A Dynamic Programming-based MCMC Framework for Solving DCOPs with GPUs*

Ferdinando Fioretto[1,2], William Yeoh[1], and Enrico Pontelli[1]

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

Abstract. The field of *Distributed Constraint Optimization* (DCOP) has gained momentum in recent years, thanks to its ability to address various applications related to multi-agent coordination. Nevertheless, solving DCOPs is computationally challenging. Thus, in large scale, complex applications, incomplete DCOP algorithms are necessary. Recently, researchers have introduced a promising class of incomplete DCOP algorithms, based on *sampling*. However, this paradigm requires a multitude of samples to ensure convergence. This paper exploits the property that sampling is amenable to parallelization, and introduces a general framework, called *Distributed MCMC* (DMCMC), that is based on a dynamic programming procedure and uses *Markov Chain Monte Carlo* (MCMC) sampling algorithms to solve DCOPs. Additionally, DMCMC harnesses the parallel computing power of *Graphical Processing Units* (GPUs) to speed-up the sampling process. The experimental results show that DMCMC can find good solutions up to two order of magnitude faster than other incomplete DCOP algorithms.

1 Introduction

In a *Distributed Constraint Optimization Problem* (DCOP), multiple agents coordinate assignments of values to their variables to maximize the sum of the resulting constraint utilities [18, 32]. DCOP is a powerful paradigm to describe and solve many practical problems in a variety of application domains, such as distributed scheduling, coordination of unmanned air vehicles, smart grid electrical networks, and sensor networks [10, 24, 28, 34]. DCOP researchers have proposed a wide variety of solution approaches, from distributed search-based algorithms [18, 15, 31] to distributed inference-based algorithms [21, 30], as well as solvers that use GPUs [3, 4] and logic programming [13, 12] formulations. Complete DCOP algorithms find optimal solutions at the cost of large runtimes, while incomplete approaches trade optimality for faster execution. Since finding optimal DCOP solutions is NP-hard, incomplete algorithms are often necessary to solve larger problems. A further challenge to the applicability of DCOPs to more general classes of problems is the common assumption that each agent controls exactly

* This research is partially supported by the National Science Foundation under grants 1345232 and 1550662. 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.

one variable during problem resolution, which is often unrealistic. To cope with such restrictions, reformulation techniques are commonly adopted to transform a general DCOP into one where each (pseudo-)agent controls exclusively one variable [1, 33]. This transformation can be inefficient in terms of agent computation and coordination, as it may limit the agents' ability to interact in pruning the search space [5]. While one can trivially extend existing algorithms to allow each agent to solve its local sub-problem (i.e., the value assignment of its local variables) in a centralized fashion, each sub-problem is still NP-hard, and can require a large amount of time if solved naively. This concern is true especially for application domains where agents may control a large number of local variables with large numbers of local constraints. We explore meeting scheduling problems as one such application domain in our experimental evaluations.

In this paper, we introduce a general framework, called *Distributed MCMC* (DMCMC), which is based on a *Dynamic Programming*-based DCOP procedure [21]; the framework allows each agent to solve its local sub-problem using *Markov Chain Monte Carlo* (MCMC) sampling algorithms and uses general-purpose *Graphical Processing Units* (GPUs) to parallelize and speed up this process. We demonstrate the generality of this framework using two popular MCMC algorithms, the Gibbs [6] and Metropolis-Hastings [8, 17] algorithms. Our experiments show that our framework is able to find better solutions up to two orders of magnitude faster than MGM and MGM2 (two incomplete DCOP algorithms). Additionally, it finds solutions that are within a 5% error of the optimum for problems that can be solved optimally. While the description of our solution focuses on DCOPs, our approach is also suitable to solve *Weighted Constraint Satisfaction Problems* (WCSPs).

2 Background

WCSPs: A *Weighted Constraint Satisfaction Problem* (WCSP) [27, 11] is a tuple $\langle \mathcal{X}, \mathcal{D}, \mathcal{F} \rangle$, where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a finite set of variables, $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of finite domains for the variables in $\mathcal{X}$, with D_i being the set of possible values for the variable x_i, $\mathcal{F}$ is a set of *weighted constraints* (or *utility functions*). A weighted constraint $f_i \in \mathcal{F}$ is a function, $f_i : \bigtimes_{x_j \in \mathbf{x}^{f_i}} D_i \rightarrow \mathbb{R}^+ \cup \{-\infty\}$, where $\mathbf{x}^{f_i} \subseteq \mathcal{X}$ is the set of variables relevant to f_i, referred to as the *scope* of f_i. A *solution* σ is a value assignment to a set of variables $X_\sigma \subseteq \mathcal{X}$ that is consistent with the variables' domains. The utility $\mathcal{U}(\sigma) = \sum_{f \in \mathcal{F}, \mathbf{x}^f \subseteq X_\sigma} f(\sigma)$ is the sum of the utilities of all the applicable utility functions in σ. A solution is said *complete* if $X_\sigma = \mathcal{X}$. The goal is to find an optimal complete solution $\sigma^* = \operatorname{argmax}_\sigma \mathcal{U}(\sigma)$.

DCOPs: When the elements of a WCSP are distributed among a set of autonomous agents, we refer to it as a *Distributed Constraint Optimization Problem* (DCOP) [18, 21, 32]. Formally, a DCOP is described by a tuple $\langle \mathcal{X}, \mathcal{D}, \mathcal{F}, \mathcal{A}, \alpha \rangle$, where $\mathcal{X}$, $\mathcal{D}$ and $\mathcal{F}$ are the set of variables, their domains, and the set of utility functions, defined as in a classical WCSP, $\mathcal{A} = \{a_1, \ldots, a_m\}$ $(m \leq n)$ is a set of autonomous agents, and $\alpha : \mathcal{X} \rightarrow \mathcal{A}$ is a surjective function, from variables to agents, which assigns the control of each variable $x \in \mathcal{X}$ to an agent $\alpha(x)$. The goal in a DCOP is to find a complete solution that maximizes its utility: $\sigma^* = \operatorname{argmax}_\sigma \mathcal{U}(\sigma)$.

Given a DCOP P, $G = (\mathcal{A}, E)$ is the *constraint graph* of P, where $(i, j) \in E$ iff

x_2	x_4	Utilities
0	0	max(60,24,50,19) = 60
0	1	max(50,19,40,14) = 50
1	0	max(50,19,40,14) = 50
1	1	max(40,14,30, 9) = 40

(a) Computations of a_3

x_2	Utilities
0	max(100,66,90,61) = 100
1	max(80,51,70,46) = 80

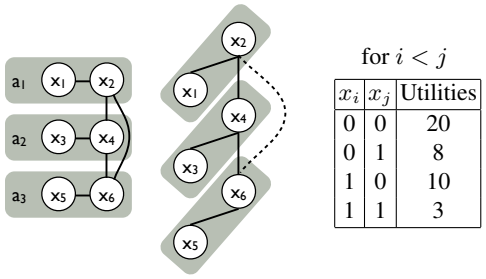
(b) Computations of a_2

Utilities
max(120,110) = 120

(c) Computations of a_1

Fig. 2: Example UTIL Phase Computations

$\exists f \in \mathcal{F}$, where $\exists x_i, x_j \in \mathcal{X}$ with $\alpha(x_i) = a_i$ and $\alpha(x_j) = a_j$ s.t. $\{x_i, x_j\} \subseteq \mathbf{x}^f$. A ***DFS pseudo-tree*** arrangement for G is a *spanning tree* $T = \langle \mathcal{A}, E_T \rangle$ of G s.t. if $f \in \mathcal{F}$ and $\exists x_i, x_j \in \mathcal{X}$ with $\alpha(x_i) = a_i$ and $\alpha(x_j) = a_j$ s.t. $\{x_i, x_j\} \subseteq \mathbf{x}^f$, then x and y appear in the same branch of T. Edges of G that are *in* (resp. *out* of) E_T are called *tree edges* (resp. *backedges*). Tree edges connect a node with its parent and its children, while backedges connect a node with its *pseudo-parents* and its *pseudo-children*. We write $a_i \succ_T a_j$ if agent a_i is an ancestor of a_j in the pseudo-tree T. We use C_i, P_i, and $sep(a_i)$ to refer to, respectively, the set of child agents, the parent agent, and to the *separator* of agent a_i in the pseudo-tree. The latter is the set of variables owned by the a_i's ancestor agents that are constrained with variables owned by a_i or by its descendant agents.



(a) Constraint Graph (b) Pseudo-tree (c) Constraint Table

Fig. 1: Example DCOP

Definition 1. *For each agent $a_i \in \mathcal{A}$, $L_i = \{x_j \in \mathcal{X} \mid \alpha(x_j) = a_i\}$ is the set of its* local *variables. $B_i = \{x_j \in L_i \mid \exists x_k \in \mathcal{X} \wedge \exists f_s \in \mathcal{F} : \alpha(x_k) \neq a_i \wedge \{x_j, x_k\} \subseteq \mathbf{x}^{f_s}\}$ is the set of its* interface *variables.*

Definition 2. *For each agent $a_i \in \mathcal{A}$, its* local constraint graph *$G_i = (L_i, E_{\mathcal{F}_i})$ is a subgraph of the constraint graph, where $\mathcal{F}_i = \{f_j \in \mathcal{F} \mid \mathbf{x}^{f_j} \subseteq L_i\}$.*

Fig. 1(a) shows the constraint graph of a sample DCOP with 3 agents a_1, a_2, and a_3, where $L_1 = \{x_1, x_2\}$, $L_2 = \{x_3, x_4\}$, $L_3 = \{x_5, x_6\}$, $B_1 = \{x_2\}$, $B_2 = \{x_4\}$, and $B_3 = \{x_6\}$. The domains are $D_1 = \cdots = D_6 = \{0, 1\}$. Fig. 1(b) shows one possible pseudo-tree (the dotted line is a *backedge*). Fig. 1(c) shows the constraints.

DPOP: *Distributed Pseudo-tree Optimization Procedure* (DPOP) [21] is a complete DCOP algorithm. composed of three phases:[3]

[Phase 1] Pseudo-tree Generation: DPOP agents constructs a pseudo-tree using existing distributed pseudo-tree construction methods [7].

[3] It is a distributed variant of Bucket Elimination [2].

Algorithm 1: METROPOLIS-HASTING($\mathbf{z}$)

$\mathbf{z}^{(0)} \leftarrow$ INITIALIZE($\mathbf{z}$)
for $t = 1$ **to** T **do**
  $\mathbf{z}^* \leftarrow$ SAMPLE($q(\mathbf{z}^* \mid \mathbf{z}^{(t-1)})$)
  $\mathbf{z}^{(t)} \leftarrow \begin{cases} \mathbf{z}^* \text{with } p = \min(1, \frac{\tilde{\pi}(\mathbf{z}^*) q(\mathbf{z}^{(t-1)}, \mathbf{z}^*)}{\tilde{\pi}(\mathbf{z}^{(t-1)}) q(\mathbf{z}^*, \mathbf{z}^{(t-1)})}) \\ \mathbf{z}^{(t-1)} \text{with } 1-p \end{cases}$

for $i = 1$ **to** n **do**
  $z_i^t \leftarrow$ SAMPLE($\frac{1}{Z_\pi} \tilde{\pi}(z_i \mid z_1^t, \ldots, z_{i-1}^t, z_{i+1}^{t-1}, \ldots, z_n^{t-1})$)

[Phase 2] Utility Propagation: Each agent, starting from the leafs 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 summing the utilities of its constraints with the variables in its separator and the utilities in the UTIL messages received from its children agents, and then projecting out its own variables by optimizing over them. In our example problem, agent a_3 computes the optimal utility for each value combination of variables x_2 and x_4 (see Fig. 2(a)), and sends the utilities to its parent agent a_2 in a UTIL message. Agent a_2 then computes the optimal utility for each value of the variable x_2 (see Fig. 2(b)), and sends the utilities to its parent agent a_1 in a UTIL message. Finally, agent a_1 computes the optimal utility of the entire problem (see Fig. 2(c)).
[Phase 3] Value Propagation: Each agent, starting from the root of the pseudo-tree, determines the optimal value for its variables. The root agent does so by choosing the values of its variables from its UTIL computation. In our example, agent a_1 determines that the values for both its variables leading to the largest utility are both 0 (with a overall utility of 120). It then sends the value of variable x_2 to its child agent a_2 in a VALUE message. Upon receiving the VALUE message from its parent agent, agent a_2 determines that the value with the largest utility for both its variables, assuming that $x_2 = 0$, is 0, with a utility of 100. In turn, it sends the value of variables x_2 and x_4 to its child agent a_3 in another VALUE message. Finally, upon receiving the VALUE message from its parent agent, agent a_3 determines that the value with the largest utility for both its variables, assuming that $x_2 = 0$ and $x_4 = 0$, is 0, with a utility of 60.

MCMC Sampling Algorithms: *Markov Chain Monte Carlo* (MCMC) sampling algorithms are commonly used to solve the *Maximum A Posteriori* (MAP) estimation problem. Recently, Nguyen *et al.* [19] have shown that DCOPs can be mapped to MAP estimation problems, allowing the use of MCMC algorithms to solve DCOPs. However, this mapping assumes that the constraint utilities are bounded, as they are normalized into distribution functions that MCMC algorithms aim to approximate. Therefore, MCMC algorithms cannot be used to solve DCOPs with hard constraints. Let us describe two popular MCMC algorithm—Gibbs [6] and Metropolis-Hastings [8, 17].

Suppose we have a joint probability distribution $\pi(\mathbf{z})$ over n variables, $\mathbf{z} = z_1, z_2, \ldots, z_n$, that we would like to approximate. Moreover, suppose that it is easy to evaluate $\pi(\mathbf{z})$ for any given $\mathbf{z}$ up to some normalizing constant Z_π, such that: $\pi(\mathbf{z}) = \frac{1}{Z_\pi} \tilde{\pi}(\mathbf{z})$, where $\tilde{\pi}(\mathbf{z})$ can be easily computed but Z_π may be unknown. In

order to draw the samples $\mathbf{z}$ to be fed to $\tilde{\pi}(\cdot)$, we use a *proposal distribution* $q(\mathbf{z} \mid \mathbf{z}^{(\tau)})$, from which we can easily generate samples, each depending on the current state $\mathbf{z}^{(\tau)}$ of the process. The latter can be interpreted as saying that when the process is in the state $\mathbf{z}^{(\tau)}$, we can generate a new state $\mathbf{z}$ from $q(\mathbf{z} \mid \mathbf{z}^{(\tau)})$. The proposal distribution is thus used to generate a sequence of samples $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \ldots$, which forms a Markov chain.

Algorithm 1 shows the pseudocode of the *Metropolis-Hastings* algorithm. It first initializes $\mathbf{z}^{(0)}$ to any arbitrary value of the variables $z_1, \ldots, z_n$ (line 1). Then, it iteratively generates a candidate $\mathbf{z}^*$ for $\mathbf{z}^{(t)}$ by sampling from the proposal distribution $q(\mathbf{z}^* \mid \mathbf{z}^{(t-1)})$ (line 3). The candidate sample is then accepted with probability p (line 4). If the sample is accepted, then $\mathbf{z}^{(t)} = \mathbf{z}^*$, otherwise $\mathbf{z}^{(t-1)}$ is left unchanged. This process continues for a fixed number of iterations or until convergence [25] is achieved.

The *Gibbs* sampling algorithm is a special case of the Metropolis-Hastings algorithm, where line 3 is replaced by lines 5-6. Additionally, note that Gibbs requires the computation of the normalizing constant Z_π while Metropolis-Hasting does not, as the calculation of the proposal distribution does not require that information. This is desirable when the computation of the normalizing constant becomes prohibitive (e.g., with increasing problem dimensionality). In this paper, we describe how one could parallelize the operations of MCMC sampling algorithms using GPU hardware.

GPUs: Modern *Graphics Processing Units* (GPUs) are multiprocessor devices, offering thousands of computing cores to support graphical processing. In this paper, we use the *Compute Unified Device Architecture* (CUDA) programming model proposed by NVIDIA [26], which enables the use of the multiple cores of a graphic card to accelerate general (non-graphical) applications by providing programming models and APIs that enable the full programmability of the GPU. The underlying model of parallelism supported by CUDA is *Single-Instruction Multiple-Thread* (SIMT), where the same instruction is executed by different threads that run on identical cores, while data and operands may differ from thread to thread.

A typical CUDA program is a C/C++ program that includes parts meant for execution on the CPU (referred to as the *host*) and parts meant for parallel execution on the GPU (referred as the *device*). A parallel computation is described by a collection of *kernels*, where each kernel is a function to be 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 (e.g., multiple banks that can be concurrently accessed), and capacity. Each thread stores its private variables in very fast registers. Threads within a block can communicate by reading and writing a common area of memory (called *shared memory*). Communication between blocks and communication between blocks and the host (i.e., the CPU) is realized through a large (but slow) global memory.

3 Distributed MCMC Framework

We now describe our *Distributed MCMC* (DMCMC) framework, which extends centralized MCMC sampling algorithms and DPOP. At a high level, its operations are similar to those of DPOP, except that the computation of the utility tables sent by agents during the UTIL phase is done by sampling with GPUs. Notice that the computation of

Algorithm 2: DMCMC(R, S)

7 Generate pseudo-tree
8 GPU-INITIALIZE( )
9 $\langle M_i^1, U_i^1\rangle, \ldots, \langle M_i^R, U_i^R\rangle \leftarrow$ GPU-MCMC-SAMPLE(R, S)
10 $UTIL_{a_i} \leftarrow$ GET-BEST-SAMPLE($\langle M_i^1, U_i^1\rangle, \ldots, \langle M_i^R, U_i^R\rangle$)
11 **if** $C_i = \emptyset$ **then**
12 $UTIL_{a_i} \leftarrow$ CALCUTILS( )
13 Send *UTIL* message $(a_i, UTIL_{a_i})$ to P_i
14 Activate UTILMessageHandler($\cdot$)
15 Activate VALUEMessageHandler($\cdot$)

Procedure VALUEMessageHandler($a_k, VALUE_{a_k}$)

16 $VALUE_{a_i} \leftarrow VALUE_{a_k}$
17 **for** $x_i^j \in L_i$ **do** $d_i^{j*} \leftarrow$ CHOOSEBESTVALUE($VALUE_{a_i}$) ;
18 **for** $a_c \in C_i$ **do**
19 $VALUE_{a_i} \leftarrow \{(x_i^j, d_i^{j*}) \mid x_i^j \in sep(a_c)\} \cup \{(x_k, d_k^*) \in VALUE_{a_k} \mid x_k \in sep(a_c)\}$
20 Send *VALUE* message $(a_i, VALUE_{a_i})$ to a_c

each row in a utility table is independent of the computations in the other rows. Thus, DMCMC exploits this independence and samples the values in each row in parallel.

Algorithm 2 shows the pseudocode of DMCMC for an agent a_i. It takes as inputs R, the number of sampling runs to perform from different initial value assignments, and S, the number of sampling trials. Like DPOP, DMCMC also exhibits three phases. The first phase is identical to that of DPOP (line 7). In the second phase:

- Each agent a_i calls GPU-INITIALIZE() to set up the GPU kernel specifics (e.g., number of threads and amount of shared memory to be assigned to each block, and to initialize the data structures on the GPU device memory) (line 8). The GPU kernel settings are decided according to the shared memory requirements and the number of registers used by the successive function call, in order to maximize the number of blocks that can run in parallel—this step can be automated.
- Each agent a_i, *in parallel*, calls GPU-MCMC-SAMPLE() which performs the local MCMC sampling process to compute the best utility and the corresponding solution (value assignments for all non-interface local variables $x_i^j \in L_i \setminus B_i$) for each combination of values of the interface variables $x_i^k \in B_i$ (line 9). This computation process is done via sampling with GPUs and the results are then transferred from the device to the host (line 10). In our example in Fig. 1, agent a_3 determines that its best utility is 20 if its interface variable $x_6 = 0$, and 8 if $x_6 = 1$. This utility table is stored in $UTIL_{a_i}$. Note that all the agents call this procedure immediately after the pseudo-tree is constructed. In contrast, agents in DPOP compute the best utility only after receiving UTIL messages from all children agents.
- Each agent a_i computes the utilities for the constraints between its interface variables and variables in its separator, joins them with the sampled utilities (line 12), and sends them to its parent (line 13). The agent repeats this process each time it receives a UTIL message from a child (lines 20-27).

Procedure UTILMessageHandler(a_k, $UTIL_{a_k}$)

21 Store $UTIL_{a_k}$
22 **if** received *UTIL* message from each child $a_c \in C_i$ **then**
23 $\quad UTIL_{a_i} \leftarrow$ CALCUTILS( )
24 $\quad$ **if** $P_i = NULL$ **then**
25 $\quad\quad$ **for** $x_i^j \in L_i$ **do** $d_i^{j*} \leftarrow$ CHOOSEBESTVALUE($\emptyset$) ;
26 $\quad\quad$ **for** $a_c \in C_i$ **do**
27 $\quad\quad\quad VALUE_{a_i} \leftarrow \{(x_i^j, d_i^{j*}) \mid x_i^j \in sep(a_c)\}$
28 $\quad\quad\quad$ Send *VALUE* message $(a_i, VALUE_{a_i})$ to a_c
29 $\quad$ **else** Send *UTIL* message $(a_i, UTIL_{a_i})$ to P_i ;

Function CalcUtils( )

30 $UTIL_{sep} \leftarrow$ utilities for all value comb. of $x_i \in B_i \cup sep(a_i)$
31 $UTIL_{a_i} \leftarrow$ JOIN($UTIL_{a_i}, UTIL_{sep}, UTIL_{a_c}$) for all $a_c \in C_i$
32 $UTIL_{a_i} \leftarrow$ PROJECT($a_i, UTIL_{a_i}$)
33 **return** $UTIL_{a_i}$

At the end of the second phase (line 23), like in DPOP, the root agent will know the overall utility for each combination of values of its variables $x_i^j \in B_i$. It chooses its best value combination that results in the maximum utility (line 25), and starts the third phase by sending to each child agent a_c the values of variables $x_i^j \in sep(a_c)$ that are in the separator of the child (lines 26-28). The *MessageHandlers* of lines 14 and 15 are activated for any new incoming message.

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 devices. Each DMCMC agent stores all the information it needs in the GPU global memory. This allows each agent running on a GPU device to communicate with the CPU host only once, which is at the end of the sampling process, to transfer the results. Each agent a_i maintains the following information:

- Its *local* variables $L_i \subseteq \mathcal{X}$ (including its interface variables $B_i \subseteq L_i$).
- The domains of its local variables, D_i (assumed to have all equal size for simplicity).
- A matrix M_i of size $|D_i|^{|B_i|} \times |L_i|$, where the j-th row is associated with the j-th permutation of the interface variable values, in lexicographic order, and the k-th column is associated with the k-th variable in L_i. The matrix columns associated with the local variables in L_i are initialized with random value assignments in $[0, |D_i| - 1]$. At the end of the sampling process it contains the converged domain values of the local variables for each value combination of the interface variables.
- A vector U_i of size $|D_i|^{|B_i|}$, which stores the utilities of the solutions in M_i.
- The *local constraint graph* G_i, which includes the constraints in $\mathcal{F}_i$.

The GPU-INITIALIZE() procedure of line 8 stores the data structures above for each agent on its CUDA device. All the data stored on the GPU devices is organized in mono-dimensional arrays, so as to facilitate *coalesced memory accesses*. The set of

Procedure GPU-MCMC-Sample(R, S)

34 $\langle \mathbf{z}, \mathbf{z}^*, [q, Z_\pi], G_i \rangle \leftarrow$ ASSIGNSHAREDMEM()
35 $r_{id} \leftarrow$ the thread's row index of M_i
36 $\mathbf{z} \overset{|L_i|}{\leftleftarrows} M_i[r_{id}]$
37 $\langle \mathbf{z}^*, util^* \rangle \leftarrow \langle \mathbf{z}, \sum_{f_j \in \mathcal{F}_i} f_j(\mathbf{z}_{|\mathbf{x}^{f_j}}) \rangle$
38 **for** $t = 1$ **to** S **do**
39 $\quad \mathbf{z} \overset{k}{\leftleftarrows}$ SAMPLE($q(\mathbf{z} \mid \mathbf{z}^{(t-1)})$) w/ prob. $\min\{1, \frac{\tilde{\pi}(\mathbf{z})}{\tilde{\pi}(\mathbf{z}^{(t-1)})}\}$
40 $\quad util \leftarrow \sum_{f_j \in \mathcal{F}_i} f_j(\mathbf{z}_{|\mathbf{x}^{f_j}})$
41 $\quad$ **if** $util > util^*$ **then** $\langle \mathbf{z}^*, util^* \rangle \leftarrow \langle \mathbf{z}, util \rangle$;
42 $\langle M_i^R[r_{id}], U_i^R[r_{id}] \rangle \leftarrow \langle \mathbf{z}^*, util^* \rangle$

local variables L_i are ordered, for convenience, in lexicographic order and so that the interface variables B_i are listed first.

3.2 Local Sampling Process

The GPU-MCMC-SAMPLE procedure of line 9 is the core of the local sampling algorithm, and can be performed by any MCMC sampling method. It executes S sampling trials for the subset of non-interface local variables $L_i \setminus B_i$ of agent a_i. Since the MCMC sampling procedure is stochastic, we can run R parallel sampling processes with different initial value assignments and take the best utility and corresponding solution across all runs. Each parallel run is executed by a *group of CUDA blocks*. Independent operations within each sample are also exploited in parallel using *groups of threads* within each block. For example, the proposal distribution adopted by Gibbs is computed using $|D_i|$ parallel *threads*. Fig. 4 illustrates the different parallelizations performed by the GPU-MCMC-Sample process with Gibbs.

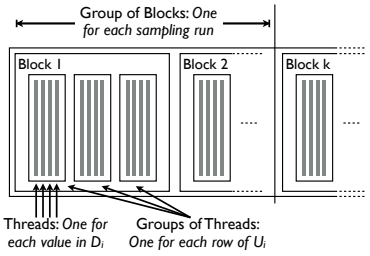


Fig. 4: Parallelization Illustration

The general GPU-MCMC-Sample procedure is shown in lines 34-42 and we use the symbols $\leftarrow$ and $\overset{k}{\leftleftarrows}$ to denote sequential (single thread) and parallel (k threads) operations, respectively. We also denote with n the size of the state $\mathbf{z}$ being sampled, with $n = |L_i| - |B_i|$. The function takes in as inputs the number of desired sampling trials S and the number of parallel sampling runs R. It first assigns the shared memory allocated to the arrays $\mathbf{z}$ and $\mathbf{z}^*$, which are used to store the current and best sample of value assignments for all local variables, respectively; the local constraint graph G_i; and, if the MCMC sampling algorithm requires computing the normalization constant of the proposal distribution explicitly, the array q and Z_π, which are used to store the probabilities for each value of the non-interface local variables and the normalization constant, respectively (line 34).

Each thread identifies its row index r_{id} of the matrix M_i, initializes its sample with the values stored in $M_i[r_{id}]$, calculates the utility for that sample, and stores the initial

Procedure CUDA Gibbs Proposal Distribution Calculation

43 $d_{id} \leftarrow$ the thread's value index of D_i
44 **for** $k = |B_i|$ **to** $|L_i| - 1$ **do**
45 $\quad q[d_{id}] \overset{|D_i|}{\leftleftarrows} \exp\left[\sum_{f_j \in \mathcal{F}_i} f_j(\mathbf{z}_{|\mathbf{x}^{f_j}})\right]$
46 $\quad Z_\pi \leftarrow \sum_{i=0}^{|D_i|-1} q[i]$
47 $\quad q[d_{id}] \overset{|D_i|}{\leftleftarrows} q[d_{id}] \cdot \frac{1}{Z_\pi}$
48 $\quad \mathbf{z} \leftarrow \text{SAMPLE}(q(\mathbf{z} \mid \mathbf{z}^{(t-1)}))$

sample and utility as the best sample and utility found so far (lines 35-37). It then runs S sampling trials, where in each trial, it samples a new state $\mathbf{z}$ from a proposal distribution $q(\mathbf{z} \mid \mathbf{z}^{(t-1)})$ and updates that state according to the accept/reject probabilities described in the MCMC background (line 39).

The proposal distribution q and the accept/reject probabilities depend on the choice of MCMC algorithm. We now describe them for Metropolis-Hasting and Gibbs.

- **Metropolis-Hastings**: The proposal distribution that we adopt is a multivariate normal distribution $q \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, with $\boldsymbol{\mu}$ being a n-dimensional vector of mean values, whose elements $\mu_j^{(t)}$ have the value of the corresponding component in the previous sample $z_j^{(t-1)}$ and $\boldsymbol{\Sigma}$ is the covariance matrix defined with the only non-zero elements being their diagonal ones and set to be all equal to $\sqrt{D_i}$. We compute the proposal distribution q using n parallel threads. The proposal distribution for Metropolis-Hastings is symmetric and, thus, the accept/reject probabilities are simplified as shown in line 39.
- **Gibbs**: For Gibbs, line 39 needs to be replaced with lines 43-48. Gibbs sequentially iterates through all the non-interface local variable $x_k \in L_i \setminus B_i$ and computes in parallel the probability $q[d_{id}]$ of each value d_{id} according to the equation:

$$q(x_k = d_{id} \mid x_l \in L_i \setminus \{x_k\}) = \frac{1}{Z_\pi} \exp \sum_{f_j \in \mathcal{F}_i} f_j(\mathbf{z}_{|\mathbf{x}^{f_j}})$$

  where $\mathbf{z}_{|\mathbf{x}^{f_j}}$ is the set of value assignments for the variables in the scope of constraint f_j, and Z_π is the normalizing constant. We compute q using $|D_i|$ parallel threads.

To ensure that the procedure returns the best sample found, we verify whether there is an improvement on the best utility (lines 40-41). At the end of the sampling trials, it stores its best sample and utility in the r_{id}-th row in the matrix M_i and vector U_i, respectively (line 42).

4 Theoretical Properties

We now relate the quality of DCOP solutions to MCMC sampling strategies, and provide some complexity analyses of the DMCMC algorithms.

Let us first introduce some background on Markov Chains and on the structural properties that they need to satisfy to guarantee convergence to a stationary distribution.

Let $\mathbf{Z} = (\mathbf{z}^0, \mathbf{z}^1, \ldots, \mathbf{z}^t, \ldots)$, with $\mathbf{z}^t \in \mathbf{D} \subseteq \mathbb{R}$ be a *Markov chain* with finite state space $\mathbf{S} = \{s_1, s_2, \ldots, s_L\}$ and a $L \times L$ *transition matrix* T whose entries define the probability of transitioning from one state to another as $P(\mathbf{z}^{t+1} = s_j \mid \mathbf{z}^t = s_i) = T_{ij}$.

The Markov chain $\mathbf{Z}$ converges to a stationary distribution if it is *irreducible* and *aperiodic*. These two concepts are introduced as follows.

Definition 3 (Irreducibility). *A Markov chain is* irreducible *if it is possible to reach any state from any other state using only transitions of positive probability. That is,* $\forall s_i, s_j \in \mathbf{S}, \exists m < \infty : P(\mathbf{z}^{t+m} = s_j \mid \mathbf{z}^t = s_i) > 0$ *for a given instance* t.

Definition 4 (Periodicy). *A state* $s_i \in \mathbf{S}$ *has a period* k *if any return of the chain in it is possible with multiple of* k *time steps. The period of a state is defined as* $k = gcd\{t : P(\mathbf{z}^t = s_i \mid \mathbf{z}^0 = s_i) > 0\}$*, where gcd is the greatest common divisor. A state is said to be aperiodic if* $k = 1$*, that is, visits of the Markov chain to such state (i.e.,* $P(\mathbf{z}^t = s_i \mid \mathbf{z}^0 = s_i) > 0$*) can occur at irregular times. A Markov chain is said to be* aperiodic *if every state in* $\mathbf{S}$ *is aperiodic.*

Note that for an irreducible Markov chain, if at least one state is aperiodic, then the whole Markov chain is aperiodic.

We now provide bounds on convergence rates for the DMCMC algorithms based on MCMC sampling.

Definition 5 (Top α_i-Percentile Solutions). *For an agent* a_i *the* top α_i-percentile solutions S_{α_i} *is a set containing solutions for the local variables* L_i *that are no worse than any solution in the supplementary set* $D_i \setminus S_{\alpha_i}$*, and* $\frac{|S_{\alpha_i|}}{|D_i|} = \alpha_i$*. Given a list of agents* $a_1, \ldots, a_m$*, the top* $\bar{\alpha}$*-percentile solutions* $S_{\bar{\alpha}}$ *is defined as* $S_{\bar{\alpha}} = S_{\alpha_1} \times \ldots \times S_{\alpha_m}$.

Property 1. After $N_i = \frac{1}{\alpha_i \epsilon_i}$ samples with an MCMC algorithm T, the probability that the best solution found thus far $\mathbf{z}_{N_i}$ is in the top α_i for an agent a_i is at least $1 - \epsilon_i$:

$$P_{\mathrm{T}}\left(\mathbf{z}_{N_i} \in S_{\alpha_i} \mid N_i = \frac{1}{\alpha_i \cdot \epsilon_i}\right) \geq 1 - \epsilon_i.$$

Definition 5 and Property 1 are introduced by Nguyen *et al.* [19] and can be generalized to any MCMC sampling algorithm whose Markov chain generated is irreducible and aperiodic as convergence is guaranteed in a finite number of time steps.

Theorem 1. *Given* m *agents* $a_1, \ldots, a_m \in \mathcal{A}$*, and a number of samples* $N_i = \frac{1}{\alpha_i \cdot \epsilon_i}$ $(i = 1, \ldots, m)$*, the probability that the best complete solution found thus far* $\mathbf{z}_{\mathbf{N}}$ *is in the top* $\bar{\alpha}$*-percentile is greater than or equal to* $\prod_{i=1}^{m}(1 - \epsilon_i)$*, where* $\mathbf{N} = \bigwedge_{i=1}^{m} N_i$*. In other words,*

$$P_{\mathrm{T}}\left(\mathbf{z}_{\mathbf{N}} \in S_{\bar{\alpha}} \mid \mathbf{N}\right) \geq \prod_{i=1}^{m}(1 - \epsilon_i).$$

Proof. Let $\mathbf{z}_N$ denote the best solution found so far in the process resolution and $\mathbf{z}_{N_i}$ denote the best partial assignment over the variables held by agent a_i found after N_i samples. Let $\mathbf{S}_i$ be a random variable describing whether $\mathbf{z}_{N_i} \in S_{\alpha_i}$. Thus:

$$P_{\mathrm{T}}(\mathbf{z}_{\mathbf{N}} \in S_{\bar{\alpha}} \mid \mathbf{N}) \tag{1a}$$

$$= P_{\mathrm{T}}(\mathbf{z_N} \in S_{\bar{\alpha}} \mid \mathbf{N}_1, \ldots, \mathbf{N}_m) \tag{1b}$$
$$= P_{\mathrm{T}}(\mathbf{z_N} \in S_{\alpha_1} \times \ldots \times S_{\alpha_m} \mid \mathbf{N}_1, \ldots, \mathbf{N}_m) \tag{1c}$$
$$= P_{\mathrm{T}}(\mathbf{S}_1, \ldots, \mathbf{S}_m \mid \mathbf{B}_1, \ldots, \mathbf{B}_m, \mathbf{N}_1, \ldots, \mathbf{N}_m) \tag{1d}$$

where each $\mathbf{B}_i$ $(i{=}1, \ldots, m)$ is a random variable describing a particular value assignment associated to the interface variables B_i for the agent a_i. They are introduced to relate each of the $\mathbf{z}_{\mathbf{N_i}}$ to each other, which are sampled independently.

Since the values sampled in the local variable of a_i are dependent only of the values of the interface values B_i, it follows that $\mathbf{S}_i$ is conditionally dependent of $\mathbf{B}_i$ but conditionally independent of all other $\mathbf{B}_j$, with $j \neq i$:

$$S_i \perp\!\!\!\perp B_j \mid B_i$$

for all $j = 1 \ldots m$ and $j \neq i$. Noticing that, given random variables a, b, c, whenever $a \perp\!\!\!\perp b \mid c$ we can write: $P(a \mid b, c) = P(a \mid c)$, and that $P(a, b \mid c) = P(a \mid b, c)$, it follows that Equation (1d) can be rewritten as:

$$\begin{aligned}
&P_{\mathrm{T}}(\mathbf{S}_1 \mid \mathbf{B}_1, \mathbf{N}_1) \cdot \ldots \cdot P_{\mathrm{T}}(\mathbf{S}_m \mid \mathbf{B}_m, \mathbf{N}_m) \\
&\quad = P_{\mathrm{T}}(\mathbf{z_{N_1}} \in S_{\alpha_1} \mid \mathbf{B}, \mathbf{N}) \cdot \ldots \cdot P_{\mathrm{T}}(\mathbf{z_{N_m}} \in S_{\alpha_m} \mid \mathbf{B}, \mathbf{N}) && \text{(2a)} \\
&\quad \geq (1 - \epsilon_1) \cdot \ldots \cdot (1 - \epsilon_m) && \text{(2b)} \\
&\quad = \prod_{i=1}^{m} (1 - \epsilon_i). && \text{(2c)}
\end{aligned}$$

for any of the assignments of the variables in B_i, as the utility functions involving variables in the interface of any two agents are solved optimally. □

Theorem 2. *The number of messages required by DMCMC is* $O(|\mathcal{A}|)$.

Proof. DMCMC agents exchange $|\mathcal{A}|-1$ UTIL messages (one through each tree-edge) and $|\mathcal{A}| - 1$ VALUE messages. Thus, the total number of messages required by the algorithm is $O(|\mathcal{A}|)$. □

Note that, unlike DPOP, which requires $O(|\mathcal{X}|)$ messages, no message exchange is required to solve the constraints defined over the scope of the local variables each agent, which is achieved via local sampling.

Theorem 3. *The memory complexity of each DMCMC agent* $a_i \in \mathcal{A}$ *is* $O(|D_i|^{|S_i \setminus B_i|})$, *where* $S_i = \{x | x \in sep(a_i) \wedge \alpha(x) \succ_T a_i \wedge \exists f \in \mathcal{F}. x \in \mathbf{x}^f \wedge \mathbf{x}^f \cap B_i \neq \emptyset\}$, *is the set of the ancestors agent's variables in its separator which are involved in a constraint with some variable in* B_i.

Proof. Each agent $a_i \in \mathcal{A}$ needs to store its own utilities and the corresponding solution (value assignment for all non-interface local variables $x_i^j \in L_i \setminus B_i$) for each combination of values of the interface variables $x_i^k \in B_i$, thus requiring $O(|D_i|^{|B_i|})$ space. Moreover during the UTIL propagation phase, each agent a_i stores the UTIL messages of each of its children $a_c \in C_i$, which also sends messages of size $O(|D_i|^{|S_c \setminus B_c|})$. Joint and projection operations can be performed efficiently within $O(|D_i|^{|S_i \setminus B_i|})$ space. Thus the memory complexity of each agent is exponential in its *induced width*, $O(|D_i|^{|S_i \setminus B_i|})$. □

One can bound the maximum message size and serialize large messages by letting the backedge handlers ask explicitly for solutions and utilities for a subset of their values sequentially. Moreover, one could reduce the memory requirements at cost of sacrificing completeness, by propagating solutions for a bounded set of value combinations rather than all combination of values of the interface variables. Several approaches have been proposed to reduce the memory requirement of DPOP [3, 22, 23].

5 Related Work

To the best of our knowledge, there are only two sampling algorithms developed to solve DCOPs thus far, namely DUCT [20] and Distributed Gibbs [19]. DUCT is a distributed version of the UCT algorithm [9]. It maintains and uses upper confidence bounds on each value of a variable to determine which value to choose during the sampling process. It updates the bounds to make them more informed after each sampling trial.

Like DMCMC (with Gibbs as the MCMC algorithm), Distributed Gibbs is also a distributed version of Gibbs. However, Distributed Gibbs uses different communication protocols and computation procedures, which results in slow convergence due to high network load requirements [19]. More specifically, Distributed Gibbs performs the Gibbs sampling process on the entire space of all variables (i.e., in each sampling trial, it assigns a value to each variable sequentially until all variables are assigned a value), while DMCMC performs multiple sampling processes in parallel, one for each subset of local variables of an agent. As a result, DMCMC is able to better exploit the parallel processes with the use of GPUs.

6 Experimental Results

We implemented CPU and GPU versions of the DMCMC framework with Gibbs (Gibbs) and Metropolis-Hastings (MH) as the MCMC sampling algorithms. The CPU versions sample sequentially, while the GPU versions sample in parallel with GPUs. We compare them against DPOP [21] (an optimal algorithm), MGM and MGM2 [15] (suboptimal algorithms).We use publicly-available implementations of these algorithms, which are implemented in the FRODO framework [14]. We run our experiments on a Intel(R) Xeon(R) CPU, 2.4GHz, 32GB of RAM, Linux x86_64, equipped with a Tesla C2075, 14SM, 448-core, 1.15 clock rate, CUDA 2.0. Note that we do not parallelize at the level of CPU cores, thus the number of cores in the CPU is immaterial. We measure the algorithms' runtime using the wall clock (wct) and the simulated time (st) [29] metrics, and perform evaluations on random graphs and meeting scheduling problems. All reported results are averaged among 100 runs. The underlying constraint graphs are generated as follows: We create an n-node network, whose local constraint graphs density p_1^ℓ produces $\lfloor |L_i|(|L_i|-1)p_1^\ell \rfloor$ edges among the local variables of each agent a_i, and whose (global) density p_1^g produces $\lfloor b(b-1)p_1^g \rfloor$ edges among non-local interface variables, where b is the total number of interface variables of the problem. All constraints utilities are randomly chosen from the interval $[1, 1000]$.

We first evaluate the effect of the initial parameters R and S for our DMCMC algorithms in a setting in which DPOP could terminate its execution, and thus report

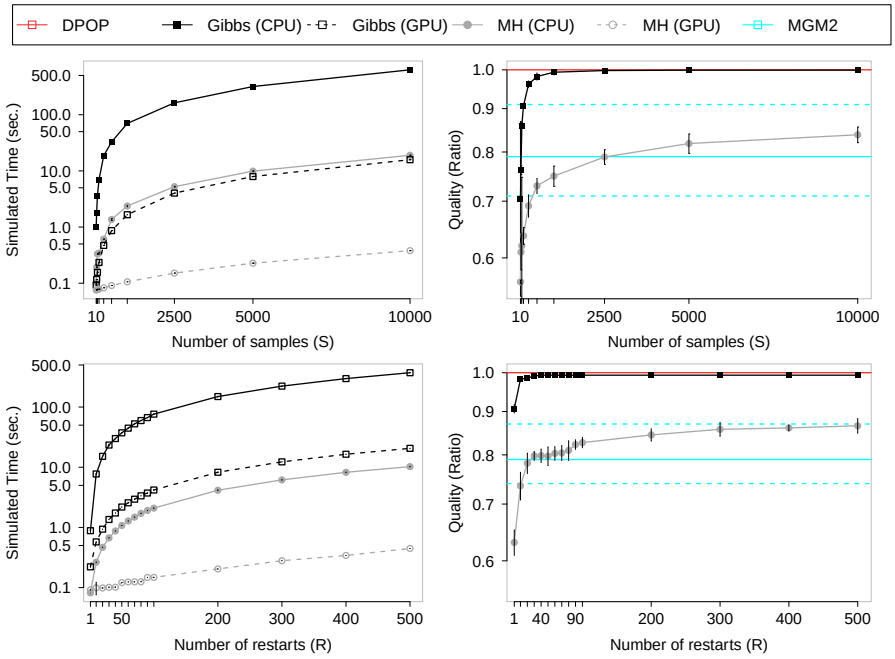


Fig. 5: Experimental Results: Random Graph Instances

its (optimal) solution. We fix the number of agents to 5, the number of local variables for each agent to 10, their domain sizes to 10, and the graph densities $p_1^g = p_1^\ell = 0.5$. Fig. 5(left) illustrates the runtime (in seconds) for the CPU and GPU implementations of our DMCMC algorithms for a range of the initial parameters $R \in [1, 100]$ and $S \in [10, 10000]$. These results shows that there is a clear benefit to parallelize the sampling operations with GPUs, exhibiting more than one order of magnitude speed up.

In the rest of the experiment, we show the GPU version only. Fig. 5(right) reports the ratio of the quality of the solutions returned by Gibbs and MH at varying of the parameters S and R, over that returned by DPOP. Additionally, we report the average (solid line) and variance (dotted lines) solution quality returned by MGM2. We observe that the prediction quality increases with increasing R and T. Gibbs is slower than MH, as it requires the computation of the normalization constants, which are computationally expensive even when parallelized. However, Gibbs finds better solutions than MH. Additionally, Gibbs finds better solutions than MGM2 for $S > 20$, and MH finds solutions whose quality is comparable to those returned by MGM2.

Next, we evaluate our algorithms at varying of several problem parameters on meeting scheduling problems. In these problems, meetings need to be scheduled between members of a hierarchical organization, (e.g., employees of a company; students, faculty members, and staff of a university), taking restrictions in their availability as well as their priorities into account. We used the Private Events as Variables (PEAV) problem formulation, which is commonly used in the literature, where the variables model the

$\|\mathcal{A}\|$	5			10			25			50		
	wct	*st*	*quality*	*wct*	*st*	*quality*	*wct*	*st*	*quality*	*wct*	*st*	*quality*
DPOP	125.39	94.98	**1661**	*oot*	*oot*	-	*oot*	*oot*	-	*oot*	*oot*	-
MGM	7.435	0.435	1379	11.910	0.446	2766	24.211	0.417	6692	45.771	0.462	13802
MGM2	8.939	0.979	1389	23.903	1.526	2783	56.035	1.629	7116	112.54	1.788	14145
Gibbs$_{CPU}$	6.146	1.101	1638	12.093	1.190	3.319	31.031	1.347	**8344**	62.411	1.489	**16577**
Gibbs$_{GPU}$	0.162	0.033	1635	0.301	0.034	**3338**	0.708	0.041	**8344**	1.416	0.048	16550
MH$_{CPU}$	0.561	0.113	1131	1.091	0.121	2775	2.281	0.176	6921	3.921	0.185	12112
MH$_{GPU}$	**0.047**	**0.014**	1143	**0.102**	**0.016**	2663	**0.196**	**0.017**	6925	**0.360**	**0.022**	11856
$\|\mathcal{X}_i\|$	5			10			25			50		
	wct	*st*	*quality*	*wct*	*st*	*quality*	*wct*	*st*	*quality*	*wct*	*st*	*quality*
DPOP	6.720	0.668	**1136**	*oot*	*oot*	-	*oot*	*oot*	-	*oot*	*oot*	-
MGM	5.260	0.242	947	11.910	0.446	2766	46.861	1.581	11652	180.05	5.749	35972
MGM2	8.701	0.602	941	23.903	1.526	2783	184.63	9.477	11889	*oot*	*oot*	-
Gibbs$_{CPU}$	2.336	0.489	1115	12.093	1.190	3319	182.68	2.446	13811	*oot*	*oot*	-
Gibbs$_{GPU}$	0.098	0.014	1104	0.301	0.34	**3338**	1.896	0.368	**13874**	12.707	1.384	**42124**
MH$_{CPU}$	0.351	0.048	986	1.091	0.121	2775	4.982	0.879	9850	56.077	6.506	33114
MH$_{GPU}$	**0.050**	**0.011**	972	**0.102**	**0.016**	2663	**0.146**	**0.022**	9716	**0.489**	**0.046**	32405
$\|D_i\|$	12			24			48			96		
	wct	*st*	*quality*	*wct*	*st*	*quality*	*wct*	*st*	*quality*	*wct*	*st*	*quality*
DPOP	22.230	9.996	**1332**	*oot*	*oot*	-	*oot*	*oot*	-	*oot*	*oot*	-
MGM	11.300	0.222	1077	11.910	0.446	2766	13.317	0.560	6133	18.177	1.106	13058
MGM2	19.723	0.541	1134	23.903	1.526	2783	53.972	5.314	6660	148.40	10.954	13866
Gibbs$_{CPU}$	3.348	0.530	1323	12.093	1.190	3319	51.669	5.716	7343	200.53	21.546	**16769**
Gibbs$_{GPU}$	0.214	0.029	1319	0.301	0.034	**3338**	0.763	0.090	**7357**	3.149	0.364	16278
MH$_{CPU}$	0.321	0.047	1086	1.091	0.321	2775	2.030	0.712	6135	6.081	1.306	12277
MH$_{GPU}$	**0.051**	**0.010**	1102	**0.102**	**0.016**	2663	**0.159**	**0.041**	6147	**0.218**	**0.146**	12189
p_1^ℓ	0.25			0.50			0.75			1.00		
	wct	*st*	*quality*	*wct*	*st*	*quality*	*wct*	*st*	*quality*	*wct*	*st*	*quality*
DPOP	10.850	0.885	**2305**	*oot*	*oot*	-	*oot*	*oot*	-	*oot*	*oot*	-
MGM	8.037	0.231	1835	11.910	0.446	2766	16.124	0.435	3342	19.832	0.605	3974
MGM2	12.908	0.708	1906	23.903	1.526	2783	47.258	2.554	3364	46.270	3.035	4091
Gibbs$_{CPU}$	7.991	0.981	2.269	12.093	1.190	3319	19.004	2.347	4032	25.691	2.821	**4751**
Gibbs$_{GPU}$	0.216	0.024	2300	0.301	0.034	**3338**	0.389	0.043	**4074**	0.451	**0.053**	4706
MH$_{CPU}$	0.775	0.101	1983	1.091	0.121	2775	1.225	0.135	3454	1.491	0.179	3921
MH$_{GPU}$	**0.090**	**0.013**	1931	**0.102**	**0.016**	2663	**0.170**	**0.021**	3458	**0.215**	**0.027**	3814

Table 1: Experimental Results: Meeting Scheduling Problems

meetings, their domains are the time slots when they can be held, and the constraints are between meetings that share participants [16]. In our experiments, we vary the number of agents $|\mathcal{A}| = \{5, 10, 25, 50\}$, the number of variables $|\mathcal{X}_i| = \{5, 10, 25, 50\}$ of each agent a_i, the domain size $|D_i| = \{12, 24, 48, 96\}$ of each variable x_i, the density of the local constraint graph $p_1^\ell = \{0.25, 0.5, 0.75, 1.0\}$ of each agent a_i. For each of the experiments below, we vary only one parameter and fix the rest in their "default" values: $|\mathcal{A}| = 10, |\mathcal{X}_i| = 10, |D_i| = 24, p_1^\ell = 0.5$. We set the number of samples for the D-MCMC algorithms to 100.

Table 1 reports the runtime (in seconds) and solution qualities for all algorithms, where *oot* indicates that the algorithm timed out after 5 minutes of wall-clock time. The best runtimes and solution qualities are shown in bold. We make the following observations:

- In all parameter settings, DMCMC with Gibbs finds better solutions than MGM and MGM2. Additionally, while the runtime for Gibbs$_{CPU}$ are comparable to those of MGM and MGM2, Gibbs$_{GPU}$ found those solutions by one order of magnitude faster than MGM and MGM2.
- The solutions quality reported by DMCMC with MH are comparable to those reported by MGM and MGM2, and MH$_{GPU}$ is at least one, and up to two order of magnitude times faster than MGM and MGM2.
- The GPU versions of our DMCMC algorithms are in general up to one order of magnitude faster than their CPU counterparts, and up to two orders when the local problem size increases. This result indicates that the GPUs can take advantage of the inherent parallelism present in the algorithm as a result of the partitioning of the problem into independent subproblems.
- Finally, for the problems for which DPOP successfully terminated within the time limit, we could measure the error in the quality of solutions found by DMCMC with Gibbs, which is only up to 5%.

Due to the unavailability of a public implementation, we did not compare our approaches against DUCT, however, Nguyen *et al.* [19] showed that DPOP outperforms DUCT especially when the problem sizes are small. In contrast, our approach consistently outperforms DPOP even on small problems. Additionally, they show that Distributed Gibbs [19] requires a large number of iterations to converge since it is estimating the joint distribution of the entire problem. In contrast, our MCMC framework with Gibbs requires a much smaller number of iterations, since it is only estimating the joint distribution of agent's local variables.

7 Conclusions

Our work is motivated by several factors: (*i*) the assumption in most DCOP algorithms that each agents owns exactly one variable; (*ii*) the recent introduction of sampling-based DCOP algorithms, which have been shown to outperform existing incomplete DCOP algorithms; and (*iii*) the advances in GPUs. These combination of factors provides a unique opportunity for us to harness the power of parallel computation of GPUs to solve general DCOPs with multiple variables per agent. In this paper, we introduce the Distributed MCMC framework, which decomposes a DCOP into independent subproblems that can each be sampled in parallel by GPUs. Our experimental results show that it can find good solutions up to one order of magnitude faster than MGM and MGM2. These results demonstrate the potential for using GPUs to scale up DCOP algorithms, which is exciting as GPUs provide access to thousands of computing cores at a very affordable cost. While the description of our solution focuses on DCOPs, our approach is also suitable to solve WCSPs. In the future, we plan to explore this direction, as well as extending the proposed framework to reduce its memory requirement similar to MB-DPOP [23].

References

1. 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.
2. R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1–2):41–85, 1999.
3. 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.
4. F. Fioretto, T. Le, W. Yeoh, E. Pontelli, and T. C. Son. Exploiting GPUs in Solving (Distributed) Constraint Optimization Problems with Dynamic Programming. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, 2015.
5. F. Fioretto, W. Yeoh, and E. Pontelli. Multi-Variable Agent Decomposition for DCOPs. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2016.
6. S. Geman and D. Geman. Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, 1984.
7. Y. Hamadi, C. Bessière, and J. Quinqueton. Distributed Intelligent Backtracking. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 219–223, 1998.
8. W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.
9. L. Kocsis and C. Szepesvári. Bandit Based Monte-Carlo Planning. In *Proceedings of the European Conference on Machine Learning (ECML)*, pages 282–293, 2006.
10. 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.
11. J. Larrosa. Node and arc consistency in weighted CSP. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 48–53, 2002.
12. T. Le, F. Fioretto, W. Yeoh, T. C. Son, and E. Pontelli. ER-DCOPs: A Framework for Distributed Constraint Optimization with Uncertainty in Constraint Utilities. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2016.
13. T. Le, T. C. Son, E. Pontelli, and W. Yeoh. Solving Distributed Constraint Optimization Problems with Logic Programming. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2015.
14. T. Léauté, B. Ottens, and R. Szymanek. FRODO 2.0: An Open-Source Framework for Distributed Constraint Optimization. In *Proceedings of the Distributed Constraint Reasoning Workshop*, pages 160–164, 2009.
15. R. Maheswaran, J. Pearce, and M. Tambe. Distributed Algorithms for DCOP: A Graphical Game-Based Approach. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 432–439, 2004.
16. 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.
17. N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21:1087, 1953.

18. 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.
19. D. T. Nguyen, W. Yeoh, and H. C. Lau. Distributed Gibbs: A Memory-Bounded Sampling-Based DCOP Algorithm. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 167–174, 2013.
20. B. Ottens, C. Dimitrakakis, and B. Faltings. DUCT: An Upper Confidence Bound Approach to Distributed Constraint Optimization Problems. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 528–534, 2012.
21. 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.
22. A. Petcu and B. Faltings. Approximations in Distributed Optimization. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 802–806, 2005.
23. A. Petcu and B. Faltings. MB-DPOP: A New Memory-Bounded Algorithm for Distributed Optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1452–1457, 2007.
24. S. D. Ramchurn, P. Vytelingum, A. Rogers, and N. Jennings. Agent-based control for decentralised demand side management in the smart grid. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 5–12, 2011.
25. G. O. Roberts and A. F. Smith. Simple conditions for the convergence of the Gibbs sampler and Metropolis-Hastings algorithms. *Stochastic Processes and Their Applications*, 49(2):207–216, 1994.
26. J. Sanders and E. Kandrot. *CUDA by Example. An Introduction to General-Purpose GPU Programming*. Addison Wesley, 2010.
27. L. G. Shapiro and R. M. Haralick. Structural descriptions and inexact matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (5):504–519, 1981.
28. R. Stranders, A. Farinelli, A. Rogers, and N. Jennings. Decentralised Coordination of Mobile Sensors Using the Max-Sum Algorithm. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 299–304, 2009.
29. E. Sultanik, R. Lass, and W. Regli. DCOPolis: a Framework for Simulating and Deploying Distributed Constraint Reasoning Algorithms. In *Proceedings of the Distributed Constraint Reasoning Workshop*, 2007.
30. M. Vinyals, J. Rodríguez-Aguilar, and J. Cerquides. Constructing a Unifying Theory of Dynamic Programming DCOP Algorithms via the Generalized Distributive Law. *Journal of Autonomous Agents and Multi-Agent Systems*, 22(3):439–464, 2011.
31. 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.
32. W. Yeoh and M. Yokoo. Distributed Problem Solving. *AI Magazine*, 33(3):53–65, 2012.
33. M. Yokoo, editor. *Distributed Constraint Satisfaction: Foundation of Cooperation in Multi-agent Systems*. Springer, 2001.
34. R. Zivan, S. Okamoto, and H. Peled. Explorative Anytime Local Search for Distributed Constraint Optimization. *Artificial Intelligence*, 212:1–26, 2014.