EasyChair Preprint

№ 8553

DPS: a Framework for Deterministic Parallel SAT Solvers

Hidetomo Nabeshima, Tsubasa Fukiage, Yuto Obitsu, Xiao-Nan Lu and Katsumi Inoue

August 2, 2022

DPS: A Framework for Deterministic Parallel SAT Solvers

Hidetomo Nabeshima[1], Tsubasa Fukiage[1], Yuto Obitsu[1], Xiao-Nan Lu[1], and Katsumi Inoue[2]

[1] University of Yamanashi, JAPAN
{nabesima,xnlu}@yamanashi.ac.jp
[2] National Institute of Informatics, JAPAN
inoue@nii.ac.jp

Abstract

In this study, we propose a new framework for easily constructing efficient deterministic parallel SAT solvers, providing the delayed clause exchange technique introduced in ManyGlucose. This framework allows existing sequential SAT solvers to be parallelized with just as little effort as in the non-deterministic parallel solver framework such as PaInleSS. We show experimentally that parallel solvers built using this framework have performance comparable to state-of-the-art non-deterministic parallel SAT solvers while ensuring reproducible behavior.

1 Introduction

The efficiency of the Boolean satisfiability (SAT) solvers makes it a widely used fundamental tool for problem solving in a variety of application areas, including hardware and software verification [4], planning [11], scheduling [7], and combinatorial designs [18]. Today, with the spread of multi-core computing environments, parallel SAT solving has become desirable to achieve more efficient problem solving through effective use of computing resources.

Since SAT Race 2008, competitions for the performance of parallel solvers have been held continuously, and many parallel SAT solvers have been developed. With few exceptions, many of them do not guarantee reproducible behavior in order to maximize their performance. This means that the execution time may vary from run to run, and a found assignment in satisfiable cases may be different. This is in contrast to most sequential solvers, which guarantee reproducible behavior. A solver whose results are reproducible (or non-reproducible) is also called *deterministic* (or *non-deterministic*). The non-deterministic behavior of solvers can be an obstacle in application areas. For example, in model checking, different bugs (corresponding to satisfiable assignments) may be found in different runs. In the case of scheduling, even if a good schedule is found, it may not be reproduced the next time. In the development of parallel SAT solvers, instability of execution results leads to difficulty in tuning performance.

The cause of non-reproducible behavior in parallel solvers is due to the lack of reproducibility in the process of exchanging learnt clauses between workers. A simple solution is for all workers to synchronize before communication and then exchange clauses according to a fixed order on the workers. This is the method implemented in the first deterministic parallel SAT solver ManySAT 2.0 [9]. Although this method is easy to implement, the waiting time for synchronization becomes a non-negligible burden as the number of workers increases. The delayed clause exchange introduced in ManyGlucose [15] allows a certain delay in the timing of exchanges, thereby absorbing fluctuations in the exchange interval and significantly reducing the waiting time, even in many-core environments. ManyGlucose took third place in the parallel track of the SAT competition 2020. However, implementing delayed clause exchange requires expert knowledge of concurrent programming, so introducing it into existing sequential SAT solvers

is a time-consuming task. In this work, we propose a framework that enables the construction of deterministic parallel SAT solvers based on the delayed clause exchange with little effort. The work required for parallelization is almost the same as that required in PaInleSS [12], a general framework for building non-deterministic parallel SAT solvers, which mainly consists of clause import/export interfaces. In addition to these, our framework requires measurement of the amount of processing inside the base solvers to determine the clause exchange interval. Experimental results show that deterministic parallel SAT solvers based on our framework can achieve performance comparable to state-of-the-art non-deterministic parallel SAT solvers.

The paper is organized as follows: Section 2 describes non-deterministic and deterministic parallel SAT solvers. Section 3 reviews the delayed clause exchange which is the main feature of our framework. In Section 4 we propose our framework called DPS. Section 5 presents implementation details and experimental results. Section 6 concludes and discusses future work.

2 Parallel SAT Solvers

Parallel SAT solvers are often categorized as portfolio, divide-and-conquer, or a hybrid of the two. In the portfolio approach [10], many sequential solvers, referred as *workers* in this study, competitively solve the same problem instance in parallel. In the divide-and-conquer approach, a given problem is decomposed recursively using the guiding path method [19], and workers solve subproblems in parallel. In both approaches, workers exchange learnt clauses with each other to reduce overlap in the search space.

In this study, we classify parallel SAT solvers according to a criterion different from the aforementioned one: whether or not they guarantee reproducible behavior. When reproducible, they are also referred to as *deterministic* and when not, as *non-deterministic*.

Most of parallel SAT solvers are non-deterministic. It is a well-known behavior of non-deterministic parallel SAT solvers that the execution time and the found model in satisfiable cases are different for each run. The lack of reproducibility is due to the asynchronous exchange of learnt clauses between workers[1]. Especially for portfolio parallel SAT solvers, where each worker competitively solves the same instance, the learnt clause exchange is an important cooperative mechanism to reduce overlap in each worker's search space. Non-deterministic parallel solvers perform the clause exchange asynchronously. This is to avoid the latency that would occur when synchronized and to maximize performance. The timing of sending and receiving clauses is usually affected by system workload, cache misses, and communication delays. Therefore, asynchronous exchange will result in non-reproducible behavior.

If all workers synchronize before exchanging and then exchange clauses in a certain order between workers, the reproducibility of execution results can be guaranteed[2]. ManySAT 2.0 [9] is the first deterministic parallel SAT solver[3]. It is a portfolio parallel SAT solver for shared memory multi-core systems. ManySAT synchronizes all workers before and after each clause exchange. This synchronization could easily be implemented with OpenMP's barrier instruction[4], but synchronizing all workers would increase latency and significant performance degradation. ManyGlucose is another deterministic parallel SAT solver based on Glucose-syrup [2] and imple-

[1] Some parallel solvers, such as ppfolio [16] and PaKis [17], do not exchange any information between workers. They simply run sequential solvers of different kinds and search strategies independently and in parallel. This study assumes information exchange among workers.

[2] We assume that the sequential SAT solver run by each worker guarantees a deterministic behavior.

[3] ManySAT 2.0 supports both deterministic and non-deterministic behavior.

[4] The barrier is implemented by `#pragma omp barrier` directive in OpenMP.

ments the delayed clause exchange technique [15] (see next section), which can reduce waiting time by allowing a certain delay in the timing of clause exchange, and achieves performance comparable to non-deterministic parallel SAT solvers. However, its implementation requires expertise in concurrent programming, and building a deterministic parallel SAT solver based on existing state-of-the-art sequential SAT solvers would involve significant effort. In this study, we propose a framework that facilitates the construction of deterministic parallel SAT solvers.

3 Delayed Clause Exchange

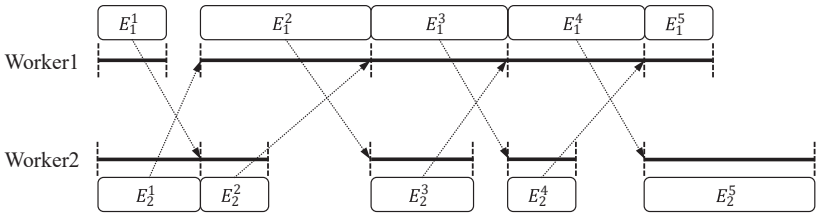
This section provides an overview of the *delayed clause exchange* (DCE) [15].

An important issue in deterministic parallel SAT solvers is how to define the interval between clause exchanges. For example, if the interval is defined as "every second," reproducibility cannot be guaranteed. This is because the amount of processing performed per second inside the solver may vary due to system load, CPU cache usage, and time measurement errors. Therefore, the interval between clause exchanges must be defined on the basis of a reproducible measure. The interval defined in ManySAT is based on the number of conflicts encountered during the search, while ManyGlucose is based on the number of literal accesses or the number of executions of each code block (in C++, the statements enclosed in curly braces).

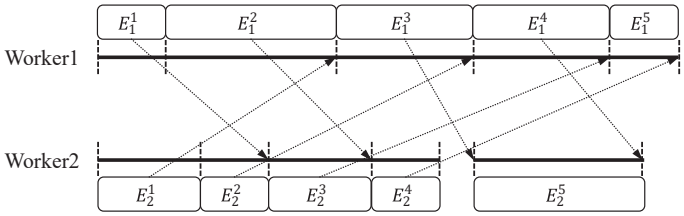
The interval between adjacent clause exchanges is called a *period*. Note that the time it takes to execute a period usually varies from period to period. For example, each worker uses a different search strategy to ensure diversity, which affects the frequency of conflicts and the number of literal accesses, resulting in differences in the execution time of each period. There are also factors that are difficult to predict such as system load and CPU cache usage. To build an efficient deterministic parallel SAT solver, it is important to align the period execution times for each worker as closely as possible, but fluctuations in execution times are inevitable.

In DCE, fluctuations are absorbed by allowing a delay in the exchange of learnt clauses. Let n be the number of workers, $T = \{1, \ldots, n\}$ the set of worker IDs, p_t the current period ID of a worker $t \in T$ ($p_t \geq 1$), E_t^p a set of clauses exported by a worker t at a period p and m an admissible delay, called *margin*, is denoted by the number of periods ($m \geq 0$). In DCE, each worker t imports clauses $E_i^{p_t - m}$ from another worker $i(\neq t)$ every time its latest period p_t ends, where $E_i^{p_t - m}$ is a set of learnt clauses acquired by the worker i in the period $p_t - m$. If $p_i < p_t - m$, worker t waits until worker i finishes the period $p_t - m$. Intuitively, when worker t completes the current period p_t, it is expected that other workers have already completed period $p_t - m$. In other words, if the period difference between the fastest and slowest workers is less than m, there is no latency. However, the receiver always takes clauses acquired by other workers m periods past.

Figure 1 shows an example of waiting time reduction by DCE between 2 workers. (a) and (b) show the process flow of clause exchange for margins $m = 0$ and $m = 1$, respectively. The former corresponds to ManySAT. In this case, it is necessary to wait for another worker after the end of worker 1's period 1, worker 2's periods 2, 3, and 4 (the white gaps between periods denote the waiting time). If m=1, some of these waits are unnecessary, and only after worker 2's period 4, it is necessary to wait for worker 1's period 3 to end. That is, worker 2 needs to wait for the completion of the construction of E_1^3 in order to import it before starting period 5 (if $m = 2$, there is no need to wait in this example). In this way, DCE can absorb some of the fluctuations in the period execution time of each worker, thus significantly reducing waiting time. Note that the execution time of each period in this figure does not change in (a) and (b) to make the effect of DCE easier to understand. In practice, the execution time of each period may differ in (a) and (b) due to the different learning clauses received. ManyGlucose, which



(a) margin $m = 0$



(b) margin $m = 1$

Figure 1: Examples of DCE between 2 workers. Bold lines surrounded by vertical dashed lines at both ends represent periods, and the gap between the bold lines represents the waiting time. E_t^p denotes the learnt clauses acquired by worker t at period p that are to be exported. The dotted arrows indicate clause importation, which are performed before the start of each period.

implements DCE, shows performance comparable to the non-deterministic parallel SAT solver Glucose-syrup both of which parallelize Glucose [1].

The implementation of DCE requires expertise in concurrent programming. For example, if the difference in periods between workers is more than or equal to m, the preceding worker must wait for the slower worker. This is accomplished by wait and signal calls for condition variables in POSIX threads. Each worker t has a queue of learnt clauses $E_t^1, E_t^2, \cdots$. During the construction of each E_t^p, there are only write accesses from the owner thread t, and after construction, there are only read accesses from other threads, so no exclusive processing is required. However, when enqueue/dequeue operations are performed on the queue[5], appropriate exclusion is required. Thus, implementing DCE is not an effortless task, and we propose a framework with a DCE implementation.

4 A Framework for Deterministic Parallel SAT Solvers

In this section we propose DPS, a framework for building deterministic parallel SAT solvers, which targets the construction of portfolio parallel SAT solvers in a shared-memory many-core environment.

For non-deterministic parallel SAT solvers, there is a generic framework called PaInleSS [12] that makes it easy to incorporate existing sequential SAT solvers or develop new strategies for parallel SAT solvers. PaInleSS provides adapter classes to incorporate the popular SAT solvers and representative strategies for clause exchange, portfolio and divide-and-conquer. Although

[5]When worker t starts a new period p, an empty E_t^p is enqueued to the queue. Also, $E_t^{p'}$ whose clauses have been already exported to all workers except the owner t is dequeued.

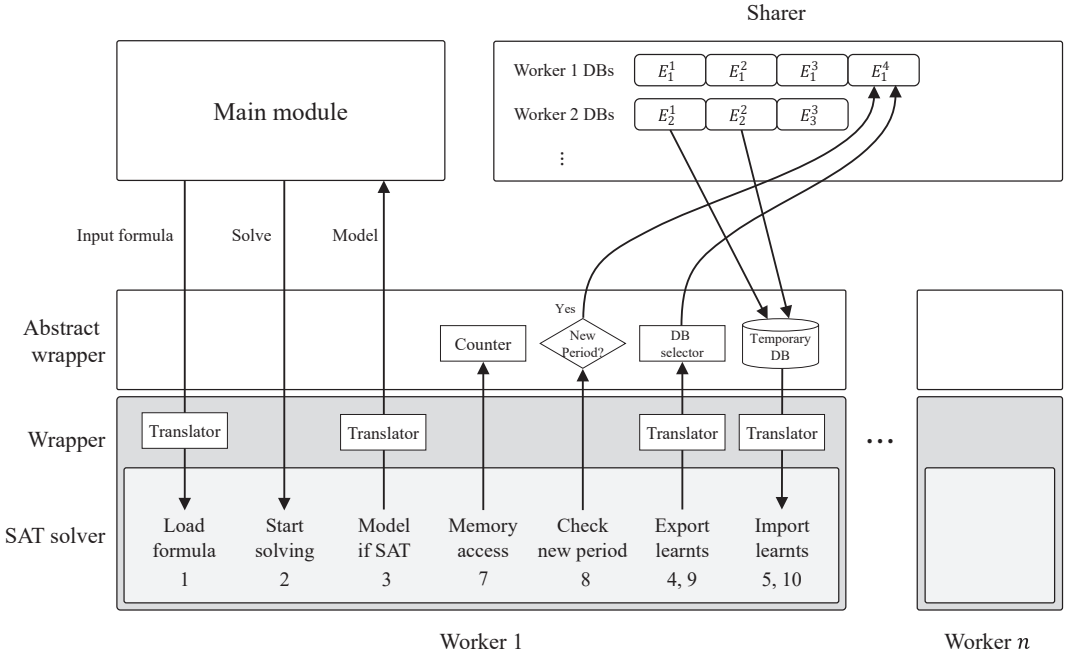


Figure 2: Overview of DPS framework.

our framework supports only simple clause exchange and portfolio strategies, it enables the realization of a deterministic parallel SAT solver based on existing SAT solvers with nearly the same effort as PaInleSS.

Figure 2 shows an overview of the DPS framework. DPS is written in C++ and consists of about 3000 lines (excluding option parser libraries, etc.). Parallel processing is achieved with the standard POSIX threads library and thread-related features of C++11, and can be compiled in a standard environment without installing any special libraries. There is a main module that loads the given instance into each worker, instructs them to start solving, and receive the satisfiability of the instance found by one of them. As an intermediary between the main module and the various sequential SAT solvers (light gray area in the figure), DPS has wrapper classes (dark gray area) that absorb the differences between the base SAT solvers and provides a unified interface. An instance of this wrapper class is corresponding to a worker. When introducing a new sequential SAT solver to DPS, the user needs to define this wrapper class. This wrapper class inherits the abstract wrapper class provided by DPS, which has a mechanism to realize reproducible behavior. First, the following three types of method definitions are required for the wrapper class. The numbers of the following list correspond to the numbers in the figure.

1. `bool loadFormula(DPS::Instance& clauses)`: loads CNF formula into the base solver.
2. `DPS::SATResult solve()`: solves the loaded formula and returns the result (satisfiable, unsatisfiable, or unknown due to resource limitations (time or memory)).
3. `std::vector<int> getModel()`: returns a model if the worker found to be satisfiable.

The role of these methods in the wrapper is to call the corresponding method in the base solver and to translate (or back-translate) the clause and variable representations in DPS into the ones in the base solver. Next, we need methods for exporting and importing learnt clauses.

4. `void exportClause(BaseSolver::Clause learnt)`: exports a learnt clause acquired by the base solver to the database of the current period after translating it into a DPS representation if it is worth exporting.

5. `void importClauses()`: gets clauses from a temporary pool that holds all clauses imported to this worker and adds them to the base solver. The temporary pool is provided by the abstract wrapper class and will be emptied after import.

These two methods may be more appropriately placed in the base solver rather than in a wrapper if they require access to private members of the base solver. The base solver then requires the following modifications:

6. Holds a pointer to the wrapper class object in order to call methods of it.

7. Counts the number of major memory accesses in the base solver. This is because DPS defines the length of a period based on that value. The counting is achieved by incrementing the thread-specific global variable `DPS::num_mem_accesses` (using `thread_local` specifier of C++11). Counting memory accesses as a measure of processing time corresponds to what Donald Knuth calls "mems". The number of memory accesses is a reproducible measure, and such a measure is essential for DPS.

8. Periodically calls `checkPeriod()` function implemented in the abstract wrapper class. This function is the key element for reproducible behavior, and it performs the transition to a new period if `DPS::num_mem_accesses` exceeds a threshold. In that case, it waits for workers that are slower than the margin m, if any, and then fetches clauses m periods past from all workers and holds them in the temporary pool. A new database is then created to store clauses for export in the new period (E_1^4 in the figure).

9. When learnt clause is acquired, calls `exportClause(learnt)` of the wrapper class.

10. When it becomes possible to import clauses (e.g., when the decision level is 0), calls `importClauses()` of the wrapper class to import clauses from the temporary pool.

11. Periodically calls `shouldBeTerminated()` function implemented in the abstract wrapper class. This function returns true if another worker finishes solving first or if the resource limit is exceeded. In this case, the base solver must exit immediately.

Except for 7 and 8, the requirements are essentially the same as those in PaInleSS.

In addition to the above, it is necessary to introduce strategies to ensure search diversity. In non-deterministic parallel SAT solvers, one source of diversity is due to the non-determinism of the clause exchange. However, this is not expected in deterministic parallel SAT solvers, making diversity more important. In the next section, we present our diversity strategies for several base SAT solvers.

Algorithm 1 shows the modifications required for the base solver in pseudo code, with the exception of lines 3-5 and 9 (red colored lines), which show typical CDCL procedure. When a conflict occurs in unit propagation (line 6), it is unsatisfiable if no decision has been made (line 7); otherwise, a learnt clause is acquired by conflict analysis (line 8) and backtracked to the appropriate level (line 10). If there are no conflicts in the unit propagation, pick an unassigned variable and assign a value to it (line 12). If there are no unassigned variables, then it is satisfiable. The modifications to the CDCL algorithm are as follows: lines 3-5 are executed periodically and consists of a termination check of solving (item 11 above), importing clauses (items 5,10), and calling `checkPeriod()` function (item 8), respectively. The implementation

Algorithm 1: Modifications to existing SAT solvers

```
1  Function solve()
2      loop
3          if wrapper.shouldBeTerminated() = true then return unknown;
4          if noDecision() = true then  wrapper.importClauses() ;
5          wrapper.checkPeriod();
6          if propagate() = false then
7              if noDecision() = true then return unsat;
8              learnt ← analyze();
9              wrapper.exportClause(learnt);
10             backtrack();
11         else
12             if decide() = false then return sat;
```

Listing 1: Counting the number of memory accesses

```
1  namespace DPS {
2    extern thread_local uint64_t num_mem_accesses;
3  }
4  class Clause {
5                  ⋮
6    Lit&     operator [](int i)       { DPS::num_mem_accesses++;
7                                         return data[i].lit; }
8    Lit      operator [](int i) const { DPS::num_mem_accesses++;
9                                         return data[i].lit; }
10   operator const Lit* (void)  const { DPS::num_mem_accesses++;
11                                        return (Lit*)data; }
12   float&   activity   ()             { DPS::num_mem_accesses++;
13                                        return data[header.size].act; }
14   uint32_t abstraction() const       { DPS::num_mem_accesses++;
15                                        return data[header.size].abs; }
16                 ⋮
17 };
```

of `checkPeriod()` is provided by the abstract wrapper class in DPS, so the base solver only needs to call it. Line 9 exports an acquired learnt clause (items 4,9).

Counting memory accesses may seem like a daunting task. If the base solver has a class that represents a clause, it is sufficient to count the number of literal accesses in that class. Listing 1 shows an example of the modification in the clause class of a MiniSAT [8] family of solvers. It only increments the variable `DPS::num_mem_accesses` in five methods. Since it is a thread-specific global variable, there is no need to carry around a pointer to the wrapper. The experimental results in the next section show that even with this simple measurement method, a wait time reduction comparable to ManyGlucose can be achieved.

Incorporating a solver written in C, such as Kissat [5] into DPS, requires additional interfaces that mediate between the two languages: interfaces to call C++ member functions from C, and vice versa. In Kissat, a clause is defined by a structure in C, and all members are public, so it is not easy to count the number of accesses to the literals contained in this structure.

However, Kissat already has a mechanism to measure the number of memory accesses [5]. It is called "ticks," which counts cache line accesses, and is a refinement of what Donald Knuth calls "mems". This is used to determine when to switch between search strategies and various inprocessing techniques, and was introduced to ensure reproducible behavior across runs. Therefore, it is relatively easy to incorporate Kissat into DPS by using ticks.

DPS supports not only deterministic execution, but also non-deterministic execution that requires no waiting time. Both execution modes are supported by the framework, and no modifications to the base solver are required for this. In non-deterministic mode, even if there is a slow worker over the margin, the preceding worker does not wait for it and moves to a new period (although the margin is meaningless in non-deterministic mode). The clause database of the slower workers will be imported after its construction is completed (i.e., no exported clauses are lost if we do not wait for slower workers).

5 Implementation and Experimental Results

We have constructed four deterministic parallel SAT solvers using DPS with MiniSAT 2.2 [8], Glucose 3.0 [1], MapleCOMSPS [13] (hereinafter referred as MCOMSPS), and Kissat-SC2021 [6] as base solvers, respectively. The latter two solvers are also executed as non-deterministic solvers for comparison. The source code and all experimental results (including additional results) are publicly available[6]. MiniSAT, Glucose, and MCOMSPS were selected for comparison with the parallel SAT solvers ManySAT, ManyGlucose, and PaInleSS [12], which use these as their base solvers. PaInleSS with MCOMSPS was the winner in the parallel track of the SAT Competition 2021. Kissat is a state-of-the-art sequential SAT solver that won the main track of the SAT Competition 2020.

For each of MiniSAT, Glucose and MCOMSPS, the creation of the wrapper classes and the modification to the base solver took about 300 lines. Kissat took about 800 lines, including the interface between C++ and C. The diversity strategy among workers and the sharing strategy of learnt clauses are as follows:

- DPS-MiniSAT: the diversity and sharing strategies are the same as in ManySAT. That is, except for the first worker, variables are selected randomly until the first conflict occurs. The random seeds use different values for each worker. Learnt clauses of length 10 or less are shared.
- DPS-Glucose: The diversity strategy is the same as ManySAT, which is considerably simpler than that of Glucose-syrup (it chooses among several strategies based on the results of an initial search. ManyGlucose is also the same as Glucose-syrup). The sharing strategy is the same as in Glucose-syrup and ManyGlucose, exporting learnt clauses whose LBDs are 2, or LBDs and lengths are less than or equal to 7 and 24, respectively.
- DPS-MCOMSPS, NPS-MCOMSPS: NPS means non-deterministic mode. The diversity and sharing strategies are similar to those of PaInleSS-MCOMSPS, which uses four different variable selection heuristics, in addition to which we also use the ManySAT diversity strategy. PaInleSS's sharing strategy is based on the HordeSat [3] strategy, which shares 1500 literals of clauses every 0.5 seconds. If the shared literals do not reach 1500, the LBD threshold for determining export clauses is raised, and vice versa. DPS shares 150 literals per period. This value was determined from preliminary evaluations.

[6]The source code and all experimental results are available at https://github.com/nabesima/DPS-pos2022 and https://nabesima.github.io/DPS-pos2022/, respectively.

- DPS-Kissat, NPS-Kissat: the diversity strategy, in addition to the ManySAT strategy, disabled elimination in half of the workers. The sharing strategy is the same as in MCOMSPS described above.

We use the diversification strategy used in ManySAT in each parallel solver. This is because diversity is more important for the deterministic parallel SAT solvers, as mentioned in the previous section. Clauses imported to a worker are added to the base solver when the solver reaches decision level 0. This is the same as in Glucose-syrup, ManyGlucose, and PaInleSS-MCOMSPS. In ManySAT, imported clauses are added immediately after backtracking to a level at which they are consistent. We chose the former method because it is easier to implement and does not interfere too much with the search in the base solver.

We fixed the margin of DCE in DPS at 20 and 0 for NPS[7], and set the number of memory accesses per period at 5 million for all but Kissat and 1 million for Kissat based on preliminary evaluations. In the experiments described below, we disabled preprocessing in MiniSAT, Glucose, and MCOMSPS. This is because preliminary evaluations showed that not only DPS but also ManySAT and PaInleSS performed better without preprocessing.

We compared our solvers with the following parallel SAT solvers:

- Non-deterministic solvers
  - Glucose-syrup 4.1: Glucose-based parallel solver.
  - PaInleSS-MCOMSPS: winner of the parallel track of the SAT Competition 2021.
- Deterministic solvers
  - ManySAT 2.0: MiniSAT-based parallel solver (equivalent to DCE with margin 0). The length of the period is defined by the number of conflicts, and fixed-length mode was used in this evaluation[8].
  - ManyGlucose: Glucose-syrup-based parallel solver implementing DCE that won 3rd place in the parallel track of the SAT Competition 2020. Two types of period length definitions are provided based on the number of literal accesses (denoted as lit) and the number of executions of each code block (denoted as blk). The former corresponds to the period definition in DPS. The latter was introduced to more accurately estimate period execution time and requires pre-training to determine the execution time of each code block, which is a very time-consuming task.

We used instances from SAT Race 2019, SAT Competition 2020, and 2021 (1200 in total) as benchmark set. Experiments were performed on a cluster with 68-core Intel Xeon Phi KNL (1.4 GHz) with a memory limit of 85GB[9]. The time limit was set at 5000 seconds. All parallel solvers were run with 64 threads.

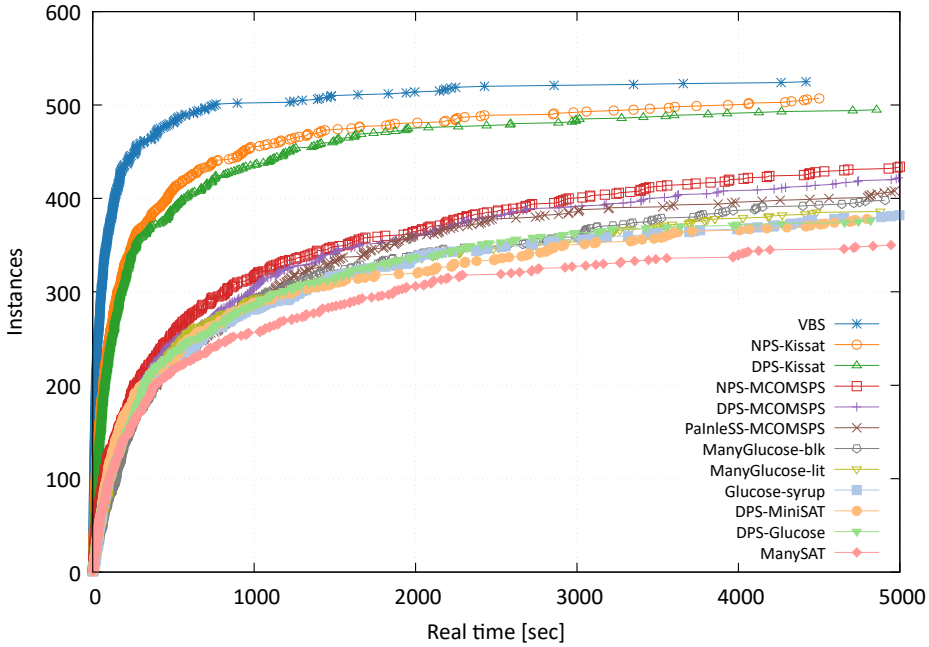
Table 1 shows the number of solved instances for each solver. The non-deterministic parallel solvers were run three times to show the variability, and conversely, DPS-Kissat was also run three times to show stability. The CDF (cumulative distribution function) of runtime is shown in Figure 3. Table 2 and Figure 4 present the waiting time ratio and the CDF plot, respectively.

In the following, we discuss each solver combination.

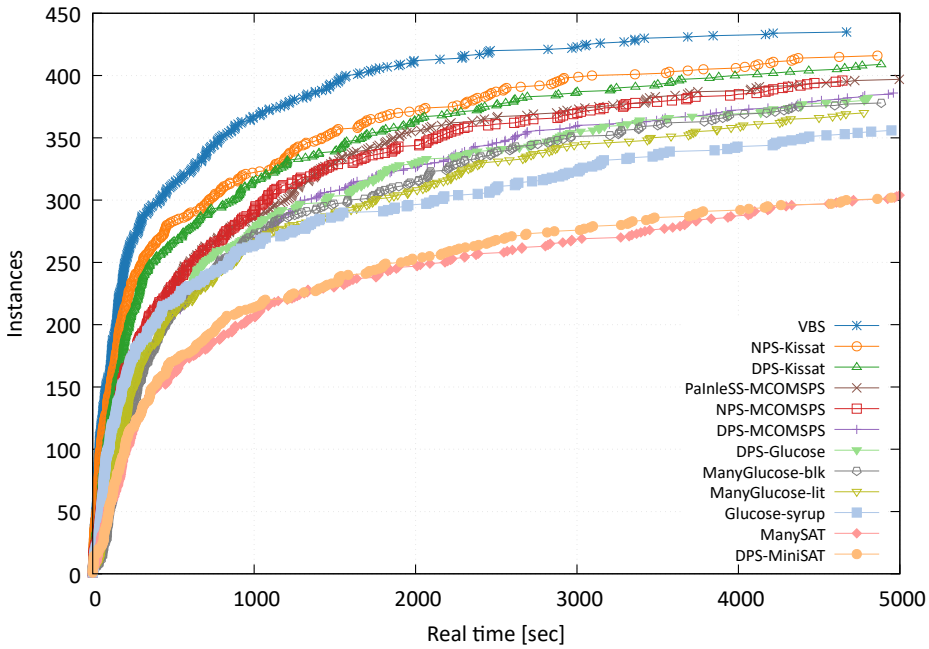
[7] If $m > 0$ in NPS, the import of clauses is delayed by m periods, even if the execution time of each worker's period is the same. Therefore, we set $m = 0$.

[8] In our evaluation, the fixed-length mode performed better than the variable-length mode.

[9] We used the supercomputer of ACCMS, Kyoto University.



(a) SAT instances



(b) UNSAT instances

Figure 3: Runtime CDF for solved instances. Solvers that have been run three times are shown only in the best case.

Table 1: Solved instances and PAR-2 scores. "X+Y" or "'Z (X+Y)" denotes the number of satisfiable instances (X) and unsatisfiable instances (Y) solved, and Z represents their sum. P-MCOMSPS stands for PaInleSS-MCOMSPS. Above the first double line is non-deterministic solvers and below is deterministic solvers.

Solver	# of solved instances				PAR-2
	2019	2020	2021	Total	
Glucose-syrup	148 + 98	104 + 112	130 + 146	738 (382 + 356)	5235522
	145 + 95	104 + 113	132 + 141	730 (381 + 349)	5286075
	145 + 96	109 + 112	130 + 136	728 (384 + 344)	5283988
P-MCOMSPS	156 + 107	118 + 124	134 + 166	805 (408 + 397)	4604576
	152 + 105	115 + 123	131 + **168**	794 (398 + 396)	4697998
	155 + 107	107 + 124	134 + 167	794 (396 + 398)	4707163
NPS-MCOMSPS	160 + 105	135 + 123	139 + **168**	830 (434 + 396)	4386010
	159 + 104	140 + 121	137 + **168**	829 (436 + 393)	4352294
	163 + 105	126 + 122	138 + 167	821 (427 + 394)	4451212
NPS-Kissat	**175** + 114	**178** + **134**	154 + **168**	**923** (**507** + 416)	**3245708**
	173 + 114	175 + **134**	155 + **168**	919 (503 + 416)	3266045
	170 + **115**	175 + **134**	155 + **168**	917 (500 + **417**)	3296766
ManySAT	136 + 79	95 + 106	119 + 119	654 (350 + 304)	6053978
ManyGlucose-lit	146 + 107	110 + 120	130 + 143	756 (386 + 370)	5084256
ManyGlucose-blk	149 + 108	117 + 121	132 + 149	776 (398 + 378)	4935944
DPS-MiniSAT	145 + 74	109 + 107	124 + 121	680 (378 + 302)	5783256
DPS-Glucose	142 + 103	104 + 122	130 + 157	758 (376 + 382)	5018397
DPS-MCOMSPS	156 + 101	129 + 119	137 + 166	808 (422 + 386)	4636427
DPS-Kissat	168 + 112	170 + 130	**157** + 167	904 (495 + 409)	3451073
	168 + 112	170 + 130	**157** + 167	904 (495 + 409)	3451074
	168 + 112	170 + 130	**157** + 167	904 (495 + 409)	3451445
VBS	180 + 120	185 + 139	159 + 176	959 (524 + 435)	2741798

- DPS-MiniSAT vs. ManySAT: both are based on MiniSAT, but the latency of DPS is greatly reduced by DCE, and the number of solved instances is improved. The large increase in SAT while UNSAT remained almost the same, in contrast to DPS-Glucose discussed below, may be due to the long restart interval and the absence of the LBD [1] criterion.
- DPS-Glucose vs. ManyGlucose vs. Glucose-syrup: both DPS and ManyGlucose outperform Glucose-syrup, which frequently locks the clause database shared by all workers because each worker requests exclusive access each time it imports or exports a clause. DCE does not need the exclusivity of Glucose-syrup because it maintains a dedicated clause database for each worker and each period. ManyGlucose-blk shows the best results among these solvers, but its implementation is very time-consuming. It requires detailed profiling to estimate the execution time of each process inside the solver. DPS-Glucose is strong on UNSAT and weak on SAT. This may be due to the use of the simple diversity strategy.

Table 2: Waiting time ratios in deterministic parallel solvers.

Solver	Waiting time ratio
ManySAT	41.1%
ManyGlucose-lit	21.7%
ManyGlucose-blk	10.3%
DPS-MiniSAT	10.4%
DPS-Glucose	9.9%
DPS-MCOMSPS	12.4%
DPS-Kissat	15.0%
	15.0%
	15.0%

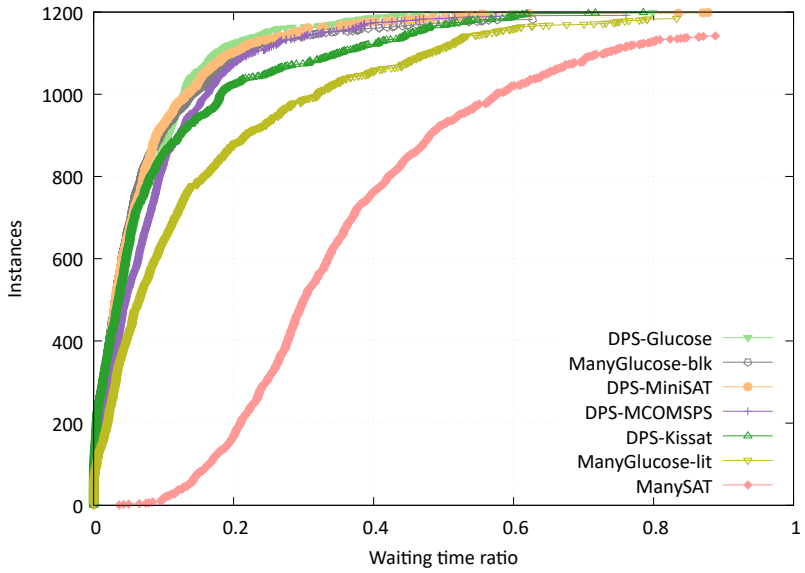
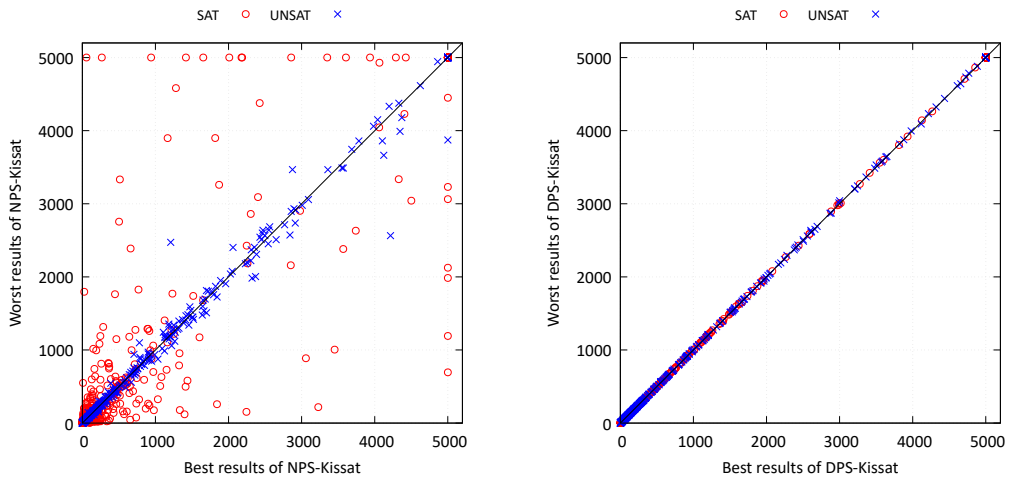


Figure 4: Comparison of waiting time ratio. The right end of some lines did not reach 1200 instances, which means that they could not be logged due to out of memory.

This simplicity is also effective in reducing latency because it suppresses the fluctuation of each worker's execution time. ManyGlucose-lit and DPS-Glucose are both based on the number of literal accesses. However, the latency of DPS-Glucose is about half that of ManyGlucose-lit, which is almost equal to ManyGlucose-blk.

- DPS-MCOMSPS vs. NPS-MCOMSPS vs. PaInleSS-MCOMSPS: NPS-MCOMSPS shows comparable results on UNSAT but superior results on SAT compared to PaInleSS-MCOMSPS. Since both solvers use almost the same search strategies, it may be influenced by the clause exchange intervals (every 0.5 seconds for PaInleSS and every period for DPS). This result indicates that our framework is capable of building efficient non-deterministic parallel solvers. DPS-MCOMSPS also shows comparable performance to PaInleSS. The difference between DPS and NPS represents the cost of ensuring reproducible behavior. The latency of DPS-MCOMSPS is greater than that of DPS-MiniSAT and DPS-Glucose. This is because



(a) NPS-Kissat (b) DPS-Kissat

Figure 5: Comparison of best and worst running times for NPS-Kissat and DPS-Kissat. Points on 5000 seconds mean they are solved only by either the best or worst case.

DPS-MCOMSPS uses a more diverse search strategy.

- DPS-Kissat vs. NPS-Kissat: NPS-Kissat showed the best results in our evaluation, followed by DPS-Kissat. The waiting time increased for DPS-Kissat compared to DPS-MCOMSPS. Kissat switches two different search strategies (stable and focus modes), furthermore, half of the workers in DPS-Kissat disable the elimination technique. This may have resulted in a greater variation in processing time among workers. Kissat is a SAT solver that has shown significant performance gains in satisfiable instances, which is also evident in parallelization. Figure 5 compares the best- and worst-case runtimes for these solvers. The figure shows that NPS is unstable, especially in satisfiable instances, while DPS-Kissat has no such instability. Figure 6 shows the best-case comparison of the both solvers: there are some instances that only DPS-Kissat was able to solve, but they are few. Overall, it can be seen that ensuring the reproducibility reduces the execution speed. This is caused by variations in period execution time for each worker, which is an inherent challenge for deterministic parallel SAT solvers.

6 Conclusion and Future Work

Making the behavior of parallel SAT solvers reproducible will facilitate the application of parallel solvers in practical fields and further promote research in parallel SAT solving. In this paper, we proposed DPS, a framework for building deterministic parallel SAT solvers that can incorporate state-of-the-art sequential SAT solvers with little effort and without requiring expertise in concurrent programming. Deterministic parallel SAT solvers based on DPS can achieve performance comparable to existing state-of-the-art non-deterministic parallel SAT solvers. When performance is more important than reproducibility, it can be run as non-deterministic parallel SAT solvers to achieve higher performance.

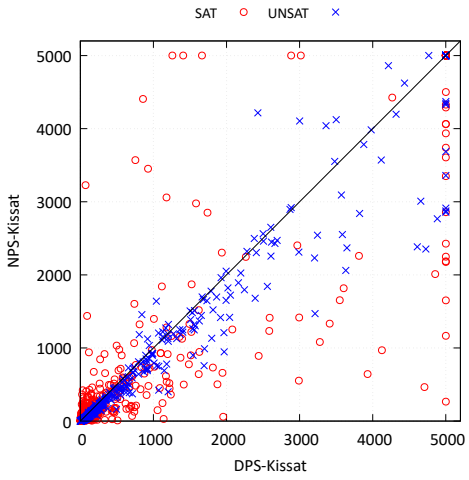


Figure 6: Comparison of NPS-Kissat and DPS-Kissat.

DPS is targeted at shared-memory environments, and extending it to non-shared-memory environments is one of the key challenges for the future. In such a large cluster environment, several different kinds of SAT solvers may be executed in parallel to enhance performance, and efficient clause exchange among heterogeneous solvers is also an important issue. In addition to SAT, deterministic parallel solving is required in extensions of SAT such as MAXSAT [14]. Building a DPS-like framework for these extensions is one of the future challenges.

Acknowledgment. This work was supported by JSPS KAKENHI Grant Numbers JP20H05963, JP20K11934. In this research work we used the supercomputer of ACCMS, Kyoto University.

References

[1] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of IJCAI-2009*, pages 399–404, 2009.

[2] Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel SAT solvers. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT 2014), LNCS 8561*, pages 197–205, 2014.

[3] Tomáš Balyo, Peter Sanders, and Carsten Sinz. HordeSat: A massively parallel portfolio SAT solver. In *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT 2015), LNCS 9340*, pages 156–172, 2015.

[4] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1999), LNCS 1579*, pages 193–207, 1999.

[5] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. http://hdl.handle.net/10138/318450, 2020. SAT Competition 2020 Solver Description.

[6] Armin Biere, Mathias Fleury, and Maximilian Heisinger. CaDiCaL, Kissat, Paracooba entering the SAT competition 2021. http://hdl.handle.net/10138/333647, 2021. SAT Competition 2021 Solver Description.

[7] James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI 1994)*, pages 1092–1097, 1994.

[8] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), LNCS 2919*, pages 502–518, 2003.

[9] Youssef Hamadi, Saïd Jabbour, Cédric Piette, and Lakhdar Sais. Deterministic parallel DPLL. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):127–132, 2011.

[10] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2009.

[11] Henry A. Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, *Proceedings of 10th European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363. John Wiley and Sons, 1992.

[12] Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. PaInleSS: A framework for parallel SAT solving. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT 2017), LNCS 10491*, pages 233–250, 2017.

[13] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT 2016), LNCS 9710*, pages 123–140, 2016.

[14] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Deterministic parallel MaxSAT solving. *International Journal on Artificial Intelligence Tools*, 24(3):1550005:1–1550005:25, 2015.

[15] Hidetomo Nabeshima and Katsumi Inoue. Reproducible efficient parallel SAT solving. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT 2020), LNCS 12178*, pages 123–138, 2020.

[16] Olivier Roussel. Description of ppfolio. https://www.cril.univ-artois.fr/~roussel/ppfolio/, 2011. SAT Competition 2011 Solver Description.

[17] Rodrigue Konan Tchinda and Clémentin Tayou Djamegni. hKis, hCaD, PaKis and PaInleSS_ExMapleLCMDistChronoBT in the SC21. http://hdl.handle.net/10138/333647, 2021. SAT Competition 2021 Solver Description.

[18] Hantao Zhang. Combinatorial designs by SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 533–568. IOS Press, 2009.

[19] 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.