Hypertree Decomposition for Constraint Programming in Parallel

Ke Liu, Sven Loeffler, Petra Hofstedt

Brandenburg University of Technology Cottbus-Senftenberg, Germany

Constraint programming (CP) is a declarative programming paradigm in which the problem is modeled as a group of relations over a finite set of variables. Formally, 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, $S_j \subseteq X$.

Note that from the problem definition point of view, a problem modeled as a constraint network is also called constraint satisfaction problem, or CSP. By using constraint programming, one can easily attack some NP-complete problems in a reasonable execution time without developing specific, and fairly often complex, algorithms.

Backtrack search and constraint propagation lie at the heart of the techniques for constraint programming. But the potential of these techniques might have been exhausted due to the extensive and effective researches in this area during last decades. On the other hand, parallel computing is often an effective approach to enhance the performance of sequential algorithm. However, the research of parallel constraint solving is still in its infant stage although parallel search and parallel propagation for constraint programming have been studied.

This paper aims at presenting a new mapping algorithm as the first step for parallel constraint solving, where the term parallel constraint solving refers to executing the whole constraint network simultaneously by means of partition and mapping it onto the different cores. The idea behind this parallel constraint solving is to divide the original intractable problem into the tractable sub-problems because many NP-complete and NP-hard problems can be solved in polynomial time if the corresponding *hypergraph* has the bounded *hypertree-width* [1]. A hypergraph $\mathcal{H} = (V, S)$ corresponding to a constraint network $\mathcal{N}$ is composed of a set of vertexes $V = \{v_1, ..., v_n\}$ and a set of hyperedges of these vertexes $\mathcal{S} = \{s_1, ..., s_i\}$, where each hyperedge s_i might contain more than two vertexes. Moreover, each hyperedge s_i in a hypergraph $\mathcal{H}$ has a one-to-one correspondence with a constraint in the corresponding constraint network $\mathcal{N}$; similarly, any vertex v_n in the hypergraph $\mathcal{H}$ has a one-to-one correspondence with a variable in $\mathcal{N}$ as well. A *hypertree* of a hypergraph $\mathcal{H}$ is a triple $(T,\ \chi,\ \lambda)$, where $T = (V_T, E_T)$ denotes the tree structure of the hypergraph, χ and λ are labeling functions that stand for the set of variables and the set of constraints of each

node for the hypertree. The hypertree width of a given hypertree is the maximal number of hyperedges (constraints) among all nodes of that hypertree.

Although there exists several hypertree decomposition algorithms, e.g. $det-k-decomp$ [1], in the literature published in the last decades, the target decomposition tree of these algorithms are the hypertree with width as small as possible because the small hypertree width indicates the problem can be solved faster. Hence, we develop a dedicated hypergraph decomposition method $detk-k-decomp$ for mapping constraints onto cores. The idea behind $det-k-CP$ is to utilize the property that one edge between two nodes can be eliminated without changing the set of all solutions for the constraint network [2] because there exists an alternative path between the two nodes. The edge between two nodes represents shared variables. As can be seen from Figure 1, the edge between node c_1 and c_3 can be removed since there is an alternative path $c_1 \rightarrow c_2 \rightarrow c_3$ from c_1 to c_2.

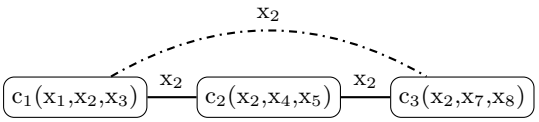


Fig. 1: A simple constraint network shows the edge between two nodes that can be eliminated.

The target decomposition tree of $det-k-CP$ can be represented by a degeneration decomposition tree in which the edges between any pair of non-adjacent nodes are eliminated, and only the edges between adjacent nodes are preserved. A target degeneration decomposition tree with four nodes for a given constraint network is shown in Figure 2 below, where solid lines are drawn for edges between adjacent nodes, and the dot-dashed lines stand for the edges between non-adjacent nodes that can be removed. Besides, the path from $Node1$ to $Node4$, which is drawn by solid line in 2, is also called *main path* of the degeneration decomposition tree.

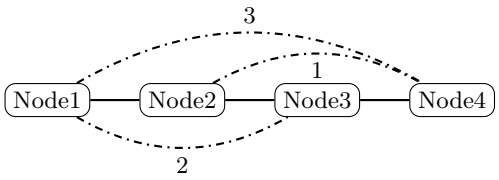
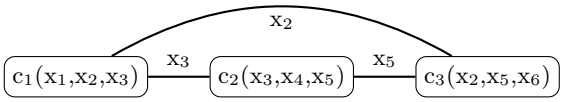
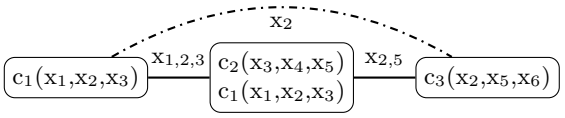


Fig. 2: A degeneration decomposition tree decomposed by $det-k-CP$. The number next to each arc denotes the execution order of adding procedure.



(a) The original constraint network



(b) A simple degeneration decomposition tree

Fig. 3: Adding procedure for eliminating non-adjacent edge

We are now going to sketchily introduce the details of algorithm of $det-k-CP$ by using Figure 2, and Figure 3. In the first step, $det-k-CP$ sorts the constraints based on the weight of each constraint, where the weight of a constraint depends on the time complexity propagation algorithm used by the constraint, the size of domains for variables, and the number of variables for the given constraint network. Then, $det-k-CP$ adds constraints from i_{th} node onto $i+1_{th}$ node if necessary, so that the shared variables between two non-adjacent nodes are covered by the corresponding main path's variables from node j to node k, where $j \leq i < i+1 \leq k$. For example, in Figure 3a, a simple constraint network is composed of three constraints in which the edge between c_1 and c_3 is due to common shared variable x_2. By adding c_1 to the second node, the edge between c_1 and c_3 can be eliminated since the main path contains x_2 now, as shown in Figure 3b. We call this procedure is *adding procedure* for eliminating non-adjacent edge. $det-k-CP$ carries out the adding procedure from the tail to the head of degeneration decomposition tree in turn, e.g., in Figure 2, the edge between $Node2$ and $Node4$ is firstly removed, and then is the edge between $Node1$ and $Node3$. Finally, the edge between $Node1$ and $Node4$ is removed. However, the adding procedure cannot ensure the hypergraph of a given constraint network is decomposed successfully. For instance, in order to eliminate the edge between $Node1$ and $Node3$ in Figure 2, $det-k-CP$ might need to add constraints from $Node1$ to $Node2$ so that the main path between $Node1$ and $Node3$ can cover arc 2. The adding procedure to $Node2$ might regenerate the arc 1 that had been removed by the adding procedure to $Node3$ because the new common shared variables between $Node2$ and $Node4$ might reappear. Therefore, in the last step of $det-k-CP$, the stochastic exchange for nodes is introduced. The heuristic used by stochastic exchange could be randomly exchange, switching nodes that have the fewest and most number of constraints, or changing the permutation of the indexes of degenerate decomposition tree in order. For instance, if the number of nodes of a degenerate decomposition tree is 4, we might firstly use the permutation (0,1,2,3), then (0,1,3,2) and so on. In short, $det-k-CP$ performs adding procedure for every edge between non-adjacent

nodes, and then the stochastic exchange will be executed if the hypergraph of a given constraint network is not decomposed successfully. Adding procedure and stochastic exchange are carried out in each iteration until a degeneration decomposition tree is obtained.

We have briefly presented the decomposition algorithm $det-k-CP$ to map constraints for parallel constraint solving. This algorithm can be solved in polynomial time, because even if there is no degeneration decomposition tree to be found, the algorithm will eventually stop, in that case, every node would contain the entire constraint network. But we have not encountered such a situation so far after evaluating the benchmark suit provided by Gottlob et al. used in study [1]. The future work will be focused on using $det-k-CP$ to gain the speed up for parallel constraint solving.

References

1. Gottlob, G., Samer, M.: A backtracking-based algorithm for hypertree decomposition. Journal of Experimental Algorithmics (JEA) **13** (2009) 1
2. Dechter, R.: Constraint processing. Morgan Kaufmann (2003)