pFactory: A generic library for designing parallel solvers[1]

Gilles Audemard (1), Gael Glorian (1), Jean-Marie Lagniez (2), Valentin Montmirail (3), Nicolas Szczepanski (1)

(1) CRIL, CNRS, UMR 8188, Université d'Artois, Lens, France

(2) Huawei Technologies Ltd, French Research center

(3) Université Côte d'Azur, CNRS, I3S, Nice, France

ABSTRACT

With the advent of multi-core processors, it makes sense to design multithreaded solvers. Nevertheless, implementing such solvers is often a cumbersome task. Indeed, multithreaded SAT solvers are not easy to write, and only experienced programmers should undertake to code for these types of applications. To overcome this problem, we propose a new library that simplifies the design of multi thread solvers. Particular care was given to make efficient the transfer of a large amount of information between solver units. We show that it is easy to implement parallel solvers by using pFactory on SAT paradigm. We also experimentally validate that such solvers are competitive against state-of-the-art parallel solvers.

1. INTRODUCTION

Nowadays, combinatorial solvers are used in several domains. For example, mathematical conjecture, model checking, and planning are commonly solved using SAT solvers [10]. Nevertheless, problems are increasingly hard to solve, whereas it becomes more and more challenging to improve solvers.

In order to solve more instances within a reasonable amount of time, a basic approach from a technological side consists of running existing solvers on more efficient processing units. This seems to be interesting since Moore's Law states that processor speeds for computers will double every two years. Moore's Law may not be able to go on indefinitely, and even if high-tech industry might love to talk about the exponential growth, there are physical limits to the ability to continually improve the overall processing power for computers [17]. Gordon Moore himself, expect it will end by 2025. Thus, instead of increasing computational throughput, processor manufacturers prefer to arrange multiple processors onto the same chip, this referred to as multi-core architecture.

Such a strategy has been followed with some success for solving hard combinatorial problems, as illustrated by the performances of the existing parallel SAT solvers [3]. Basically, these parallel solvers can be divided into two categories: portfolio and divide-and-conquer. Roughly speaking, portfolio solvers try to solve the same formula with different strategies or even different solvers, whereas divide-and-conquer methods divide the original formula into smaller ones being easier to solve [4]. In order to improve their effectiveness, both approaches share information (*e.g.* learnt clauses).

Even if it seems natural to design parallel solvers in order to take advantage of technological advances, only a few parallel solvers have been designed so far. Indeed, multithreaded applications are generally hard to implement, and only experienced programmers should undertake to code for these kinds of applications. While sharing information between cores seems obvious, in practice we have to deal with a producers/consumers problem (also known as the bounded-buffer problem), and it is crucial to be cautious about the way concurrent accesses are managed. In practice, as reflected by the results of the winner of the 2018 parallel SAT competition, a good communication strategy is one of the key to be successful. It is essential to note that communications can really slow down the unit solver efficiency, which can make parallel solvers far less efficient than sequential solver. Indeed, the way data are exchanged as well as the data

[1] Authors were partially supported by the "SATAS" ANR-15-CE40-001

structures used to manage them, play an essential role in the overall effectiveness of the parallelization. Moreover, most of the parallel strategies request some strong abilities with synchronization, atomic operation, load balancing, and data consistency in order to avoid the common parallel pitfalls such as data-races or deadlocks.
To overcome these disadvantages, we propose pFactory, a parallel library designed to support and facilitate the implementation of parallel solvers in C++. It provides robust implementations of parallel algorithms and allows seamlessly sharing mechanisms, divide-and-conquer or portfolio methods. Even if we have the same guideline than the Painless library [6], contrary to it, we are not only restricted to design parallel SAT solvers. More precisely, our library is not related to a specific problem and can very easily be incorporated in order to solve another combinatorial problem. However, due to lack of space, we present in this paper only pFactory with SAT solvers. Finally, we try to make our library as least intrusive as possible. To sum up, our contributions are the following:

– We propose pFactory, a solver-independent and a problem-independent library that can be used to implement parallel strategies with only a few lines of codes (approximately 10 to 50 lines according to the parallel strategy to implement);
– We define a new communication algorithm and show its robustness compared to other approaches, especially when much information is shared;
– We evaluate our approach on SAT problems showing great performances;

The paper is organized as follows: the next section gives some preliminaries, presents related works and other parallel frameworks. Section 3 shows the concept of our library. In Section 4, we describe the communication algorithm we implemented in our library, and show differences with existing methods. Before concluding, we provide experiments showing competitive results.

2. PRELIMINARIES

A CNF formula is a conjunction of clauses built on a finite set of Boolean variables where a clause is a disjunction of literals. A literal is either a Boolean variable (x) or the negation of a Boolean variable (x) . A unit (resp. binary) clause is a clause with only one literal (resp. two literals), called unit literal (resp. binary clause). An interpretation assigns a value from 0, 1 to every Boolean variable, and, following usual compositional rules, naturally extended to a CNF formula. A formula φ is consistent or satisfiable when it exists at least one interpretation that satisfies it. This interpretation is then called a model of φ and is represented by the set of literals that it satisfies.
SAT is the NP-complete problem that consists in checking whether or not a CNF is satisfiable, i.e. whether or not the CNF has a model. Several techniques have been proposed to tackle this problem in practice. In this paper, we focus on CDCL SAT solvers exploiting the so-called Conflict Driven Clause Learning features (see e.g. [12]) which are currently the most efficient complete SAT solvers. Let us recall briefly the global architecture of a CDCL SAT solver. CDCL solving is a branching search process, where at each step a literal is selected for branching. Usually, the variable is picked *w.r.t.* the VSIDS heuristic [14] and its value is taken from a vector, called polarity vector, in which the previous value assigned to the variable is stored. Afterwards, Boolean constraint propagation is performed. When a literal and its opposite are propagated, a conflict is reached, a clause is learnt from this conflict and a backjump is executed. These operations are repeated until a solution is found (satisfiable) or the empty clause is derived (unsatisfiable). CDCL SAT solvers can be enhanced by considering restart strategies and deletion policies for learnt clauses. Among the measures used to identify relevant clauses, the Literal Blocked Distance measure (LBD in short) proposed in GLUCOSE is one of the most efficient. Then, as experimentally shown by the authors of [2], clauses with smaller LBD should be considered more relevant.
Parallel solvers can be divided into two categories. On the one hand, portfolio-based approaches [7, 15, 3], run different strategies/heuristics or even solvers concurrently, each on the whole formula. On the other hand, the well-known divide-and-conquer paradigm [9]. In such solvers, the search space is divided into sub-spaces, which are successively sent to sequential solvers running on different processors, so-called

workers. Generally, each time a solver finishes its task (while the others are still working), a load balancing strategy is invoked, which dynamically transfers sub-spaces to this idle worker. The sub-spaces can be defined using the guiding path concept [18], generated statically, i.e., before the search [9], or dynamically, i.e. during the search process [11, 9]. Both methods can share information between sequential solvers. In the case of SAT, learnt clauses are shared between sequential solvers. Since a large number of clauses are learnt through the search process, it is crucial to thoroughly choose the clauses that have to be shared [8, 3]. Furthermore, it is also essential to design carefully the sharing mechanism. However, this task is quite complex to achieve. To remedy this problem, a library, called Painless, has been recently developed [6].

3. GENERAL DESCRIPTION

Like most of the parallel tools, our library, called pFactory, is based on an abstraction of parallel processing composed by tasks. More precisely, it is based on the thread library provided by C++11 which gives necessary features to model parallel strategies and to perform communications. In order to make our library solver and problem independent, contrary to Painless, pFactory does not require implementation of interfaces. Instead, various objects that implement all functionalities (such as task management, synchronizations and communications) are provided.

```
 1  std::vector<Solver *> solvers;
 2  for (int num = 0; num < pFactory::getNbCores(); num++) //Create all solvers
 3    solvers.push_back(new Solver());
 4
 5  pFactory::Group* group = new Group(pFactory::getNbCores());  // Create the group
 6
 7  for (auto solver : solvers){
 8    solver->setGroup(group);    // Link the group to solvers
 9    group->add([=](){           // Create all tasks
10       return solver->solve();
11    });
12  }
13
14  group->start(true);  // Start all tasks in concurrent mode
15
16  std::cout << group->wait() << std::endl; // Wait and display the return code
```

Fig. 1: A simple concurrent (portfolio) strategy

Our aim is to propose a tool that is accessible, while remaining efficient and which is the least intrusive possible. To show how simple is the implementation of a parallel solver using pFactory, we present in the following a concurrent one. Of course, our library allows to implement a divide and conquer solver, but, due to lack of space, we do not present it in this paper.

Figure 1 depicts a simplistic concurrent mode composed of several sequential SAT solvers (we suppose a solver is an instance of the class Solver, which must have at least a public default constructor and a method solve()). This is all the code required to implement a parallel portfolio solver, which clearly shows the library easiness of usage. As highlight by the listing, only few steps are needed to parallelize a sequential solver. The first step is to create the given number of solvers as well as a Group object (lines 1 to 5). The group object is one of the principal components of our library; it allows to execute a set of tasks defined as lambda functions on a given number of threads. In the example, we choose to dynamically set the number of threads (the number of solvers in this case) to the number of physical cores (pFactory::getNbCores()) available on the computer. After the initialization step, the tasks are attached to the group (lines 7-12); in portfolio mode, a task is made of a sequential solver working on the original formula. At the end of this loop, each solver is associated with a task. Obviously for the sake of clarity, this example contains only one type of solver, but it could be easily extended to deal with several types of solver by adding them independently to the group. Moreover, as it is done implicitly by all the other libraries, we assume that the solvers are thread-safe before parallelized them. The solving process is launched using the method start (line 14); which runs the assigned tasks belonging to the group in parallel (i.e.the lambda function associated with each task). The method can take a Boolean variable as input enabling the concurrent mode; it allows, as soon as a task is completed, to stop all the other running tasks. To be able to achieve this behavior, each solver needs to know its group (line 8); this way, with a slight modification, solvers can stop their search. For example, this can be done at each decision and/or conflict by adding this line in the solver we search to parallelize: `if (group->isStopped()) return UNDEF;` Finally, the method `group->wait()` returns the value requested within the lambda function when it finishes its work (SAT or UNSAT for example).

4. COMMUNICATIONS

In a collaborative behavior, it is often the case that making possible communication between agents helps to improve group effectiveness. Indeed, it is well known that sharing learned clauses between solvers allows to improve the effectiveness of parallel SAT solvers. In this case, clause sharing can be viewed as the well known producers/consumers problem, where all producers are also consumers (considering every solver send their clauses to the others). In SAT, because of the abundance of clauses that are learned by the solvers (approximately 10^4 clauses per second), a special care is required to avoid a bottleneck at the communication level. Before explaining our method, we introduce the communication algorithms used in two state-of-the-art SAT solvers [3,6].

Communications in Syrup

In SYRUP [3], clauses are exported using a buffer of limited size (by default the size is set to 0.4MB $\times$ the number of threads) located in the shared memory, and a clause is removed from the buffer when all threads have imported it. Hence, when the buffer is full, oldest clauses that should have been exchanged are discarded and consequently not exported (even if it was not yet sent to all threads). The approach, therefore, prevents an overload of the shared memory.
Each clause is copied in the buffer (by value) with some extra information (its size and its source) and a specific value saving the number of times it has been imported. This last value allows detecting when a clause must be removed from the buffer, i.e. when this value is equal to the number of thread minus 1. Furthermore, a pointer is used to know the position of the last exported clause. Since it is a cyclic buffer, when this pointer is at the end, it comes back at the beginning and overwrites the oldest clauses. Finally, all operations done on the buffer are protected by a unique mutex.

Communications in Painless

PAINLESS [6] uses the communication algorithm proposed in [13], which try to reduce the critical section in order to avoid a bottleneck due to concurrent accesses of threads. For this purpose, each solver thread holds two buffers (of unlimited sizes): one for exportation and one for importation. In addition, data exchange is realized using a dedicated thread, called sharer. According to the exchange strategy, the sharer method gets clauses from the producers (exportation buffers) and copy (their pointer) to the consumers (importation buffers). The originality of this algorithm is its critical section, the buffers are not protected by a mutex but by a compare-and-swap instruction. It compares the content of a memory location P with a given value old value. If they are the same, it changes the contents of P by a new value. Since this instruction may fail and therefore not write the new value in the memory location, it is encapsulated into a while loop.

PFACTORY communications

In SYRUP, because the unique `mutex`, there is a high possibility of bottleneck. The cost to verify if a clause has already been shared can be important. Moreover, the clauses are copied literal by literal, whereas a copy of the address could have been faster. Even if PAINLESS tries to correct the SYRUP drawbacks with multiple buffers and copy of addresses, the busy waiting loop per data exchanged (compare-and-swap) can potentially lead to a high CPU consumption. Moreover, the dedicated thread `sharer` of PAINLESS moves data from exported buffers to imported ones every 0.5 seconds. But, sharing clauses as soon as possible among search units can provide better results [1]. Therefore, the problem is, without considering the copy time, some shared clauses can have an additional delay of transmission of 0.5 seconds. To avoid such pitfalls, we propose a new communication algorithm that considers some advantages of PAINLESS (several buffers) and SYRUP (the use of a `mutex`).

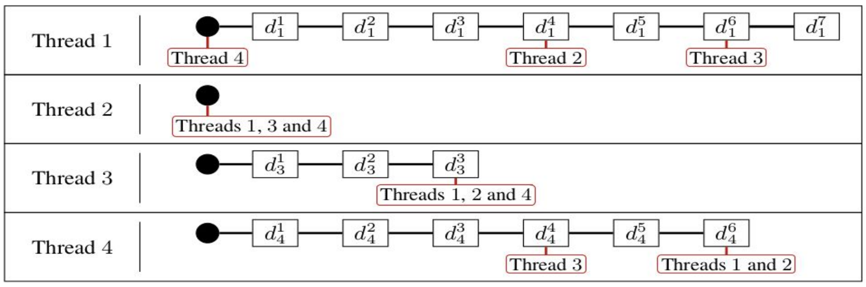


Fig. 2: Buffers of the PFACTORY communication algorithm

Algorithm In our approach, each thread i uses a buffer containing all data sent by the i^{th} thread. Figure 2 illustrates these buffers with 4 threads. Thus, to exchange all data, each thread i has to get the data present in buffers of threads j s.t. $j \neq i$. To do this, each thread i knows the positions in its buffer of other threads j. These positions designate data that were already imported and those that are not. In Figure 2, when we focus on the buffer of first thread, we can see that it has sent 7 data ($d_1^1, \ldots, d_1^7$). At this time, some other threads have already received some data from the thread 1 (d_1^1, d_1^2, d_1^3 and d_1^4 for the thread 2, $d_1^1, \ldots, d_1^6$ for the thread 3 and (nothing) for the thread 4). Threads 2, 3 and 4 have sent, respectively, no data, 3 data (d_3^1, d_3^2, d_3^3) and 6 data ($d_4^1, \ldots, d_4^6$).
Each buffer is protected by a `mutex`, *i.e.* each operation (adding/removing) in the buffer is realized inside of a lock/unlock part with this `mutex`. A send operation from a thread is performed by putting the data at the end of its buffer. Receiving operations are more complex. A new temporary reception buffer has to be used to store the addresses of collected data. For a thread i wishing to collect data, we iterate on the other threads j s.t. $j \neq i$ and execute the following functions:

– `collectData()`: Get the position p of the thread i in the buffer of j. All data from p to the end of the buffer are added in the reception buffer of i.

– `updatePosition()`: Recall that each thread i knows the positions in its buffer of others threads. To be able to delete data received by all threads, we also need to get the thread b that collected less data (the more left thread in the buffer of i in Figure 2, $b = 4$ when $i = 1$ for example). Another queue of threads (one by buffer) is then used to keep at any time the order of the thread positions (of each buffer). Queues are updated by moving to the end the thread that just gets data using function `collectData()`.

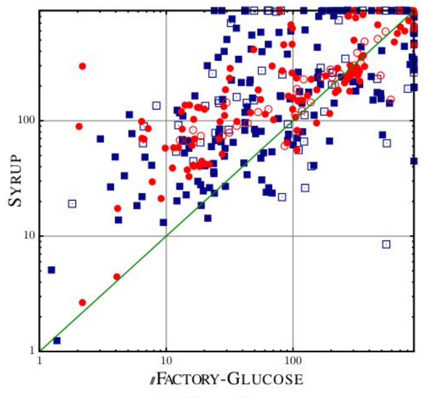
– `removeData()`: When the most backward thread is at position b in the buffer i, data from the beginning to the position b can be removed from the buffer. Indeed, these data were already received by all other threads. Note that, when b is at the end of the buffer, all data can be deleted (all threads have already received all data).

For each buffer, the critical section is composed of these three functions. The size of critical sections has an important impact on the sharing efficacy. For this purpose, we use data structures based on a double-ended queue to remove element in constant time. Note also that the number of operations of each function is minimal (the number of data to receive for `collectData()`, 4 operations for `updatePosition()` and the number of data to delete for `removeData()`). Finally, the temporary reception buffer and the copy of addresses also helps to ensure the best possible performances.

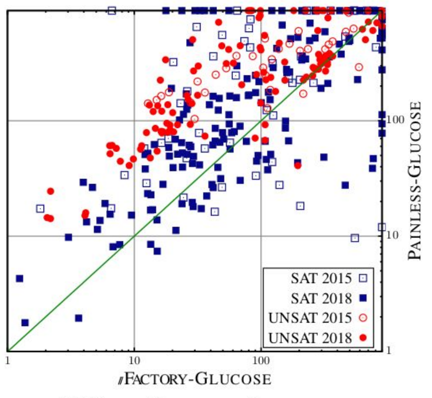
Communicators To make user-friendly the usage of communications, our library contains an object called `Communicator<T>` using templates. The object allows to implement any communication algorithms and its two main operations (`send()` and `recv()`) are drawn from MPI (Message Passing Interface). The template allows defining the type of the data to be exchanged and to choose if the copy is done by values (for primitive type) or by addresses (for more complex type). Hence, memory management is left to the user. The following line shows the creation of a `communicator` used to share clauses in any *Minisat* [5] like solver (glucose [2], Maple): `pFactory::Communicator<vec<Lit>*>* clausesCommunicator = new MultipleQueuesCommunicator<vec<Lit>*>(group);`
In this example, clauses are represented by vectors of literals (vec<Lit>*) like in MINISAT-based solvers and the star (`*`) shows that the copy is done by addresses. Note that <vec<Lit>*> is a template argument and can be changed to another type (<int*>, <char*>, …) in others applications. The communicator is associated with a group (parameter of the constructor). Using this technique, one can add as many communicators as needed and associate them to different groups. To exchange information between workers, one has to use the functions `send(vec<Lit>*)` and/or `recvAll(vec<vec<Lit>*>)` where the parameters are, respectively, the clause to send and the vector of all clauses to receive.

5. EXPERIMENTS

The experiments conducted in this section were run on a computer containing 4 Xeon processors. Each processor has 20 cores at 2.4 GHz, making 80 cores available per solver per instance. The runtime limit was set to 900 seconds for each instance and the memory limit was set to 768 GB. All data (source code, logs, ...) are available here https://bit.ly/2W6WqJr. The library is available at http://githubcom/crillab/pfactory. In order to make the experimental evaluation fair, all the parallel solvers implemented in this paper are based on the same sequential SAT solver: GLUCOSE 4.1. These solvers are: PAINLESS-GLUCOSE (the authors kindly provided a GLUCOSE-based version), SYRUP [3] (the *official* parallel version of GLUCOSE) and our solver, called PFACTORY-GLUCOSE. The same strategy as the one used in SYRUP [3] has been implemented to share the learnt clauses.



(a) Versus SYRUP



(b) Versus PAINLESS-GLUCOSE

SAT Competition	2015			2018		
Solver	SAT	UNSAT	Total	SAT	UNSAT	Total
GLUCOSE	9	6	15	67	66	133
PAINLESS-GLUCOSE	42	32	74	141	99	240
SYRUP	43	32	75	141	114	255
//FACTORY-GLUCOSE	49	32	81	149	114	263

(c) Number of instances solved

Fig. 3: Comparing pFactory-GLUCOSE against PAINLESS-GLUCOSE (Figure 3b) and pFactory-GLUCOSE against Syrup (Figure 3a) on both SAT'15 (100 instances) and SAT'18 (400 instances) competitions. For each scatter plot, each dot represents an instance, the X-axis and the Y-axis represent the runtimes of approach compared, dots above the diagonal correspond to instances solved faster by pFactory-GLUCOSE. Table 3c presents the number of instances solved according to their solution (Satisfiable or Unsatisfiable).

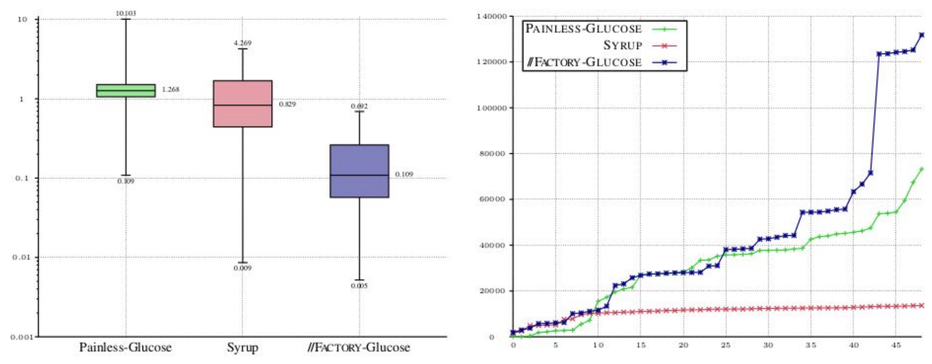
Besides the 100 instances of the parallel track of the 2015 SAT competition, we also considered 400 new benchmarks originating from the application track of the 2018 SAT competition. Figure 3 highlights the results achieved by the different approaches we considered. We considered two kinds of evaluation. Two scatter plots, that make an apple-to-apple comparison between pFactory-Glucose against Syrup (Figure 3a) and Painless-Glucose (Figure 3b), are reported. As we can see, on the set of problems considered, pFactory-Glucose clearly outperforms Syrup and Painless-Glucose. The scatter plots show that our solver is generally faster than the two others. Particularly, the cumulated solving time for the instances that are solved by all solvers are, respectively, for Painless-Glucose, Syrup, and pFactory-Glucose, 4h47m, 3h29m, and 2h48m for the SAT'15 instances, and, 10h34m, 10h01m, and 7h05m for SAT'18 ones. Regarding pFactory-Glucose, it is comparable to a gain of 20% (resp. 29%) versus Syrup and 41% (resp. 33%) versus Painless-Glucose on the SAT'15 (resp. SAT'18) instances. To better understand why our library is much more efficient, we did some additional experiments.

Figure 4 displays a study of the communication costs of all instances of the parallel track of the 2015 SAT competition. We used the profiler OProfile (available in https://oprofile.sourceforge.io) in order to measure the CPU usage of communications. Figure 4a shows the outputs of this profiler on the communication functions. The obtained results are the percentages of CPU usage for the communication functions (the y-axis) by the three parallel solvers (the x-axis) in the form of box-and-whisker plots. The bottom and top of each box are respectively the first and third quartiles, and the line inside the box is the second quartile (the median). The ends of the whiskers represent the minimum and maximum of all data. This figure shows that, in general, our communication algorithm is ten times less CPU-intensive than the two other programs.

A low CPU usage could mean a smaller amount of imported clauses; the next experiment clearly shows this is not the case in our communication algorithm. To support our claims, we computed the average number of clauses imported per second and per thread, on instances not solved by all solvers within a timeout of 100 seconds. Discarding easy instances provides a more accurate depiction. Figure 4b exhibits a cactus plot, designating for this period of time, the number of imported clauses per second and per thread, on the intersection of 48 unsolved instances by the three parallel solvers. We can see that pFactory-Glucose shares at least as many clauses as other methods, while Syrup appears to cap at about $13,000$ clauses per second. This phenomenon can be explained by the limited size of its exchange buffer. Note that our algorithm scales, when a larger quantity of clauses is shared, pFactory-Glucose exchanges more clauses than Painless-Glucose (up to $40,000$ clauses per seconds). Hence, our library exchanges more clauses in a reduced amount of time.

6. CONCLUSION

We have presented pFactory, a library designed to make simple and efficient implementation of parallel solvers. The library is the least intrusive possible and involved few modifications inside the embedded solver. As designated in the experimental section, pFactory also provides a simple but extremely effective communication unit. Of course, our library is not only related to SAT problems. One can use it on other kinds of combinatorial problems, CP (Constraint Programming) and SMT (SATisfiability Modulo Theory) for example. Since we want to help the user in the development of parallel solvers, we are aiming to design other generic tools like an object in charge of the creation of subproblems independently of the kind of solved problems in the context of divide-and-conquer parallel paradigm. Finally, we also plan to extend our library in a distributed environment (data exchange among different computers by message passing).



(a) CPU usage of communications (b) Speed of communications

Fig. 4: Study of the communication costs on SAT'15 instances (parallel track). Fig. 4a shows some box-and-whisker plots on the percentage of CPU usage of communications and Fig. 4b displays a cactus plot on the number of imported clauses per second for the intersection of the unsolved instances by the three solvers during 100 seconds.

REFERENCES

1. Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary. A Distributed Version of Syrup. *Theory and Applications of Satisfiability Testing - SAT 2017*, pages 215–232. Springer, 2017.

2. Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. *IJCAI 2009,* pages 399–404, 2009.

3. Gilles Audemard and Laurent Simon. Lazy Clause Exchange Policy for Parallel SAT Solvers. *Theory and Applications of Satisfiability Testing - SAT 2014,* pages 197–205. Springer, 2014.

4. Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013, 2013.

5. Niklas Een and Niklas Sorensson. An Extensible SAT-solver. *Theory and Applications of Satisfiability Testing- SAT 2003,* pages 502–518. Springer, 2003.

6. Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. PaInleSS: A Framework for Parallel SAT Solving. *Theory and Applications of Satisfiability Testing - SAT 2017,* pages 233–250. Springer, 2017.

7. Youssef Hamadi, Saıd Jabbour, and Lakhdar Sais. Manysat: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2009.

8. Youssef Hamadi, Saıd Jabbour, and Lakhdar Sais. Control-Based Clause Sharing in Parallel SAT Solving. *Autonomous Search*, pages 245–267. Springer, 2012.

9. Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and Conquer: Guid- ing CDCL SAT Solvers by Lookaheads. *Hardware and Software: Verification and Testing Conference, HVC 2011*, pages 50–65. Springer, 2011.

10. Armin Biere, Marijn Heule, H. van Maaren, and Toby Walsh. Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam, The Netherlands, 2009.

11. Antti Eero Johannes Hyvarinen, Tommi Junttila, and Ilkka Niemela. Partitioning SAT Instances for Distributed Solving. *Logic for Programming, Artificial Intelligence, and Reasoning -LPAR-17,* pages 372–386. Springer, 2010.

12. Joao Marques-Silva and Karem. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *International Conference on Computer-aided Design* - ICCAD'96, pages 220–227, 1996.

13. Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. *ACM Symposium on Principles of Distributed Computing,* pages 267–275. ACM, 1996.

14. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. *Design Automation Conference, DAC 2001,* pages 530–535. ACM, 2001.

15. Olivier Roussel. Description of ppfolio. Technical report, 2011.

16. The 2018 sat competition, 2018. http://sat2018.forsyte.tuwien.ac.at/.

17. M. Mitchell Waldrop. The chips are down for Moore's law. *Nature*, 530(7589):144–147, 2016.
18. Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a Distributed Propositional Prover and its Application to Quasigroup Problems. *Journal of Symbolic Computation*, 21(4):543–560, 1996.