IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 9, 2019, accepted October 27, 2019, date of publication November 1, 2019, date of current version December 3, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2951027

A Novel Multi-Thread Parallel Constraint Propagation Scheme

ZHE LI[1,2], ZHEZHOU YU[1,2], PENG WU[3], JIANAN CHEN[1,2], AND ZHANSHAN LI[1,2]

[1]Key Laboratory for Symbol Computation and Knowledge Engineering of National Education Ministry, Changchun 130012, China
[2]College of Computer Science and Technology, Jilin University, Changchun 130012, China
[3]Key Laboratory of Space Utilization, Technology and Engineering Center for space Utilization, Chinese Academy of Sciences, Beijing 100094, China

Corresponding author: Zhanshan Li (lizs@jlu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61672261 and Grant 61802056, in part by the Natural Science Foundation of Jilin Province under Grant 20180101043JC, in part by the Industrial Technology Research and Development Project of Jilin Development and Reform Commission under Grant 2019C053-9, in part by the Jilin Provincial Key Laboratory of Big Data Intelligent Computing under Grant 20180622002JC, and in part by the Open Research Fund of Key Laboratory of Space Utilization, Chinese Academy of Sciences, under Grant LSU-KFJJ-2019-08.

ABSTRACT Constraint Programming (CP) is an efficient technique for solving combinatorial (optimization) problems. In modern constraint solver, a CP Model is defined over reversible variables that take values in domains and propagators which filter the domains of the variables. Constraint propagation scheme schedules the propagators. A reasonable constraint propagation algorithm can improve the efficiency of solving CP problems. In this paper, we propose two efficient parallel propagation schemes based on multi-thread technique for table constraint. First, we give the formal definition of the parallel consistency and prove that the parallel propagation scheme is equivalent to the classic serial propagation scheme. Then, we propose two parallel propagation schemes: static submission and dynamic submission, which exploit work stealing thread pool and atomic operations to parallelize the classic propagation of table constraint. Finally, extensive experiments on various types of problems show that the two parallel schemes outperform their original serial version on a large number of instances. The results demonstrate the competitiveness of parallel propagation algorithms on solving extensional constraints.

INDEX TERMS Constraint programming, constraint satisfaction problem, parallel constraint propagation, parallel generalized arc consistency, simple tabular reduction.

I. INTRODUCTION

Constraint Programming (CP) is an efficient technique for solving combinatorial (optimization) problems. It is widely used for solving real-world and academic problems such as routing, configuring, scheduling, car sequencing, etc [1]. Theoretically, a CP problem is defined by variables and constraints. Each variable is associated with a domain containing its possible values and each constraint contains properties that must be satisfied by a set of variables. Backtracking search is a complete approach that guarantees systematic exploration of the search space of a CP instance. In this process, the search tree grows to find a solution or prove that no solution exists [2]. In practical applications, the modeling and solving of a CP problem is completed through CP solvers [3]–[5]. There are many modules in modern CP solver to implement the two above procedures, such as variables, propagators, propagation schedule methods, search engine, etc. Variable is typically implemented as a special data structure of set, which allows adding or removing elements, changing the range by some intentional constraints, and reversing or cloning for backtracking search. Constraint can be considered as a class of sub-problems with restriction rules. It is typically implemented as a subclass of propagator [16], which provides a mechanism called filtering to remove the values that do not belong to any solution of the sub-problem. Propagation schedule scheme is a process that iterates each propagator to reduce the domains of the variables. Search engine provides a series of search methods to traverse the solution space of CP problem. These methods include backtracking search (BS), branch-and-bound search (BAB), local search (LS), etc. Besides, branching and learning are also essential components in modern CP solver. Branching determines which decisions to take and how to find a solution, and learning collects

The associate editor coordinating the review of this manuscript and approving it for publication was Xiaowen Chu.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

information during search so as to facilitate the subsequent search process. Each of these components has many possible implementations. The improvement of such solver tools has been an active topic in the CP community for a long time.

Table constraints, also called extension(al) constraints, play an important role in constraint programming because they are configured easily and explicitly by listing all allowed (or disallowed) combination of values for each constraint. Table constraints naturally arise in many application areas, such as configuration and data mining. In addition, they can be viewed as a general mechanism for representing any constraints. For decades, many filtering algorithms have been proposed for enforcing Generalized Arc Consistency (GAC) on table constraints [7]–[12]. Among them, STRbit [11], Compact-Table [12] and its extensions [13]–[15] are considered to be the state-of-the-art algorithms. In modern constraint solver, enforcing GAC on a constraint model is a schedule scheme for the filter function of propagators. To simplify the description, we use *c.propagate*() to denote enforcing GAC on certain propagator *c*.

As shown in [17], the problem of establishing arc consistency is P-complete, which is not inherently parallelizable under the usual complexity assumptions. That is to say, in the worst case, we cannot establish arc consistency polynomially faster with a polynomial number of processors [18]. Parallel constraint programming can be roughly divided into the following main categories: parallel propagators and propagation; search-space splitting; portfolio algorithms; distributed CSPs; problem decomposition. For decades, due to the synchronization issues of the domains, there are only a few studies on the parallelization of the propagation mechanism.

The pf_{all} algorithm [34] combines multiple local consistencies (such as AC and other consistencies stronger than AC) in the inference process. It is mainly used to solve the binary constraint problems. A stronger pruning capability is obtained without high additional time overhead.

In [25] and [26], Rolf and Kuchcinski presented a parallel propagation scheme (depicted in Figure 1). Parallel propagation interleaves with backtracking search, which is done by waking the consistency threads available to the constraint checking. These threads will then retrieve work from Q (the queue of constraints) whose scope variables have been changed. Once all tasks in Q have been processed in parallel, all prunings are committed to the solver. If there were no changes to any variable, a fixed point has been reached and the solver continues to search. If an inconsistency is detected in some threads, it will inform other threads and they all enter the waiting state, after that the solver needs to backtrack. To avoid serious data conflict, the clipping of domains (update model operation) is postponed until the synchronization caused by thread barrier. However, the method still has some shortcomings. First of all, it lacks a formal definition of parallel propagation and a clear theoretical proof of the equivalence to the serial propagation. Then, during its propagation, it needs a block to synchronize updates, which seriously affects the computational throughput. Finally, only

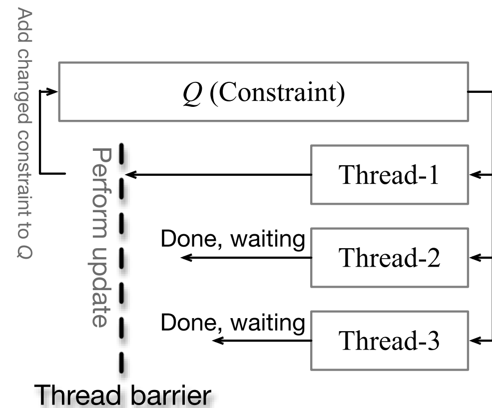


FIGURE 1. The execution model for parallel consistency.

a few instances are tested in the experiment, which makes it difficult to illustrate the scalability of the algorithm.

According to the characteristics of modern CPU with multiple cores, we improve the efficiency of existing algorithms through parallelization without changing the computing device. In this paper, we address those shortcomings and propose two parallel (a.k.a. multi-thread) propagation algorithms: static submission and dynamic submission propagation. This is the first attempt to accelerate the parallel propagation of STR style propagators (called STRs for short) using multithreading schemes. First, we present the definitions of snapshots and temporary consistency, which are the theoretical bases for guaranteeing the correctness of the proposed parallel algorithm in the propagation process. Next, some thread safe data structures are introduced into the parallel propagation algorithm, such as *AtomicVar*, STR style parallel propagators (called PSTRs for short) and so on. Then, we propose static submission propagation and dynamic submission propagation schemes, which exploit work-stealing thread pool with above improvements, and apply these propagation schemes to the state-of-the-art table constraint reduction algorithms - STRbit and Compact-Table (CT). We will show that most of the tabular reduction based on GAC algorithms can be easily parallelized. Finally, our extensive experiments on various types of problems show that the two parallel schemes outperform their original serial version on a large number of instances. The results demonstrate the competitiveness of parallel propagation algorithms on solving extensional constraint.

The rest of this paper is organized as follows. After presenting some background in Section 2, we introduce some definitions and theories for parallel propagation algorithms in Section 3. In Section 4, we describe our two parallel propagation schemes: static submission and dynamic submission. Section 5 compares the parallel schemes in several parallelism to original serial scheme through experiments conducted on a large variety of benchmarks. Finally, we conclude this paper in Section 6.

II. BACKGROUND

A constraint satisfaction problem (CSP) $\mathcal{P}$ is a triple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{X}$ is a set of n variables, $\mathcal{D}$ is a set of domains of $\mathcal{X}$ and $\mathcal{C}$ is a set of e constraints.

For each variable $x \in \mathcal{X}$, we use $D(x)$ to denote a finite set of possible values that can be assigned to x, and (x, a) to denote the value $a \in D(x)$. A constraint c consists of two parts, an ordered set of variables $scp(c) = \{x_1, x_2 \ldots x_r\}$ and a subset of the Cartesian product $D(x_1) \times D(x_2) \times \ldots \times D(x_r)$ that specifies the allowed combinations of values for $scp(c)$, denoted by $rel(c)$. Let $\tau = \{a_1, a_2, \ldots, a_r\}$ be an r-ary tuple, and the individual value a_i is denoted by $\tau[x_i]$. A tuple $\tau \in rel(c)$ is valid iff $\forall x \in scp(c)$, $\tau[x] \in D(x)$, otherwise τ is invalid. A tuple τ is a support for (x, a) on c iff τ is valid and $\tau[x] = a$. Accordingly, a variable x has an ordered set of its *subscription* constraints $srb(x) = \{c \in \mathcal{C} \mid x \in scp(c)\}$.

Definition 1 (Generalized Arc Consistency, GAC): Given a constraint network $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$:

- A constraint c is generalized arc consistent iff $\forall\, x \in scp(c)$ and $\forall\, a \in D(x)$, there exists a support for (x, a) on c.
- A constraint network $\mathcal{P}$ is generalized arc consistent iff all constraints in $\mathcal{C}$ are generalized arc consistent.

A value (x, a) is generalized arc consistent, or GAC-consistent iff it has at least one support in each constraint involving x, and GAC-inconsistent otherwise. It is easy to see that a GAC-inconsistent value cannot occur in any solution and will be dropped after enforcing GAC algorithm. At each level of the search tree, a variable x and a value $a \in D(x)$ are selected and GAC is established by propagating the assignment. A dead-end is reached if the propagation fails, and then backtracking occurs.

Example 1: Let $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be the constraint network depicted in Figure 2(a), where $\mathcal{X} = \{x_1, x_2, x_3\}$, $\mathcal{D} = \{1, 2, 3\}$ and $\mathcal{C} = \{c_1, c_2\}$. Constraint c_1 is $x_1 = x_2$ and c_2 is $x_2 < x_3$. $\mathcal{P}$ is not generalized arc consistent because there are some values inconsistent with some constraints. Checking constraint c_1 does not permit to remove any value. But when checking constraint c_2, we see that $(x_2, 3)$ must be removed because there is no value greater than it in $D(x_3)$. We can also remove value 1 from $D(x_3)$ because of constraint c_2. Removing 3 from $D(x_2)$ causes in turn the removal of value 3 for x_1 because of constraint c_1. Now, all remaining values are compatible with all constraints. Finally, we get a new $\mathcal{P}$ after GAC, as shown in Figure 2(b).

For many constraint solvers, e.g., Gecode, a constraint problem can be modelled as an object of Gecode *home* class,

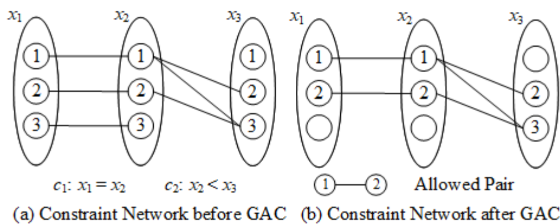


FIGURE 2. A constraint network.

which is constitutive of the array of variables, propagators (implementation of constraints) and branchers (implementation of branching). A table constraint, which is an *extensional* constraint in Gecode, is a subclass of propagator. A general constraint solver decomposes the process of enforcing GAC into two parts: the filtering algorithm that acts as a member function of propagator and the schedule method that dispatches filtering functions.

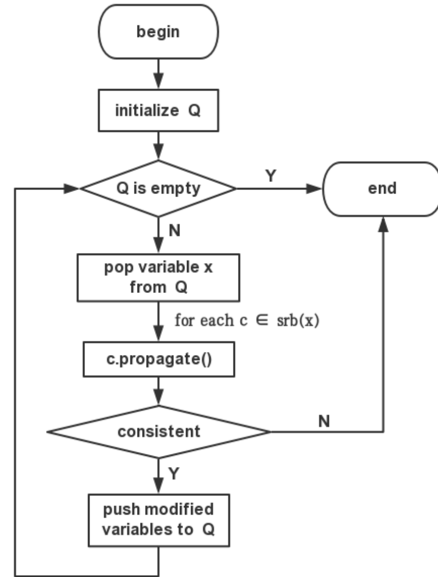


FIGURE 3. A flow diagram of propagation schedule method.

Figure 3 briefly shows a flow diagram of propagation schedule method. The schedule method maintains a propagation queue to restore the modified variables. The data structure of Q can be array or heap. When Q pops a variable x, the function iteratively calls *c.propagate*() for all $c \in srb(x)$ to prune the GAC-inconsistent values depending on the remaining tuples of c, detect inconsistency and push the modified variables to Q.

Some unnecessary work can be avoided by a time-stamp mechanism. A time-stamp is a value denoting the time at which certain events occur such as domain and tuples reduction. Time-stamps enable the progress of algorithms to be tracked over time. The basic idea of time-stamp is presented in Figure 4: the algorithm maintains a global time-stamp and attaches a time-stamp to each variable and constraint object. The global time-stamp is incrementally changed and it helps the algorithm update the time-stamp of relevant object whenever the above events occur. A constraint c needs updating iff $\exists x \in scp(c)$ s.t. $stamp[x] > stamp[c]$; a variable x needs filtering iff x is not assigned and $\exists c \in srb(x)$ s.t. $stamp[c] > stamp[x]$.

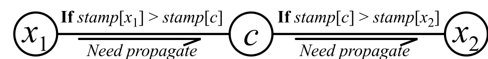


FIGURE 4. The time-stamp mechanism.

Algorithm 1 CLASS STR PROPAGATOR

```
1  Method updateTable():
2      ...
3  Method filterDomains():
4      ...
5  Method initial():
6      /*initialization of S^sup, S^val and lastSize*/
7  Method propagate():
8      initial();
9      updateTable();
10     if TWO is detected then
11         return false
12     filterDomains();
13     if DWO is detected then
14         return false
```

Algorithm 1 gives a general description of STRs (for more detailed versions, see [12]). The entry function of propagator is *propagate*(), which is an abstract function that must be overridden by all its subclasses. After initializing some class fields at line 5 (for more detail, see section 4 or [12]), there are two main tasks to be executed. First, function *updateTable*() (*deleteInvalidTuple*() in STRbit [11]) is called to remove invalid tuples based on the current domain of scope variables. And then, in function *filterDomains*() (*searchSupport*() in STRbit [11]), the domains need filtering according to valid tuples. The *propagate*() returns *false* if it detects that all tuples become invalid (*TWO*, for short, at line 10) or the domain of some variable is wiped out (*DWO*, for short, at line 13).

III. THEORIES FOR PARALLEL PROPAGATION ALGORITHMS

In this section we will introduce some definitions and theories for parallel propagation algorithms.

A. FIXED POINT

We know that the change in the domain of a variable causes constraint propagation. In constraint propagation, a fixed point includes the following two cases:

- If all constraints have been checked and no new changes to the domain have been triggered, the propagation is successful.
- If *DWO* or *TWO* is detected, the propagation fails.

To enforce GAC on $\mathcal{P}$, if the constraint propagation is successful, no matter what kind of scheduling algorithm is adopted, the filtering of the variable domain is equivalent, but not the converse. For example, the propagator that causes the failure of constraint propagation may be different. This may affect some heuristics (wdeg [23], ABS [24] etc.) for recording failure information and thus may affect the search tree.

B. THREAD POOL

A thread pool is a parallel computing model for implementing concurrent execution. It manages a group of threads to process a large number of tasks. Since multiple threads can be executed in parallel, this approach may be very efficient regarding the overall program performance on many computer systems. By restricting the number of threads and the reuse of threads resources are saved and additionally the system stability is increased.

The implementations of thread pools are mainly in major languages. Java introduces thread pools since JDK5. Other programming languages also implement multiple types of thread pools, such as TBB, Cpp-taskflow, etc. In our implementation, we use the work stealing thread pool, which is an efficient type of thread pool. Our parallel propagation schemes create all the tasks (i.e. *propagate*()) simultaneously and dispatch them to a thread pool with a fixed number of threads. The number of threads in a thread pool is called the capacity of *pool*. If one thread has finished its works, it can "steal" work from others. Therefore, the load-balancing is improved. For ease of description, we abstract some common methods of thread pool as follows:

- *pool.submit*(c): Submits a propagator c to *pool*.
- *pool.batchSubmit*(b): Submits the given propagators b to *pool*.
- *pool.awaitQuiescence*(): Waits and/or attempts to assist performing tasks until this pool is *Quiescence*.

A *pool* reaches *Quiescence* when all worker threads in *pool* are idle. An idle worker is unable to obtain an executable task because none is available to steal from other threads, and there are no pending submissions to the pool. In this article, we exploit *awaitQuiescence*() method to synchronize various data.

C. GLOBAL DOMAIN AND SNAPSHOTS OF VARIABLE

In [25], Rolf and Kuchcinski present shared intermediate domains to synchronize the changes of domains, and present local thread intermediate domains to cache the changes made by the local thread. We will improve these two intermediate domains and put forward definitions of global domain and snapshot.

At the beginning of *propagate*() in PSTRs, it will cache the domains of scope variables. The subsequent propagation only depends on the cached domains, without accessing the original domains. At the end of propagation, the propagator submits cached domains back to original domains, that is, intersecting the original domains with cached domains. We call the cached domain of a variable x in a propagator c as a snapshot, denoted by $\Sigma_c(x)$, and the original domain as the global domain, denoted by $D(x)$.

During parallel propagation, a variable x may have different snapshots in different propagators at different time. However, when parallel constraint propagation reaches a fixed point, if the constraint propagation succeeds, all snapshots are submitted to their global domainand and no value is removed.

Otherwise, the snapshot is not necessarily the same as its global domain, because *DWO* or *TWO* is detected by another propagator.

Recalling that the core idea of using the time-stamp is to mark the changes of each variable and constraint. In the serial algorithm, these changes occur serially, and the last change must be the latest state. But parallel algorithms are not necessarily the case. For example, if two workers modify the variable x concurrently, *worker*1 deletes the value (x, a) and *worker*2 deletes the value (x, b), then *worker*1 thinks that (x, b) is still valid and worker2 thinks that (x, a) is still valid, that is, neither of their snapshots of x is up to date. Therefore, time-stamp needs improving to handle such situation.

In addition, there is another way to avoid redundant validation, which uses the *lastSize* data structure in some STRs. This kind of methods compares the domain size of variables in the previous propagation, which is recorded in *lastSize*, with current domain size to determine which variables have been changed. However, the *lastSize* is a one-dimensional record and doesn't work in the above example. Therefore, we introduce the concept of *snapshot* to directly record the domain in a two-dimensional vector. So in the initialization, our PSTRs need to get the snapshots from scope variables. During the subsequent propagation, the table is updated according to the snapshots instead of the global domains and then the snapshots are filtered. All modified snapshots are submitted to the global domain at last. The disadvantage of the snapshot mechanism is that it does not immediately inform the propagator of changes to the global domain, which increases the number of calls to the method *propagate*(). But the correctness of the parallel program is guaranteed.

D. TEMPORARY GENERALIZED ARC CONSISTENCY

Parallel propagation has not been well formalized since it was proposed. In the following, we present the definition of temporary generalized arc consistency and its equivalence to generalized arc consistency at the fixed point is proved theoretically. To distinguish it from the existing temporal consistency [29], we call it temporary consistency.

A tuple $\tau \in rel(c)$ is temporary valid iff $\forall x \in scp(c)$, $\tau[x] \in \Sigma_c(x)$, otherwise τ is invalid. A tuple τ is a temporary support for (x, a) on c iff τ is temporary valid and $\tau[x] = a$.

Definition 2: (Temporary Generalized Arc Consistency, TGAC): Given a constraint network $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$:

- A constraint c is temporary generalized arc consistent iff $\forall x \in scp(c)$ and $\forall a \in \Sigma_c(x)$, there exists a temporary support for (x, a) on c.
- A constraint network $\mathcal{P}$ is temporary generalized arc consistent iff all constraints in $\mathcal{C}$ are temporary generalized arc consistent.

Proposition 1: When the parallel constraint propagation reaches a fixed point, TGAC is equal to GAC.

Proof: According to the definition of the fixed point, when the parallel propagation is successful, i.e., $\mathcal{P}$ is temporary generalized arc consistent, every variable domain is the same as its snapshots, that is, $\forall x \in \mathcal{X}, \forall c \in srb(x)$, we have $\Sigma_c(x) = D(x)$. Therefore, all constraints are checked based on the global domains. At the moment, $\mathcal{P}$ is generalized arc consistent.

Otherwise, when the parallel propagation fails, $\exists x \in \mathcal{X}, \exists c \in srb(x)$, we submit $\Sigma_c(x)$ to $D(x)$, and we get $D(x) = \emptyset$. $\mathcal{P}$ is not generalized arc consistent at the moment. And then, we get $\exists c' \in srb(x)$, $\Sigma_{c'}(x) = \emptyset$ when it accesses the global domain. That is, $\mathcal{P}$ is not temporary generalized arc consistent.

Hence, the proposition is proved.

IV. PARALLEL PROPAGATION SCHEME

In this section, we present a variety of data structures for implementing parallel propagation algorithms. First, we introduce some reversible thread-safe variable classes used in our algorithms. Then, we propose a static submission algorithm, which improves the original scheme presented in [25] in the following aspects: using the atomic operation to improve the synchronization efficiency, using the new time-stamp mechanism to reduce the redundancy computation and using work-stealing thread pool to accelerate the parallel scheduling. In order to adapt static propagation scheme, we propose PSTR^{ss} as the underlying filtering algorithm for table constraint. Finally, we further optimize the static submission scheme and propose a dynamic submission scheme, which removes the thread barrier in the original scheme. It allows propagator PSTR^{ds}, an improvement for PSTR^{ss}, to dynamically submit the new tasks to the thread pool. Hence, it improves the throughput of the computation. Please note that, PSTRs maintain TGAC, so when the propagation reaches the fixed point, the two parallel schemes maintain GAC.

A. REVERSIBLE THREAD-SAFE VARIABLE CLASSES

We use *SafeVar* as the basic class for reversible thread-safe variable classes, and present the following additional abstract methods for implementing the *SafeVar* class:

- *getSnapshot*(): Takes a snapshot from the current variable before a constraint check starts.
- *submit*(b): Submits the snapshot b to the global variable domain. The values that are absent in b will be deleted in the global domain, i.e., $D(x) = D(x) \cap \Sigma(x)$.

In CP solvers, the representation of variable domains is an important design choice, which has a significant impact on the performance. The paper [22] discusses the usage of sparse sets in integer domain implementation and comprehensively compares the space-time complexity of various implementations. In fact, these implementations can also be modified to thread-safe version. In this paper, we choose the atomic bit vector as the basic implementation of thread-safe variable. This is mainly based on the following three considerations:

- The method based on bit representation is efficient on most problems. So far, many CP solvers have adopted bit vector as the default implementation.

- With bit-level parallelism and atomic operations provided by programming languages, the atomic bit vector makes it easy to implement non-blocking parallel algorithm.
- The assignment and flipping of bit vectors is lightweight, so propagators can easily access the snapshots.

To facilitate a more abstract expression, no matter how long the bit length of a word (natural data units of the computer architecture) is, we abstract the bit vector (an array of Integer or Long) into a *BitSet* class. A *BitSet* is an array data structure that compactly stores bits. Here we use it as the basic data structure to represent the domain and snapshot. If a variable has 120 values, then the length of its *BitSet* is 120.

As the name implies, *AtomicBitSet* is an atomically operable *BitSet* class. The implementation of *BitSet* varies in different programming languages. Mainstream programming languages may not implement the *AtomicBitSet* class. But most of them provide atomic basic data types and methods, such as atomic integer and atomic long.

The following are the primary methods of class *AtomicBitSet*. Most of them provide atomic and non-atomic versions, and atomic versions are followed by *. Here we assume that a is an instantiated object of *AtomicBitSet*.

- $a.Set()$: Sets all bits of a to 1.
- $a.Set(pos)$: Sets the bit at position pos to 1.
- $a.Reset()$: Sets all bits of a to 0.
- $a.Reset(pos)$: Sets the bit at position pos to 0.
- $a.And(b)$: Sets the bits of a to the bitwise AND result of a with b.
- $a.Or(b)$: Sets the bits of a to the bitwise OR result of a with b.
- $a.And\&Get^*(b)$: Atomically sets the bits of a to the bitwise AND result of a with b and returns this result. This method only provides an atomic implementation.
- $a.Count()$: Returns the number of bits that are set to 1.

In addition, we list some common methods for atomic integer as follows. Here, we assume that a is an instantiated object of atomic integer.

- $a.Set^*(b)$: Atomically sets a to the given value b.
- $a.Try\&Set^*(expected, updated)$: Atomically sets a to the given $updated$ value if $a = expected$ and returns *true*. Otherwise, doesn't change the value of a and returns *false*.
- $a.Get\&Inc^*()$: Atomically increments the value of a by one and returns original value.

In algorithm 2, we present two additional methods of class *AtomicBitVar*, which uses *bitDoms*, an array of *AtomicBitSet*, to represent the reversible global domain of variable. The $bitDoms[0]$ represents the initial domain and $bitDoms[currentLevel]$ represents the current domain.

Method $getSnapshot^*()$ is used to get the snapshot of the variable atomically. Method $submit^*(b)$ submits *AtomicBitSet* b to the global domain and returns the result.

Algorithm 2 Class *AtomicBitVar*

```
  Data: bitDoms:Array of AtomicBitSet
  currentLevel: current level of search
1 Method getSnapshot*():
2 |   return bitDoms[currentLevel];
3 Method submit*(b : AtomicBitSet):
4 |   return bitDoms[currentLevel].And&Get*(b);
```

Algorithm 3 STATIC SUBMISSION PROPAGATION

```
  Data: X_evt: set of modified variables
  C_evt: set of propagators to be executed
  consistent: global Boolean
  pool: global thread pool
  stamp: time-stamp of variables
  time: global time-stamp
1 consistent ⟵ true;
2 foreach x ∈ X_evt do
3 |   stamp[x] ⟵ time;
4 repeat
5 |   C_evt ⟵ {c ∈ C | ∃x ∈ scp(c) ∧ stamp[x] = time};
6 |   time ⟵ time + 1;
7 |   pool.batchSubmit(C_evt);
8 |   pool.awaitQuiescence();
9 until C_evt = ∅ ∨ ¬consistent;
```

B. STATIC SUBMISSION PROPAGATION SCHEME

We first introduce the static submission propagation in algorithm 3, which iteratively submits a set of propagators to the thread pool *pool*. The algorithm 3 has the following fields: The X_{evt} is a set of variables whose domain has been changed. The C_{evt} is a set of propagators that need propagating. The *consistent* is a global Boolean value used to mark whether the current network is GAC. The *time* is a global time-stamp and is incremented by one in each iteration of propagation. The *stamp* is used to mark a variable that has just been changed. If the domain of a variable x is changed, the $stamp[x]$ is set to the *time*. The *pool* is a global thread pool for scheduling the propagators. Algorithm 3 first sets *consistent* to *true*, and then sets the *stamp* of each variable in X_{evt} to *time*. Lines 4-9 is the loop used to concurrently propagate until the C_{evt} is empty or inconsistency is detected. Line 5 obtains the set of propagators whose at least one scope variable is just modified, i.e., $\exists x \in scp(c), stamp[x] = time$. Next, *time* is incremented by 1 at line 6. And then C_{evt} is submitted to the *pool* for parallel propagation at line 7. Line 8 is the synchronization point where the main program waits for all propagation tasks to complete.

The algorithm 4 is the framework of $PSTR^{ss}$, which is called at line 7 of algorithm 3. To show the extensibility of the algorithm, we omit the same part as STRs. If we want to parallelize Compact-Table in this way, we just need to fill the omitted part with the corresponding code of Compact-Table.

Algorithm 4 CLASS PSTR^{ss} PROPAGATOR

```
Data: scp: array of scope variables
      S^val, S^sup: temporary sets of variables
      Σ: array of local snapshots
      Σ̀: array of last snapshots
1  Method initial():
2      remove all elements in S^val and S^sup;
3      foreach x ∈ scp and consistent do
4          Σ(x) ⟵ x.GetSnapshot();
5          if Σ̀(x) ≠ Σ(x) then
6              Σ̀(x) ⟵ Σ(x);
7              S^val ⟵ S^val ∪ {x};
8          if x is unbind then
9              S^sup ⟵ S^sup ∪ {x};
10 Method updateTable():
11     foreach x ∈ S^val and consistent do
12         update table according to Σ(x);
13     /*detect inconsistent*/
14     if TWO detected then
15         consistent ⟵ false;
16         return;
17 Method filterDomains():
18     foreach x ∈ S^sup and consistent do
19         filter Σ(x) according to the updated table;
20         /*submit snapshots*/
21         tmp ⟵ x.Submit*(Σ(x));
22         /*detect inconsistent*/
23         if DWO detected then
24             consistent ⟵ false;
25             return;
26         Σ̀(x) ⟵ Σ(x);
27         /*if global domain is changed*/
28         if tmp ≠ Σ(x) then
29             stamp[x] ⟵ time;
30 Method propagate():
31     initial();
32     updateTable();
33     filterDomains();
```

And STRbit can be parallelized in the same way. The entry to the algorithm is the method *propagate*(), which is called by the worker thread of the *pool*. Similar to STRs, the method *propagate*() calls three methods (i.e., *initial*(), *updateTable*() and *filterDomains*()) in turn, and exits if *TWO* or *DWO* occurs. PSTR^{ss} also has some same fields to STRs. The array *scp* represents the scope variables of c. The set S^{val} contains variables whose domain has been reduced since the previous invocation of the filtering algorithm on c. The set S^{sup} contains unbind variables (from the scope of the constraint c) whose domain contains each at least one value for which a support has not been found. These two sets enable us to restrict loops on variables to relevant ones. Next, we introduce two additional fields of PSTRs: local snapshots (Σ) and last snapshots ($\grave{\Sigma}$). Unlike STRs that record the domain size of each modified variable x by *lastSize*, we use $\grave{\Sigma}$ to record the last snapshot right after the execution on c. The algorithm obtains a local snapshot of each variable at line 4. If the last snapshot and the local snapshot are not equal at line 5, the variable x has been changed. Therefore, x is added to S^{val}, and the last snapshot needs updating. The method *updateTable*() filters the tuples according to the local snapshots. And in turn, the method *filterDomains*() filters the local snapshots according to the tuples. At line 21, PSTR^{ss} submits the snapshot to the global domain and gets the updated snapshot atomically. Line 26 updates the last snapshots. Variables whose domain has been changed will mark its *stamp* to *time* at line 29 so that algorithm 3 can recognize the modified variables (line 5, algorithm 3)in the next iteration. Besides, it needs to set the global vairable *consistent* to *false* and returns when *TWO* (line 14) or *DWO* (line 23) is detected. Then other propagators detect *consistent* $=$ *false* in iteration, algorithm 4 immediately exits and returns to algorithm 3. After the jobs of the thread pool are all ended (line 8, algorithm 3), the propagation exits (line 9, algorithm 3). The parallelism of the thread pool is less than that of the CPU, so task preemption generally does not occur in theory.

C. DYNAMIC SUBMISSION PROPAGATION SCHEME

In the previous section, we introduce the static submission propagation. It uses the snapshot mechanism to guarantee the correctness of concurrent execution. The use of atomic operations avoids blocking the access to the variable domain. However, this algorithm still needs a block after submitting C_{evt} in the loop. This is just like a faucet that is frequently switched and does not always keep the maximum flow, which reduces the efficiency of water storage. An obvious optimization is removing the frequent switching to maintain the maximum flow.

Therefore, we propose dynamic submission propagation in algorithm 5, which removes the block between iterations. Unlike algorithm 3, this algorithm only submits propagators to the thread pool once at the beginning of the constraint propagation. During the parallel propagation, one propagator can dynamically submit other propagators depending on modified variables. The dynamic submission scheme also needs to preserve a block for synchronization at line 8. After this, the propagation is completed and the solver returns to the backtracking search algorithm.

The dynamic submission scheme needs to call PSTR^{ds} (as shown in algorithm 6), which is an improvement of PSTR^{ss}. The entry method in PSTR^{ds} is *propagate*(). Instead of blocking the program for synchronization between iterations, PSTR^{ds} calls method *submitOthers*() at line 30 to dynamically submits other propagators based on modified variables.

Algorithm 5 DYNAMIC SUBMISSION PROPAGATION

```
   Data: X_evt: set of modified variables
   pool: global thread pool
1  consistent ⟵ true;
2  foreach x ∈ X_evt do
3      foreach propagator c ∈ srb(x) do
4          if ¬consistent then
5              return;
6          if c.numReq.get&Inc*() = 0 then
7              pool.submit(c);
8  pool.awaitQuiescence();
```

Algorithm 6 CLASS PSTR^{ds} PROPAGATOR

```
   Data: Same as PSTR^ss
   Additional data structures:
   Y_evt: set of domain changed Variables
   pool: global thread pool
   numReq: atomic integer
1  Method initial():
2      /*same as PSTR^ss*/
3  Method updateTable():
4      /*same as PSTR^ss*/
5  Method filterDomains():
6      foreach x ∈ S^sup and consistent do
7          filter Σ(x) according to the updated table;
8          /*submit snapshots*/
9          tmp ⟵ x.Submit*(Σ(x));
10         /*detect inconsistent*/
11         if DWO detected then
12             consistent ⟵ false;
13             return;
14         Σ̀(x) ⟵ Σ(x);
15         /*if global domain is changed*/
16         if tmp ≠ Σ(x) then
17             Y_evt ⟵ Y_evt ∪ x;
18 Method submitOthers():
19     foreach x ∈ Y_evt and consistent do
20         foreach propagator c ∈ srb(x) do
21             if c ≠ this then
22                 if numReq.get&Inc*() = 0 then
23                     pool.submit(c);
24 Method propagate():
25     repeat
26         numReq.set*(1);
27         initial();
28         updateTable();
29         filterDomains();
30         submitOthers();
31     until numReq.Try&Set*(1, 0) ∨ ¬consistent;
```

This is obviously different from the static submission, which submits propagators only by itself.

We introduce two additional data structures in PSTR^{ds}. Y_{evt} is used to record the modified variables. *numReq* is used to record the number of propagation requirement and prevent a propagator from being executed by multiple workers at the same time. Initially, *numReq* is set to 0, indicating that the propagator does not have an execution request. If the propagator c needs submitting to the *pool*, the value of $c.numReq$ is obtained and then increased by 1 via method $numReq.get\&Inc^*()$ (line 6 in algorithm 5 and line 22 in algorithm 6). If the original value is 0, c can be submitted to *pool*. Otherwise, c is executed and does not need resubmitting. If c is successfully submitted, at the end of its propagation (line 30 in algorithm 6), PSTR^{ds} attempts to modify the value of *numReq* by calling method $numReq.Try\&Set^*(1, 0)$. If the original value of $c.numReq$ is 1, the only execution request has been completed. Then $c.numReq$ is set to 0 atomically. The statement $Try\&Set^*(1, 0)$ returns *true*, and the loop exits. Otherwise, the value of $c.numReq$ is greater than 1 and the statement $Try\&Set^*(1, 0)$ returns *false*. The reason is that c receives other execution requests during propagation, which causes $c.numReq$ to be greater than 1. And if *consistent* is *true*, the propagation continues. To reduce the value of $c.numReq$, PSTR^{ds} atomically sets the *numReq* to 1 at line 26. This indicates that no matter how many execution requests were received before, the propagator starts to respond.

In short, if a new execution request is received during propagation, it will be processed in the next loop. Hence, the correctness of the data is guaranteed.

D. DISCUSS

1) PARALLEL PROPAGATION HYPERACTIVITY

In parallel propagation, while a task is waiting in the queue, concurrent modification generated by other executing tasks accumulates. The more worker threads, the more computing resources are available to execute the propagators. This may result in the propagation task being executed as soon as it is submitted, without acquiring more modifications. So propagators are scheduled to execute more frequently. We call this phenomenon as Parallel Propagation Hyperactivity. For example, let $c \in \mathcal{C}$ is a constraint of $\mathcal{P}$ and $scp(c) = \{x, y, z\}$. In a serial parallel scheme, (x, a), (y, b) and (z, d) have been deleted by other propagators while c is waiting in the queue. At this point, if c starts to propagate, the filtering algorithm considers these three values. However, in the parallel scheme, since computing resources are easier to obtain, c may be propagated after deleting only two values (x, a) and (y, b). Therefore, c should be propagated again after deleting (z, d). Hence, the number of propagator executions generally increases as the computing resources increase.

2) COMPLEXITY

In paper [30], the worst-case time complexity of serial propagation excluding *propagate*() (i.e. ignoring the calls to *propagate*()) is given as $O(erd)$, which is the number of c-values in $\mathcal{P}$. e, r and d are the number of constraints, the largest constraint arity and the greatest domain size, respectively. This indicates that all c-values needs to be propagated in the worst case. Since our improvements are for most of the STR algorithms and their worst-case time complexity varies, we assume that the worst-case time complexity of the STRs is $O(T)$. Therefore, the worst-case time complexity of the serial propagation embedded STRs is $O(erdT)$.

In the worst case, the parallel algorithms also need to propagate *erd* times. Besides, the parallel propagation does not specify which thread pool to use, so we assume that the worst-case time complexity of concurrent scheduling is $O(S)$. If there are p threads for performing propagation, we conclude that the worst-case time complexity of parallel propagation schemes is $O(erdTS/p)$.

We have given the worst-case time complexity, but in practice, we use another formula to intuitively illustrate the performance of the algorithms. We assume the actual execution time of parallel propagation is nTS/p, where n is the actual number of the calls to *propagate*() and S is the time overhead for processing concurrent transactions (such as waking up thread, switching context and stealing work). Therefore, the execution time of serial propagation is nT. For some instances with fewer constraints (resp. smaller table size of a constraint), the n (resp. T) is smaller. The parallel algorithm may lose competitiveness to the serial algorithm when S is large. In addition, owing to the parallel propagation hyperactivity, high parallelism may have a larger n and may lose competitiveness to the low parallelism.

V. EXPERIMENTS

A. EXPERIMENTS ON NON-BINARY TABLE CONSTRAINT INSTANCES

Datasets. We first evaluate the performances of static submission and dynamic submission schemes by comparing the search algorithms embedding serial and parallel propagation schemes on non-binary table constraint problems. To provide a comprehensive evaluation, we sample various CSPs from Lecoutre's XCSP repository [31] and XCSP3 website [32] including both synthetic and real-world problems: MODEL RB, RENAULT, CROSSWORD, CRIL, TRAVELLING SALESMAN (TSP-20, TSP-25), MDD, LARGE TABLES, DIMACS, etc.

Table 1 shows the sizes of some test instances. For each group of instances, Table 1 presents the number of variables (n), the number of constraints (e), and the maximum arity of constraints (r), the average and maximum size of variable domain (d(avg/max)) and the average and maximum number of tuples (#τ(avg/max)). The category of the group instances is marked on the top left of each grid. By observing Table 1, we can see that for the instances of MODEL RB and LARGE TABLE, the difference in size between variables (constraints) is not big. For BDD and DIMACS, the size of their domain is only 2. The average number of tuples of aim and dubios is less than 10, so these problems are quite small. The domain of the CROSSWORD is alphabet, so the size of their domain is 26. In the problems MODIR, TSP and JNH, there is a large difference between the average and the maximum number of the tuples, which may affect the efficiency of our parallel algorithm. This scenario will be analyzed later.

TABLE 1. The sizes of some test instances.

Series	$n/e/r$	d(avg/max)	#τ(avg/max)
MODEL RB			
rand-3-20-20	20/58/3	20/20	2890/2944
rand-3-20-20-fcd	20/58/3	20/20	2890/2944
LARGE TABLES			
rand-5-12-12	12/200/5	12/12	12442/12442
rand-8-20-5	20/18/8	5/5	78122/78592
rand-10-20-10	20/5/10	10/10	10000/10000
rand-10-60-20	60/30/10	20/20	51200/51200
RENAULT			
modiR	110/151/10	5/10	1339/48721
CROSSWORD			
lexVg	129/22/12	26/26	1702/2038
ogdVg	152/23/13	26/26	20630/30380
ukVg	159/24/13	26/26	10098/14159
wordsVg	134/22/12	26/26	3149/3868
TRAVELLING SALESMAN			
tsp-20	61/230/3	387/1001	1531/14712
tsp-25	76/350/3	391/1001	1511/14508
MDD			
MDD0.7	25/58/7	5/5	39072/39662
MDD0.9	25/58/7	5/5	39048/39746
half	25/56/7	5/5	39799/47480
BDD			
bddSmall	21/133/18	2/2	57757/57757
DIMACS			
aim-50	50/130/3	2/2	7/7
aim-100	100/300/3	2/2	7/7
aim-200	200/899/3	2/2	7/7
dubois	98/65/3	2/2	4/4
jnhSat	100/818/11	2/2	91/4607
jnhUnsat	100/848/11	2/2	87/1927

Implementation Details. First, we extend Compact-table (CT) to PCT^{ss} for static submission and PCT^{ds} for dynamic submission. Then, we extend STRbit to PSTRbit^{ss} for static submission and PSTRbit^{ds} for dynamic submission. We use PCTs to represent PCT^{ss} and PCT^{ds}, and PSTRbits to represent PSTRbit^{ss} and PSTRbit^{ds}. Both of PCTs and PSTRbits are embedded in the backtracking search. All the algorithms have been implemented in Scala 2.12 and Java 11. We use the *ForkJoinPool* [28] as the basic thread scheduling mechanism. All codes that can reproduce our experiments are available at https://github.com/leezear2019/sub1. Our experiments run on a computer with AMD R7 1700 3.2GHz (8 cores, 16 threads), 16G RAM, Manjaro 18. To highlight the efficacies of two propagation schemes, we evaluate the search algorithms using a relatively simple heuristic dom/ddeg. Other efficient heuristics such as dom/wdeg or ABS are not chosen because these heuristics record information about propagation failures, which may be different in serial and parallel algorithms, affecting the shape of search tree and the number of search nodes. The advantage of dom/ddeg heuristic

is that as long as an instance has a solution, all algorithms have the same number of search nodes, which makes the comparison experiment more intuitive.

In addition, to show the performance of parallel algorithms with different parallelism, we set the parallelism (the number of working threads) of thread pool for the two algorithms to 2,3,5,7,9,13,16. If the parallelism of the thread pool is set to 2, there are actually two threads handling the propagation tasks. In fact, we perform a total of 30 tests on each instance: 1 test for CT, 7 tests for PCT^{ss} for different parallelisms, 7 tests for PCT^{ds}, 1 test for STRbit, 7 tests for $PSTRbit^{ss}$ and 7 tests for $PSTRbit^{ds}$. We use "algorithm name @ parallelism" to express an algorithm with specified parallelism of the thread pool. For example, PCT^{ds}@7 means the PCT^{ds} algorithm with 7 threads.

Metrics. The time limit is set to 900 seconds, and the instance that is not solved within the time limit is called time-out instance. We remove a small number of known timeout instances before the experiments, because these instances will waste a lot of time and we can't get more useful information. The average results of the remaining instances are presented in Table 2. Table 2 shows the average results of original serial algorithm (serial) and the best parallel algorithm ($best^p$). In addition, for each group of instances, we present the average solve time (cpu^s), average propagation time (cpu^p) in seconds, speed-up ratio of propagation time (ratio) and the number of the serial/parallel execution of propagators (#prop) in the millions. The best speed-up ratio of each row is shown in bold. For the sake of fairness, the timeout instances tested on either algorithm are also excluded in statistical averaging. Because parallel algorithms and serial algorithms are equivalent, the number of search nodes per group is the same. To save space, these numbers are omitted.

CT vs. STRbit. In [33], Schneider and Choueiry first evaluate CT and STRbit, which are state-of-the-art STR algorithms. They use a different bit vector technology as the primary acceleration approach. By comparing the experimental data of STRbit and CT, the STRbit is nearly as efficient as the CT in some instances, but the CT is faster than the STRbit in most instances. In general, CT outperforms STRbit in terms of the size of the current problems. By comparing CT and STRbit with their respective parallel versions, we can find that the parallel algorithms achieve a significantly higher speed-up ratio on STRbit. This demonstrates that the longer the time of serial propagation, the greater the improvement of parallel propagation.

STRs vs. PSTRs. We compare all the results of serial and parallel schemes from a whole perspective. In terms of solving time, for CT algorithm, its serial scheme performs best on 8 groups of instances. The static submission scheme performs best on 1 group. The dynamic submission scheme performs best on 14 groups. For STRbit algorithm, its serial scheme performs best on 6 groups. And the static submission scheme performs best on 1 group. And the dynamic submission performs best on 16 groups. In general, among the 23 groups of instances, the serial scheme performs better on about 1/3 of the total, while the parallel schemes perform better on the remaining 2/3. From the perspective of details, the examples with better performance under serial scheme are concentrated on individual examples such as DIMACS, RENAULT and so on, which generally have the following characteristics:

- Fewer constraints. Since our algorithms use propagator as the basic parallel task, problems with fewer constraints (e.g., the rand-10-20-10 has 5 constraints per instance) do not take more advantage of the improvements brought by parallelism.
- The table size of a constraint is small. For example, some constraints on DIMACS and RENAULT only contain fewer than 10 tuples. Obviously, the propagation time of such constraints is extremely short. Therefore, it is not worth to parallelize them.
- The domain is small. For example, the domain of SAT problems: AIM and JNH, is $\{0, 1\}$. This may lead to a shorter propagation time. So it is not worth the parallelize them either.

By observing the number of serial/parallel executions of propagators (#prop), we find that the #prop of dynamic submission propagation is the most, followed by static submission propagation, and serial propagation is the least. This is because of the parallel propagation hyperactivity (we have discussed in Section 4).

PCT^{ss} vs. PCT^{ds}. Figure 5 shows the speed-up ratio of the solving time for each series. We use the CT algorithm as the baseline to measure other algorithms. The x-axis represents the name of each group instances, and the y-axis represents the acceleration ratio of the solving time. Each group of instances is arranged from small to large according to the PCT^{ds}@7 speed-up ratio. The PCT^{ds} algorithm is generally superior to the PCT^{ss} algorithm and is superior to CT algorithm on majority of the group instances. Even on many instances, PCT^{ds} achieves more than three times the acceleration. Typically, the most efficient algorithm is PCT^{ds}@7, which means PCTs have good performance when the parallelism is set to 7. It also confirms our observation in the previous paragraph. This is because we have optimized the block used for synchronization in the propagation, which greatly improves the throughput of parallel computing. Because the scale of MODEL RB and LARGE TABLE is so large, in this experiment, we get the best performance when the parallelism is set to 16.

B. EXPERIMENTS ON BINARY CONSTRAINT INSTANCES

Although the presented parallel propagation schemes are mainly used to improve the efficiency of solving non-binary table constraint problems, we still evaluate our algorithms on some binary constraint problems. Table 3 presents the experimental results of the CT, PCT^{ds}@5 and pf_{all} algorithms on some (series and singleton) binary constraints instances. In this experiment, we combine two consistency algorithms: CT and $lMaxRPC^{bit}$ [35] for the pf_{all} algorithm.

TABLE 2. **Results of comparing STRs and the best PSTRs.**

Series	kind	algorithm	cpu^s	cpu^p	ratio	#prop	algorithm	cpu^s	cpu^p	ratio	#prop
rand-3-20-fcd	serial	CT	15.539	14.762	1.000	122.487	STRbit	23.513	20.393	1.000	122.486
#=50	$best^p$	PCT^{ds}@16	12.680	10.373	**1.423**	213.805	$PSTRbit^{ds}$@7	17.926	13.501	**1.510**	207.024
rand-3-20	serial	CT	26.751	25.452	1.000	210.507	STRbit	40.512	35.278	1.000	210.506
#=50	$best^p$	PCT^{ds}@16	21.089	17.250	**1.475**	365.775	$PSTRbit^{ds}$@7	31.371	23.713	**1.488**	352.409
rand-3-24-fcd	serial	CT	74.866	71.233	1.000	504.254	STRbit	112.086	101.516	1.000	504.253
#=17	$best^p$	PCT^{ds}@16	47.270	37.061	**1.922**	876.489	$PSTRbit^{ds}$@16	63.559	46.119	**2.201**	893.402
rand-5-12	serial	CT	0.859	0.807	1.000	2.402	STRbit	1.494	1.342	1.000	2.400
#=50	$best^p$	PCT^{ds}@16	0.325	0.227	**3.555**	5.060	$PSTRbit^{ds}$@16	0.490	0.349	**3.845**	5.018
rand-8-20	serial	CT	4.905	4.546	**1.000**	32.869	STRbit	14.510	11.908	1.000	32.869
#=20	$best^p$	PCT^{ds}@3	6.631	5.635	0.807	50.381	$PSTRbit^{ds}$@5	13.299	9.702	**1.227**	53.171
rand-10-20	serial	CT	0.499	0.476	**1.000**	1.348	STRbit	1.299	1.199	1.000	1.348
#=20	$best^p$	PCT^{ds}@2	0.625	0.557	0.855	2.104	$PSTRbit^{ds}$@3	0.998	0.848	**1.414**	2.384
rand-10-60	serial	CT	9.409	9.126	1.000	4.645	STRbit	27.267	26.304	1.000	4.645
#=32	$best^p$	PCT^{ds}@16	3.154	2.657	**3.435**	9.897	$PSTRbit^{ds}$@7	9.632	8.639	**3.045**	10.455
half	serial	CT	62.683	57.529	1.000	412.813	STRbit	213.962	179.289	1.000	412.812
#=20	$best^p$	PCT^{ds}@3	60.043	49.570	**1.161**	673.176	$PSTRbit^{ds}$@5	123.065	80.266	**2.234**	692.172
MDD0.7	serial	CT	37.345	34.321	1.000	146.049	STRbit	120.665	71.820	1.000	146.048
#=8	$best^p$	PCT^{ds}@7	27.011	19.738	**1.739**	250.277	$PSTRbit^{ds}$@7	51.585	33.469	**2.146**	258.706
MDD0.9	serial	CT	1.851	1.712	1.000	8.247	STRbit	6.480	3.972	1.000	8.247
#=9	$best^p$	PCT^{ds}@7	1.469	1.103	**1.552**	14.373	$PSTRbit^{ds}$@7	2.674	1.783	**2.228**	14.907
uk	serial	CT	10.336	10.101	1.000	7.843	STRbit	16.629	15.715	1.000	7.842
#=42	$best^p$	PCT^{ds}@16	3.588	3.133	**3.224**	12.774	$PSTRbit^{ds}$@16	5.511	4.304	**3.651**	14.038
ogd	serial	CT	5.973	5.852	1.000	2.353	STRbit	6.640	6.264	1.000	2.353
#=43	$best^p$	PCT^{ds}@7	1.672	1.484	**3.943**	3.845	$PSTRbit^{ds}$@16	2.416	1.943	**3.224**	4.836
lex	serial	CT	1.448	1.408	1.000	3.978	STRbit	1.969	1.802	1.000	3.978
#=63	$best^p$	PCT^{ds}@3	1.081	0.982	**1.434**	5.735	$PSTRbit^{ds}$@3	1.440	1.180	**1.527**	5.737
words	serial	CT	4.418	4.292	1.000	10.769	STRbit	6.151	5.618	1.000	10.768
#=65	$best^p$	PCT^{ds}@3	3.198	2.879	**1.491**	15.515	$PSTRbit^{ds}$@3	4.352	3.544	**1.585**	15.535
modR	serial	CT	0.306	0.122	**1.000**	1.727	STRbit	1.039	0.168	**1.000**	1.726
#=26	$best^p$	PCT^{ds}@2	0.585	0.322	0.379	2.459	$PSTRbit^{ds}$@2	1.351	0.360	0.467	2.470
cril	serial	CT	190.421	185.812	1.000	183.374	STRbit	303.654	292.989	1.000	183.369
#=2	$best^p$	PCT^{ss}@7	36.940	30.442	**6.104**	137.687	$PSTRbit^{ss}$@13	46.464	34.577	**8.474**	146.625
aim-50	serial	CT	0.125	0.056	**1.000**	1.808	STRbit	0.245	0.050	**1.000**	1.807
#=24	$best^p$	PCT^{ds}@2	0.249	0.155	0.361	2.089	$PSTRbit^{ds}$@2	0.408	0.164	0.305	2.087
aim-100	serial	CT	3.544	0.937	**1.000**	33.765	STRbit	7.752	0.932	**1.000**	33.761
#=17	$best^p$	PCT^{ds}@2	6.966	3.529	0.266	38.569	$PSTRbit^{ds}$@2	11.882	3.593	0.259	38.588
aim-200	serial	CT	6.179	1.916	**1.000**	51.329	STRbit	13.017	1.779	**1.000**	51.318
#=8	$best^p$	PCT^{ds}@7	10.224	3.522	0.544	91.666	$PSTRbit^{ds}$@13	18.847	3.661	0.486	87.983
tsp-20	serial	CT	2.340	2.152	1.000	30.770	STRbit	3.576	2.633	1.000	30.769
#15	$best^p$	PCT^{ds}@16	2.047	1.570	**1.371**	37.085	$PSTRbit^{ds}$@7	3.619	2.148	**1.226**	37.074
tsp-25	serial	CT	20.652	18.620	1.000	265.080	STRbit	31.749	22.626	1.000	265.078
#=15	$best^p$	PCT^{ds}@16	14.519	10.362	**1.797**	311.875	$PSTRbit^{ds}$@7	29.570	16.008	**1.413**	311.809
jnhSat	serial	CT	1.047	0.564	**1.000**	11.504	STRbit	2.280	0.562	**1.000**	11.496
#=16	$best^p$	PCT^{ds}@9	1.447	0.635	0.888	16.452	$PSTRbit^{ds}$@7	2.937	0.667	0.843	16.473
jnhUnsat	serial	CT	0.403	0.224	**1.000**	4.393	STRbit	0.896	0.237	**1.000**	4.385
#=34	$best^p$	PCT^{ds}@7	0.514	0.228	0.982	6.254	$PSTRbit^{ds}$@7	1.084	0.247	0.960	6.251

Both algorithms are bit-based algorithms, and $lMaxRPC^{bit}$ has stronger consistency. To be fair, as with PCT^{ds}@5, we also use 5 threads to perform pf_{all}. One thread is the main thread that executes the backtracking search emended by CT, one thread executes the $lMaxRPC^{bit}$ synchronously, and the other three threads execute the $lMaxRPC^{bit}$ asynchronously.

From Table 3, we find that pf_{all} has the best results on many instances. There are two main reasons: On the one hand, pf_{all} combines the strong consistency algorithm $lMaxRPC^{bit}$, which can prompt the main thread to delete the invalid value and exit from the invalid search subtree. This greatly improves the pruning ability. Thus the algorithm traverses fewer nodes. On the other hand, the CT algorithm executed in the main thread interrupts other threads in time, so that the whole solving process does not fall into a long-time inference due to the strong consistencies. This ensures that the algorithm has an upper bound of the CT algorithm solving time, so the solving time of the pf_{all} algorithm is typically the shortest. Due to the above two reasons, the timeout rate of the pf_{all} algorithm for solving grouped instances is also minimal. In addition, since CT and PCT have the same consistency, the number of traversal nodes is the same. Overall, the PCT is slightly better than CT, but it does not sound massive. Another finding is that CT and PCT perform better on the instances that have similar number of traversal nodes in the three algorithms, such as rand-2-30-15(-fcd), ewddr2 and scen7. This is because AC is easier to achieve than lMaxRPC. If the number of traversal nodes is similar, PCT and CT algorithms are faster than pf_{all}.

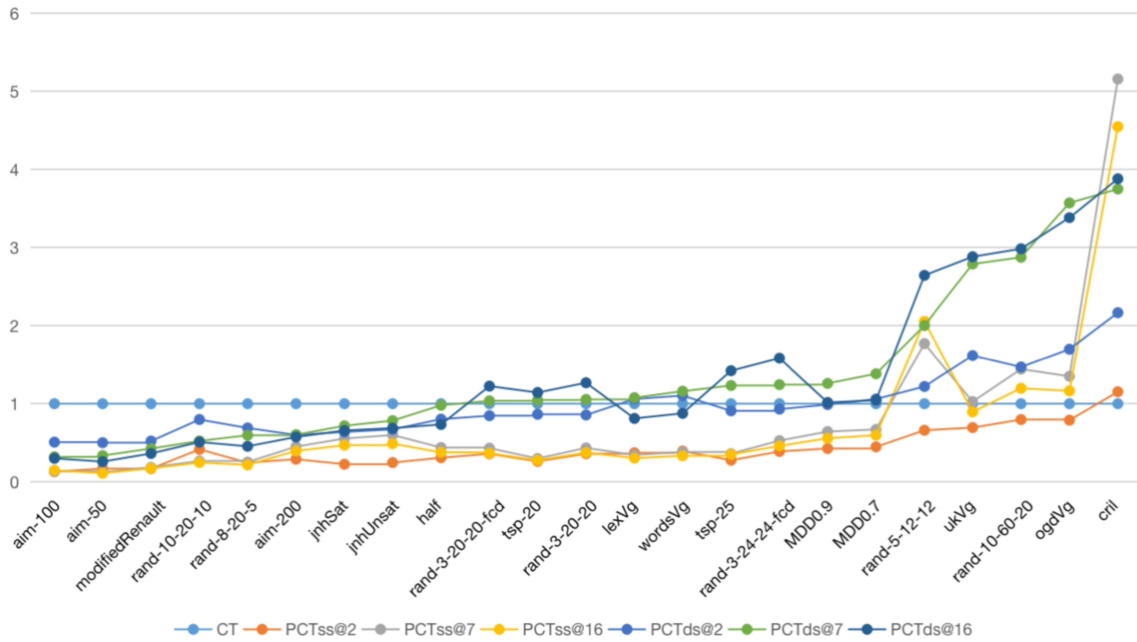


FIGURE 5. The comparison of speed-up ratio of solving time for series instances.

TABLE 3. Results on binary constraint instances with dom/ddeg.

Instances		CT	$PCT^{ds}@5$	pf_{all}
Series instances				
BH-4-4	time	4.909	5.206	**0.752**
	#nodes(M)	0.491	0.491	0.026
	TO(ratio)	0	0	0
QCP-15	time	316.642	293.551	**26.869**
	#nodes(M)	5.274	5.274	0.322
	TO(ratio)	0.6	0.6	0
bqwh-18-141	time	125.351	153.352	**7.204**
	#nodes	9.59	9.59	0.208
	TO(ratio)	0.06	0.07	0
coloring	time	3.914	3.781	**1.699**
	#nodes(M)	0.154	0.154	0.031
	TO(ratio)	0.05	0.05	0.05
driver	time	30.830	33.750	**5.155**
	#nodes(M)	0.090	0.090	0.011
	TO(ratio)	0.14	0.14	0
queensKnights	time	196.139	90.999	**73.009**
	#nodes(M)	10.887	10.887	2.868
	TO(ratio)	0.6	0.6	0.5
rand-2-30-15-fcd	time	0.738	**0.605**	1.096
	#nodes(M)	0.041	0.041	0.036
	TO(ratio)	0	0	0
rand-2-30-15	time	1.547	**1.229**	1.908
	#nodes(M)	0.079	0.079	0.061
	TO(ratio)	0	0	0
Singleton instances				
ewddr2-10-by-5-1	time	0.061	**0.02**	0.077
	#nodes	50	50	50
graph10	time	timeout	timeout	**1.18**
	#nodes	-	-	694
scen11	time	105.572	71.982	**36.524**
	#nodes	32054	32054	12223
scen7-w1-f4	time	0.034	**0.031**	0.138
	#nodes	426	426	424

VI. CONCLUSION

There has been a lot of work on parallel constraint programming for parallel search tree and problem decomposition. In this paper, we illustrate the feasibility of research in a different direction, i.e., parallel propagation, by revisiting the parallel consistency model. We propose two (parallel) propagation schemes: static and dynamic, which utilize work-stealing thread pool and atomic operations to improve the efficiency of propagation. In addition, to adapt the two parallel propagation schemes, we improve the STR algorithms and propose two parallel STR algorithms: $PSTR^{ss}$ and $PSTR^{ds}$. Our extensive experiments show that both $PSTR^{ss}$ and $PSTR^{ds}$ outperform serial propagation scheme on most of the larger non-binary table constraint instances, which sufficiently illustrates the potential of parallel propagation. In the future, we will combine the proposed parallel propagation schemes with stronger consistencies to improve the solving efficiency. And we will apply these schemes to parallel search tree and problem decomposition instead of the serial version.

REFERENCES

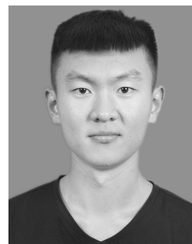
[1] L. Christophe, "Introduction," in *Constraint Networks: Techn. Algorithms*, 1st ed. London, U.K: Wiley, 2009, sec. 1, pp. 27–28.
[2] F. Rossi, P. Van Beek, and T. Walsh, "Constraint satisfaction: An emerging paradigm," in *Handbook Constraint Programming*, 1st ed. Amsterdam, The Netherlands: Elsevier, 2006, sec. 1, pp. 14–16.
[3] *GECODE—An Open, Free, Efficient Constraint Solving Toolkit*. Accessed: Apr. 26, 2019. [Online]. Available: https://www.gecode.org/
[4] C. Prud'homme, J. G. Fages, and X. Lorca, *Choco Solver Documentation*, document TASC LS2N CNRS UMR 6241, COSLING SAS, 2017.
[5] OscaR Team. (2012). *OscaR: Scala OR*. [Online]. Available: https://bitbucket.org/oscarlib/oscar
[6] C. Schulte, "Parallel search made simple," in *Proc. TRICS*, Singapore, Sep. 2000, pp. 41–57.
[7] C. Lecoutre, "STR_2: Optimized simple tabular reduction for table constraints," *Constraints*, vol. 16, no. 4, pp. 341–371, Oct. 2011.
[8] C. Lecoutre, C. Likitvivatanavong, and R. Yap, "A path-optimal GAC algorithm for table constraints," in *Proc. ECAI*, Montpellier, France, 2012, pp. 510–515.

[9] J. B. Mairy, P. Van Hentenryck, and Y. Deville, "Optimal and efficient filtering algorithms for table constraints," *Constraints*, vol. 19, no. 1, pp. 77–120, Jan. 2014.
[10] G. Perez and J.-C. Régin, "Improving GAC-4 for table and MDD constraints," in *Proc. CP*, Lyon, France, 2014, pp. 606–621.
[11] R. Wang, W. Xia, R. H. Yap, and Z. Li, "Optimizing simple tabular reduction with a bitwise representation," in *Proc. IJCAI*, New York, NY, USA, Jul. 2016, pp. 787–795.
[12] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J.-C. Régin, and P. Schaus, "Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets," in *Proc. Int. Conf. Princ. Pract. Constraint Program.*, 2016, pp. 207–223.
[13] H. Verhaeghe, C. Lecoutre, Y. Deville, and P. Schaus, "Extending compact-table to basic smart tables," in *Proc. CP*, Melbourne, VIC, Australia, 2017, pp. 297–307.
[14] H. Verhaeghe, C. Lecoutre, and P. Schaus, "Extending compact-table to negative and short tables," in *Proc. AAAI*, San Francisco, CA, USA, Feb. 2017, pp. 3951–3957.
[15] H. Verhaeghe, C. Lecoutre, and P. Schaus, "Compact-MDD: Efficiently filtering (s) MDD constraints with reversible sparse bit-sets," in *Proc. IJCAI*, New Orleans, LA, USA, 2018, pp. 1383–1389.
[16] L. Ingmar and C. Schulte, "Making compact-table compact," in *Proc. CP*, Lille, France, 2018, pp. 210–218.
[17] S. Kasif, "On the parallel complexity of discrete relaxation in constraint satisfaction networks," *Artif. Intell.*, vol. 45, no. 3, pp. 275–286, Oct. 1990.
[18] I. P. Gent, I. Miguel, P. Nightingale, C. Mccreesh, P. Prosser, N. C. A. Moore, and C. Unsworth, "A review of literature on parallel constraint solving," *Theory Pract. Logic Program.*, vol. 18, nos. 5–6, pp. 725–758, Sep. 2018.
[19] L. Michel, A. See, and P. Van Hentenryck, "Parallelizing constraint programs transparently," in *Proc. CP*, Providence, RI, USA, 2007, pp. 514–528.
[20] T. Nguyen and Y. Deville, "A distributed arc-consistency algorithm," *Sci. Comput. Program.*, vol. 30, nos. 1–2, pp. 227–250, Jan. 1998.
[21] A. Ruiz-Andino, L. Araujo, F. Sáenz-Pérez, and J. J. Ruz, "Parallel arc-consistency for functional constraints," in *Proc. ICLP*, Manchester, U.K., 1998, pp. 86–100.
[22] V. C. le de Saint-Marcq, P. Schaus, C. Solnon, and C. Lecoutre, "Sparse-sets for domain implementation," in *Proc. TRICS*, Uppsala, Sweden, 2013, pp. 1–10.
[23] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, "Boosting systematic search by weighting constraints," in *Proc. ECAI*, Valencia, Spain, 2004, pp. 146–150.
[24] L. Michel and P. Van Hentenryck, "Activity-based search for black-box constraint programming solvers," in *Proc. CPAIOR*, Nantes, France, 2012, pp. 228–243.
[25] C. C. Rolf and K. Kuchcinski, "Parallel consistency in constraint programming," in *Proc. PDPTA*, vol. 9, 2009, pp. 638–644.
[26] C. C. Rolf and K. Kuchcinski, "Combining parallel search and parallel consistency in constraint programming," in *Proc. TRICS*, St Andrews, Scotland, Sep. 2010, pp. 38–52.
[27] D. Lea, "A java fork/join framework," in *Proc. Java Grande*, Jun. 2000, pp. 36–43.
[28] *Fork Join Pool (Java Platform SE 8)*. Accessed: May. 8, 2019. [Online]. Available: https://dwz.cn/o6kZmDPR
[29] M. Koubarakis, "Temporal CSPs," in *Handbook Constraint Programming*, 1st ed. Amsterdam, The Netherlands: Elsevier, 2006, sec. 19, pp. 663–667.
[30] L. Christophe, "Generic GAC algorithm," in *Constraint Networks: Techniques and Algorithms*, 1st ed. London, U.K.: Wiley, 2009, sec. 4, pp. 204–205.
[31] *Christophe Lecoutre - Home Page*. Accessed: May 28, 2019. [Online]. Available: http://www.cril.univ-artois.fr/ lecoutre/index.html#/
[32] *XCSP3*. Accessed: May 28, 2019. [Online]. Available: http://www.xcsp.org/
[33] A. Schneider and B. Y. Choueiry, "PW-CT: Extending compact-table to enforce pairwise consistency on table constraints," in *Proc. CP*, Lille, France, 2018, pp. 345–361.
[34] M. Dasygenis and K. Stergiou, "Methods for Parallelizing Constraint Propagation through the Use of Strong Local Consistencies," *Int. J. Artif. Intell. Tools*, vol. 27, no. 4, Jun. 2018, Art. no. 1860002.
[35] J. Guo, Z. Li, L. Zhang, and X. Geng, "Max RPC algorithms based on bitwise operations," in *Proc. CP*, Perugia, Italy, 2011, pp. 373–384.

ZHE LI is currently pursuing the D.S. degree in software engineering with Jilin University, Jilin, China. He is also a member of the Key Laboratory for Symbol Computation and Knowledge Engineering of National Education Ministry, China. His research interests include the constraint problem and parallel computing.

ZHEZHOU YU received the Ph.D. degree from Jilin University, in 2007. He is currently a Professor with Jilin University. He is also a member of the Key Laboratory for Symbol Computation and Knowledge Engineering of National Education Ministry, China. His research interests include computational intelligence and embedded system application. He is a Committee Member of the Undergraduate Electronic Design Competition Organization of Jilin Province, China.

PENG WU is currently an Engineer with the Technology and Engineering Center for Space Utilization of the Chinese Academy of Sciences. His research interest includes planning and constraining reasoning.



JIANAN CHEN received the B.S. degree in computer science and technology from Jilin University, Jilin, China, in 2018, where he is currently pursuing the M.S. degree in software engineering. He is also a member of the Key Laboratory for Symbol Computation and Knowledge Engineering of National Education Ministry, China. His research interests include the development of combinatorial optimization problem and parallel computing.

ZHANSHAN LI is currently a Professor of computer science with Jilin University. He is also a member of the Key Laboratory for Symbol Computation and Knowledge Engineering of National Education Ministry, China. His main research interests include constraint reasoning and machine learning.