This is a post-peer-review, pre-copyedit version
2012 International Conference on Tools for Artificial Intelligence (ICTAI)
doi: 10.1109/ICTAI.2012.17

Massively Parallel Local Search for SAT

Alejandro Arbelaez
University of Tokyo / JFLI,
Tokyo, Japan,
arbelaez@is.s.u-tokyo.ac.jp

Philippe Codognet
JFLI-CNRS / UPMC / University of Tokyo,
Tokyo, Japan,
codognet@is.s.u-tokyo.ac.jp

***Abstract*—Parallel portfolio-based algorithms have become a standard methodology for both complete and incomplete solvers for SAT solving. In this methodology several algorithms explore the search space in parallel, either independently or cooperatively with some communication between the solvers. Unlike previous work where parallel algorithms are limited to few cores (usually up to 16 cores), this work studies the performance of parallel local search for SAT with a large degree of parallelism, up to 256 cores, and compares various cooperation strategies. The strategy with the best performance consists in considering small groups of solvers (e.g. 4 or 8) sharing information and performing no inter-group communication.**

INTRODUCTION

Parallel implementation of local search meta-heuristics have been studied since the early 90's, when multiprocessors started to become available, see [1] for a general survey and concepts, or [2] for basic parallel version of Tabu search, simulated annealing, GRASP, and genetic algorithms. With the availability of clusters in the early 2000's, this domain became again active and further developed in the last years due to the increasing computing power offered by multi-core architectures.

A classical manner to devise parallel SAT solvers is the divide-and-conquer approach [3] where the search space is divided into several sub-spaces, these sub-spaces being then explored in parallel. Another approach for achieving high performance is the parallel portfolio-based search [4], [5], where several algorithms compete and cooperate to solve a given problem instance.

The portfolio-based approach in local search (so-called multi-start or multiple-walk) consists in developing parallel explorations of the search space, either independently or cooperatively with some communication between the parallel processes. The non-cooperative approach has been previously used in the gNovelty+ solver [6], winner of the 2009 SAT competition (parallel track - random category), for a multi-core architecture. Other parallel local search solvers for SAT comprehend PGSAT [7] and MiniWalk [8]. The former can be seen as a parallel version of the GSAT algorithm, where the entire set of variables is divided into subsets which are then allocated to different processors. MiniWalk 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 by suggesting values for the selected variables.

Regarding complete parallel SAT solvers, several multi-core parallel algorithms based on the portfolio paradigm have been developed. For instance, [4], [5] belong to the portfolio architecture where independent DPLL algorithms are launched in parallel to solve a given problem instance. In addition, parallel search engines exchange learned clauses in order to improve performance. Interestingly all the solvers successfully qualified in the 2010 SAT competition (parallel track) were based on a portfolio architecture. Also note that non-portfolio complete parallel solvers have been proposed for a cluster with 64 nodes [9] and a grid system with up to 60 parallel jobs [10].

To the best of our knowledge, this paper represents the first attempt to study the performance of parallel local search for SAT when considering a large degree of parallelism, i.e. a few hundreds of cores. To this end, we present a detailed analysis of portfolios with and without cooperation on a large number of instances coming from the latest SAT competition. In order to take advantage of both approaches, we devise a strategy with small-scale groups of solvers sharing information while no inter-group communication is allowed. This is the strategy whose performance scales reasonably well up to a few hundreds of cores.

The rest of the paper is structured as follows. Section I reviews the key concepts used in the paper, including the satisfiability problem, local search, and parallel local search. Section II details the experimental settings used in the paper, then before the general conclusions and future work, Sections III and IV present extensive experiments of massively parallel local search for SAT.

I. PRELIMINARIES

The satisfiability problem (SAT) consists in determining whether a Boolean formula $\mathcal{F}$ is satisfiable or not. $\mathcal{F}$ is represented by a pair $\langle \mathcal{V}, \mathcal{C} \rangle$, where $\mathcal{V}$ is a set of Boolean variables and $\mathcal{C}$ is a set of clauses in Conjunctive Normal Form (CNF), and each clause c is a disjunction of literals (a variable v or its negation $\overline{v}$). A k-SAT problem indicates that $\mathcal{F}$ contains k literals per clause. For instance, a 3-SAT formula can be represented as follows: $\mathcal{F}$=$(v_{11} \vee v_{12} \vee v_{13}) \wedge (v_{21} \vee v_{22} \vee v_{23}) \wedge \ldots \wedge (v_{n1} \vee v_{n2} \vee v_{n3})$.

Solving $\mathcal{F}$ consists in finding a truth assignment for the variables that satisfies all clauses, or demonstrating that no such assignment can be found. An algorithm for the SAT problem is complete if the algorithm is guaranteed to find a satisfying truth assignment for all SAT problems where such an assignment exists, and is incomplete otherwise. Well-known complete solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and combine tree-base search with constraint propagation, conflict-clause learning,

and intelligent backtracking, on the other hand, current-state-of-the-art incomplete solvers are based on local search to quickly find a truth assignment for the variables that makes the formula true. Alternative, some local search procedures [11] integrate clause learning into local search to solve unsatisfiable instances. In the following, we will limit our attention to incomplete local search.

A. Local Search

Algorithm 1 describes a well-known local search algorithm for SAT. Broadly speaking, the algorithm starts with random values for the variables (line 2), then iteratively selects a variable x (line 7) and changes the truth value of x (line 8). If no solution is obtained after a given number of iterations (i.e. MaxFlips) the algorithm is restarted, i.e. variables are reinitialized with random values (line 2). However, modern local search solvers, such as TNM [12], Sparrow [13], and gNovelty+ [6] have eliminated the restart mechanism from their default configuration as they usually perform better without restarts.

Algorithm 1 Local Search For SAT (CNF formula F, MaxFlips, MaxTries)

```
1: for try := 1 to MaxTries do
2:     A := variable-initialization(F)
3:     for Iteration := 1 to MaxFlips 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'
```

Several algorithms have been proposed in the literature to identify the most appropriate variable at each iteration of the algorithm. Among these, we would like to highlight Sparrow [14], [13], winner of the latest SAT competition (random category), which largely outperformed other participants. This algorithm borrows ideas from gNovelty+, where each clause is equipped with a penalty that is updated whenever there are no promising decreasing variables[1], clause penalties are smoothed in a SAPS-like [16] style with a probability ps and increased in a PAWS-like [17] style with a probability $1\text{-}ps$. Additionally, Sparrow uses a distribution based method, a unique feature, in which a variable is selected with a probability proportional to the score and age of the variables.

In a nutshell, Sparrows maintains a list of promising decreasing variables L, and iteratively selects the best variable from the list. If L is empty, the algorithm selects a random unsatisfied clause c and from c selects a variable using the following probability function p.

[1]The concept of promising variables is fully described in [15]

$$p(x) = \frac{P_s(x).P_a(x)}{\sum_{i=1}^{k} P_s(x_i).P_a(x_i)}$$

Where, k represents the number of literals in c, $P_s(x)$ and $P_a(x)$ control the score and age of the variable as follows: $P_s(x)$=$c_1^{score(x)}$, P_a=$1 + (\frac{age(x)}{c_3})^{c_2}$, score indicates the sum of all unsatisfied clause penalties when flipping x and age indicates the number of iterations in Algorithm 1 since x was last flipped. In addition, whenever there are no elements in L, the penalty for unsatisfied clauses is increased 1 unit with a probability ps and decreased 1 unit with a probability $1\text{-}ps$. In total, the algorithm includes 4 parameters: c_1, c_2, c_3, and ps that need to be fine tuned.

B. Portfolio-based Parallel Local Search

Parallel portfolio-based algorithms have become a standard methodology to solve SAT instances for both complete and incomplete solvers. On one hand, parallel portfolios for complete solvers (e.g. [4], [5]) are based on the DPLL algorithm and combine tree-base search with constraint propagation, conflict-clause learning, and intelligent backtracking. In addition, parallel search engines exchange learned clauses in order to improve performance. While on the other hand, parallel portfolios for incomplete solvers (e.g. [6]) mainly execute different algorithms (or the same one with different random seeds) independently with no cooperation and the global search is stopped as soon as a solution is observed or a given timeout is reached.

Recently, [18] proposed seven strategies in order to exploit cooperation in the context of parallel local search for SAT. In this cooperative framework, each solver shares the best assignment for the variables found so far with all other members of the portfolio to properly craft a new assignment for the variables to restart from. These strategies range from a voting mechanism, where each algorithm in the portfolio suggest a value for each variable, to probabilistic constructions. This way, the *variable-initialization* function (line 2, Algorithm 1) uses cooperation (after the second restart) in lieu of random values for the variables.

In particular, in this paper we focus on *Prob NormalizedW*, which overall exhibited an outstanding performance in [18]. This strategy maintains a pair $\langle M,C\rangle$ as indicated in Figure 1.

$$M = \begin{pmatrix} v_{11} & v_{12} & \dots & v_{1n} \\ v_{21} & v_{22} & \dots & v_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ v_{p1} & v_{p2} & \dots & v_{pn} \end{pmatrix}$$

$$C = [C_1, C_2, \dots, C_p]$$

Figure 1. *Prob NormalizedW* shared information.

Where, the n denotes number of variables in the problem, p the number of cores in the portfolio, and the i^{th} row in M represents the best assignment for the variables found so far

for the i^{th} algorithm in the portfolio, similarly the i^{th} element in C represents the cost (number of unsatisfied clauses) for the i^{th} element in M.

In order to build a new assignment for the variables, *Prob NormalizedW* defines a probability distribution based on two main features; the number of occurrences of variables with positive and negative value and the cost of such assignment. Intuitively, values involved in better truth assignments are most likely to be used. *Prob NormalizedW* is formally described in definition 1.

Definition 1: Prob NormalizedW:
Let us consider a set of clauses $\mathcal{C}$ with n variables. Let p be the number of solvers and $C = [C_1, C_2, \ldots, C_p]$ the costs of the current assignments for each solver. Moreover let us note the value assigned by a solver k to a variable i by v_{ki}.
For a solver k the normalized weight $NormW_k$ of the variable assignment is $NormW_k = \frac{|\mathcal{C}|-C_k}{|\mathcal{C}|}$.
Then for each variable i,
Prob NormalizedW(i=1) = $\frac{\sum_{k\in[1,p]} v_{ki} * NormW_k}{n}$
and Prob NormalizedW(i=0) = 1- Prob NormalizedW(i=1)

Cooperation takes place periodically every MaxFlip iterations, however, if no new information has been observed in M during the previous restart window, the algorithm might restart from the same matrix again and again. In order to avoid this situation, a restart is allowed if and only if activity is observed in M during the previous restart. Otherwise the current restart is ignored. This restart policy is formally described in definition 2, where bc_{ki} denotes the cost of the best known assignment for the variables for a given solver i up to the $(k-1)^{th}$ restart.

Definition 2: At a given restart k for a given algorithm i the current variable assignment is reinitialized iff there exists an algorithm q such that $bc_{kq} \neq bc_{(k-1)q}$ and $q \neq i$.

C. Communication Protocol

Figure 2 depicts the communication protocol used to exchange information between solvers in the portfolio. First, we would like to point out that the communication is asynchronous (or non-blocking), that is, a given solver sends data (i.e. best assignments) and continues the execution of Algorithm 1 without waiting for the data to be transferred.

The automaton has three states `WAIT`, `SEND`, and `SLEEP`. The automaton is initialized in `SLEEP`, in state `SEND` a message with n*+1* elements is sent to all respective solvers, n elements corresponding to the values of the best assignment found so far plus the cost of such assignment. Once all messages have properly arrived to their destinations[2] the state changes to `SLEEP`. Additionally, whenever the solver identifies a new better assignment (after line 8 in Algorithm 1), the automaton goes from any state to `WAIT` and remains in this state until all previous messages from the sending process have been received.

This protocol can be easily included after line 3 in Algorithm 1. However, in order to avoid too much message traffic, the solvers only exchange assignments for the variables every φ iterations. Let us recall than MaxFlips (in Algorithm 1) differs from φ in a way that the former represents the size of the restart window and the latter represents a point to exchange information between solvers in the portfolio.

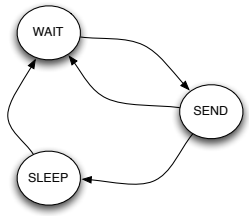


Figure 2. Communication protocol to send a message

II. Experimental Settings

This section presents experiments with massively parallel local search for SAT. In these experiments we consider a collection of 369 known SAT instances from the 2011 SAT competition[3]. These instances are divided into two main groups: large and medium size, each group itself contain sets of k-SAT instances, where k indicates the number of literals for each clause. Large-size represents a set of 100 3-SAT, 50 5-SAT, and 18 7-SAT instances, while medium-size represents a set of 100 3-SAT, 50 5-SAT, and 51 7-SAT instances, and the ratio clauses/variables for large (L) and medium (M) instances is 3-SAT (L: 4.2 - M: 4.26), 5-SAT (L: 20 - M: 21.3), and 7-SAT (L: 20 - M: 89).

All the experiments were performed on the Grid'5000 platform, the french national grid for research, in particular we used a 44-node cluster with 24 cores (2 AMD Opteron 6164 HE processors at 1.7 Ghz) and 44 GB of RAM per node, all nodes being interconnected on a 1 Gb network. Overall, our experiments for these 369 instances took approximately 204 days of CPU time.

We used openMPI to build our parallel solver on top of Sparrow [13], which is implemented in UBCSAT [19]. Additionally, we equipped this parallel portfolio with the cooperative strategy *Prob-NormalizedW* described in Section I-B and the communication protocol indicated in Section I-C with φ=10^4.

We ran each solver 10 times on each instance (each time with a different random seed) using a timeout of 5 minutes wall-clock time. We report three metrics, the *Penalized Average Runtime* (PAR-10) [20] which computes the average runtime over all instances, but where unsolved instances are considered as $10\times$ the cutoff time; the number of solved instances within the time limit, where the runtime for a single

[2]Whenever a solver i receives an assignment from another solver j, i sends back a reception acknowledge message to j.

[3]www.satcompetition.org/2011

instance is calculated as the median across the 10 runs; and the speedup against a sequential version of Sparrow, which is calculated by means of the PAR-10 using the following formula: speedup=PAR_1/PAR_c, where c indicates the number of cores.

It is worth noting that, due to the Sparrow solver largely outperforms other local search algorithms in the 2011 SAT competition –*random* instances–, in this paper we only use independent copies of Sparrow, unlike [18] where the best portfolio construction was selecting different and complementary algorithms. Moreover, we use the parameters proposed for the competition, that is 3-SAT (c_1: 2.15, c_2: 4, c_3: 100000, and ps: 0.347), 5-SAT (c_1: 2.85, c_2: 4, c_3: 75000, and ps: 1.0), and 7-SAT (c_1: 6.5, c_2: 4, c_3: 100000, and ps: 0.83). Notice that the value for these parameters were obtained after extensive offline sessions of automatic and human tuning. It is worth noticing that Sparrow removed the restart flag for the competition, taking this into account, in this paper portfolios without cooperation do not consider restarts, cooperative portfolios set the MaxFlips parameter to 10^6.

The robustness of Sparrow against a portfolio of different algorithms was demonstrated in the latest competition where this algorithm outperformed CSLS [21] (silver medal and best parallel algorithm for the *random* category). CSLS is a cooperative solver which is built on top of UBCSAT, the algorithms in the portfolio correspond to the best virtual portfolio construction for instances of the 2009 SAT competition, and implements the cooperative strategy *Prob NormalizedW* detailed in Section I-B.

III. Experiments with basic parallel schemes

We start our analysis with Figure 3, where we display the number of solved instances for a portfolio without cooperation. As can be observed, this strategy scales reasonably well in a sense that increasing the number of cores increases the number of solved instances. For instance, using 256 cores helps to solve 85 (about 31%) more instances than the sequential algorithm.

Table I presents further details of portfolios without cooperation reporting the number of solved instances (Sol), the PAR-10 metric, and the overall speedup on the entire set of instances. As can be observed, the performance improvement is also observed in the PAR-10. For instance, using 256 cores is on average 618.7 seconds faster than the sequential solver and the speedup factor is 5.94. However, this table also suggests that the parallel algorithm is reaching a performance plateau where little improvement would be observed by dramatically increasing the number of cores. Notice that the PAR-10 improvement after 128 cores (doubling the number of cores) is only 27 seconds.

Now we switch our attention to Figure 4, where we display the results for cooperative portfolios. In this case, the best performance is obtained using 8 and/or 16 cores by solving 338 instances. However, after this point performance degrades as the number of cores increases. For instance, a portfolio with 256 cores requires about 40 seconds to solve 180 instances, while portfolios using up to 64 cores solve 180 instances in less than 10 seconds.

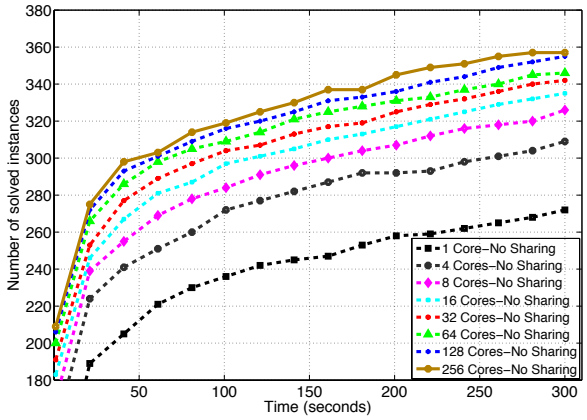


Figure 3. Performance of parallel portfolios without cooperation

Table II shows detailed results for cooperative portfolios. Here, we observe a good speedup factor only up to 16 cores. However, a cooperative portfolio using 256 cores performs worse than the sequential algorithm and the speedup factor is 0.95. We attribute this to the fact that increasing the number of cores includes more diversification, so that algorithms in the portfolio might restart with quasi-random assignment for the variables (see [22] for a full description of this phenomenon using up to 8 cores). Moreover, the communication overhead is another factor to consider. To given an order of idea, a portfolio with 256 cores needs to send and receive 65280 messages (255 $\times$ 256) to acquire the first best assignment for the variables for each algorithm in the portfolio.

Finally, it is worth mentioning that we have also experimented on crafted instances (see [23] for detailed results). However, the performance of portfolios with and without cooperation is quite similar for this family of problems and the speedup factor using 256 cores is only 1.38 when compared against sequential search. We attribute the difference in the speedup factor between crafted and random instances to the fact that solutions for random instances might be more uniformly distributed in the search space, while this is not necessarily the case for crafted problems.

IV. Cooperation with groups of solvers

Considering the scalability issues of the cooperative approach, we propose to limit cooperation to groups of solvers, that is the entire set of available solvers is divided into groups of n solvers, each group of solvers exploits cooperation (*Prob NormalizedW* in this paper) and no knowledge sharing is allowed between solvers of different groups.

Figure 5 shows the performance of using groups of solvers, that is the total number of solved instances (Figure 5(a)) and the penalized average runtime (Figure 5(b)) when limiting knowledge sharing to groups of 2, 4, 8, and 16 solvers (note the

Total cores	Sol	PAR	Speedup
1	272	743.7	–
4	309	467.9	1.58
8	326	365.4	2.03
16	335	281.6	2.64
32	342	235.1	3.16
64	346	187.8	3.96
128	355	152.0	4.89
256	**357**	**125.0**	**5.94**

Table I
PARALLEL PORTFOLIO WITHOUT COOPERATION

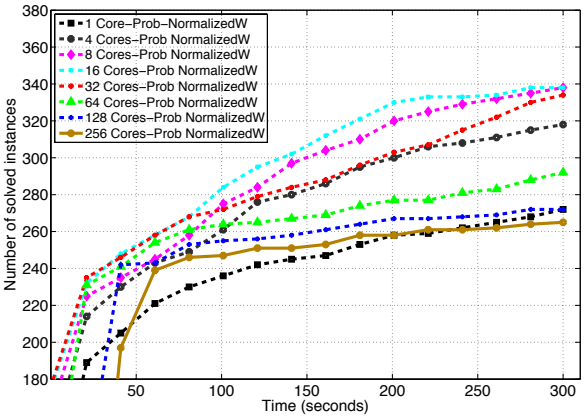


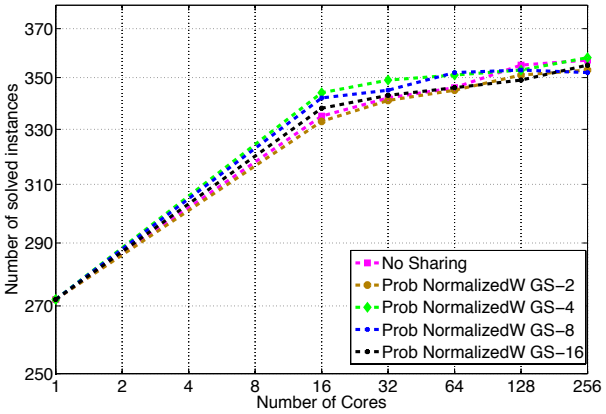
Figure 4. Performance of portfolios with cooperation

log-scale). In this figure, we also display a portfolio without cooperation (No Sharing), and a cooperative portfolio, *Prob NormalizedW GS-G* where G indicates the number of solvers for each group. Let us start by pointing out that using 2 solvers per group exhibits a poor performance and it is not good enough to outperform the non-cooperative solver. Using 16 solvers per group scales as good as a portfolio without cooperation up to 64 cores. On the other hand, using 4 and 8 solvers per group seem to be a good approach to build a scalable solver, in fact, the performance in both cases is better than the non-cooperative solver up to 128 cores, and for 256 cores there is little difference in the penalized average runtime.

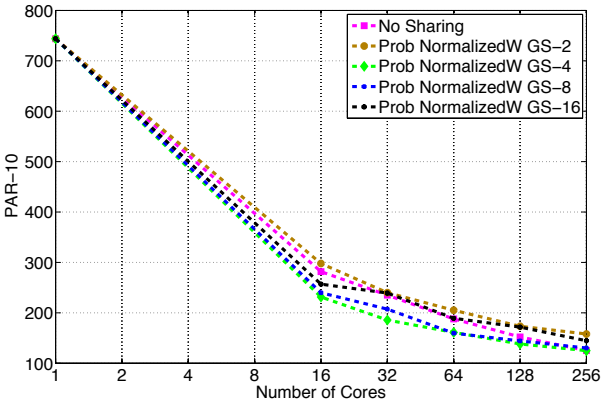
Table III summarizes our experiments when using groups of

Total cores	Sol	PAR	Speedup
1	272	743.7	–
4	318	408.9	1.81
8	**338**	306.9	2.42
16	**338**	**256.8**	**2.89**
32	334	320.9	2.31
64	292	601.6	1.23
128	272	726.0	1.02
256	265	779.8	0.95

Table II
PARALLEL PORTFOLIO USING *Prob NormalizedW*



(a) Number of solved instances for a cooperative portfolio and groups of solvers.



(b) PAR-10 for a cooperative portfolio and groups of solvers.

Figure 5. Number of solved instances and PAR-10 for a cooperative portfolio and groups of solvers.

Total cores	GS	Sol	PAR	Speedup
1	–	272	743.7	–
16	2	333	297.8	2.50
	4	**344**	**231.4**	**3.21**
	8	342	239.9	3.10
	16	338	256.8	2.89
32	2	341	239.9	3.10
	4	**349**	**185.7**	**4.00**
	8	345	207.3	3.59
	16	343	239.2	3.11
64	2	345	205.1	3.63
	4	351	161.1	4.62
	8	**352**	**159.7**	**4.66**
	16	346	189.2	3.93
128	2	351	173.4	4.29
	4	**353**	**138.1**	**5.38**
	8	**353**	144.2	5.16
	16	349	171.7	4.33
256	2	353	158.0	4.71
	4	**358**	**124.9**	**5.95**
	8	352	130.2	5.71
	16	355	144.6	5.14

Table III
PARALLEL PORTFOLIO WITH COOPERATION AND GROUPS OF SOLVERS

Total Cores	GS	Large				Medium			
		3-SAT (100)	5-SAT (50)	7-SAT (18)	Total (168)	3-SAT (100)	5-SAT (50)	7-SAT (51)	Total (201)
16	NS	82 586.0	43 499.2	9 1122.3	134 617.6	100 0.3	50 1.3	51 1.4	201 0.8
	NG	93 296.8	35 834.9	9 1285.7	137 562.9	100 0.6	50 1.2	51 1.6	201 1.0
	2	81 576.5	41 582.5	10 1270.3	132 652.6	100 0.5	50 1.6	51 1.9	201 1.1
	4	95 265.8	38 724.6	10 1243.1	143 507.1	100 0.6	50 1.4	51 1.6	201 1.0
	8	95 268.2	37 792.2	9 1213.5	141 525.4	100 0.5	50 1.8	51 1.9	201 1.2
	16	93 296.8	35 834.9	9 1285.7	137 562.9	100 0.6	50 1.2	51 1.6	201 1.0
32	NS	85 503.0	45 395.9	11 921.0	141 515.9	100 0.2	50 0.7	51 0.8	201 0.5
	NG	87 532.3	36 876.2	10 1178.7	133 703.9	100 0.5	50 1.1	51 1.1	201 0.8
	2	87 460.7	43 457.8	10 1080.4	140 526.2	100 0.3	50 0.9	51 0.9	201 0.6
	4	94 235.6	42 563.2	12 927.5	148 407.2	100 0.3	50 0.8	51 0.8	201 0.6
	8	95 218.5	40 684.8	9 1126.7	144 454.6	100 0.3	50 0.8	51 1.0	201 0.6
	16	93 305.5	38 773.2	11 1051.9	142 524.6	100 0.4	50 0.8	51 1.0	201 0.7
64	NS	87 427.3	46 258.5	12 757.3	145 412.4	100 0.1	50 0.3	51 0.3	201 0.2
	NG	47 1508.0	33 1006.2	11 1133.6	91 1318.5	100 2.8	50 2.5	51 1.4	201 2.4
	2	87 426.2	45 341.7	12 885.0	144 450.2	100 0.2	50 0.4	51 0.4	201 0.3
	4	95 233.8	43 412.9	12 853.2	150 353.5	100 0.2	50 0.5	51 0.4	201 0.3
	8	96 177.7	42 527.9	13 816.2	151 350.4	100 0.2	50 0.4	51 0.4	201 0.3
	16	95 212.4	39 641.5	11 912.3	145 415.1	100 0.2	50 0.5	51 0.5	201 0.3
128	NS	87 400.5	50 125.6	17 542.1	154 333.8	100 0.1	50 0.2	51 0.2	201 0.1
	NG	30 1916.0	30 1096.0	11 956.0	71 1569.1	100 21.6	50 21.6	51 21.2	201 21.5
	2	87 395.4	47 259.0	16 635.4	150 380.5	100 0.1	50 0.3	51 0.2	201 0.2
	4	95 203.8	43 360.8	14 695.2	152 303.2	100 0.1	50 0.3	51 0.2	201 0.2
	8	96 184.5	41 461.9	15 645.0	152 316.4	100 0.2	50 0.3	51 0.3	201 0.2
	16	95 215.4	40 573.1	13 728.9	148 376.9	100 0.1	50 0.3	51 0.2	201 0.2
256	NS	88 369.0	50 86.1	18 272.1	156 274.4	100 0.1	50 0.2	51 0.2	201 0.1
	NG	26 2049.4	27 1162.4	11 992.2	64 1672.2	100 32.7	50 35.2	51 35.7	201 34.1
	2	88 402.2	47 210.5	17 418.0	152 346.8	100 0.1	50 0.2	51 0.2	201 0.2
	4	95 226.7	45 316.5	17 420.1	157 274.1	100 0.2	50 0.3	51 0.2	201 0.2
	8	95 199.0	41 410.9	15 420.7	151 285.8	100 0.1	50 0.2	51 0.2	201 0.2
	16	95 217.1	41 492.8	18 387.7	154 317.5	100 0.1	50 0.2	51 0.2	201 0.2

Table IV
OVERALL RESULTS FOR EACH k-SAT SUBSET OF INSTANCES. EACH CELL INDICATES THE NUMBER OF SOLVED INSTANCES (TOP) AND THE PAR-10 (BOTTOM)

solvers. Here, we display the number of solvers for each group (GS), the number of solved instances, and the speedup. Using 4 and 8 solvers per group scale reasonably well as we increase the number of cores in both number of solved instances and relative speedup factor. However, as previously observed for portfolios without cooperation a performance plateau seems to be reached after a few hundreds of cores (see Figure 5); little improvement is observed afterwards. Therefore, a main goal of a parallel solver should be maximizing the overall number of solved instances and at the same time minimizing the number of cores needed to reach the performance plateau.

Table IV summarizes all the experiments presented so far in this paper. This table shows detailed information for each category of instances (i.e. large and medium size) and each k-SAT subset, each cell indicates the number of solved instances (top) and the penalized average runtime (bottom), GS indicates the number of solvers for each group, No Sharing (NS) represents a portfolio without cooperation, and No Groups (NG) indicates that there are no groups of solvers and cooperation is performed as detailed in Section I-B.

As one might have expected, medium-size instances are easy and all parallel strategies are able to solve this set of instances without restarts, however, we would like to point out that here we observe a potential communication overhead induced by sharing the best known configuration with all solvers in the portfolio, for instance, a portfolio of 256 cores without cooperation requires on average less than 1 second to solve medium-size instances, while the same portfolio using full knowledge sharing (NG) requires on average 34.1 seconds to find a solution. On the other hand, for large-size instances using groups of 4 solvers usually outperform its counterpart portfolio without cooperation for 3-SAT, 5-SAT instances are dominated by portfolios without cooperation, and although a portfolio without cooperation solves more instances when using 128 and 256 cores, the overall PAR-10 is dominated by cooperation with groups of solvers.

Now, we switch our attention to Figure 6, where we study the performance improvement of parallel portfolios (using 4 and 8 solvers per group) when increasing the number of cores (i.e. 16, 64, and 256 cores). To this end, we launched an extra experiment of 200 runs with a timeout of 5 minutes for each portfolio on a single hard instance[4] (unif-k3-r4.2-v25000-c105000-S30704505-092-UNKNOWN.cnf).

This figure shows the empirical run-time distribution (RTD) for the selected instance and indicates the probability of finding a solution within a given amount of time. For instance, in Figure 6(a) it can be observed that the probability of solving the reference instance using a portfolio of 16 cores without cooperation (red line) and a time limit of 200 seconds is about 10% (P(*timeout*<200) $\approx$ 10%). Similarly, *Prob NormalizedW* using 4 solvers per group (green line) reports P(*timeout*<200) $\approx$ 50% and *Prob NormalizedW* using 8 solvers per group (blue line) reports P(*timeout*<200) $\approx$ 60%. It is important to point out that this instance was selected due to its hardness for the reference solver, for this reason, even though the instance is known to be satisfiable the probability of finding a solution using 16 cores is still lower than 100% when using the maximum amount of time (i.e. 300 seconds).

Additionally, this figure also helps to observe the importance of cooperation and the scalability of parallel portfolios, for instance, cooperative portfolios using 64 cores (Figure 6(b)) report P(*timeout*<300) $\approx$ 100% and the non-cooperative one reports P(*timeout*<300) $\approx$ 78%. Moreover, cooperative portfolios using 256 cores (Figure 6(c)) require less than 250 seconds to solve the reference instance (i.e. P(*timeout*<250) $\approx$ 100%) and the non-cooperative one also increases the chances of successfully solving the instance to P(*timeout*<300) $\approx$ 97%.

[4]This instance reported a timeout using our baseline solver (a sequential execution of Sparrow)

V. Conclusions and Future Work

This paper has presented an experimental analysis of parallel portfolios of local search algorithms for SAT with and without cooperation. Overall, the experiments suggest that cooperation is a powerful technique that helps to improve performance up to a given number of cores (usually 16 cores), and after this point the performance degrades. To overcome this limitation, we limit cooperation to groups of solvers where each group exploits cooperation but no information is shared between solvers of different groups. This mechanism scales reasonably well up to a few hundreds of cores

Our future work involves the use of additional information to be exchanged, for instance: tabu-list, age and score of a variable, most frequently unsatisfied clauses, information on local minima, etc. In addition, we plan to conduct an in-depth investigation of the application of machine learning techniques in order to identify promising and potentially bad runs in the parallel portfolio. To this end, we are exploring two kinds of features or descriptors: the well-known feature set used in the SATzilla framework [24] and local statistics (e.g. variable's score, number of plateaus reached so far, etc.) to check the evolution of the landscape of the search. Taking all this into account, we plan to study the performance of parallel local search for SAT on other architectures, such as supercomputers and GPUs.

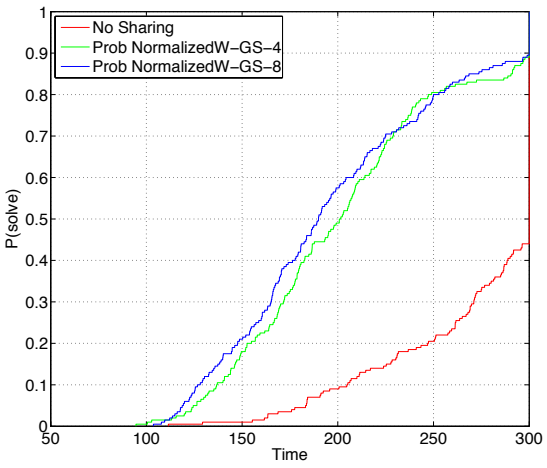
VI. Acknowledgements

The first author was supported by the Japan Society for the Promotion of Science (JSPS) under the JSPS Postdoctoral Program and the *kakenhi* Grant-in-aid for Scientific Research. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

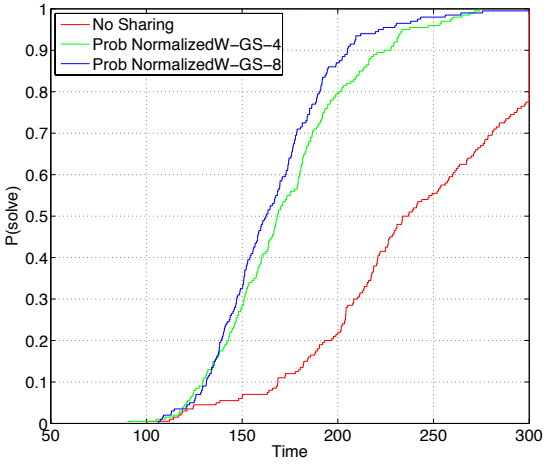
References

[1] M. Verhoeven and E. Aarts, "Parallel local search," *Journal of Heuristics*, vol. 1, no. 1, pp. 43–65, 1995.

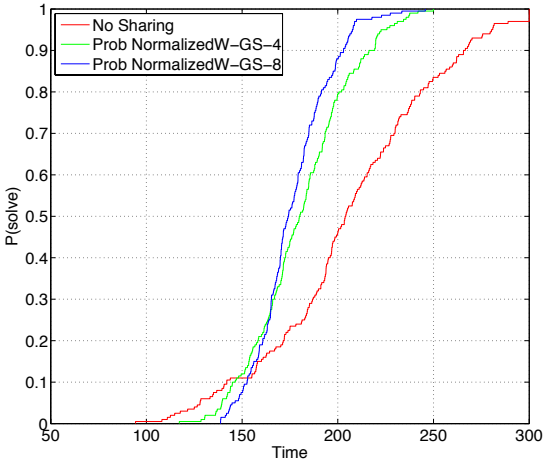
[2] P. M. Pardalos, L. S. Pitsoulis, T. D. Mavridou, and M. G. C. Resende, "Parallel Search for Combinatorial Optimization: Genetic Algorithms, Simulated Annealing, Tabu Search and GRASP," in *Parallel Algorithms for Irregularly Structured Problems (IRREGULAR)*, 1995, pp. 317–331.



(a) 16 cores.



(b) 64 cores.



(c) 256 cores.

Figure 6. RTD for portfolios with and without cooperation (using groups of 2 and 4 solvers) to solve the instance unif-k3-r4.2-v25000-c105000-S30704505-092-UNKNOWN.cnf

[3] L. Bordeaux, Y. Hamadi, and H. Samulowitz, "Experiments with Massively Parallel Constraint Solving," in *IJCAI'09*, C. Boutilier, Ed., Pasadena, California, USA, July 2009, pp. 443–448.

[4] Y. Hamadi, S. Jabbour, and L. Sais, "ManySAT: A Parallel SAT Solver," *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, vol. 6, no. 4, pp. 245–262, 2009.

[5] A. Biere, "Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT race 2010," FMV Reports Series, Tech. Rep. 10/1, August 2010.

[6] D. N. Pham and C. Gretton, "gNovelty+," in *Solver description, SAT competition 2007*, 2007.

[7] A. Roli, "Criticality and Parallelism in Structured SAT Instances," in *CP'02*, ser. LNCS, P. V. Hentenryck, Ed., vol. 2470. Ithaca, NY, USA: Springer, September 2002, pp. 714–719.

[8] L. Kroc, A. Sabharwal, C. P. Gomes, and B. Selman, "Integrating Systematic and Local Search Paradigms: A New Strategy for MaxSAT," in *IJCAI'09*, C. Boutilier, Ed., Pasadena, California, July 2009, pp. 544–551.

[9] K. Ohmura and K. Ueda, "c-sat: A Parallel SAT Solver for Clusters," in *SAT'09*, ser. LNCS, O. Kullmann, Ed. Swansea, UK: Springer, June 2009, pp. 524–537.

[10] A. E. J. Hyvärinen, T. A. Junttila, and I. Niemelä, "Grid-based sat solving with iterative partitioning and clause learning," in *CP'11*, ser. LNCS, J. H.-M. Lee, Ed., vol. 6876. Perugia, Italy: Springer, September 2011, pp. 385–399.

[11] G. Audemard, J.-M. Lagniez, B. Mazure, and L. Sais, "Learning in Local Search," in *21st International Conference on Tools with Artificial Intelligence (ICTAI'09)*. Newark, New Jersey, USA: IEEE Computer Society, November 2009, pp. 417–424.

[12] W. Wei and C. M. Li, "Switching Between Two Adaptive Noise Mechanisms in Local Search for Sat," in *Solver description, SAT competition 2009*, 2009.

[13] A. Balint, A. Fröhlich, D. A. D. Tompkins, and H. H. Hoos, "Sparrow11," in *Solver description, SAT competition 2011*, 2011.

[14] A. Balint and A. Fröhlich, "Improving Stochastic Local Search for SAT with a New Probability Distribution," in *SAT'10*, ser. LNCS, O. Strichman and S. Szeider, Eds., vol. 6175. Edinburgh, UK: Springer, July 2010, pp. 10–15.

[15] C. M. Li and W. Q. Huang, "Diversification and Determinism in Local Search for Satisfiability," in *SAT'05*, ser. LNCS, F. Bacchus and T. Walsh, Eds., vol. 3569. St. Andrews, UK: Springer, June 2005, pp. 158–172.

[16] F. Hutter, D. A. D. Tompkins, and H. H. Hoos, "Scaling and Probabilistic Smoothing: Efficient Dynamic Local Search for SAT," in *CP'02*, ser. LNCS, P. V. Hentenryck, Ed., vol. 2470. Ithaca, NY, USA: Springer, September 2002, pp. 233–248.

[17] J. Thornton, D. N. Pham, S. Bain, and V. Ferreira Jr, "Additive versus Multiplicative Clause Weighting for SAT," in *AAAI'04/IAAI'04*, D. L. McGuinness and G. Ferguson, Eds. San Jose, California, USA: AAAI Press / The MIT Press, July 2004, pp. 191–196.

[18] A. Arbelaez and Y. Hamadi, "Improving Parallel Local Search for SAT," in *Learning and Intelligent Optimization, 5th International Conference, LION'11*, ser. LNCS, C. A. C. Coello, Ed., vol. 6683. Springer, 2011, pp. 46–60.

[19] D. A. D. Tompkins and H. H. Hoos, "UBCSAT: An Implementation and Experimentation Environment for SLS algorithms for SAT and MAX-SAT," in *SAT'04*, ser. LNCS, H. H. Hoos and D. G. Mitchell, Eds., vol. 3542. Vancouver, BC, Canada: Springer, 2004, pp. 306–320.

[20] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Tradeoffs in the Empirical Evaluation of Competing Algorithm Designs," *Annals of Mathematics and Artificial Intelligence (AMAI), Special Issue on Learning and Intelligent Optimization*, vol. 60, no. 1-2, pp. 65–89, 2010.

[21] A. Arbelaez and Y. Hamadi, "Cooperative Stochastic Local Search," in *Solver description, SAT competition 2011*, 2011.

[22] A. Arbelaez, "Learning During Search," Ph.D. dissertation, Universté Paris-Sub, Orsay, France, 2011.

[23] A. Arbaleaz and P. Codognet, "Towards Massivelly Parallel Local Search for SAT (Poster Paper)," in *SAT'12*, ser. LNCS, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Trento, Italy: Springer, June 2012, pp. 481–482.

[24] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: Portfolio-based algorithm selection for sat," *Journal of Artificial Intelligence Research*, vol. 32, pp. 565–606, 2008.