# A Multi-thread Parallel Constraint Propagation Scheme

## Abstract

In constraint programming, the process of reducing the search space is called constraint propagation. A constraint propagation (or filtering algorithm) is usually described in terms of (local) consistencies, such as (generalized) arc consistency (GAC), path consistency (PC), singleton arc consistency(SAC) and so on. Consistency algorithms often maintain a propagation queue, which restores elements needed to enforce GAC. For traditional serial constraint propagation scheme, the queue pops elements and then tuples and domains would be filtered iteratively until the queue is empty or inconsistency occurs. With the ubiquity of multi-core architectures, it attracts great interests of constraint researchers. In this paper, we present a non-blocking parallel constraint propagation scheme based on multi-thread technology. Then we apply this scheme to Compact-Table (CT) and STR2, the well-known GAC algorithms, and propose Compact-Table$^p$ (CT$^p$) and STR2$^p$. It can enforce parallel GAC on multi-core architectures, and significantly benefit from parallel computing ability of modern processor. Experiment results show the competitiveness of parallel scheme. Furthermore, it outperforms original algorithms in many problems with large arity and domain size.

## Introduction

In the process of seeking solution for a constraint satisfaction problem(CSP), local consistency is one of the most common ways to reduce the search space. Recently, many consistencies have been presented, e.g., (generalized) arc consistency (GAC), path consistency(PC) (Stergiou 2015), singleton arc consistency (SAC) (Wallace 2016) (Paparrizou and Stergiou 2017). Among these consistencies, generalized arc consistency (GAC), which replaces AC on non-binary constraint, is considered as the most popular local consistency, which is a trade-off between the ability of pruning values and algorithm complexity. Backtracking search is a complete method to solve CSP instances, where systematic exploration finds the full set of solutions or proves that no solution exists. Within backtracking search, depth-first exploration instantiates variables and executes backtracking when reaches the dead-ends. In this process, a search tree is developed from level 0 to level $n$, where $n$ is the number of variables. Maintaining arc consistency (MAC) is the backtrack search algorithm that enforces GAC before and during search.

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, and besides, they can be viewed as a universal mechanism for representing any constraints, although they may cause combinatorial explosion of time and space. For decades, many filtering algorithms have thus been proposed for enforcing Generalized Arc Consistency (GAC) on table constraints(Lecoutre 2011) (Lecoutre, Likitvivatanavong, and Yap 2012) (Mairy, Van Hentenryck, and Deville 2014) (Perez and Régin 2014) (Wang et al. 2016) (Demeulenaere et al. 2016). Compact-Table (Demeulenaere et al. 2016) and its extensions (Verhaeghe et al. 2017) (Verhaeghe, Lecoutre, and Pierre 2017) (Verhaeghe, Lecoutre, and Schaus 2018) are considered as the most efficient algorithms at the moment. In modern object-oriented constraint solver, constraint is implemented as a subclass of $propagator$ (Ingmar and Schulte 2018). The filtering algorithm is implemented as a member function of $propagator$. Thus, enforcing GAC on a constraint model is an iterative process of calling each relevant propagator's filter function. To simplify the description, we use $c.enforceGAC()$ to denote enforce GAC on certain constraint $c$.

As shown in (Kasif 1990), the problem of establishing arc consistency is P-complete, which is not inherently parallelisable 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 (Gent et al. 2011). Parallel constraint programming can be roughly divided into two aspects: parallel search (Schulte 2000) (Michel, See, and Van Hentenryck 2007) and parallel consistency (Nguyen and Deville 1998) (Ruiz-Andino et al. 1998). On one hand, as shown in figure 1(a), parallel search engine assigns the different parts of the search tree to processors (or other concurrent processing entities, e.g. threads and agents). Thereupon, the model data can be split into different processors. On the other hand, as shown in figure 1(b), each constraint can be checked independently of each other on different processors, as long as their filtering is synchronized. Because of the independence of filtering between different processors, it may have to perform extra iterations of consistency.

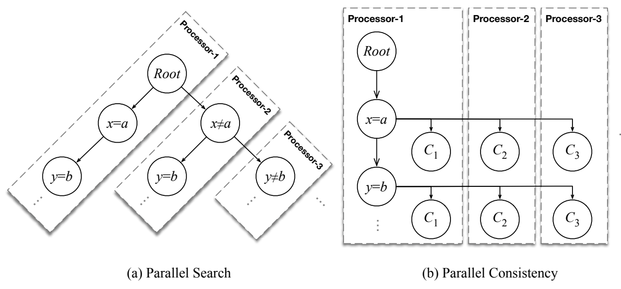


Figure 1: Parallel constraint programming

In this paper we are interested in parallelizing and optimizing the original constraint propagation scheme. We improve the time-stamps labelling method so that it can parallelize the serial constraint propagation scheme on multi-core processors and reorganize the parallel propagation flow to avoid a lot of blocking due to synchronization. Next we apply this parallel scheme to current popular table constraint reduction algorithm - Compact-Table and STR2, and propose Compact-Table$^p$ (CT$^p$) and STR2$^p$ respectively. In the experiment, we first compare the performance of CT to CT$^p$ with different parallelism on some instances of table constraint problem with large size of arity and domain. Then we evaluate STR2 and STR2$^p$ on the *factor-decomposition encoding* (FDE) for CSP instance. The results show the competitiveness.

The rest of this paper is organized as follows. Firstly some background issues are introduced in section 2. Then, in section 3, we summarize the classical STR framework to enforce GAC. Next, we show the pseudo-code of parallel schedule and filter algorithms in section 4. Afterward, in section 5, we implement the schemes and compare them in different parallelism to original algorithms on ordinary model and FDE model. Finally, we conclude this paper in section 6.

## 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 $\mathcal{X} = \{x_1, x_2 \ldots x_n\}$, $\mathcal{D}$ is a set of domains $\mathcal{D} = \{D(x_1), D(x_2) \ldots D(x_n)\}$ where $D(x_i)$ is a finite set of possible values for variable $x_i$, $\mathcal{C}$ is a set of $e$ constraints $\mathcal{C} = \{c_1, c_2 \ldots c_e\}$.

A constraint $c$ consists of two parts, an ordered set of variables $scp(c) = \{x_{i_1}, x_{i_2} \ldots x_{i_r}\}$ and a subset of the Cartesian product $D(x_{i_1}) \times D(x_{i_2}) \times \ldots \times D(x_{i_r})$ that specifies the allowed combinations of values for $scp(c)$, denoted by $rel(c)$. Let $\tau = \{a_1, a_2, ..., a_r\} \in rel(c)$ be an r-ary tuple, and the value of variable $x_{i_j}$ of $\tau$ is denoted by $\tau[x_{i_j}]$. We say that $\tau$ is valid on $c$ iff $\forall x \in scp(c)$, $\tau[x] \in D(x)$. If $\tau$ is a support on a constraint $c$ involving a variable $x$ and such that $\tau[x] = a$, we say that $\tau$ is a support for $(x, a)$ on $c$. 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 $(x_i,\ a)$ is generalized arc consistent, or GAC-consistent iff it has at least one support in each constraint involving $x_i$, 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 to propagate the assignment. A dead-end is reached if the propagation fails, then backtracking occurs.

In (Rolf and Kuchcinski 2009) and (Rolf and Kuchcinski 2010), Rolf and Kuchcinski presented a parallel consistency model (depicted in figure 2). Parallel consistency checking is done by waking the consistency threads available to the constraint program. These threads will then retrieve work from $Q$ (the queue of constraints) whose variables have changed. Once all tasks in $Q$ have been parallel processed, all prunings are committed to solver. If there were no changes to any variable, a fix-point has reached and the solver continues to search. If an inconsistency is detected in some thread, 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.

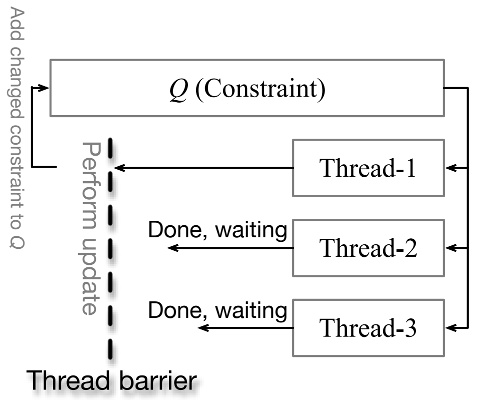


Figure 2: The execution model for parallel consistency

## STR Framework

For many object-oriented constraint solvers, e.g., Gecode, a constraint problem can be modelled as an object of Gecode *home* class, which is constitutive of array of variables, propagators (implementation of constraints) and branchers (implementation of branching). A table constraint, which is *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 member function of $propagator$ and the schedule method that dispatches filtering functions.

Figure 3 briefly shows a flow diagram of propagation schedule method. The schedule method maintains a propagation queue to restore the elements which need to propagate. The data structure of $Q$ can be array or heap. When $Q$ pops a variable $x$, the function needs to iteratively call $c.enforceGAC()$ for all $c \in srb(x)$, prune the GAC-inconsistent values depending on the remaining tuples of $c$, detect inconsistency and push back the variables whose domains have been reduced, denoted by $X_{evt}$.

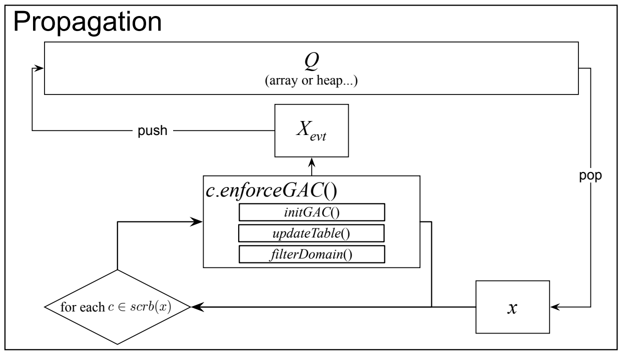


Figure 3: A flow diagram of propagation schedule method

Some unnecessary work can be avoided by using 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 allow the progress of algorithms to be tracked over time. The basic idea of time-stamp is presented as 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 incremented and then 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. $stamp[c] > stamp[x]$.

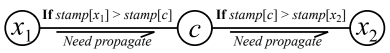


Figure 4: Time-stamp

Algorithm 1 gives a general description of STR-based $propagator$ class (for more detailed versions, see (Demeulenaere et al. 2016)). When the users build the constraint problem model, the $constructor$ function is called in which most data structures of GAC algorithm are initialized. The function $enforceGAC()$ is an abstract function that must be overridden by all subclasses of $propagator$. After initialization of $S^{val}$, $S^{sup}$ and $lastSize$ in $initGAC()$ at line 5, the function $updateTable()$ is called to reduce the set of tuples which are used to filter the variables' domain in the function $filterDomain()$. The function returns false if it detects all tuples become invalid (TWO, for short, line 10) or domain is wiped out (DWO, for short, line 13).

**Algorithm 1:** class STR Propagator

```
1  Method updateTable():
2      ...
3  Method filterDomain():
4      ...
5  Method initGAC():
6      /*Generation of $S^{sup}$, $S^{val}$ and $lastSize[x]$, where
         $x \in scp(c)$*/
7  Method enforceGAC():
8      $initGAC()$
9      $updateTable()$
10     if TWO is detected then
11         return false
12     $filterDomain()$
13     if DWO is detected then
14         return false
15 Method constructor($scp(c)$: subset of $\mathcal{X}$, tuples:
     tuple set of c, ...):
16     /*Initialization of the data structure of Propagator*/
```

## Multi-thread Constraint Propagation Scheme

In this section, we first succinctly explain the work principle of the work-stealing thread pool. It is wildly used in parallel constraint programming (Michel, See, and Van Hentenryck 2007) (Chu, Schulte, and Stuckey 2009) and regarded as a basic parallel scheduling mechanism of the propagation in this paper. Then, we introduce the data structures applied in our parallel constraint propagation scheme. Finally, we present parallel STR algorithms, called STR$^p$, and show how the multi-thread propagation scheme works.

In order to parallelize constraint propagation, we use a work-stealing pool as the scheduling strategy where worker threads that have finished their own tasks can steal pending tasks from other threads. In parallel propagation, we deploy the constraint filtering function to the work-stealing pool and then the tasks are dispatched among multi-core processors. When a thread has no work, it should be assigned a task from overloaded queue of another thread rather than being idle.

In this paper, we simply use the $newWorkStealingPool$ in concurrency package, which is introduced in Java 8, as the basic thread scheduling mechanism. Java gives a very succinct definition of this pool as:

> "Creates a work-stealing thread pool using all available processors as its target parallelism level."

In our implementation, we create all the tasks (i.e. $updateTable()$ or $filterDomains()$) simultaneously and dispatch them to a thread pool with a fixed number of threads, called the capacity of $pool$. If one thread has finished its works, it can "steal" work from others.

Now we introduce some data structures used in this paper:

$pool$ is a work-stealing thread pool, its member function $invokeAll(\mathcal{S})$ concurrently executes the tasks in task set $\mathcal{S}$and blocks the program until all tasks have been completed.

$PropsOfVar$ restores a variable $x$ and propagators whose table has been updated since the previous invocation. These propagators are used to filter the domain of $x$.

$P_{evt}$: In algorithm 2 and 3, we maintain a map $P_{evt}$ instead of $S^{sup}$, which is a collection of variable-$PropsOfVar$ pairs whose key is a variable and corresponding value is an object of $PropsOfVar$.

$C_{evt}$: In section 2, we have mentioned $X_{evt}$, a set of variables whose domains have been reduced. Correspondingly, we can obtain $C_{evt}$, a set of constraints whose tuples have been reduced according to $X_{evt}$. It is used to deploy every $c.updateTable()$ , where $c \in C_{evt}$, to the thread pool.

$STR^p$ is an improve version of STR-based $propagator$ class. We make minor changes to $propagator$ so that it can be invoked by the thread pool. Instead of $filterDomain()$ for all $x \in S_{sup}$ in STR-based $propagator$, we decompose this process into $filterDomain(x)$ so that it only filters domain of $x$ and can avoid blocking caused by filtering the same domain simultaneously by multiple threads.

## Parallel Propagation

Before backtracking search or after taking a positive or negative decision, the search algorithm calls $propagate^p$ to enforce parallel GAC on constraint network $\mathcal{P}$. As described in algorithm 2, it maintains two global variables: $failed$ and $finished$, which represent propagation failure and success respectively. Both of them can be accessed by any thread. Once $finished$ is set to $true$, it exits the loop and returns $true$ meaning that it satisfies GAC. If TWO/DWO is detected, $failed$ is set to $true$, and then the algorithm returns $false$. At the beginning of loop, the algorithm resets $finished$, $X_{evt}$, $C_{evt}$ and $P_{evt}$. If the domain of variable has been reduced within $filterDomain()$ in certain thread, $finished$ is set to $false$ (at line 8, algorithm 3), and the loop continues. At line 6, the algorithm iterates all $x \in \mathcal{X}$ and initializes $X_{evt}$. If $stamp[x]$ equals to $stamp$, which means the domain of $x$ has been changed since previous invocation, and $x$ needs to be added to $X_{evt}$. After that, it iterates over each constraint $c$ whose scope contains at least one variable in $X_{evt}$ to invoke $c.initGAC()$ and add $c$ to $C_{evt}$.

At line 15, the set $C_{evt}$ is submitted to $pool$. For each $c \in C_{evt}$, its $c.updateTable()$ will be scheduled and executed through the $pool$. After that, all the tables in $C_{evt}$ have been updated. At line 19, it submits $P_{evt}$ to $pool$. For each element of $P_{evt}$, a variable-$PropsOfVar$ pair actually, it calls $filterDomain()$ at line 1 in algorithm 4, which literately calls $c.filterDomain(x)$ , where $c \in Q_p$, for filtering the domain of variable $x$. If TWO/DWO is detected, $failed$ is set to $true$ (line 15 and line 19) and algorithm returns false (line 17 and line 21). At line 14 and 18, the global timestamp increases by 1. So every $stamp[c]$, where $c \in C_{evt}$, is updated to the latest global $stamp$ at line 3 in algorithm 3, and every $stamp$ of variable $x$ whose domain has been reduced is updated too at line 7 in algorithm 3. The algorithm

**Algorithm 2:** $propagate^p(P)$

```
1  while $\neg finished$ do
2      $finished \leftarrow true$
3      $X_{evt} \leftarrow \emptyset$
4      $P_{evt}.clear()$
5      $C_{evt} \leftarrow \emptyset$
6      foreach $x \in \mathcal{X} \mid stamp[x] = stamp$ do
7          /* initialization of $X_{evt}$ */
8          $X_{evt} \leftarrow X_{evt} \cup \{x\}$
9      foreach constraint $c \in srb(x) \mid x \in X_{evt}$ do
10         /* initialization of $P_{evt}$ */
11         $c.initGAC()$
12         /* initialization of $C_{evt}$ */
13         $C_{evt} \leftarrow C_{evt} \cup \{c\}$
14     $stamp \leftarrow stamp + 1$
15     $pool.invokeAll(C_{evt}.updateTable())$
16     if $failed$ then
17         return $false$
18     $stamp \leftarrow stamp + 1$
19     $pool.invokeAll(P_{evt}.filterDomain())$
20     if $failed$ then
21         return $false$
22 return $true$
```

always keeps $stamp$s unbalanced between constraints and variables so that it can propagate constantly.Note that, at line 15 and 19, each table of $c \in C_{evt}$ and each domain of key $x \in P_{evt}$ are modified independently through one thread. Therefore, it's unnecessary to block the process for synchronization, so called "non-blocking scheme" in abstract.

Figure 5 intuitively illustrates the parallel propagation scheme. Firstly, $C_{evt}$ and $P_{evt}$ are obtained through $X_{evt}$. Next, $C_{evt}$ is submitted to $pool$ to execute $updateTable()$ concurrently, so it updates each table independently. Finally, $P_{evt}$ is submitted to $pool$ in order to invoke $filterDomain()$ for each key $x \in P_{evt}$, so it filters each domain independently. During $propagate^p$, the algorithm needs to detect TWO/DWO and update $stamp$s.

## class STR$^p$

We introduce propagator class STR$^p$ in algorithm 3, which is a subclass of $propagator$ and can be easily extend to STR-base algorithm, such as CT$^p$ and STR2$^p$.

In overriding function $initGAC()$, propagator additionally maintains a global map of propagator queue $P_{evt}$ instead of $S^{sup}$. For each $x \in scp(c)$, if $P_{evt}$ doesn't contain key $x$, it creates new key (can be accessed by $P_{evt}[x]$ (line 16)) and attaches a propagator queue/set within current propagator $c$ as the value of $P_{evt}[x]$(line 17). Otherwise it just directly adds $c$ to $P_{evt}[x]$ (line 19). The member function $updateTable()$ is the same as one of its superclasses except updates the $stamp[c]$ at line 3. Whereas we slightly modify function $filterDomain()$, we restrict that method only filters a certain variable $x$. In algorithm 4, via calling $PropsOfVar.filterDomain()$ for $x$, the function iterates

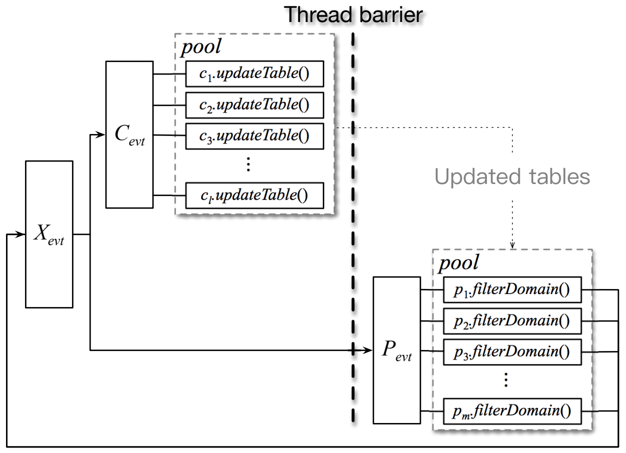


Figure 5: Parallel propagation scheme

**Algorithm 3:** class STR$^p$

```
Data: $P_{evt}$: A map of $variable$ - $PropsOfVar$ pair
1  Method updateTable():
2      ...
3      $stamp[c] \leftarrow stamp$
4  Method filterDomain($x : variable$):
5      ...
6      if $D(x)$ has been reduced then
7          $stamp[x] \leftarrow stamp$
8          $finished \leftarrow false$
9  Method initGAC():
10     $S^{val} \leftarrow \emptyset$
11     foreach $x \in scp(c)$ do
12         if $stamp[x] > stamp[c]$ then
13             $S^{val} \leftarrow S^{val} \cup \{x\}$
14             $lastSize[x] = |D(x)|$
15         if $x \notin P_{evt}$ then
16             add new key $x$ to $P_{evt}$
17             $P_{evt}[x].add(c)$
18         else
19             $P_{evt}[x].add(c)$
```

over $c \in Q_p$ and invokes $c.filterDomain(x)$ at line 2.

**Algorithm 4:** class $PropsOfVar$

```
Data: $x$: variable, $Q_p$: a set(queue) of $propagator$s in
      $srb(x)$ which need propagate
1  Method filterDomain():
2      foreach $c \in Q_p$ do
3          if $failed \wedge c.filterDomain(x)$ then
4              return $false$
5      return $true$
6  Method add($c : propagator$):
7      $Q_p \leftarrow Q_p \cup c$
8  Method clear():
9      $Q_p \leftarrow \emptyset$
```

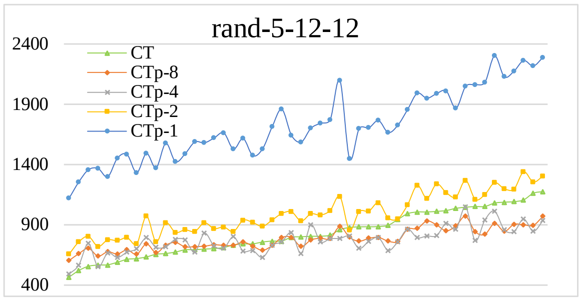
## Experiments

We first evaluate the performance of CT and CT$^p$ on original model, then compare STR2 to STR2$^p$ on FDE model. To make a comprehensive evaluation, we select the extensional instances from the XCSP3 website(Boussemart 2017). These benchmarks include large size of arity and domain so that can reflect the advantages of parallel computing. In addition, for generating FDE model, we implement a FDE model converter, which takes an input file in XCSP3 format. It generates the encoding (the generate algorithm is described in (Likitvivatanavong, Xia, and Yap 2015) ) and outputs its FDE model as a json file. To minimize the interaction between parallel consistency with search, we choose dom/ddeg instead of more efficient dom/wdeg as variable ordering heuristic and min value as value ordering heuristic. The experiments were performed on a 2.40GHz Intel CPU i7-4700HQ with 8GiB of RAM under 64bit-Linux. It is worth noting that all algorithms is written in Java10 and Scalar 2.12, as for the thread pool, we adopt $newWorkStealPool$ class in the Java concurrency package. Therefore, the final performance is closely related to the efficiency of $newWorkStealPool$.

The results for different series of instances are presented in figure 6 to 9. The $x$-axis values are the different cpu time (in millisecond) and the $y$-axis are instances sorted by solve time on original algorithm in ascending order, e.g., in figure 6, the plots present the solve time of each instance, and these lines denote the growth trends of each parallelism over the base algorithm (i.e. CT). The label CT$^p$-$n$ following a line in the legend of a chart indicates the number of active threads in the work-stealing thread pool applied to CT$^p$.

The results of CT$^p$ are presented in figure 6 to 8. In random instances, we can see that CT$^p$-1 has poorest performance, while CT$^p$-8 and CT$^p$-4 outperform others. In BDD instances, we find the classical CT$^p$-8 outperforms in front part of instances while CT is more efficient than others in the latter part of instances. Besides, it obtains a better acceleration by parallelizing the long-running $updateTable()$ and $filterDomain()$ for each constraint $c$. Because its domain size is 2, it has a short iteration over domain in function

$updateTable()$ and $filterDomain()$, which may be detrimental to the parallel approach. In crossword instances, CT is more efficient especially for ukVg, but the line of CT$^p$-8 approaches that of CT and even beat CT in some instances, and CT$^p$-1 ranks in the bottom. Apparently, in most instances, CT$^p$-4 is no less efficient than CT$^p$-8.



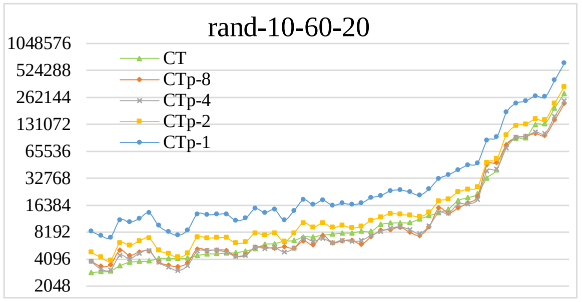
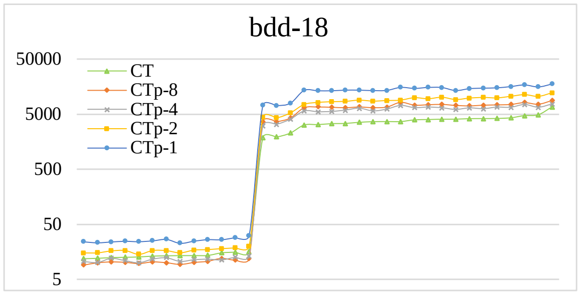


Figure 6: Comparing CT and CT$^p$ on random problems



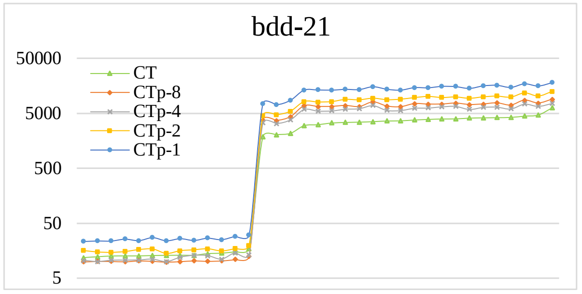
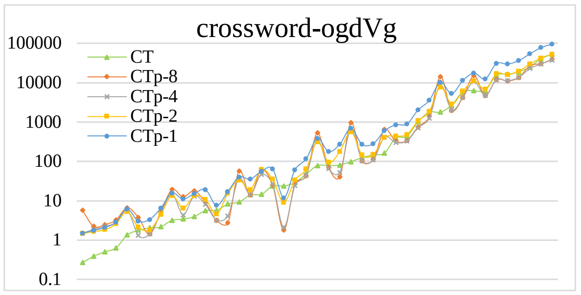


Figure 7: Comparing CT and CT$^p$ on BDD problems

When CT is used to represent the factor variable, it might encounter the memory overflow due to the explosion of



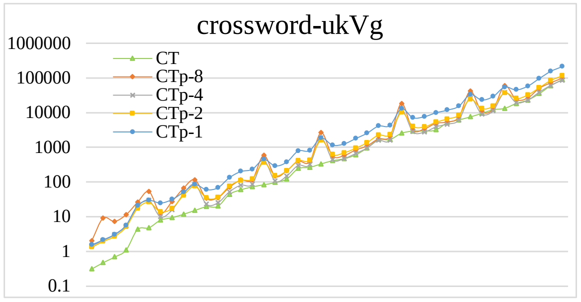


Figure 8: Comparing CT and CT$^p$ on crossword problems

the domain combination[1]. Therefore, we use STR2 and STR2$^p$ to evaluate the performance of parallel scheme on FDE model (Likitvivatanavong, Xia, and Yap 2015). FDE is an encoding of factor variables and creates new constraints including the factor variables. A FDE of constraints network $\mathcal{P}$ is denoted by $fde(\mathcal{P})$, thereby enforcing GAC on $fde(\mathcal{P})$ is equal to enforcing FPWC (Lecoutre, Paparrizou, and Stergiou 2013) on $\mathcal{P}$, for detailed proof, see (Likitvivatanavong, Xia, and Yap 2015). In figure 9, STR2$^p$-8 has the lowest efficiency in rand-5-10 and STR2$^p$-1 is just as bad in others. However, STR2$^p$-4 outperforms others in series of rand-3-20, rand-8-10 and rand-10-20, which supports our earlier observation. It is not the larger the capacity of the thread pool, the more efficient the execution. Because rand-10-20 does not satisfy FPWC, its FDE model does not satisfy AC. The reasoning algorithm quits with enforcing STR2 or STR2$^p$ one time before search, so the solve times in all instances are very close.

Combining the experiments with relative works, we get the following issues about our scheme:

- For most instances, the parallel algorithm with 1 parallelism has the worst efficiency. Properly increasing the parallelism might help improve the efficiency.
- A large capacity of thread pool is not necessarily efficient. Execution efficiency may depend on the number of tasks and the performance of *pool*.
- Due to the existence of threads barrier, CT$^p$-1 is not equal to CT and so is the STR2$^p$-1. Before threads arrive at the barrier, isolation of data may cause additional iterations.

[1]We are aware that a related work(Schneider and Choueiry 2018) has just been presented, which is an extension of CT to enforce FPWC. Since we mainly focus on parallel GAC in this paper, we do not parallelize it and count it into the experiment.

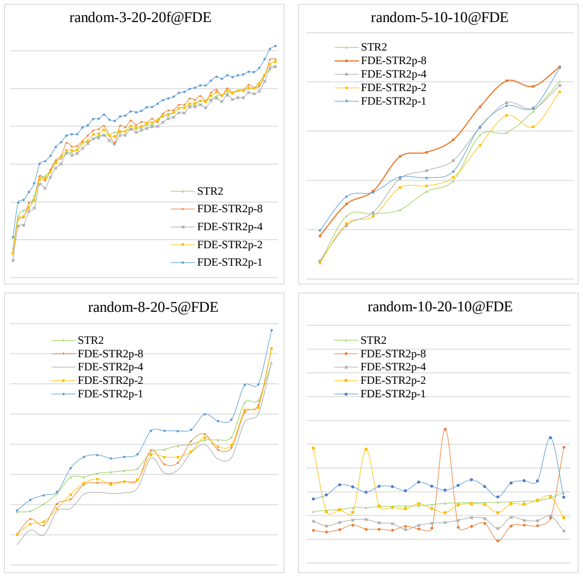


Figure 9: Comparing STR2 and STR2$^p$ on FDE models

## Conclusion

In this paper, we have introduced a multi-thread parallel propagation scheme. Compared to past parallel model, it avoids the blocking problem by splitting the $enforceGAC$ function into two stage and using the work-stealing thread pool for efficient scheduling. For enforcing parallel GAC on table constraint we develop CT$^p$ and STR2$^p$ by applying such scheme to STR2 and Compact-Table. In the experiments, we show the advantages and disadvantages of the parallel algorithm, and that it can alternate the original algorithm in some large instances. In addition, we also test higher-order parallel consistency algorithm STR2$^p$ on FDE model, the results show its competitiveness.

## References

Boussemart. 2017. Xcsp3 web page. `http://www.xcsp.org/`.

Chu, G.; Schulte, C.; and Stuckey, P. J. 2009. Confidence-based work stealing in parallel constraint programming. In *International conference on principles and practice of constraint programming*, 226–241. Springer.

Demeulenaere, J.; Hartert, R.; Lecoutre, C.; Perez, G.; Perron, L.; Régin, J. C.; and Schaus, P. 2016. Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In *Proc. of CP'16*, 207–223.

Gent, I. P.; Jefferson, C.; Miguel, I.; Moore, N. C.; Nightingale, P.; Prosser, P.; and Unsworth, C. 2011. A preliminary review of literature on parallel constraint solving. In *Proceedings PMCS 2011 Workshop on Parallel Methods for Constraint Solving*, 499–504.

Ingmar, L., and Schulte, C. 2018. Making compact-table compact. In *International Conference on Principles and Practice of Constraint Programming*, 210–218. Springer.

Kasif, S. 1990. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence* 45(3):275–286.

Lecoutre, C.; Likitvivatanavong, C.; and Yap, R. 2012. A path-optimal gac algorithm for table constraints. In *20th European Conference on Artificial Intelligence (ECAI'12)*, 510–515.

Lecoutre, C.; Paparrizou, A.; and Stergiou, K. 2013. Extending str to a higher-order consistency. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 576–582.

Lecoutre, C. 2011. Str2: optimized simple tabular reduction for table constraints. *Constraints* 16(4):341–371.

Likitvivatanavong, C.; Xia, W.; and Yap, R. H. 2015. Decomposition of the factor encoding for csps. In *IJCAI*, 353–359.

Mairy, J.-B.; Van Hentenryck, P.; and Deville, Y. 2014. Optimal and efficient filtering algorithms for table constraints. *Constraints* 19(1):77–120.

Michel, L.; See, A.; and Van Hentenryck, P. 2007. Parallelizing constraint programs transparently. In *International Conference on Principles and Practice of Constraint Programming*, 514–528. Springer.

Nguyen, T., and Deville, Y. 1998. A distributed arc-consistency algorithm. *Science of Computer Programming* 30(1-2):227–250.

Paparrizou, A., and Stergiou, K. 2017. On neighborhood singleton consistencies. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI*, 19–25.

Perez, G., and Régin, J.-C. 2014. Improving gac-4 for table and mdd constraints. In *International Conference on Principles and Practice of Constraint Programming*, 606–621. Springer.

Rolf, C. C., and Kuchcinski, K. 2009. Parallel consistency in constraint programming. In *International Conference on Parallel and Distributed Processing Techniques and Applications, Pdpta 2009, Las Vegas, Nevada, Usa, July 13-17, 2009, 2 Volumes*, 638–644.

Rolf, C., and Kuchcinski, K. 2010. Parallel solving in constraint programming. In *MCC 2010: Third Swedish Workshop on Multi-Core Computing*.

Ruiz-Andino, A.; Araujo, L.; Sáenz, F.; Ruz, J. J.; Ruiz-Andino, A.; Araujo, L.; and Sáenz, F. 1998. Parallel arc-consistency for functional constraints. In *International Conference on Logic Programming/Joint International Conference and Symposium on Logic Programming*, 86–100.

Schneider, A., and Choueiry, B. Y. 2018. Pw-ac: Extending compact-table to enforce pairwise consistency on table constraints. In *International Conference on Principles and Practice of Constraint Programming*, 345–361. Springer.

Schulte, C. 2000. Parallel search made simple. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP*, 41–57.

Stergiou, K. 2015. Restricted path consistency revisited.

In *International Conference on Principles and Practice of Constraint Programming*, 419–428. Springer.

Verhaeghe, H.; Lecoutre, C.; Deville, Y.; and Schaus, P. 2017. Extending compact-table to basic smart tables. In *International Conference on Principles and Practice of Constraint Programming*, 297–307. Springer.

Verhaeghe, H.; Lecoutre, C.; and Pierre, S. 2017. Extending compact-table to negative and short tables. In *Proc. of AAAI'17*, 3951–3957.

Verhaeghe, H.; Lecoutre, C.; and Schaus, P. 2018. Compact-mdd: Efficiently filtering (s) mdd constraints with reversible sparse bit-sets. In *IJCAI*, 1383–1389.

Wallace, R. J. 2016. Neighbourhood sac: Extensions and new algorithms. *AI Communications* 29(2):249–268.

Wang, R.; Xia, W.; Yap, R. H. C.; and Li, Z. 2016. Optimizing simple tabular reduction with a bitwise representation. In *Proc. of IJCAI'16*, 787–793.