Exploiting GPUs in Solving (Distributed) Constraint Optimization Problems with Dynamic Programming*

Ferdinando Fioretto[1,2], Tiep Le[1], Enrico Pontelli[1], William Yeoh[1], and Tran Cao Son[1]

[1] Department of Computer Science, New Mexico State University
[2] Department of Mathematics & Computer Science, University of Udine
{ffiorett,tile,epontell,wyeoh,tson}@cs.nmsu.edu

Abstract. This paper proposes the design and implementation of a dynamic programming based algorithm for (distributed) constraint optimization, which exploits modern massively parallel architectures, such as those found in modern Graphical Processing Units (GPUs). The paper studies the proposed algorithm in both centralized and distributed optimization contexts. The experimental analysis, performed on unstructured and structured graphs, shows the advantages of employing GPUs, resulting in enhanced performances and scalability.

1 Introduction

The importance of constraint optimization is outlined by the impact of its application in a range of *Constraint Optimization Problems* (COPs), such as supply chain management (e.g., [27, 15]) and roster scheduling (e.g., [1, 8]). When resources are distributed among a set of autonomous agents and communication among the agents are restricted, COPs take the form of *Distributed Constraint Optimization Problems* (DCOPs) [21, 33]. In this context, agents coordinate their value assignments to maximize the overall sum of resulting constraint utilities. DCOPs are suitable to model problems that are distributed in nature, and where a collection of agents attempts to optimize a global objective within the confines of localized communication. They have been employed to model various distributed optimization problems, such as meeting scheduling [20, 32, 35], resources allocation [13, 36], and power network management problems [17].

Dynamic Programming (DP) based approaches have been adopted to solve COPs and DCOPs. The *Bucket Elimination* (BE) procedure [10] iterates over the variables of the COP, reducing the problem at each step by replacing a variable and its related utility functions with a single new function, derived by optimizing over the possible values of the replaced variable. The *Dynamic Programming Optimization Protocol* (DPOP) [25] is one of the most efficient DCOP solvers, and it can be seen as a distributed version of BE, where agents exchange newly introduced utility functions via messages.

The importance of DP-based approaches arises in several optimization fields including constraint programming [2, 28]. For example, several *propagators* adopt DP-based

* This research is partially supported by the National Science Foundation under grant number HRD-1345232. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, or the U.S. government.

techniques to establish constraint consistency; for instance, **(1)** the *knapsack* constraint propagator proposed by Trick applies DP techniques to establish arc consistency on the constraint [31]; **(2)** the propagator for the *regular* constraint establishes arc consistency using a specific digraph representation of the DFA, which has similarities to dynamic programming [24]; **(3)** the *context free grammar* constraint makes use of a propagator based on the CYK parser that uses DP to enforce generalized arc consistency [26].

While DP approaches may not always be appropriate to solve (D)COPs, as their time and space requirements may be prohibitive, they may be very effective in problems with particular structures, such as problems where their underlying constraint graphs have small induced widths or distributed problems where the number of messages is crucial for performance, despite the size of the messages. The structure exploited by DP-based approaches in constructing solutions makes it suitable to exploit a novel class of massively parallel platforms that are based on the *Single Instruction Multiple Thread* (SIMT) paradigm—where multiple threads may concurrently operate on different data, but are all executing the same instruction at the same time. The SIMT-based paradigm is widely used in modern *Graphical Processing Units* (GPUs) for general purpose parallel computing. Several libraries and programming environments (e.g., *Compute Unified Device Architecture* (CUDA)) have been made available to allow programmers to exploit the parallel computing power of GPUs.

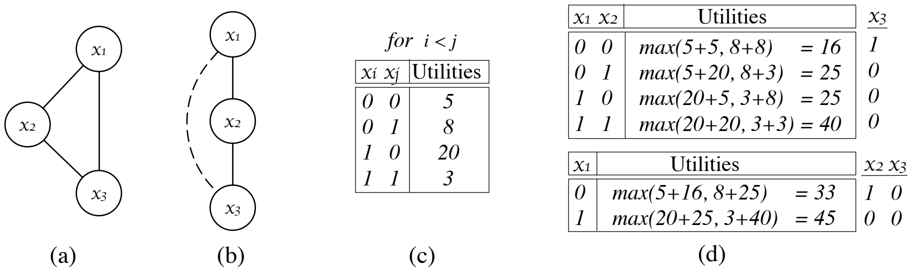
In this paper, we propose a design and implementation of a DP-based algorithm that exploits parallel computation using GPUs to solve (D)COPs. Our proposal aims at employing GPU hardware to speed up the inference process of DP-based methods, representing an alternative way to enhance the performance of DP-based constraint optimization approaches. This paper makes the following contributions: **(1)** We propose a novel design and implementation of a centralized and a distributed DP-based algorithm to solve (D)COPs, which harnesses the computational power offered by parallel platforms based on GPUs; **(2)** We enable the use of concurrent computations between CPU(s) and GPU(s), during (D)COP resolution; and **(3)** We report empirical results that show significant improvements in performance and scalability.

2 Background

2.1 Centralized Constraint Optimization Problems (COPs)

A (centralized) *Constraint Optimization Problem* (COP) is defined as $(\mathbf{X}, \mathbf{D}, \mathbf{C})$ where: $\mathbf{X} = \{x_1, \dots, x_n\}$ is a set of variables; $\mathbf{D} = \{D_1, \dots, D_n\}$ is a set of *domains* for the variables in $\mathbf{X}$, where D_i is the set of possible values for the variable x_i; $\mathbf{C}$ is a finite set of utility functions on variables in $\mathbf{X}$, with $f_i : \times_{x_j \in \mathbf{x}^i} D_j \to \mathbb{R}^+ \cup \{-\infty\}$, where $\mathbf{x}^i \subseteq \mathbf{X}$ is the set of variables relevant to f_i, referred to as the *scope* of f_i, and $-\infty$ is used to denote that a given combination of values for the variables in $\mathbf{x}^i$ is not allowed.[3] A *solution* is a value assignment for a subset of variables from $\mathbf{X}$ that is consistent with their respective domains; i.e., it is a partial function $\theta : \mathbf{X} \to \bigcup_{i=1}^{n} D_i$ such that, for each $x_j \in \mathbf{X}$, if $\theta(x_j)$ is defined, then $\theta(x_j) \in D_j$. A solution is *complete* if it assigns a value to each variable in $\mathbf{X}$. We will use the notation σ to

[3] For simplicity, we assume that tuples of variables are built according to a predefined ordering.



for i < j

xi	xj	Utilities
0	0	5
0	1	8
1	0	20
1	1	3

x1	x2	Utilities	x3
0	0	max(5+5, 8+8) = 16	1
0	1	max(5+20, 8+3) = 25	0
1	0	max(20+5, 3+8) = 25	0
1	1	max(20+20, 3+3) = 40	0

x1	Utilities	x2 x3
0	max(5+16, 8+25) = 33	1 0
1	max(20+25, 3+40) = 45	0 0

(a) (b) (c) (d)

Fig. 1: Example (D)COP (a-c) and *UTIL* phase computations in DPOP (d).

denote a complete solution, and, for a set of variables $\mathbf{V} = \{x_{i_1}, \ldots, x_{i_h}\} \subseteq \mathbf{X}$, $\sigma_{\mathbf{V}} = \langle \sigma(x_{i_1}), \ldots, \sigma(x_{i_h}) \rangle$, where $i_1 < \cdots < i_h$. The goal for a COP is to find a complete solution σ^* that maximizes the total problem utility expressed by its utility functions, i.e., $\sigma^* = \operatorname{argmax}_{\sigma \in \mathbf{\Sigma}} \sum_{f_i \in \mathbf{C}} f_i(\sigma_{\mathbf{x}^i})$, where Σ is the *state space*, defined as the set of all possible complete solutions.

Given a COP P, $G_P = (\mathbf{X}, E_{\mathbf{C}})$ is the *constraint graph* of P, where $\{x, y\} \in E_{\mathbf{C}}$ iff $\exists f_i \in \mathbf{C}$ such that $\{x, y\} \subseteq \mathbf{x}^i$. Fig. 1(a) shows the constraint graph of a simple COP with three variables, x_1, x_2, and x_3. The domain of each variable is the set $\{0, 1\}$. Fig. 1(c) describes the utility functions of the COP.

Definition 1 (Projection). *The* projection *of a utility function f_i on a set of variables $\mathbf{V} \subseteq \mathbf{x}^i$ is a new utility function $f_{i|\mathbf{V}} : \mathbf{V} \to \mathbb{R}^+ \cup \{-\infty\}$, such that for each possible assignment $\theta \in \bigtimes_{x_j \in \mathbf{V}} D_j$, $f_{i|\mathbf{V}}(\theta) = \max\limits_{\sigma \in \Sigma, \sigma_{\mathbf{V}} = \theta} f_i(\sigma_{\mathbf{x}^i})$.*

In other words, $f_{i|\mathbf{V}}$ is constructed from the tuples of f_i, removing the values of the variable that do not appear in $\mathbf{V}$ and removing duplicate values by keeping the maximum utility of the original tuples in f_i.

Definition 2 (Concatenation). *Let us consider two assignments θ', defined for variables V, and θ'', defined for variables W, such that for each $x \in V \cap W$ we have that $\theta'(x) = \theta''(x)$. Their* concatenation *is an assignment $\theta' \cdot \theta''$ defined for $V \cup W$, such as for each $x \in V$ (resp. $x \in W$) we have that $\theta' \cdot \theta''(x) = \theta'(x)$ (resp. $\theta' \cdot \theta''(x) = \theta''(x)$).*

We define two operations on utility functions:

- The *aggregation* of two functions f_i and f_j, is a function $f_i + f_j : \mathbf{x}^i \cup \mathbf{x}^j \to \mathbb{R}^+ \cup \{-\infty\}$, such that $\forall \theta' \in \bigtimes_{x_k \in \mathbf{x}^i} D_k$ and $\forall \theta'' \in \bigtimes_{x_k \in \mathbf{x}^j} D_k$, if $\theta' \cdot \theta''$ is defined, then we have that $(f_i + f_j)(\theta' \cdot \theta'') = f_i(\theta') + f_j(\theta'')$.
- *Projecting out* a variable $x_j \in \mathbf{x}^i$ from a function f_i, denoted as $\pi_{-x_j}(f_i)$, produces a new function with scope $\mathbf{x}^i \setminus \{x_j\}$, and defined as the projection of f_i on $\mathbf{x}^i \setminus \{x_j\}$, i.e., $\pi_{-x_j}(f_i) = f_{i|\mathbf{x}^i \setminus \{x_j\}}$.

Bucket Elimination (BE): BE [10, 11] is a dynamic programming based procedure that can be used to solve COPs. Algorithm 1 illustrates its pseudocode. Given a COP $(\mathbf{X}, \mathbf{D}, \mathbf{C})$ and an ordering $o = \langle x_1, \ldots, x_n \rangle$ on the variables in $\mathbf{X}$, we say that a variable x_i has a higher *priority* with respect to variable x_j if x_i appears after x_j in o. BE operates from the highest to lowest priority variable. When operating on variable x_i, it creates a bucket B_i, which is the set of all utility functions that involve x_i as the

Algorithm 1: BE

```
1 for $i \leftarrow n$ downto $1$ do
2     $B_i \leftarrow \{f_j \in \mathbf{C} \mid x_i \in \mathbf{x}^j \wedge i = \max\{k \mid x_k \in \mathbf{x}^j\}\}$
3     $\hat{f}_i \leftarrow \pi_{-x_i}\left(\sum_{f_j \in B_i} f_j\right)$
4     $\mathbf{X} \leftarrow \mathbf{X} \setminus \{x_i\}$
5     $\mathbf{C} \leftarrow (\mathbf{C} \cup \{\hat{f}_i\}) \setminus B_i$
```

highest priority variable in their scope (line 2). The algorithm then computes a new utility function $\hat{f}_i$ by aggregating the functions in B_i and projecting out x_i (line 3). Thus, x_i can be removed from the set of variables $\mathbf{X}$ to be processed (line 4) and the new function $\hat{f}_i$ replaces in $\mathbf{C}$ all the utility functions that appear in B_i (line 5). In our example, BE operates, in order, on the variables x_3, x_2, and x_1. When x_3 is processed, the bucket B_3 is $\{f_{13}, f_{23}\}$, and the $\hat{f}_3$ utility function is shown in Fig. 1(d) top. The rightmost column shows the values for x_3 after its projection. BE updates the sets $\mathbf{X} = \{x_1, x_2\}$ and $\mathbf{C} = \{f_{12}, \hat{f}_3\}$. When x_2 is processed, $B_2 = \{f_{12}, \hat{f}_3\}$ and $\hat{f}_2$ is shown in Fig. 1(d) bottom. Thus, $\mathbf{X} = \{x_1\}$ and $\mathbf{C} = \{\hat{f}_2\}$. Lastly, the algorithm processes x_1, sets $B_1 = \{\hat{f}_2\}$, and $\hat{f}_1$ contains one value combination $\sigma^* = \langle 1, 0, 0 \rangle$, which corresponds to an optimal solution to the problem.

The complexity of the algorithm is bounded by the time needed to process a bucket (line 3), which is exponential in number of variables in the bucket.

2.2 Distributed Constraint Optimization Problems (DCOPs)

In a *Distributed Constraint Optimization Problem* (DCOP) [21, 25, 33], the variables, domains, and utility functions of a COP are distributed among a collection of *agents*. A DCOP is defined as $(\mathbf{X}, \mathbf{D}, \mathbf{C}, \mathbf{A}, \alpha)$, where $\mathbf{X}, \mathbf{D}$, and $\mathbf{C}$ are defined as in a COP, $\mathbf{A} = \{a_1, \dots, a_p\}$ is a set of *agents*, and $\alpha : \mathbf{X} \rightarrow \mathbf{A}$ maps each variable to one agent. Following common conventions, we restrict our attention to binary utility functions and assume that α is a bijection: Each agent controls exactly one variable. Thus, we will use the terms "variable" and "agent" interchangeably and assume that $\alpha(x_i) = a_i$. This is a common assumption in the DCOP literature as there exist pre-processing techniques that transform a general DCOP into this more restrictive DCOP [7, 34]. In DCOPs, solutions are defined as for COPs, and many solution approaches emulate those proposed in the COP literature. For example, ADOPT [21] is a distributed version of *Iterative Deepening Depth First Search*, and DPOP [25] is a distributed version of BE. The main difference is in the way the information is shared among agents. Typically, a DCOP agent knows exclusively its domain and the functions involving its variable. It can communicate exclusively with its neighbors (i.e., agents directly connected to it in the constraint graph[4]), and the exchange of information takes the form of messages. Given a DCOP P, a *DFS pseudo-tree* arrangement for G_P is a spanning tree $T = \langle \mathbf{X}, E_T \rangle$ of G_P such that if $f_i \in \mathbf{C}$ and $\{x, y\} = \mathbf{x}^i$, then x and y appear in the same branch of T. Edges of G_P that are *in* (resp. *out* of) E_T are called *tree edges* (resp. *backedges*). The tree edges connect parent-child nodes, while backedges connect a node with its

[4] The *constraint graph* of a DCOP is equivalent to that of the corresponding COP.

pseudo-parents and its *pseudo-children*. We use $N(a_i) = \{a_j \in \mathbf{A} \mid \{x_i, x_j\} \in E_T\}$ to denote the neighbors of agent a_i; C_i, PC_i, P_i, and PP_i to denote the set of children, pseudo-children, parent, and pseudo-parents of agent a_i; and $sep(a_i)$ to denote the *separator* of agent a_i, which is the set of ancestor agents that are constrained (i.e., they are linked in G_P) with agent a_i or with one of its descendant agents in the pseudo-tree. Fig. 1(b) shows one possible pseudo-tree for the problem, where the agent a_1 has one pseudo-child a_3 (the dotted line is a backedge).

Dynamic Programming Optimization Protocol (DPOP): DPOP [25] is a dynamic programming based DCOP algorithm that is composed of three phases. **(1)** *Pseudo-tree generation*: Agents coordinate to build a pseudo-tree, realized through existing distributed pseudo-tree construction algorithms [16]. **(2)** *UTIL propagation*: Each agent, starting from the leaves of the pseudo-tree, computes the optimal sum of utilities in its subtree for each value combination of variables in its separator. The agent does so by aggregating the utilities of its functions with the variables in its separator and the utilities in the *UTIL* messages received from its child agents, and then projecting out its own variable. In our example problem, agent a_3 computes the optimal utility for each value combination of variables x_1 and x_2 (Fig. 1(d) top), and sends the utilities to its parent agent a_2 in a *UTIL* message. When the root agent a_1 receives the *UTIL* message from each of its children, it computes the maximum utility of the entire problem. **(3)** *VALUE propagation*: Each agent, starting from the root of the pseudo-tree, determines the optimal value for its variable. The root agent does so by choosing the value of its variable from its *UTIL* computations—selecting the value with the maximal utility. It sends the selected value to its children in a *VALUE* message. Each agent, upon receiving a *VALUE* message, determines the value for its variable that results in the maximum utility given the variable assignments (of the agents in its separator) indicated in the *VALUE* message. Such assignment is further propagated to the children via *VALUE* messages.

The complexity of DPOP is dominated by the *UTIL propagation* phase, which is exponential in the size of the largest separator set $sep(a_i)$ for all $a_i \in \mathbf{A}$. The other two phases require a polynomial number of linear size messages, and the complexity of the local operations is at most linear in the size of the domain.

Observe that the *UTIL propagation* phase of DPOP emulates the BE process in a distributed context [6]. Given a pseudo-tree and its *preorder listing* o, the *UTIL* message generated by each DPOP agent a_i is equivalent to the aggregated and projected function $\hat{f}_i$ in BE when x_i is processed according to the ordering o.

2.3 Graphical Processing Units (GPUs)

Modern *GPUs* are multiprocessor devices, offering hundreds of computing cores and a rich memory hierarchy to support graphical processing. We consider the NVIDIA *CUDA* programming model [29], which enables the use of the multiple cores of a graphics card to accelerate general (non-graphical) applications. The underlying model of parallelism is *Single-Instruction Multiple-Thread* (SIMT), where the same instruction is executed by different threads that run on identical cores, grouped in *Streaming Multiprocessors* (SMs), while data and operands may differ from thread to thread.

A typical CUDA program is a C/C++ program. The functions in the program are distinguished based on whether they are meant for execution on the CPU (referred to

Algorithm 2: GPU-(D)BE

```
(1) Generate pseudo-tree
2   GPU-INITIALIZE( )
3   if $C_i = \emptyset$ then
4       $UTIL_{x_i} \leftleftarrows$ PARALLELCALCUTILS( )
(5)     Send $UTIL$ message $(x_i, UTIL_{x_i})$ to $P_i$
6   else
7       Activate UTILMessageHandler(·)
(8) Activate VALUEMessageHandler(·)
```

as the *host*) or in parallel on the GPU (referred as the *device*). The functions executed on the device are called *kernels*, and are executed by several *threads*. To facilitate the mapping of the threads to the data structures being processed, threads are grouped in *blocks*, and have access to several memory levels, each with different properties in terms of speed, organization, and capacity. CUDA maps blocks (coarse-grain parallelism) to the SMs for execution. Each SM schedules the threads in a block (fine-grain parallelism) on its computing cores in chunks of 32 threads (*warps*) at a time. Threads in a block can communicate by reading and writing a common area of memory (*shared memory*). Communication between blocks and communication between the blocks and the host is realized through a large slow *global memory.* The development of CUDA programs that efficiently exploit SIMT parallelism is a challenging task. Several factors are critical in gaining performance. Memory levels have significantly different sizes (e.g., registers are in the order of dozens per thread, shared memory is in the order of a few kilobytes per block) and access times, and various optimization techniques are available (e.g., *coalesced* of memory accesses to contiguous locations into a single memory transaction).

3 GPU-based (Distributed) Bucket Elimination (GPU-(D)BE)

Our *GPU-based (Distributed) Bucket Elimination* framework, extends BE (resp. DPOP) by exploiting GPU parallelism within the *aggregation* and *projection* operations. These operations are responsible for the creation of the functions $\hat{f}_i$ in BE (line 3 of Algorithm 1) and the *UTIL* tables in DPOP (*UTIL* propagation phase), and they dominate the complexity of the algorithms. Thus, we focus on the details of the design and the implementation relevant to such operations. Due to the equivalence of BE and DPOP, we will refer to the *UTIL tables* and to the aggregated and projected functions $\hat{f}$ of Algorithm 1, as well as variables and agents, interchangeably. Notice that the computation of the utility for each value combination in a *UTIL* table is independent of the computation in the other combinations. The use of a GPU architecture allows us to exploit such independence, by concurrently exploring several combinations of the *UTIL* table, computed by the aggregation operator, as well as concurrently projecting out variables.

Algorithm 2 illustrates the pseudocode, where we use the following notations: Line numbers in parenthesis denote those instructions required exclusively in the distributed case. Starred line numbers denote those instructions executed concurrently by both the CPU and the GPU. The symbols $\leftarrow$ and $\leftleftarrows$ denote sequential and parallel (multiple GPU-threads) operations, respectively. If a parallel operation requires a copy from host

```
Procedure UTILMessageHandler($a_k$, $UTIL_{a_k}$)
(9)  Store $UTIL_{a_k}$
10   if received UTIL message from each child $a_c \in C_i$ then
11       $UTIL_{a_i} \leftleftarrows$ ParallelCalcUtils( )
12       if $P_i = NULL$ then
13           $d_i^* \leftarrow$ ChooseBestValue($\emptyset$)
(14)         foreach $a_c \in C_i$ do
(15)             $VALUE_{a_i} \leftarrow (x_i, d_i^*)$
(16)             Send VALUE message $(a_i, VALUE_{a_i})$ to $a_c$
(17)     else Send UTIL message $(a_i, UTIL_{a_i})$ to $P_i$
```

```
Procedure VALUEMessageHandler($a_k$, $VALUE_{a_k}$)
(18) $VALUE_{a_i} \leftarrow VALUE_{a_k}$
(19) $d_i^* \leftarrow$ ChooseBestValue($VALUE_{a_i}$)
(20) foreach $a_c \in C_i$ do
(21)     $VALUE_{a_i} \leftarrow \{(x_i, d_i^*)\} \cup \{(x_k, d_k^*) \in VALUE_{a_k} \mid x_k \in sep(a_c)\}$
(22)     Send VALUE message $(a_i, VALUE_{a_i})$ to $a_c$
```

(device) to device (host), we write $\overset{D\leftarrow H}{\leftleftarrows}$ ($\overset{H\leftarrow D}{\leftleftarrows}$). Host to device (resp. device to host) memory transfers are performed immediately before (resp. after) the execution of the GPU kernel. Algorithm 2 shows the pseudocode of GPU-(D)BE for an agent a_i. Like DPOP, also GPU-(D)BE is composed of three phases; the first and third phase are executed exclusively in the distributed version. The first phase is identical to that of DPOP (line 1). In the second phase:

- Each agent a_i calls GPU-INITIALIZE() to set up the GPU kernel. For example, it determines the amount of global memory to be assigned to each *UTIL* table and initializes the data structures on the GPU device memory (line 2).
- Each agent a_i aggregates the utilities for the functions between its variables and its separator, projects its variable out (line 4), and sends them to its parent (line 5). The *MessageHandlers* of lines 7 and 8 are activated for each new incoming message.

By the end of the second phase (line 11), the root agent knows the overall utility for each values of its variable x_i. It chooses the value that results in the maximum utility (line 13). Then, in the distributed version, it starts the third phase by sending to each child agent a_c the value of its variable x_i (lines 14-16). These operations are repeated by every agent receiving a *VALUE* message (lines 18-22). In contrast, in the centralized version, the value assignment for each variable is set by the root agent directly.

3.1 GPU Data Structures

In order to fully capitalize on the parallel computational power of GPUs, the data structures need to be designed in such a way to limit the amount of information exchanged between the CPU host and the GPU device, and in order to minimize the accesses to the (slow) device global memory (and ensure that they are coalesced). To do so, each agent identifies the set of relevant *static entities*, i.e., information required during the GPU computation, which does not mutate during the resolution process. The static entities are communicated to the GPU once at the beginning of the computation. This allows each agent running on a GPU device to communicate with the CPU host exclusively to exchange the results of the aggregation and projection processes. The complete set of utility functions, the constraint graph, and the agents ordering, all fall in such category. Thus, each agent a_i stores:

- The set of utility functions involving exclusively x_i and a variable in a_i's separator set: $S_i = \{f_j \in \mathbf{C} \mid x_i \in \mathbf{x}^j \wedge sep(a_i) \cap \mathbf{x}^j \neq \emptyset\}$. For a given function $f_j \in S_i$, its utility values are stored in an array named $gFunc_j$.
- The domain D_i of its variable (for simplicity assumed to be all of equal cardinality).
- The set C_i of a_i's children.
- The separator sets $sep(a_i)$, and $sep(a_c)$, for each $a_c \in C_i$.

The GPU-INITIALIZE() procedure of line 2, invoked after the pseudo-tree construction, stores the data structures above for each agent on the GPU device. As a technical detail, all the data stored on the GPU global memory is organized in mono-dimensional arrays, so as to facilitate *coalesced memory accesses*. In particular, the identifier and scope of the functions in S_i as well as identifiers and separator sets of child agents in C_i are stored within a single mono-dimensional array. The utility values stored in the rows of each function are padded to ensures that a row is aligned to a memory word—thus minimizing the number of memory accesses.

GPU-INITIALIZE() is also responsible for reserving a portion of the GPU global memory to store the values for the agent's *UTIL* table, denoted by $gUtils_i$, and those of its children, denoted by $gChUtils_c$, for each $a_c \in C_i$. As a technical note, an agent's *UTIL* table is mapped onto the GPU device to store only the utility values, not the associated variables values. Its j-th entry is associated with the j-th permutation of the variable values in $sep(a_i)$, in lexicographic order. This strategy allows us to employ a simple perfect hashing to efficiently associate row numbers with variables' values and vice versa. Note that the agent's *UTIL* table size grows exponentially with the size of its separator set; more precisely, after projecting out x_i, it has $|D_i|^{sep(a_i)}$ entries. However, the GPU global memory is typically limited to a few GB (e.g., in our experiments it is 2GB). Thus, each agent, after allocating its static entities, checks if it has enough space to allocate its children's *UTIL* tables and a consistent portion (see next subsection for details) of its own *UTIL* table. In this case, it sets the $project\_on\_device$ flag to true, which signals that both aggregate and project operations can be done on the GPU device.[5] Otherwise it sets the flag to false and bounds the device *UTIL* size table to the maximum storable space on the device. In this case, the aggregation operations are performed only partially on the GPU device.

3.2 Parallel Aggregate and Project Operations

The PARALLELCALCUTILS procedure (executed in lines 4 and 11) is responsible for performing the aggregation and projection operations, harnessing the parallelism provided by the GPU. Due to the possible large size of the *UTIL* tables, we need to separate two possible cases and devise specific solutions accordingly:

[5] If the *UTIL* table of agent a_i does not fit in the global memory, we partition such table in smaller chunks, and iteratively execute the GPU kernel until all rows of the table are processed.

```
Procedure ParallelCalcUtils( )
23  if $project\_on\_device$ then
24      $gChUTIL_{a_c} \overset{D\leftarrow H}{\leftleftarrows} UTIL_{a_c}$    for all $a_c \in C_i$
25  $R \leftarrow 0\,; UTIL_{a_i} \leftarrow \emptyset$
26  while $R < |D_i|^{sep(a_i)}$ do
27      if $project\_on\_device$ then
28*         $UTIL'_{a_i} \overset{H\leftarrow D}{\leftleftarrows}$ GPU-AGGREGATE-PROJECT($R$)
29      else
30*         $UTIL'_{a_i} \overset{H\leftarrow D}{\leftleftarrows}$ GPU-AGGREGATE($R$)
31*         $UTIL'_{a_i} \leftarrow$ AGGREGATECH-PROJECT($a_i, UTIL'_{a_i}, UTIL_{a_c}$)    for all $a_c \in C_i$
32*     $UTIL_{a_i} \leftarrow UTIL_{a_i} \cup$ COMPRESS($UTIL'_{a_i}$)
33      $R \leftarrow R + |UTIL'_{a_i}|$
34  return $UTIL_{a_i}$
```

(a) When the device global memory is sufficiently large to store all a_i's children *UTIL* tables as well as a significant portion of a_i's *UTIL* table[6] (i.e., when $project\_on\_device$ = `true`), both aggregation and projection of the agent's *UTIL* table are performed in parallel on the GPU. The procedure first stores the *UTIL* tables received from the children of a_i into their assigned locations in the GPU global memory (lines 23-24). It then iterates through successive GPU kernel calls (line 28) until the $UTIL_{a_i}$ table is fully computed (lines 26-33). Each iterations computes a certain number of rows of the $UTIL_{a_i}$ table (R serves as counter).

(b) When the device global memory is insufficiently large to store all a_i's children *UTIL* tables as well as a significant portion of a_i's *UTIL* table (i.e., when $project\_on\_device$ = `false`), the agent alternates the use of the GPU and the CPU to compute $UTIL_{a_i}$. The GPU is in charge of aggregating the functions in S_i (line 30), while the CPU aggregates the children *UTIL* table,[7] projecting out x_i. Note that, in this case, the $UTIL_{a_i}$ storage must include all combinations of values for the variables in $sep(x_i) \cup \{x_i\}$, thus the projection operation is performed on the CPU host. As in the previous case, the $UTIL_{a_i}$ is computed incrementally, given the amount of available GPU global memory.

To fully capitalize on the use of the GPU, we exploit an additional level of parallelism, achieved by running GPU kernels and CPU computations concurrently; this is possible when the $UTIL_{a_i}$ table is computed in multiple chunks. Fig. 2 illustrates the concurrent computations between the CPU and GPU. After transferring the children *UTIL* tables into the device memory (Init)—in case **(a)** only—the execution of kernel K_1 produces the update of the first chunk of $UTIL_{a_i}$, denoted by U_1 in Fig. 2, which is transferred to the CPU host. The successive parallel operations are performed

[6] In our experiments, we require that at least 1/10 of the *UTIL* table can be stored in the GPU. We experimentally observed that a partitioning of the table in at most 10 chunks provides a good time balance between memory transfers and actual computation.

[7] The CPU aggregates only those child *UTIL* table that could not fit in the GPU memory. Those that fit in memory are integrated through the GPU computation as done in the previous point.

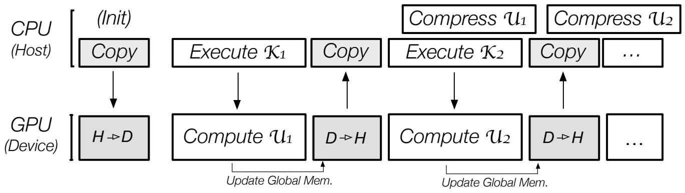


Fig. 2: Concurrent computation between host and device.

asynchronously with respect to the GPU, that is, the execution of the j-th CUDA kernel K_j ($j > 1$), returns the control immediately to the CPU, which concurrently operates a compression operation on the previously computed *UTIL*$'_{a_i}$ chunk (line 32), referred to as U_{k-1} in Fig. 2. For case **(b)**, the CPU also executes concurrently the AGGREGATECH-PROJECT of line 31. We highlight the concurrent operations by marking with a * symbol their respective lines in the procedure PARALLELCALCUTILS.

Technical Details: We now describe in more detail how we divide the workload among parallel blocks, i.e., the mapping between the *UTIL* table rows and the CUDA blocks. A total of $T = 64 \cdot k$ ($1 \leq k \leq 16$) threads (a block) are associated to the computation of T permutations of values for $sep(a_i)$. The value k depends on the architecture and it is chosen to maximize the number of concurrent threads running at the same time. In our experiments, we set $k = 3$. The number of blocks is chosen so that the corresponding aggregate number of threads does not exceed the total number of *UTIL*$'_{a_i}$ permutations currently stored in the device. Let h be the number of stream multiprocessors of the GPU. Then, the maximum number of *UTIL* permutations that can be computed concurrently is $M = h \cdot T$. In our experiments $h = 14$, and thus, $M = 2688$. Fig. 3 provides an illustration of the *UTIL* permutations computed in parallel on GPU. The blocks B_i in each row are executed in parallel on different SMs. Within each block, a total of (at most) 192 threads operate on as many entries of the *UTIL* table.

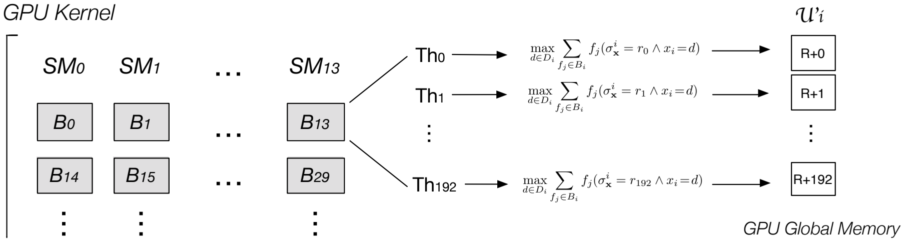

GPU Kernel
SM0
SM1
...
SM13
B0
B1
...
B13
B14
B15
...
B29
Th0
Th1
Th192
$\max_{d \in D_i} \sum_{f_j \in B_i} f_j(\sigma^i_{\mathbf{x}} = r_0 \wedge x_i = d)$
$\max_{d \in D_i} \sum_{f_j \in B_i} f_j(\sigma^i_{\mathbf{x}} = r_1 \wedge x_i = d)$
$\max_{d \in D_i} \sum_{f_j \in B_i} f_j(\sigma^i_{\mathbf{x}} = r_{192} \wedge x_i = d)$
$\mathcal{U}'i$
R+0
R+1
R+192
GPU Global Memory

Fig. 3: GPU kernel parallel computations.

The GPU kernel procedure is shown in lines 35-49. We surround line numbers with $|\cdot|$ to denote parts of the procedure executed by case **(b)**. The kernel takes as input the number R of the *UTIL* table permutations computed during the previous kernel calls. Each thread identifies its entry index r_{id} within the table chunk *UTIL*$'_{a_i}$ (line 35). It then assigns the shared memory allocated to local arrays to store the static entities S_i, C_i, and $sep(a_c)$, for each $a_c \in C_i$. In addition it reserves the space θ to store the assignments corresponding to the *UTIL* permutation being computed by each thread,

Procedure GPU-Aggregate-Project(R)

```
|35| $r_{id} \leftarrow$ the thread's entry index of $\mathit{UTIL}'_i$
|36| $d_{id} \leftarrow$ the thread's value index of $D_i$
|37| $\langle |\theta, S_i|, C_i, sep(x_c) \rangle \leftarrow$ AssignSharedMem()    for all $x_c \in C_i$
|38| $\theta \leftarrow \text{DECODE}(R + r_{id})$
|39| $util \leftarrow -\infty$
 40  foreach $d_{id} \in D_i$ do
|41|     $util_{d_{id}} \leftarrow 0$
|42|     foreach $f_j \in S_i$ do
|43|         $\rho_j \leftarrow \text{ENCODE}(\theta_{\mathbf{x}^j} \mid x_i = d_{id})$
|44|         $util_{d_{id}} \leftarrow util_{d_{id}} + gFunc_j[\rho_j]$
 45      foreach $a_c \in C_i$ do
 46          $\rho_c \leftarrow \text{ENCODE}(\theta_{sep(a_c)} \mid x_i = d_{id})$
 47          $util_{d_{id}} \leftarrow util_{d_{id}} + gChUtils_c[\rho_c]$
|48|     $util \leftarrow \max(util, util_{d_{id}})$
|49| $gUtils_i[r_{id}] \leftarrow util$
```

which is retrieved using the thread entry index and the offset R (line 38). Decode implements a *minimal perfect hash function* to convert the entry index of the *UTIL* table to its associated variables value permutation. Each thread aggregates the functions in S_i (lines 42-44) and the *UTIL* tables of a_i's children (lines 45-47), for each element of its domain (lines 40-48). The Encode routine converts a given assignments for the variables in the scope of a function f_j (line 43), or in the separator set of child a_c (line 46), to the corresponding array index, sorted in lexicographic order. The value for the variable x_i within each input, is updated at each iteration of the for loop. The projection operation is executed in line 48. Finally, the thread stores the best utility in the corresponding position of the array $gUtils_i$

The GPU-Aggregate procedure (called in line 30), is illustrated in lines 35-49—line numbers surrounded by $| \cdot |$. Each thread is in charge of a value combination in $sep(a_i) \cup \{x_i\}$, thus, the **foreach** loop of lines 40-48 is operated in parallel by $|D_i|$ threads. Lines 45-47 are not executed. The AggregateCh-Project procedure (line 31), which operates on the CPU, is similar to the GPU-Aggregate-Project procedure, except that lines 36-37, and 42-44, are not executed.

The proposed kernel has been the result of several investigations. We experimented with other levels of parallelism, e.g., by unrolling the for-loops among groups of threads. However, these modifications create divergent branches, which degrade the parallel performance. We experimentally observed that such degradation worsen consistently as the size of the domain increases.

3.3 General Observations

Observation 1. *GPU-DBE requires the same number of messages as those required by DPOP, and it requires messages of the same size as those required by DPOP.*

Observation 2. *The UTIL messages constructed by each GPU-DBE agent are identical to those constructed by each corresponding DPOP agent.*

The above observations follow from the pseudo-tree construction and VALUE propagation GPU-DBE phases, which are identical to those of DPOP. Thus, their corresponding messages and message sizes are identical in both algorithms. Moreover, given a pseudo-tree, each DPOP/GPU-DBE agent computes the *UTIL* table containing each combination of values for the variables in its separator set. Thus, the *UTIL* messages of GPU-DBE and DPOP are identical.

Observation 3. *The memory requirements of GPU-(D)BE is, in the worst case, exponential in the induced width of the problem (for each agent).*

This observation follows from the equivalence of the *UTIL* propagation phase of DPOP and BE [6] and from Observation 2.

Observation 4. *GPU-(D)BE is complete and correct.*

The completeness and correctness of GPU-(D)BE follow from the completeness and correctness of BE [10] and DPOP [25].

4 Related Work

The use of GPUs to solve difficult combinatorial problems has been explored by several proposals in different areas of constraint optimization. For instance, Meyer *et al.* [18] proposed a multi-GPU implementation of the *simplex tableau* algorithm which relies on a vertical problem decomposition to reduce communication between GPUs. In constraint programming, Arbelaez and Codognet [3] proposed a GPU-based version of the *Adaptive Search* that explores several *large neighborhoods* in parallel, resulting in a speedup factor of 17. Campeotto *et al.* [9] proposed a GPU-based framework that exploits both parallel propagation and parallel exploration of several large neighborhoods using local search techniques, leading to a speedup factor of up to 38. The combination of GPUs with dynamic programming has also been explored to solve different combinatorial optimization problems. For instance, Boyer *et al.* [5] proposed the use of GPUs to compute the classical DP recursion step for the knapsack problem, which led to a speedup factor of 26. Pawłowski *et al.* [23] presented a DP-based solution for the *coalition structure formation problem* on GPUs, reporting up to two orders of magnitude of speedup. Differently from other proposals, our approach aims at using GPUs to exploit SIMT-style parallelism from DP-based methods to solve general COPs and DCOPs.

5 Experimental Results

We compare our centralized and distributed versions of GPU-(D)BE with BE [10] and DPOP [25] on binary constraint networks with *random*, *scale-free*, and *regular grid* topologies. The instances for each topology are generated as follows:

Random: We create an n-node network, whose density p_1 produces $\lfloor n\,(n-1)\,p_1 \rfloor$ edges in total. We do not bound the tree-width, which is based on the underlying graph.

Scale-free: We create an n-node network based on the Barabasi-Albert model [4]: Starting from a connected 2-node network, we repeatedly add a new node, randomly connecting it to two existing nodes. In turn, these two nodes are selected with probabilities that are proportional to the numbers of their connected edges. The total number of edges is $2\,(n-2)+1$.

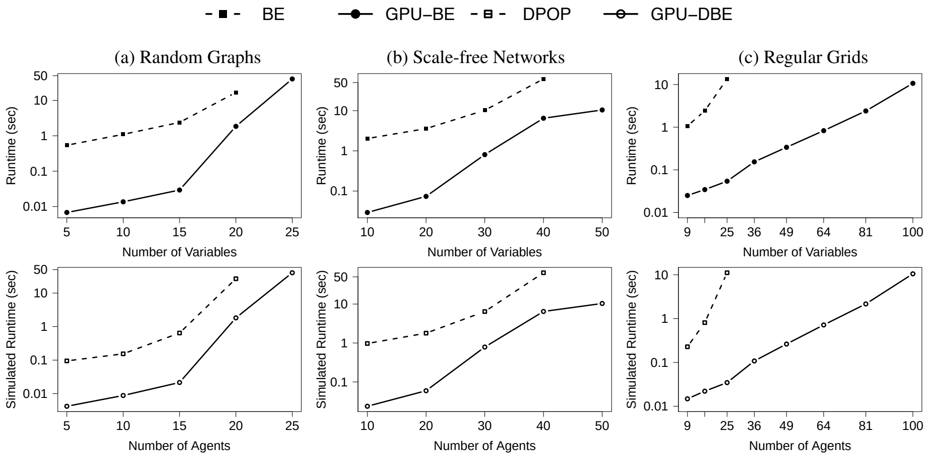


Fig. 4: Runtimes for COPs (top) and DCOPs (bottom) at varying number of variables/agents.

Regular grid: We create an n-node network arranged as a rectangular grid, where each internal node is connected to four neighboring nodes, while nodes on the grid edges (resp. corners) are connected to two (resp. three) neighboring nodes.

We generate 30 instances for each topology, ensuring that the underlying graph is connected. The utility functions are generated using random integer costs in $[0, 100]$, and the constraint tightness (i.e., ratio of entries in the utility table different from $-\infty$) p_2 is set to 0.5 for all experiments. We set as default parameters, $|\mathbf{A}| = |\mathbf{X}| = 10$, $|D_i| = 5$ for all variables, and $p_1 = 0.3$ for random networks, and $|\mathbf{A}| = |\mathbf{X}| = 9$ for regular grids. Experiments for GPU-DBE are conducted using a multi-agent DCOP simulator, that simulates the concurrent activities of multiple agents, whose actions are activated upon receipt of a message. We use the publicly-available implementation of DPOP available in the FRODO framework v.2.11 [19], and we use the same framework to run the BE algorithm, in a centralized setting.

Since all algorithms are complete, our focus is on runtime. Performance of the centralized algorithms are evaluated using the algorithm's wallclock runtime, while distributed algorithms' performances are evaluated using the *simulated runtime* metric [30]. We imposed a timeout of 300s of wallclock (or simulated) time and a memory limit of 32GB. Results are averaged over all instances and are statistically significant with p-values $< 1.638\,e^{-12}$.[8] These experiment are performed on an *AMD Opteron 6276*, 2.3GHz, 128GB of RAM, which is equipped with a GPU device *GeForce GTX TITAN* with 14 multiprocessors, 2688 cores, and a clock rate of 837MHz.

Fig. 4 illustrates the runtime, in seconds, for random (a), scale-free (b), and regular grid (c) topologies, varying the number of variables (resp. agents) for the centralized (resp. distributed) algorithms. The centralized algorithms (BE and GPU-BE) are shown at the top of the figure, while the distributed algorithms (DPOP and GPU-DBE) are illustrated at the bottom. All plots are in log-scale. We make the following observations:

- The GPU-based DP-algorithms (for both centralized and distributed cases) are consistently faster than the non-GPU-based ones. The speedups obtained by GPU-BE

[8] t-test performed with null hypothesis: GPU-based algorithms are faster than non-GPU ones.

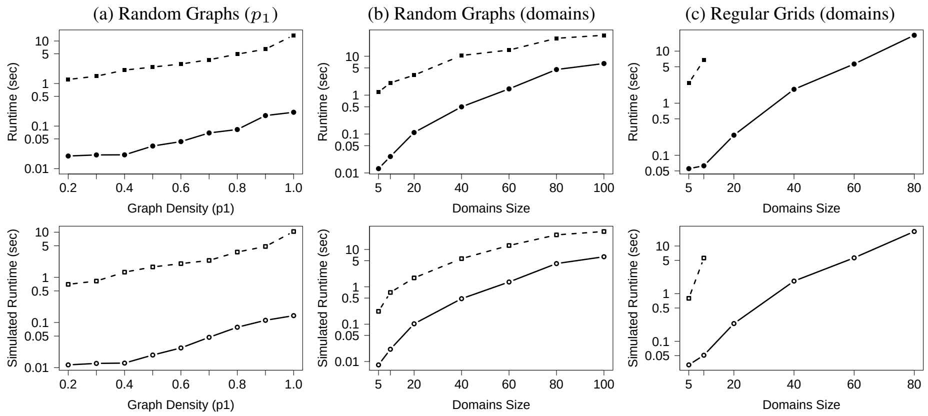


Fig. 5: Runtimes for COPs (top) and DCOPs (bottom) at varying number of variables/agents.

vs. BE are, on average, and minimum (showed in parenthesis) 69.3 (16.1), 34.9 (9.5), and 125.1 (42.6), for random, scale-free, and regular grid topologies, respectively. For the distributed algorithms, the speedups obtained by GPU-DBE vs. DPOP are on average (minimum) 44.7 (14.7), 22.3 (8.2), and 124.2 (38.8), for random, scale-free, and regular grid topologies, respectively.

- In terms of scalability, the GPU-based algorithms scale better than the non-GPU-based ones. In addition, their scalability increases with the level of structure exposed by each particular topology. On random graphs, which have virtually no structure, the GPU-based algorithms reach a timeout for instances with small number of variables (25 variables—compared to 20 variables for the non-GPU-based algorithms). On scale-free networks, the GPU-(D)BE algorithms can solve instances up to 50 variables,[9] while BE and DPOP reach a timeout for instances greater than 40 variables. On regular grids, the GPU-based algorithms can solve instances up to 100 variables, while the non-GPU-based ones, fail to solve any instance with 36 or more variables. We relate these observations to the size of the separator sets and, thus, the size of the *UTIL* tables that are constructed in each problem. In our experiments, we observe that the average sizes of the separator sets are consistently larger in random graphs, followed by scale-free networks, followed by regular grids.
- Finally, the trends of the centralized algorithms are similar to those of the distributed algorithms: The simulated runtimes of the DCOP algorithms are consistently smaller than the wallclock runtimes of the COP ones.

Fig. 5 illustrates the behavior of the algorithms when varying the graph density p_1 for the random graphs (a), and the domains size for random graphs (b) and regular grids (c). As for the previous experiments, the centralized (resp. distributed) algorithms are shown on the top (resp. bottom) of the figure. We can observe:

- The trends for the algorithms runtime, when varying both p_1 and domains size, are similar to those observed in the previous experiments.
- GPU-(D)BE achieves better speed-up for smaller p_1 (Fig. 4 (a)). The result is explained by observing that small p_1 values correspond to smaller induced width of

[9] With 60 variables, we reported 12/30 instances solved for GPU-(D)BE.

the underlying constraint graph. In turn, for small p_1 values, GPU-(D)BE agents construct smaller *UTIL* tables, which increases the probability of performing the complete inference process on the GPU, through the GPU-AGGREGATE-PROJECT procedure. This observation is also consistent with what observed in the previous experiments in terms of scalability.

- GPU-(D)BE achieves greater speedups in presence of large domains. This is due to the fact that large domains correspond to large *UTIL* tables, enabling the GPU-based algorithms to exploit a greater amount of parallelism, provided that the *UTIL* tables can be stored in the global memory of the GPU.

6 Conclusions and Discussions

In this paper, we presented an investigation of the use of GPUs to exploit SIMT-style parallelism from DP-based methods to solve COPs and DCOPs. We proposed a procedure, inspired by BE (for COPs) and DPOP (for DCOPs), that makes use of multiple threads to parallelize the aggregation and projection phases. Experimental results show that the use of GPUs may provide significant advantages in terms of runtime and scalability. The proposed results are significant—the wide availability of GPUs provides access to parallel computing solutions that can be used to improve efficiency of (D)COP solvers. Furthermore, GPUs are renowned for their complex architectures (multiple memory levels with very different size and speed characteristics; relatively slow cores), which often create challenges to the effective exploitation of parallelism from irregular applications; the strong experimental results indicate that the proposed algorithms are well-suited to GPU architectures. While envisioning further research in this area, we anticipate several challenges:

- In terms of implementation, GPU programming can be more demanding when compared to a classical sequential implementation. One of the current limitations for (D)COP-based GPU approaches is the absence of solid abstractions that allow component integration, modularly, without restructuring the whole program.
- Exploiting the integration of CPU and GPU computations is a key factor to obtain competitive solvers performance. Complex and repeated calculations should be delegated to GPUs, while simpler and memory intensive operations should be assigned to CPUs. It is however unclear how to determine good tradeoffs of such integrations. For instance, repeatedly invoking many memory demanding GPU kernels could be detrimental to the overall performance, due to the high cost of allocating the device memory (e.g., shared memory). Creating lightweight communication mechanisms between CPU and GPU (for instance, by taking advantage of the asynchronism of CUDA streams) to allow active GPU kernels to be used in multiple instances could be a possible solution to investigate.
- While this paper describes the applicability of our approach to BE and DPOP, we believe that analogous techniques can be derived and applied to other DP-based approaches to solve (D)COPs—e.g., to implement the logic of DP-based propagators. We also envision that such technology could open the door to efficiently enforcing higher form of consistencies than domain consistency (e.g., *path consistency* [22], *adaptive consistency* [12], or the more recently proposed *branch consistency* for DCOPs [14]), especially when the constraints need to be represented explicitly.

References

1. S. Abdennadher and H. Schlenker. Nurse Scheduling using Constraint Logic Programming. In *Proceedings of the Conference on Innovative Applications of Artificial Intelligence (IAAI)*, pages 838–843, 1999.
2. K. Apt. *Principles of constraint programming*. Cambridge University Press, 2003.
3. A. Arbelaez and P. Codognet. A GPU implementation of parallel constraint-based local search. In *Proceedings of the Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP)*, pages 648–655, 2014.
4. A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
5. V. Boyer, D. El Baz, and M. Elkihel. Solving knapsack problems on GPU. *Computers & Operations Research*, 39(1):42–47, 2012.
6. I. Brito and P. Meseguer. Improving DPOP with function filtering. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 141–158, 2010.
7. D. Burke and K. Brown. Efficiently handling complex local problems in distributed constraint optimisation. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 701–702, 2006.
8. E. K. Burke, P. De Causmaecker, G. V. Berghe, and H. Van Landeghem. The state of the art of nurse rostering. *Journal of scheduling*, 7(6):441–499, 2004.
9. F. Campeotto, A. Dovier, F. Fioretto, and E. Pontelli. A GPU implementation of large neighborhood search for solving constraint optimization problems. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 189–194, 2014.
10. R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Learning in graphical models*, pages 75–104. Springer, 1998.
11. R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.
12. R. Dechter and J. Pearl. *Network-based heuristics for constraint-satisfaction problems*. Springer, 1988.
13. A. Farinelli, A. Rogers, A. Petcu, and N. Jennings. Decentralised coordination of low-power embedded devices using the Max-Sum algorithm. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 639–646, 2008.
14. F. Fioretto, T. Le, W. Yeoh, E. Pontelli, and T. C. Son. Improving DPOP with branch consistency for solving distributed constraint optimization problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 307–323, 2014.
15. J. Gaudreault, J.-M. Frayret, and G. Pesant. Distributed search for supply chain coordination. *Computers in Industry*, 60(6):441–451, 2009.
16. Y. Hamadi, C. Bessière, and J. Quinqueton. Distributed intelligent backtracking. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 219–223, 1998.
17. A. Kumar, B. Faltings, and A. Petcu. Distributed constraint optimization with structured resource constraints. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 923–930, 2009.
18. M. E. Lalami, D. El Baz, and V. Boyer. Multi GPU implementation of the simplex algorithm. In *Proceedings of the International Conference on High Performance Computing and Communication (HPCC)*, volume 11, pages 179–186, 2011.
19. T. Léauté and B. Faltings. Distributed constraint optimization under stochastic uncertainty. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 68–73, 2011.

20. R. Maheswaran, M. Tambe, E. Bowring, J. Pearce, and P. Varakantham. Taking DCOP to the real world: Efficient complete solutions for distributed event scheduling. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 310–317, 2004.
21. P. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1–2):149–180, 2005.
22. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information sciences*, 7:95–132, 1974.
23. K. Pawłowski, K. Kurach, T. Michalak, and T. Rahwan. Coalition structure generation with the graphic processor unit. Technical Report CS-RR-13-07, Department of Computer Science, University of Oxford, 2104.
24. G. Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 482–495, 2004.
25. A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1413–1420, 2005.
26. C.-G. Quimper and T. Walsh. Global grammar constraints. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 751–755. Springer, 2006.
27. L. Rodrigues and L. Magatao. Enhancing Supply Chain Decisions Using Constraint Programming: A Case Study. In *MICAI 2007: Advances in Artificial Intelligence*, volume LNCS 4827, pages 1110–1121. Springer Verlag, 2007.
28. F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.
29. J. Sanders and E. Kandrot. *CUDA by Example. An Introduction to General-Purpose GPU Programming*. Addison Wesley, 2010.
30. E. Sultanik, P. J. Modi, and W. C. Regli. On modeling multiagent task scheduling as a distributed constraint optimization problem. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1531–1536, 2007.
31. M. A. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research*, 118(1-4):73–84, 2003.
32. W. Yeoh, A. Felner, and S. Koenig. BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research*, 38:85–133, 2010.
33. W. Yeoh and M. Yokoo. Distributed problem solving. *AI Magazine*, 33(3):53–65, 2012.
34. M. Yokoo, editor. *Distributed Constraint Satisfaction: Foundation of Cooperation in Multi-agent Systems*. Springer, 2001.
35. R. Zivan, S. Okamoto, and H. Peled. Explorative anytime local search for distributed constraint optimization. *Artificial Intelligence*, 212:1–26, 2014.
36. R. Zivan, H. Yedidsion, S. Okamoto, R. Glinton, and K. Sycara. Distributed constraint optimization for teams of mobile sensing agents. *Journal of Autonomous Agents and Multi-Agent Systems*, 29(3):495–536, 2015.