Reproducible Efficient Parallel SAT Solving

Hidetomo Nabeshima[1(✉)] and Katsumi Inoue[2]

[1] University of Yamanashi, Kofu, Japan
nabesima@yamanashi.ac.jp
[2] National Institute of Informatics, Chiyoda, Japan
inoue@nii.ac.jp

Abstract. In this paper, we propose a new reproducible and efficient parallel SAT solving algorithm. Unlike sequential SAT solvers, most parallel solvers do not guarantee reproducible behavior due to maximizing the performance. The unstable and non-deterministic behavior of parallel SAT solvers hinders a wider adoption of parallel solvers to the practical applications. In order to achieve robust and efficient parallel SAT solving, we propose two techniques to significantly reduce idle time in deterministic parallel SAT solving: delayed clause exchange and accurate estimation of execution time of clause exchange interval between solvers. The experimental results show that our reproducible parallel SAT solver has comparable performance to non-deterministic parallel SAT solvers even in a many-core environment.

1 Introduction

Most modern computers have multiple cores, and the number of cores is increasing. To exploit the performance of multi-core systems, parallel processing software which efficiently utilizes each core is required. The same applies to SAT solvers, and parallel SAT solving is an active area of research. The parallel track of the SAT Competition is continuously held since 2011[1].

There are mainly two approaches of parallel SAT solving: portfolio and divide-and-conquer approaches. The former approach launches multiple SAT solvers with different search strategies in parallel, and each solver tries to solve the same SAT instance competitively [1,2,4]. The latter approach divide a given SAT instance in an attempt to distribute the total workload among computing units, and then solve them in parallel [5–7,10,12,13]. In both approaches, clause exchange techniques are combined into parallel systems in order to share the pruning information of the search space between solvers [1,2,4,11,12].

Most of parallel SAT solvers do not provide reproducible behavior in both runtime and found solutions due to maximizing the performance. Even for the same instance and computational environment, the execution time often varies for each run, and found models or unsatisfiability proofs may also differ. This is

[1] http://www.satcompetition.org/.

L. Pulina and M. Seidl (Eds.): SAT 2020, LNCS 12178, pp. 123–138, 2020.
https://doi.org/10.1007/978-3-030-51825-7_10

because there is no specific order in clause exchange between computing units. The timing of sending and receiving clauses can change due to system workload, cache misses and/or communication delays. The non-deterministic behavior of parallel SAT solvers causes various difficulties. In model checking, one may find different bugs (corresponding to satisfiable assignments) for each run. In the case of scheduling, even if a good solution is found, it may not be reproduced next time. If a bug occurs in software with an embedded non-deterministic SAT solver, the bug may not be reproduced. Researchers of parallel SAT solvers should have a number of experiments for stable evaluation of solvers. In contrast, most sequential SAT solvers guarantee reproducible behavior. The above-mentioned issues can be avoided if we use sequential SAT solvers. Reproducibility is thus an important property that directly affects the usability of SAT solvers as tools.

ManySAT 2.0 [3] is the first parallel SAT solver that supports reproducible behavior[2]. It is a portfolio parallel SAT solver for shared memory multi-core systems. To achieve deterministic behavior, it periodically synchronizes all threads, each of which executes a SAT solver, before and after the clause exchange. After the former synchronization, each solver exchanges clauses according to a specific order of threads until the latter synchronization. In ManySAT, all threads need to be synchronized periodically. Hence, waiting threads frequently occur as the number of CPU cores increases. As a result, there is a performance gap between deterministic and non-deterministic modes of ManySAT.

In this paper, we present two techniques to reduce the waiting time of threads: (1) delayed clause exchange and (2) refining the interval of clause exchange. The former suppresses the fluctuation of intervals between clause exchange, and the latter enables accurate prediction of exchange timing. We demonstrate that our approach significantly reduces the waiting time of threads and achieves the comparable performance with non-deterministic parallel SAT solvers even in a many-core environment[3].

The outline of this paper is as follows. The next section experimentally demonstrates the non-deterministic and unstable behavior of parallel SAT solvers. Section 3 describes the mechanism of ManySAT to realize the reproducibility and shows the experimental evaluation of the performance. In Sects. 4 and 5, we present two techniques in order to reduce the waiting time: delayed clause exchange and refining the interval of clause exchange, respectively. Experimental results are presented in Sect. 6. We conclude in Sect. 7.

2 Non-deterministic Behavior in Parallel SAT Solvers

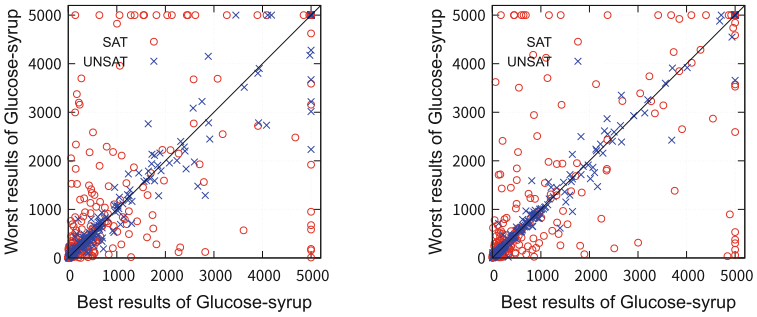
In this section, we reexamine the unreproducible behavior of existing parallel SAT solvers. Here we consider ManySAT and Glucose-syrup as such parallel solvers developed for shared memory multi-core systems. ManySAT is the first portfolio parallel SAT solver [4] developed as a non-deterministic parallel solver,

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

[3] The solver source code and experimental results (including colored graphs in this paper) are available at http://www.kki.yamanashi.ac.jp/~nabesima/sat2020/.

Table 1. Solved instances on SAT Competition 2018 and SAT Race 2019 (800 instances in total). "X (Y + Z)" denotes the number of solved instances (X), solved satisfiable instances (Y) and solved unsatisfiable instances (Z), respectively. Non-deterministic solvers (ManySAT with non-det and Glucose-syrup) were run three times, and the first and last lines of the results denote the best and worst results, respectively.

Solver	# of solved instances	
	4 threads	64 threads
ManySAT 2.0 with non-det	434 (265 + 169)	475 (292 + 183)
	425 (265 + 160)	475 (294 + 181)
	420 (257 + 163)	473 (288 + 185)
ManySAT 2.0 with det-static	414 (251 + 163)	457 (284 + 173)
ManySAT 2.0 with det-dynamic	418 (258 + 160)	448 (281 + 167)
Glucose-syrup 4.1	465 (263 + 202)	524 (301 + 223)
	462 (263 + 199)	519 (295 + 224)
	458 (255 + 203)	515 (293 + 222)



(a) 4 threads (b) 64 threads

Fig. 1. Comparison of runtimes of the best and worst results of Glucose-syrup for each instance. Satisfiable instances are denoted with ∘, unsatisfiable instances with ×. Points on 5000 s mean that these instances are solved only by either the best or worst cases.

and ManySAT 2.0 supports both deterministic and non-deterministic strategies. Glucose-syrup [1] is one of the state-of-the-art parallel portfolio solvers.

We have run ManySAT 2.0 and Glucose-syrup 4.1 on instances from SAT Competition 2018 and SAT Race 2019, and show the experimental results as the numbers of solved instances in Table 1. In this work, we conducted all experiments on the following two computing environments: (1) a cluster equipped with 4-core Intel Core i5-6600 (3.3 GHz) machines using a memory limit of 8 GB, and (2) a cluster equipped with 68-core Intel Xeon Phi KNL (1.4 GHz) machines using a memory limit of 96 GB[4]. The time limit was set to 5000 s. We ran each

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

solver with 4 threads on the first environment and 64 threads on the second. "ManySAT 2.0 with non-det" denotes the non-deterministic mode of ManySAT, and "det-static" and "det-dynamic" mean the two deterministic modes described in the next section.

The results of non-deterministic solvers show that different runs yield different numbers of solved instances. For Glucose-syrup, the difference between the best and worst results is 7 and 9 instances on the 4 and 64 threads environments, respectively. Figure 1 gives scatter plots comparing the runtimes for each instance in which we compared the best and worst results of Glucose-syrup. The runtimes of satisfiable instances vary greatly with runs. Unsatisfiable instances have more stable results but there are some instances solved by either one. Such behavior is typically encountered when using parallel SAT solvers.

Clause exchange is a cooperative and fundamental mechanism in parallel SAT solvers in order to share the pruning information of the search space between computing units. Typically, the timing of sending and receiving clauses is affected by system workload, cache misses, and/or communication delays. However, most of parallel SAT solvers do not have synchronization mechanism of the timing in order to avoid the communication overhead and to maximize the performance. This is the cause of unreproducible behavior of parallel SAT solvers.

3 A Deterministic Parallel SAT Solver

In this section, we describe the algorithm called deterministic parallel DPLL ($(DP)^2LL$ in short) proposed by [3] which is implemented in the first deterministic parallel SAT solver ManySAT 2.0. The pseudo code is shown in Algorithm 1. Let n be the number of solvers to be executed in parallel. Firstly, n threads are launched to execute the function *search*() (lines 2 and 4). Each thread is identified by an ID number $t \in \{1, \ldots, n\}$. After all threads have finished (line 5), the algorithm outputs the solution obtained by the thread with the lowest thread ID among all the threads which succeeded to decide the satisfiability of the instance (line 6). The reason for choosing the lowest ID is to avoid non-deterministic behavior if two or more threads find solutions at the same time.

The function *search*() (lines 8–25) is the same as usual CDCL solvers, except for sending and receiving clauses. Each thread periodically receives clauses from the other threads. We call the reception interval a *period*. The function *endOfPeriod*() decides whether the current period has ended (line 10). In ManySAT, it returns true when the number of conflicts in the period exceeds a certain threshold. In that case, all threads are synchronized *before* and *after* clause exchange by "< barrier >" instruction[5] (lines 11 and 14). The former barrier is necessary for each thread to start importing clauses simultaneously. Suppose that a thread starts importing clauses at the end of period x. The latter barrier prevents the thread importing a clause which is exported from another thread at the next period $x+1$. In order to avoid deadlocks, when a thread finds

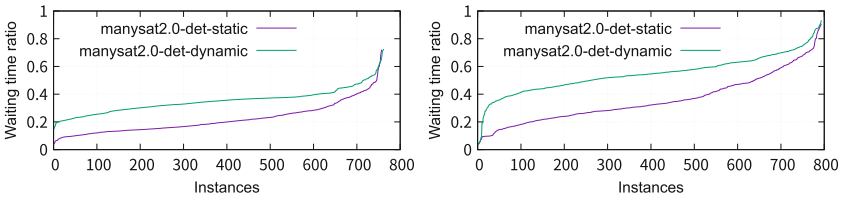
[5] The barrier is implemented by `#pragma omp barrier` directive in OpenMP.

Algorithm 1: Deterministic Parallel DPLL [3]

```
1  Function solve(n)                                  // n is the number of threads
2      foreach t ∈ {1, ··· , n} do
3          ans_t ← unknown;
4          launch thread t which executes ans_t ← search(t);
5      wait for all threads to finish;
6      t_min ← min {t | ans_t ≠ unknown};
7      return ans_{t_min};
8  Function search(t)
9      loop
10         if endOfPeriod() = true then
11             nextPeriod: <barrier>;
12             if ∃i(ans_i ≠ unknown) then return ans_t;
13             importExtraClauses(t);
14             <barrier>;
15         if propagate() = false then
16             if noDecision() = true then
17                 ans_t = unsat;
18                 goto nextPeriod;
19             learnt ← analyze();
20             exportExtraClause(learnt);
21             backtrack();
22         else
23             if decide() = false then
24                 ans_t = sat;
25                 goto nextPeriod;
26 Procedure importExtraClauses(t)
27     foreach i ∈ ⟨1, ··· , t − 1, t + 1, ··· , n⟩ do
28         import clauses from thread i;
```

a solution (lines 18 and 25), it needs to go to the first barrier on line 11 instead of exiting immediately. This is because other threads that have not found a solution are waiting there. After synchronization, each thread t exits with its own status ans_t, if any thread finds a solution (line 12). The function *importExtraClauses*() receives learnt clauses acquired by the other threads according to a fixed order of the threads (line 27), because different ordering of clauses will trigger off different ordering of unit propagations and consequently different behavior.

The rest of the search function follows CDCL algorithm. The *propagate*() function (line 15) applies unit propagation (or Boolean constraint propagation) and returns *false* if a conflict occurs, and *true* otherwise. In the former case, if the conflict occurs without any decision (line 16), it means the unsatisfiability is proved. Otherwise, a cause of the conflict is analyzed (line 19) and a clause is learnt to prevent occurring the same conflict. If the learnt clause is eligible for export (for example, the length is short), it is marked to export. These exported clauses are periodically imported by the function *importExtraClauses*(). In the latter case, the function *decide*() chooses an unassigned variable as the next



(a) 4 threads (b) 64 threads

Fig. 2. Waiting time ratio of ManySAT with static and dynamic periods. The ratio is defined for each instance as the total waiting time of all threads divided by the total solving time of all threads. Results are sorted by the ratio.

Table 2. Ratio of waiting time to the total solving time of all threads for all instances.

Solver	4 threads	64 threads
ManySAT 2.0 with det-static	23.0%	40.6%
ManySAT 2.0 with det-dynamic	34.5%	56.6%

decision and assigns it *true* or *false* (line 23). Otherwise it returns *false* as all the variable are assigned, that is, a model is found.

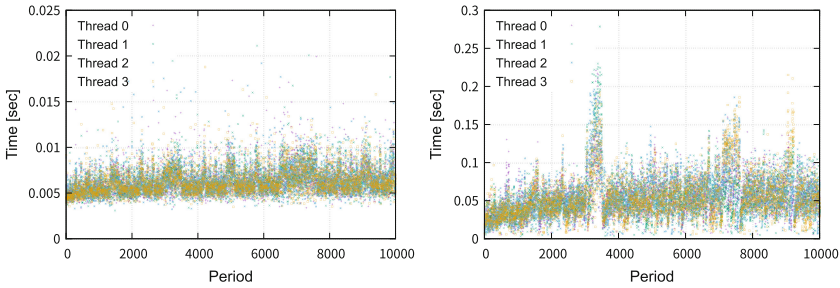
This $(DP)^2LL$ algorithm periodically requires synchronization with all threads. Runtime variation of periods on each thread causes idle time for each synchronization, that is, each thread should wait the slowest threads. A simple way to suppress the variation is to measure the execution time of threads and synchronize based on the elapsed time. However, the measurement of CPU or real time usually contains errors, so this approach is hard to hold reproducibility. In ManySAT, the length of a period is defined as the number of conflicts. There are two kinds of definitions of the period: *static* and *dynamic*. The *static period* is simply defined as a fixed number c of conflicts ($c = 700$ in default). The *dynamic period* is intended to provide a better approximation of progression speed of each solver. Let L_t^k denote the length of the k-th period of a thread t, defined as $L_t^k = c + c(1 - r_t^{k-1})$, where r_t^{k-1} is the ratio of the number of learnt clauses held by the thread t to the maximum number of learnt clauses among all threads at the $(k-1)$-th period. In this modeling, a thread with a large (or small) number of learnt clauses (the ratio tending to 1 (or 0)) is considered to be slow (or fast) and the length of period becomes shorter (or longer).

In Table 1, "det-static" and "det-dynamic" denote the results of the static and dynamic periods, respectively. There is a performance gap between deterministic and non-deterministic solvers. The cause is high waiting time ratio to the running time. Figure 2 shows the waiting time ratio of ManySAT for each instance, and the ratio on all instances is shown in Table 2. In our experiments, the waiting time ratio of the static period is lower than dynamic, but it reaches 23% for 4 threads and 40% for 64 threads environments. These results indicate that it is

difficult to realize efficient solving by synchronizing all threads in a many-core environment.

4 Delayed Clause Exchange

In order to reduce the idle time in deterministic parallel SAT solvers, we propose a new clause exchange schema called *delayed clause exchange* (DCE). Figure 3 shows the runtime distribution of the static periods in ManySAT for two instances. These results indicate that the execution time of periods fluctuates very frequently, but in the long term it seems to be stable (roughly, 0.005 s for (a), 0.05 s for (b)). Other instances have a similar tendency. In order to take advantage of this property and absorb frequent fluctuations, we consider allowing clause reception to be delayed for a certain number of periods.



(a) `mchess_18` (from SAT Comp 2018) (b) `58bits_13` (from SAT Race 2019)

Fig. 3. Distribution of period execution time (up to 10000 periods) on two instances by ManySAT 2.0 with det-static on the 4 threads environment. The results are sampled to half. The x-axis and y-axis show the number of periods and the execution time of the period, respectively. Each color represents different threads.

Let n be the number of threads, $T = \{1, \ldots, n\}$ the set of thread IDs, p_t the current period ID of a thread $t \in T$ ($p_t \geq 1$), E_t^p a set of clauses exported by a thread t at a period p and m an admissible delay, called *margin*, is denoted by the number of periods ($m \geq 0$). Algorithm 2 shows the pseudo code of $(\mathrm{DP})^2$LL with DCE. There are two differences from Algorithm 1. The first point is clause reception. For each thread t, when the current period p_t ends, then the thread increments the current period ID (line 14) and imports clauses from the other threads (line 15). If another thread i has not yet finished the period $p_t - m$ (that is, $p_i < p_t - m$), then the thread t should wait for it to complete (line 25). After that, the thread t imports $E_i^{p_t - m}$. The second point concerns termination conditions. When multiple threads find solutions in DCE, to keep reproducibility, the algorithm select a thread that found at the earliest period. In case of a tie, the thread with the lowest ID is selected (line 7). Running threads that have

Algorithm 2: $(\mathrm{DP})^2$LL with Delayed Clause Exchange

```
1  Function solve(n)                                   // n is the number of threads
2      ans ← unknown; p_min ← ∞; t_min ← ∞;
3      foreach t ∈ {1, ··· , n} do
4          p_t ← 1 ;                          // p_t is the number of periods in thread t
5          launch thread t which executes the followings:
6              ans_t ← search(t);
7              if ans_t ≠ unknown and (p_t < p_min or (p_t = p_min and t < t_min))
                then ans ← ans_t; p_min ← p_t; t_min ← t ;
8      wait for all threads to finish;
9      return ans;
10 Function search(t)
11     loop
12         if endOfPeriod() = true then
13             if ans ≠ unknown and p_min < p_t then return unknown;
14             p_t ← p_t + 1;
15             importExtraClauses(t);
16         if propagate() = false then
17             if noDecision() = true then return unsat;
18             learnt ← analyze();
19             exportExtraClause(learnt);
20             backtrack();
21         else
22             if decide() = false then return sat;
23 Function importExtraClauses(t)
24     foreach i ∈ ⟨1, ··· , t − 1, t + 1, ··· , n⟩ do
25         wait until p_i ≥ p_t − m ;  // synchronization between thread t and i
26         import clauses from E_i^{p_t − m};
```

not yet found a solution can be terminated if their periods exceed $p_{\min}$ (line 13). Note that when $m = 0$, $(\mathrm{DP})^2$LL with DCE is same as $(\mathrm{DP})^2$LL.

DCE can reduce the total waiting time of threads. Firstly, we consider the best case of DCE. At some point, if for any two threads $i, j \in T$ $(i \neq j)$ the difference $p_i - p_j$ is less than or equal to m, then any thread can import clauses immediately without waiting for other threads at the point. This is because a set of clauses to be imported had already been exported by other threads. Secondly, the worst case is that only one thread is extremely slow and all other threads are ahead by m periods. In this case, the fast $|T| - 1$ threads must wait the slowest thread until the difference less than or equal to m. In other cases, if there exists two threads $i, j \in T$ such that $p_i - p_j > m$, then the preceding thread i should wait the postdating thread j until the difference less than or equal to m. The execution time of periods fluctuates frequently, but if the total execution time of m consecutive periods is almost the same for each thread, the DCE can be expected to reduce the waiting time. The disadvantage of DCE is that the clause reception is always delayed by m periods, even if there is no difference in the period of each thread.

5 Refining Periods

In ManySAT, the length of a period is defined as the number of conflicts. The generation speed of conflicts is affected by the number and length of clauses. The number of clauses varies during search by learning and reduction of clauses, and the length of learnt clauses also changes sometimes significantly. As the result, runtime of a period fluctuates frequently as shown in Fig. 3. Accurate estimation of period execution time is important to reduce the waiting time. In this section, we introduce two new definitions of a period based on reproducible properties.

5.1 Refinement Based on Literal Accesses

Most of the memory used by SAT solvers is occupied by literals in clauses. Accessing literals in memory is a fundamental operation and occurs very frequently in unit propagation, conflict analysis, and so on. We consider defining the length of a period as the number of accesses to literals. The speed of accessing literals can be considered to be more stable than the generation speed of conflicts since it is almost independent of the number and length of clauses. With this definition, the function *endOfPeriod*() (line 12 in Algorithm 2) returns true if the number of literal accesses in the period exceeds a certain threshold. In our implementation, we count the number of accesses to literals in unit propagation, conflict analysis and removal of clauses that are satisfied without any decision.

5.2 Refinement Based on Block Executions

In order to estimate the runtime of a period more accurately, we consider measuring not only the number of literal accesses, but also the number of executions of various operations performed by a SAT solver. It is similar to profiling a program which measures the number of calls and runtime of each function to detect performance bottlenecks. As a finer granularity than functions, we focus on compound statements called *blocks* (statements enclosed in curly braces in C++) and measure the number of the executions of each block during the search. For example, the runtime of one call of *propagate*() (line 16 in Algorithm 2) depends obviously on a given instance (proportional to the number of clauses). Whereas *propagate*() has a loop block that checks the value of each literal in a clause to determine whether the clause is unit or falsified. The time to execute the block once can be considered almost constant. We apply linear regression analysis to estimate the time required for one execution of each block.

Let n be the number of blocks to be measured, $x_k^{i,j}$ the number of executions of a block k of a thread j in an instance i, d_k the runtime to be required for one execution of a block k, and $y^{i,j}$ the execution time of a thread j in an instance i without waiting time. Each d_k is non-negative. Hence, if a block k has a nested block l, d_k indicates the execution time of k excluding l. Then, $y^{i,j}$ can be expressed as:

$$y^{i,j} = d_1 x_1^{i,j} + d_2 x_2^{i,j} + \cdots + d_n x_n^{i,j}. \tag{1}$$

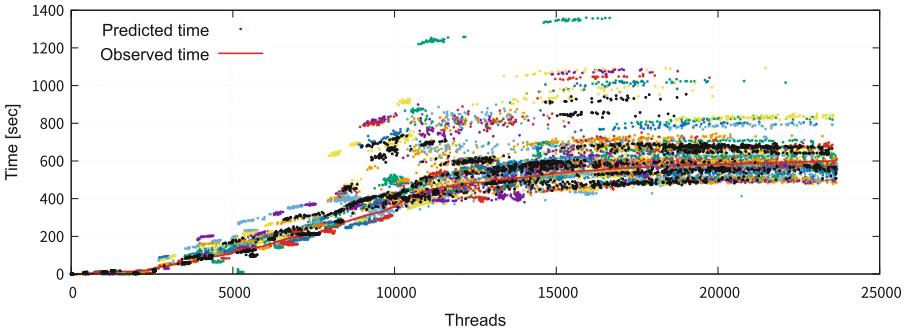


Fig. 4. Results of prediction of thread execution time on SAT Competition 2018. The solid line denotes the observed execution time for each thread and each instance (sorted by the execution time). Each point indicates the estimated execution time for each thread, and each color represents different instances.

With this definition, the function *endOfPeriod*() returns true if the value of (1) in a period exceeds a certain threshold.

Our developed deterministic parallel SAT solver (introduced in the next section) counts the number of executions for 71 blocks that almost cover the whole of operations performed by the solver. We used the application instances of SAT Competition 2016 and 2017 (300 and 350 instances, respectively) as training instances to estimate each d_k. The evaluation was performed on the 64 threads environment with a time limit of 600 s. To avoid overfitting, we manually selected 29 out of 71 blocks, which are mainly contained in unit propagation, conflict analysis and search loop (corresponding to *search*() function in Algorithm 2). Then, we estimated the regression coefficients d_k from the results using the Elastic Nets method [14]. The coefficient of determination (R^2) was 0.94.

Figure 4 shows the results of prediction of thread execution time on SAT Competition 2018 as testing instances. Most of estimated results are close to the observed one. Some points are far from observations, but such points of the same color are often equidistant from observations. This means that the difference between the predicted and the observed time is approximately the same for each thread that solves the same instance, and in such cases, synchronization between threads can be expected to have less idle time.

6 Experimental Results

We have developed a new deterministic parallel SAT solver called ManyGlucose based on Glucose-syrup 4.1, which implements the delayed clause exchange and three types of periods (one is conflict based period used in ManySAT and the others are described in the previous section). In this work, we set the margin to a fixed value of 20 and adjust the length of the three types of periods. Suppose that p_{conf}, p_{accs} and p_{exec} denote the length of a period based on the number of

Table 3. Results of three types of periods. In this evaluation, we executed ManyGlucose with margin 20 for 400 instances used in SAT Competition 2018 for the 4 threads environment. The best result of each column in (a) is typeset in boldface.

(a) The numbers of solved instances

p_{conf}	# of solved	p_{accs}	# of solved	p_{exec}	# of solved
50	231 (**128** + 103)	1M	231 (126 + **105**)	0.2	230 (125 + **105**)
100	**232** (**128** + 104)	**2M**	**236** (**132** + 104)	0.3	232 (127 + **105**)
200	229 (126 + 103)	4M	231 (128 + 103)	0.4	233 (129 + 104)
300	226 (121 + **105**)	6M	225 (121 + 104)	**0.5**	**236** (**132** + 104)
400	225 (122 + 103)	8M	230 (126 + 104)	0.6	231 (126 + **105**)

(b) The ratio of waiting time and average runtime per period

p_{conf}	Wait time ratio	Avg time/period	p_{accs}	Wait time ratio	Avg time/period	p_{exec}	Wait time ratio	Avg time/period
50	16.9%	0.044	1M	9.5%	0.029	0.2	6.0%	0.043
100	15.6%	0.087	2M	8.8%	0.056	0.3	5.6%	0.065
200	13.9%	0.172	4M	8.0%	0.112	0.4	5.3%	0.087
300	13.1%	0.263	6M	7.5%	0.167	0.5	5.1%	0.109
400	12.6%	0.350	8M	7.2%	0.219	0.6	4.8%	0.131

conflicts, literal accesses and block executions (corresponding to the threshold in the function *endOfPeriod*()). We determine the appropriate length of each period by preliminary experiments.

Table 3 shows the results of three types of periods. As the length of period becomes longer, the waiting time is reduced since the number of clause exchanges is diminished, but the number of solved instances also tends to decreases. There is a trade-off between the number of clause exchanges and solved instances. From these results, we determined the appropriate length for each period type to be $p_{conf} = 100, p_{accs} = 2M$, and $p_{exec} = 0.5$. Table 3 (b) denotes that the period based on the block executions has the smallest waiting time ratio. The average runtime of periods when $p_{exec} = 0.5$ is 0.109 s. Hence, with this setting, time delay to receive learnt clauses acquired by other threads is about 2 s ($0.109 * 20$).

Figure 5 shows the runtime distribution of periods based on these thresholds for some instances. These are results of unsolved instances by ManyGlucose without DCE within a 1000 s time limit (that is, the right end of x-axis corresponds to 1000 s). The execution time per period is normalized by the z-score to compare three period types. These graphs show that the period based on conflicts has large amplitude, and the block executions has small amplitude. For most instances, the block executions shows the best results, but Fig. 5 (c) is an example in which the literal access shows the best results.

We ran ManyGlucose configured with three types of periods and with and without DCE for the application instances used in SAT Competition 2018 and SAT Race 2019 in the 4 and 64 threads environments using the parameters obtained in the preliminary experiment (that is, $p_{conf} = 100, p_{accs} = 2M$, and $p_{exec} = 0.5$). ManyGlucose with DCE and block executions were run three times to show the robustness of our deterministic parallel SAT solver. Table 4 shows the number of solved instances and waiting time ratio for each solver and Fig. 6

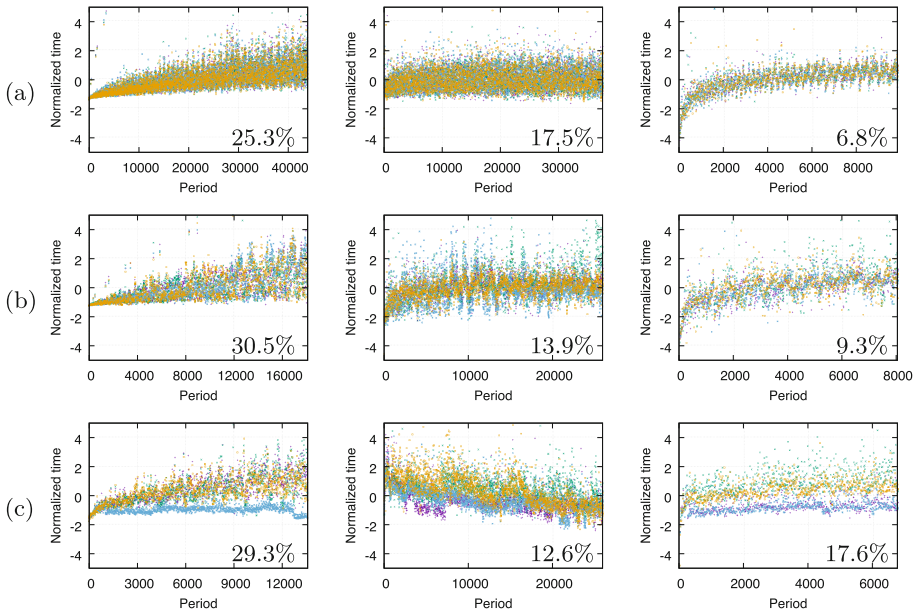


Fig. 5. Distribution of period execution time on (a) `mchess_18`, (b) `58bits_13` and (c) `eqspwtcl14bpwtcl14` (from SAT Race 2019). The results are sampled to 10%. The value at the bottom right of each graph shows the waiting time ratio.

and 7 are cactus plots of them. DCE can reduce the waiting time and increase the number of solved instances. The effect is remarkable in the 64 threads environment. ManyGlucose with DCE can solve 50, 36 and 34 more instances in conflict, literal access and block execution based periods than without DCE, respectively. For the ratio of waiting time, DCE reduces approximately 24%, 32% and 35% in conflict, literal access and block execution based periods than without DCE, respectively. The ratio of waiting time in 64 threads is greater than in 4 threads. When using block execution based period, it increases about 3.5 times without DCE (12.5% to 44.0%), but 1.5 times with DCE (5.7% to 8.8%). This means DCE is more effective in many-core systems. The regression coefficients of block execution based period are estimated in the 64 threads environment. Figure 7(a) shows that it is effective for reducing the waiting time even in the different environment. The difference of average runtime per period between 4 and 64 threads denotes the performance gap of sequential computation of each system. The 64 threads environment has a large number of CPU cores, although the sequential computing performance is not high. In the 64 threads environment, time delay to receive learnt clauses acquired by other threads is about 8 to 12 s.

Compared with Glucose-syrup, ManyGlucose shows the stable results due to its determinism. In the results of running ManyGlucose with DCE and block execution based period three times, the difference between the best and worst

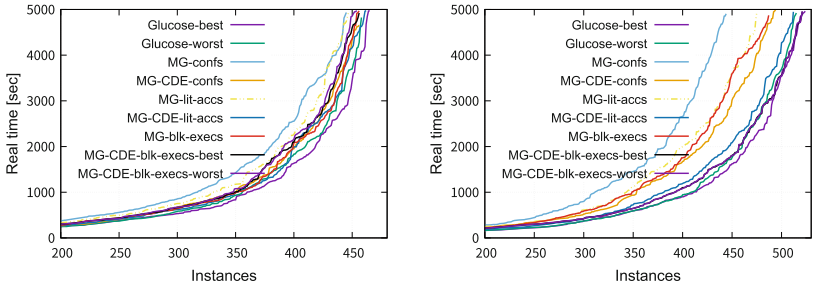
Table 4. Results of (a) solved instances and (b) waiting time ratio on SAT Competition 2018 and SAT Race 2019. Results of Glucose-syrup are same as Table 1. The best result of each column in (a) is typeset in boldface. "confs", "lit accs" and "blk execs" mean the period type based on conflicts, literal accesses and block executions, respectively.

(a) The numbers of solved instances

Solver	# of solved instances	
	4 threads	64 threads
Glucose-syrup 4.1	**465** (**263** + 202)	**524** (**301** + 223)
	462 (**263** + 199)	519 (295 + 224)
	458 (255 + **203**)	515 (293 + 222)
ManyGlucose + confs	445 (250 + 195)	444 (254 + 190)
ManyGlucose + DCE + confs	456 (262 + 194)	494 (283 + 211)
ManyGlucose + lit accs	447 (252 + 195)	476 (272 + 204)
ManyGlucose + DCE + lit accs	462 (265 + 197)	512 (291 + 221)
ManyGlucose + blk execs	456 (259 + 197)	487 (275 + 212)
ManyGlucose + DCE + blk execs	456 (258 + 198)	521 (293 + **228**)
	455 (258 + 197)	521 (293 + **228**)
	454 (258 + 196)	521 (293 + **228**)

(b) The ratio of waiting time and average runtime per period

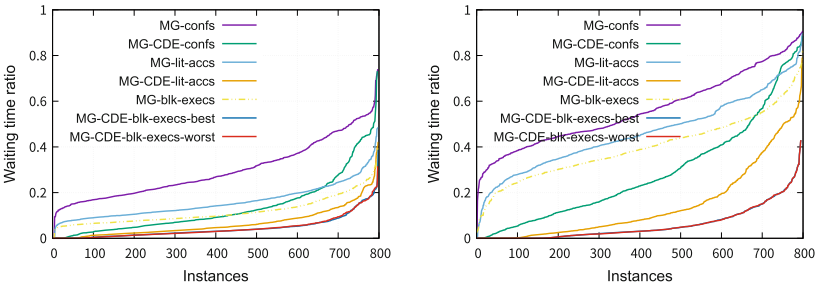
Solver	4 threads		64 threads	
	Waiting time ratio	Avg time/period	Waiting time ratio	Avg time/period
ManyGlucose + confs	29.7%	0.099	58.3%	0.528
ManyGlucose + DCE + confs	14.8%	0.104	34.4%	0.392
ManyGlucose + lit accs	17.9%	0.058	51.8%	0.398
ManyGlucose + DCE + lit accs	8.9%	0.060	20.2%	0.366
ManyGlucose + blk execs	12.5%	0.122	44.0%	0.621
ManyGlucose + DCE + blk execs	5.7%	0.126	8.8%	0.594
	5.6%	0.125	8.8%	0.593
	5.6%	0.126	8.8%	0.594



(a) 4 threads (b) 64 threads

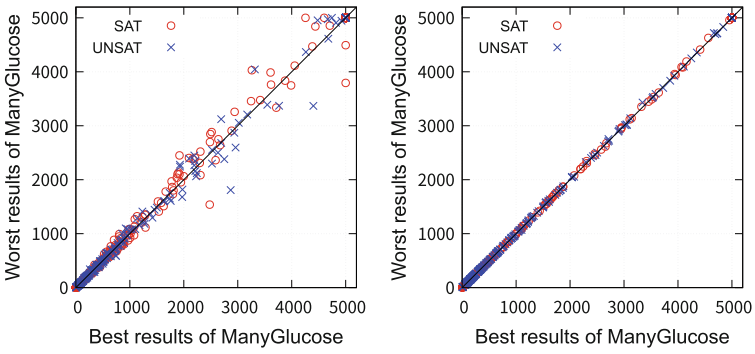
Fig. 6. Cactus plot comparing total instances solved within a given time bound for Glucose-syrup and ManyGlucose configured with three types of periods and with/without DCE. MG means ManyGlucose. The best and worst results of MG-DCE-blk-execs are almost overlapped.

results is 2 instances in the 4 threads and no difference in the 64 threads environment. We have confirmed that ManyGlucose can find the same model for each run for satisfiable instances. Our 4 threads environment is a cluster built on educational PCs and cannot be used exclusively, and the results fluctuate slightly. In contrast, the results of 64 threads are very stable due to the exclusive use of the system. Figure 8 shows comparisons of runtime of each instance in the best and worst results. The results for 4 threads vary slightly over time, while the results for 64 threads are almost completely distributed on the diagonal. In contrast to



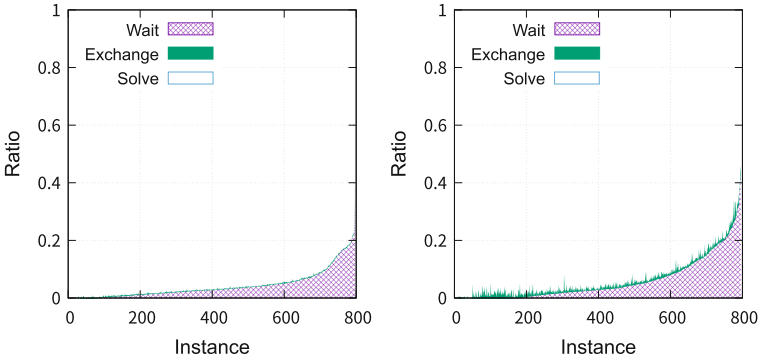
(a) 4 threads (b) 64 threads

Fig. 7. Waiting time ratio of ManyGlucose. Results are sorted by the ratio.



(a) 4 threads (b) 64 threads

Fig. 8. Comparison of runtimes of the best and worst results of ManyGlucose for each instance.



(a) 4 threads (b) 64 threads

Fig. 9. Waiting time and clause exchanging time of ManyGlucose.

Fig. 1, this demonstrates the robustness of our deterministic parallel SAT solver. The number of solved instances in the best case of ManyGlucose exceeds the worst case of Glucose-syrup. This indicates that DCE and accurate period estimation can achieve performance comparable to non-deterministic solvers while holding deterministic behavior.

Figure 9 shows the time ratio required for clause exchange, which is very small ratio compared to the solving and waiting time. In DCE, for each thread t and each period p, there is a database E_t^p that stores clauses exported by the thread t at the period p. If the current period of the thread t is greater than p, then write access to E_t^p no longer exists, so any thread can read it without mutual exclusive control. In contrast, shared clause databases in non-deterministic parallel SAT solvers usually have a mixture of write access to add clauses and read access to get clauses. Hence, the mutual exclusive control is required to access the clause database. One of the advantages of DCE is that it does not require the cost of mutual exclusive control to access clause databases.

7 Conclusion

The non-deterministic behavior of parallel SAT solvers is one of the obstacles to the promotion of application and research of parallel SAT solvers. In this paper, we have presented techniques to realize efficient and reproducible parallel SAT solving. The main technique is the delayed clause exchange (DCE), which absorbs fluctuations of intervals between clause exchanges. In order to enhance the effect of DCE, it is important to estimate exchange intervals accurately based on reproducible criterion. In this work, we presented two methods based on the number of literal accesses and block executions. The experimental results show that the combination of these techniques can achieve comparable performance to non-deterministic parallel SAT solvers even in many-core environments. Our approach can be applicable to deterministic parallel MaxSAT solving [9] which is based on the synchronization mechanism used in ManySAT. As future work it would be interesting to consider a general framework for building deterministic parallel SAT solvers (like PaInleSS [8] for non-deterministic parallel SAT solvers) in which state-of-the-art sequential solvers can easily participate.

Acknowledgment. This work was supported by JSPS KAKENHI Grant Number JP17K00300 and JP20K11934. In this research work we used the supercomputer of ACCMS, Kyoto University.

References

1. Audemard, G., Simon, L.: Lazy clause exchange policy for parallel SAT solvers. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 197–205. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_15
2. Biere, A.: Lingeling, Plingeling and Treengeling entering the SAT Competition 2013. In: SAT Competition 2013: Solver and Benchmark Descriptions, pp. 51–52 (2013)

3. Hamadi, Y., Jabbour, S., Piette, C., Sais, L.: Deterministic parallel DPLL. J. Satisf. Boolean Model. Comput. **7**(4), 127–132 (2011)
4. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. J. Satisf. Boolean Model. Comput. **6**(4), 245–262 (2009)
5. Heule, M.J.H., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: guiding CDCL SAT solvers by lookaheads. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC 2011. LNCS, vol. 7261, pp. 50–65. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34188-5_8
6. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: A distribution method for solving SAT in grids. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 430–435. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_39
7. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Partitioning SAT instances for distributed solving. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR 2010. LNCS, vol. 6397, pp. 372–386. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16242-8_27
8. Le Frioux, L., Baarir, S., Sopena, J., Kordon, F.: PaInleSS: a framework for parallel SAT solving. In: Gaspers, S., Walsh, T. (eds.) SAT 2017. LNCS, vol. 10491, pp. 233–250. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_15
9. Martins, R., Manquinho, V.M., Lynce, I.: Deterministic parallel MaxSAT solving. Int. J. Artif. Intell. Tools **24**(3), 1550005:1–1550005:25 (2015)
10. Plaza, S., Markov, I., Bertacco, V.: Low-latency SAT solving on multicore processors with priority scheduling and XOR partitioning. In: Proceedings of the 17th International Workshop on Logic and Synthesis (2008)
11. Schubert, T., Lewis, M.D.T., Becker, B.: PaMira - A parallel SAT solver with knowledge sharing. In: Proceedings of Sixth International Workshop on Microprocessor Test and Verification (MTV 2005), Common Challenges and Solutions, pp. 29–36 (2005)
12. Sinz, C., Blochinger, W., Küchlin, W.: PaSAT- parallel SAT-checking with lemma exchange: implementation and applications. Electron. Notes Disc. Math. **9**, 205–216 (2001)
13. Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. J. Symb. Comput. **21**(4), 543–560 (1996)
14. Zou, H., Hastie, T.: Regularization and variable selection via the elastic net. J. Roy. Stat. Soc. B **67**, 301–320 (2005)