Undefined 1 (2014) 1–5
IOS Press

Distributed Multi-Agent Optimization for Smart Grids and Home Automation

Ferdinando Fioretto [a,*] Agostino Dovier [b] Enrico Pontelli [c]
[a] *Department of Industrial and Operation Engineering, University of Michigan, Ann Arbor, MI, USA*
E-mail: fioretto@umich.edu
[b] *Department of Mathematics, Computer Science, and Physics, University or Udine, Udine, Italy.*
E-mail: agostino.dovier@uniud.it
[c] *Department of Computer Science, New Mexico State University, NM, USA*
E-mail: epontell@cs.nmsu.edu

Abstract.
Distributed Constraint Optimization Problems (DCOPs) have emerged as one of the prominent multi-agent architectures to govern the agents' autonomous behavior in a cooperative multi-agent system (MAS) where several agents coordinate with each other to optimize a global cost function taking into account their local preferences. They represent a powerful approach to the description and resolution of many practical problems. However, typical real-world MAS applications are characterized by complex dynamics and interactions among a large number of entities, which translate into hard combinatorial problems, posing significant challenges from a computational and coordination standpoints.

This paper reviews two methods to promote a hierarchical parallel model for solving DCOPs, with the aim of improving the performance of the DCOP algorithm. The first is a *Multi-Variable Agent* (MVA) DCOP decomposition, which exploits co-locality of an agent's variables allowing the adoption of efficient centralized techniques to solve the subproblem of an agent. The second is the use of *Graphics Processing Units* (GPUs) to speed up a class of DCOP algorithms.

Finally, exploiting these hierarchical parallel model, the paper presents two critical applications of DCOPs for *demand response* (DR) program in smart grids. The *Multi-agent Economic Dispatch with Demand Response* (EDDR), which provides an integrated approach to the economic dispatch and the DR model for power systems, and the *Smart Home Device Scheduling* (SHDS) problem, that formalizes the device scheduling and coordination problem across multiple smart homes to reduce energy peaks.

Keywords: DCOP, GPUs, Smart Grid, Smart Homes

1. Introduction

The power network is the largest operating *machine* on earth, generating more than US$400 billion a year.[1] A significant concern in power networks is for the energy providers to be able to generate enough power to supply the demands at any point in time. Short terms demand peaks are however hard to predict and hard to address while meeting increasingly higher levels of sustainability. Thus, in the modern *smart electricity grid*, the energy providers can exploit the demand-side flexibility of the consumers to reduce the peaks in load demand.

This control mechanism is called *Demand response (DR)* and can be obtained by scheduling *shiftable loads* (i.e., a portion of power consumption that can be moved from a time slot to another) from peak to off-peak hours [16,21,37]. Due to concerns about privacy and users' autonomy, such an approach requires a decentralized, yet, coordinated and cooperative strategy,

*Corresponding author. E-mail: fioretto@umich.edu.
The research summarized in the paper was developed during Fioretto's PhD program at University of Udine (b) and New Mexico State University (c).
[1] Source: U.S. Energy Information Administration

with active participation of both the energy providers and consumers.

In a cooperative *multi-agent system* (MAS) multiple autonomous agents interact to pursue personal goals and to achieve shared objectives. The *Distributed Constraint Optimization Problem (DCOP)* model [11, 25,40] is an elegant formalism to describe *cooperative* multi-agent problems that are distributed in nature and where a collection of agents attempts to optimize a global objective within the confines of localized communication. DCOPs have been applied to solve a variety of coordination and resource allocation problems [20,42], sensor networks [9], and are shown to be suitable to model varius DR programs in the smart grid [15,16,24,31].

DCOP resolution algorithms are classified as either *complete* or *incomplete*. Complete algorithms guarantee the computation of an optimal solution to the problem, employing one of two broad approaches: distributed search-based techniques [25,26,39] or distributed inference-based techniques [27,36]. In search-based techniques, agents visit the search space by selecting value assignments and communicating them to other agents. Inference-based techniques rely instead on the notion of agent-*belief*, describing the best cost an agent can achieve for each value assignment of its variables. These beliefs drive the value selection process of the agents to find an optimal solution to the problem.

Despite their success, the adoption of DCOPs on large, complex, instances of problems faces several limitations, including restricting assumptions on the modeling of the problem and the inability of current solvers to capitalize on the presence of structural information.

In this paper, we review a *multi-variable-agent (MVA)* DCOP decomposition to allow agents to solve complex subproblems efficiently, and a technique to speed up the execution of inference-based algorithms via *graphic processing units (GPUs)* parallel architectures. These techniques enable us to design practical algorithms to efficiently solve large, complex, DCOPs. We review two DR applications of DCOPs, one where the goal is to optimize the power generator dispatch, while taking into account dispatchable loads, and another in the context of home automation, in which a home agent coordinates the schedule of smart devices in order to satisfy the user preferences while minimizing the global peaks of energy consumption.

The paper is organized as follows: Section 2 recalls the main definitions and describe two important complete and approximated algorithms for solving DCOPs. Section 3 briefly reviews the GPU programming model. Section 4 describes a decomposition technique that is used for solving DCOPs where each agents is responsible of solving a complex subproblem. Section 5 describes a DCOPs solving technique that is accelerated through the use of GPUs. Section 6 reviews two complex multi-agent applications of DR that are solved using DCOP techniques. The first application, presented in Section 7, presents an approach to optimize demand response at large scale, from a system operator perspective, by acting on large electric generators and loads. The second application, presented in Section 8, presents an approach to minimize the energy peaks through a demand response program that makes use of automated schedules of smart home devices. Finally, 9 concludes the work.

2. Background: DCOP

Let us start by defining the theoretical framework that is used for modeling smart grid and home automation problems.

2.1. Preliminaries

A Distributed Constraint Optimization Problem [12, 25,40] is a tuple $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{F}, \mathcal{A}, \alpha \rangle$, where:

- $\mathcal{X} = \{x_1, \dots, x_n\}$ is a set of *variables*;
- $\mathcal{D} = \{D_{x_1}, \dots, D_{x_n}\}$ is a set of finite *domains*, with $x_i \in D_{x_i}$ for all $x_i \in \mathcal{X}$;
- $\mathcal{F} = \{f_1, \dots, f_m\}$ is a set of *cost functions* (or *constraints*), where $f_i : \prod_{x_j \in \mathbf{x}^i} D_{x_j} \to \mathbb{R}_+ \cup \{\infty\}$ and $\mathbf{x}^i \subseteq \mathcal{X}$ is the set of the variables relevant to f_i, called its *scope*.
- $\mathcal{A} = \{a_1, \dots, a_p\}$ is a set of *agents*; and
- $\alpha : \mathcal{X} \to \mathcal{A}$ maps each variable to one agent.

With a slight abuse of notation, we consider the function $\alpha : \wp(\mathcal{A}) \to \wp(\mathcal{X})$,[2] where $\alpha(A)$ denotes the set of all variables controlled by the agents in A.

A *partial assignment* σ_X is an assignment of values to a set of variables $X \subseteq \mathcal{X}$ that is consistent with the domains of the variables; i.e., it is a partial function $\sigma_X : \mathcal{X} \to \bigcup_{i=1}^{n} D_{x_i}$ such that, for each $x_j \in \mathcal{X}$, if $\sigma_X(x_j)$ is defined (i.e., $x_j \in X$), then $\sigma_X(x_j) \in D_{x_j}$.

[2] $\wp(S)$ is the power set of a set S.

The *cost*

$$F(\sigma_X) = \sum_{f_i \in \mathcal{F}, \mathbf{x}^i \subseteq X} f_i(\sigma_{\mathbf{x}^i})$$

of an assignment σ_X is the sum of the evaluation of the constraints whose scope falls within X. A *solution* is a (partial) assignment σ_X (written σ for shorthand) for all the variables of the problem, i.e., with $X = \mathcal{X}$, whose cost is finite: $F(\sigma_X) \neq \infty$. The goal is to find a solution with *minimum cost*:

$$\mathbf{x}^* = \operatorname*{argmin}_{\mathbf{x}} F(\mathbf{x}).$$

Let us observe that the notion of *constraint* is expressed by the cost functions. Using terminology derived from local search [30], all constraints are considered to be *"soft"*. *"Hard"* constraints, i.e., constraints defining combinations of values for their variables that are either allowed or violated, can be represented through the value set $\{0, \infty\}$, where ∞ represents an infeasible value combination.

Given a DCOP P, $G_P = (\mathcal{X}, E_{\mathcal{F}})$, where $E_{\mathcal{F}} = \{\{x, y\} : (\exists f_i \in \mathcal{F})\{x, y\} \subseteq \mathbf{x}^i\}$, is the *constraint graph* of P. The representation as a constraint graph cannot deal explicitly with k-ary constraints (with $k > 2$). A typical artifact to deal with such constraints is to introduce a virtual variable that monitors the value assignments for all the variables in the scope of the constraint and generates the cost values [2]. One of the existing variables can take the role of a virtual variable.

Following [14], we introduce the concepts of *local variables* for an agent a_i: $L_i = \{x_j \in \mathcal{X} \,|\, \alpha(x_j) = a_i\}$, and the set of its *interface variables* $B_i = \{x_j \in L_i \,|\, \exists x_k \in \mathcal{X} \wedge \exists f_s \in \mathcal{F} : \alpha(x_k) \neq a_i \wedge \{x_j, x_k\} \subseteq \mathbf{x}^s\}$. This concept defines a clear separation between agent's local problems and the agents' problem dependencies. The former is the set of functions involving exclusively the variables controlled by the agent. The latter is defined through the functions whose scope involves variables controlled by some other agent. Such separation allows us to give a structure to the constraint graph as follows:

- 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 G_P, where $E_{\mathcal{F}_i} = \{\{x_a, x_b\} \mid (\exists f_j \in \mathcal{F})(\{x_a, x_b\} \subseteq \mathbf{x}^j \wedge \mathbf{x}^j \subseteq L_i)\}$.
- The *agent interaction graph* $A_P = (\mathcal{A}, E_{\mathcal{A}})$ is the graph where $E_{\mathcal{A}} = \{\{a_i, a_j\} \,|\, (\exists f_k \in \mathcal{F})(\{a_i, a_j\} \subseteq \alpha(\mathbf{x}^k))\}$.
- For each a_i, we denote with $N_{a_i} = \{a_j \in \mathcal{A} | \{a_i, a_j\} \in E_{\mathcal{A}}\}$ the set of its *neighbors*.

A widespread assumption in the communication model of a DCOP algorithm is that each agent can communicate exclusively with its neighboring agents in the agent interaction graph.

A *DFS pseudo-tree* arrangement of A_P is a subgraph $T_P = (\mathcal{A}, E_T)$ of A_P such that E_T is a spanning tree of A_P and for each $f_i \in \mathcal{F}$ and $x, y \in \mathbf{x}^i$, $\alpha(x)$ and $\alpha(y)$ appear on the same branch of T_P.

Edges of A_P that appear in T_P are referred to as *tree edges*,while the remaining edges of A_P are referred to as *back edges*. Figure 1 (right) shows an example of a pseudo tree for the constraint graph in Figure 1 (left).

Example 1 *Let us suppose that there are three agents. Agent 1 controls variables x_1, x_2, x_3; agent 2 controls variables x_4 and x_6, and agent 3 controls variable x_4. For example, x_2, x_5, x_6 could represent power generators (x_6 is the most powerful of the three), x_1, x_2 represent lamps, and x_3 a reading device that requires light. The domains for the variables are $D_{x_1} = \cdots = D_{x_5} = \{0, 1\}$ while $D_{x_6} = \{0, 1, 2, 3\}$. Constraints between the variables are encoded through cost functions represented by the following tables:*

A	B	$f_{A,B}$
0	0	∞
0	1	∞
1	0	10
1	1	0

x_4	x_5	f_{x_4,x_5}
0	0	0
0	1	5
1	0	10
1	1	0

x_2	x_5	x_6	f_{x_2,x_5,x_6}
–	–	0	0
1	1	–	10
1	0	–	0
0	–	i	$10 \cdot i$

The leftmost table models the cases with $(A, B) = (x_1, x_2)$ and $(A, B) = (x_1, x_3)$. Intuitively, the cost ∞ is used to ensure that lamp x_1 be switched on. If x_1 is switched, but it does not use energy from generator x_1, a penalty of 10 is enforced. Reading is important. Thus, x_3 must be set on 1, as well. However, reading without light cause a penalty. The combination lamp x_4 switched on and generator x_5 active is allowed without any penalty (its cost is 0). If generator x_5 is working it's a pity to keep x_4 switched off; costs are computed accordingly. These constraints generate the local constraint graphs. Let us focus now on agent's interactions (rightmost table). If the local generator is not working, energy from x_6 (in agent (3)) must be used, and this is costly ($-$means "any").

In the following, we describe two popular DCOP algorithms: a complete inference-based algorithm and an incomplete search-based algorithm. These algorithms

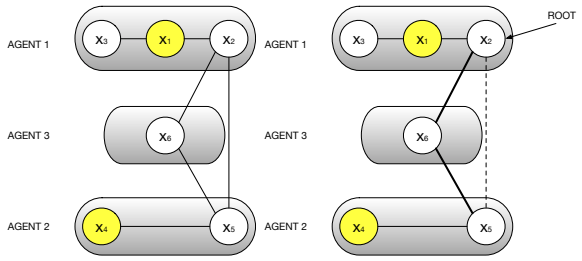

Fig. 1. Constraint Graph (left) and Pseudo Tree (right) of the DCOP of Example 1. x_2, x_5, x_6 are the interface variables. Boldface edges are tree edges. $\{x_2, x_5\}$ is a back edge.

were originally proposed in the context in which each agent controls a single variable. We thus restrict our attention to this special case and describe a generalization technique to handle multiple-variable agents in Section 4.

2.2. DPOP

he *Distributed Pseudo-tree Optimization Procedure (DPOP)* [27] is one of the most popular DCOP resolution algorithms. DPOP is a complete, inference-based algorithm, consisting of three phases.

In the first phase, the variables are ordered through a depth-first search visit of G_P into a DFS *pseudo-tree*. A pseudo-tree construction is achieved through a distributed algorithm (e.g., [1]). A set of variables, called the *separator set* $sep(x_i)$ is computed for each node x_i. $sep(x_i)$ contains all ancestors of x_i in the pseudo-tree that are connected with via tree edges or back edges to x_i or one of the descendants of x_i in the pseudo-tree.

In the second phase, called the *UTIL* propagation phase, each agent, starting from the leaves of the pseudo-tree executes two operations: (*1*) It *aggregates* the costs in its subtree for each value combination of variables in its separator and the variables it controls, and (*2*) It *eliminates* the variables it controls by optimizing over the other variables (i.e., for each combination of values for the variables in its separator, it selects the one with the smallest cost). The aggregated costs are encoded in a *UTIL* message, which is propagated from children to their parents, up to the root.

In the third phase, called the *VALUE* propagation phase, each agent, starting from the root of the pseudo-tree, selects the optimal value for its variables. The optimal values are calculated based on the *UTIL* messages received from the children and the *VALUE* message received from its parent. The *VALUE* messages contain the optimal values of the variables and are propagated from parents to their children, down to the leaves of the pseudo-tree.

The time and the space complexities of DPOP are dominated by the *UTIL* propagation phase, which is exponential in the maximum number of variables in a set $sep(x_i)$. The other two phases require a polynomial number of linear sized messages (in the number of variables of the problem), and the complexity of the local operations is at most linear in the size of the domain [27].

2.3. MGM

Maximum Gain Message (MGM) [23] is an incomplete, search-based algorithm that performs a distributed local search to solve a DCOP. Each agent a_i starts by assigning a random value to each of its variables. The agent then sends this information to all of its neighbors in G_P. Upon receiving the values of its neighbors, each agent calculates the maximum gain (i.e., the maximum decrease in cost) if it changes its values and sends this information to all of its neighbors as well. Upon receiving the gains of its neighbors, it changes its values if its gain is the largest among its neighbors. This process is repeated until a termination condition is met. MGM provides no quality guarantees on the returned solution.

Let us indicate with $l = \max_{a_i \in \mathcal{A}} |N_{a_i}|$, where N_{a_i} is the set of neighbors of a_i in the constraint graph. Let us denote with $d = \max_{x_i \in \mathcal{X}} |D_{x_i}|$. MGM agents perform $O(ld)$ number of operations in each iteration, as each agent needs to compute the cost for each of its values by taking into account the values of all its neighbors. The memory requirement per MGM agent is $O(l)$ since it needs to store the values of all its neighboring agents. In terms of communication requirement, each MGM agent sends $O(l)$ messages, one to each of its neighboring agents and each message is of constant size $O(1)$ as it contains either the agent's current value or the agent's current gain.

3. Background: GPU

Modern *Graphics Processing Units* (GPUs) are massive parallel architectures, offering thousands of computing cores and a rich memory hierarchy to support graphical processing. NVIDIA's *Compute Unified Device Architecture* (CUDA) [32] aims at enabling the use of the multiple cores of a graphics card to accel-

erate *general purpose* (non-graphical) applications by providing programming models and APIs that enable the full programmability of the GPU. The computational model supported by CUDA is *Single-Instruction Multiple-Threads (SIMT),* where multiple threads perform the same operation on multiple data points simultaneously.

A GPU is constituted by a series of *Streaming Multi-Processors (SMs),* whose number depends on the specific class of GPUs. For example, the Fermi architecture provides 16 SMs, as illustrated in Figure 2(left). Each SM contains a number of computing cores, each containing an ALU and a floating-point processing unit. Figure 2(right) shows a typical CUDA logical architecture. A 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 *GPU kernels*, where each kernel is a function to be executed by several *threads*. When mapping a kernel to a specific GPU, CUDA schedules groups of threads (*blocks*) on the SMs. In turn, each SM partitions the threads within a block in *warps*[3] for execution, which represents the smallest work unit on the device. Each thread instantiated by a kernel can be identified by a unique, sequential, identifier (T_{id}), which allows differentiating both the data read by each thread and the code to be executed.

3.1. Memory Organization

GPU and CPU are, in general, separate hardware units with physically distinct memory types connected by a system bus. Thus, for the device to execute some computation invoked by the host and to return the results to the caller, a data flow needs to be established from the host memory to the device memory and vice versa. The device memory architecture is entirely different from that of the host, in that it is organized in several levels differing to each other for both physical and logical characteristics.

Each thread can utilize a small number of *registers*,[4] which have thread lifetime and visibility. Threads in a block can communicate by reading and by writing a common area of memory, called *shared memory*. The total amount of shared memory per block is typically 48KB. Communication between blocks and communication between the blocks and the host is realized through a large *global memory*. The data stored in the global memory has global visibility and lifetime. Thus, it is visible to all threads within the application (including the host) and lasts for the duration of the host allocation.

Apart from lifetime and visibility, different memory types have also different dimensions, bandwidths, and access times. Registers have the fastest access memory, typically consuming within a clock cycle per instruction, while the global memory is the slowest but largest memory accessible by the device, with access times ranging from 300 to 600 clock cycles. The shared memory is partitioned into 32 logical banks, each serving exactly one request per cycle. Shared memory has a minimal access latency, provided that multiple thread memory accesses are mapped to different memory banks.

3.2. Bottlenecks and Common Optimization Practices

While it is relatively simple to develop correct GPU programs (e.g., by incrementally modifying an existing sequential program), it is nevertheless challenging to design an efficient solution. Several factors are critical to gaining performance. In this section, we discuss a few common practice that is important for the design of efficient CUDA programs.

Memory bandwidth is widely considered to be a critical bottleneck for the performance of GPU applications. Accessing global memory is relatively slow compared to accessing the shared memory in a CUDA kernel. However, even if not cached, global accesses that cover a segment of 128 contiguous Bytes data are fetched at once. Thus, most of the global memory access latency can be hidden if the GPU kernel employs a *coalesced* memory access pattern. Figure 3(left) illustrates an example of coalesced memory access pattern, in which aligned threads in a warp accesses aligned entries in a memory *segment*, which results in a single transaction. Thus, coalesced memory accesses allow the device to reduce the number of fetches to global memory for every thread in a warp. In contrast, when threads adopt a *scattered* data accesses (Figure 3(right)), the device serializes the memory transaction, drastically increasing its access latency.

Data transfers between the host and device memory are performed through a system bus, which translates to slow transactions. In general, it is convenient to store the data onto the device memory. Additionally, batch-

[3] A warp is typically composed of 32 threads.
[4] In modern devices, each SM allots 64KB for registers space.

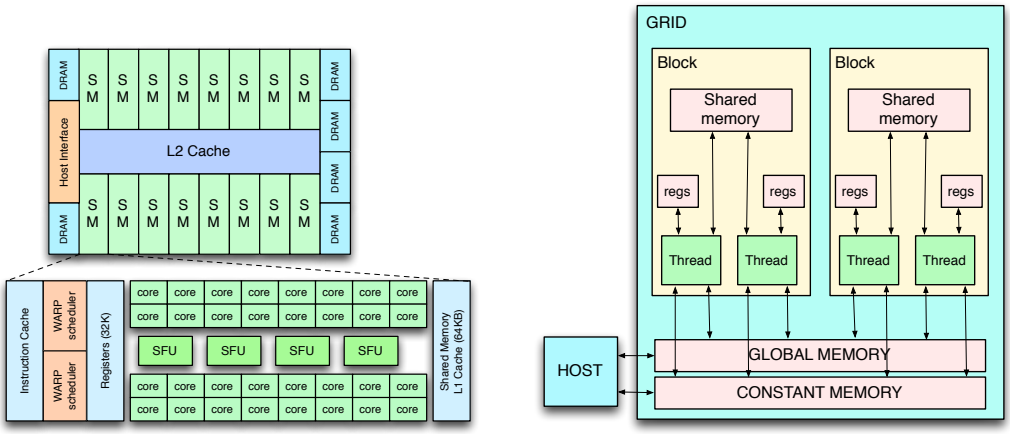

Fig. 2. Fermi Hardware Architecture (left) and CUDA Logical Architecture (right)

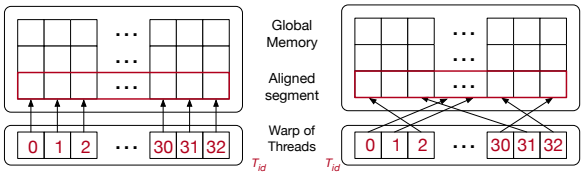

Fig. 3. Coalesced (left) and scattered (right) data access patterns

ing small memory transfers into a large one will reduce most of the per-transfer processing overhead [32].

The organization of the data in data structures and data access patterns plays a fundamental role in the efficiency of the GPU computations. Due to the computational model employed by the GPU, it is important that each thread in a warp executes the same branch of execution. When this condition is not satisfied (e.g., two threads execute different branches of a conditional construct), the degree of concurrency typically decreases, as the execution of threads performing separate control flows have to be serialized. This is referred to as *branch divergence*, a phenomenon that has been intensely analyzed within the *High Performance Computing (HPC)* community [6,8,18].

The line of work in this paper, focused on using GPUs to improve performance of DCOP solvers, extends recent approaches to make use of GPUs to enhance performance of constraint solves [4,5,7,10].

4. DCOP Applications Challenge: Agents with Multiple Variables

The vast majority of the DCOP resolution algorithms have been proposed in a restrictive setting, in which each agent controls exactly one variable. While this assumption simplifies the algorithm description and its development, it does not hold true in many of the applications of our interest. For instance, microgrids are self-sustainable and autonomous organizations composed of generators and loads within an electric smart grid; this can be modeled as agents whose variables represent the amount of energy to dispatch, transmit, and withdraw from the grid. In a smart home, multiple devices can be coordinated by a single autonomous home assistant, which, too, has to control a large number of variables.

There are two commonly used reformulation techniques to cope with DCOP in which agents control multiple variables [3,41]: *(i) Compilation*, where each agent creates a new *pseudo-variable*, whose domain is the Cartesian product of the domains of all variables of the agent; and *(ii) Decomposition*, where each agent creates a *pseudo-agent* for each of its variables. While both techniques are relatively simple, they can be inefficient. In compilation, the memory requirement for each agent grows exponentially with the number of variables that it controls. In decomposition, the DCOP algorithms will treat two pseudo-agents as independent entities, resulting in unnecessary computation and communication costs.

To overcome these limitations, the following subsections discusses the *Multi-Variable Agent (MVA)* DCOP decomposition, originally proposed in [14]. It enables a separation between the agents' *local subproblems*, which can be solved independently using centralized solvers, and the DCOP *global problem*, which requires coordination of the agents. The decomposition does not lead to any additional privacy loss

and enables the use of different centralized and distributed solvers in a hierarchical and parallel fashion.

4.1. The MVA Framework

In the MVA decomposition, a DCOP problem P is decomposed into $|\mathcal{A}|$ subproblems $P_i = (L_i, B_i, E_{\mathcal{F}_i})$, where P_i is associated to agent $a_i \in \mathcal{A}$. In addition to the decomposed problem P_i, each agent receives: (*1*) the *global* DCOP algorithm P_G, which is common to all agents in the problem and defines the agent's coordination protocol and the behavior associated with the receipt of a message, and (*2*) the *local* algorithm P_L, which can differ between agents and is used to solve the agent's subproblem.

Figure 4 shows a flowchart illustrating the four conceptual phases in the execution of the MVA framework for each agent a_i:

Phase 1—Wait: The agent waits for a message to arrive. If the received message results in a new value assignment $\sigma(x_r, k)$ for an interface variable x_r of B_i, then the agent will proceed to Phase 2. If not, it will proceed to Phase 4. The value $k \in \mathbb{N}$ establishes an enumeration of the interface variables' assignments.

Phase 2—Check: The agent checks if it has performed a new complete assignment for all its interface variables, indexed with $k \in \mathbb{N}$. If it has, then the agent will proceed to Phase 3. Otherwise it will return to Phase 1.

Phase 3—Local Optimization: When a complete assignment is given, the agent passes the control to a local solver, which solves the following problem:

$$\begin{aligned} &\text{minimize} \quad \sum_{f_j \in \mathcal{F}_i} f_j(\mathbf{x}^j) \\ &\text{subject to:} \quad x_r = \sigma(x_r, k) \quad \forall x_r \in B_i \end{aligned}$$

where $\mathcal{F}_i = \{f_i \in \mathcal{F} \mid \mathbf{x}^i \subseteq L_i\}$. Solving this problem results in finding the best assignment for the agent's local variables given the particular assignment for its interface variables. Notice that the local solver P_L is independent of the DCOP structure and it can be customized based on the agent's local requirements. For example, agents can use GPU-optimized solvers, if they have access to GPUs, or use off-the-shelf CP, MIP, or ASP solvers. Once the agent solves its subproblem, it proceeds to Phase 4.

Phase 4—Global Optimization: The agent processes the new assignment as established by the DCOP algorithm P_G, executes the necessary communications and returns to Phase 1. The agents can execute these phases independently of one another because they exploit the co-locality of their local variables without any additional privacy loss, which is a fundamental aspect in DCOPs [17].

In addition, the local optimization process can operate on $m \geqslant 1$ combinations of value assignments of the interface variables, before passing control to the next phase. This is the case when the agent explores m different assignments for its interface variables in Phases 2 and 3. These operations are performed by storing the best local solution and their corresponding costs in a cost table of size m, which can be visualized as a *cache memory*. The minimum value of m depends on the choice of the global DCOP algorithm P_G. For example, for common search-based algorithms such as asynchronous forward-bounding (AFB), it is 1, while for common inference-based algorithms such as DPOP, it is exponential in size of the separator set.

Correctness, completeness, asymptotical complexity, and an extensive evaluation of the MVA decomposition are presented in [14].

5. DCOP Applications Challenge: Scalability through Hierarchical Parallel Models

As mentioned in Section 2.2, the *aggregation* and *elimination* operations within the UTIL propagation phase of DPOP are the most expensive operations of the algorithm. In this section, we review the *GPU-based Distributed Bucket Elimination* (GpuBE) framework [13], which extends DPOP by exploiting GPU parallelism within the aggregation and elimination (i.e., UTIL) operations to speed up these operations. The key observation that allows us to parallelize these operations is that the computation of the cost for each value combination in a *UTIL* message is independent of the computation of the other combinations. The use of a GPU architecture allows us to exploit such independence, by concurrently exploring several value combinations of the *UTIL* message, computed by the aggregation operator, as well as concurrently eliminating variables.

5.1. GPU Data Structures

To fully utilize 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, minimizing

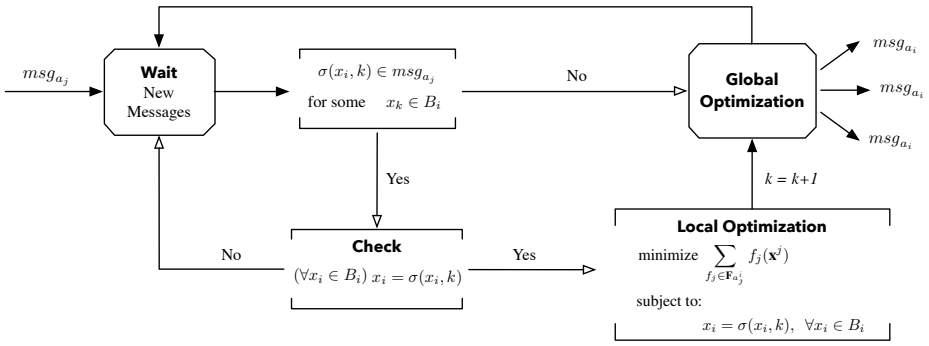

Fig. 4. MVA Execution Flow Chart.

the accesses to the slow device global memory while ensuring that the data access pattern enforced is coalesced. To do so, we store in the device global memory exclusively the minimal information required to compute the UTIL functions, which are communicated to the GPU once at the beginning of the computation. This allows the GPU kernels to communicate with the CPU host exclusively to exchange the results of the aggregation and elimination processes.

We introduce the following concept: An *UTIL-table* is a 4-tuple, $T = \langle \mathbf{S}, \mathbf{R}, \chi, \prec \rangle$, where:

- $\mathbf{S} \subseteq \mathcal{X}$, is a list of variables denoting the *scope* of T.
- $\mathbf{R}$ is a list of tuples of values, each tuple having length $|\mathbf{S}|$. Each element in this list (called *row* of T) specifies an assignment of values for the variables in $\mathbf{S}$ that is consistent with their domains. We denote with $\mathbf{R}[i]$ the tuple of values corresponding to the i-th row in $\mathbf{R}$, for $i = \{1, \ldots, |\mathbf{R}|\}$.
- χ is a list of length $|\mathbf{R}|$ of cost values corresponding to the costs of the assignments in $\mathbf{R}$. In particular, the element $\chi[i]$ represents the cost of the assignment $\mathbf{R}[i]$ for the variables in $\mathbf{S}$, with $i = \{1, \ldots, |\mathbf{R}|\}$.
- $\prec$ denotes an ordering relation used to sort the variables in the list $\mathbf{S}$. In turn, the value assignments, and cost values, in each row of $\mathbf{R}$ and χ, respectively, obey the same ordering.

As a technical note, a UTIL-table T is mapped onto the GPU device to store exclusively the cost values χ, not the associated variables values. We assume that the rows of $\mathbf{R}$ are sorted in lexicographic order—thus, the i-th entry $\chi[i]$ is associated with the i-th permutation $\mathbf{R}[i]$ of the variable values in $\mathbf{S}$, in lexicographic order. This strategy allows us to employ a simple, perfect hashing to efficiently associate row numbers with variables' values. Additionally, all the data stored in the GPU global memory is organized in mono-dimensional arrays, to facilitate *coalesced memory accesses*.

5.2. GPU-based Constraint Aggregation

The constraint aggregation takes as input two *UTIL*-tables: T_i and T_o, and aggregates the cost values in χ_i to those of χ_o for all the corresponding assignments of the shared variables in the scope of the two *UTIL*-tables. We refer to T_i and T_o as to the *input* and *output* *UTIL*-tables, respectively.

Consider the example in Figure 5, the cost values χ_i of the input *UTIL*-table T_i (left) are aggregated to the cost values χ_o of the output *UTIL*-table T_o (right)—which were initialized to 0. The rows of the two tables with identical value assignments for the shared variables x_2 and x_3 are shaded with the same color.

To optimize the performance of the GPU operations and to avoid unnecessary data transfers to/from the GPU global memory, we only transfer the list of cost values χ for each *UTIL*-table that need to be aggregated and employ a simple, perfect hashing function to efficiently associate row numbers with variables' values. This allows us to compute the indexes of the cost vector of the input *UTIL*-table relying exclusively on the information of the thread ID and, thus, avoid accessing the scope $\mathbf{S}$ and assignment vectors $\mathbf{R}$ of the input and output *UTIL*-tables. We refer the interested reader to [13] for the complete details.

To exploit the highest degree of parallelism offered by the GPU device, we:

1. map one GPU thread T_{id} to one element of the output *UTIL*-table, and
2. adopt the ordering relation $\prec_T$ for each input and output *UTIL*-table processed.

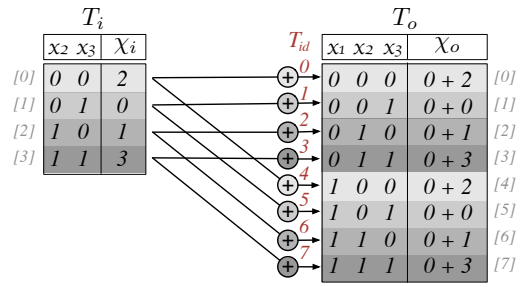

Fig. 5. Example of aggregation of two tables on GPU.

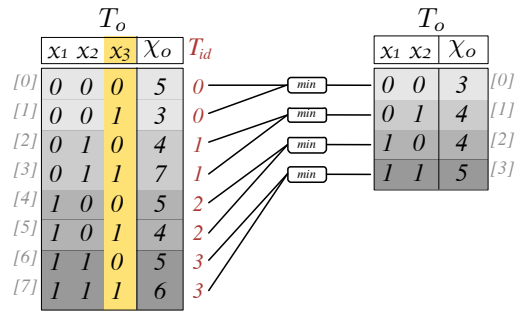

Fig. 6. Example of variable elimination on GPU.

Adopting such techniques allows each thread to be responsible for performing exactly two reads and one write from/to the GPU global memory. Additionally, the ordering relation enforced on the *UTIL*-tables allows us to exploit the locality of data and to encourage coalesced data accesses. As illustrated in Figure 5, this paradigm allows threads (whose IDs are identified in red by their T_{id}'s) to operate on contiguous chunks of data and, thus, minimizes the number of actual read (from the input *UTIL*-table, on the right) and write (onto the output *UTIL*-table, on the left) operations from/to the global memory performed by a group of threads with a single data transaction.[5]

As a technical detail, the *UTIL*-tables are created and processed so that the variables in their scope are sorted according to the order $\prec_T$. This means that the variables with the highest priority appear first in the scope list, while the variables to be eliminated always appear last. Such detail allows us to efficiently encode the elimination operation on the GPU, as explained next.

[5] Accesses to the GPU global memory are cached into cache lines of 128 Bytes, and can be fetched by all requiring threads in a warp.

5.3. GPU-based Variable Elimination

The variable elimination takes as input a *UTIL*-table T_o and a variable $x_i \in \mathbf{S}_o$. It removes x_o from the *UTIL*-table's scope by optimizing over its cost rows. As a result, the output *UTIL*-table rows list the unique assignments for the value combinations of $\mathbf{S}_o \backslash \{x_i\}$ in the input *UTIL*-table $\mathbf{R}_o$ which minimizes the costs values for each $d \in D_{x_i}$. Figure 6 illustrates this process, where the variable x_3 is eliminated from the *UTIL*-table T_o. The column being eliminated is highlighted yellow in the input *UTIL*-table. The different row colors identify the unique assignments for the remaining variables x_1, x_2, and exposes the high degree of parallelization that is associated to such operation. To exploit this level of parallelization, we adopt a paradigm similar to that employed in the aggregation operation on GPU, where each thread is responsible of the computation of a single output element.

The variable elimination is executed in parallel by a number of GPU threads equal to the number of rows of the output *UTIL*-table. Each thread identifies its row index r_o within the output *UTIL*-table cost values χ_o, given its thread ID. It hence retrieves an input row index r_j to the value of the first χ_o input *UTIL*-table row to analyze. Note that, as the variable to eliminate is listed last in the scope of the *UTIL*-table, it is possible to retrieve each unique assignment for the projected output bucket table, simply by offsetting r_o by the size of D_{x_i}. Additionally, all elements listed in $\chi_o[r_j], \ldots, \chi_o[r_j + |D_{x_i}|]$ differ exclusively on the value assignment to the variable x_i (see Figure 6). Thus, the GPU kernel evaluates the input *UTIL*-table cost values associated to each element in the domain of x_i, by incrementing the row index r_j, $|D_{x_i}| - 1$ times, and chooses the minimum cost value. At last, it saves the best cost found to the associated output row.

Note that each thread reads $|D_{x_i}|$ adjacent values of the vector χ_o, and writes one value in the same vector. Thus, this algorithm (*1*) perfectly fits the SIMT paradigm, (*2*) minimizes the accesses to the global memory as it encourages a coalesced data access pattern, and (*3*) uses a relatively small amount of global memory, as it recycles the memory area allocated for the input *UTIL*-table, to output the cost values for the output *UTIL*-table.

For additional details on these operations, and on the theoretical and experimental results of the GPU parallel version of DPDOP, we refer the reader to [13].

6. Applications to the Smart Grid

With the growing complexity of the current power grid, there is an increasing need for intelligent operations coordinating energy supply and demand. A key feature of the *smart grid* vision is that intelligent mechanisms will coordinate the production, transmission, and consumption of energy in a distributed way, guaranteeing sustainability, reliability, and resilience. The distributed nature of the grid makes cooperative multi-agent based solutions a natural fit. A critical problem in this domain is that of minimizing the peak load demands, as these are expensive from both a system operational perspective and for the risk associated with power failure due to overload. *Demand response (DR)* strategies allow customers to make autonomous decisions on their energy consumption and have been shown to improve various power system operations, including load balancing and frequency regulation.

From a large scale power generation perspective, the *economic dispatch (ED)* of power generators is applied to allocate the power to be produced by each generator minimizing the production costs while satisfying the physical constraints of the power system. Demand response programs enable customers to make informed decisions regarding their energy consumption and can be used to reduce the total peak demand, reshape the demand profile, and thus increase grid sustainability. Maintaining a constant balance between generation and consumption of power is critical for effective power system operations. Section 7 proposes a proactive DCOP approach to the economic dispatch which includes demand response and can quickly respond to the dynamic changes of the grid loads (within a few minutes).

At a smaller scale, from a consumer perspective, demand response can be used to reduce the residential electricity spending by designing smart buildings capable of making autonomous decisions to control power loads, production, and transmission. Through the proliferation of smart devices (e.g., smart thermostats, smart washers) in our homes and offices, the buildings' automation within the broader smart grid is becoming crucial. Home automation is defined as the automated control of the buildings electrical devices with the objective of the improved comfort of the occupants, improved energy efficiency, and reduced operational costs. Section 8 focuses on the scheduling of smart devices in a decentralized way. In the proposed model, each household is responsible for the schedule of the devices in her building under the assumption that each user has personal preferences and comforts levels. The DR model enables users to act cooperatively to reduce the global energy peaks.

7. Multi-agent Economic Dispatch with Demand Response

In traditional operations, ED and DR are implemented separately, despite the strong interdependencies between these two problems. Fioretto et al. [16] proposed an integrated approach to solve the ED and DR problems to simultaneously maximizes the benefits of customers and minimizes the generation costs. We survey such approach next.

7.1. The EDDR Model

A power grid can be viewed as a network of nodes (called *buses* in the power systems literature) that are connected by distribution cables and whose loads are served by (electromechanical) power generators. Typically, a group of such power generators is located in a single power station, and a number of power stations are distributed in different geographic areas. Such a power grid can be visualized as an undirected graph $(\mathcal{V}, \mathcal{E})$, in which buses (in $\mathcal{V}$) are modeled as graph nodes and transmission lines (in $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$) as edges. This representation captures the ability of the current to flow in both directions in a circuit.

A n-bus power system is considered where each bus injects and withdraws power through a generator $g \in \mathcal{G}$ and a load $l \in \mathcal{L}$, respectively, where $\mathcal{G}$ and $\mathcal{L}$ are, respectively, the set of generators and loads in the problem. Load buses can be *dispatchable* or *non-dispatchable*, based on whether it is possible to defer a portion of the load.

We model each bus as an agent, capable of making autonomous decisions on its power consumption and generation. We assume that there is *exactly* one generator and one load in each bus. The case with multiple loads and generators per bus can be easily transformed into this simpler model by precomputing the best operational conditions for each output combination of loads and generation power.

When a load difference is revealed to the power system, the ED problem is to regulate the output of the appropriate units so that the new generation output meets the new load and the generators are at economic dispatch (i.e., they are running efficiently according to some objective function). Near real-time power con-

sumption monitoring from smart meters allows short-term load prediction, which can supply the smart grid with predictions on power consumption levels.

The D-EDDR problem with a time horizon H is presented in [16]. Due to a rippling effect on the generators' power-cost curve, the generators' cost functions contain nonlinear terms, which makes the optimization problem non-convex. Additionally, due to the non-linearity of the transmission lines capacity constraints, Fioretto et al. consider a relaxation of the D-EDDR problem in which the transmission lines capacities constraints are modeled as soft constraints with a penalty on the degree of their violation. This approach is similar to the one used by researchers in the power systems community [34,38].

The D-EDDR problem has been modeled as a *Dynamic DCOP* [28], which is a sequence of DCOPs $\langle P_1, P_2, \ldots \rangle$, where each P_t is associated with a particular time step t. Each DCOP P_t in the Dynamic DCOP has the same agents $\mathcal{A}$ of the network buses, each agent $a \in \mathcal{A}$ controls two variables: $a.x_g^t$ and $a.x_l^t$, for a generator and a load. The domains of these variables correspond to the possible power that can be injected by generator g, or withdrawn by load l, respectively. In addition to the cost functions associated with the generators' costs and the loads' utilities, the problem includes hard constraints that represent the physical constraints of the energy flow conservation and the generators' ramp constraints and soft constraints that represent the transmission lines capacities.

7.2. R-Deeds

The *Relaxed Distributed Efficient ED with DR Solver* (R-DEEDS) [16] is a multi-agent solver for solving the (relaxed) D-EDDR problem which is based on the dynamic DCOP model described above. R-DEEDS bears similarities with DPOP (Section 2.2) and it is composed of four phases: (*1*) Pseudo-tree Generation, (*2*) *UTIL* Initialization, (*3*) *UTIL* Propagation, and (*4*) *VALUE* Propagation.

The first and last phases of R-DEEDS are identical to that of DPOP. In the *UTIL* Initialization phase, each agent initializes its *UTIL*-tables by computing, for all the possible combinations of loads and generators outputs within the bus controlled by the agent, and for each time in the horizon, their costs and their effect on the each of the transmission lines of the network. Finally, the *UTIL* Propagation phase is similar to that of DPOP: R-DEEDS agents propagate the *UTIL*-tables up to the root agent analogously to agents in DPOP. However, differently, from DPOP, when an agent aggregates the *UTIL*-tables of its children, it also updates the effect on the congestion of the transmission lines of the network. Finally, when the root agent checks for satisfiability of some "global" constraints:

- If none of the values in its *UTIL*-table satisfies all the constraints, it suggests that the problem is insufficiently relaxed. Thus, it increases a parameter ω by a factor of 2 based on the difference with respect to its value in the last iteration,[6] propagates this information to all agents and informs them to reinitialize their *UTIL* tables with the new updated ω, and solve the new relaxed problem in a new iteration.
- If there is a row satisfying all the constraints, then the problem may be overly relaxed. Thus, it decreases ω by a factor of 2 based on the difference with respect to its value in the last iteration, and this new relaxed problem is solved in a new iteration.

This process repeats itself until some criteria is met (e.g., a maximum number of iterations, convergence in the solution).

A number of the operations of the algorithm can be sped up through parallelization using GPUs. In particular, the initialization and consistency checks of the different rows of the *UTIL*-table can be done independently from each other. Additionally, the aggregation of the different rows of the *UTIL*-tables can also be computed in parallel, fitting well the SIMT processing paradigm. Thus, R-DEEDS adopts a scheme similar to that described in Section 5 to implement a GPU version of the *UTIL* Initialization and *UTIL* Propagation phases.

7.3. Empirical Evaluation

We evaluate the proposed algorithms on 5 standard IEEE Bus Systems,[7] all having non-convex solution spaces, and ranging from 5 to 118 buses.

All the generators are set with a 5MW ramp rate limit. We use a 1MW discretization unit for each load and generators. Thus, the maximum domain sizes of the variables are between 100 and 320 in every experiment. The horizon H is set to 12, and we define a parameter H_{opt} that denotes the number of time steps that are considered by R-DEEDS agents at a time. Thus, it first solves the subproblem with the first H_{opt} time steps. Next, it solves the following subproblem with

[6] The larger the value of ω, the more relaxed the problem. The relaxed problem is identical to the original one if $\omega = 0$.

[7] http://publish.illinois.edu/smartergrid/

IEEE Buses		Simulated Runtime (sec)								Normalized Quality			
		CPU Implementation				GPU Implementation							
	H_{opt}	1	2	3	4	1	2	3	4	1	2	3	4
	5	0.010	0.044	3.44	127.5	0.025 (0.4x)	0.038 (1.2x)	0.128 (26.9x)	2.12 (60.2x)	0.8732	0.8760	0.9569	1.00
	14	0.103	509.7	–	–	0.077 (1.3x)	3.920 (130x)	61.70 (n/a)	–	0.6766	0.8334	1.00	–
	30	0.575	9084	–	–	0.241 (2.4x)	79.51 (114x)	–	–	0.8156	1.00	–	–
	57	4.301	–	–	–	0.676 (6.4x)	585.4 (n/a)	–	–	0.8135	1.00	–	–
	118	174.4	–	–	–	4.971 (35.1x)	–	–	–	1.00	–	–	–

Table 1

R-DEEDS CPU and GPU Runtimes, Speedups (in parenthesis), and Normalized Solution Qualities.

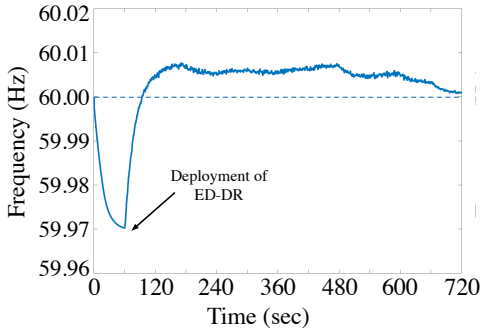

Fig. 7. Dynamic Simulation of the IEEE 30 Bus with heavy loads.

the next H_{opt} time steps, and so on, until the entire problem is solved. The satisfiability of the constraints between the subsequent H_{opt} subproblems is ensured in Phase 2 of R-DEEDS.

We generate 30 instances for each test case. The performance of the algorithms are evaluated using the *simulated runtime* metric [33], and we imposed a time-out of 5 hours. Results are averaged over all instances. These experiments are performed on a *AMD Opteron 6276*, 2.3GHz, 128GB of RAM, which is equipped with a *GeForce GTX TITAN* GPU device with 6GB of global memory. If an instance cannot be completed, it is due to runtime limits for the CPU implementation of the algorithm, and memory limits for the GPU implementation. We set the number of iterations to 10.

We do not report evaluation against existing (Dynamic) DCOP algorithms as the standard (Dynamic) DCOP models cannot fully capture the complexities of the D-EDDR problem (e.g., environment variables with continuous values).

Table 1 tabulates the runtimes in seconds of R-DEEDS with both CPU and GPU implementations at varying H_{opt}. It also tabulates the average solution quality found normalized with the best solution found for each IEEE bus. We make the following observations:

- The solution quality increases as H_{opt} increases.
- For the smaller test case with $H_{opt} = 1$, the CPU implementation is faster than the GPU one. However, for all other configurations, the GPU implementation is much faster than its CPU counterpart, with speedups up to 130 times. The reported speedup increase with increasing complexity of the problem (i.e., bus size and H_{opt}).
- The GPU implementation scales better than the CPU implementation. The latter could not solve any of the instances for the configurations with $H_{opt} = 3$ for the 14-bus system and $H_{opt} = 2$ for the 57-bus system.
- We observe that the algorithms report satisfiable instances within the first four iterations of the iterative process.

To validate the solutions returned by R-DEEDS, we tested the stability of the returned control strategy for the IEEE 30-Bus System on a dynamic simulator (Matlab *SimPowerSystems*) that employs a detailed time-domain model and recreates conditions analogous to those of physical systems. Figure 7 shows the dynamic response of the system frequency, whose nominal value is 60 Hz. The system frequency is at the nominal value when the power supply-demand is balanced. If more power is produced than consumed, the frequency would rise and vice versa. Deviations from the nominal frequency value would damage synchronous machines and other appliances. Thus, to ensure stable operating conditions, it is important to ensure that the system frequency deviation is confined to be within a small interval (typically 0.1 Hz).

In our experiment, the first 60 seconds of the simulation are tested enforcing the ED solution in a full load scenario (100% of full load) using the same setting as in [22]. The rest of the simulation deploys the D-EDDR solution returned by R-DEEDS. While the reso-

lution of the simulation is in microseconds, R-DEEDS agents send only desired power injected and withdrawn (schedules), computed offline, in one-minute intervals; the simulator interpolates the power generated between such intervals. This scenario reflects real-life conditions, where control signals are sent to actual generators at intervals of several minutes. As expected, the frequency deviation is more stable when the load predictions are accurate. Crucially, the deviation in both cases is within 0.05 Hz, thereby ensuring system stability. In addition, we observe, during simulation, that the D-EDDR solution can reduce the total load up to 68.5%, showing the DR contribution in peak demand reduction. Finally, the frequency response of the D-EDDR solution converges faster than that of the ED only, with smaller fluctuations. The reason is that the supply-demand balance is better maintained by coordinating the generators and loads in the system simultaneously.

8. Smart Home Device Scheduling

Residential and commercial buildings are progressively being automated through the introduction of smart devices (e.g., smart thermostats, circulator heating, washing machines). Besides, a variety of smart plugs, which allow users to control devices by remotely switching them on and off intelligently, are now commercially available. Device scheduling can, therefore, be executed by users, without the control of a centralized authority. This schema can be used for demand-side management in a DR program. However, uncoordinated scheduling may be detrimental to DR performance without reducing peak load demands [35]. For an effective DR, a coordinated device scheduling within a neighborhood of buildings is necessary. Privacy concerns arise when users share resources or cooperate to find suitable schedules. This section surveys the *Smart Home Device Scheduling (SHDS)* problem, which formalizes the problem of coordinating smart devices schedules across multiple smart homes as a multi-agent system, presented initially in [15].

8.1. The Problem

A *Smart Home Device Scheduling (SHDS)* problem is defined by the tuple $\langle \mathbf{H}, \mathcal{Z}, \mathcal{L}, \mathbf{P}_H, \mathbf{P}_Z, H, \theta \rangle$, where:

- $\mathbf{H} = \{h_1, h_2, \ldots\}$ is a neighborhood of smart homes, capable of communicating with one another.

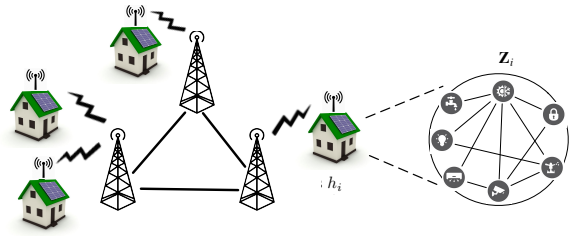

Fig. 8. Illustration of a Neighborhood of Smart Homes

- $\mathcal{Z} = \cup_{h_i \in \mathbf{H}} \mathbf{Z}_i$ is a set of smart devices, where $\mathbf{Z}_i$ is the set of devices in the smart home h_i (e.g., vacuum cleaning robot, smart thermostat).
- $\mathcal{L} = \cup_{h_i \in \mathbf{H}} \mathbf{L}_i$ is a set of locations, where $\mathbf{L}_i$ is the set of locations in the smart home h_i (e.g., living room, kitchen).
- $\mathbf{P}_H$ is the set of the state properties of the smart homes (e.g., cleanliness, temperature).
- $\mathbf{P}_Z$ is the set of the devices state properties (e.g., battery charge for a vacuum cleaning robot).
- H is the planning horizon of the problem, and $\mathbf{T} = \{1, \ldots, H\}$ denotes the set of time points.
- $\theta : \mathbf{T} \rightarrow \mathbb{R}^+$ represents the real-time pricing schema adopted by the energy utility company, which expresses the cost per kWh of energy consumed by consumers.

Finally, Ω_p is used to denote the set of all possible states for state property $p \in \mathbf{P}_H \cup \mathbf{P}_Z$ (e.g., all the different levels of cleanliness for the cleanliness property). Figure 8 shows an illustration of a neighborhood of smart homes with each home controlling a set of smart devices.

8.1.1. Smart Devices

For each home $h_i \in \mathbf{H}$, the set of smart devices $\mathbf{Z}_i$ is partitioned into a set of actuators $\mathbf{A}_i$ and a set of sensors $\mathbf{S}_i$. Actuators can affect the states of the home (e.g., heaters and ovens can affect the temperature in the home) and possibly their states (e.g., vacuum cleaning robots drain their battery power when running). On the other hand, sensors monitor the states of the home.

A tuple $\langle \ell_z, A_z, \gamma_z^H, \gamma_z^Z \rangle$ defines each device $z \in \mathbf{Z}_i$ of a home h_i , where $\ell_z \in \mathbf{L}_i$ denotes the relevant location in the home that it can act or sense, A_z is the set of actions that it can perform, and $\gamma_z^H : A_z \rightarrow \wp(\mathbf{P}_H)$ and $\gamma_z^Z : A_z \rightarrow \wp(\mathbf{P}_Z)$ map the actions of the device to the relevant state properties of the home and to those of the device, respectively.

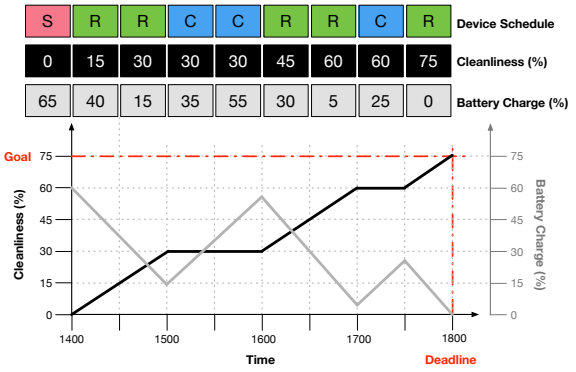

Fig. 9. Smart Home Device Scheduling Example

Example 2 *Consider a vacuum cleaning robot z_v with location $\ell_{z_v} =$ living_room. The set of possible actions is $A_{z_v} = \{$run, charge, stop$\}$ and the mappings are:*

$\gamma^H_{z_v}$: run$\rightarrow\{cleanliness\}$; charge$\rightarrow\varnothing$; stop$\rightarrow\varnothing$,

$\gamma^Z_{z_v}$: run$\rightarrow\{charge\}$; charge$\rightarrow\{charge\}$; stop$\rightarrow\varnothing$,

where $\varnothing$ represents a null *state property.*

8.1.2. Device Schedules

To control the energy profile of a smart home, we need to describe the behavior of the smart devices acting in the smart home during the time horizon. We formalize this concept with the notion of *device schedules*. We use $\xi^t_z \in A_z$ to denote the action of device z at time step t, and $\xi^t_X = \{\xi^t_z \mid z \in X\}$ to denote the set of actions of the devices in $X \subseteq \mathcal{Z}$ at time step t.

A *schedule* $\xi^{[t_a \rightarrow t_b]}_X = \langle \xi^{t_a}_X, \ldots, \xi^{t_b}_X \rangle$ is a sequence of actions for the devices in $X \subseteq \mathcal{Z}$ within the time interval from t_a to t_b.

Consider the illustration of Figure 9. The top row shows a possible schedule $\langle R, R, C, C, R, R, C, R \rangle$ for a vacuum cleaning robot starting at time 1400 hrs, where each time step is 30 minutes. The robot's actions at each time step are shown in the colored boxes with letters in them: red with 'S' for stop, green with 'R' for run, and blue with 'C' for charge.

At a high level, the goal of the SHDS problem is to find a schedule for each of the devices in every smart home that achieves some user-defined objectives (e.g., the home is at a particular temperature within a time window, the home is at a particular cleanliness level by some deadline) that may be personalized for each home. We refer to these objectives as *scheduling rules*.

8.1.3. Scheduling Rules

The SHDS introduces two types of scheduling rules:

- *Active Scheduling Rules (ASRs)* that define user-defined objectives on a desired state of the home (e.g., the living room is cleaned by 1800 hrs).
- *Passive Scheduling Rules (PSRs)* that define implicit constraints on devices that must hold at all times (e.g., the battery charge on a vacuum cleaning robot is always between 0% and 100%).

Example 3 *The following* ASR *defines a goal state where the living room floor is at least 75% clean (i.e., at least 75% of the floor is cleaned by a vacuum cleaning robot) by 1800 hrs:*

$$\texttt{living\_room cleanliness} \geqslant 75 \texttt{ before } 1800,$$

and the following PSRs *state that the battery charge of the vacuum cleaning robot z_v needs to be between 0% and 100% of its full charge at all the times:*

$$z_v \texttt{ charge} \geqslant 0 \texttt{always} \ \wedge \ z_v \texttt{ charge} \leqslant 100 \texttt{always}$$

We denote with $R^{[t_a \rightarrow t_b]}_p$ a scheduling rule over a state property $p \in \mathbf{P}_H \cup \mathbf{P}_Z$ and time interval $[t_a, t_b]$.

Each scheduling rule indicates a goal state at a home location or on a device $\ell_{R_p} \in \mathbf{L}_i \cup \mathbf{Z}_i$ of a particular state property p that must hold over the time interval $[t_a, t_b] \subseteq \mathbf{T}$. The scheduling rule's goal state is either the desired state of a home if it is an ASR (e.g., the cleanliness level of the room floor) or a required state of a device or a home if it is a PSR (e.g., the battery charge of the vacuum cleaning robot).

Each rule is associated with a set of actuators $\Phi_p \subseteq \mathbf{A}_i$ that can be used to reach the goal state. For instance, in our Example (3), Φ_p correspond to the vacuum cleaning robot z_v, which can operate on the living room floor. Additionally, a rule is associated with a sensor $s_p \in \mathbf{S}_i$ capable of sensing the state property p. Finally, in a PSR, the device can also sense its own internal states. The ASR of Example 3 is illustrated in Figure 9 by dotted red lines on the graph. The PSRs are not shown as they must hold for all time steps.

8.1.4. Feasibility of Schedules

To ensure that a goal state can be achieved across the desired time window, the system uses a *predictive model* of the various state properties. This predictive model captures the evolution of a state property over time and how this state property is affected by a given joint action of the relevant actuators.

A *predictive model* Γ_p for a state property p is a function $\Gamma_p : \Omega_p \times \times_{z\in\Phi_p} A_z \cup \{\perp\} \to \Omega_p \cup \{\perp\}$, where $\perp$ denotes an infeasible state and $\perp + (\cdot) = \perp$. In other words, the model describes the transition of state property p from state $\omega_p \in \Omega_p$ at time step t to time step $t+1$ when it is affected by a set of actuators Φ_p running joint actions $\xi^t_{\Phi_p}$: $\Gamma_p^{t+1}(\omega_p, \xi^t_{\Phi_p}) = \omega_p + \Delta_p(\omega_p, \xi^t_{\Phi_p})$ where $\Delta_p(\omega_p, \xi^t_{\Phi_p})$ is a function describing the effect of the actuators' joint action $\xi^t_{\Phi_p}$ on state property p.

We assume here, without loss of generality, that the state of properties is numeric—when this is not the case, a mapping of the possible states to a numeric representation can be easily defined.

Example 4 *Consider the* charge *state property of the vacuum cleaning robot z_v. Assume it has 65% charge at time step t and its action is $\xi^t_{z_v}$ at that time step. Thus:*

$$\Gamma^{t+1}_{charge}(65, \xi^t_{z_v}) = 65 + \Delta_{charge}(65, \xi^t_{z_v})$$

$$\Delta_{charge}(\omega, \xi^t_{z_v}) = \begin{cases} \min(20, 100-\omega) & \text{if } \xi^t_{z_v} = \textit{charge} \wedge \omega < 100 \\ -25 & \text{if } \xi^t_{z_v} = \textit{run} \wedge \omega > 25 \\ 0 & \text{if } \xi^t_{z_v} = \textit{stop} \\ \perp & \text{otherwise} \end{cases}$$

In other words, at each time step, the charge of the battery will increase by 20% if it is charging until it is fully charged, decrease by 25% if it is running until it has less than 25% charge, and no change if it is stopped.

The predictive model of the example above models a device state property. Notice that a recursive invocation of a predictive model allows us to predict the trajectory of a state property p for future time steps, given a schedule of actions of the relevant actuators Φ_p.

Given a state property p, its current state ω_p at time step t_a, and a schedule $\xi^{[t_a \to t_b]}_{\Phi_p}$ of relevant actuators Φ_p, the *predicted state trajectory* $\pi_p(\omega_p, \xi^{[t_a \to t_b]}_{\Phi_p})$ of that state property is defined as:

$$\Gamma_p^{t_b}(\Gamma_p^{t_{b-1}}(\ldots(\Gamma_p^{t_a}(\omega_p, \xi^{t_a}_{\Phi_p}), \ldots), \xi^{t_{b-1}}_{\Phi_p}), \xi^{t_b}_{\Phi_p}).$$

Consider the device scheduling example in Figure 9. The predicted state trajectories of the *charge* and *cleanliness* state properties are shown in the second and third rows. These trajectories are predicted given that the vacuum cleaning robot will take on the schedule shown in the first row of the figure. The predicted trajectories of these state properties are also illustrated in the graph, where the dark gray line shows the states for the robot's battery charge and the black line shows the states for the cleanliness of the room.

Notice that, in order to verify if a schedule satisfies a scheduling rule, it is sufficient to check that the predicted state trajectories are within the set of feasible state trajectories of that rule. Additionally, notice that each active and passive scheduling rule defines a set of feasible state trajectories. For example, the active scheduling rule of Example 3 allows all possible state trajectories as long as the state at time step 1800 is no smaller than 75. We use $R_p[t] \subseteq \Omega_p$ to denote the set of states that are feasible according to rule R_p of state property p at time step t.

More formally, a schedule $\xi^{[t_a \to t_b]}_{\Phi_p}$ satisfies a scheduling rule $R_p^{[t_a \to t_b]}$ (written as $\xi^{[t_a \to t_b]}_{\Phi_p} \models R_p^{[t_a \to t_b]}$) iff: $\forall t \in [t_a, t_b] : \pi_p(\omega_p^{t_a}, \xi^{[t_a \to t]}_{\Phi_p}) \in R_p[t]$, where $\omega_p^{t_a}$ is the state of state property p at time step t_a.

A schedule is *feasible* if it satisfies *all* the passive and active scheduling rules of each home in the SHDS problem.

8.1.5. Cost of Schedules

In addition to finding feasible schedules, the goal in the SHDS problem is to optimize for the aggregated total cost of energy consumed.

Each action $\xi \in A_z$ of device $z \in \mathbf{Z}_i$ in home $h_i \in \mathbf{H}$ has an associated energy consumption $\rho_z : A_z \to \mathbb{R}^+$, expressed in kWh. The aggregated energy $E_i^t(\xi^{[1\to H]}_{\mathbf{Z}_i})$ across all devices consumed by h_i at time step t under trajectory $\xi^{[1\to H]}_{\mathbf{Z}_i}$ is:

$$E_i^t(\xi^{[1\to H]}_{\mathbf{Z}_i}) = \ell_i^t + \sum_{z\in\mathbf{Z}_i} \rho_z(\xi_z^t)$$

where ℓ_i^t is the home background load produced at time t, which includes all non-schedulable devices (e.g., TV, refrigerator), and sensor devices—which are always active, and ξ_z^t is the action of device z at time t in the schedule $\xi^{[1\to H]}_{\mathbf{Z}_i}$. The cost $c_i(\xi^{[1\to H]}_{\mathbf{Z}_i})$ associated to schedule $\xi^{[1\to H]}_{\mathbf{Z}_i}$ in home h_i is:

$$c_i(\xi^{[1\to H]}_{\mathbf{Z}_i}) = \sum_{t\in\mathbf{T}} \left(E_i^t(\xi^{[1\to H]}_{\mathbf{Z}_i})\right) \cdot \theta(t) \qquad (1)$$

where $\theta(t)$ is the energy price per kWh at time t.

8.1.6. Optimization Objective

The objective of an SHDS problem is that of minimizing the following weighted bi-objective function:

$$\min_{\xi_{\mathbf{Z}_i}^{[1\rightarrow H]}} \quad \alpha_c \cdot C^{\text{sum}} + \alpha_e \cdot E^{\text{peak}} \tag{2}$$

$$\text{st: } \xi_{\Phi_p}^{[t_a \rightarrow t_b]} \models R_p^{[t_a \rightarrow t_b]} \quad (\forall h_i \in \mathbf{H}, R_p^{[t_a \rightarrow t_b]} \in \mathbf{R}_i) \tag{3}$$

where $C^{\text{sum}} = \sum_{h_i \in \mathbf{H}} c_i(\xi_{\mathbf{Z}_i}^{[1\rightarrow H]})$ is the aggregated monetary cost across all homes h_i; and $E^{\text{peak}} = \sum_{t \in \mathbf{T}} \sum_{\mathbf{H}_j \in \mathcal{H}} \sum_{h_i \in \mathbf{H}_j} \left(E_i^t(\xi_{\mathbf{Z}_i}^{[1\rightarrow H]})\right)^2$ is a quadratic penalty function on the aggregated energy consumption across all homes h_i, and $\alpha_c, \alpha_e \in \mathbb{R}$ are weights. Additionally, $\mathcal{H}$ is a partition set of $\mathbf{H}$ defining a set of homes coalitions, into each of which homes may share their energy profile with each other so to optimize the energy peaks. These coalitions can be exploited by a distributed algorithm to (1) parallelize computations between multiple groups and (2) avoid data exposure over long distances or sensitive areas.

Finally, constraint (3) defines the valid trajectories for each scheduling rule $r \in \mathbf{R}_i$, where $\mathbf{R}_i$ is the set of all scheduling rules of home h_i.

8.2. DCOP Representation

A DCOP mapping of the SHDS problem is as follows:

- AGENTS: Each agent $a_i \in \mathcal{A}$ in the DCOP is mapped to a home $h_i \in \mathbf{H}$.
- VARIABLES and DOMAINS: Each agent a_i controls:
  - For each actuator $z \in \mathbf{A}_i$ and time $t \in \mathbf{T}$, a variable $x_{i,z}^t$ whose domain is the set of actions in A_z. The sensors in $\mathbf{S}_i$ are considered always active, and thus not directly controlled by the agent.
  - An auxiliary interface variable $\hat{x}_j^t$ whose domain is $[0, \sum_{z \in \mathbf{Z}_i} \rho(\text{argmax}_{a \in A_z} \rho_z(a))]$ and represents the aggregated energy consumed by all the devices in the home at each time step t.
- CONSTRAINTS: There are three types of constraints:
  - *Local* soft constraints (i.e., constraints that involve only variables controlled by one agent) whose costs correspond to the weighted summation of monetary costs, as defined in Equation (1).
  - *Local* hard constraints that enforce Constraint (3). Feasible schedules incur a cost of 0 while infeasible schedules incur a cost of ∞.
  - *Global* soft constraints (i.e., constraints that involve variables controlled by different agents) whose costs correspond to the peak energy consumption, as defined in the second term in Equation (2).

The neighbors N_{a_i} of agent a_i are defined as all the agents in the coalition $\mathcal{H}$ that contains h_i.

8.3. MVA-based MGM

The SHDS problem requires several homes to solve a complex scheduling subproblem, which involves multiple variables, and to optimize the resulting energy profiles among the neighborhood of homes. The problem can be thought of as each agent solving a local complex subproblem, whose optimization is influenced by the energy profiles of the agent's neighbors, and by a coordination algorithm that allows agents to exchange their newly computed energy profiles. We thus adopt the multiple-variable agent (MVA) decomposition [14] to delegate the resolution of the agent's local problem to a centralized solver while managing inter-agent coordination through a message-passing procedure similar to that operated by MGM, described in Section 2.3. The resulting algorithm is called *Smart Home MGM (SH-MGM)*. SH-MGM is a distributed algorithm that operates in synchronous cycles. The algorithm first finds a feasible DCOP solution and then iteratively improves it, at each cycle, until convergence or time out. We leave the details out and refer the interested reader to [15].

8.4. Empirical Evaluation

Our empirical evaluations compare the SH-MGM algorithm against an uncoordinated greedy approach, where each agent computes a feasible schedule for its home devices without optimizing over its cost and the aggregated peak energy incurred.

Each agent controls nine smart actuators to schedule—Kenmore oven and dishwasher, GE clothes washer and dryer, iRobot vacuum cleaner, Tesla electric vehicle, LG air conditioner, Bryant heat pump, and American water heater—and five sensors. We selected these devices as they can be available in a typical (smart) home and they have published statistics (e.g., energy consumption profiles). The algorithms take as inputs a list of smart devices to schedule as well as their associated scheduling rules and the real-time pricing scheme adopted. Each device has an associated active scheduling rule that is randomly generated for

each agent and a number of passive rules that must always hold. To find local schedules at each SH-MGM iteration, each agent uses a Constraint Programming solver[8] as a subroutine. An extended description of the smart device properties, the structural parameters (i.e., size, material, heat loss) of the homes, and the predictive models for the homes and devices state properties is reported in [19]. Finally, we set $H = 12$ and adopted a pricing scheme used by the Pacific Gas & Electric Co. for its customers in parts of California,[9] which accounts for seven tiers ranging from \$0.198 per kWh to \$0.849 per kWh.

In our first experiment, we implemented the algorithm on an actual distributed system of Raspberry Pis. A Raspberry Pi (called "PI" for short) is a bare-bone single-board computer with limited computation and storage capabilities. We used Raspberry Pi 2 Model B with quadcore 900MHz CPU and 1GB of RAM. We implemented the SH-MGM algorithm using the Java Agent Development Environment (JADE) framework,[10] which provides agent abstractions and peer-to-peer agent communication based on asynchronous message passing. Each PI implements the logic for one agent, and the agent's communication is supported through JADE and using a wired network connected to a router.

We set up our experiments with seven PIs, each controlling the nine smart actuators and five sensors described above. All agents belong to the same coalition. We set the equal weights α_c and α_e of the objective (see Equation (2)) to 0.5. Figure 10(left) illustrates the results of this experiment, where we imposed a 60 seconds timeout for the CP solver. As expected, the SH-MGM solution improves with increasing number of cycles, providing an economic advantage for the users as well as peak energy reduction, when compared to the uncoordinated schema. These results, thus, show the feasibility of using a local search-based schema implemented on hardware with limited storage and processing power to solve a complex problem.

In our second set of experiments, we generate synthetic microgrid instances sampling neighborhoods in three cities in the United States (Des Moines, IA; Boston, MA; and San Francisco, CA) and estimate the density of houses in each city. The average density (in houses per square kilometers) is 718 in Des Moines, 1357 in Boston, and 3766 in San Francisco. For each city, we created a 200m×200m grid, where the distance between intersections is 20m, and randomly placed houses in this grid until the density is the same as the sampled density. We then divided the city into k (=$|\mathcal{H}|$) coalitions, where each home can communicate with all homes in its coalition. In generating our SHDS topologies, we verified that the resulting topology is connected. Finally, we ensure that there are no disjoint coalitions; this is analogous to the fact that microgrids are all connected to each other via the main power grid.

In these experiments, we imposed a 10 seconds timeout for the agents' CP solver. We first evaluate the effects of the weights α_c and α_e (see Equation (2)) on the quality of solutions found by the algorithms. We evaluate three configurations: $(\alpha_c = 1.0, \alpha_e = 0.0)$, $(\alpha_c = 0.5, \alpha_e = 0.5)$, and $(\alpha_c = 0.0, \alpha_e = 1.0)$, using randomly generated microgrids with the density of Des Moines, IA. Figure 10(middle) shows the total energy consumption per hour (in kWh) of the day under each configuration. Figure 10(right) illustrates the average daily cost paid by one household under the different objective weights. The uncoordinated greedy approach achieves the worst performance, with the highest energy peak at 2852 kWh and the most expensive daily cost (\$3.84). The configuration that disregards the local cost reduction ($\alpha_c = 0$) reports the best energy peak reduction (peak at 461 kWh) but highest daily cost among the coordinated configurations (\$2.31). On the other extreme, the configuration that disregards the global peak reduction ($\alpha_p = 0$) reports the worst energy peak reduction (peak at 1738 kWh) but the lowest daily cost among the coordinated configurations (\$1.44). Finally, the configuration with $\alpha_c = \alpha_e = 0.5$ reports intermediate results, with the highest peak at 539 kWh and a daily cost of \$2.18.

For a more extensive analysis of the results, we refer the reader to [15].

9. Conclusions

Responding to the recent call to action for AI researchers to contribute towards the smart grid vision [29], this paper presented two important applications of *Distributed Constraint Optimization Problems* (DCOPs) for *demand response* (DR) programs in smart grids. The multi-agent *Economic Dispatch with Demand Response* (EDDR) provides an integrated approach to the economic dispatch and the DR model for

[8] We adopt the JaCoP solver (http://www.jacop.eu/)
[9] https://goo.gl/vOeNqj/
[10] http://jade.tilab.com/

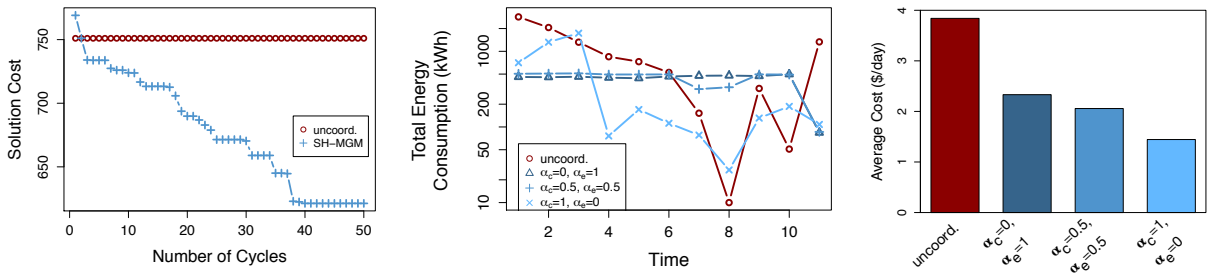

Fig. 10. Physical Experimental Result with PIs (left); Synthetic Experimental Results Varying α_c and α_e (middle and right).

power systems. The *Smart Home Device Scheduling* (SHDS) problem formalizes the device scheduling and coordination problem across multiple smart homes to reduce the energy peaks.

Due to the complexity of the problems the adoption of off-the-shelf DCOPs resolutions algorithms is infeasible. Thus, the paper introduces a methodology to exploit the structural problem decomposition (called MVA) for DCOPs, which enables agents to solve complex local sub-problems efficiently. It further discusses a general GPU parallelization schema for inference-based DCOP algorithms, which produces speed-ups of up to 2 order of magnitude.

In the context of EDDR applications, evaluations on a state-of-the-art power systems simulator show that the solutions found are stable within acceptable frequency deviations. In the context of home automation, the proposed algorithm was deployed on a distributed system of Raspberry Pis, each capable of controlling and scheduling smart devices through hardware interfaces. The experimental results show that this algorithm outperforms a simple uncoordinated solution on realistic small-scale experiments as well as large-scale synthetic experiments.

Therefore, this work continues to pave the bridge between the AI and power systems communities, highlighting the strengths and applicability of AI techniques in solving power system problems.

Acknowledgments

We would like to thank all the friends that have worked with us in this challenging research area, namely, Federico Campeotto, Alessandro Dal Palù, Rina Dechter, Khoi D. Hoang, Ping Hou, William Kluegel, Muhammad Aamir Iqbal, Tiep Le, Tran Cao Son, Athena M. Tabakhi, Makoto Yokoo, Roie Zivan, and, in particular, William Yeoh.

A. Dovier is partially supported by Indam GNCS 2014–2017 grants, and by PRID ENCASE. E. Pontelli has been supported by NSF grants CNS-1440911, CBET-14016539, HRD-1345232.

References

[1] Baruch Awerbuch. A new distributed depth-first-search algorithm. *Information Processing Letters*, 20(3):147–150, 1985.

[2] Emma Bowring, Milind Tambe, and Makoto Yokoo. Multiply-constrained distributed constraint optimization. In *Proc. of AAMAS*, pages 1413–1420, 2006.

[3] David Burke and Kenneth Brown. Efficiently handling complex local problems in distributed constraint optimisation. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 701–702, 2006.

[4] Federico Campeotto, Agostino Dovier, Ferdinando Fioretto, and Enrico 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.

[5] Federico Campeotto, Alessandro Dal Palù, Agostino Dovier, Ferdinando Fioretto, and Enrico Pontelli. Exploring the Use of GPUs in Constraint Solving. In *Proceedings of the Practical Aspects of Declarative Languages (PADL)*, pages 152–167, 2014.

[6] Imen Chakroun, Mohand-Said Mezmaz, Nouredine Melab, and Ahcene Bendjoudi. Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm. *Concurrency and Computation: Practice and Experience*, 25(8):1121–1136, 2013.

[7] Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. CUD@SAT: SAT solving on GPUs. *J. Exp. Theor. Artif. Intell.*, 27(3):293–316, 2015.

[8] Gregory Frederick Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. SIMD re-convergence at thread frontiers. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, pages 477–488, 2011.

[9] Alessandro Farinelli, Alex Rogers, Adrian Petcu, and Nicholas 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.

[10] Ferdinando Fioretto, Federico Campeotto, Luca Da Rin Fioretto, William Yeoh, and Enrico Pontelli. GD-GIBBS: a GPU-based sampling algorithm for solving distributed constraint optimization problems. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1339–1340, 2014.

[11] Ferdinando Fioretto, Enrico Pontelli, and William Yeoh. Distributed constraint optimization problems and applications: A survey. *CoRR*, abs/1602.06347, 2016.

[12] Ferdinando Fioretto, Enrico Pontelli, and William Yeoh. Distributed constraint optimization problems and applications: A survey. *Journal of Artificial Intelligence Research (JAIR)*, (to appear), 2018.

[13] Ferdinando Fioretto, Enrico Pontelli, William Yeoh, and Rina Dechter. Accelerating exact and approximate inference for (distributed) discrete optimization with GPUs. *Constraints*, 23(1):1–43, Jan 2018.

[14] Ferdinando Fioretto, William Yeoh, and Enrico Pontelli. Multi-variable agents decomposition for DCOPs. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 2480–2486, 2016.

[15] Ferdinando Fioretto, William Yeoh, and Enrico Pontelli. A multiagent system approach to scheduling devices in smart homes. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 981–989, 2017.

[16] Ferdinando Fioretto, William Yeoh, Enrico Pontelli, Ye Ma, and Satishkumar Ranade. A DCOP approach to the economic dispatch with demand response. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2017.

[17] Rachel Greenstadt, Jonathan Pearce, and Milind Tambe. Analysis of privacy loss in DCOP algorithms. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 647–653, 2006.

[18] Tianyi David Han and Tarek S. Abdelrahman. Reducing Branch Divergence in GPU Programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pages 3:1–3:8, New York, NY, 2011. ACM Press.

[19] William Kluegel, Muhammad A. Iqbal, Ferdinando Fioretto, William Yeoh, and Enrico Pontelli. A realistic dataset for the smart home device scheduling problem for DCOPs. In Gita Sukthankar and Juan A. Rodriguez-Aguilar, editors, *Autonomous Agents and Multiagent Systems: AAMAS 2017 Workshops, Visionary Papers, São Paulo, Brazil, May 8-12, 2017, Revised Selected Papers*, pages 125–142, Cham, 2017. Springer International Publishing.

[20] Thomas Léauté and Boi Faltings. Distributed constraint optimization under stochastic uncertainty. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 68–73, 2011.

[21] T. Logenthiran, D. Srinivasan, and T. Shun. Demand side management in smart grid using heuristic optimization. *IEEE Transactions on Smart Grid*, 3(3):1244–1252, 2012.

[22] Ye Ma, Wei Zhang, Wenxin Liu, and Qinmin Yang. Fully distributed social welfare optimization with line flow constraint consideration. *IEEE Transaction on Industrial Informatics*, 11(6):1532–1541, 2015.

[23] Rajiv T. Maheswaran, Jonathan P. Pearce, and Milind Tambe. Distributed Algorithms for DCOP: A Graphical-Game-Based Approach. In *Proceedings of the ISCA 17th International Conference on Parallel and Distributed Computing Systems*, pages 432–439, 2004.

[24] Sam Miller, Sarvapali D. Ramchurn, and Alex Rogers. Optimal decentralised dispatch of embedded generation in the smart grid. In *AAMAS*, pages 281–288, 2012.

[25] Pragnesh Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1–2):149–180, 2005.

[26] Arnon Netzer, Alon Grubshtein, and Amnon Meisels. Concurrent forward bounding for distributed constraint optimization problems. *AI Journal*, 193:186–216, 2012.

[27] Adrian Petcu and Boi Faltings. A scalable method for multiagent constraint optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1413–1420, 2005.

[28] Adrian Petcu and Boi Faltings. Superstabilizing, fault-containing distributed combinatorial optimization. In *Proc. of AAAI*, pages 449–454, 2005.

[29] Sarvapali D Ramchurn, Perukrishnen Vytelingum, Alex Rogers, and Nicholas R Jennings. Putting the 'smarts' into the smart grid: A grand challenge for artificial intelligence. *Communications of the ACM*, 55(4):86–97, 2012.

[30] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[31] Pierre Rust, Gauthier Picard, and Fano Ramparany. Using message-passing DCOP algorithms to solve energy-efficient smart environment configuration problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 468–474, 2016.

[32] Jason Sanders and Edward Kandrot. *CUDA by Example. An Introduction to General-Purpose GPU Programming*. Addison Wesley, 2010.

[33] Evan Sultanik, Pragnesh Jay Modi, and William 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.

[34] D. I. Sun, B. Ashley, B. Brewer, A. Hughes, and W. F. Tinney. Optimal power flow by Newton approach. *IEEE Transactions on Power Apparatus and Systems*, PAS-103(10):2864–2880, Oct 1984.

[35] Menkes Van Den Briel, Paul Scott, Sylvie Thiébaux, et al. Randomized load control: A simple distributed approach for scheduling smart appliances. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2013.

[36] Meritxell Vinyals, Juan A. Rodríguez-Aguilar, and Jesús Cerquides. Constructing a unifying theory of dynamic programming DCOP algorithms via the generalized distributive law. *Autonomous Agents and Multi-Agent Systems*, 22(3):439–464, 2011.

[37] T. Voice, P. Vytelingum, S. Ramchurn, A. Rogers, and N. Jennings. Decentralised control of micro-storage in the smart grid.

In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 1421–1427, 2011.

[38] S. Wang, S. Shahidehpour, D. Kirschen, S. Mokhtari, and G. Irisarri. Short-term generation scheduling with transmission and environmental constraints using an augmented lagrangian relaxation. *IEEE Transaction on Power Systems*, 10(3):1294–1301, 1995.

[39] William Yeoh, Ariel Felner, and Sven Koenig. BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research*, 38:85–133, 2010.

[40] William Yeoh and Makoto Yokoo. Distributed problem solving. *AI Magazine*, 33(3):53–65, 2012.

[41] Makoto Yokoo, editor. *Distributed Constraint Satisfaction: Foundation of Cooperation in Multi-agent Systems*. Springer, 2001.

[42] Roie Zivan, Harel Yedidsion, Steven Okamoto, Robin Glinton, and Katia Sycara. Distributed constraint optimization for teams of mobile sensing agents. *Journal of Autonomous Agents and Multi-Agent Systems*, 29(3):495–536, 2015.