Maximilian Eibl, Martin Gaedke. (Hrsg.): INFORMATIK 2017,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 615

Using Hypertree Decomposition for Parallel Constraint Solving

Ke Liu[1], Sven Löffler[1] and Petra Hofstedt[1]

Abstract: Multi-core processors or many-core processors have become the standard configuration for computers nowadays. Yet, the mainstream constraint solvers have not fully utilized these available computation resources due to the intrinsic difficulty on decomposing constraint satisfaction problems (CSPs). This paper reviews the previous research in parallel constraint solving and proposes a new approach for mapping constraint networks on multi-core or many-core processors by means of hypertree decomposition. We give theoretical considerations and our plans for future research.

Keywords: CSP, constraint networks, parallel constraint solving, hypertree decomposition

1 Introduction

The constraint programming (CP) community has been engaged in enhancing the performance of solving constraint satisfaction problems (CSPs). The development of sequential methods, e.g. sophisticated propagation algorithms or efficient search algorithms and variable and value selectors, might have been matured. However, research in parallelism in CP is still in its early stage, although much effort has been put into parallel constraint solving.

We propose a structure-driven analysis method to statically decompose the constraint network of a given CSP. Compared to prior dynamical decomposition [RRM13] which allocates the work on demand, we use a statical decomposition method, and we try to equally distribute workload to workers or processors at the beginning. Moreover, the need for communication is reduced in contrast to parallel consistency [RK09] due to tree structure.

This paper is organized as follows. In Sect. 2 we review some background knowledge about tree and hypertree decomposition. We explain how hypertree decomposition can be used to partition a constraint network and we present a theoretical analysis and discussion of the method in Sect. 3. Finally, in Sect. 4 we give a conclusion and discuss future work.

[1] Brandenburg University of Technology Cottbus - Senftenberg, Programming Languages and Compiler Construction, Konrad-Wachsmann-Allee 5, D-03044 Cottbus, {Ke.Liu, Sven.Loeffler, Petra.Hofstedt}@b-tu.de

doi:10.18420/in2017_58

2 Preliminaries

A *constraint network* $\mathcal{R}$ is a triple (X, D, C), which consists of:

- a finite set of variables $X = \{x_1, \ldots, x_n\}$,
- a set of respective finite domains $D = \{D_1, \ldots, D_n\}$, where D_i is the domain of the variable x_i, and
- a set of constraints $C = \{c_1, ..., c_t\}$, where a constraint c_j is a relation R_j defined on a subset of variables S_j, $S_j \subseteq X$.

Any constraint network can be graphically represented by a *hypergraph*. A hypergraph $\mathcal{H}$ is a tuple (V, E), where V is a set of vertexes and E is a set of *hyperedges*. A hyperedge of a hypergraph is composed of two or more vertexes, which makes hyperedges fundamentally different from normal edges in a graph. Any constraint in a given constraint network corresponds to a hyperedge in a hypergraph, and the variables of a constraint can be seen as vertexes of a hyperedge.

A *hypertree* of a hypergraph $\mathcal{H}$ is a triple (T, χ, λ), where $T = (V_T, E_T)$ is a tree, χ and λ are labeling functions. We denote a set of variables for a given node ($node_i$) in a hypertree by v_i. Therefore, $v_i = \chi(node_i)$ and $v_i \subseteq 2^{vertexes(\mathcal{H})}$, where $vertexes(\mathcal{H})$ are vertexes of hypergraph $\mathcal{H}$. Similarly, we denote a set of edges of $node_i$ by e_i. Therefore, $e_i = \lambda(node_i)$ and $e_i \subseteq 2^{edges(\mathcal{H})}$, where $edges(\mathcal{H})$ are hyperedges of hypergraph $\mathcal{H}$. By $root(T)$ we denote the root of a tree T, for every $p \in V_T$, let T_p denote the subtree of T with root p. The *width of a hypertree* is the maximum number of hyperedges among the nodes of it, which is given by $hw(T) = max \mid \lambda(p) \mid, \forall p \in V_T$.

Hypertree decomposition is a procedure, which converts a hypergraph into a hypertree. In order to demonstrate hypertree decomposition on a given constraint network, assume we have a simple problem over a set of variables $\{x_1, \ldots, x_{10}\} \subseteq X$ modeled by the following constraints[2]

- $atLeastNValues(x_1, x_2, x_3)$
- $allDifferent1(x_3, x_4, x_5, x_7)$
- $allDifferent2(x_1, x_4, x_6, x_9)$
- $table1(x_5, x_8, x_{10})$
- $table2(x_7, x_8, x_9)$
- $arithm1(x_5, x_6)$

[2] The name of the constraints is consistent with the name of constraints used in the Choco Solver [PFL16].

- $arithm2(x_6, x_8)$

The hypergraph for this constraint network is depicted in Figure 1, where the variables x_i, $i \in \{1, \ldots, 10\}$ are the vertexes, while the edges are represented by the enclosing ellipses. Figure 2 shows a possible hypertree decomposition of this hypergraph.

Given a hypergraph $\mathcal{H}$ and a constant k, deciding whether $hw(T) \leq k$ for a hypertree T received by decomposition of $\mathcal{H}$ is *NP-complete* [Go05]. However, it is feasible to determine, whether there exists a hypertree decomposition T for k, such that $hw(T) \leq k$. Furthermore, to compute a hypertree decomposition with width $\leq k$ for a given hypergraph is in polynomial time [GLS99]. Herein, we do not go into details of hypertree decomposition algorithm. For more details of the hypertree decomposition algorithm *det-k-decomp*, please refer to [GS08].

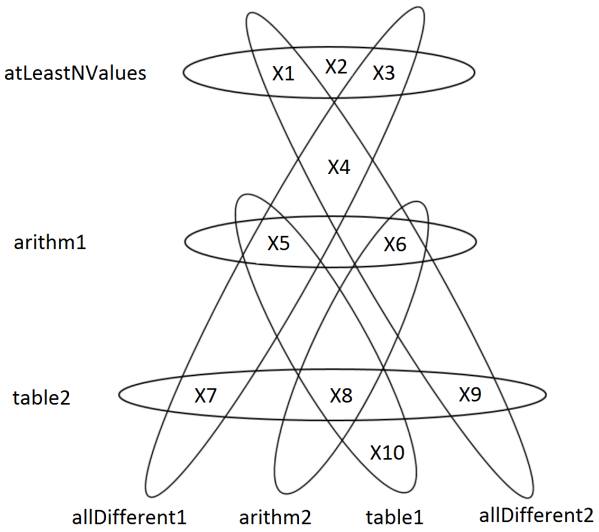

Fig. 1: The Hypergraph for the Constraint Network. This graph is recomposed based on [GLS03]

3 Parallel Solving Constraint Satisfaction Problem

Constraint programming and its solvers are typically used to tackle NP-complete problems. Thus, to obtain better performance, we naturally think of utilizing parallel computing to gain performance improvements. Of course, the growth of the number of cores can hardly overtake the growth of problem size which might be exponentially in the input size. However, one possible way to deal with large CSPs is to decompose its constraint networks into a number of connected sub-networks. This claim is based on the following

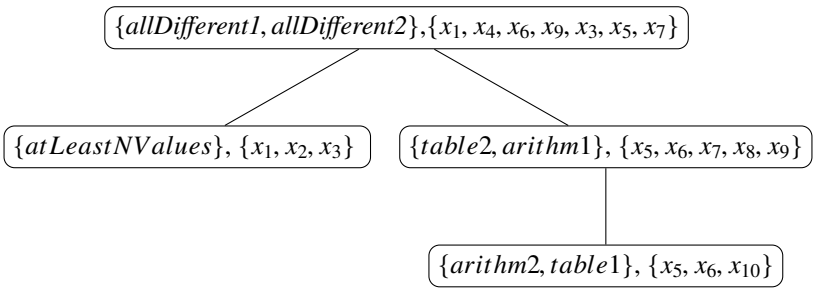

Fig. 2: Hypertree Decomposition for Hypergraph 1. This graph is recomposed based on [GLS03]

simple analysis. Let's assume a constraint network $\mathcal{N}$ with number of variables n, and the maximum domain size among all variables is d. Thus, the worst case execution time is $O(d^n)$. But if the constraint network could be divided into the number of s sub-networks, the worst case execution time for the given problem would be $s * d^{\frac{n}{s}}$, which can result in much less computation work than the original problem. Moreover, the speedup of parallel execution of s subproblems is $o(d^{n(1-\frac{1}{s})})$, which is a significant performance improvement. Note that this improvement is based on the assumption that the s subproblems are disjoint. However, completely disjoint subproblems for a constraint network are uncommon in practical problems. Thus, it is necessary to decompose the constraint network into a tree network because the tree structure implies the network is tractable [Fr82, DP87, RVBW06]. After obtaining the decomposition tree T, backtrack-free search can be realized along a topological ordering d of T, if T is directional arc-consistent relative to d [De03]. Each node of the decomposition tree T consists of tuples that are the outcomes from solving the subproblems. At the moment, the decomposition tree T can be viewed as a binary representation of the original non-binary constraint network.

In short, we propose a procedure that solves constraint programming in parallel with the following four steps:

1. Decompose the hypergraph of the original constraint network into a tree in which the number of the tree nodes is equal to the number of cores on the processors.
2. Simultaneously solve the sub-network on each node of the decomposition tree.
3. Perform directional arc-consistency along a topological ordering of the decomposition tree.
4. Combine all solutions starting from the root node along the topological ordering.

Alternatively, if the goal is to find one solution, the steps 3 and 4 can be replaced by a join selection algorithm (eg., Hash Join) known from relational databases.

3.1 Hypergraph decomposition

The purpose of hypergraph decomposition for a given constraint network in our approach is twofold. First, hypergraph decomposition is a way to statically map the workload evenly to the processors. Second, the tree structure of the resulting hypertree reduces the number of communications in comparison to other mapping approaches. For example, the worst case of a mapping approach for a constraint network requires $\binom{n}{2}$ communications (i. e. a complete graph); in contrast, mapping with hypertree decomposition requires $n - 1$ communications, where n is the number of edges in the communication graph after mapping.

Hypertree decomposition dominates all other decomposition methods (eg., cycle cutset, hinge decomposition) [GLS00, Le13] because it is the most general decomposition algorithm. Nevertheless, the original *det-k-decomp* algorithm [GS08] was not designed for executing CSPs in parallel. Two reasons make it unsuitable for our application scenario. First of all, if the width k is large and the hypergraph has many hyperedges, then the exact algorithm may take long execution time [Go16]. Furthermore, the parameter k only guarantees the width of a decomposition tree is k, which means k is the largest number of constraints among all nodes of the decomposition tree. This also implies that the nodes may have width from 1 to $k - 1$. However, k, fairly often, need to be larger than 10 in our case. For instance, assume a constraint network is composed of 104 constraints. In order to map 104 constraints onto 8 cores, a good choice of k for load balancing could be 13. Our experiments of decomposing hypergraph using *det-k-decomp* shows that it cannot avoid generating a hypertree decomposition that has nodes with width smaller than k.

In order to map constraints to cores, we investigated a new procedure to merge nodes of a hypertree decomposition on the basis of *det-k-decomp*. The outcome of the merge process follows three basic principles: the acyclic property of the hypertree decomposition must not be destroyed, the number of nodes of the new hypertree decomposition must be equal to the number of cores (or logical cores), and the set of solutions for the two constraint network must be equivalent.

3.2 Parallel Framework

In this subsection, we briefly sketch the parallel framework to run multiple constraint solvers on different worker threads simultaneously. Our framework applies two means to coordinate worker threads. Firstly, the main thread waits for all worker threads to finish their tasks, and afterwards it begins to perform directional arc-consistency [De03]. This method allows to deals with small constraint network problems, where waiting for all working threads to finish is affordable. Yet, if the problem is non-trivial, then to wait for all solutions of the single solving processes is unaffordable (e. g., it may take several hours). Therefore, it is necessary to timely share partial solutions to the main thread which is responsible for performing steps 3 and 4 (cf. Sect. 3.1). This synchronization is the second means for the coordination of the worker threads.

```
1  l_start = current time ;
2  l_end = ZERO ;
3  INTERVAL = constant time interval ;
4  while solver still has a solution do
5  |  add the solution into the synchronized shared variable ;
6  |  l_end = current time ;
7  |  if l_end -l_start ≥ INTERVAL then
8  |  |  pause this working thread ;
9  |  end
10 end
```

Algorithm 1: *Working thread as a Producer*

Algorithm 1 shows that the working threads as producers are synchronized by a timer. This way might be better than, e. g., using a unified number of solutions to coordinate the working threads because it is hard to predict the execution time for different solvers. In line 8 of Algorithm 1, the working thread pauses itself if the time is up. Then the consumer thread blocks itself by the same time interval rate, as shown line 2 of Algorithm 2, and gathers the partial solutions from the working threads. Afterwards, the working threads are resumed by the consumer thread. The advantage of this mechanism is that each solver can continue searching at the same point of the backtracking search tree after resuming and enable the consumer thread working without staying idle. The whole procedure can also be seen as a two-stage pipeline.

```
1  INTERVAL = constant time interval ;
2  while !awaitTermination(INTERVAL) do
3  |  gather solutions from working threads ;
4  |  resume all unfinished working threads ;
5  end
```

Algorithm 2: *Main thread as Consumer*

4 Conclusion and Future Work

In this paper, we have presented a new approach on parallel solution of constraint problems using hypergraph decomposition. Up to now, we have just implemented and verified parts of the project. Therefore, future work will, first of all, focus on completing implementation and on verification of the theoretical analysis by more experiments. Furthermore, the following questions should be addressed. First, what is the best algorithm for the join operation to get solutions in the last step; can we, possibly, directly port join algorithms from the database community? Second, which is better between the tight constraint or loose constraint for each solver? In other words, more jobs are assigned to constraint solver or more jobs are left to join selection step? Third, how to cope with memory explosion when several solvers execute simultaneously and generate a huge amount of intermediate solution tuples? Should we

apply data compression techniques? If so, how do we find the trade-off between performance and compression ratio?

References

[De03] Dechter, Rina: Constraint processing. Morgan Kaufmann, 2003.

[DP87] Dechter, Rina; Pearl, Judea: Network-based heuristics for constraint-satisfaction problems. Artificial Intelligence, 34(1):1–38, 1987.

[Fr82] Freuder, Eugene C: A sufficient condition for backtrack-free search. Journal of the ACM (JACM), 29(1):24–32, 1982.

[GLS99] Gottlob, Georg; Leone, Nicola; Scarcello, Francesco: Hypertree decompositions and tractable queries. In: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS). ACM, pp. 21–32, 1999.

[GLS00] Gottlob, Georg; Leone, Nicola; Scarcello, Francesco: A comparison of structural CSP decomposition methods. Artificial Intelligence, 124(2):243–282, 2000.

[GLS03] Gottlob, Georg; Leone, Nicola; Scarcello, Francesco: Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. Journal of Computer and System Sciences, 66(4):775–808, 2003.

[Go05] Gottlob, Georg; Grohe, Martin; Musliu, Nysret; Samer, Marko; Scarcello, Francesco: Hypertree decompositions: Structure, algorithms, and applications. In: International Workshop on Graph-Theoretic Concepts in Computer Science. LNCS 3787. Springer, pp. 1–15, 2005.

[Go16] Gottlob, Georg; Greco, Gianluigi; Leone, Nicola; Scarcello, Francesco: Hypertree decompositions: Questions and answers. In: Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS). ACM, pp. 57–74, 2016.

[GS08] Gottlob, Georg; Samer, Marko: A backtracking-based algorithm for hypertree decomposition. Journal of Experimental Algorithmics (JEA), 13, 2008.

[Le13] Lecoutre, Christophe: Constraint Networks: Targeting Simplicity for Techniques and Algorithms. John Wiley & Sons, 2013.

[PFL16] Prud'homme, Charles; Fages, Jean-Guillaume; Lorca, Xavier: Choco Documentation. 2016. `http://www.choco-solver.org`.

[RK09] Rolf, Carl Christian; Kuchcinski, Krzysztof: Parallel Consistency in Constraint Programming. In: International Conference on Parallel and Distributed Processing Techniques and Applications, (PDPTA). pp. 638–644, 2009.

[RRM13] Régin, Jean-Charles; Rezgui, Mohamed; Malapert, Arnaud: Embarrassingly parallel search. In: International Conference on Principles and Practice of Constraint Programming. LNCS 8124. Springer, pp. 596–610, 2013.

[RVBW06] Rossi, Francesca; Van Beek, Peter; Walsh, Toby: Handbook of constraint programming. Elsevier, 2006.