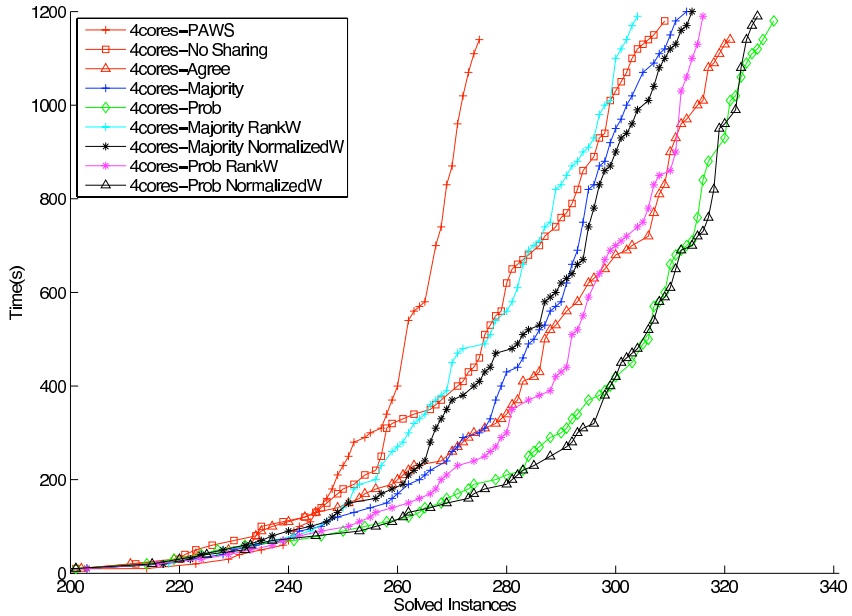

Fig. 1. Number of solved instances using 4 cores in a given amount of time

implements the *Prob* strategy without any weighting (329). This corresponds to a gain of 20 problems against the corresponding *4cores-No Sharing* baseline.

A detailed examination of *4cores-Prob* and *4cores-No Sharing* is presented in Figs. 2 and 3. These Figures show, respectively, a runtime and a best configuration cost comparison. In both figures, points below (resp. above) the diagonal line indicate that *4cores-Prob* performs better (resp. worse) than *4cores-No Sharing*. In the runtime comparison, we observe that easy instances are correlated as they require few steps to be solved, and for the remaining set of instances *4cores-Prob* usually exhibits a better performance. On the other hand, the second figure shows that when the instances are not solved, the median cost of the best configuration (number of unsatisfied clauses) found by *4cores-Prob* is usually better than for *4cores-No Sharing*. Notice that some points are overlapped because the two strategies reported the same cost.

All the experiments using 4 cores are summarized in Table 1, reporting for each portfolio the number of solved instances (#solved), the median time across all instances (median time), the *Penalized Average Runtime* (PAR) and the total number of instances that timed out in all the 10 runs (never solved). These results confirm that sharing best known configurations outperforms independent

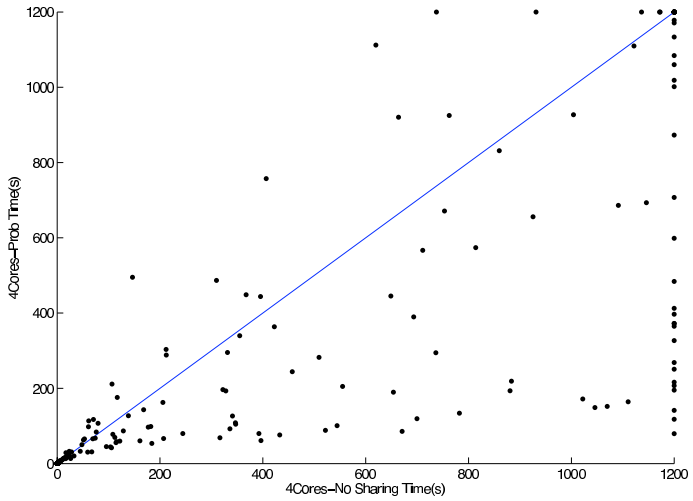

Fig. 2. Runtime comparison, each point indicates the runtime to solve a given instance using *4cores-Prob* (y-axis) and *4cores-No Sharing* (x-axis)

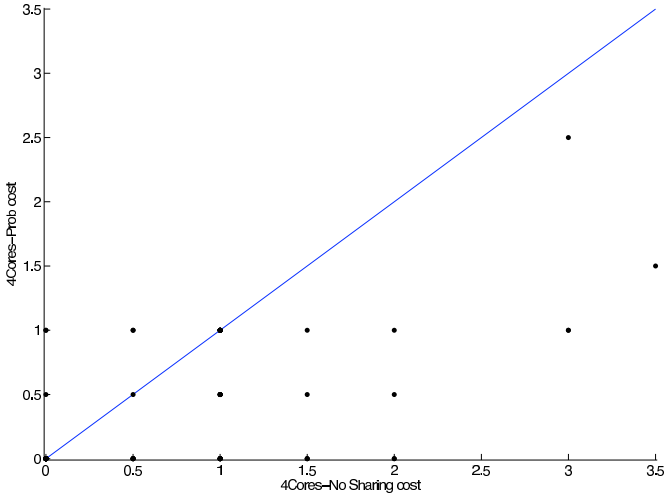

Fig. 3. Best configuration cost comparison on unsolved instances. Each point indicates the best configuration (median) cost of a given instance using *4cores-Prob* (y-axis) and *4cores-No Sharing* (x-axis)

searches, for instance *4cores-Prob* and *4cores-Prob NormalizedW* solved respectively 20 and 17 more instances than *4cores-No Sharing* and all the cooperative strategies (except *4cores-Majority RankW*) exhibit better PAR. Interestingly, 4cores-PAWS exhibited the best median runtime overall the experiments with 4 cores, this fact suggests that PAWS by itself is able to quickly solve an important number of instances. Moreover, only 2 instances timeout in all the 10 runs for *4cores-Agree* and *4cores-Prob NormalizedW* against 7 for *4cores-No Sharing*. Notice that this Table also includes *1core-PAWS*, the best sequential local search on this set of problems. The PAR score for *1core-PAWS* is lower than the other values of the table because this portfolio uses only 1 algorithm, therefore the timeout is only 300 seconds, while 4 cores portfolios use a timeout of 1200 seconds.

Strategy	#solved	median time	PAR	never solved
1core-PAWS	249	1.76	911.17	71
4cores-PAWS	275	**1.63**	2915.19	61
4cores-No Sharing	309	2.19	1901.00	7
4cores-Agree	321	2.54	1431.33	**2**
4cores-Majority	313	2.53	1724.94	11
4cores-Prob	**329**	2.51	**1257.93**	4
4cores-Majority RankW	304	2.47	1930.61	11
4cores-Majority NormalizedW	314	2.48	1807.42	9
4cores-Prob RankW	316	2.53	1621.33	7
4cores-Prob NormalizedW	326	2.50	1261.82	**2**

Table 1. Overall evaluation using 4 cores

5.3 Practical Performances with 8 Cores

We now move on to portfolios with 8 cores. The results of these experiments are depicted in Fig. 4 indicating the total number of solved instances within a given amount of time. As in previous experiments, we report the results of baseline portfolios *8cores-No Sharing* and *8cores-PAWS*, and in this case we focus the experiments on *Prob* and *Prob NormalizedW* (the best two strategies using 4 cores). We can observe that the cooperative portfolios largely outperform the non-cooperative ones.

Table 2 summarizes these results, and once again it includes the best individual algorithm running in a single core. We can remark that *8cores-Prob* and *8cores-Prob NormalizedW* solve respectively 24 and 16 more instances than *8cores-No Sharing*. Furthermore, it shows that knowledge sharing portfolios are faster than individual searches, with a PAR of 3743.63 seconds for *8cores-No Sharing* against respectively 2247.97 for *8cores-Prob* and 2295.99 for *8cores-Prob NormalizedW* . Finally, it is also important to note that only 1 instance

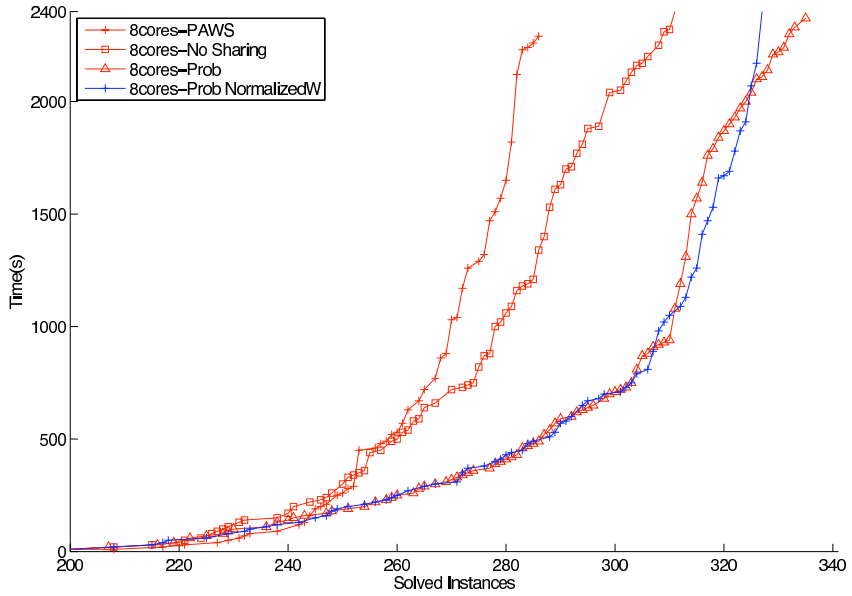

Fig. 4. Number of solved instances using 8 cores in a given amount of time

timed out in all the 10 runs for *8cores-Prob NormalizedW* against 8 for *8cores-No Sharing*.

Strategy	#solved	median time	PAR	never solved
1core-PAWS	249	1.76	911.17	71
8cores-PAWS	286	**2.00**	5213.84	56
8cores-No Sharing	311	2.33	3743.63	8
8cores-Prob	**335**	2.45	**2247.97**	2
8cores-Prob NormalizedW	327	2.47	2295.99	**1**

Table 2. Overall evaluation using 8 cores

Extensive experimental results presented in this paper show that *Prob* (4 and 8 cores) exhibited the overall best performance. We attribute this to the fact that the probability component of this method balances the exploitation of best solutions found so far with the exploration of other values for the variables, helping in this way, to diversify the new starting configuration.

5.4 Hardware Impact

In this section, we wanted to assess the inherent slowdown caused by increased cache, and bus contingency when more processing cores are used at the same time. To this end we decided to run our PAWS baseline portfolio where each independent algorithm uses the same random seed on respectively 1, 4 and 8 cores. Since all the algorithms are executing the same search, this experiment measures the slowdown caused by hardware limitations. The results are presented in Fig. 5.

The first case executes a single copy of PAWS with a timeout of 300 seconds, the second case executes 4 parallel copies of PAWS with a timeout of 1200 seconds (4×300) and the third case executes 8 parallel copies of PAWS with a timeout of 2400 seconds (8×300).

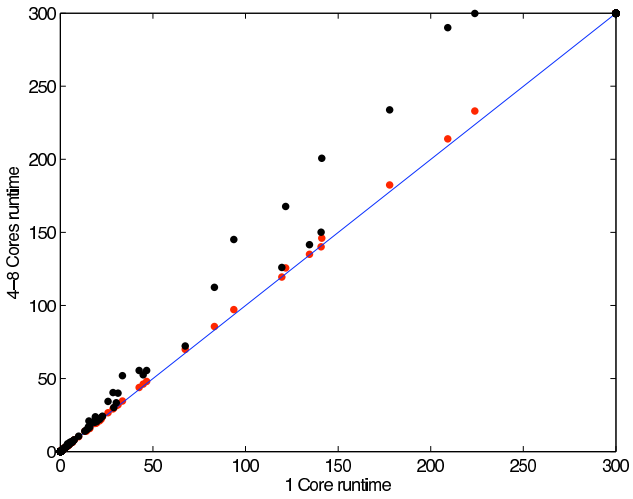

Fig. 5. Runtime comparison using parallel local search portfolios made of respectively 1, 4, and 8 identical copies of PAWS (same random seed). Red points indicate the performance of 4 cores vs 1 core. Black points indicate the performance of 8 cores vs 1 core, points above the blue line indicate that 1 core is faster

Finally, we estimate the runtime of each instance as the median across 10 runs (each time with the same seed) divided by the number of cores. In this figure, it can be observed that the performance overhead is almost not distinguishable between 1 and 4 cores (red points). However, the overhead between 1 and 8 cores is important for difficult instances (black points).

6 Conclusions and Future Work

In this work, our objective was to integrate knowledge sharing strategies in parallel local search for SAT. We were motivated by the recent developments in parallel DPLL solvers. We decided to restrict the information shared to the best configuration found so far by the algorithms in a portfolio. From that we defined several simple knowledge aggregation strategies along a specific lazy restart policy which creates a new initial configuration when a fix cutoff is meet and when the quality of the shared information has been improved.

Extensive experiments were done on a large number of instances coming from the latest SAT competition. They showed that adding the proposed sharing policies improves the performance of a parallel portfolio, this improvement is exhibited in both number of solved instances and the *Penalized Average Runtime* (PAR). It is also reflected in the best configuration cost of problems which could not be solved within the time limit.

We believe that our work represents a very first step in the incorporation of knowledge sharing strategies in parallel local search for SAT. Further work will investigate the use of additional information to exchange, for instance: tabu-list, the age and score of a variable, information on local minima, etc. It should also investigate the best way to integrate this extra knowledge in the course of a given algorithm. As said earlier, state-of-the-art local search perform better when they do not restart. Incorporating extra information without forcing the algorithm to restart is likely to be important.

7 Acknowledgements

We would like to thank Said Jabbour and Ibrahim Abdoulahi for helpful discussions about parallel SAT solving and the anonymous reviewers for their comments which helped to improve this paper.

References

1. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: A Parallel SAT Solver. Journal on Satisfiability, Boolean Modeling and Computation, JSAT **6** (2009) 245–262
2. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th Design Automation Conference (DAC'01). (2001) 530–535
3. Bordeaux, L., Hamadi, Y., Zhang, L.: Propositional Satisfiability and Constraint Programming: A Comparative Survey. ACM Comput. Surv. **38**(4) (2006)
4. Hoos, H.H., Stützle, T.: Local Search Algorithms for SAT: An Empirical Evaluation. J. Autom. Reasoning **24**(4) (2000) 421–481
5. Selman, B., Levesque, H.J., Mitchell, D.G.: A New Method for Solving Hard Satisfiability Problems. In 440-446, ed.: AAAI. (1992)
6. Selman, B., Kautz, H.A., Cohen, B.: Noise Strategies for Improving Local Search. In: AAAI. (1994) 337–343

7. McAllester, D.A., Selman, B., Kautz, H.A.: Evidence for Invariants in Local Search. In: AAAI/IAAI. (1997) 321–326
8. Li, C.M., Huang, W.Q.: Diversification and Determinism in Local Search for Satisfiability. In Bacchus, F., Walsh, T., eds.: SAT'05. Volume 3569 of LNCS., St. Andrews, UK, Springer (June 2005) 158–172
9. Hutter, F., Tompkins, D.A.D., Hoos, H.H.: Scaling and Probabilistic Smoothing: Efficient Dynamic Local Search for SAT. In Hentenryck, P.V., ed.: CP'02. Volume 2470 of LNCS., Ithaca, NY, USA, Springer (September 2002) 233–248
10. Thornton, J., Pham, D.N., Bain, S., Ferreira Jr, V.: Additive versus Multiplicative Clause Weighting for SAT. In McGuinness, D.L., Ferguson, G., eds.: AAAI, San Jose, California, USA, AAAI Press / The MIT Press (July 2004) 191–196
11. Hoos, H.H.: An Adaptive Noise Mechanism for WalkSAT. In: AAAI/IAAI. (2002) 655–660
12. Li, C.M., Wei, W., Zhang, H.: Combining Adaptive Noise and Look-Ahead in Local Search. In Marques-Silva, J., Sakallah, K.A., eds.: SAT'07. Volume 4501 of LNCS., Lisbon, Portugal, Springer (May 2007) 121–133
13. Chrabakh, W., Wolski, R.: GridSAT: A System for Solving Satisfiability Problems Using a Computational Grid. Parallel Computing **32**(9) (2006) 660–687
14. Zhang, W., Huang, Z., Zhang, J.: Parallel Execution of Stochastic Search Procedures on Reduced SAT Instances. In Ishizuka, M., Sattar, A., eds.: Pacific Rim International Conferences on Artificial Intelligence (PRICAI). Volume 2417 of LNCS., Tokyo, Japan, Springer (August 2002) 108–117
15. Roli, A.: Criticality and Parallelism in Structured SAT Instances. In Hentenryck, P.V., ed.: CP'02. Volume 2470 of LNCS., Ithaca, NY, USA, Springer (September 2002) 714–719
16. Roli, A., Blesa, M.J., Blum, C.: Random Walk and Parallelism in Local Search. In: Metaheuristic International Conference (MIC'05), Vienna, Austria (2005)
17. Pham, D.N., Gretton, C.: gNovelty+ (v.2). In: Solver description, SAT competition 2009. (2009)
18. Kroc, L., Sabharwal, A., Gomes, C.P., Selman, B.: Integrating Systematic and Local Search Paradigms: A New Strategy for MaxSAT. In Boutilier, C., ed.: IJCAI, Pasadena, California (July 2009) 544–551
19. Hogg, T., Williams, C.P.: Solving the Really Hard Problems with Cooperative Search. In: AAAI. (1993) 231–236
20. Silva, D.L., Burke, E.K.: Asynchronous Cooperative Local Search for the Office-Space-Allocation Problem. INFORMS Journal on Computing **19**(4) (2007) 575–587
21. Selman, B., Kautz, H.A.: Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems. In: IJCAI. (1993) 290–295
22. Tompkins, D.A.D., Hoos, H.H.: UBCSAT: An Implementation and Experimentation Environment for SLS algorithms for SAT and MAX-SAT. In Hoos, H.H., Mitchell, D.G., eds.: SAT'04. Volume 3542 of LNCS., Vancouver, BC, Canada, Springer (2004) 306–320
23. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Tradeoffs in the Empirical Evaluation of Competing Algorithm Designs. Annals of Mathematics and Artificial Intelligenc (AMAI), Special Issue on Learning and Intelligent Optimization (2010)