From Sequential to Parallel Local Search for SAT

Alejandro Arbelaez, Philippe Codognet

► To cite this version:

Alejandro Arbelaez, Philippe Codognet. From Sequential to Parallel Local Search for SAT. Evolutionary Computation in Combinatorial Optimization, Apr 2013, Vienna, Australia. pp.157-168, 10.1007/978-3-642-37198-1_14 . hal-04419667

HAL Id: hal-04419667
https://inria.hal.science/hal-04419667

Submitted on 26 Jan 2024

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.

Copyright

From Sequential to Parallel Local Search for SAT

Alejandro Arbelaez*[1] and Philippe Codognet[2]

[1] JFLI / University of Tokyo
[2] JFLI - CNRS / UPMC / University of Tokyo
University of Tokyo, Dept. of Computer Science,
7-3-1, Hongo, Bunkyo-ku, 113-0033 Tokyo, Japan
{arbelaez,codognet}@is.s.u-tokyo.ac.jp

Abstract. In the domain of propositional Satisfiability Problem (SAT), parallel portfolio-based algorithms have become a standard methodology for both complete and incomplete solvers. In this methodology several algorithms explore the search space in parallel, either independently or cooperatively with some communication between the solvers. We conducted a study of the scalability of several SAT solvers in different application domains (crafted, verification, quasigroups and random instances) when drastically increasing the number of cores in the portfolio, up to 512 cores. Our experiments show that on different problem families the behaviors of different solvers vary greatly. We present an empirical study that suggests that the best sequential solver is not necessary the one with the overall best parallel speedup.

1 Introduction

The propositional Satisfiability Problem (SAT) is the first known NP-complete problem [1] and consists in determining whether a Boolean formula $\mathcal{F}$ is satisfiable or not. $\mathcal{F}$ is represented by a pair $\langle \mathcal{X}, \mathcal{C} \rangle$, where $\mathcal{X}$ is a set of Boolean variables and $\mathcal{C}$ is a set of clauses in Conjunctive Normal Form (CNF). Each clause is a disjunction of literals (a variable x or its negation $-x$). Additionally, an assignment is a mapping from the variables in the problem to truth values, i.e. 0 (false) or 1 (true).

SAT solvers are able to tackle instances from a wide variety of domains ranging from software verification to computational biology and automated planning. As these are very hard and computationally intensive problems, the use of parallelism to speed up SAT solvers has attracted the attention of a growing number of researchers in the last decade. Currently, there are two well-established techniques to develop parallel SAT solvers: divide-and-conquer (e.g. [2]) and parallel portfolios (e.g. [3] and [4]). The former, divides the search space into several sub-spaces being explored in parallel. The second one consists in parallel executions of different algorithms, either independently or cooperatively with some communication between the parallel solvers.

* Corresponding Author

The performance of parallel SAT solvers is classically measured by means of the overall number of solved instances, as it is for sequential solvers. That is, increasing the number of cores might also increase the total number of solved instances within the same wall-clock time. This is what is usually called *capacity solving*. For instance, in the annual SAT competition solvers are ranked based on the total number of solved instances within a given time limit (breaking ties using the wall-clock time). However, another interesting feature is the *scalability* of the parallel solvers, that is, the *speedup* obtained when using several cores with respect to sequential execution. But the notion of *speedup* obtained for parallel solvers has received up to now limited attention in the classical SAT literature, which was mostly focused on *capacity solving*.

The question we would like to investigate in this paper is: *Is the best sequential solver also the best one in a massively parallel context?* Indeed the best sequential solver might not scale up so well and its performance on n cores (for some large n) might not be better than that of another solver whose sequential performance is maybe less good but with a better parallel scalability.

In order to study this issue, we conducted a study of the scalability (w.r.t. the speedup of the algorithms) of SAT solvers in different application domains when drastically increasing the number of cores in the portfolio. We focused our attention in this paper on local search algorithms for SAT. In SAT local search there exist a large number of heuristics [5] with different performances for different problem families. For this reason, unlike sequential settings, where it is necessary to select only one algorithm to solve a given problem instance, the parallel portfolio needs to identify n solvers. Moreover, in addition of selecting solvers, the parallel portfolio might also consider their scalability when increasing the number of cores.

The rest of the paper is organized as follows. Section 2 presents background material including a description of local search algorithms for SAT and related work in the area. Section 3 describes all the benchmark families used in this paper, Section 4 presents extensive experimental results on parallel portfolios, and Section 5 presents general conclusions and future work.

2 Background

2.1 Local Search

A local search algorithm to tackle SAT instances starts with an initial random assignment for the variables (i.e. random values for the variables), then iteratively flips the truth value of one of the variables. The flipped variable usually minimizes the number of unsatisfied clauses, however, from time to time random selections are performed in order to avoid search stagnation.

In the following, we describe five well-known variable selection algorithms in local search for SAT. These algorithms have shown great performances in the annual SAT competitions. It is important to notice, that most of SAT local search algorithms have been inspired by the WalkSAT architecture [6], which selects an

unsatisfied clause and from that clause identifies the most appropriated variable to flip using some heuristic function. In the following, we describe a set of state-of-the-art local search algorithms to solve SAT instances.

Novelty [7] employs a function $score(x) = make + break$ to select a variable at each iteration of the local search procedure. Intuitively, *make* indicates the number of clauses that are satisfied under the current assignment but become unsatisfiable when flipping x, and *break* represents the number of clauses that are unsatisfiable under the current assignment and will be satisfied when flipping x. Then, *Novelty* selects, uniformly at random, an unsatisfied clause c, and from c identifies v_{best} and v_{2best}, the best and the second best variables in c according to the *score* function. v_{best} is flipped if this variable is not the most recently flipped one in c, otherwise v_{2best} is flipped with a given probability p and v_{best} with a probability *1-p*.

Pure Additive Weighting Scheme (PAWS) [8] adds a weight clause penalty to each clause and selects the variable that provides the highest reduction in the sum of all unsatisfied clause penalties. All weights are initialized to 1 and updated during the search process, i.e. increased 1 unit when search stagnation is observed and decreased after a given number of weight increases.

Variable Weighting (VW) [9] maintains a counter for each variable, indicating the number of flips of the variable. Then, VW (known as VW1 in [9]) selects an unsatisfied clause c and if no variable in c reports *break*=0, VW selects with a probability p a random variable in c. Otherwise, the variable with the smallest *break* value is selected, breaking ties by minimizing the number of flips of the variables.

Adaptive G2WSAT (AG2) [10]: introduces the concept of decreasing variables. Broadly speaking, a variable is decreasing if flipping it reduces the over all number of failed clauses. Taking this into account, the algorithm maintains a lists of promising decreasing variables L and selects the variable with minimal score in L. If L is empty, AG2 selects, with a probability *dp* the most recently flipped variable from a violated clause. Otherwise, with a probability *1-dp Novelty* is used as a backup heuristic.

Sparrow [11] exploits features of the previously mentioned local search algorithms. First, similar to PAWS a weight penalty is added to all clauses. Second, similar to AG2 a list of promising variables L is maintained during the search. Whenever L is empty the penalty for unsatisfied clauses is increased 1 unit with a probability *ps* and decreased with probability *1-ps.* Sparrow selects the best variable from L, and if L is empty the algorithm selects, uniformly at random, an unsatisfied clause c and from c selects a variable using a probabilistic function which considers two criteria: the sum of all unsatisfied clauses and the last time the variable was last flipped.

2.2 Parallel SAT

A straightforward approach to parallelize local search algorithms consists in the parallel portfolio-based approach (so-called multi-start or multiple-walk). In this approach, several algorithms (or different copies of the same one with different

random seeds) are executed in parallel until a solution is found or a given timeout is reached.

The parallel portfolio has two important properties. First, no load balancing is required to parallelize the sequential algorithm. Second, in theory, it is possible to reach linear and super-linear speedups [12]. Indeed, in Section 4 we will observe that in practice some scenarios report super-linear speedups.

The portfolio approach without cooperation has been previously used in the gNovelty+ solver [13]. This portfolio executes independent copies of the gNovelty+ heuristic. Other parallel local search solvers for SAT comprehend PGSAT [14] and MiniWalk [15]. PGSAT divides the entire set of variables into independent subsets which are then allocated to different processors, then iteratively performs multiples flips in parallel (one for each subset). 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.

Other work in the area includes [4], where the authors use cooperation to improve the performance of the parallel portfolio. In this work, each algorithm exchanges the best assignment for the variables found so far in order 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 suggests a value for each variable to probabilistic constructions. However, as pointed out in [16] the performance of the cooperative portfolio considerably degrades as the number of cores increases.

Regarding complete parallel SAT solvers, several multicore algorithms (see [17] for a recent survey) have been proposed. Most of these algorithms are also based on the parallel portfolio architecture. However, in this case, the portfolio executes different and complementary backtracking search algorithms based on the DPLL method. Moreover, algorithms exchange learned clauses in order to improve performance.

3 Experimental settings

This section describes the set of benchmark families used to test the performance of the algorithms. That is, crafted, quasigroups, verification, and random instances.

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.

In addition, we used openMPI to build our parallel solver on top of UBCSAT [18]. When running on n cores, each parallel portfolio executes n independent copies of a given algorithm. Moreover, all algorithms were executed with their default parameters, except for Sparrow where we use the parameter configuration reported for the international SAT'11 competition.

3.1 Crafted Instances

This problem family corresponds to a selection of instances designed to be difficult for SAT solvers. In this paper, we used a set of 149 known SAT instances from the 2011 SAT competition (crafted category) and filtered out too easy and hard instances by running a portfolio of 16 copies of Sparrow with a timeout of 5 minutes. We use all instances whose median runtime across 10 runs was greater than 100 seconds and lower than 300 seconds. The final set consists in the following 9 instances (denoted crafted-[1 to 9] in this paper): srhd-sgi-m37-q505.75-n35-p15-s48276711 (crafted-1); srhd-sgi-m42-q585-n40-p15-s54275047 (crafted-2); srhd-sgi-m42-q663-n40-p15-s72490337 (crafted-3); srhd-sgi-m47-q742.5-n45-p15-s28972035 (crafted-4); em_8_4_5_fbc (crafted-5); rbsat-v1150c84314g7 (crafted-6); rbsat-v1375c111739g4 (crafted-7); sgen3-n240-s78945233-sat (crafted-8); sgen3-n260-s62321009-sat (crafted-9).

3.2 Quasigroup Instances

The Quasigroup with holes problem (qwh) consists in completing a pre-filled $N \times N$ matrix with the numbers $[1, 2, ..., N]$ such that for each column (resp. row) of the matrix, each element occurs exactly once. Instances were generated using the *lsencode* instance generator [19]. It is also worth to notice that these instances have been widely used to test the performance of SAT and CSP solvers. The final set consists in the following 8 instances (denoted qwh-[1 to 8] in this paper): qwh.order.35.holes.405 (qwh-1); qwh.order.40.holes.528 (qwh-2); qwh.order.40.holes.544 (qwh-3); qwh.order.40.holes 560 (qwh-4); qwh.order.60. holes.1440 (qwh-5); qwh.order.60.holes.1620 (qwh-6); qwh.order.70.holes.2450 (qwh-7); qwh.order.70.holes.2940 (qwh-8).

3.3 Verification Instances

This problem family corresponds to a collection of SAT encoded CBMC (Bounded Model Checking) instances generated using [20]. In this paper, we used a set of 40 instances also used to test SAT solvers, such as [21]. We filtered out too easy and hard instances by running a portfolio of 16 copies of Sparrow with a timeout of 5 min. and selected all the instances whose median runtime across 10 runs was greater than 100 sec. and lower than 5 min. The final set of instances is the following (as named in [21] and denoted cbmc-[1 to 9] in this paper): 23 (cbmc-1); 25 (cbmc-2); 26 (cbmc-3); 28 (cbmc-4); 31 (cbmc-5); 32 (cbmc-6); 33 (cbmc-7); 35 (cbmc-8); 36 (cbmc-9).

3.4 Random Instances

Random instances (also known as Uniform Random k-SAT) are frequently used to test the performance of SAT solvers. These instances are automatically generated using three parameters: number of clauses (m), number of variables (n),

and the number of literals per clause (k). Clauses for a given instance are generated by iteratively selecting, uniformly at random, a variable id i and then with a probability 0.5 x_i is included into the clause, otherwise -x_i is added to the clause (literals of different polarity are not accepted in the same clause). It is worth pointing out that random k-SAT instances around the phase transition (i.e. m/n=4.2) are known to be difficult [22].

In this paper, we consider a collection of 369 known satisfiable instances from the international SAT'11 competition. From this set we filtered out too easy and too hard instances by running a portfolio of 16 copies of Sparrow and removed instances whose median runtime were greater than 100 sec. and lower than 300 sec.. The final set consists in the following 8 instances (denoted rand-[1 to 8] in this paper), where Seed indicates the unique seed number used to generate the instance, v represents the number of variables, and r represents the ratio variables/clauses: Seed: 1854039067 - v: 30000 - c: 126000 - r: 4.2 (rand-1); Seed: 970100151 - v: 35000 - c: 147000 - r: 4.2 (rand-2); Seed: 1184456903 - v: 40000 - c: 168000 - r: 4.2 (rand-3); Seed: 1170024351- v: 50000 - c: 210000 - r: 4.2 (rand-4); Seed: 537193780 - v: 50000 - c: 210000 - r: 4.2 (rand-5); Seed: 957916968 - v: 50000 - c: 210000 - r: 4.2 (rand-6); Seed: 969405384 - v: 1500 - c: 30000 - r: 20 (rand-7); Seed: 922811046 - v: 2000 - c: 30000 - r: 20 (rand-8).

4 Experiments

In this section, we present experiments of parallel portfolios when drastically increasing the number of cores. For the sake of clarity, we use the following notation: [Solver Name]-N, where N represents the number of cores, for example: Sparrow-128, AG2-512, and VW-32 represent respectively a portfolio of Sparrow on 128 cores, AG2 on 512 cores, and VW on 32 cores. Each core executing one copy of the indicated algorithm.

In addition, the speedup is reported against a portfolio of 16 cores and computed as follows: $\mathit{Speedup}=\frac{\mathit{median\text{-}time}([\mathit{Solver}]-N)}{\mathit{median\text{-}time}([\mathit{Solver}]-16)}$, where *median-time* reports the median time across 50 independent executions of a given portfolio strategy.

Moreover, we also study the runtime distribution (RTD) for each benchmark family (see chapter 4 in [5]). The RTD is a probability function $P(\mathit{timeout} < t)$ which assigns probabilities to a given random variable, i.e. the runtime needed until completion, and can be seen as the probability of solving a given instance within a given time limit t.

4.1 Crafted Instances

Let us start our analysis with Figure 1(a), where we observe the overall performance improvement when increasing the number of cores from 16 to 512. This figure shows the RTD for Sparrow-16, AG2-16, Sparrow-512, and AG2-512. As one might have expected, increasing the number cores also increases the chances of solving a given instance for this problem family. For instance, the probability of solving an instance with a time limit of 10 seconds for Sparrow increases from

Instance	Alg	Number of Cores					
		16	32	64	128	256	512
crafted-1	SP	145.3 100%	90.7 100%	68.2 100%	23.8 100%	10.7 100%	5.9 100%
	AG2	**27.0** **100%**	**13.8** **100%**	**4.5** **100%**	**2.2** **100%**	**0.9** **100%**	**0.8** **100%**
crafted-2	SP	122.6 100%	73.6 100%	47.5 100%	16.2 100%	9.6 100%	5.4 100%
	AG2	**48.7** **100%**	**15.0** **100%**	**7.3** **100%**	**4.2** **100%**	**1.6** **100%**	**1.1** **100%**
crafted-3	SP	146.6 100%	95.6 100%	60.9 100%	29.9 100%	11.3 100%	6.7 100%
	AG2	**51.9** **100%**	**23.3** **100%**	**6.0** **100%**	**2.9** **100%**	**1.9** **100%**	**1.2** **100%**
crafted-4	SP	150.3 100%	63.8 100%	47.7 100%	22.7 100%	10.9 100%	4.6 100%
	AG2	**30.7** **100%**	**17.9** **100%**	**5.9** **100%**	**3.5** **100%**	**2.3** **100%**	**1.9** **100%**
crafted-5	SP	**129.0** **100%**	**54.7** **100%**	**27.2** **100%**	**14.0** **100%**	**8.8** **100%**	**5.8** **100%**
	AG2	742.8 96%	428.6 100%	224.5 100%	105.4 100%	50.3 100%	38.5 100%

Instance	Alg	Number of Cores					
		16	32	64	128	256	512
crafted-6	SP	278.2 100%	141.3 100%	41.7 100%	30.0 100%	14.8 100%	11.5 100%
	AG2	**123.6** **100%**	**71.3** **100%**	30.9 **100%**	**18.2** 100%	**11.1** **100%**	**6.5** **100%**
crafted-7	SP	167.3 100%	89.4 100%	63.7 100%	32.1 100%	16.4 100%	7.3 100%
	AG2	**70.8** **100%**	**41.2** **100%**	**18.6** **100%**	**11.2** **100%**	**4.8** **100%**	**3.0** **100%**
crafted-8	SP	111.3 100%	52.4 100%	28.8 100%	23.9 100%	12.2 100%	8.8 100%
	AG2	**46.5** **100%**	**32.2** **100%**	**13.2** **100%**	**7.6** **100%**	**6.3** **100%**	**3.6** **100%**
craft-9	SP	125.7 100%	75.7 100%	44.6 100%	23.2 100%	13.5 100%	10.6 100%
	AG2	79.89 100%	42.64 100%	22.79 100%	13.45 100%	9.69 100%	7.32 100%

Table 1. Performance summary for crafted instances. Each cell indicates the median runtime (top) and the percentage of solved instances (bottom) for each instance.

about $P(\textit{timeout}{<}10) \approx 0.03$ using 16 cores to $P(\textit{timeout}{<}10) \approx 0.69$ using 512 cores.

Additionally, Figure 2(a) shows the speedup (relative to a portfolio using 16 cores) using 512 cores for each algorithm. Algorithms above the dashed line indicate that a super-linear speedup is reached and below the line indicate a sub-linear speedup. In this figure, we observe linear speedups for the following instances: crafted-1 (AG2-512), crafted-2 (AG2-512), crafted-3 (AG2-512), and crafted-4 (Sparrow-512). Moreover, except for crafted-8, crafted-9, and crafted-4 (AG-512), we observe an interesting speedup for all algorithms, this shows the scalability of the parallel portfolio approach when considering an important number of cores. It is also worth noticing that the speedup observed for both algorithms (up to 512 cores) is similar for nearly all instances.

Table 1 summarizes the results for each portfolio using 16, 32, 64, 128, 256, and 512 cores[3]. Hereafter, bold figures indicate statistically significant differences (Mann-Whitney U test with 95% confidence level). It can be observed that AG2-16 is considerably better than Sparrow-16, however, the difference in the performances becomes smaller as the number of cores increases. In addition, to solve crafted-5, the effectiveness (i.e. percentage of solved instances) of AG2 increases as the number of cores increases, i.e. from 96% (AG2-16) to 100% (AG2-32).

4.2 Quasigroup Instances

Figure 1(c) shows the RTD for Sparrow and AG2 for QWH problems. Here, we observe that overall Sparrow is better than AG2 for QWH instances using 16 cores. Notice that the probability of solving a given instance within 10 seconds for Sparrow-512 and AG2-512 is $P(\textit{timeout} < 10) > 0.9$.

Unlike crafted instances where both algorithms exhibit a good speedup for nearly all instances up 512 cores, here only AG2 exhibits an interesting speedup

[3] In the following tables 'SP' stands for Sparrow.

Instance	Alg	Number of Cores					
		16	32	64	128	256	512
qwh-1	SP	**1.90** **100%**	**1.57** **100%**	**1.26** **100%**	**0.95** **100%**	**0.70** **100%**	**0.81** **100%**
	AG2	104.07 100%	63.06 100%	39.54 100%	18.82 100%	9.40 100%	4.94 100%
qwh-2	SP	**1.33** **100%**	**0.80** **100%**	**0.67** **100%**	**0.57** **100%**	**0.55** **100%**	**0.51** **100%**
	AG2	101.81 100%	33.02 100%	22.40 100%	9.68 100%	6.16 100%	4.11 100%
qwh-3	SP	**1.02** **100%**	**0.69** **100%**	**0.57** **100%**	**0.53** **100%**	**0.44** **100%**	**0.47** **100%**
	AG2	29.16 100%	17.87 100%	10.11 100%	8.86 100%	4.46 100%	2.70 100%
qwh-4	SP	**0.78** **100%**	**0.57** **100%**	**0.37** **100%**	**0.34** **100%**	**0.34** **100%**	**0.36** **100%**
	AG2	5.02 100%	3.59 100%	1.83 100%	0.93 100%	0.54 100%	0.57 100%

Instance	Alg	Number of Cores					
		16	32	64	128	256	512
qwh-5	SP	20.98 100%	10.61 100%	4.77 100%	2.89 100%	2.01 100%	1.99 100%
	AG2	**0.72** 100%	**0.71** 100%	**0.71** 100%	**0.72** 100%	**0.75** 100%	**0.81** 100%
qwh-6	SP	6.15 100%	3.91 100%	2.81 100%	1.78 100%	1.61 100%	1.60 100%
	AG2	**1.11** **100%**	**1.12** **100%**	**1.13** **100%**	**1.14** **100%**	**1.16** **100%**	**1.22** **100%**
qwh-7	SP	20.15 100%	18.06 100%	11.18 100%	7.47 100%	5.70 100%	4.99 100%
	AG2	**4.05** **100%**	**4.04** **100%**	**4.09** **100%**	**4.09** **100%**	**4.16** **100%**	**4.29** **100%**
qwh-8	SP	17.88 100%	12.87 100%	12.32 100%	12.10 100%	12.13 100%	12.37 100%
	AG2	**11.25** **100%**	**11.15** **100%**	**11.29** **100%**	**11.23** **100%**	**11.49** **100%**	**11.57** **100%**

Table 2. Performance summary for QWH instances. Each cell indicates the median runtime (top) and the percentage of solved instances (bottom) for each instance.

for four instances, i.e. qwh-[1-4]. Additionally, we would like to point out that in Table 2, it can be observed that AG2 (qwh-1) exhibits a super-linear speedup up to 256 cores. More importantly, due to the great scalability of the portfolio-base approach the difference in the performance between the algorithms decreases as the number of cores increases. For instance, to solve qwh-1, Sparrow-16 is 54 times faster than AG2-16, while Sparrow-512 is only 6 times faster than AG2-512 to solve the same instance.

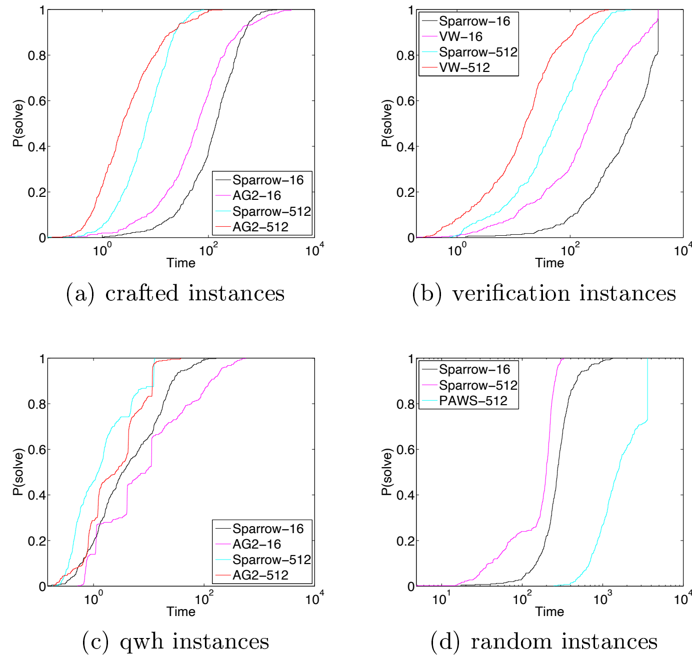
4.3 Verification Instances

For this set of instances we limit our attention to our reference solver Sparrow and VW. VW has been reported in the literature as a very efficient algorithm for this set of problems (see [21]) Figure 1(b) shows an important difference between VW-16 and Sparrow-16. The difference lies primarily in the probability of solving a given instance within the time limit, i.e. VW-16 reports $P(timeout < 3600) \approx 0.96$ while Sparrow-16 reports $P(timeout < 3600) \approx 0.82$. Both algorithms exhibit an improvement when the number of cores increases (from 16 to 512), $P(timeout < 3600) \approx 1$ in both cases.

Figure 2(b) shows that Sparrow achieves a near optimal speedup for 5 out of 9 instances (cbmc-[2,3,5,7,8]), and VW achieves a near optimal speedup for cbmc-7. Moreover, Table 3 shows the benefit of increasing the number of cores. For instance, the effectiveness of Sparrow (crafted-8) gradually increases as the number cores increases from 44% to 100%, i.e. 44% (Sparrow-16); 77% (Sparrow-32); 92% (Sparrow-64); and 100% (Sparrow-128). Whereas, the effectiveness of VW (crafted-7) increases from 90% to 100%, i.e. 90% (VW-16); 98% (VW-32); and 100% (VW-64).

4.4 Random Instances

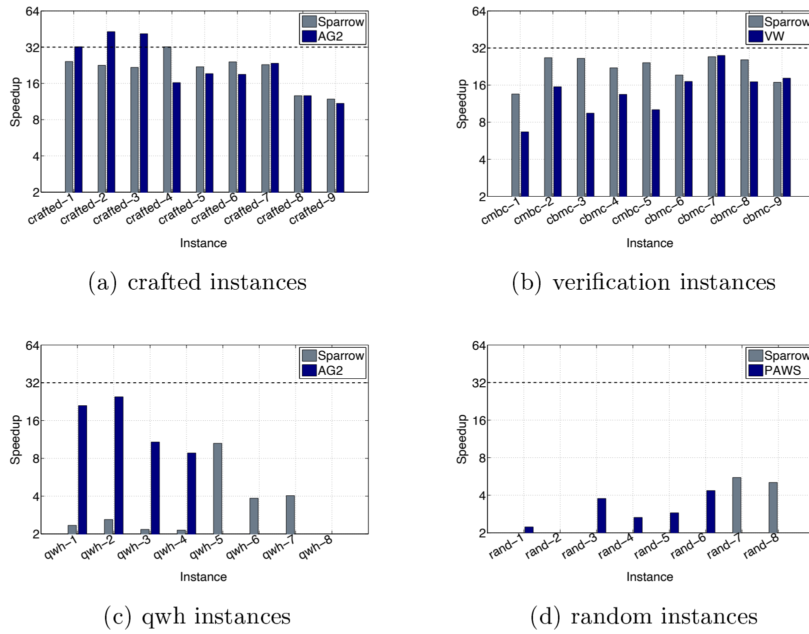
Because these instances are known to be hard for SAT solvers, we limit our attention to two solvers; Sparrow and PAWS using 16, 128, 256, and 512 cores. Figure 1(d) shows the RTD for Sparrow-16, Sparrow-512, and PAWS-512. Notice



(a) crafted instances (b) verification instances

(c) qwh instances (d) random instances

Fig. 1. RTD for portfolios using 16 and 512 cores



(a) crafted instances (b) verification instances

(c) qwh instances (d) random instances

Fig. 2. Speedup (w.r.t. 16 cores) using 512 cores

Instance	Alg	Number of Cores					
		16	32	64	128	256	512
cbmc-1	SP	141.2 100%	104.7 100%	69.3 100%	53.3 100%	22.0 100%	10.4 100%
	VW	**5.9** **100%**	**3.4** **100%**	**2.4** **100%**	**0.9** **100%**	**0.8** **100%**	**0.8** **100%**
cbmc-2	SP	827.8 98%	409.1 100%	223.0 100%	103.5 100%	43.2 100%	31.0 100%
	VW	**153.5** **100%**	**67.8** **100%**	**35.5** **100%**	**16.2** **100%**	**12.6** **100%**	**9.9** **100%**
cbmc-3	SP	1052.6 94%	500.3 98%	216.2 100%	135.8 100%	83.8 100%	39.9 100%
	VW	**112.8** **100%**	**95.1** **100%**	**34.5** **100%**	**20.9** **100%**	**16.1** **100%**	**11.9** **100%**
cbmc-4	SP	910.4 88%	395.0 98%	192.3 100%	117.3 100%	78.8 100%	41.2 100%
	VW	**198.1** **100%**	**106.75** **100%**	**53.78** **100%**	**27.5** **100%**	**14.3** **100%**	**14.7** **100%**
cbmc-5	SP	1782.0 90%	820.8 92%	406.7 100%	238.7 100%	136.5 100%	73.3 100%
	VW	**161.2** **100%**	**84.8** **100%**	**50.1** **100%**	**35.0** **100%**	**18.5** **100%**	**15.9** **100%**

Instance	Alg	Number of Cores					
		16	32	64	128	256	512
cbmc-6	SP	1772.0 78%	842.9 98%	505.1 100%	168.1 100%	116.2 100%	91.8 100%
	VW	**200.7** **100%**	**160.8** **100%**	**58.7** **100%**	**33.5** **100%**	**20.8** **100%**	**11.7** **100%**
cbmc-7	SP	2510.1 64%	1703.1 80%	417.3 98%	288.3 100%	182.0 100%	92.3 100%
	VW	**1526.3** **90%**	462.2 **98%**	**289.5** 100%	**151.5** **100%**	**89.6** **100%**	**54.8** **100%**
cbmc-8	SP	3600.0 48%	1808.0 72%	807.4 92%	458.0 100%	230.1 100%	140.2 100%
	VW	**1229.3** **86%**	**631.6** **100%**	**398.2** **100%**	**143.7** **100%**	**84.9** **100%**	**72.2** **100%**
cbmc-9	SP	2140.4 74%	900.5 90%	533.1 100%	325.1 100%	190.3 100%	127.0 100%
	VW	**1209.2** **88%**	**547.9** **98%**	**313.4** **100%**	**249.3** **100%**	**129.4** **100%**	**66.4** **100%**

Table 3. Performance summary for cbmc instances. Each cell indicates the median runtime (top) and the percentage of solved instances (bottom) for each instance.

that PAWS-16 is not included because it solves a limited number of instances (see Table 4). In this figure, we observe a small improvement when increasing the number of cores. In particular, the runtime that Sparrow requires to solve a given instance within the time limit decreases from $P(timeout < 1250) \approx 1$ for Sparrow-16 to $P(timeout < 290) \approx 1$ for Sparrow-512.

On the other hand, the speedup observed for these instances is considerably different than the previous benchmarks. In fact, the speedup reported for PAWS in the figure is an approximation, because PAWS-16 timed-out for an important number of instances. For this reason, this figure (for PAWS) should be taken as a lower bound of the actual speedup.

Another interesting behavior observed in these experiments is that the speedup varies from instance to instance (see Table 4). For example, the speedup for instances near the phase transition (i.e. rand-[1-6]) is substantially lower than the speedup for the remaining instances (i.e. rand-[7-8]). We plan to conduct a more detailed investigation to fully characterize the performance of random instances near the phase transition region when using massively parallel systems.

Finally, it is worth noticing that although the speedup is limited for these instances, Table 4 shows an important improvement in the effectiveness of PAWS for nearly all instances. For instance, the effectiveness to solve rand-1 increases as follows: 16% (PAWS-16), 72% (PAWS-128), 88% (PAWS-256), and 94% (PAWS-512). However, the the effectiveness of PAWS-512 degrades 2% with regard to PAWS-256, this corresponds to 1 timeout out of 50 executions of PAWS-512 to solve rand-4.

5 Conclusions and Future Work

This paper has presented extensive experimental results using parallel portfolios of local search algorithms for SAT. Overall the experiments suggest that the portfolio approach scales reasonably well up to an important number of cores (i.e. 512 cores) without the need of any particular tuning of the algorithm.

Instance	Alg	Number of Cores			
		16	128	256	512
rand-1	SP	**260.00 100%**	**188.00 100%**	**180.08 100%**	**192.26 100%**
	PAWS	3600.00 16%	2320.55 72%	1951.96 88%	1616.24 94%
rand-2	SP	**299.28 100%**	**240.75 100%**	**227.34 100%**	**232.95 100%**
	PAWS	3600.00 12%	3216.08 68%	2933.89 78%	1855.01 94%
rand-3	SP	**277.68 100%**	**190.56 100%**	**192.59 100%**	**200.32 100%**
	PAWS	3600.00 32%	1501.92 98%	1108.49 100%	956.37 100%
rand-4	SP	**274.04 100%**	**221.14 100%**	**204.24 100%**	**214.89 100%**
	PAWS	3394.61 50%	1602.76 96%	1289.28 100%	1279.33 98%

Instance	Alg	Number of Cores			
		16	128	256	512
rand-5	SP	**305.66 100%**	**252.28 100%**	**242.26 100%**	**242.82 100%**
	PAWS	3600.00 30%	1564.17 92%	1287.87 98%	1246.72 100%
rand-6	SP	**248.93 100%**	**200.02 100%**	**198.36 100%**	**213.42 100%**
	PAWS	3600.00 36%	1210.02 100%	1024.20 100%	826.30 100%
rand-7	SP	**179.21 100%**	**54.95 100%**	**39.74 100%**	**32.32 100%**
	PAWS	3600.00 0%	3600.00 0%	3600.00 0%	3600.00 0%
rand-8	SP	**325.62 100%**	**90.00 100%**	**73.11 100%**	**64.39 100%**
	PAWS	3600.00 0%	3600.00 0%	3600.00 0%	3600.00 0%

Table 4. Performance summary for random instances. Each cell indicates the median runtime (top) and the percentage of solved instances (bottom) for each instance.

In two out of four benchmark families (crafted and verification) the algorithms exhibit near optimal speedups, and super-linear in some particular cases. However for the quasigroup instances, we have observed that the best sequential algorithm reports surprisingly for half of the instances a very limited speedup factor (roughly a factor 2 w.r.t. 16 cores, even with 512 cores), while the other (slower) algorithm scales well up to an important number of cores, without however reaching the same raw performance. For the random instances, which are very hard problems close to the phase transition, it is worth noticing that parallel speedups are quite limited for all the algorithms studied. However, we would also like to point out that, as expected, the parallel portfolio approach helps to increase the effectiveness of the algorithms when increasing the number of cores. For instance, for the rand-5 problem, the effectiveness of the PAWS algorithm increases from 30% (PAWS-16) to 100% (PAWS-512).

Therefore our experiments show that on different problem families the behaviors of different solvers vary greatly. Although our initial question is still open, i.e. *Is the best sequential solver also the best one in a massively parallel context?*, we have presented an empirical study which suggests that the best sequential solver is not necessarily the one with the over all best speedup.

Our future work involves the study of cooperative algorithms that scale up significantly for a large number of cores (e.g. 512 cores). Indeed, current cooperative methods for parallel local search for SAT scale only up to 16 or 32 cores (see [16]). In addition, we plan to investigation the use of machine learning to identify potentially bad and good runs in the parallel portfolio.

6 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. Cook, S.A.: The Complexity of Theorem-Proving Procedures. In: Third annual ACM symposium on Theory of computing. STOC '71, ACM (1971) 151–158
2. Chrabakh, W., Wolski, R.: GridSAT: A System for Solving Satisfiability Problems Using a Computational Grid. Parallel Computing **32**(9) (2006) 660–687
3. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: A Parallel SAT Solver. Journal on Satisfiability, Boolean Modeling and Computation, JSAT **6**(4) (2009) 245–262
4. Arbelaez, A., Hamadi, Y.: Improving Parallel Local Search for SAT. In: LION'11. Volume 6683 of LNCS., Springer (January 2011) 46–60
5. Hoos, H., Stützle, T.: Stochastic Local Search: Foundations & Applications. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2004)
6. Selman, B., Kautz, H.A., Cohen, B.: Noise Strategies for Improving Local Search. In: AAAI'94. Volume 1. (July 1994) 337–343
7. McAllester, D.A., Selman, B., Kautz, H.A.: Evidence for Invariants in Local Search. In: AAAI'97. (1997) 321–326
8. Thornton, J., Pham, D.N., Bain, S., Ferreira Jr, V.: Additive versus Multiplicative Clause Weighting for SAT. In: AAAI'04. (July 2004) 191–196
9. Prestwich, S.D.: Random Walk with Continuously Smoothed Variable Weights. In: SAT'05. Volume 3569 of LNCS., St. Andrews, UK, Springer (June 2005) 203–215
10. Li, C.M., Huang, W.Q.: Diversification and Determinism in Local Search for Satisfiability. In: SAT'05. Volume 3569 of LNCS., Springer (June 2005) 158–172
11. Balint, A., Fröhlich, A.: Improving Stochastic Local Search for SAT with a New Probability Distribution. In: SAT'10. Volume 6175 of LNCS., Springer (July 2010) 10–15
12. Shylo, O.V., Middelkoop, T., Pardalos, P.M.: Restart Strategies in Optimization: Parallel and Serial Cases. Parallel Computing **37**(1) (2011) 60–68
13. Pham, D.N., Gretton, C.: gNovelty+. In: Solver description, SAT competition 2007. (2007)
14. Roli, A.: Criticality and Parallelism in Structured SAT Instances. In: CP'02. Volume 2470 of LNCS., Springer (September 2002) 714–719
15. Kroc, L., Sabharwal, A., Gomes, C.P., Selman, B.: Integrating Systematic and Local Search Paradigms: A New Strategy for MaxSAT. In: IJCAI'09. (July 2009) 544–551
16. Arbeleaz, A., Codognet, P.: Massivelly Parallel Local Search for SAT. In: ICTAI'12, Athens, Greece, IEEE Computer Society (November 2012) 57–64
17. Martins, R., Manquinho, V., Lynce, I.: An Overview of Parallel SAT Solving. Constraints **17** (2012) 304–347
18. Tompkins, D.A.D., Hoos, H.H.: UBCSAT: An Implementation and Experimentation Environment for SLS algorithms for SAT and MAX-SAT. In: SAT'04. Volume 3542 of LNCS., Springer (2004) 306–320
19. Achlioptas, D., Gomes, C.P., Kautz, H.A., Selman, B.: Generating satisfiable problem instances. In: AAAI'00. (July 2000) 256–261
20. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: TACAS'04. Volume 2988 of LNCS., Springer (March 2004) 168–176
21. Tompkins, D.A.D., Balint, A., Hoos, H.H.: Captain Jack: New Variable Selection Heuristics in Local Search for SAT. In: SAT'11. Volume 6695 of LNCS., Springer (June 2011) 302–316
22. Gent, I.P., Walsh, T.: The SAT Phase Transition. In: ECAI'94. (August 1994) 105–109