GRAPH-BASED MODELING AND SIMULATION OF CYBER-PHYSICAL SYSTEMS

by

JORDAN H. JALVING

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy
(Chemical Engineering)

at the
UNIVERSITY OF WISCONSIN-MADISON
2020

Date of final oral examination: July 10, 2020

The dissertation is approved by the following members of the Final Oral Committee:

Victor M. Zavala, Associate Professor, Chemical and Biological Engineering
Reid Van Lehn, Assistant Professor, Chemical and Biological Engineering
Ophelia Venturelli, Assistant Professor, Chemical and Biological Engineering
James Luedtke, Professor, Industrial and Systems Engineering

To the most amazing dog. The one, the only, Cash Monroe.

ACKNOWLEDGMENTS

I would like to acknowledge everyone who inspired and helped me in the pursuit of a doctoral degree. First and foremost, I thank my advisor, Prof. Victor M. Zavala for his guidance and support all these years. There were many instances throughout my time here where my focus strayed heavily into tangential software development activities and Victor kept me grounded in the important research questions (which ultimately produced better software).

I would like to acknowledge everyone in the Zavalab research group for all of their support and friendship over these last five years. I would particularly like to thank Apoorva Sampat and Ranjeet Kumar for spending all the late nights in the office with me working on model predictive control assignments. I would also like to thank Prof. Alex Dowling and Prof. Yankai Cao for their mentorship early on in the program for which I would have struggled considerably without. I am certainly grateful for having the chance to collaborate with Sungho Shin who provided tremendous help in creating new optimization solvers and helping me to better develop my own research.

I am also grateful for the valuable input from my defense committee: Prof. Reid Van Lehn, Prof. Ophelia Venturelli, and Prof. Jim Luedtke. I continue to find new interesting research topics related to this thesis and their input was extremely helpful towards coming up with new future research ideas.

During my time at UW-Madison, I received considerable help and support from my collaborators at Argonne National Laboratory. In that regard, I would like to acknowledge Kibaek Kim for his mentorship during my summer spent there and for helping to produce the initial prototype of the `Plasmo.jl` software package. I would also like to acknowledge Eric Tatara for his valuable input in the application of `Plasmo.jl` for infrastructure modeling, as well as Charles Macal and Megan Clifford for their continuous support to further develop the `Plasmo.jl` platform.

I thank all of my family members for being supportive during my time here. I especially thank my older brother Jonathan for being a continuous role model throughout my life and for convincing me to pursue a doctoral degree despite my original resistance to do so. I would also like to thank my work mentor Dr. Timothy Laska for getting me inter-

ested in research and computation before I attended graduate school. Before I met Tim, I had no idea what research really was and I would never have pursued a PhD if it wasn't for his influence and support.

This doctoral degree was funded by awards from both the National Science Foundation and the Department of Energy. I gratefully acknowledge the financial support from U.S. Department of Energy (DOE), Office of Science, under Contract No. DE-AC02-06CH11357 and the DOE Office of Electricity Delivery and Energy ReliabilityâĂŹs Advanced Grid Research and Development program at Argonne National Laboratory. I am also grateful for the financial support from the National Science Foundation under award NSF-EECS-1609183.

Most of all, I thank my strongest supporters – Cash the dog and Marilyn my fiancée – for supporting me through the entirety of my time in Madison while living 1,525 miles away in Miami. Those warm sun-filled visits kept me going through the coldest Wisconsin winters.

Jordan H. Jalving
Madison, WI
July 2020

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This dissertation presents new computational abstractions to model complex behaviors in cyber-physical systems. The need to model cyber-physical systems is becoming increasingly important, but capturing their underlying behavior in a coherent fashion is technically challenging. Physical aspects of cyber-physical systems are described by algebraic equations that contain complex dependencies, and cyber aspects are described by algorithms that require capturing complex communication topologies and computational timings.

To address these challenges, this dissertation first presents a new algebraic modeling framework called the `OptiGraph`. The `OptiGraph` captures physical connectivity in complex optimization (physical) models, enables the construction of hierarchical optimization structures, and facilitates systematic modeling and manipulation capabilities. The `OptiGraph` also facilitates the decomposition of complex optimization problems which enables the exploitation of advanced parallel optimization solvers and algorithms.

Next, this dissertation presents a new modeling abstraction to capture cyber aspects called the `ComputingGraph`. The `ComputingGraph` captures complex cyber behaviors such as latency and asynchronous communication which can be used to evaluate the performance of various distributed control algorithms that execute in real-time subject to delays and failures. The `ComputingGraph` can also be used to simulate and benchmark distributed algorithms which run on heterogeneous computing architectures.

The proposed abstractions are scalable and are used as the backbone of open-source `Julia`-based software packages called `Plasmo.jl` (which implements the `OptiGraph`) and `PlasmoCompute.jl` (which implements the `ComputingGraph`). With the developed packages, this dissertation tackles challenging problems that include the optimization of large-scale infrastructure systems, the evaluation of complex distributed control architectures, and the simulation of distributed optimization and machine learning algorithms.

Chapter 1

INTRODUCTION

In this chapter, we present the overall objectives of the considered research problem. We introduce cyber-physical systems and discuss the primary challenges that arise in modeling such systems. We also summarize existing modeling paradigms that target different aspects of cyber-physical systems and present new graph-based modeling concepts. We finally provide some background information and graph terminology used throughout the dissertation.

1.1 Cyber-Physical Systems

Emerging problems in complex systems are becoming more *cyber-physical* in nature, in the sense that a physical system is driven by decisions made by a cyber (or computing) system [81]. As an example, a chemical process can be considered a physical system that is driven by decisions made by a control system, which is in turn a cyber system comprised of devices (e.g., sensors, controllers, actuators) that execute diverse computing tasks (e.g., data processing and control action computation) and that exchange signals and data (e.g., measurements and actions) through a communication network. As another example, consider Figure 1.1 which realizes a cyber-physical system that consists of a physical natural gas network that is driven by a distributed control system. In this depiction, the natural gas system is comprised of physical equipment (pipelines, junctions,

and compressors) and is driven by a cyber system comprised of three control systems that exchange information and drive the gas network through control signals. The devices that execute the tasks of a control system form a *computing architecture*, similar in spirit to a parallel computing cluster in which processors are connected through a communication network. Modeling and simulating the behavior of cyber-physical systems is

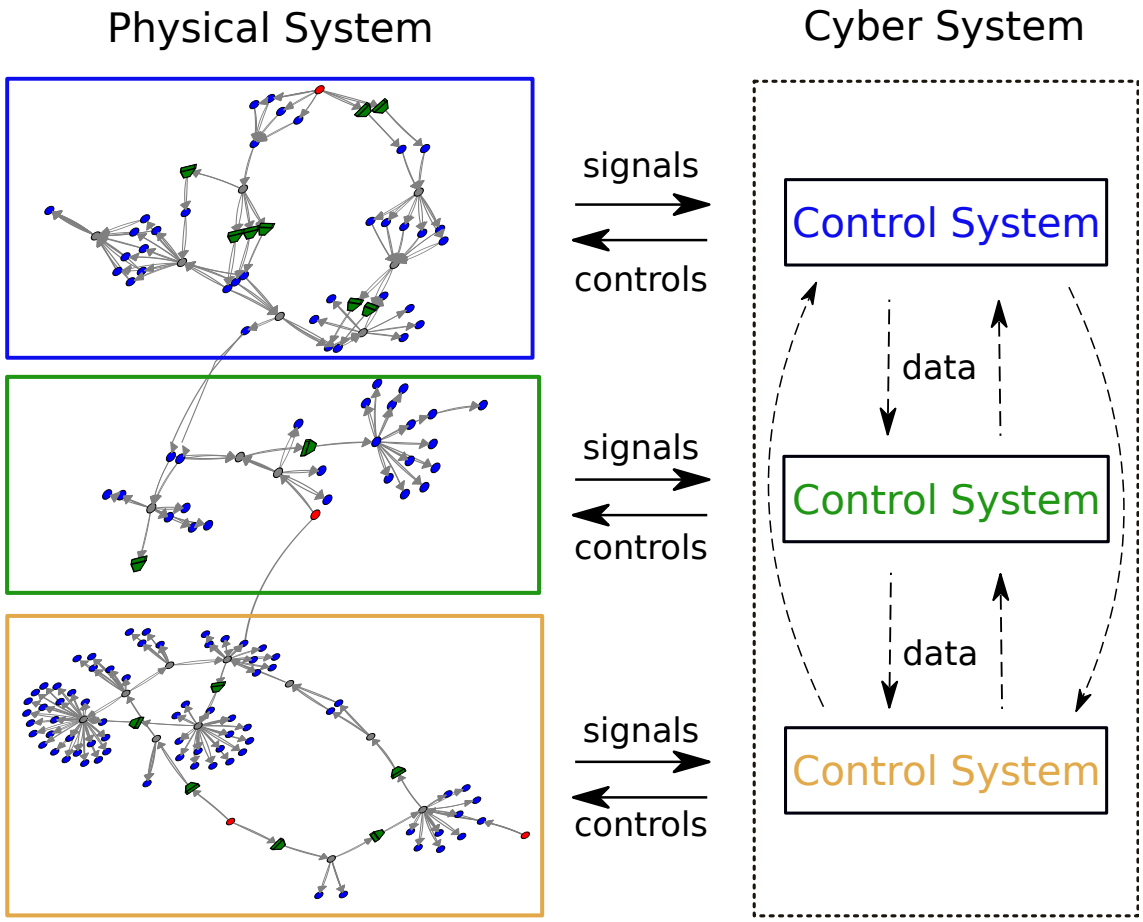
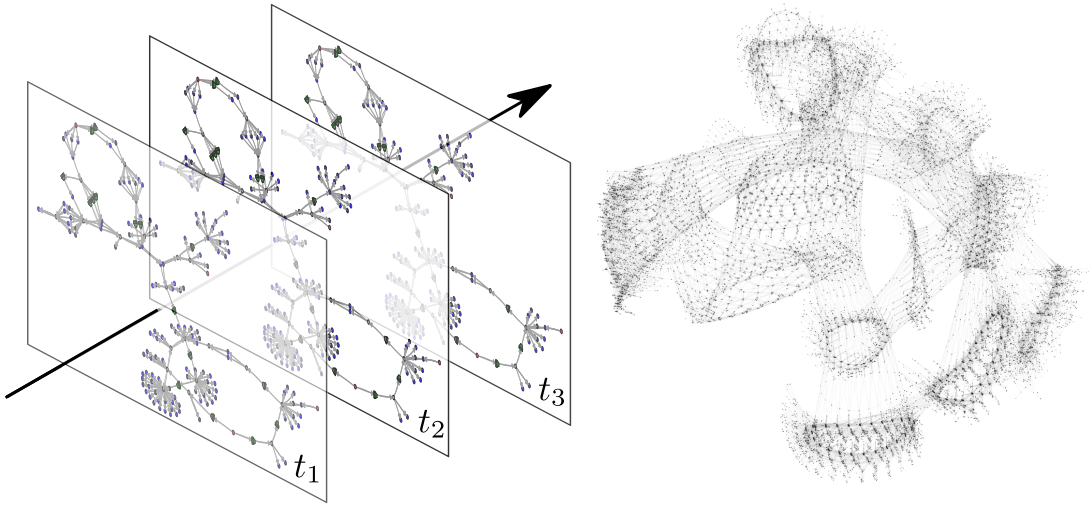


Figure 1.1: Representation of a cyber-physical system.

becoming increasingly important but it is technically challenging. In particular, emerging paradigms such as distributed and high performance computing are drastically changing the landscape of decision-making architectures driven by the need to process increasingly larger amounts of data in a distributed manner while making decisions faster and in a more scalable manner. Cyber-physical architectures also need to balance diverse issues

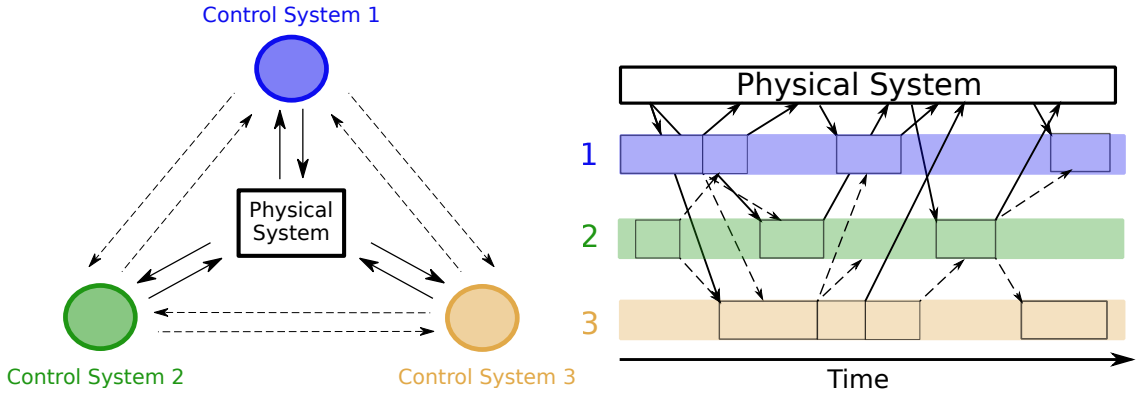
such as economic performance, safety, data privacy, as well as computing and communication latency and failures. For instance, failure of a computing device (e.g., a sensor) can lead to significant losses in performance or to full collapse of the physical system.

Much of the challenge in modeling and simulating cyber-physical systems arises from their underlying physical and cyber elements which are difficult to capture in a coherent way. For instance, the behavior of a physical system is expressed in the form of *algebraic equations* while the behavior of a cyber system is expressed mathematically in the form of *algorithms*. In the case of realistic large-scale systems, the physical equations can produce complex space-time dependencies which are difficult to manage and exploit. For instance, the left side of Figure 1.2 depicts a seemingly straightforward physical system to model (a line through time), but the underlying algebraic structure on the right reveals considerable complexity. Moreover, in modeling a cyber system, one must consider the fact that algorithms are executed under highly heterogeneous and dynamic computing architectures that exhibit complex computing and communication protocols and logic (e.g., synchronous and asynchronous) and associated time delays [80] such as depicted in Figure 1.3.



Space-Time Physical System Space-Time Algebraic Structure

Figure 1.2: Depiction of complexity that arises in modeling complex physical systems.



Cyber System Topology Cyber System Timings

Figure 1.3: Depiction of complexity that arises in modeling complex cyber systems.

1.2 Prominent Modeling and Simulation Approaches

This dissertation ultimately presents new *graph-based* modeling abstractions that target the modeling and simulation of cyber-physical systems. Graph-based representations (such as pictured in Figure 1.4) have appeared in diverse modeling platforms over the last few decades owing to their flexible approach for representing complex structures and relationships.

1.2.1 *Modeling Physical Aspects*

In the context of modeling and optimization over *physical systems*, graph-based abstractions are promising because they provide a flexible modeling paradigm to analyze complex structures and they facilitate model processing and manipulation tasks. For example, early chemical process modeling utilized sequential modular and equation-based flowsheeting tools which used graph-theoretical insights to express equations in a modular form and to facilitate object-oriented software implementations, analysis, and algorithmic development [133]. Such developments are the basis of powerful simulation environments such as `Ascend` [101], `AspenPlus`, and `gPROMS` [40]. Graph-theoretical concepts have also been widely used for expressing and processing algebraic models in platforms such as

AMPL [47], GAMS [33], AIMMS [17], CasADi [9], JuMP [37], and Pyomo [59]. The Modelica [48] simulation platform also uses graph concepts such as modularity and inheritance to instantiate and simulate complex systems.

1.2.2 *Modeling Cyber Aspects*

In the context of modeling and simulating *cyber systems*, the most popular framework is by far Simulink [89]. Simulink uses graph concepts to express cyber systems with blocks of operators which represent computing tasks and are connected by communication channels between input and output ports. Simulink provides a consistent representation of the dynamics that arise in cyber systems but it is challenging to create large-scale models. Agent-based modeling platforms can also be used to simulate cyber systems; under this abstraction, agents make decisions and communicate under channels [86] in the same spirit as information flow in a graph. Popular agent-based simulation tools include RePast [96, 30], MASON [85], and Swarm [92]. Agent-based platforms often target scalable model development, but it is often challenging to capture the timing aspects that arise in cyber systems (e.g. latency and delays) using agent-based approaches. Popular agent-based simulation tools include RePast [96, 30], MASON [85], and Swarm [92].

1.3 Research Objectives

As discussed, graph concepts pervade many powerful modeling paradigms and frameworks and graph-based modeling is promising for its ability to manage and analyze the complexity that arises in cyber-physical systems. While the aforementioned physical and cyber modeling tools exploit such graph concepts to facilitate their implementation, they do not use a *coherent* abstraction, which is key to extensibility. Consequently, typical abstractions are implemented in ad-hoc fashion and/or target very specific engineering domains such as control systems, networks, and chemical processes.

This dissertation introduces new graph-based abstractions that facilitate the diverse

modeling and simulation approaches needed to model complex cyber-physical systems. To this end, we seek to develop *general* abstractions to model cyber-physical systems that are intuitive and extensible. More specifically, we propose the concept of an `OptiGraph` to facilitate modeling of optimization problems over physical systems and the concept of a `ComputingGraph` to facilitate the simulation of cyber systems. The graph abstractions exploit *physical and communication topologies* to facilitate model construction, data management, and analysis.

With our goal to introduce powerful new modeling abstractions, this dissertation seeks to accomplish the following objectives:

- to present the novel graph-based abstraction called the `OptiGraph` to model optimization problems over complex physical systems
- to introduce the `ComputingGraph` abstraction to capture computation and communication aspects to simulate cyber systems
- to present the open-source software frameworks `Plasmo.jl` and `PlasmoCompute.jl` which implement the `OptiGraph` and `ComputingGraph` abstractions and show such abstractions facilitate extensible modeling syntax
- to provide detailed case studies that demonstrate the above abstractions in challenging problems in infrastructure optimization and distributed algorithm simulation

1.4 Thesis Overview

This dissertation is organized as follows. We first introduce and highlight the capabilities of the `OptiGraph` abstraction in Chapters 2 and 3. In Chapter 2 we present the `OptiGraph` representation and show how it can model complex optimization problems in physical systems and perform data and model manipulation tasks. We also introduce the corresponding software `Plasmo.jl` and provide a challenging state estimation case study for a natural gas network to demonstrate the presented modeling capabilities. In Chapter 3

we present decomposition capabilities with the `OptiGraph` and `Plasmo.jl`. We show how the `OptiGraph` is a natural interface to use partitioning tools to decompose optimization problems and interface with advanced decomposition-based optimization solvers.

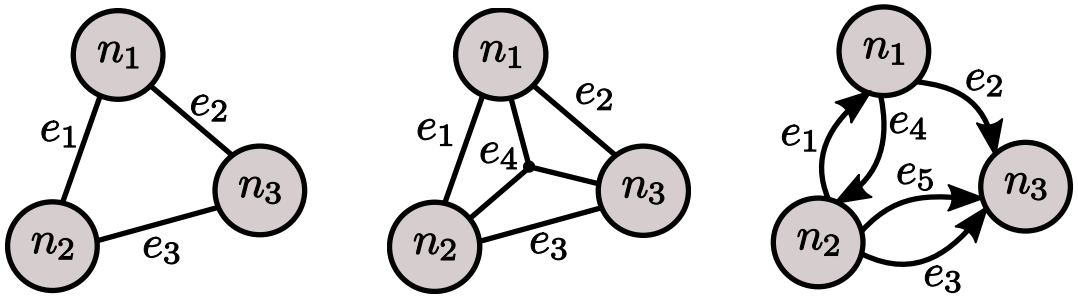
Next, Chapter 4 discusses large-scale optimization approaches in infrastructure networks and presents challenging case studies. This chapter utilizes `OptiGraph` concepts to model a large-scale coupled infrastructure problem, and to decompose the inherent space-time structure in an optimal control problem. This chapter also shows how the `OptiGraph` facilitates the creation of interfaces with high-fidelity simulation tools and provides a case study that combines the presented optimization approaches with a mature commercial simulator.

Chapter 5 introduces the `ComputingGraph` to model and simulate computing and timing aspects that arise cyber systems. This chapter also introduces the associated software framework `PlasmoCompute.jl` and provides a case study that shows how to simulate real-time model predictive control architectures and study computational and communication failures.

Finally, Chapter 6 uses `ComputingGraph` concepts with `PlasmoCompute.jl` to perform distributed algorithm simulation. We discuss the challenges that arise in implementing distributed optimization and machine learning algorithms, and show how different distributed algorithm variants can be developed and simulated with a `ComputingGraph`. We close with Chapter 7 where we highlight major contributions and discuss future research avenues in cyber-physical systems.

1.5 Graph Notation

Throughout this dissertation we commonly refer to graph concepts to express the proposed abstractions. As such, it helpful to introduce background and notation which we follow throughout each chapter. A graph $\mathcal{G}(\mathcal{N}, \mathcal{E})$ is described by a collection of nodes $\mathcal{N}$ and edges $\mathcal{E}$. The set of nodes that belong to a specific graph $\mathcal{G}$ are denoted using the



(a) Graph (b) Hypergraph (c) Multigraph

Figure 1.4: Graph representations. A simple graph (left) with three nodes and three simple edges. A hypergraph (middle) with three nodes, three edges and one hyperedge. A multigraph (right) with three nodes and five directed edges.

syntax $\mathcal{N}(\mathcal{G})$ and the elements are denoted by the index $n \in \mathcal{N}(\mathcal{G})$. Similarly, the set of edges that belong to $\mathcal{G}$ are denoted by $\mathcal{E}(\mathcal{G})$ the edge elements are given using $e \in \mathcal{E}(\mathcal{G})$. The set of supporting nodes of an edge e (nodes that the edge connects) are denoted as $\mathcal{N}(e) \subseteq \mathcal{N}$ and the set of supporting edges for a node n (edges connected to the node) are given by $\mathcal{E}(n) \subseteq \mathcal{E}$. In a standard graph, two nodes are connected by an edge e. For example, nodes n_1 and n_2 are connected by the single edge e_1 in the left panel of Figure 1.4. In a hypergraph, multiple nodes can be connected by a single edge. Hypergraphs are useful for describing algebraic structures of systems. For example, the middle panel in Figure 1.4 is a hypergraph that contains an edge e_4 that connects all three nodes. In a multigraph, multiple edges can connect two nodes (the supporting nodes $\mathcal{N}(e)$ may correspond to more than one edge).

Connectivity between nodes and edges is usually expressed in terms of the incidence matrix $A \in \mathcal{R}^{|\mathcal{N}|\times|\mathcal{E}|}$ (where the notation $|\mathcal{S}|$ for set $\mathcal{S}$ denotes its cardinality). For an undirected standard graph we have $A_{n,e} = 1$ if $n \in \mathcal{N}(e)$ or $e \in \mathcal{E}(n)$. For a standard undirected graph we have $\sum_{n\in\mathcal{N}} A_{n,e} = 2$ for all $e \in \mathcal{E}$ (because in a standard graph an edge only has two supporting nodes). The degree of a node n is the number of edges connected to the node and can be computed as $\deg(n) = |\mathcal{E}(n)| = \sum_{e\in\mathcal{E}} |A_{n,e}|$. The set of nodes connected to node n (without counting self-loops) is denoted as $\mathcal{N}(n)$ and for a standard graph we have that $|\mathcal{N}(n)| = |\mathcal{E}(n)|$.

Chapter 2

GRAPH-BASED MODELING FOR PHYSICAL SYSTEMS

This chapter explores challenges that arise in creating optimization problems over complex *physical systems*. We introduce a new *graph-based* modeling abstraction called the `OptiGraph` which facilitates the flexible representation of optimization models and provides capabilities to manage complex creation, processing, and data management tasks. We introduce the associated software framework called `Plasmo.jl` which implements the `OptiGraph` abstraction and we showcase the presented capabilities with a case study in state estimation.

2.1 Introduction

A key contribution of this dissertation is the introduction of the `OptiGraph` [69] as a new modeling paradigm for optimization. At its core, the `OptiGraph` is a *graph-based* modeling abstraction that facilitates the creation, processing, analysis, and decomposition of complex optimization models using graph concepts. This abstraction stems from the idea that an optimization model can be represented as a *graph*. Specifically, an optimization model can be represented as a collection of algebraic functions (constraints and objectives) that are connected via variables which naturally forms a graph. Such graph-based concepts are not necessarily new; graph representations of optimization models are used in mod-

eling environments to perform automatic differentiation and model processing tasks [46]. The underlying graph structure of an optimization model is also implicitly communicated to solvers in the form of sparsity patterns of constraint, objective, and derivative matrices. However, these graph representations operate at a level of granularity that is not particularly useful to a modeler.

Fundamentally, the `OptiGraph` extends from algebraic and object-oriented modeling paradigms. In an optimization context, most modeling approaches adhere to an algebraic modeling paradigm wherein a model is assembled by adding functions and variables (e.g., using packages such as `GAMS`, `AMPL`, `Pyomo`, or `JuMP`). In contrast, the object-oriented paradigm assembles models by adding *blocks* of functions and variables. The object-oriented paradigm is widely used in engineering communities to perform simulation using packages such as `Modelica`, `AspenPlus`, and `gProms` [48, 90, 36]. An important observation is that object-oriented modeling more naturally lends itself to the expression and exploitation of problem structures because the underlying graph is progressively built by the modeler. For instance, packages such as `AspenPlus` use the underlying graph structure induced by blocks to partition the model and communicate it to a decomposition algorithm. In a sense, object-oriented modeling could be considered as a form of graph-based modeling.

In an `OptiGraph`, an optimization model is treated as a *hierarchical graph*. At a given level, a graph comprises a set of nodes and edges; each node contains an optimization model (with variables, objectives, constraints, and data) and each edge captures connectivity between node models. Importantly, the optimization model in each node can be expressed algebraically (as in a standard algebraic modeling language) or as a graph (which enables the creation of hierarchical graphs). This provides flexibility to capture both algebraic and object-oriented modeling paradigms under the same framework. The abstraction naturally exposes problem structure to algorithms and provides a modular approach to construct models. Such modularization enables collaborative model building, independent processing of model components (e.g., automatic differentiation), and

data management. The OptiGraph can naturally capture a wide range of problem classes such as stochastic optimization (a graph is a tree), dynamic optimization (a graph is a line), networks (a graph is the network itself), PDE optimization (a graph is a mesh), and multi-scale optimization (a graph is a meshed tree). Moreover, an OptiGraph can be constructed by combining graphs from different classes (e.g., stochastic PDE optimization). The OptiGraph is thus more general than graph-based abstractions that target specific problem classes such as network optimization and control [60, 71, 94, 127, 58]. The OptiGraph also generalizes abstractions used in simulation packages such as Modelica, AspenPlus, and gProms, which are tailored to specific physical systems. Finally, the hierarchical nature of an OptiGraph can be communicated to decomposition solvers or the graph can be collapsed into a standard optimization model that can be solved with off-the-shelf solvers (e.g., Gurobi [57] or Ipopt [130]).

In the context of software, graph-based representations have been adopted in *modeling environments* for specific problem classes. For instance, SnapVX [58] uses a graph topology to formulate network optimization problems and solves them using the alternating direction method of multipliers (ADMM) [20], DISROPT is a Python-based environment that performs distributed network optimization [41], and DeCODe [8] automatically creates graph representations of optimization models created with Pyomo and uses community detection to find partitions for use with decomposition algorithms.

In the spirit of providing software that captures OptiGraph capabilities, this chapter also presents the Julia-based package Plasmo.jl [68, 69]. Plasmo.jl provides modeling syntax to systematically create hierarchical graph models and integrates with the algebraic language JuMP.jl [37] to facilitate the expression of connectivity between node models. A key feature of Plasmo.jl is that it provides useful data processing capabilities to facilitate model reduction tasks and re-use (e.g. for real-time optimization applications) [66] and provides an interface to decomposition-based solvers (this is discussed more in Chapter 3).

2.2 OptiGraphs

In this section we introduce the mathematical formulation of the `OptiGraph` and show how it facilitates data and model management tasks over complex optimization problems.

2.2.1 *Representation*

The `OptiGraph` is composed of a set of *OptiNodes* $\mathcal{N}$ (each embedding an optimization model with its local variables, constraints, objective function, and data) and a set of *OptiEdges* $\mathcal{E}$ (each embedding a set of *linking constraints*) that capture coupling between *OptiNodes* such as depicted in Figure 2.1. The `OptiGraph` is denoted as $\mathcal{OG}(\mathcal{N}, \mathcal{E})$ and contains the optimization model of interest. The `OptiEdges` $\mathcal{E}$ are hyperedges that connect two or more `OptiNodes` which makes the `OptiGraph` adhere to an undirected hypergraph representation (i.e. the hypergraph in Figure 1.4b). Whenever clear from context, we simply refer to the `OptiGraph` as graph, to the `OptiNodes` as nodes, and to `OptiEdges` as edges. We denote the set of nodes that belong to $\mathcal{OG}$ as $\mathcal{N}(\mathcal{OG})$ and the set of edges as $\mathcal{E}(\mathcal{OG})$. The topology of the underlying hypergraph is encoded in the incidence matrix $A \in \mathcal{R}^{|\mathcal{N}|\times|\mathcal{E}|}$, where the notation $|\mathcal{S}|$ denotes cardinality of set $\mathcal{S}$. The neighborhood of node n is denoted as $\mathcal{N}(n)$ and this is the set of nodes connected to n. The set of nodes that support an edge e are denoted as $\mathcal{N}(e)$ and the set of edges that are incident to node n are denoted as $\mathcal{E}(n)$.

The optimization model associated with an `OptiGraph` can be represented mathematically as:

$$\min_{\{x_n\}_{n\in\mathcal{N}(\mathcal{OG})}} \quad \sum_{n\in\mathcal{N}(\mathcal{OG})} f_n(x_n) \qquad \text{(Objective)} \tag{2.1a}$$

$$\text{s.t.} \quad x_n \in \mathcal{X}_n, \quad n \in \mathcal{N}(\mathcal{OG}), \qquad \text{(Node Constraints)} \tag{2.1b}$$

$$g_e(\{x_n\}_{n\in\mathcal{N}(e)}) = 0, \quad e \in \mathcal{E}(\mathcal{OG}). \qquad \text{(Link Constraints)} \tag{2.1c}$$

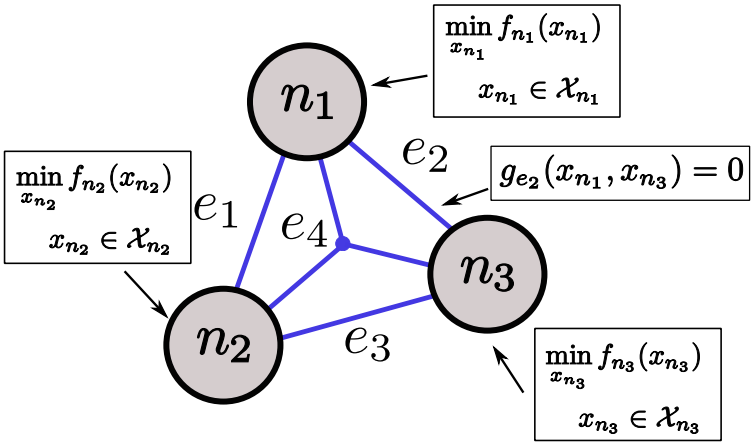


Figure 2.1: Representation of a simple `OptiGraph`. The depicted `OptiGraph` contains three `OptiNodes` connected by four `Optiedges`

.

Here, $\{x_n\}_{n \in \mathcal{N}(\mathcal{OG})}$ is the collection of decision variables over the entire set of nodes $\mathcal{N}(\mathcal{OG})$ and x_n is the set of variables on node n. Function (2.1a) is the graph objective function which is given by the sum of objective functions $f_n(x_n)$, (2.1b) represents the collection of constraints over all nodes $\mathcal{N}(\mathcal{OG})$, and (2.1c) is the collection of linking constraints over all edges $\mathcal{E}(\mathcal{OG})$. Here, the constraints of a node n are represented by the set $\mathcal{X}_n$ while the linking constraints induced by an edge e are represented by the vector function $g_e(\{x_n\}_{n \in \mathcal{N}(e)})$ (an edge can contain multiple linking constraints). Here we assume that the graph objective is obtained via a linear combination of the node objectives but other combinations are possible (e.g., to handle conflict resolution formulations wherein nodes represent different stakeholders). In addition, we assume that coupling between nodes arises in the form of complicating constraints but the definition of complicating variables is also possible (coupling variables can always be represented as coupling constraints via lifting procedures).

2.2.2 *Model and Data Management*

Besides model construction, there are several other key advantages of using the `OptiGraph`. In particular, component models are isolated from the graph topology. A benefit of this

is that it is possible to apply automatic differentiation or convexification techniques (or other processing techniques) to each component `OptiNode` $n \in \mathcal{N}(\mathcal{OG})$ separately, which often results in computational savings. It is also possible to exchange component models in a graph without altering the core topology. Moreover, model definitions remain local to an `OptiNode` and it is thus possible to *reuse* a component model template in multiple `OptiNodes` without having to alter definitions (which is not easy to do in algebraic modeling languages such as `AMPL` or `GAMS`). This feature enables modularity and re-usability and enables the implementation of models as parametric functions of *data*. To highlight this, we consider the following model graph representation:

$$\min_{\{x_n\}_{n\in\mathcal{N}(\mathcal{OG})}} \quad \sum_{n\in\mathcal{N}(\mathcal{OG})} f_n(x_n, \eta_n) \tag{2.2a}$$

$$\text{s.t.} \quad x_n \in \mathcal{X}_n(\eta_n), \quad n \in \mathcal{N}(\mathcal{OG}), \tag{2.2b}$$

$$g_e(\{x_n\}_{n\in\mathcal{N}(e)}) = 0, \quad e \in \mathcal{E}(\mathcal{OG}). \tag{2.2c}$$

where η_n is an input data (attribute) vector associated with the model of node n. This modularization approach also facilitates the implementation of warm-starting procedures, which is key in control or estimation applications [66]. The output data (solution attributes) structure inherits the structure of the `OptiGraph`, thus facilitating analysis and post-processing.

Figure 2.2 visualizes some of the data management functions one might perform with an `OptiGraph`. For instance, after creating (and solving) an `OptiGraph`, one could swap out underlying `OptiNode` component models (blue lines in the figure), one could update data which may inherently change the underlying model (red lines in the figure), or one might use the partial solution of an `OptiNode` to warm-start a new optimization problem (green lines in the figure).

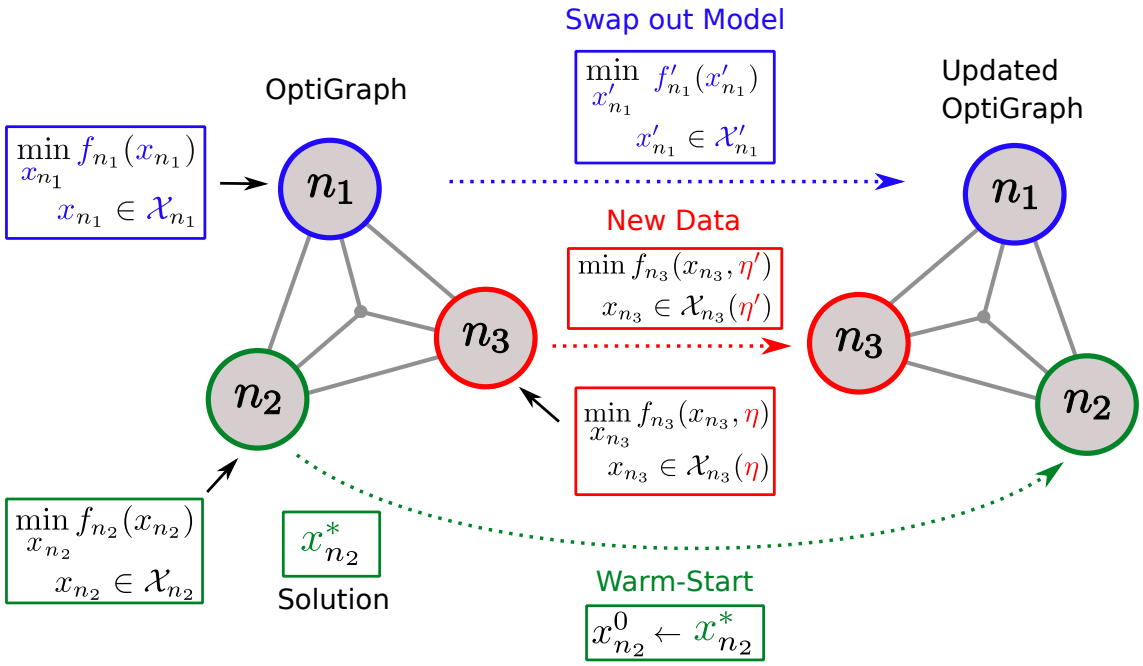


Figure 2.2: Depiction of Model and Data Management Functions with an `OptiGraph`.

2.2.3 *Hierarchical Graphs*

A key novelty of the `OptiGraph` is that it can cleanly represent hierarchical structures. This feature enables expression of models with multiple embedded structures and enables modular model building (e.g., by merging existing models). To illustrate this, consider we have the `Optigraphs` $\mathcal{OG}_i, \mathcal{OG}_j$ each with their own local nodes and edges. We assemble these low-level `OptiGraphs` (subgraphs) to build a high-level `OptiGraph` $\mathcal{OG}(\{\mathcal{OG}_i, \mathcal{OG}_j\}, \mathcal{N}_g, \mathcal{E}_g)$. We use the notation $\mathcal{OG}(\mathcal{N}, \mathcal{E})$ to indicate that the high-level graph has nodes $\mathcal{N} = \mathcal{N}_g \cup \mathcal{N}(\mathcal{OG}_i) \cup \mathcal{N}(\mathcal{OG}_j)$ and edges $\mathcal{E} = \mathcal{E}_g \cup \mathcal{E}(\mathcal{OG}_i) \cup \mathcal{E}(\mathcal{OG}_j)$. Here, $\mathcal{E}_g$ are global edges in $\mathcal{OG}$ (connect nodes across low-level graphs $\mathcal{OG}_i, \mathcal{OG}_j$ but are not elements of such subgraphs) and $\mathcal{N}_g$ are global nodes in $\mathcal{OG}$ (can be connected to nodes in low-level graphs but that are not elements of such subgraphs). For every global edge $e \in \mathcal{E}_g$, we have that $\mathcal{N}(e) \in \mathcal{N}(\mathcal{OG}_i), \mathcal{N}(\mathcal{OG}_j)$, and $\mathcal{N}_g$ where $\mathcal{N}(\mathcal{OG}_i) \cap \mathcal{N}(\mathcal{OG}_j) \cap \mathcal{N}_g = \emptyset$. In other words, the edges $\mathcal{E}_g$ only connect nodes across low-level graphs $\mathcal{OG}_i$ and $\mathcal{OG}_j$ or between global nodes $\mathcal{N}_g$ in $\mathcal{OG}$ and low-level graphs. Consequently, if we

treat the elements of a node set $\mathcal{N}$ as a graph (i.e., we combine subgraphs into a single `OptiNode`), we can represent $\mathcal{OG}(\{\mathcal{OG}_i, \mathcal{OG}_j\}, \mathcal{E}_g, \mathcal{N}_g)$ as $\mathcal{OG}(\mathcal{N}, \mathcal{E})$. This nesting procedure can be carried over multiple levels to form a hierarchical graph.

The formulation given by Equation (2.1) can be extended to describe a hierarchical graph with an arbitrary number of subgraphs. This is shown in Equation (2.3) where we again denote that $\{x_n\}_{n \in \mathcal{N}(\mathcal{OG})}$ represents the collection of variables over the entire set of nodes in the `OptiGraph`. Equation (2.3a) denotes the sum of `OptiNode` objective functions for every node in the graph (including nodes in subgraphs), Equation (2.3b) captures the constraints over all of the `OptiNodes` in the graph, Equation (2.3c) represents the linking constraints over the edges for each subgraph (including nested subgraphs) and Equation (2.3d) captures the highest level global linking constraints. Here, we use the notation $\mathcal{SG} \in AG(\mathcal{OG})$ to indicate that the subgraph elements $\mathcal{SG}$ are part of the recursive set of subgraphs in $\mathcal{OG}$ (i.e., the set contains all subgraphs in the graph). Specifically, we define the mapping $AG : \mathcal{OG} \rightarrow \{\mathcal{SG}_1, \mathcal{SG}_2, \mathcal{SG}_3, ..., \mathcal{SG}_N\}$ where N is the total number of subgraphs in $\mathcal{OG}$.

$$\min_{\{x_n\}_{n \in \mathcal{N}(\mathcal{OG})}} \quad \sum_{n \in \mathcal{N}_g(\mathcal{OG})} f_n(x_n) + \sum_{\mathcal{SG} \in AG(\mathcal{OG})} \sum_{n \in \mathcal{N}(\mathcal{SG})} f_n(x_n) \tag{2.3a}$$

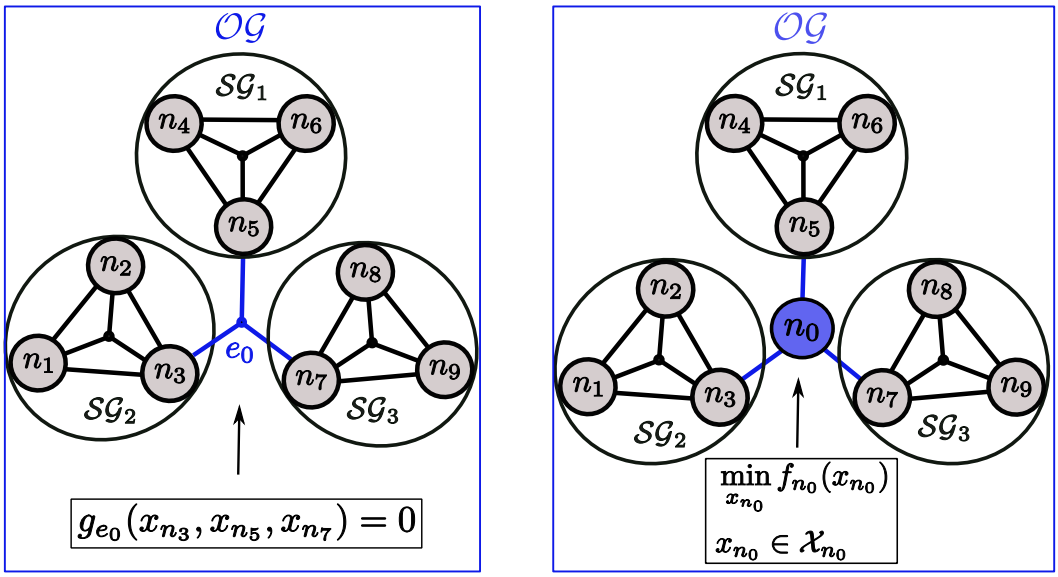
$$\text{s.t. } x_n \in \mathcal{X}_n, \; n \in \mathcal{N}_g(\mathcal{OG}) \cup \{\mathcal{N}(\mathcal{SG}), \mathcal{SG} \in AG(\mathcal{OG})\} \quad \text{(Node Constraints)} \tag{2.3b}$$

$$g_e(\{x_n\}_{n \in \mathcal{N}(e)}), \; e \in \mathcal{E}(\mathcal{SG}), \mathcal{SG} \in AG(\mathcal{OG}) \quad \text{(Subgraph Links)} \tag{2.3c}$$

$$g_e(\{x_n\}_{n \in \mathcal{N}(e)}), \; e \in \mathcal{E}_g(\mathcal{OG}). \quad \text{(Graph Links)} \tag{2.3d}$$

Figure 2.3 depicts examples of graphs with subgraph nodes. In Figure 2.3a we have a graph $\mathcal{OG}$ that contains three subgraphs $\mathcal{SG}_1, \mathcal{SG}_2, \mathcal{SG}_3$ with a total of nine nodes (three in each subgraph). The graph $\mathcal{OG}$ contains a global `OptiEdge` that connects to local nodes in the subgraphs. Figure 2.3b also shows a hierarchical graph $\mathcal{OG}$ with three subgraphs $\mathcal{SG}_1, \mathcal{SG}_2, \mathcal{SG}_3$ and nine total nodes. This graph $\mathcal{OG}$ contains a global node n_0 that is connected to nodes in the subgraphs. This type of structure arises when there is a parent

(master) optimization problem that is connected to children subproblems.



(a) An `OptiGraph` with a global `OptiEdge` (b) An `OptiGraph` with a global `OptiNode`

Figure 2.3: Example hierarchical `OptiGraphs`. In the left figure, the subgraphs are coupled through the global edge e_0. In the right figure the subgraphs are coupled to the global node n_0.

2.3 Software Framework: Modeling with Plasmo.jl

We now find it informative to introduce `Plasmo.jl`, a graph-based modeling package developed in the `Julia` language. `Plasmo.jl` implements the graph models described by (2.1) and (2.3) and illustrated in Figures 2.1 and 2.3. This section covers basic syntax and shows how to modify `OptiGraph` models and create hierarchical models.

2.3.1 *Basic Syntax*

In the `Plasmo.jl` implementation, an `OptiNode` encapsulates a `Model` object from the `JuMP.jl` modeling language. This harnesses the algebraic modeling syntax and processing functionality of `JuMP.jl`. An `OptiEdge` object encapsulates the linking constraints that define coupling between the nodes.

Example 1 : Creating an OptiGraph

We start with the simple example given by (2.4) to demonstrate `OptiGraph` syntax:

$$\begin{aligned}
\min \quad & y_{n_1} + y_{n_2} + y_{n_3} && \text{(Objective)} && \text{(2.4a)}\\
\text{s.t.} \quad & x_{n_1} \geq 0, y_{n_1} \geq 2, x_{n_1} + y_{n_1} \geq 3 && \text{(Node 1 Constraints)} && \text{(2.4b)}\\
& x_{n_2} \geq 0, x_{n_2} + y_{n_2} \geq 3 && \text{(Node 2 Constraints)} && \text{(2.4c)}\\
& x_{n_3} \geq 0, x_{n_3} + y_{n_3} \geq 3 && \text{(Node 3 Constraints)} && \text{(2.4d)}\\
& x_{n_1} + x_{n_2} + x_{n_3} = 5 && \text{(Link Constraint)} && \text{(2.4e)}
\end{aligned}$$

In this model, equations (2.4b), (2.4c), and (2.4d) represent individual node constraints and (2.4e) is a linking constraint that couples the three nodes. We formulate and solve this optimization model as shown in Snippet 2.1. We import `Plasmo.jl` into a `Julia` session on Line 1 as well as the off-the-shelf linear programming solver `GLPK` [87] to solve the problem. We define `graph1` (an `OptiGraph`) on Line 5 and then create three `OptiNodes` on Lines 8-24 using the `@optinode` macro. We also use the `@variable`, `@constraint`, and `@objective` macros (extended from `JuMP.jl`) to define node model attributes. Next, we use the `@linkconstraint` macro on Line 27 to create a linking constraint between the three nodes. Importantly, this *automatically* creates an underlying `OptiEdge`. This feature is key, as the user does not have to express the topology of the graph (it is automatically created as linking constraints are added). In other words, the user does not need to provide an adjacency matrix (which can be highly complex). We lastly solve the problem using the `GLPK` optimizer on Line 30. Since `GLPK` does not exploit graph structure, `Plasmo.jl` automatically transforms the graph into a standard linear programming format. We query the solution for each variable using the `value` function on Line 33 which accepts the corresponding node and variable we wish to query. Another important feature is that the solution data retains the structure of the `OptiGraph`, and this facilitates query and analysis. We note that the syntax presented here is similar to that used in `JuMP.jl` but

operates at a higher level of abstraction. The structure of an `OptiGraph` can be visualized using functions extended from `Plots.jl`. We layout the `OptiGraph` topology on Line 39 using the `plot` function and we plot the underlying adjacency matrix structure on Line 44 using the `spy` function. Both of these functions can accept keyword arguments to customize their appearance. The matrix visualization also encodes information on the number of variables and constraints in each node and edge. The results are depicted in Figure 2.4; the left figure shows a standard graph visualization which draws an edge between each pair of nodes if they share an `OptiEdge`, and the right figure shows the matrix representation where labeled blocks correspond to nodes and blue marks represent linking constraints that connect their variables. The node layout helps visualize the overall connectivity of the graph while the matrix layout helps visualize the size of nodes and edges.

Code Snippet 2.1: Creating and Solving an `OptiGraph` in Plasmo.jl

```
using Plasmo
using GLPK
using Plots

graph1 = OptiGraph()

#Create three OptiNodes
@optinode(graph1,n1)
@variable(n1, y >= 2)
@variable(n1,x >= 0)
@constraint(n1,x + y >= 3)
@objective(n1, Min, y)

@optinode(graph1,n2)
@variable(n2, y)
@variable(n2,x >= 0)
@constraint(n2,x + y >= 3)
@objective(n2, Min, y)

@optinode(graph1,n3)
@variable(n3, y )
@variable(n3,x >= 0)
@constraint(n3,x + y >= 3)
@objective(n3, Min, y)

#Create link constraint between nodes (automatically creates an optiedge on graph1)
@linkconstraint(graph1, n1[:x] + n2[:x] + n3[:x] == 5 )

#Optimize with GLPK
optimize!(graph1,GLPK.Optimizer)

#Query Solution
value(n1,n1[:x])
value(n2,n2[:x])
value(n3,n3[:x])
objective_value(graph1)

#Visualize graph topology
plt_graph1 = plt_graph1 = Plots.plot(graph1,node_labels = true,
markersize = 60,labelsize = 30, linewidth = 4,
layout_options = Dict(:tol => 0.01,:iterations => 2));

#Visualize graph adjacency
plt_matrix1 = Plots.spy(graph1,node_labels = true,markersize = 30);
```

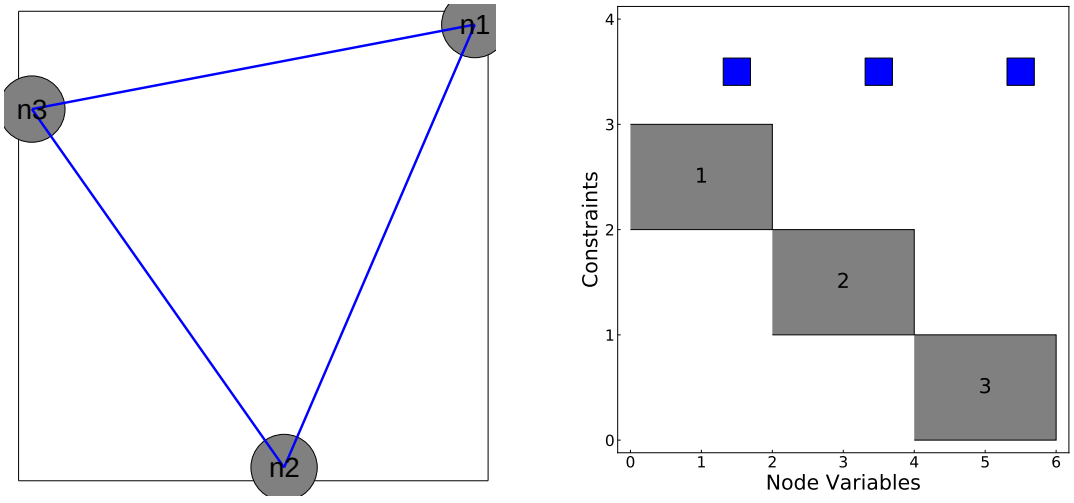


Figure 2.4: Output visuals for Code Snippet 2.1. Graph topology obtained with `plot` function (left) and graph matrix representation obtained with `spy` function (right).

Example 2 : Modifying an OptiGraph

The `OptiGraph` created in Snippet 2.1 can be modified in a modular fashion. For example, in Code Snippet 2.2 we now load in the `Ipopt` optimizer on Line 2 and we swap out node `n1` with a new node `new_n1` that contains a nonlinear constraint on Lines 4 through 12. We also specifically warm-start the values on node `n2` (as opposed to on every node) on Lines 15 through 18 and we optimize the updated graph on Line 21.

Code Snippet 2.2: Modifying an OptiGraph

```
1  #Load Ipopt optimizer
2  using Ipopt
3
4  new_n1 = OptiNode()
5  @variable(new_n1, y >= 4)
6  @variable(new_n1, x >= 0)
7  @NLnodeconstraint(new_n1,exp(x + y) >= 3)
8  @objective(new_n1,Min,x+y)
9  @variable(new_n1, y >= 4)
10
11 #Swap out node n1 in graph1 with new_n1
12 swap_node!(graph1,n1,new_n1;preserve_links = true)
13
14 # Collect all variable values on n2
15 n2_values = value.(Ref(n2),all_variables(n2))
16
17 # Warm start the values on node n2
18 set_start_value.(all_variables(n2),n2_values)
19
20 #optimize the updated problem with Ipopt
21 optimize!(graph1,Ipopt.Optimizer)
```

2.3.2 *Hierarchical Modeling Syntax*

In this section, we show how to perform hierarchical modeling with an `OptiGraph` in `Plasmo.jl`.

Example 3: Hierarchical Graph with a Global Edge

A powerful feature of the `OptiGraph` is that it manages its own nodes and edges in a *self-contained* manner (without requiring references to other higher-level graphs). Consequently, we can define subgraphs independently (in a modular fashion) and these can be coupled together by using global edges or nodes defined in a high-level graph. A key benefit of this paradigm is that each node can have its own syntax (syntax does not need

to be consistent or non-redundant over the entire model). This is a fundamental difference compared to other algebraic modeling languages (where syntax has to be consistent and non-redundant over the entire model). The formulation in Example (2.5) illustrates how to express hierarchical connectivity using a global edge. Equation (2.5a) is the summation of every node objective function in the graph; (2.5b), (2.5c) and (2.5d) describe node constraints; (2.5e), (2.5f), and (2.5g) represent linking constraints within each subgraph; and (2.5h) defines a linking constraint at the higher level graph (that links nodes from each individual subgraph). Formulation (2.5) can be expressed as a hierarchical `OptiGraph` using the `add_subgraph!` function. This functionality is shown in Code Snippet 2.3, where we extend `graph1` from Code Snippet 2.1. We create a new graph called `graph2` on Line 2 and setup nodes and link them together on Lines 4 through 17. We also construct `graph3` in the same fashion on Lines 20 through 34. Next, we create `graph0` on Line 37 and add graphs `graph1`, `graph2`, and `graph3` as subgraphs to `graph0` on Line 39. We add a linking constraint to `graph0` that couples nodes on each subgraph on Line 41 and solve the graph on Line 43. We present the graph visualization in in Figure 2.5. Here we can see the hierarchical structure of the `OptiGraph` and the local and global coupling constraints. This structure is compatible with that shown in Figure 2.3a.

$$\min \sum_{i=1:9} y_{n_i} \qquad \text{(Node Objectives)} \qquad (2.5a)$$

$$\text{s.t. } x_{n_i} \geq 0, y_{n_i} \geq 2, x_{n_i} + y_{n_i} \geq 3, i \in \{1,2,3\} \qquad \text{(Subgraph 1 Constraints)} \qquad (2.5b)$$

$$x_{n_i} \geq 0, y_{n_i} \geq 2, x_{n_i} + y_{n_i} \geq 5, i \in \{4,5,6\} \qquad \text{(Subgraph 2 Constraints)} \qquad (2.5c)$$

$$x_{n_i} \geq 0, y_{n_i} \geq 2, x_{n_i} + y_{n_i} \geq 7, i \in \{7,8,9\} \qquad \text{(Subgraph 3 Constraints)} \qquad (2.5d)$$

$$x_{n_1} + x_{n_2} + x_{n_3} = 3 \qquad \text{(Subgraph 1 Link Constraint)} \qquad (2.5e)$$

$$x_{n_4} + x_{n_5} + x_{n_6} = 5 \qquad \text{(Subgraph 2 Link Constraint)} \qquad (2.5f)$$

$$x_{n_7} + x_{n_8} + x_{n_9} = 7 \qquad \text{(Subgraph 3 Link Constraint)} \qquad (2.5g)$$

$$x_{n_3} + x_{n_5} + x_{n_7} = 10 \qquad \text{(Global Link Constraint)} \qquad (2.5h)$$

Example 4: Hierarchical Graph with a Global Node

We can express hierarchical connectivity within a `OptiGraph` by defining a global node that is connected with subgraph nodes. Formulation (2.6) illustrates this idea; this is analogous to (2.5) where we have removed the high level linking constraint (2.5h) and have replaced it with a high level node (2.6h) and three linking constraints that couple the graph it to its subgraphs (2.6i). The implementation of formulation in (2.6) is shown in Code Snippet 2.4. Here, we assume that we already have `graph1`, `graph2`, and `graph3` defined from Snippets 2.1 and 2.3. We recreate `graph0` and setup the node `n0` on Lines 2 through 7. We add subgraphs `graph1`, `graph2`, and `graph3` on Lines 10 through 12 like in the previous snippet, and add linking constraints that connect node `n0` to nodes in each subgraph on Lines 15 through 17. We solve the newly created `graph0` on Line 20 and present the visualization in Figure 2.6. This structure is compatible with that shown in Figure 2.3b.

$$\min \sum_{i=1:9} y_{n_i} \qquad \text{(Objective)} \quad (2.6a)$$

$$\text{s.t. } x_{n_i} \geq 0, y_{n_i} \geq 2, x_{n_i} + y_{n_i} \geq 3, i \in \{1,2,3\} \qquad \text{(Subgraph 1 Constraints)} \quad (2.6b)$$

$$x_{n_i} \geq 0, y_{n_i} \geq 2, x_{n_i} + y_{n_i} \geq 5, i \in \{4,5,6\} \qquad \text{(Subgraph 2 Constraints)} \quad (2.6c)$$

$$x_{n_i} \geq 0, y_{n_i} \geq 2, x_{n_i} + y_{n_i} \geq 7, i \in \{7,8,9\} \qquad \text{(Subgraph 3 Constraints)} \quad (2.6d)$$

$$x_{n_1} + x_{n_2} + x_{n_3} = 3 \qquad \text{(Subgraph 1 Link Constraint)} \quad (2.6e)$$

$$x_{n_4} + x_{n_5} + x_{n_6} = 5 \qquad \text{(Subgraph 2 Link Constraint)} \quad (2.6f)$$

$$x_{n_7} + x_{n_8} + x_{n_9} = 7 \qquad \text{(Subgraph 3 Link Constraint)} \quad (2.6g)$$

$$x_{n_0} \geq 0 \qquad \text{(Graph Constraint)} \quad (2.6h)$$

$$x_{n_0} + x_{n_3} = 3, x_{n_0} + x_{n_5} = 5, x_{n_0} + x_{n_7} = 7 \qquad \text{(Global Link Constraints)} \quad (2.6i)$$

Code Snippet 2.3: Hierarchical Connectivity using Global Edge

```
#Create low−level graph2
graph2 = OptiGraph()

@optinode(graph2,n4)
@variable(n4, x >= 0); @variable(n4, y >= 2)
@constraint(n4,x + y >= 5); @objective(n4, Min, y)

@optinode(graph2,n5)
@variable(n5, x >= 0); @variable(n5, y >= 2)
@constraint(n5,x + y >= 5); @objective(n5, Min, y)

@optinode(graph2,n6)
@variable(n6, x >= 0); @variable(n6, y >= 2 )
@constraint(n6,x + y >= 5); @objective(n6, Min, y)

#Create graph2 link constraint
@linkconstraint(graph2, n4[:x] + n5[:x] + n6[:x] == 5)

#Create low−level graph 3
graph3 = OptiGraph()

@optinode(graph3,n7)
@variable(n7, x >= 0); @variable(n7, y >= 2)
@constraint(n7,x + y >= 7); @objective(n7, Min, y)

@optinode(graph3,n8)
@variable(n8, x >= 0); @variable(n8, y >= 2)
@constraint(n8,x + y >= 7); @objective(n8, Min, y)

@optinode(graph3,n9)
@variable(n9,x >= 0); @variable(n9, y >= 2)
@constraint(n9,x + y >= 7); @objective(n9, Min, y)
#Create graph3 link constraint
@linkconstraint(graph3, n7[:x] + n8[:x] + n9[:x] == 7)

#Create high−level graph0
graph0 = OptiGraph()
#Add subgraphs to graph0
add_subgraph!(graph0,graph1); add_subgraph!(graph0,graph2); add_subgraph!(graph0,graph3)

#Add link constraint to graph0 connecting its subgraphs
@linkconstraint(graph0,n3[:x] + n5[:x] + n7[:x] == 10)

optimize!(graph0,GLPK.Optimizer) #Optimize with GLPK
```

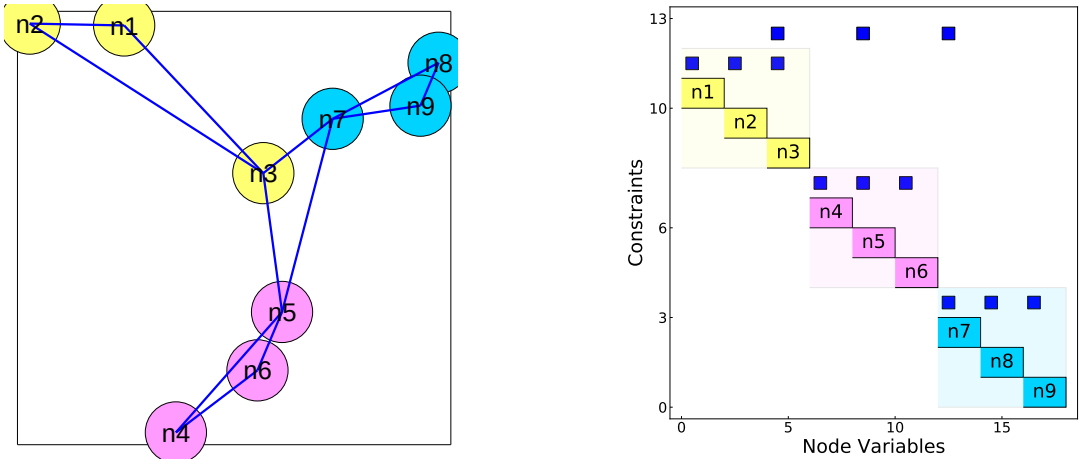


Figure 2.5: Output visuals for Code Snippet 2.3 showing hierarchical structure of an `OptiGraph` with three subgraphs connected by a global edge.

Code Snippet 2.4: Hierarchical Connectivity using Global Node

```
#Create graph0
graph0 = OptiGraph()

#Create a node on graph0
@optinode(graph0,n0)
@variable(n0,x)
@constraint(n0,x >= 0)

#Add subgraphs to graph0
add_subgraph!(graph0,graph1)
add_subgraph!(graph0,graph2)
add_subgraph!(graph0,graph3)

#Create link constraints on graph0 connecting it to its subgraphs
@linkconstraint(graph0,n0[:x] + n3[:x] == 3)
@linkconstraint(graph0,n0[:x] + n5[:x] == 5)
@linkconstraint(graph0,n0[:x] + n7[:x] == 7)

#Optimize with GLPK
optimize!(graph0,GLPK.Optimizer)
```

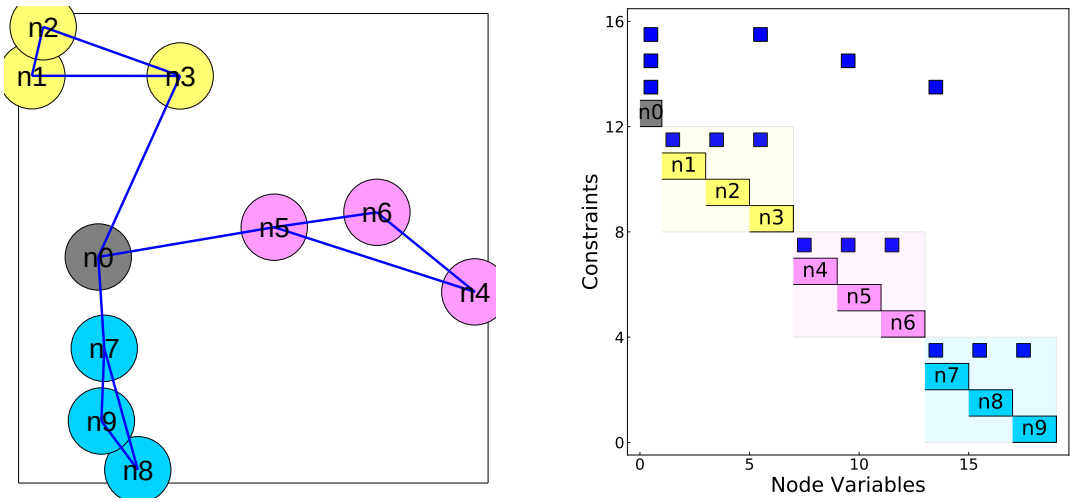


Figure 2.6: Output visuals for Code Snippet 2.4 showing hierarchical structure of an `OptiGraph` with three subgraphs connected by a global node.

Function	Description
OptiGraph()	Create a new OptiGraph object.
@optinode(g::OptiGraph,expr::Expr)	Create OptiNodes using Julia expression
@linkconstraint(graph::OptiGraph,expr::Expr)	Create linking constraint between nodes in g using expr
add_subgraph!(g::OptiGraph,sg::OptiGraph)	Add subgraph sg to g
getoptinodes(g::OptiGraph)	Retrieve local OptiNodes in g
getoptiedges(g::OptiGraph)	Retrieve local OptiEdges in g
getlinkconstraints(g::OptiGraph)	Retrieve linking constraints in g
getsubgraphs(g::OptiGraph)	Retrieve subgraphs in g
all_optinodes(g::OptiGraph)	Retrieve all OptiNodes in g (including subgraphs)
all_optiedges(g::OptiGraph)	Retrieve OptiEdges in g (including subgraphs)
all_linkconstraints(g::OptiGraph)	Retrieve all linking constraints in g (including subgraphs)
all_subgraphs(g::OptiGraph)	Retrieve all subgraphs in g (including subgraphs)
OptiNode()	Create a new OptiNode, but do not add it to a graph
add_node!(g::OptiGraph,node::OptiNode)	Add the node to g
swap_node!(g::OptiGraph,n1,n2)	Swap out n1 with n2 in g
swap_graph!(g::OptiGraph,sg1,sg2)	Swap out subgraph sg1 with sg2 in g

Table 2.1: Overview of OptiGraph construction and management functions in Plasmo.jl.

2.3.3 *Overview of Modeling Functions*

In addition to the graph construction functions presented in the previous examples (@optinode, @linkconstraint, add_subgraph!), it is also possible to query an OptiGraph object to retrieve its attributes. Table 2.1 summarizes the main Plasmo.jl functions used to create and inspect an OptiGraph. We inspect the nodes, edges, linking constraints, and subgraphs using getoptinodes, getoptiedges, getlinkconstraints, and getsubgraphs functions, and we can collect *all* of the corresponding graph attributes using recursive versions of these functions (all_nodes, all_edges, all_linkconstraints and all_subgraphs). It is also possible to modify and swap out individual nodes (or subgraphs) with OptiNode(), add_node!, swap_node!, and swap_graph! functions.

2.4 Case Study: State Estimation in a Natural Gas Network

The OptiGraph modeling capabilities are best expressed with a realistic case study. We conclude this chapter with a detailed case study that demonstrates how data and model management capabilities in the OptiGraph facilitate formulating a challenging estimation problem for a natural gas pipeline system. Throughout this section we refer to the equa-

tions that describe the pipeline system (e.g. dynamic flows and equipment models) in a general sense and the detailed model equations we refer to are described in Appendix A. We also present the corresponding `Plasmo.jl` model implementation that shows the hierarchical construction of the model.

This case study examines the problem of state estimation (also called data assimilation in some fields) in a natural gas network using the `OptiGraph`. Our goal is to model a gas pipeline network and develop a state estimation scheme that estimates the internal transient pipeline pressure and flow profiles using limited pressure sensor information available at pipeline and compressor junctions.

2.4.1 *Problem Overview*

Figure 2.7 depicts the modeled system and includes some of the notation we use to describe the gas network. The network consists of sets of junctions $j \in \mathcal{J}$ and links $\ell \in \mathcal{L}$. Each junction j may contain sets of gas supplies $s \in \mathcal{S}_j$ and/or gas demands $d \in \mathcal{D}_j$. The links can be divided into sets of pipeline links $\ell \in \mathcal{L}_p$ and compressor links $\ell \in \mathcal{L}_c$ such that the complete set of links is given by $\mathcal{L} := \mathcal{L}_p \cup \mathcal{L}_c$. The sketch in Figure 2.7 consists of a straight line of 13 pipelines with 11 compressors resulting in 25 total junctions (this setup is called a gunbarrel in gas network operations). Junction j_1 includes a single gas supply (gas injection) and junction j_{25} contains a single gas demand (gas withdrawal). Pressure at each junction at each time period t is given by $\theta_{j,t}$ and the flows and pressures we seek to estimate at time t are given by $\{p_{\ell,t,k}\}_{\ell \in \mathcal{L}_p, k \in \mathcal{X}}$ and $\{f_{\ell,t,k}\}_{\ell \in \mathcal{L}_p, k \in \mathcal{X}}$ respectively where $\mathcal{X}$ is a set of spatial points defined along each pipeline.

The proposed problem incorporates detailed physical models (which are highly nonlinear PDEs) and complex constraint sets. The estimation problem we develop is inherently ill-posed due to the infinite-dimensional nature of the states (flows and pressures along the length of the pipelines) which leads to instability of the proposed estimator. To circumvent this issue, we employ a moving horizon strategy that utilizes initial prior

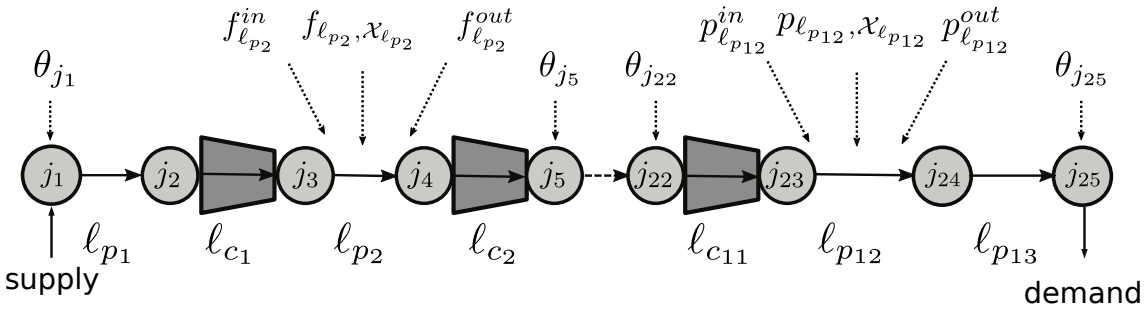


Figure 2.7: Multi-pipeline system used for state estimation problem.

information using an easily-computable steady-state. Using the `OptiGraph`, we intend to model the gas network and use its data management features to implement the estimator algorithm using prior information and sensor data. We also derive approximate transport models and show that the `OptiGraph` can trade out different model components to analyze trade-offs between estimation accuracy and computational performance.

In the state estimation setting for gas networks, typical sensor measurements include the pressures at nodal junctions (compressor suction and discharge are usually measured) and the flows directed in and out of supplies and demands (purchase and sell prices are calculated using these flow measurements) [45]. In this study, we assume that only pressure measurements are available and we demonstrate that such information is sufficient to infer the internal state of the pipelines. For simplicity of implementation we also assume sensor measurement noise is sufficiently filtered, but applying such output filtering in an optimization-based estimator is straightforward [137]. We consider sensor measurements distributed over a historical (finite) time horizon consisting of N sampling times of equal length $\delta = t_i - t_{i-1}$. At *sampling time* t_i, we define the *past time window* $\mathcal{T}_i = \{t_{i-N}, t_{i-N+1}, ..., t_i\}$. For convenience, we also define the set $\bar{\mathcal{T}}_i = \mathcal{T}_i \setminus \{t_i\}$. Using a history of measurements over $\mathcal{T}_i$ we seek to estimate the *current* state at time t_i. Specifically, at sampling time t_i, we estimate the current state using the pressure sensor history $\theta^m_{\mathcal{J},\mathcal{T}_i}$. In addition, we assume that measurements for the control (input) variables are available such as the compression ratios $\eta^m_{\mathcal{L}_c,\mathcal{T}_i}$ (these can also be inferred from suction and discharge pressures). We use $\mathcal{I}^m_{\mathcal{T}_i}$ to denote all the sensor information over time window

$\mathcal{T}_i$. With this information we estimate the dynamic states ($p^e_{\mathcal{L}_c,\mathcal{T}_i,\mathcal{X}}$, $f^e_{\mathcal{L}_c,\mathcal{T}_i,\mathcal{X}}$) as well as the algebraic states ($s^e_{\mathcal{S},\mathcal{T}_i}$ and $d^e_{\mathcal{D},\mathcal{T}_i}$). We use $\mathcal{I}^e_{\mathcal{T}_i} = (p^e_{\mathcal{L}_c,\mathcal{T}_i,\mathcal{X}}, f^e_{\mathcal{L}_c,\mathcal{T}_i,\mathcal{X}}, s^e_{\mathcal{S},\mathcal{T}_i}, d^e_{\mathcal{D},\mathcal{T}_i})$ to denote the estimated state trajectory. Table 2.2 summarizes some of the relevant terms for the state estimation problem.

Table 2.2: Estimator Parameters and Variables

Parameter	Description
N	Number of horizon points
i	Interval number
t_i	Sample time
$\mathcal{T}_i$	Set of times $\{t_{i-N}...t_i\}$
$\theta^m_{\mathcal{J},\mathcal{T}_i}$	Measured junction pressure history at t_i
$\overline{p}_{\mathcal{L}_p,\mathcal{X}}$	Pressure profile prior
$\overline{f}_{\mathcal{L}_p,\mathcal{X}}$	Flow profile prior
$p^e_{\mathcal{L}_p,\mathcal{X},\mathcal{T}_i}$	Estimated pressure history
$f^e_{\mathcal{L}_p,\mathcal{X},\mathcal{T}_i}$	Estimated flow history

We use a spatiotemporal discretization scheme to cast the developed estimation problem as a large-scale and sparse nonlinear programming problem (NLP). The states are estimated at sampling time t_i by solving the following optimization (estimation) problem:

$$\min \quad \sum_{\ell\in\mathcal{L}_p}\sum_{k\in\mathcal{X}_\ell}\left((p_{\ell,t_{i-N},k}-\overline{p}_{\ell,k})^2+(f_{\ell,t_{i-N},k}-\overline{f}_{\ell,k})^2\right)+\sum_{t\in\mathcal{T}_i}\sum_{j\in\mathcal{J}}(\theta_{j,t}-\theta^m_{j,t})^2 \tag{2.7a}$$

$$\text{s.t.} \quad \text{Junction Limits} \quad (A.1) \tag{2.7b}$$

$$\text{Compressor Equations} \quad (A.3) \tag{2.7c}$$

$$\text{Pipeline Equations} \quad (A.21) \tag{2.7d}$$

$$\text{Boundary Conditions} \quad (A.29) \tag{2.7e}$$

$$\text{Junction Conservation} \quad (A.29) \tag{2.7f}$$

The state estimation problem (2.7) is solved at each sample time t_i according the al-

gorithm given by 2.1. In the algorithm, we initialize the estimator with a prior guess of the dynamic flow and pressure states and a window (horizon) of the most recent system measurements. At each time t_i we obtain sensor measurements (for each junction), solve the estimator problem, and carry the estimate and solution data forward to initialize the next prior and warm-start the `Ipopt` solver. We study three distinct prior initialization strategies.

- (I) **No prior**: No prior initialization is used. We drop the first term in (2.7a).
- (II) **No initial prior**: The first term in (2.7a) is only dropped in the first sampling time.
- (III) **Steady-state prior**: A steady-state solution is computed using the first measurement and used as a prior in (2.7a).

Algorithm 2.1 state estimation

Given N, prior information $\bar{\mathcal{I}}$, and warm-start $\hat{\mathcal{I}}$.

Set sampling time $i \leftarrow 0$ and time window $\mathcal{T}_i \leftarrow \{t_{i-N} : t_i\}$.

loop

Obtain sensor information $\mathcal{I}^m_{\mathcal{T}_i}$ from system.

Solve estimator problem (2.7) $\mathcal{I}^e_{\mathcal{T}_i} \leftarrow \mathcal{OG}(\mathcal{I}^m_{\mathcal{T}_i}, \bar{\mathcal{I}}, \hat{\mathcal{I}})$.

Update warm-start information $\hat{\mathcal{I}} \leftarrow \mathcal{I}^e_{\mathcal{T}_i}$.

Update sampling time $i \leftarrow i + 1$ and time window $\mathcal{T}_i \leftarrow \{t_{i-N} : t_i\}$.

Update prior information $\bar{\mathcal{I}} \leftarrow \mathcal{I}^e_{t_{i-N}}$.

end loop

2.4.2 *OptiGraph Modeling Approach*

Developing the gas network model for the estimator in challenging in its own right because of the need to incorporate complex sets of equations. Moreover, our state estimation problem requires the repetitive solution of an ill-posed optimization problem (due to the

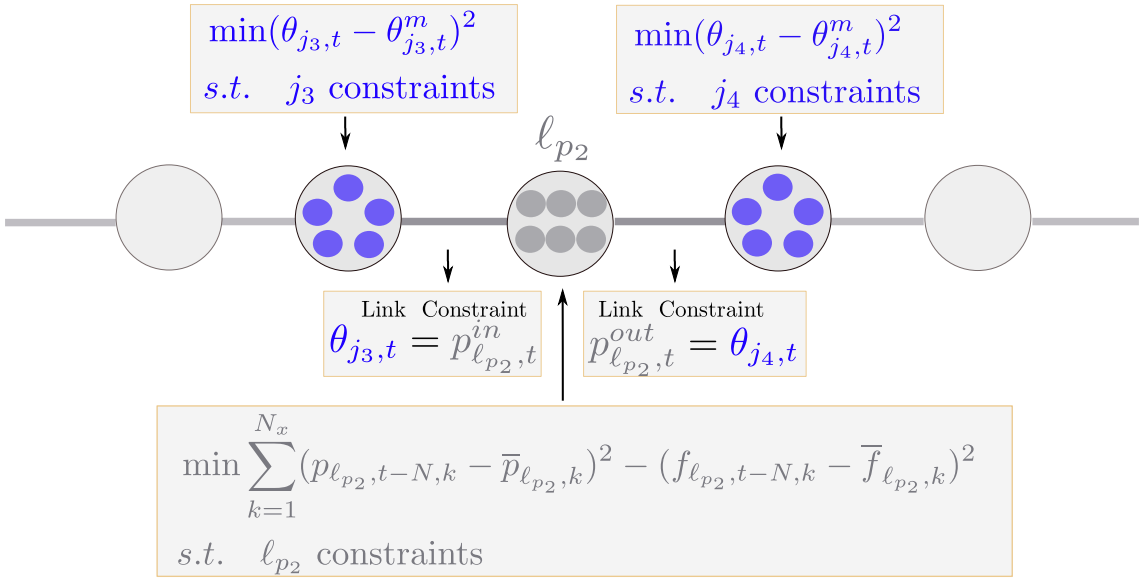


Figure 2.8: Modeling the multi-pipeline system with an `OptiGraph`.

infinite dimensional state) which creates a complex computational workflow where input data and solution data need to be dynamically updated. To facilitate the implementation of our estimator, we use the `OptiGraph` and `Plasmo.jl` which allows us to express the complex model construction and data workflow. In particular, `Plasmo.jl` enables the creation of component models that are independent of the graph structure of the problem (e.g., the gas network topology). For instance, pipeline segments can be represented as individual *subgraphs* containing the transport equations. Particularly, each pipeline can be represented as an `OptiGraph` wherein the contained `OptiNodes` capture the pipeline state variables (pressures and flows at each point on a space-time grid). Similarly, junction models are represented by subgraphs with associated `OptiNodes` containing the junction states and operating limits. Figure 2.8 more clearly demonstrates how the `OptiGraph` captures the structure of our estimation problem. In this figure, subgraphs are used to represent two junctions and a pipeline (j_3,j_4, and ℓ_{p_2}) where the `OptiEdges` capture the linking constraints that couple these models (in this case we only show the pipeline boundary conditions). The modular modeling abstraction also facilitates *data management* because the input and output data of the component models remains modularized. In

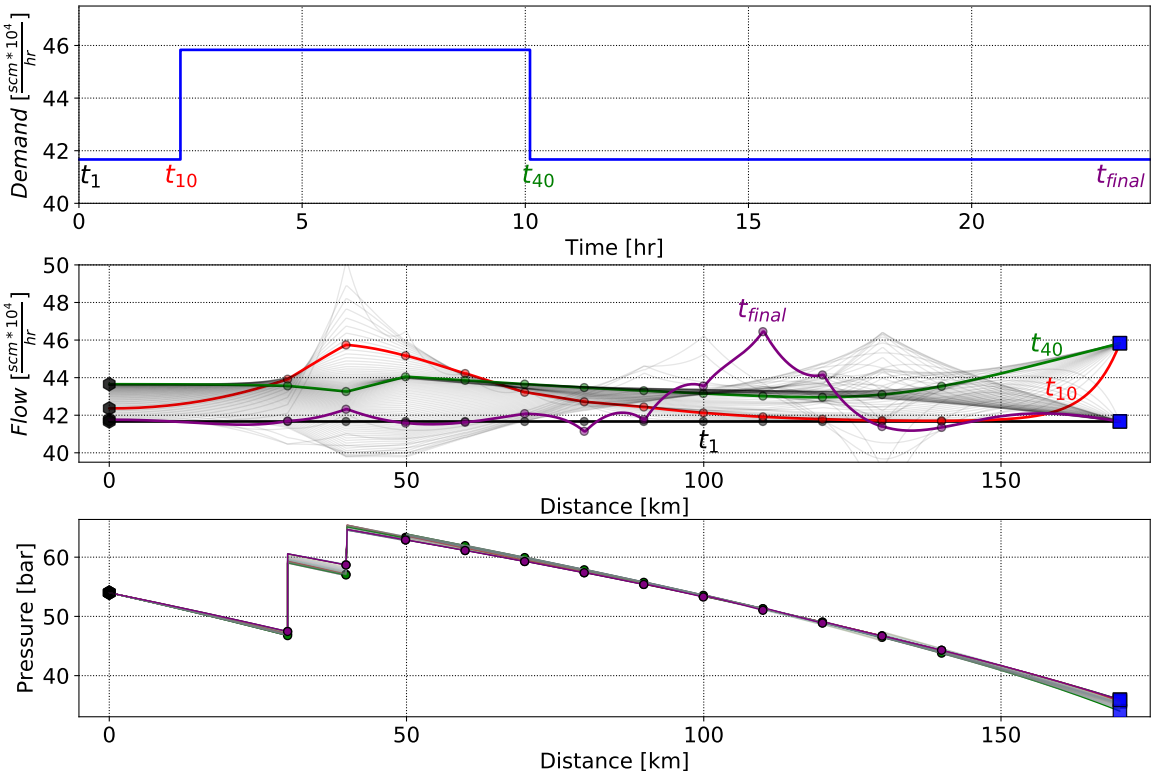


Figure 2.9: Simulated transient. Demand step (top), flow profiles (middle), and pressure profiles (bottom).

our implementation we also use the `OptiGraph` data aspects to facilitate *warm-starting* subsequent solutions of the estimation problem.

Figure 2.9 summarizes some simulation data we employ for the case study. The network is at steady-state at t_1 (given by a constant flow profile) and experiences a sudden gas withdrawal at the demand junction j_{25} at time t_{10} that induces a complex dynamic transient that triggers a forward propagating wave. At time t_{40}, the demand returns to its original state, which results in a backward propagating wave. Towards the end of the time horizon, the system is pressurized to return the total line-pack to its original value. This produces the final dynamic flow profile at t_{96} (t_{final}).

Figure 2.10 depicts how we dynamically update the `OptiGraph` model to implement the state estimation algorithm according to strategy (III). Every time we solve the `OptiGraph` estimation problem at time t_i, we use the obtained estimate $\mathcal{I}^e_{t_i-N}$ to generate a *prior* in

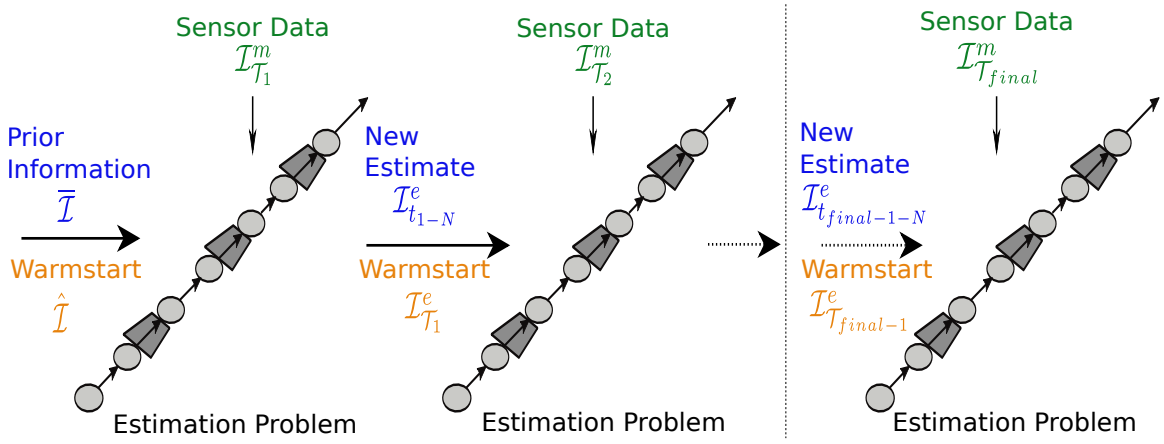


Figure 2.10: State estimation setup for multi-pipeline system.

(2.7) for the next problem at time t_{i+1}, and we use the solution $\mathcal{I}^e_{\mathcal{T}_i}$ as a warm-start. Code Snippet 2.5 details how this can be achieved in `Plasmo.jl` for strategy (III)). Line 1 loads `Plasmo.jl` and `Ipopt` (which we use to solve the estimation problem). Lines 3 through 5 load files which define network and measurement data, as well as functions we use to construct and manage gas network `OptiGraphs`. Line 8 uses `network_data` to construct the gas network `OptiGraph` using the function in Snippet A.4 (defined in Appendix A.5.2) and Line 12 sets the optimizer. Lines 15 through 18 calculate a steady-state solution using the first interval of measurement data to use as a prior and Lines 22 through 36 execute the estimation algorithm. Within the loop, Lines 34 and 35 perform the warm-start procedure.

Code Snippet 2.5: State estimation implementation for gas network in `Plasmo.jl`

```
using Plasmo, Ipopt

include("network_data.jl") # defines network_data
include("measurement_data.jl") # defines measurement_data
include("optigraph_functions.jl") # defines "set_ss_equations!" and "
     update_estimation_problem!" functions

# Create the gas network using Code Snippet
network = create_gas_network(network_data) A.4
pipelines = network[:pipelines]

# Set Ipopt optimizer for the network
set_optimizer(network,Ipopt.Optimizer)

# Solve steady state problem to obtain prior information
set_ss_equations!(network,measurement_data[1])
optimize!(network)
p_prior = [value.(Ref(pipeline),pipeline[:px]) for pipeline in pipelines]
f_prior = [value.(Ref(pipeline),pipeline[:fx]) for pipeline in pipelines]

N = 10 #define horizon length
# Begin estimation loop
for j = 1:T # loop over sampling times
    # create estimation problem
    update_estimation_problem!(network,measurement_data[j],p_prior,f_prior,N)

    # solve the estimation problem
    optimize!(network)

    # update the prior terms with the latest estimate
    p_prior = [value.(Ref(pipeline),pipeline[:px])[2]) for pipeline in pipelines]
    f_prior = [value.(Ref(pipeline),pipeline[:fx])[2]) for pipeline in pipelines]

    # update warm-start information
    result_values = value.(Ref(network),all_variables(network))
    set_start_value.(all_variables(network),result_values)
end
```

2.4.3 *State Estimation Results*

The results of our study show that managing prior information is key to the estimator convergence. Remarkably, using the correct prior initialization leads to convergence for every studied horizon length (even for a short horizon of $N = 2$). Figure 2.11 presents the estimation errors with no prior information (I), with no initial prior information (II), and with an initial steady-state prior (III) for $N = 2$. We recall that in the case of (I) , the prior term in (2.7a) is dropped at all sampling times while in the case of (II), it is only dropped in the first sampling time (we assume that no information is initially available but this is updated at later sampling times using the state prediction).

It is clear that the error levels of (I) are drastically improved by using prior informa-

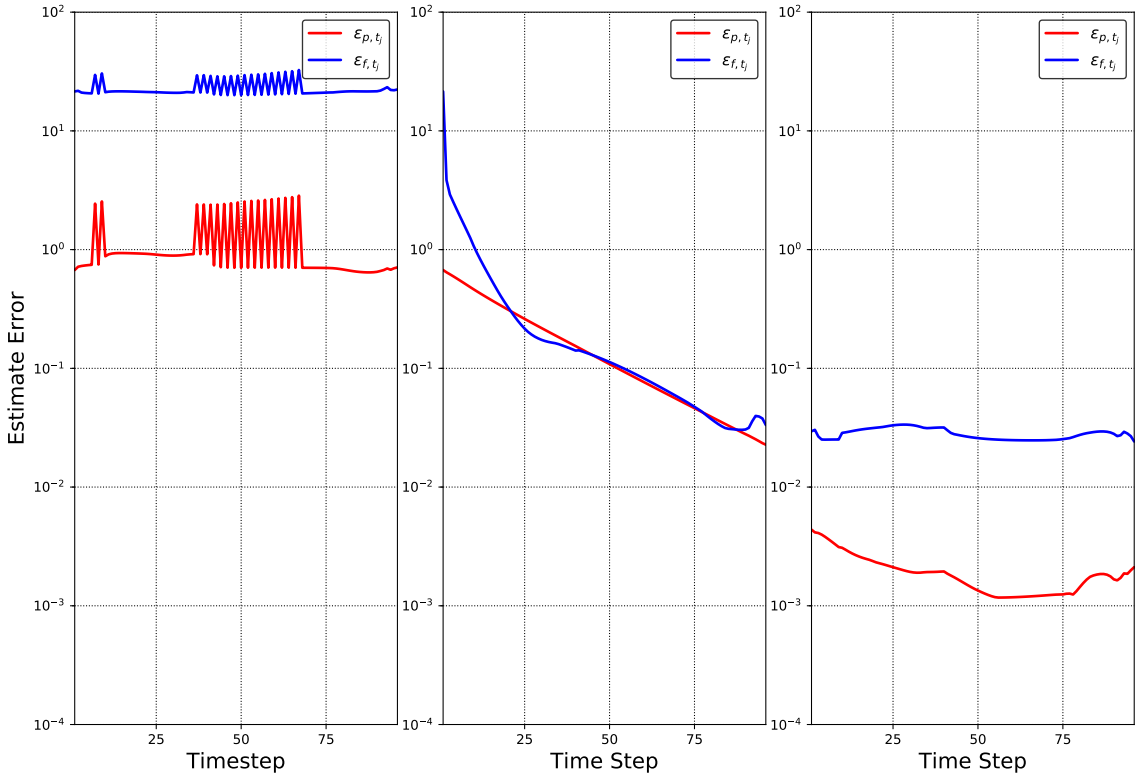


Figure 2.11: Estimation error with $N = 2$ for no prior information (I) (left), no initial prior information (II) (middle), and initial steady-state prior (III) info (right).

tion (II and III) and that the (III) provides the best performance. This highlights that, if the pipeline system is at rest at the beginning of the horizon, using an initial steady-state prior is sufficient to track the state during the dynamic transient. This result has important practical implications because most real pipeline systems are indeed returned to rest conditions nearly every day. Interestingly, even if no initial prior is used, the estimator is able to track the states and the errors decrease to the same levels of (III) for the flow profiles. The error level for the pressure profiles do not decrease to the same levels but remain fairly small. Under approach (II) however, we run the risk that the estimation problem cannot be solved due to ill-posedness. Consequently, we conclude that the steady-state approach provides a more reliable way to initialize the prior.

Figure 2.12 compares the flow profile from the original simulation with the estimated profile using (II) and (III) for $N = 2$. Notice the initial error profile of (II) corresponds

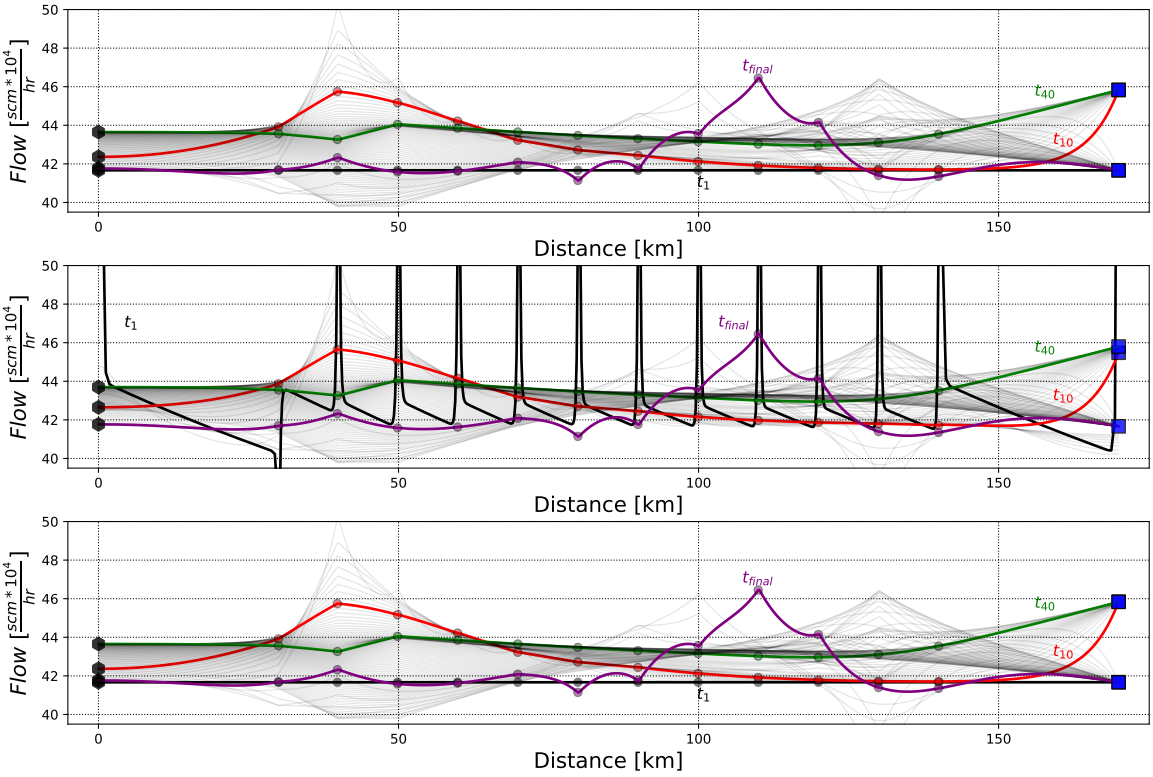


Figure 2.12: Comparison of simulated flow profile (top) and reconstructed state with $N = 2$ using no-initial-prior (II) (middle) and the steady-state prior (III) (bottom)

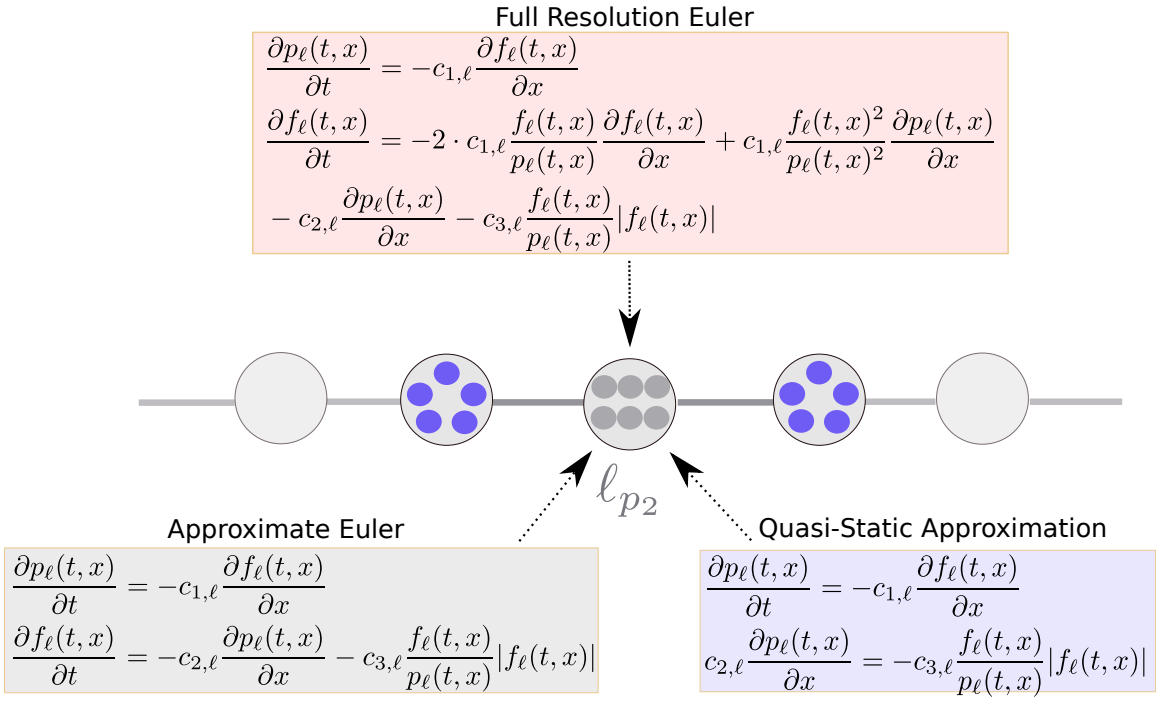


Figure 2.13: Swapping out models in the multi-pipeline system.

with its estimate error in Figure 2.11. While there is considerable error at t_1 (the black line) the error converges by time t_{10} (the red line). In contrast, the flow profiles corresponding to the steady-state prior are visually indistinguishable with the true values.

2.4.4 *Model Reduction*

With both accuracy and computational considerations, we also quantify the effect of using approximate models in the estimator formulation with the steady-state prior (III). Performing such a study is facilitated using the model swapping capabilities of the `OptiGraph` as shown in Figure 2.13. Given a fixed network topology, one can create models of different resolution by simply swapping component models (e.g., the Euler equations (A.9) can be exchanged with the approximate equations (A.14) or quasi-static approximations (A.24). Here, we show that the system modeled according to Figure 2.13 can be easily updated to experiment with different model formulations for the pipelines.

2.4.5 *Model Reduction Results*

Figure 2.14 compares the estimation error for the different approximations relative to the true state of the full-resolution Euler equations (A.9). In particular, we consider the approximate Euler (A.14) model and the quasi-static approximation (A.24) with $N = 10$. As can be seen, the approximate Euler achieves reasonably accurate estimates and the quasi-static approximation shows significant deviations towards the initial and final sampling times. This is likely because the quasi-static approximation neglects the advection term $\partial_t(\rho v)$ in the momentum balance, which is in fact significant during strong transients.

We also compare computational times at select sampling times (time steps) for the three different model approximations in Figure 2.15. The approximate Euler reduces the computational time by one half whereas the quasi-static time improvements over the approximate Euler are marginal. This figure also demonstrates the computational speedups using warm-starting where on average, the warm-started solution is about 10 times faster than the first time step solution (at $t = 1$). We conclude that such model reduction and warm-starting techniques are key to creating robust estimators, and this is facilitated by using the `OptiGraph` framework.

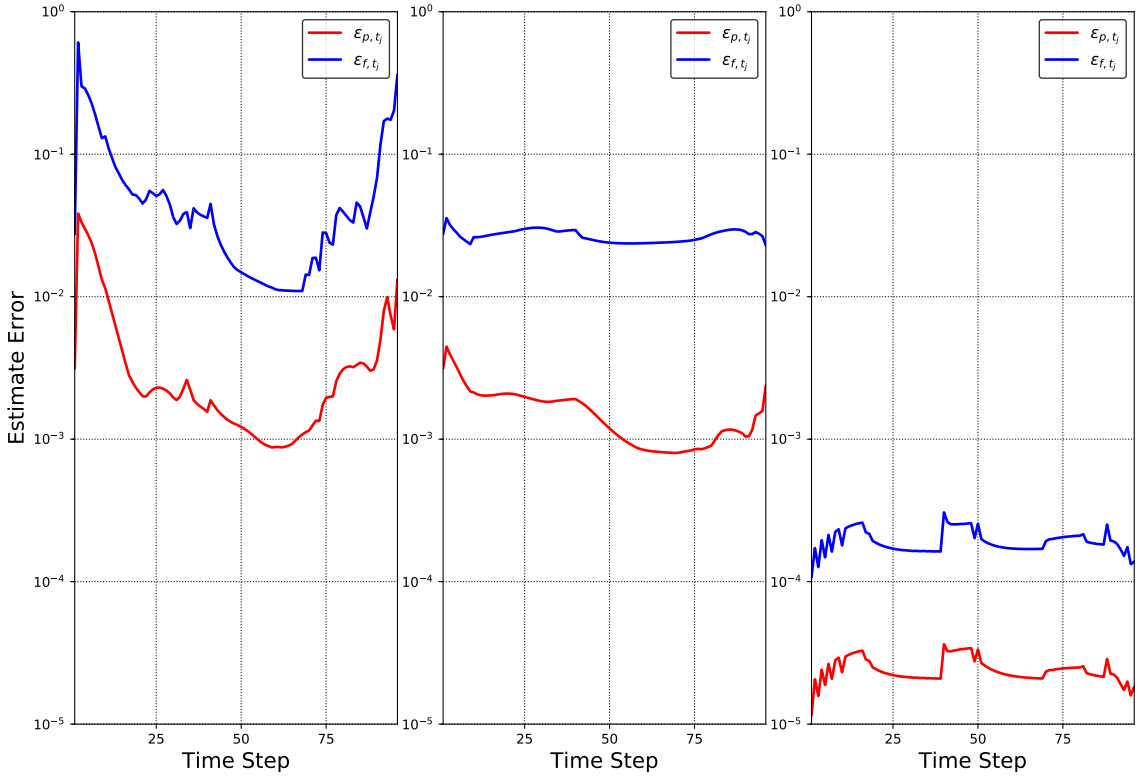


Figure 2.14: Error profiles for estimator with different model approximations for $N = 10$. Quasi-static (left), Approximate Euler (middle), and Full Euler (right).

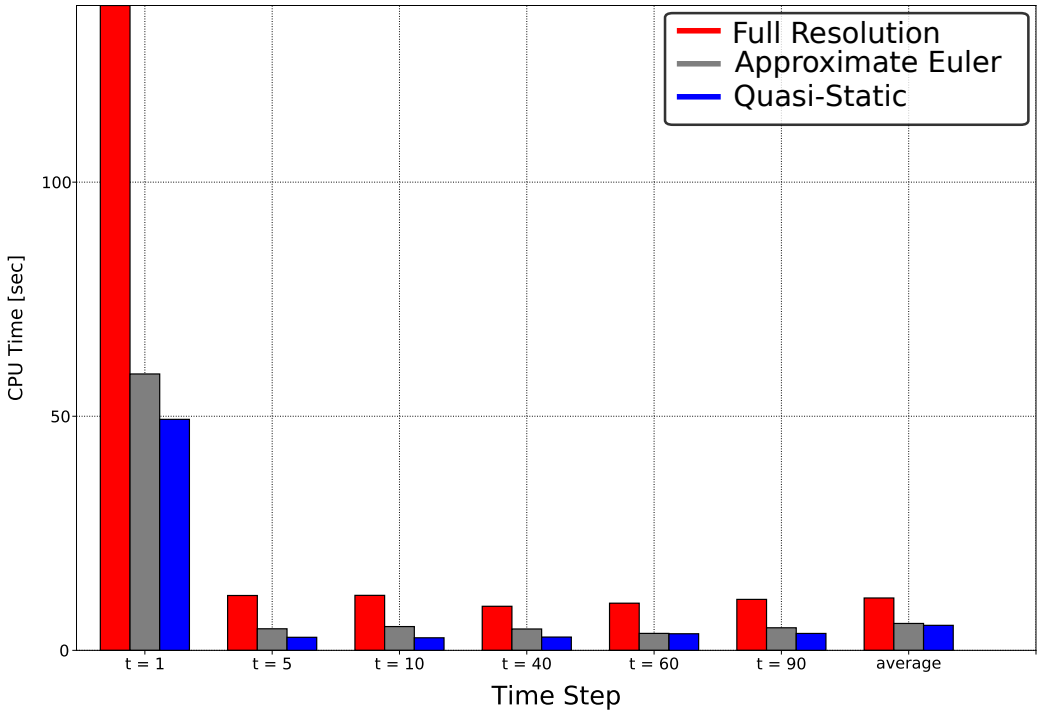


Figure 2.15: Computational performance for model approximations.

Chapter 3

DECOMPOSING OPTIMIZATION PROBLEMS

In Chapter 2 we introduced the `OptiGraph` and showcased how it can be used to model hierarchical optimization structures and perform helpful data operations. This chapter expands on the `OptiGraph` paradigm and shows how such a graph-based abstraction naturally exploits graph analysis and partitioning capabilities to *d*ecompose optimization problems and harness distributed computing architectures and decomposition algorithms.

3.1 Introduction

Advances in decomposition algorithms and computing architectures continue to expand the complexity and scale of optimization problems that can be solved [72]. Successful applications span problems in financial planning [51], supply chain scheduling [88], enterprise-wide management [54], infrastructure optimization [73], and network control [106]. Decomposition algorithms in optimization seek to address computational bottlenecks by exploiting model structures [111, 136, 23, 32, 24]. Well-known algorithms to exploit structures at the problem level include Lagrangian decomposition [43], Benders decomposition [110], Dantzig-Wolfe decomposition [12], progressive hedging [132], Gauss-Seidel [121], and the alternating direction method of multipliers (ADMM) [20].

Fundamentally, these algorithms seek to solve the original problem by solving subproblems defined over *subproblem partitions* and by coordinating subproblem solutions via communication of primal-dual information. Decomposition structures can also be exploited at the linear-algebra level (i.e. inside optimization solvers) and include block elimination and preconditioning strategies (e.g. such as Schur and Riccati decompositions) [52, 72, 109, 105]. In linear-algebra decomposition schemes, the original optimization problem is solved by using a general algorithm (e.g., interior-point, sequential quadratic programming, or augmented Lagrangian) and decomposition occurs during the computation of the search step *within* the algorithm.

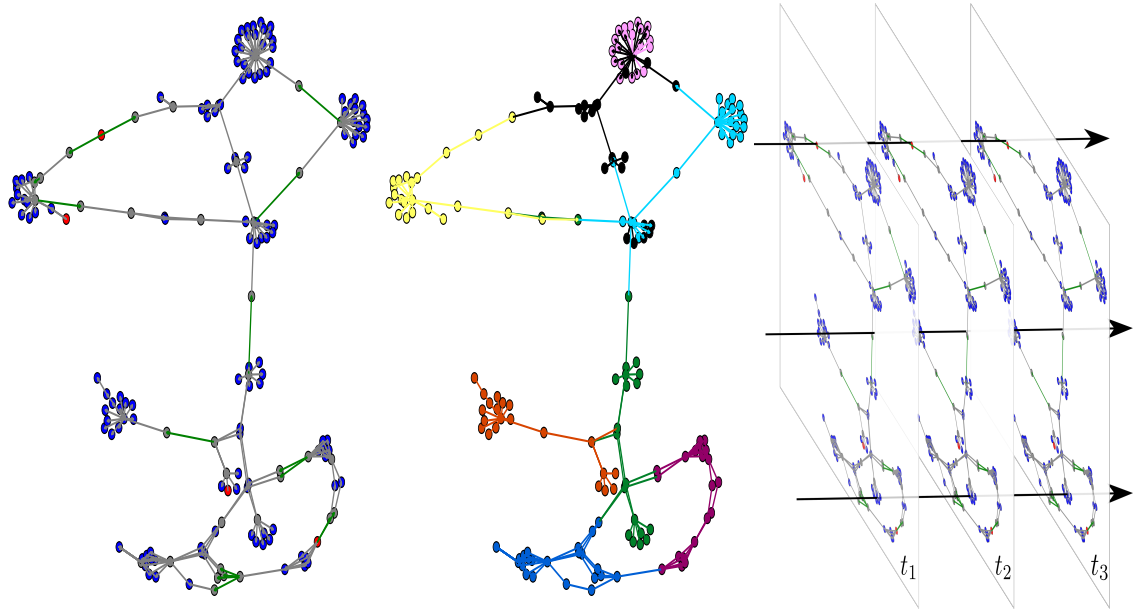
The efficient use of decomposition algorithms relies on the ability to communicate model structures in a *flexible* manner. Surprisingly, the development of *modeling environments* that support the use and development of decomposition algorithms has remained rather limited. As a result, decomposition algorithms have been mostly used to tackle models that exhibit rather obvious structures such as in stochastic optimization [124, 84, 63], network optimization [119], dynamic optimization [15], and hierarchical optimization [53]. Existing modeling environments that support structured modeling include the structure-conveying modeling language (*SML*) [31], which extends `AMPL` [46]) which conveys structures in the form of variable and constraint blocks. `Pyomo` [59] is a *Python*-based modeling package that also enables the expression of structures in the form of variable and constraint blocks and provides a modeling template called `PySP` [132] to create and solve multi-stage stochastic programs with progressive hedging and Benders decomposition. Within the *Julia* language, `StuctJuMP` [64] and `StochasticPrograms.jl` [15] are extensions of `JuMP.jl` [37] that express stochastic optimization structures.

An issue that arises in using structured modeling frameworks is that identifying blocks that are suitable for a decomposition algorithm (and a corresponding computing architecture) is not a trivial task. For instance, we may wish to identify blocks that have sparse external connectivity to reduce the amount of inter-block coupling [109], or we might want to ensure that the blocks are of similar size (to avoid load imbalance issues)

[78] if we execute on a parallel computer. Existing structured modeling environments lack the capabilities to tackle these issues.

To further highlight some of the challenges that arise in decomposing structured problems, consider the natural gas network depicted in the left panel of Figure 3.1. This is a regional gas transmission system that contains 215 pipelines, 16 compressors, and 172 junction points [29]. The problem has an obvious structure induced by the network topology which conveys connectivity between components (pipelines, compressors, and junctions). Hidden in this representation, however, is the complex internal connectivity present in the components induced by variables and constraints that capture physical coupling (e.g., conservation of mass, energy, and momentum). The issue here is that there is a severe imbalance in the complexity of the components. Pipelines might contain partial differential equations to capture flow while compressors contain simpler algebraic equations (this observation was briefly touched upon with Figure 2.8). Moreover, the network representation hides connectivity of the components across *time*, which gives rise to complex space-time coupling. Intuitively, we could decompose the model *spatially* by exploiting the network topology, as shown in the middle panel of Figure 3.1b. This approach unfortunately, does not account for inner component complexity and results in partitions with unequal sizes (i.e. disparate numbers of variables and constraints). This gives rise to load balance issues that can limit the advantages of parallel computation. To deal with this issue, we could decompose the problem *temporally*, as shown in the right panel of Figure 3.1c. This approach leads to well-balanced partitions (every partition is a copy of the network), but it also leads to significantly more coupling between partitions. The high degree of coupling limits scalability of algorithms since increased coupling usually requires more communication and slows down convergence. One would intuitively expect that there exist partitions that can balance coupling and load balancing (by traversing space-time and exploiting inner block complexity). Identifying such partitions, however, requires *unfolding* of the model structure. Figure 3.2a provides a visual rendering of the spatial representation of the natural gas system that unfolds inner component connec-

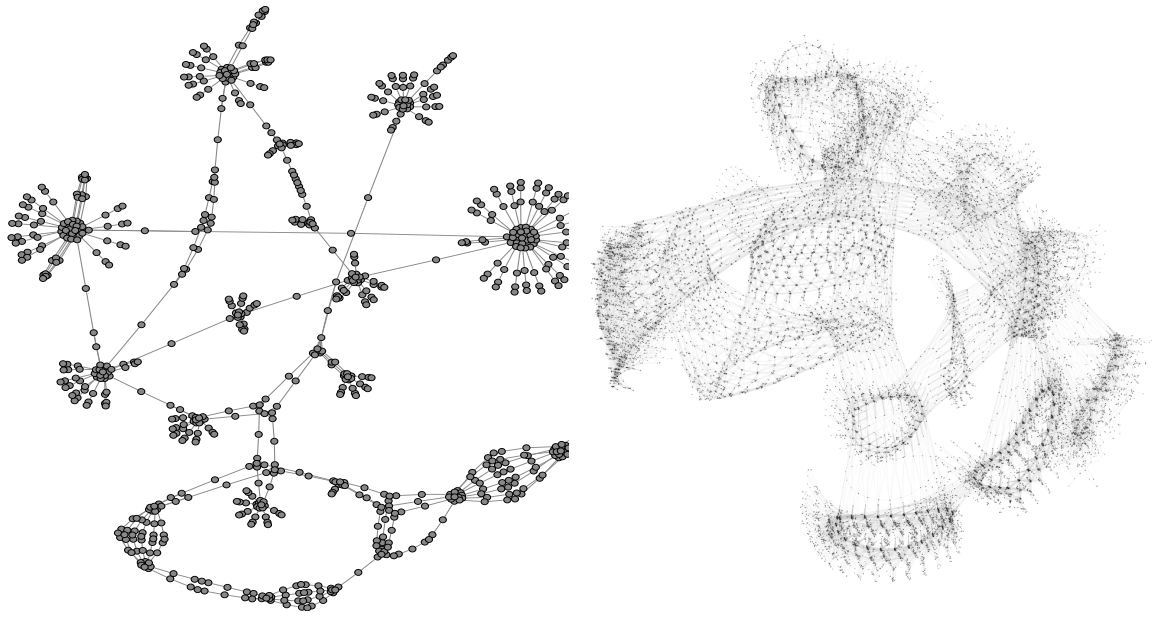
tivity. Here, the size of the clusters gives an initial indication of component complexity. Figure 3.2b further unfolds temporal connectivity between components and this clearly reveals interesting (but complex) space-time structures. Determining effective partitions from such structures requires of advanced graph partitioning techniques.



(a) Network Topology (b) Network Partition (c) Time Partition

Figure 3.1: Depiction of a regional natural gas system and possible partitions of the corresponding optimal control problem. The network layout of the system (left), the system split into eight network partitions (middle), and the system represented by three time partitions (right).

A key benefit of the `OptiGraph` paradigm presented in Chapter 2 is that it naturally captures inner component complexity in optimization models and as such it can take advantage of powerful graph partitioning techniques to decompose optimization problems. For instance, graph partitioning tools such as `Metis` [75] and `Scotch` [100] provide algorithms to analyze problem structure and automatically identify suitable partitions that can be used by decomposition algorithms. In concise terms, graph partitioning seeks to decompose a domain described by a graph such that communication between subdomains is minimized subject to load balancing (i.e., the domains are about the same size). The most ubiquitous graph partitioning applications have been used to parallelize scien-



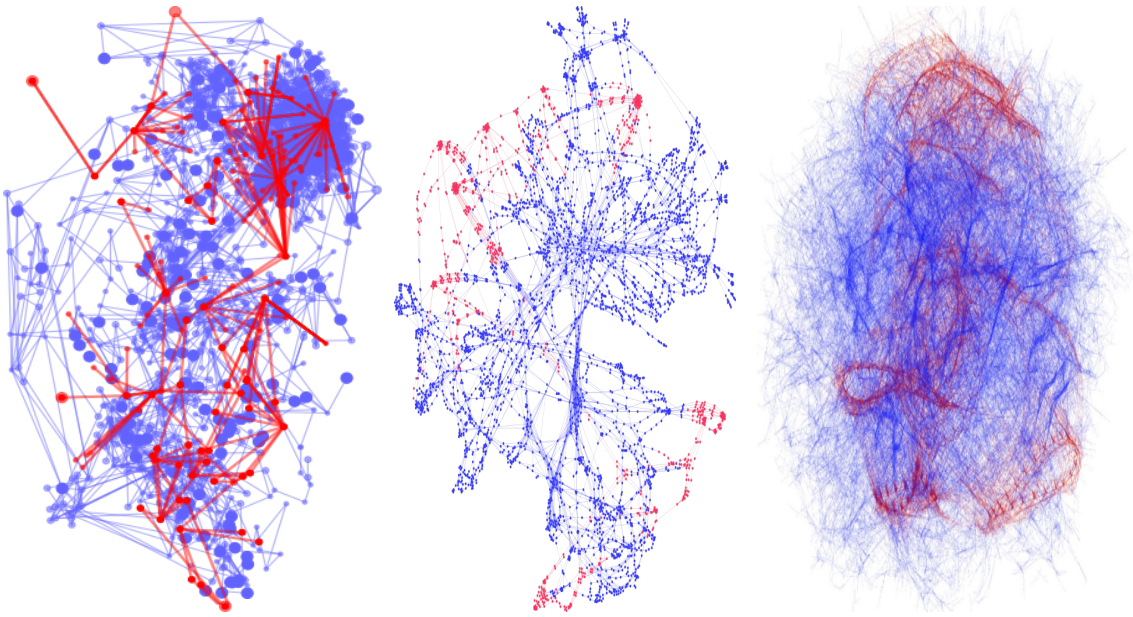
(a) Space Unfolding (b) Space-Time Unfolding

Figure 3.2: Depiction of unfolded natural gas system optimal control problem. Space unfolding of the optimal control problem (left) and space-time unfolding of the optimal control problem with 24 time periods (right).

tific simulations [113], perform sparse-matrix operations [56], and precondition systems of PDEs [114]. *Hypergraph partitioning* generalizes graph partitioning concepts to effectively decompose non-symmetric domains [35] and includes popular tools such `hMetis` [76] and `PaToH` [28]. In the context of decomposing optimization problems, graph representations have been used to partition network problems using graph coloring techniques [138] and block partitioning schemes [139], and bipartite graph representations have been used to permute linear programs [42] into block diagonal form to enable parallel solution. Hypergraph partitioning has been used to decompose mixed-integer programs to formulate Dantzig-Wolfe decompositions [131, 13] using `hMetis` and `PaToH`. Community detection approaches have been used to automate structure identification in general optimization problems by maximizing modularity and with this enable higher efficiency of decomposition algorithms [95, 128, 7].

In Chapter 2 we showed how to use `OptiGraphs` to build complex *hierarchical* optimization models. This hierarchical graph-based modeling is what fundamentally enables the

`OptiGraph` to *unfold* component complexity to exploit partitioning. To further motivate the types of capabilities enabled by hierarchical graph modeling, we take the natural-gas network in Figure 3.1 and we *couple* it to an electrical network, as shown in Figure 3.3a. Using an `OptiGraph`, the natural gas graph and the electrical power graph can be built independently and then coupled to construct a higher level combined (hierarchical) graph. This graph structure can then be communicated to a graph visualization tool which allows the graph to be analyzed using different representations. In the left panel we see a traditional representation of infrastructure coupling (that hides temporal and component coupling), whereas the middle and right panels unfold spatial and spatio-temporal coupling. The space-time graph contains over 100,000 nodes and 300,000 edges and reveals that there exist many non-obvious structures that might be exploited from a computational standpoint.



(a) Coupled Networks (b) Space Unfolding (c) Space-Time Unfolding

Figure 3.3: Network topology of coupled gas and electric systems (left), the space representation of the optimal control problem (middle), and the space-time representation of the coupled optimal control problem with 24 time periods (right). The gas and electric systems are colored red and blue respectively.

3.2 Partitioning and Manipulating OptiGraphs

Chapter 2 discussed how the `OptiGraph` facilitates systematic construction of complex hierarchical optimization models using a *bottom-up* approach. More specifically, we showed how to construct an `OptiGraph` and add subgraphs to create hierarchical structures. In this section, we show how to create `OptiGraph` structure using a *top-down* approach using graph partitioning.

3.2.1 *Hypergraph Partitioning*

The `OptiGraph` is, at its core, a hypergraph. As such, it can naturally exploit hypergraph partitioning capabilities. In this section we focus on hypergraph partitioning concepts but our discussion also applies to standard graph partitioning frameworks (a hypergraph can be projected to different standard graph representations). Hypergraph partitioning has received significant interest recently because it naturally represents complex *non-pairwise* relationships and more accurately captures such coupling compared to traditional graphs. Popular hypergraph partitioning tools include the well-known `hMetis` and `PaToH` packages, as well as the comprehensive `Zoltan` software suite which provides hypergraph partitioning algorithms for `C`, `C++`, and `Fortran` applications. More recent frameworks have made advances to create large-scale hypergraphs [70], improve hypergraph partitioning speed [91], and create high-quality hypergraph partitions [112]. To provide an overview of hypergraph partitioning techniques, we use notation that is similar to that of an `OptiGraph`. A hypergraph contains a set of hypernodes $\mathcal{N}(\mathcal{H})$ and hyperedges $\mathcal{E}(\mathcal{H})$ where we denote the hypergraph containing hypernodes and hyperedges as $\mathcal{H}(\mathcal{N}, \mathcal{E})$. In hypergraph partitioning, one seeks to partition the set of nodes $\mathcal{N}(\mathcal{H})$ into a collection $\mathcal{P}$ of at most k disjoint subsets such that $\mathcal{P} = \{P_1, P_2, ..., P_k\}$ while minimizing an objective function over the edges such as (3.1a) or (3.1b) subject to a balance constraint (3.1c) (such that partitions are roughly the same size).

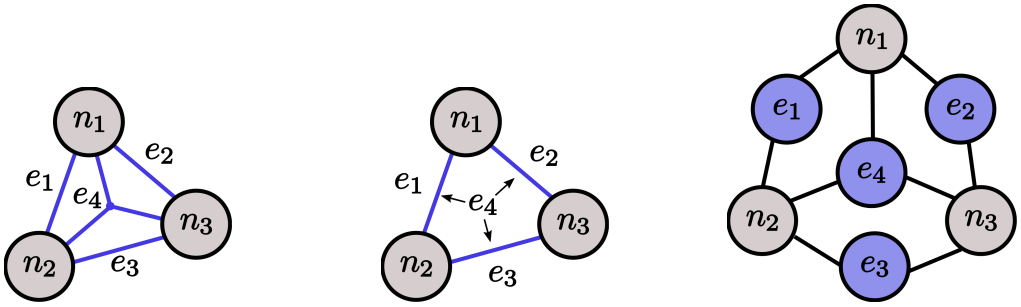
$$\Phi_{cut}(\mathcal{P}) = \sum_{e \in \mathcal{E}_{cut}(\mathcal{P})} w(e) \tag{3.1a}$$

$$\Phi_{con}(\mathcal{P}) = \sum_{e \in \mathcal{E}_{cut}(\mathcal{P})} w(e)(\lambda(e) - 1) \tag{3.1b}$$

$$\frac{1}{k} \sum_{n \in \mathcal{N}(\mathcal{H})} s(n) - \epsilon_{max} \leq \sum_{n \in \mathcal{P}_i} s(n) \leq \frac{1}{k} \sum_{n \in \mathcal{N}(\mathcal{H})} s(n) + \epsilon_{max}, \quad \forall i \in \{1, ..., k\} \tag{3.1c}$$

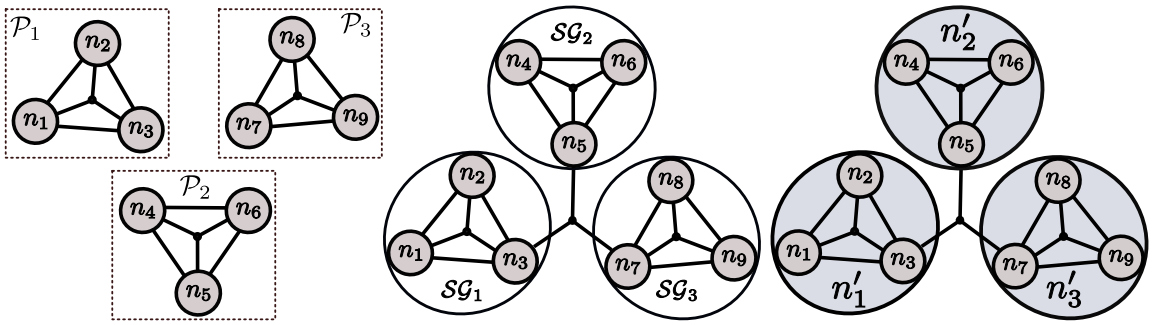
Here, Φ_{cut} and Φ_{con} are the most prominent hypergraph partitioning objectives (called the minimum edge-cut and minimum connectivity), where $\mathcal{E}_{cut}(\mathcal{P})$ is the set of *cut edges* of the partitions in $\mathcal{P}$ (i.e., all edges that *cross* partitions defined by $\mathcal{P}$). The formulation in (3.1) introduces edge weights $w(e) \rightarrow \mathbb{R}^+$ for each edge $e \in \mathcal{E}(\mathcal{H})$ and node sizes $s(n) \rightarrow \mathbb{R}^+$ for each node $n \in \mathcal{N}(\mathcal{H})$ which can be used to express problem specific attributes to the partitioner. For instance, large edge weights can be used to express tight coupling or high communication volume and node sizes can be used to represent computational load. The objective (3.1b) includes the connectivity value $\lambda(e)$ which denotes the number of partitions connected by a hyperedge e. We also define the parameter $\epsilon_{max} > 0$ in (3.1c) which specifies the maximum imbalance tolerance of the partitions. Lower values of ϵ_{max} enforce more equal-sized partitions and higher values allow for disparately sized partitions (the size of a partition is the sum of the size of its nodes).

The hypergraph is an intuitive representation for an `OptiGraph` but other representations are possible. The hypergraph in Figure 3.4a can be projected to a standard graph representation (shown in Figure 3.4b) or to a bipartite representation (shown in Figure 3.4c). Standard graph representations can utilize mature partitioning tools such as `Metis` or community detection strategies. The ability to project to different graph representations provides flexibility to experiment with different partitioning techniques. In the remainder of this chapter we utilize the hypergraph representation to partition with, but we highlight that a broader range of partitioning strategies can be captured using an `OptiGraph`.



(a) Hypergraph (b) Standard Graph (c) Bipartite Graph

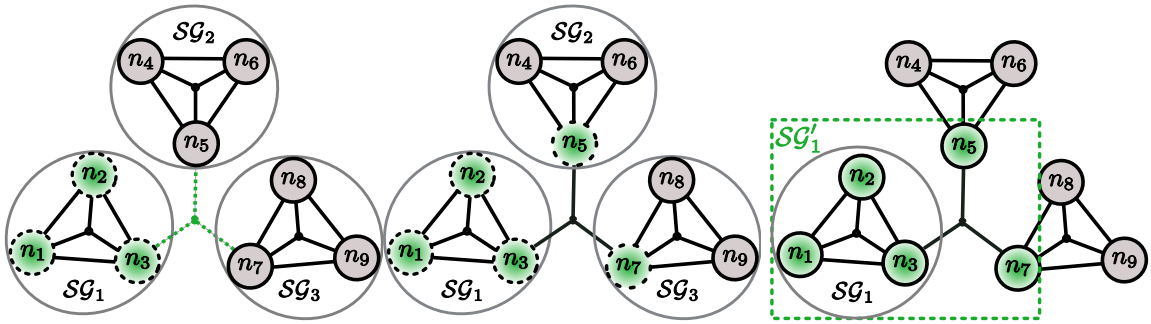
Figure 3.4: Typical graph representations used in partitioning applications. A hypergraph (left) can be projected to a standard graph (middle) or a bipartite graph (right).



(a) `OptiGraph` Partitions (b) `OptiGraph` Subgraphs (c) Aggregated Subgraphs

Figure 3.5: Depiction of the core `OptiGraph` partitioning capabilities. (Left) A partition with nine `OptiNodes`, (middle) the corresponding `OptiGraph` containing three subgraphs, and (right) the subgraphs aggregated into three new `OptiNodes`.

Here we discuss specific partitioning capabilities offered by the `OptiGraph` framework where Figure 3.5 depicts the basic partitioning concepts. Using a hypergraph partitioner (such as the ones mentioned earlier), an `OptiGraph` can be split into distinct partitions which can then be used to form *subgraphs*. The formed subgraphs can then be *combined* (aggregated) into stand-alone `OptiNodes`. For example, Figure 3.5a contains three partitions P_1,P_2, and P_3, Figure 3.5b shows the corresponding subgraphs $\mathcal{SG}_1$,$\mathcal{SG}_2$, and $\mathcal{SG}_3$ created from the partitions, and Figure 3.5c depicts aggregated `OptiNodes` n'_1,n'_2, and n'_3 which represent optimization *subproblems*.



(a) Incident Edges (b) Neighborhood (c) Expanding a Subgraph

Figure 3.6: Depiction of topology-based `OptiGraph` manipulation capabilities. (Left) querying incident edges to a subgraph, (middle) querying a subgraph neighborhood, and (right) expanding a subgraph.

3.2.2 *OptiGraph Manipulation*

The `OptiGraph` also offers topology-based manipulation which can, for instance, be used to modify subgraph structures and formulate subproblems for algorithms. Figure 3.6 depicts three core topology functions we commonly use in the `OptiGraph` framework. We can query the incident edges (Figure 3.6a) to a set of `OptiNodes` (or a subgraph) to inspect couplings (i.e. inspect incident linking constraints). We can also obtain the neighborhood (Figure 3.6b) around a set of `OptiNodes` to inspect an expanded problem domain, and we can consequently *expand* (Figure 3.6c) a subgraph into a larger domain and generate the corresponding subproblem.

3.3 Algorithms

The `OptiGraph` is a flexible abstraction that facilitates the communication of problem structure to different decomposition strategies. The structure of an `OptiGraph` can be exploited at a *problem level*, wherein the decomposition strategy treats `OptiNodes` as optimization subproblems that have their solutions coordinated to solve the entire `OptiGraph` (such as in Benders, Lagrangian, and ADMM algorithms), or the the `OptiGraph` structure can be exploited at the *linear-algebra level* using block structures that naturally arise

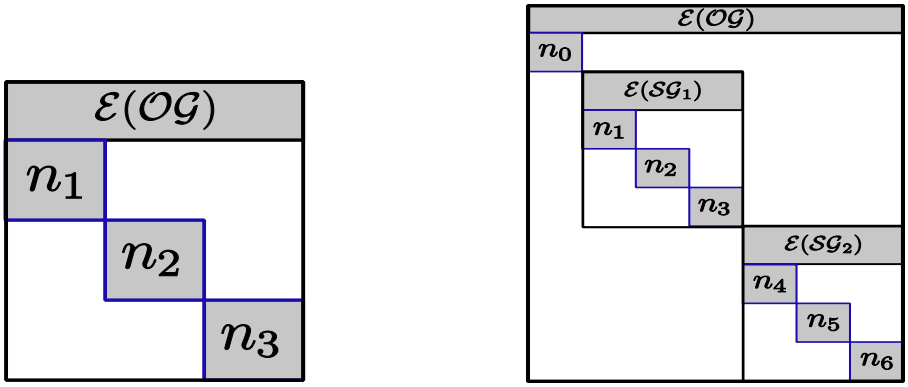


Figure 3.7: An `OptiGraph` expressed as a block structure (left) and an `OptiGraph` with subgraphs which induces a nested block structure (right)

from the graph topology (e.g. the block and nested block structures in Figure 3.7). At the linear-algebra level, the decomposition strategy is executed *within* a general optimization algorithm to compute a search step. In this case, the decomposition strategy treats `OptiNodes` as linear systems that arise from the optimality conditions of the subproblems and coordinates their solutions to find a solution of the linear system associated with the optimality conditions of the entire `OptiGraph`.

In the rest of this section we show how to use the `OptiGraph` to explain structures at the linear-algebra and problem levels. Particularly, we explain how the `OptiGraph` is used to formulate the block partitions and linear systems in an interior point algorithm, as well as how to formulate an advanced overlapping Schwarz decomposition scheme that uses the graph structure at the problem level.

3.3.1 *Linear Algebra Decomposition*

It is well-known that block structures can be exploited by linear algebra operations within interior-point algorithms. For instance, continuous problems with the partially-separable structure described by Formulation (2.1) induce structured linear algebra kernels that can be solved using tailored techniques such as Schur decomposition. We can express the continuous variant of the graph Formulation (2.1) (from Chapter 2) with feasible sets of

the form $\mathcal{X}_n := \{x \mid c_n(x) = 0\}$ which gives rise to the Karush-Kuhn-Tucker (KKT) system given by:

$$\sum_{n \in \mathcal{N}(\mathcal{OG})} \left(\nabla_{x_n} f_n(x_n) + \nabla_{x_n} c_n(x_n) \lambda_n \right) + \sum_{e \in \mathcal{E}(\mathcal{OG})} \nabla_{\{x_n\}_{n \in \mathcal{N}(e)}} g_e(\{x_n\}_{n \in \mathcal{N}(e)}) \lambda_e = 0, \quad (3.2a)$$

$$c_n(x_n) = 0, \quad n \in \mathcal{N}(\mathcal{OG}), \quad (3.2b)$$

$$g_e(\{x_n\}_{n \in \mathcal{N}(e)}) = 0, \quad e \in \mathcal{E}(\mathcal{OG}). \quad (3.2c)$$

Here, we recall that $\mathcal{N}(e)$ denotes the nodes that support edge e. For ease of presentation, we omit terms associated with barrier functions in (3.2) which exist in the presence of inequality constraints. Upon linearization of (3.2), the system of algebraic equations gives rise to the block-bordered KKT system given by:

$$\left[\begin{array}{cccc|c} K_{n_1} & & & & B_{n_1} \\ & K_{n_2} & & & B_{n_2} \\ & & \ddots & & \vdots \\ & & & K_{n_{|\mathcal{N}|}} & B_{n_{|\mathcal{N}|}} \\ \hline B_{n_1}^T & B_{n_2}^T & \dots & B_{n_{|\mathcal{N}|}}^T & \end{array}\right] \left[\begin{array}{c} \Delta w_{n_1} \\ \Delta w_{n_2} \\ \vdots \\ \Delta w_{n_{|\mathcal{N}|}} \\ \hline \Delta \lambda_{\mathcal{E}(\mathcal{OG})} \end{array}\right] = - \left[\begin{array}{c} r_{n_1} \\ r_{n_2} \\ \vdots \\ r_{n_{|\mathcal{N}|}} \\ \hline r_{\mathcal{E}(\mathcal{OG})} \end{array}\right]. \quad (3.3)$$

Here, $\Delta w_n := (\Delta x_n, \Delta \lambda_n)$ is the primal-dual step for the variables and constraints on node n and $\Delta \lambda_{\mathcal{E}(\mathcal{OG})}$ is a vector of steps corresponding to the dual variables on the `OptiEdges` in `OptiGraph` $\mathcal{OG}$, where $\Delta \lambda_{\mathcal{E}(\mathcal{OG})} := \{\Delta \lambda_e\}_{e \in \mathcal{E}(\mathcal{OG})}$. We also define

$$K_n := \left[\begin{array}{cc} W_n & J_n^T \\ J_n & 0 \end{array}\right],$$

as the block matrix corresponding to `OptiNode` n where $W_n = \nabla_{x_n, x_n} \mathcal{L}$ is the Hessian of the Lagrange function of (2.1) and $J_n := \nabla_{x_n} c_n(x_n)$ is the constraint Jacobian. We define

coupling blocks B_n as

$$B_n := \begin{bmatrix} Q_n & 0 \\ 0 & 0 \end{bmatrix},$$

where $Q_n := \nabla_{x_n}\{g_e(\{x_{n'}\}_{n' \in \mathcal{E}(n)})\}_{e \in \mathcal{E}(\mathcal{OG})}$. If all of the linking constraints in the graph are *linear*, Q_n reduces to Π_n^T where Π_n is the matrix of coefficients corresponding to the `OptiEdges` incident to node n (which corresponds to the incidence matrix of the underlying hypergraph).

Schur Decomposition

The block-bordered structure described in (3.2) can be exploited using Schur decomposition [73] or block preconditioning strategies [25, 109]. The typical Schur decomposition algorithm is described by (3.4) in terms of the `OptiGraph` abstraction where S is the schur complement matrix.

$$S = -\sum_{n \in \mathcal{N}(\mathcal{OG})} B_n^T K_n^{-1} B_n \tag{3.4a}$$

$$S\Delta\lambda_{\mathcal{E}(\mathcal{OG})} = \sum_{n \in \mathcal{N}(\mathcal{OG})} B_n^T K_n^{-1} r_n - r_{\mathcal{E}(\mathcal{OG})} \tag{3.4b}$$

$$K_n \Delta w_n = B_n \Delta\lambda_{\mathcal{E}(\mathcal{OG})} - r_n \tag{3.4c}$$

Step (3.4a) requires factorizing the linear system associated to each `OptiNode` (K_n) and computes the Schur complement contribution $B_n^T K_n^{-1} B_n$ on each node (possibly in parallel). After each contribution is computed (each requires performing a sparse factorization), the total Schur complement matrix S is created and factorized to solve the linear system (3.4b) (in serial) and take a step in the `OptiEdge` dual variables ($\lambda_{\mathcal{E}(\mathcal{OG})}$). Step (3.4c) solves for the `OptiNode` primal-dual step Δw_n given the `OptiEdge` dual step (also possibly in parallel).

General Schur decomposition exhibits two major *computational bottlenecks*. Forming the contributions $B_n^T K_n^{-1} B_n$ is expensive when there are many columns in B_n (the number of columns corresponds to the number linking constraints incident to node n). Moreover, if the node blocks have different sizes, the factorization of the blocks K_n can create load imbalance and memory issues. Second, factorizing the Schur matrix S is challenging when there are many linking constraints because this matrix is dense or is composed of dense sub-blocks. Consequently, it is important to control the amount of coupling between the blocks. The `OptiGraph` topology directly corresponds with the size of the block matrices that appear in Schur decomposition and thus can be manipulated to facilitate computational efficiency. Specifically, the partitioning of an `OptiGraph` can be used to accelerate Schur decomposition.

3.3.2 *Overlapping Schwarz Decomposition*

Overlapping Schwarz is a flexible decomposition strategy that can be exploited at linear algebra or problem levels [49, 120] and has been used to solve dynamic and network optimization problems [123, 119]. At the problem level, the overlapping Schwarz algorithm decomposes a graph structure over *overlapping* partitions and solves subproblems that exchange primal-dual information using the overlapping regions. The presence of overlap is key to improve the convergence rate of the algorithm which improves exponentially with the size of the overlapping region [120]. The Schwarz scheme is flexible in the sense that the size of overlap can be controlled to trade-off subproblem complexity (e.g. computation time and memory) with convergence speed. If the overlap becomes the entire graph, the algorithm solves the entire problem at once (and converges in one iteration). When the overlap is zero, the algorithm operates as a standard Gauss-Seidel scheme which exhibits slow convergence (or no convergence at all). The Schwarz algorithm thus spans fully centralized and fully decentralized extremes in its execution.

Development of Schwarz Algorithm

The Schwarz algorithm iteratively solves subproblems associated with overlapping subgraphs. In particular, we first construct *expanded subgraphs* $\{\mathcal{SG}_i'\}_{i=1}^N$ from the subgraphs $\{\mathcal{SG}_i\}_{i=1}^N$ obtained from `OptiGraph` partitioning. This procedure can be performed by expanding the subgraphs and adding `OptiNodes` within a prescribed distance $\omega \geq 0$. The optimization subproblems for the expanded subgraph $\mathcal{SG}_i'$ can be formulated as:

$$\min_{\{x_n\}_{n\in\mathcal{N}(\mathcal{SG}_i')}} \quad \sum_{n\in\mathcal{N}(\mathcal{SG}_i')} f_n(x_n) - \sum_{e\in\mathcal{I}_1(\mathcal{SG}_i')} (\lambda_e^k)^T g_e(\{x_n\}_{n\in\mathcal{N}(e)\cap\mathcal{N}(\mathcal{SG}_i')}, \{x_n^k\}_{n\in\mathcal{N}(e)\setminus\mathcal{N}(\mathcal{SG}_i')}) \tag{3.5a}$$

$$\text{s.t.} \quad x_n \in \mathcal{X}_n, \quad n \in \mathcal{N}(\mathcal{SG}_i'), \tag{3.5b}$$

$$g_e(\{x_n\}_{n\in\mathcal{N}(e)}) = 0, \quad e \in \mathcal{E}(\mathcal{SG}_i'), \tag{3.5c}$$

$$g_e(\{x_n\}_{n\in\mathcal{N}(e)\cap\mathcal{N}(\mathcal{SG}_i')}, \{x_n^k\}_{n\in\mathcal{N}(e)\setminus\mathcal{N}(\mathcal{SG}_i')}) = 0, \quad e \in \mathcal{I}_2(\mathcal{SG}_i'). \tag{3.5d}$$

In this formulation, the dual variables for (3.5c) and (3.5d) are denoted by λ_e, $\mathcal{N}(\mathcal{SG}_i')$ is the set of nodes in subgraph $\mathcal{SG}_i'$, and the superscript $(\cdot)^k$ denotes the iteration counter. For representation we denote $\mathcal{I}_1$ and $\mathcal{I}_2$ as two *separate* sets of incident edges where $\mathcal{I} := \mathcal{I}_1 \cup \mathcal{I}_2$. With this distinction, $\mathcal{I}(\mathcal{SG}_i')$ is the set of edges incident to $\mathcal{SG}_i'$ (i.e. edges that contain a node in another subgraph) and $\mathcal{I}_1$ and $\mathcal{I}_2$ denote how the incident linking constraints are formulated within the subproblem using either primal or dual coupling.

In our formulation, (3.5a) is the subproblem objective which is the sum of node objective functions contained in the expanded subgraph $\mathcal{SG}_i'$ where we have added the dual penalties from the incident dual linking constraints on edges $e \in \mathcal{I}_1(\mathcal{SG}_i')$, and (3.5d) represents primal constraints we have added from the edges $e \in \mathcal{I}_2(\mathcal{SG}_i')$. In this way, the incident linking constraints can be directly incorporated into the subproblem as local constraints, or they can be included in the objective function as a dual penalty (this assigning procedure can be important to the algorithm performance). The primal-dual solution of the other subproblems obtained from the previous iteration, in particular

$\{\{x_n^k\}_{n\in\mathcal{N}(e)\setminus\mathcal{N}(\mathcal{SG}_i')}\}_{e\in\mathcal{I}(\mathcal{SG}_i')}$ and $\{\lambda_e^k\}_{e\in\mathcal{I}_1(\mathcal{SG}_i')}$, also enter into the subproblem formulation. The Schwarz algorithm achieves convergence with this exchange of information. We note that the solution of each of the subproblems $i = \{1, 2, \cdots, N\}$ can be performed in parallel.

An important step in the Schwarz algorithm is the *restriction* of the subproblem solution. One obtains the primal-dual solution $\{x_n^*\}_{n\in\mathcal{N}(\mathcal{SG}_i')}$ and $\{\{\lambda_e^*\}_{e\in\mathcal{E}(\mathcal{SG}_i')\cup\mathcal{I}_2(\mathcal{SG}_i')}\}$ by solving (3.5). A key observation is that the solutions from different subproblems overlap (the solution for overlapping nodes may appear in multiple subproblems). We thus use a restriction step to eliminate the multiplicity. In particular, we discard the part of the solution associated with the nodes that are acquired by subgraph expansion. The restriction step can be expressed as:

$$\forall i \in \{1, 2, \cdots, N\}, \quad \{x_n^k\}_{n\in\mathcal{N}(\mathcal{SG}_i)},\ \{\lambda_e^k\}_{e\in\mathcal{E}(\mathcal{SG}_i)} \leftarrow \text{solution of (3.5)}.$$

Next, the primal-dual residual at any Schwarz iteration k is evaluated according to the residual to the KKT system for (3.6). We define $r_e^{k,Pr}$ as the primal residual of the linking constraints on edge e at iteration k, and $r_e^{k,Du}$ as the dual residual of the linking constraints on edge e. In the `OptiGraph` context, the primal error evaluates the linking constraints in the higher level graph $e \in \mathcal{E}(\mathcal{OG})$, and the dual error evaluates the consensus of the dual values of the expanded subgraphs that contain the edge e. The formal definition of the residuals is given by:

$$r_e^{k,Pr} := g_e(\{x_n^k\}_{n\in\mathcal{N}(e)}), \quad e \in \mathcal{E}(\mathcal{OG}), \tag{3.6a}$$

$$r_e^{k,Du} := \lambda_e^k(\mathcal{SG}_i') - \lambda_e^k(\mathcal{SG}_j'), \quad e \in \mathcal{E}(\mathcal{OG}). \tag{3.6b}$$

and the termination criteria for the algorithm can be set as follows:

$$\text{stop if: } \max_{e\in\mathcal{E}(\mathcal{OG})} \|r_e^{k,Pr}\|_\infty, \max_{e\in\mathcal{E}(\mathcal{OG})} \|r_e^{k,Du}\|_\infty \leq \epsilon^{tol} \tag{3.7}$$

where ϵ^{tol} is the prescribed convergence tolerance.

The Schwarz algorithm can be expressed using syntax that closely matches that of the `OptiGraph` abstraction, as shown in Algorithm 3.1. Figure 3.8 also depicts how primal-dual information is exchanged in the overlapping subgraph scheme using a simple `OptiGraph`. The figure depicts a simple graph that contains two subgraphs $\mathcal{SG}_1$ and $\mathcal{SG}_2$ and one edge e_2 that connects the subgraphs ($e_2 \in \mathcal{E}(\mathcal{OG})$). The right side of Figure 3.8 shows the expanded subgraphs $\mathcal{SG}'_1$ and $\mathcal{SG}'_2$. In the illustration, edge e_1 is incident to subgraph $\mathcal{SG}'_2$ and communicates primal information (i.e. it is in the set $\mathcal{I}_2(\mathcal{SG}'_2)$), and edge e_3 is incident to subgraph $\mathcal{SG}'_1$ and communicates dual information to subgraph $\mathcal{SG}'_1$ (i.e. it is in the set $\mathcal{I}_1(\mathcal{SG}'_1)$).

Algorithm 3.1 Schwarz Algorithm for Solving an `OptiGraph`

Input graph $\mathcal{OG}$, non-expanded subgraphs $\{\mathcal{SG}_1, ..., \mathcal{SG}_N\}$ and expanded subgraphs $\{\mathcal{SG}'_1, ..., \mathcal{SG}'_N\}$.

Initialize x^0, λ^0

Formulate subproblems in (3.5)

while termination criteria not satisfied **do**

  for $i = 1 : N$ (in parallel) **do**

    Retrieve $\{\{x_n^k\}_{n \in \mathcal{N}(e) \setminus \mathcal{N}(\mathcal{SG}'_i)}\}_{e \in \mathcal{I}(\mathcal{SG}'_i)}$ and $\{\lambda_e^k\}_{e \in \mathcal{I}_1(\mathcal{SG}'_i)}$ and update subproblems.

    Solve subproblem (3.5) to obtain $\{x_n^{k+1}\}_{n \in \mathcal{N}(\mathcal{SG}_i)}$, $\{\lambda_e^{k+1}\}_{e \in \mathcal{E}(\mathcal{SG}_i)}$

  end for

  Compute $\{r_e^{k,Pr}\}_{e \in \mathcal{E}(\mathcal{OG})}$, $\{r_e^{k,Du}\}_{e \in \mathcal{E}(\mathcal{OG})}$ and evaluate termination criteria (3.7).

end while

3.4 Software Framework: Decomposition with Plasmo.jl

In Section 2.3 we introduced the basic modeling functions in `Plasmo.jl` and showed how it used hierarchical graph structures to construct optimization problems. This section in-

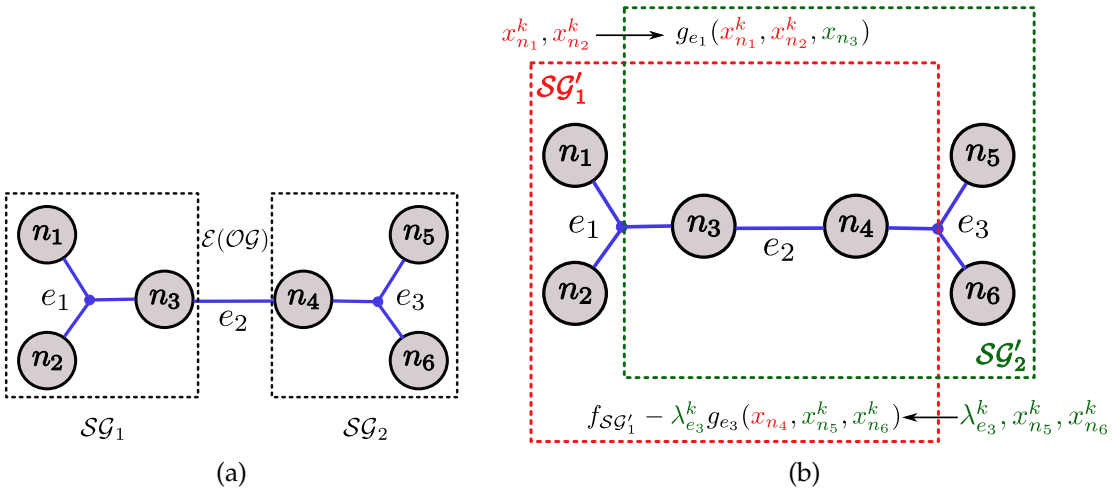


Figure 3.8: Depiction of Schwarz Algorithm. The original graph containing two subgraphs ($\mathcal{SG}_1$ and $\mathcal{SG}_2$) connected by edge e_2 (left), and the graph with expanded subgraphs ($\mathcal{SG}'_1$ and $\mathcal{SG}'_2$) (right). The expanded subgraphs overlap at nodes n_3 and n_4.

troduces the corresponding syntax to *partition* `OptiGraphs` and use graph manipulation functions to develop decomposition algorithms. To perform hypergraph partitioning we use the `KaHyPar` [112] hypergraph partitioner through the `KaHyPar.jl` interface. `KaHyPar` targets the creation of high quality partitions and offers a straightforward `C` library interface which facilitates its connection with `Plasmo.jl`. Throughout our examples we use the default `KaHyPar` configuration which uses a direct multilevel k-way algorithm with community detection initialization.

Table 3.1 summarizes the core graph partitioning and manipulation functions in `Plasmo.jl`. The `gethypergraph` function returns a hypergraph object (extends a `LightGraphs.jl` object [116]) and `reference_map` maps the hypergraph elements back to the `OptiGraph` (i.e. hypergraph node indices are mapped back to `OptiNodes`). We also introduce a `Partition` object that describes how to formulate subgraphs within a graph. As we will show, the `Partition` object is an intermediate interface to formulate subgraphs in a general way.

Functions and Descriptions
Create a hypergraph representation of `graph`. `hypergraph,ref = gethypergraph(graph::OptiGraph)`
Create a `Partition` given an `OptiGraph`, a `vector` of integers and a mapping `ref_map`. `partition = Partition(graph::OptiGraph,vector::Vector{Int},mapping::Dict{Int,OptiNode})`
Reform `graph` into subgraphs using `partition`. `make_subgraphs!(graph::OptiGraph,partition::Partition)`
Combine subgraphs in `graph` such that `n_levels` of subgraphs remain. `aggregate(graph::OptiGraph,n_levels::Int)`
Retrieve incident `OptiEdges` of `OptiNodes` in `graph`. `incident_optiedges(graph::OptiGraph,nodes::Vector{OptiNode})`
Retrieve neighborhood within `distance` of `nodes`. `neighborhood(graph::OptiGraph,nodes::Vector{OptiNode},distance::Int)`
Retrieve a subgraph from `graph` including neighborhood nodes within `distance` of `sg`. `expand(graph::OptiGraph,sg::OptiGraph,distance::Int)`

Table 3.1: Overview of core partitioning and topology functions in `Plasmo.jl`.

3.4.1 *Partitioning a Dynamic Optimization Problem*

To demonstrate the partitioning capabilities in `Plasmo.jl`, we use a simple dynamic optimization problem [121] given by (3.8).

$$\min_{\{x,u\}} \sum_{t=1}^{T} x_t^2 + u_t^2 \tag{3.8a}$$

$$\text{s.t.} \quad x_{t+1} = x_t + u_t + d_t, \quad t \in \{1, ..., T-1\} \tag{3.8b}$$

$$x_1 = 0 \tag{3.8c}$$

$$x_t \geq 0, \quad t \in \{1, ..., T\} \tag{3.8d}$$

$$u_t \geq -1000, \quad t \in \{1, ..., T-1\} \tag{3.8e}$$

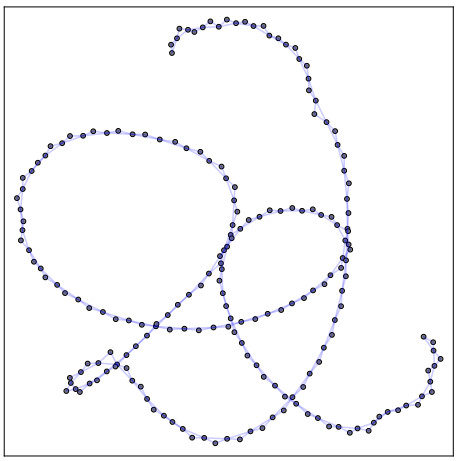
Here, x is a vector of states and u is a vector of control actions which are both indexed over the set of time indices $t \in \{1, ..., T\}$. Equation (3.8a) minimizes the state trajectory with minimal control effort (energy), (3.8b) describes the state dynamics, and (3.8c) is the initial condition. This problem can be formulated using an `OptiGraph` as shown in Code Snippet 3.1 (in much the same way as the examples in Section 2.3). We define the number of time periods $T = 100$ and create a disturbance vector `d` (data) on Lines 1 through 4. In

our implementation we create separate sets of nodes for the states and controls on Lines 10 and 11, but it is also possible to define nodes for each individual time interval and add state and control variables to the resulting nodes. Having control over this granularity is convenient for expressing what can be partitioned (i.e. features defined in an `OptiNode` will remain in the same partition). Next we setup the state and control `OptiNodes` on Lines 14 through 29, we use a linking constraint to capture dynamic coupling on Line 32 and we show how to solve the problem with `Ipopt` on Line 37. We visualize the graph topology and matrix in Figure 3.9. The layouts depict a linear graph the matrix has no obvious structure.

We partition the graph using `KaHyPar`, as shown in Code Snippet 3.2. In this snippet, Line 2 imports the `KaHyPar` interface and Line 5 creates a hypergraph representation of the `OptiGraph` using `gethypergraph`. We also return a `reference_map` which maps the hypergraph elements back to the original `OptiGraph`. Line 8 performs hypergraph partitioning using `KaHyPar` with a maximum imbalance (ϵ_{max}) of 10% and Line 11 creates a `Partition` object using the resulting partition vector and the `reference_map`. Line 14 creates subgraphs in the `graph` using the `Partition` object and the `make_subgraphs!` function. We visualize the topology and matrix of the partitioned problem on Lines 16 and 19 which are shown in Figure 3.10. This reveals eight distinct partitions and their corresponding coupling. The partitions are well-balanced and the matrix is now rearranged into a banded structure that is typical of dynamic optimization problems (partitioning automatically induces reordering). `Plasmo.jl` can also use other graph representations to perform partitioning. For instance, it is possible to create a traditional graph representation, as shown in Code Snippet 3.3, and partition it with `Metis`. The `reference_map` can then be used to obtain the original `OptiGraph` elements to create a graph `Partition`. We could also partition less intuitive representations (e.g., such as a bipartite graph) in the same way. The partitioning procedure shown here can also be repeated to create an arbitrary number of subgraph levels (inducing a multi-level hierarchical structure).

Code Snippet 3.1: Construction of dynamic optimization problem (3.8)

```
using Plasmo, Plots, Ipopt

T = 100 #number of time points
d = sin.(1:T) #disturbance

#Create an OptiGraph
graph = OptiGraph()

#Add nodes for states and controls
@optinode(graph,state[1:T])
@optinode(graph,control[1:T-1])

#Add state variables
for (i,node) in enumerate(state)
    @variable(node,x)
    @constraint(node, x >= 0)
    @objective(node,Min,x^2)
end

#Add control variables
for node in control
    @variable(node,u)
    @constraint(node, u >= -1000)
    @objective(node,Min,u^2)
end

#Initial condition
n1 = state[1]
@constraint(n1,n1[:x] == 0)

#Dynamic coupling
@linkconstraint(graph,[t = 1:T-1],state[t+1][:x] == state[t][:x] +
                                  control[t][:u] + d[t])

#Optimize with Ipopt
ipopt = Ipopt.Optimizer
optimize!(graph,ipopt)

#Plot result structure
plt_graph4 = plot(graph,
layout_options = Dict(:tol => 0.1,:iterations => 500),
linealpha = 0.2,markersize = 6)

plt_matrix4 = spy(graph)
```



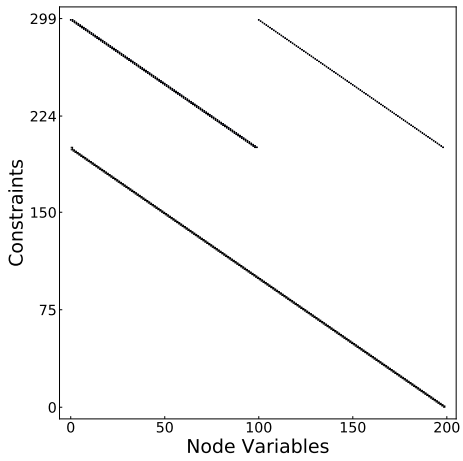


Figure 3.9: Output visuals for Code Snippet 3.1 showing graph structure of dynamic optimization problem.

Code Snippet 3.2: Creating subgraphs using hypergraph partitioning with `KaHyPar`

```
#Import KaHyPar interface
using KaHyPar

#Create a hypergraph representation of the OptiGraph
hypergraph,reference_map = gethypergraph(graph)

#Perform hypergraph partitioning using KaHyPar
node_vector = KaHyPar.partition(hypergraph,8,imbalance = 0.1)

#Create a Partition object
partition = Partition(graph,node_vector,reference_map)

#Create subgraphs using the partition object and reference_map
make_subgraphs!(graph,partition)

plt_graph5 = plot(graph,
layout_options = Dict(:tol => 0.01,:iterations => 500),
linealpha = 0.2,markersize = 6,subgraph_colors = true);
plt_matrix5 = spy(graph,subgraph_colors = true);
```

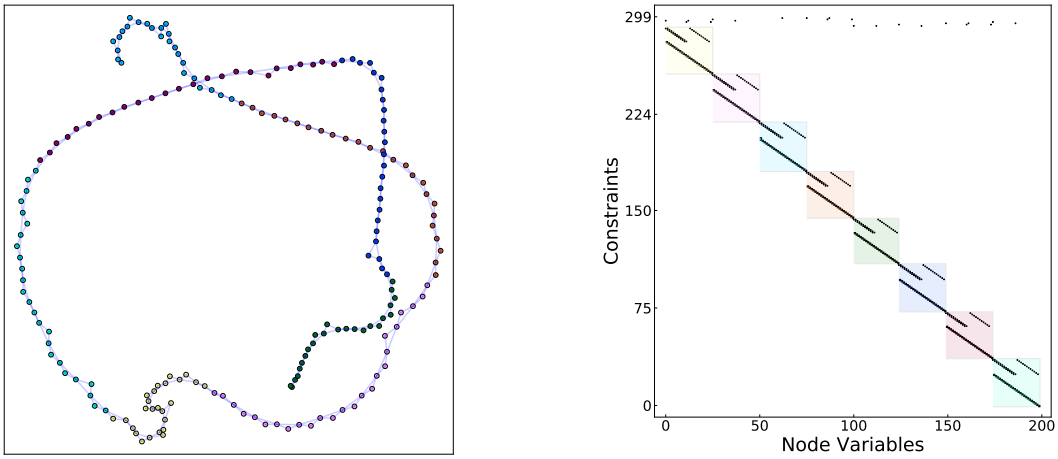


Figure 3.10: Output visuals for Code Snippet 3.2 showing partitions and reordering of dynamic optimization problem.

Code Snippet 3.3: Creating a subgraph partition using graph partitioning with `Metis`

```
#Import the Metis interface
using Metis

#Retrieve underlying hypergraph from dynamic opt problem graph
simple_graph,reference_map = getcliquegraph(graph)

#Run Metis direct k−way partitioning with a 8 maximum of partitions
node_vector = Metis.partition(simple_graph,8,alg = :KWAY)

#Create a Partition object using node vector and reference_map
partition = Partition(node_vector,reference_map)

#Create subgraphs using Partition object
make_subgraphs!(graph,partition)
```

The `OptiNodes` in a graph can be *aggregated* to form larger nodes, as shown in Figure 3.5c. Aggregation is key for communicating subproblems to decomposition algorithms that operate at different levels of granularity. Aggregation can also collapse an entire graph into a single node, producing a standard optimization problem to be solved with an off-the-shelf solver (such as was done in Chapter 2). Code Snippet 3.4 shows how to aggregate the graph of the dynamic optimization problem into a new aggregated graph with eight `OptiNodes`. We create a new combined graph `new_optigraph` on Line 2 by using `aggregate` on the `graph` from Snippet 3.2. We provide the integer `0` to the function to specify that we want zero subgraph levels which converts the eight subgraphs into `OptiNodes`. For hierarchical graphs with many levels, it is possible to define how many subgraph levels we want to retain. The graph and matrix layouts are plotted for the aggregated model on Lines 5 and 8 and these are shown in Figure 3.11.

3.4.2 *Using Graph Topology Functions*

We now demonstrate graph topology functions by computing *overlapping* partitions for the dynamic optimization example. This is shown in Code Snippet 3.5. Here, Line 2 obtains subgraphs from the `OptiGraph graph` created in Snippet 3.2, Line 8 determines and returns the `OptiEdges` incident to the `OptiNodes` in the first subgraph `sg1`, and Line 12 returns the complete neighborhood around the same of `OptiNodes` within a distance of two. On Line 16, we broadcast the `expand` function (using dot syntax `expand.()` and

Code Snippet 3.4: Aggregating nodes in an OptiGraph

```
#Combine the subgraphs in graph into OptiNodes in a new OptiGraph
aggregated_graph,ref_map = aggregate(graph,0)

#plot the graph a matrix layouts of the aggregated OptiGraph
plt_graph6 = plot(aggregated_graph,
layout_options = Dict(:tol => 0.01,:iterations => 10),
node_labels = true,markersize = 30,labelsize = 20,node_colors = true)
plt_matrix6 = spy(aggregated_graph,node_labels = true,node_colors = true)
```

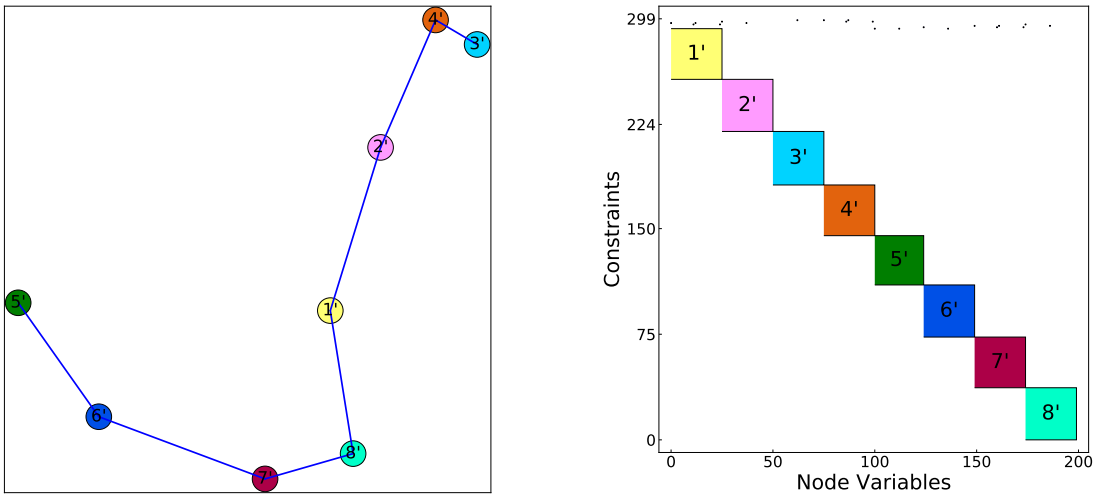


Figure 3.11: Output visuals for Code Snippet 3.4 showing aggregated graph of dynamic optimization problem.

`Ref` as typical in `Julia`) to create a new set of subgraphs, each expanded by a distance of two. Line 19 plots the layout of `graph` with the expanded subgraphs as is shown in Figure 3.12. This shows eight distinct partitions (each with a unique color) where the larger markers represent nodes that are part of multiple subgraphs (these are also the average color of their containing subgraphs). Figure 3.12 shows the corresponding matrix layout where highlighted columns indicate that the `OptiNode` appears in other subgraph blocks. The overlapping partitions can be used in decomposition algorithms such as the Schwarz algorithm presented in Section 3.3.2.

3.5 Case Study: Decomposition of a Natural Gas Optimal Control Problem

This case study demonstrates how we can use `OptiGraph` partitioning to decompose the space-time structure of an optimal control problem and exploit the Schur decomposition strategy presented in Section 3.3.1.

3.5.1 *Problem Setup*

We consider the system of connected pipelines in series shown in Figure 3.13 (this is the same pipeline system used in Section 2.4). This linear network includes a gas supply at one end, a time-varying demand at the other end, and twelve compressor stations. The gas junctions connect thirteen pipelines which forms an `OptiGraph` with a linear topology.

Code Snippet 3.5: Using Graph Topology Functions

```
#Get the current subgraphs in graph
subgraphs = getsubgraphs(graph)

#Query the first subgraph
sg1 = subgraphs[1]

#Query the edges incident to the nodes in sg1
incident_edges = incident_edges(graph,all_nodes(sg1))

distance = 2
#Query the nodes within distance 2 of sg1
nodes = neighborhood(graph,all_nodes(sg1),distance)

#Broadcast expand function to each subgraph.
#Obtain vector of expanded subgraphs
expanded_subgraphs = expand.(Ref(graph),subgraphs,Ref(distance))

#Plot the expanded subgraphs
plt_graph7 = plot(graph,expanded_subgraphs,
layout_options = Dict(:tol => 0.01,:iterations => 1000),
markersize = 6,linealpha = 0.2)
plt_matrix7 = spy(graph,expanded_subgraphs)
```

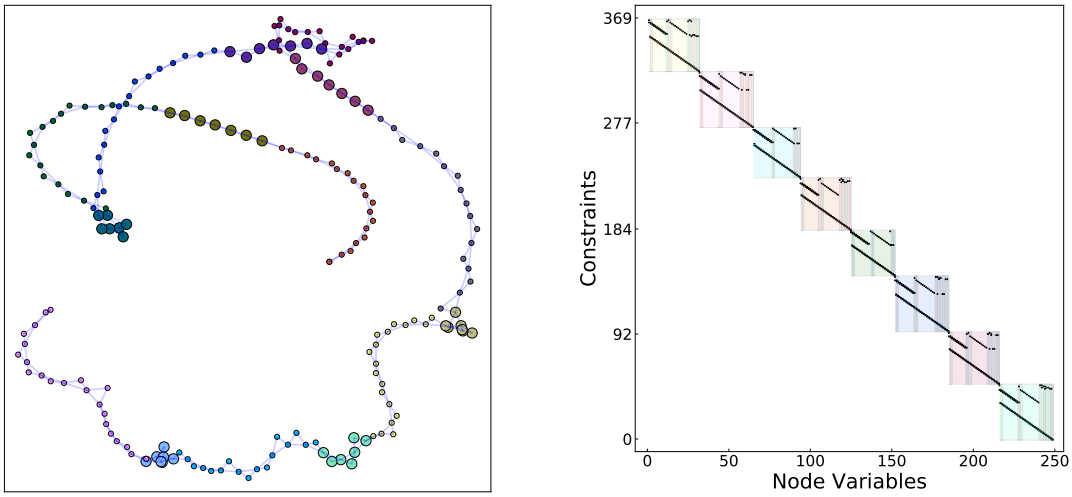


Figure 3.12: Output visuals for Code Snippet 3.5 showing overlapping subgraphs of dynamic optimization problem.

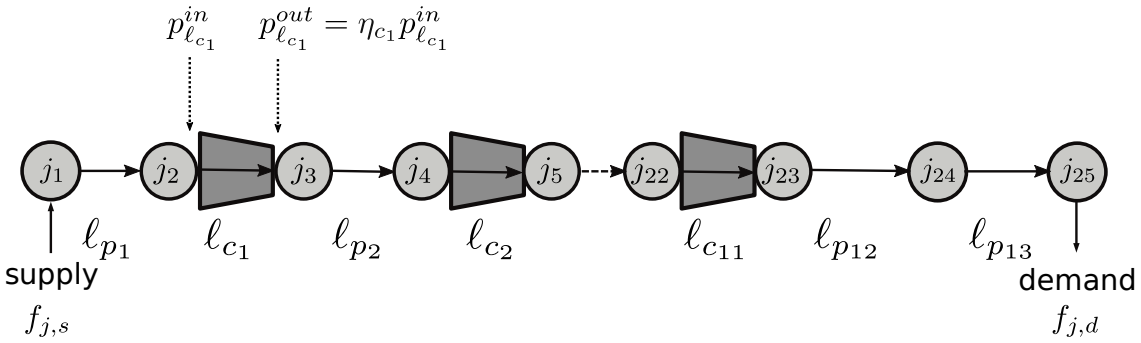


Figure 3.13: Multi-pipeline system depiction for optimal control problem.

$$\min_{\substack{\{\eta_{\ell,t}, f^{target}_{d,t}\} \\ \ell \in \mathcal{L}_c, d \in \mathcal{D}, t \in \mathcal{T}}} \quad \sum_{\substack{\ell \in \mathcal{L}_c \\ t \in \mathcal{T}}} \alpha_\ell P_{\ell,t} - \sum_{\substack{d \in \mathcal{D} \\ t \in \mathcal{T}}} \alpha_d f^{target}_{d,t} \tag{3.9a}$$

s.t. Junction Limits (A.1) (3.9b)

Compressor Equations (A.3) (3.9c)

Pipeline Equations (A.21) (3.9d)

Initial Condition (A.22) (3.9e)

Refill Line-Pack (A.23) (3.9f)

Boundary Conditions (A.29) (3.9g)

Junction Conservation (A.28) (3.9h)

In contrast to the estimation problem 2.7 in Section 2.4, this study seeks to solve an *optimal control* problem that maximizes revenue over a 24 hour time period given a forecasted demand profile. (3.9) represents the optimal control problem where (3.9a) denotes the objective function which maximizes revenue by controlling compressor actions and demand deliveries. In this formulation, α_ℓ and $P_{\ell,t}$ are the compression cost (\$/kW), and compression power for compressor ℓ at time t, and α_d and $f^{target}_{d,t}$ are the demand price and target demand flow for demand d at time t. (3.9b) captures junction pressure limits, supply flows, and demand flows, (3.9c) describes compressor equations and limits, (3.9d)

defines discretized pipeline equations for mass and momentum, (3.9e) denotes the initial condition of the partial differential equations, and (3.9f) requires the optimizer to refill the line-pack (gas inventory) in each pipeline at the end of the operating horizon. (3.9g) and (3.9h) describe boundary conditions on each pipeline and mass conservation at each junction respectively. The details of the summarized equations are found in Appendix A.

3.5.2 *Modeling and Partitioning*

For our implementation we use the the `OptiGraph` functions found in Appendix A to build junction, compressor, and pipeline `OptiGraphs` (i.e. Code Snippets A.1, A.2, and A.3), and specifically use Code Snippet A.4 to construct the gas network optimal control problem. Once we have constructed an `OptiGraph` representation of the optimal control problem, we can use hypergraph partitioning to decompose the resulting space-time structure. This highlights a key feature of the `OptiGraph` is that its hierarchical style of modeling makes it possible to *unfold* the component structure which facilitates partitioning. This idea is depicted more clearly in Figure 3.14 In this figure, we show how compressors, junctions, and pipelines are represented as distinct subgraphs that contain `OptiNodes` representing states in time and these subgraphs are connected in a higher level (network) `OptiGraph`.

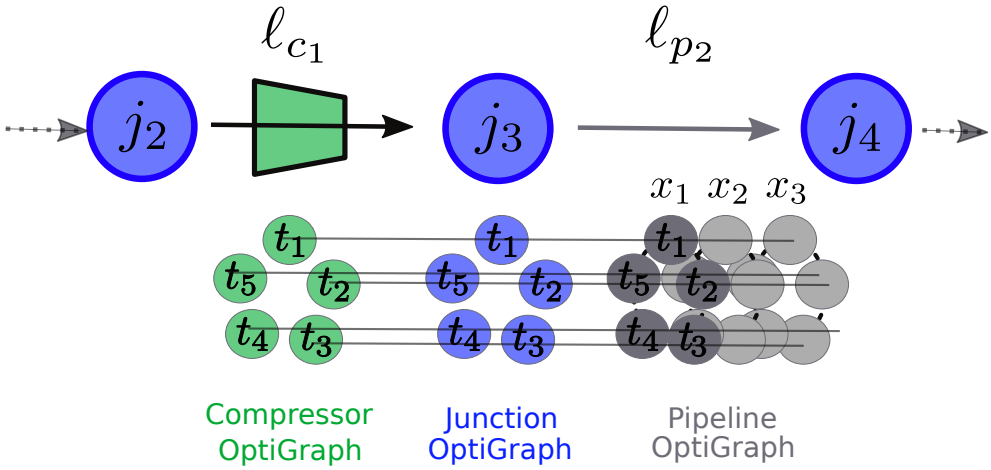


Figure 3.14: Unfolding gas network components.

We solve the optimal control problem using two strategies. We:

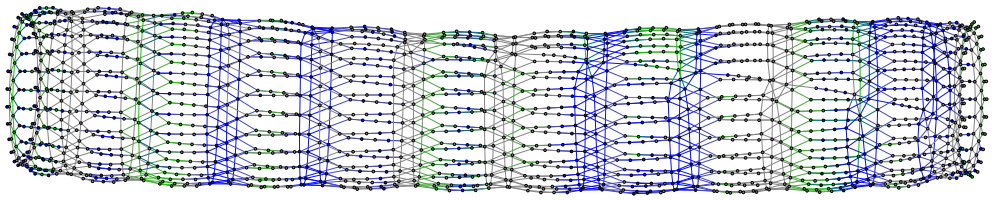
- (I) **Aggregate and solve the complete problem (with Ipopt)**: We aggregate all of the nodes and edges to produce a large optimization problem which ignores the graph structure.
- (II) **Partition and solve with general Schur decomposition (with PIPS-NLP)**: We exploit hypergraph partitioning to enable a parallel Schur decomposition scheme and solve with the `PIPS-NLP` parallel solver.

Code Snippet 3.6 depicts how straightforward it is to partition the problem. Line 2 imports the `KaHyPar` hypergraph partitioner, Line 5 formulates a hypergraph representation, and Lines 8 through 10 setup weight vectors we use for the nodes and edges (we weight nodes by their number of variables and edges by their number of linking constraints). We partition the `hypergraph` on Line 14 (into 13 partitions), we create a `Partition` object on Line 17, and we produce new subgraphs on Line 20 and finally we aggregate the subgraphs on Line 23.

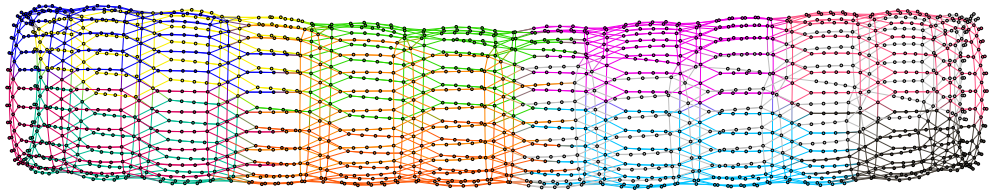
Code Snippet 3.6: Partitioning the gas network optimal control problem

```
1  #Import the KaHyPar interface
2  using KaHyPar
3
4  #Get they hypergraph representation of the gas network
5  hypergraph,ref_map = gethypergraph(gas_network)
6
7  #Setup node and edge weights
8  n_vertices = length(vertices(hypergraph))
9  node_weights = [num_variables(node) for node in all_nodes(gas_network)]
10 edge_weights = [num_link_constraints(edge) for edge in all_edges(gas_network)]
11
12 #Use KaHyPar to partition the hypergraph
13 node_vector = KaHyPar.partition(hypergraph,13,configuration = :edge_cut,
14 imbalance = 0.01, node_weights = node_weights,edge_weights = edge_weights)
15
16 #Create a model partition
17 partition = Partition(gas_network,node_vector,ref_map)
18
19 #Setup subgraphs based on partition
20 make_subgraphs!(gas_network,partition)
21
22 #Aggregate the subgraphs into OptiNodes
23 new_graph , aggregate_map = aggregate(gas_network,0)
```

The partitioned optimal control problem is visualized in Figure 3.15. The top figure (Figure 3.15a) depicts the unfolded components for the complete problem (this is equivalent to the representation in Figure 3.14) and the bottom figure (Figure 3.15b) shows the problem partitioned into 13 distinct partitions.



(a) Unfolded components



(b) Partitioned optimal control problem

Figure 3.15: Graph depictions of optimal control problem. The unfolded components (top) with blue junctions, green compressors, and grey pipelines. The partitioned hypergraph (bottom) colored with 13 distinct partitions.

`Plasmo.jl` includes an interface to the structure-exploiting interior-point solver `PIPS-NLP` called `PipsSolver.jl`. `PIPS-NLP` provides the general Schur decomposition strategy described in Section 3.3.1 and performs parallel computations via `MPI`. We highlight that `PIPS-NLP` was created to solve large-scale stochastic programming problems that exhibit a well-defined two-level tree structure (i.e. a single first stage problem coupled to second stage subproblems). With the `OptiGraph`, we can convert general graph structures into this format (e.g., via aggregation and partitioning). To solve with `PIPS-NLP` we communicate the `new_graph` created in Snippet 3.6 to the solver as shown in Snippet 3.7 to solve in parallel.

Code Snippet 3.7: Solving an `OptiGraph` model in parallel with PIPS-NLP

```
1  using Distributed
2  using MPIClusterManagers
3
4  # specify, number of mpi workers
5  manager=MPIManager(np=2)
6  # start mpi workers and add them as julia workers too.
7  addprocs(manager)
8
9  #Setup the worker environments
10 @everywhere using Plasmo
11 @everywhere using PipsSolver #Solver interface to PIPS-NLP
12
13 #get the julia ids of the mpi workers
14 julia_workers = collect(values(manager.mpi2j))
15
16 #Use the pips-nlp interface to distribute the OptiGraph among workers
17 #Here, we create the variable pips_graph on each worker
18 remote_references = PipsSolver.distribute(new_graph,julia_workers,
19 remote_name = :pips_graph)
20
21 #Solve with PIPS-NLP
22 @mpi_do manager begin
23     using MPI
24     PipsSolver.pipsnlp_solve(pips_graph)
25 end
```

Snippet 3.7 also presents a standard template for setting up distributed computing environments to use `PipsSolver.jl` which is worth discussing. Line 1 imports the `Distributed` module, which is a `Julia` package for performing distributed computing. On Line 2 we import the `MPIClusterManagers` package which allows us to map `MPI` ranks (used by `PIPS-NLP`) with `Julia` worker CPUs. We create a `manager` object and specify that we want to use 2 workers on Line 5 and we add the Julia workers on Line 7. Next we setup the model and solver environments for the added workers on Lines 10 and 11 and create a reference to the julia workers by querying the manager on Line 14. We *distribute* the `OptiGraph` (named `new_graph`) among the workers in Line 18 using the function provided by `PipsSolver`. This function sets up the relevant graph nodes on each worker and creates the graph named `pips_graph` on each worker (internally this function inspects the `OptiGraph` and allocates `OptiNodes` to worker CPUs). Finally, we use the `mpi_do` function from `MPIClusterManagers` to execute `MPI` on each worker and solve the graph. Each worker executes the `pipnlp_solve` function and communicates using `MPI` routines within `PIPS-NLP`.

3.5.3 *Results*

For our comparison, the resolution of the spatial discretization mesh of the pipeline PDEs is gradually increased to produce optimization problems that span the range of 100,000 to 2 million variables. Figure 3.16 compares the performance of the two approaches. The partitioned problem (II) can be solved with *near-linear scaling* whereas the aggregated problem solved with `Ipopt` (I) scales cubically. This result highlights the scalability that can be achieved using the partitioning approach (II), but approach (I) still benefits from having a structured solution (which we exploited in the case study in Section 2.4).

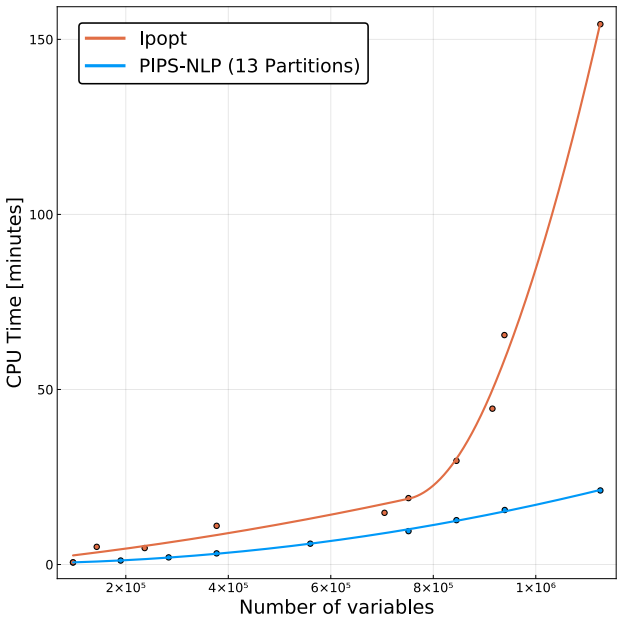


Figure 3.16: Computational times for the solution of unstructured gas pipeline formulation with `Ipopt` and for the solution of the structured formulation using `PIPS-NLP`.

3.6 Case Study: Overlapping Domain Decomposition of a DC Power Grid

This case study demonstrates how we can use `OptiGraph` partitioning and topology manipulation to pose a complex power network problem to the Schwarz overlapping domain

decomposition scheme described in Section 3.3.2.

3.6.1 *Problem Setup*

We consider a 9,241 bus test case obtained from pglib-opf (v19.05) [10]. We denote a power grid as a network $Net(\mathcal{V}, \mathcal{L}_g)$ containing a set of electric buses $\mathcal{V}$ that connect grid links (power lines) $\ell \in \mathcal{L}_g$. Each electric bus $i \in \mathcal{V}$ can include generators $q \in \Omega_i$ and serves a total power load P_i^L. The total set of generators is given by Ω such that $\bigcup_{i \in \mathcal{V}} \Omega_i = \Omega$. The network is described by the DC power flow equations [126] given by (3.10) where (3.10a) seeks to minimize the total generation cost and voltage angle difference where β is a regularization parameter. (3.10b) enforces energy conservation, and (4.2c) and (4.2d) denote limits on power generation and voltage angles. In this formulation, v_i is the bus voltage angle for each bus $i \in \mathcal{V}$, P_q is the power generation from generator q with cost coefficients $c_{q,1}$ and $c_{q,2}$ and lower and upper limits $\underline{P_q}$ and $\overline{P_q}$, and $v_{\ell,j}$ and $v_{\ell,k}$ are the inlet and outlet voltage angles on transmission line ℓ with ramp limit $\overline{v_\ell}$. We also define $\mathcal{L}_{rec}(i)$ as the set of power links received by bus i and $\mathcal{L}_{snd}(i)$ as the set of links sent from bus i. The power flow on line ℓ is defined by (4.2b), where Y_ℓ is the branch admittance for line ℓ, $v_{\ell,j}$ is the source bus voltage angle and $v_{\ell,k}$ is the destination bus voltage angle. $src(\ell)$ denotes the source bus of line ℓ and $dst(\ell)$ is destination bus for line ℓ. We denote the voltage angles on the set of reference buses in (4.2e) where $\mathcal{V}^{ref}$ is the set of reference buses.

$$\min_{\substack{\{v_i\}_{i\in\mathcal{V}} \\ \{P_q\}_{q\in\Omega}}} \sum_{q\in\Omega} \left(c_{q,1}P_q + c_{q,2}P_q^2\right) + \frac{\beta}{2}\sum_{\ell\in\mathcal{L}_g}(v_{\ell,j} - v_{\ell,k})^2 \tag{3.10a}$$

$$s.t. \sum_{q\in\Omega_i} P_q + \sum_{\ell\in\mathcal{L}_{rec}(i)} P_\ell - \sum_{\ell\in\mathcal{L}_{snd}(i)} P_\ell = P_i^L, \quad i \in \mathcal{V} \tag{3.10b}$$

$$v_{\ell,j} = v_{src(\ell)}, v_{\ell,k} = v_{dst(\ell)}, \quad \ell \in \mathcal{L}_g \tag{3.10c}$$

$$P_\ell = Y_\ell(v_{\ell,j} - v_{\ell,k}), \quad \ell \in \mathcal{L}_g \tag{3.10d}$$

$$\underline{P_q} \leq P_q \leq \overline{P_q}, \quad q \in \Omega \tag{3.10e}$$

$$-\overline{v_\ell} \leq v_{\ell,j} - v_{\ell,k} \leq \overline{v_\ell}, \quad \ell \in \mathcal{L}_g \tag{3.10f}$$

$$v_i = v_i^{ref}, \quad i \in \mathcal{V}^{ref} \tag{3.10g}$$

3.6.2 *Modeling, Partitioning, and Expansion*

We construct the DC OPF model with Code Snippet 3.9 (provided in Section 3.7), which produces an optimization problem with over 100,000 variables and constraints. We partition the produced `OptiGraph` using `KaHyPar` and then create subgraphs, expand them, and solve the problem using `SchwarzSolver.jl` (which implements the overlapping Schwarz algorithm).

Code Snippet 3.8 demonstrates how we carry out partitioning using a maximum imbalance of 10% and an overlap size of $\omega = 10$. Line 1 imports `KaHyPar` and Line 6 creates a `hypergraph` and `ref_map` which we use for partitioning. Lines 9 through 11 query the graph for edge weights and node sizes, Line 15 partitions the DC OPF problem, and Lines 18 and 21 create a `Partition` and use it to define subgraphs for the problem. Once we have subgraphs, we perform a subgraph expansion on Line 27 and execute the Schwarz solver on Line 31. We tell the solver how to treat linking constraints with the keyword arguments `primal_links` and `dual_links` which denote how subproblems are formulated. We provide `primal_links` a vector of power flow linking constraints and provide `dual_links` a vector of voltage angle linking constraints The constraints that are specified

as `primal_links` are treated as direct constraints while the constraints in `dual_links` are incorporated as dual penalty as described in (3.5).

Code Snippet 3.8: Partitioning and Formulating Overlapping Subproblems

```
using KaHyPar
using Ipopt
using SchwarzSolver

#Get the hypergraph representation of the gas network
hypergraph,ref_map = gethypergraph(dcopf)

#Setup node and edge weights
n_vertices = length(vertices(hypergraph))
node_weights = [num_variables(node) for node in all_nodes(dcopf)]
edge_weights = [num_link_constraints(edge) for edge in all_edges(dcopf)]

#Use KaHyPar to partition the hypergraph
node_vector = KaHyPar.partition(hypergraph,4,configuration = :edge_cut,
imbalance = 0.1, node_weights = node_weights,edge_weights = edge_weights)

#Create a partition object
partition = Partition(dcopf,node_vector,ref_map)

#Setup subgraphs based on partition
make_subgraphs!(dcopf,partition)

distance = 10
subgraphs = getsubgraphs(dcopf)

#Expand the subgraphs
sub_expand = expand.(Ref(dcopf),subgraphs,Ref(distance))

ipopt = Ipopt.Optimizer

schwarz_solve(dcopf,sub_expand,primal_links = power_links,
dual_links = [angle_i;angle_j]],
sub_optimizer = ipopt,max_iterations = 100,tolerance = 1e-3)
```

3.6.3 *Results*

Figure 3.17 depicts the original and overlapping partitions obtained from Snippet 3.8 (visualized using `Gephi`). We experiment with different values for maximum imbalance and overlap and obtain the results in Figure 3.18. We see that the Schwarz algorithm fails to converge with an overlap value of one (for any imbalance value) which is consistent with the convergence analysis in [123], but a sufficient overlap of 10 produces smooth convergence (for each imbalance value). We also observe that larger partition imbalance results in faster convergence (with sufficient overlap) which is likely due to the smaller edge cut and fewer linking constraints that need to be satisfied. We thus see that the trade-offs of imbalance and coupling are complex and differ under different settings.

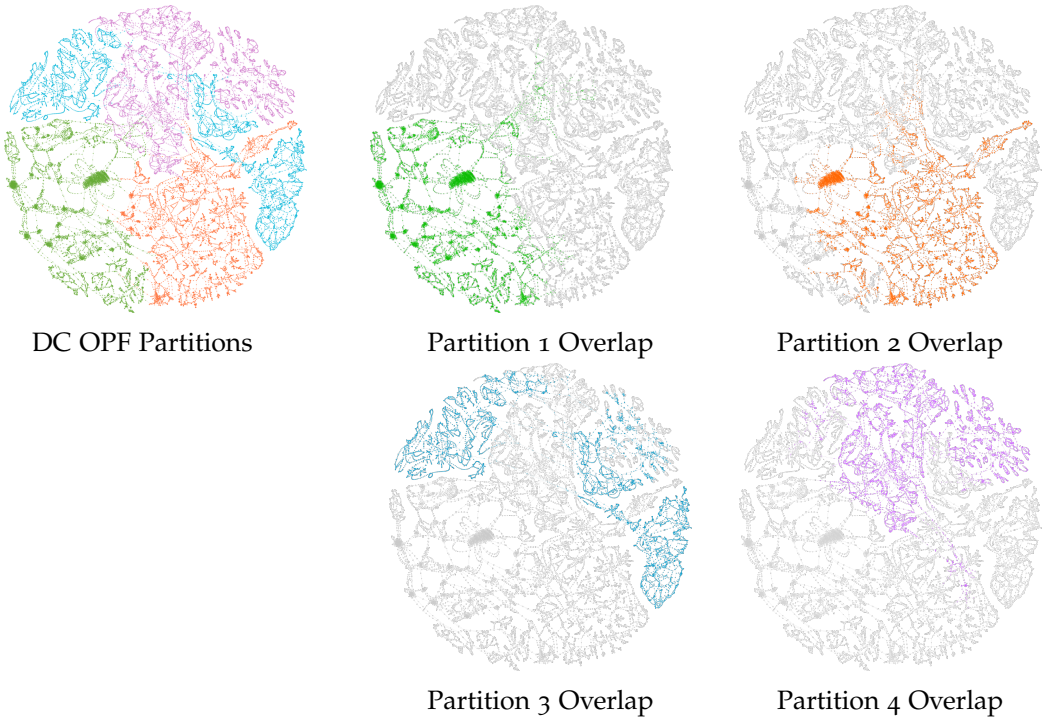


Figure 3.17: Depiction of DC OPF problem with four partitions. The original calculated partitions with $\epsilon_{max} = 0.1$ (left) and the corresponding overlap partitions with $\omega = 10$ (right).

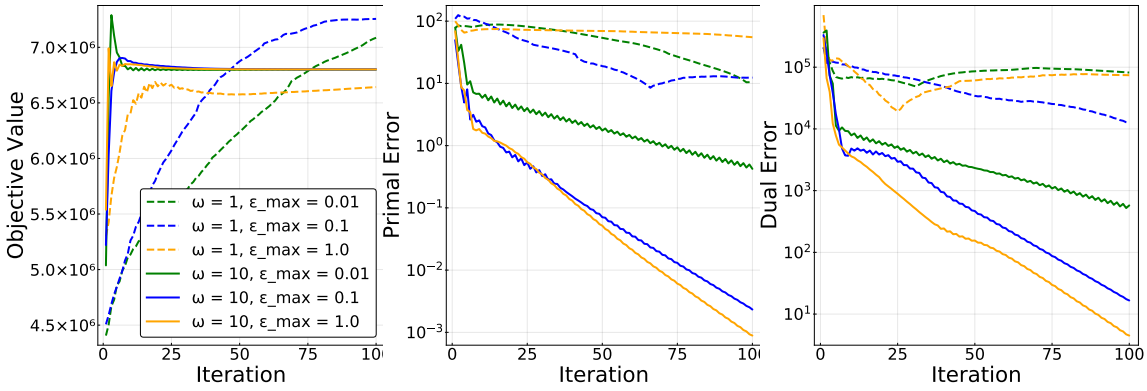


Figure 3.18: Comparison of Schwarz algorithm for different values of overlap ω and maximum partition imbalance ϵ_{max}

3.7 Appendix: DC OPF OptiGraph Implementation

The DC optimal power flow model is implemented in Code Snippet 3.9. Line 1 loads the bus system using data from the pglib-opf library, Line 4 creates the `OptiGraph`, Lines 7 and 8 create nodes for each bus and transmission line in the network, and Line 11 assigns relevant data to each bus and line node using a `load_data!` function. The DC OPF model is constructed in the same fashion as described in earlier examples where Lines 14 through 55 setup variables and constraints on each `OptiNode` and we add linking constraints to enforce power conservation and voltage angle connections.

Code Snippet 3.9: Creating the DC OPF Problem

```
include("ogplib_data.jl")

#Create graph based on network
grid = OptiGraph()

#Create bus and power line nodes
@optinode(grid,buses[1:N_buses])
@optinode(grid,lines[1:N_lines])

#Load data and setup bus and line mappings
load_data!(lines,buses)

#Setup line OptiNodes
for line in lines
    bus_from = line_map[line][1]
    bus_to = line_map[line][2]
    delta = line.ext[:angle_rate]
    @variable(line,va_i,start = 0)
    @variable(line,va_j,start = 0)
    @variable(line,flow,start = 0)
    @constraint(line,flow == B[line]*(va_i - va_j))
    @constraint(line,delta <= va_i - va_j <= -delta)
    @objective(line,Min,gamma*(va_i - va_j)^2)
end

#Setup bus OptiNodes
power_links = []
for bus in buses
    va_lower = bus.ext[:va_lower]; va_upper = bus.ext[:va_upper]
    gen_lower = bus.ext[:gen_lower]; gen_upper = bus.ext[:gen_upper]

    @variable(bus,va_lower <= va <= va_upper)
    @variable(bus,P[j=1:ngens[bus]])
    @constraint(bus,[j = 1:ngens[bus]]gen_lower[j] <= P[j] <= gen_upper[j])

    lines_in = node_map_in[bus] ; lines_out = node_map_out[bus]

    @variable(bus,power_in[1:length(lines_in)])
    @variable(bus,power_out[1:length(lines_out)])

    @constraint(bus,power_balance, sum(bus[:P][j] for j=1:ngens[bus]) -
    sum(power_in) + sum(power_out) - load_map[bus] == 0)
    @objective(bus,Min,sum(bus.ext[:c1][j]*bus[:P][j] +
    bus.ext[:c2][j]*bus[:P][j]^2 for j = 1:ngens[bus]))

    #Link power flow
    p1 = @linkconstraint(grid,[j = 1:length(lines_in)],
    bus[:power_in][j] == lines_in[j][:flow])
    p2 = @linkconstraint(grid,[j = 1:length(lines_out)],
    bus[:power_out][j] == lines_out[j][:flow])
    push!(power_links,p1) ; push!(power_links,p2)
end

#Link voltage angles
@linkconstraint(grid,angle_i[line = lines],line[:va_i] ==
line_map[line][1][:va])
@linkconstraint(grid,angle_j[line = lines],line[:va_j] ==
line_map[line][2][:va])
```

Chapter 4

MODELING LARGE-SCALE INFRASTRUCTURE SYSTEMS

This chapter uses the `OptiGraph` concepts presented in Chapters 2 and 3 to model and solve large-scale infrastructure optimization problems. We discuss existing challenges in modeling infrastructure systems and provide case studies that illustrate the benefits of the `OptiGraph` approach.

4.1 Introduction

The modeling and optimization of large-scale infrastructure systems is becoming increasingly important as their day to day operation becomes more complex. The dynamic behavior that has resulted from the continuous addition of intermittent generation sources (e.g. such as wind and solar) in the power grid and the increased stresses on power and gas systems due to natural and man-made hazards has necessitated flexible and scalable modeling approaches [102]. As an extreme example, the 2014 polar vortex exposed many operational inefficiencies and vulnerabilities in regional natural gas and power systems which resulted in failed natural gas deliveries to key power facilities which resulted in significant lost generation (35 GW at a value of lost load of 5000 $/MW h [3]). The need to create optimization models to design, operate, and analyze complex infrastructure networks is apparent, but such advances have remained technically challenging.

In the context of natural gas networks, optimization models need to capture both slow and fast transient effects using partial differential equations [26]. Such transient effects arise from sudden demand withdrawals that propagate throughout the network and consequently affect gas delivery to power plants and grid operations [118]. Developing large-scale gas network models is challenging because of the need to incorporate complex sets of equations over sophisticated pipeline networks and equipment that span thousands of miles. It is also important to consider long time horizons to develop robust control policies which lead to challenging large-scale nonlinear optimization problems.

Coupling infrastructure models poses yet another layer of complexity to manage. The equations that describe physical elements can be drastically different between systems which complicates the development of modeling tools to experiment with different model representations. For example, the power grid modeling field tends to use sophisticated simulation tools [99], but they lack the capabilities to integrate with high-fidelity gas pipeline simulators [1]). Interfacing different simulation tools becomes prohibitive because they often do not use coherent data structures. Such disparate structures result from the need to capture different physical equations and numerical techniques.

To tackle these issues, this chapter presents modeling approaches using the `OptiGraph` to model large-scale infrastructure systems. We show how the `OptiGraph` abstraction developed in Chapters 2 and 3 can be used to model coupled large-scale systems, and how it facilitates decomposition-based solutions for large-scale problems. We also show how it can interface with commercial tools to implement hybrid optimization strategies that produce high-fidelity solutions. The rest of this chapter is organized as follows. Section 4.2 presents a large-scale natural gas network we use for each of our considered problems, Section 4.3 uses the `OptiGraph` to model a coupled gas-electric power system, and Section 4.4 uses `OptiGraph` partitioning to decompose the space-time structure of the gas network to solve a a large-scale optimal control problem. Section 4.5 provides an overview of simulation-based optimization approaches to solve natural gas problems which are used in commercial packages, and Section 4.6 provides a case study that combines simulation-

based approaches with our developed `OptiGraph` models.

4.2 Natural Gas Optimization Model

Throughout the rest of this chapter we utilize the natural gas system shown in Figure 4.1 [29]. This system includes 4 gas supplies, 153 time-varying gas demands, 215 pipelines, and 16 compressor stations resulting in 172 total junction points. Our considered natural

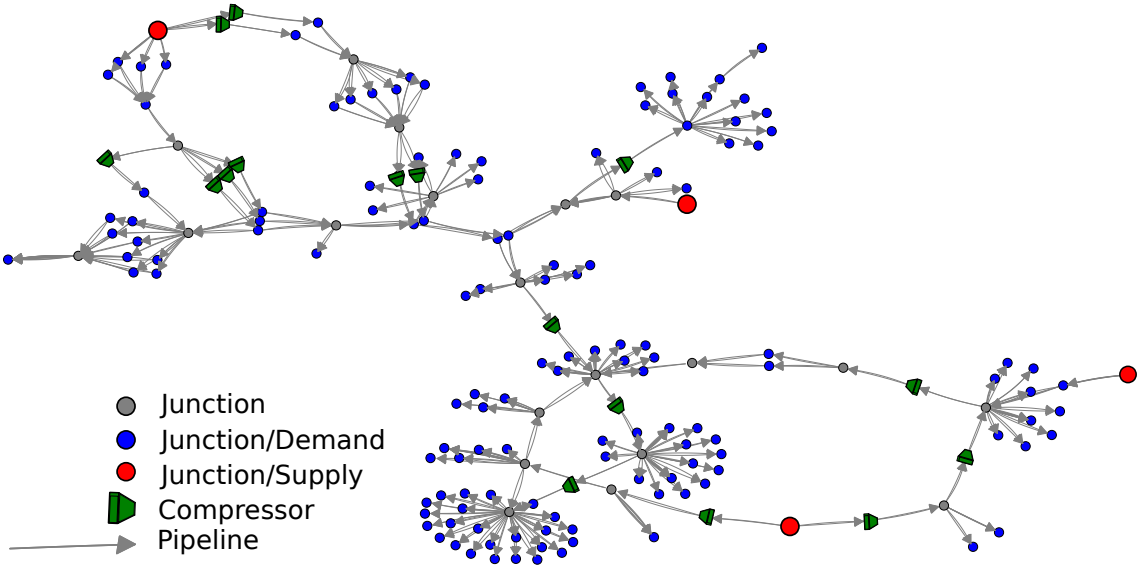


Figure 4.1: Depiction of Large-Scale Gas Network

gas system is governed by the equations described in Appendix A (and touched upon in Chapters 2 and 3), but we briefly present some of the notation here for convenience. The gas system can be modeled as a network of junctions and links $Net(\mathcal{J}, \mathcal{L})$. The set of junctions $j \in \mathcal{J}$ connect links and include gas supplies (injections) $\mathcal{S}_j$ and demands (withdrawals) $\mathcal{D}_j$. The set of links $\mathcal{L}$ is composed of both pipeline links $\mathcal{L}_p$ and compressor links $\mathcal{L}_c$ such that $\mathcal{L} := \mathcal{L}_p \cup \mathcal{L}_c$. We also specify the set of time periods as $\mathcal{T} := \{1, ..., 24\}$ for a 24 hour horizon. Our model implementation is analogous to the approach we used in Chapters 2 and 3. Each component of the system is modeled as a stand-alone `OptiGraph` with `OptiNodes` distributed over time and we connect them in a higher level

`OptiGraph` to form the complete network optimal control problem. We highlight again that this modular construction is a key benefit of the `OptiGraph` approach because it allows different infrastructure components to be developed separately (e.g., by different people).

Formulation (4.1) represents the optimal control problem we use throughout this chapter. (4.1a) denotes the objective function which seeks to maximize the total gas network revenue ϕ_{gas} by minimizing total compressor cost and maximizing target demand deliveries. Here, α_ℓ and $P_{\ell,t}$ are the compression cost (\$/kW), and compression power for compressor ℓ at time t, and α_d and $f_{d,t}^{target}$ are the demand price and target demand flow for demand d at time t. The optimal control problem includes terms which represent physical equations and operational constraints. Equation (4.1b) captures junction pressure limits, supply flows, and demand flows, (4.1c) describes compressor equations and limits, (4.1d) defines discretized pipeline equations for mass and momentum, (4.1e) denotes the initial condition of the partial differential equations, and (4.1f) requires the optimizer to refill the line-pack (gas inventory) in each pipeline at the end of the operating horizon. Equations (4.1g) and (4.1h) describe boundary conditions on each pipeline and mass conservation at each junction respectively. The details of the summarized equations are found in Appendix A.

$$\min_{\substack{\{\eta_{\ell,t}, f_{d,t}^{target}\} \\ \ell \in \mathcal{L}_c, d \in \mathcal{D}, t \in \mathcal{T}}} \quad \phi_{gas} := \sum_{\substack{\ell \in \mathcal{L}_c \\ t \in \mathcal{T}}} \alpha_\ell P_{\ell,t} - \sum_{\substack{d \in \mathcal{D} \\ t \in \mathcal{T}}} \alpha_d f_{d,t}^{target} \tag{4.1a}$$

s.t. Junction Limits (A.1) (4.1b)

Compressor Equations (A.3) (4.1c)

Pipeline Equations (A.21) (4.1d)

Initial Condition (A.22) (4.1e)

Refill Line-Pack (A.23) (4.1f)

Boundary Conditions (A.29) (4.1g)

Junction Conservation (A.28) (4.1h)

4.3 Case Study: Coordinated Gas and Electric Systems

This case study demonstrates how an `OptiGraph` can be used to model and solve a large-scale coupled gas-electric infrastructure system.

4.3.1 *Problem Overview*

For our study, we take the regional natural gas system presented in Section 4.2 (and depicted in Figure 4.1) and we *couple* it to a regional electric power grid that consists of 2522 transmission lines, 1908 buses, 870 electric loads, and 225 generators (of which 153 are gas-fired). The power grid operator seeks to solve the optimal power flow problem

given by (4.2).

$$\min_{\substack{\{P_{q,t},v_{i,t}\}\\ q\in\Omega,i\in\mathcal{V},t\in\mathcal{T}}} \quad \phi_{grid} := \sum_{\substack{q\in\Omega\\ t\in\mathcal{T}}} \alpha_{q,t} P_{q,t} \tag{4.2a}$$

$$s.t. \quad P_{\ell,t} = Y_\ell(v_{\ell,t,j} - v_{\ell,t,k}), \quad \ell \in \mathcal{L}_g,\ t \in \mathcal{T} \qquad \text{(Power Flow)} \tag{4.2b}$$

$$\underline{P_q} \le P_{q,t} \le \overline{P_q}, \quad q \in \Omega,\ t \in \mathcal{T} \qquad \text{(Generation Limits)} \tag{4.2c}$$

$$-\overline{v_\ell} \le v_{\ell,j,t} - v_{\ell,k,t} \le \overline{v_\ell}, \quad \ell \in \mathcal{L}_g,\ t \in \mathcal{T} \qquad \text{(Voltage Angle Limits)} \tag{4.2d}$$

$$v_{i,t} = v_{i,t}^{ref}, \quad i \in \mathcal{V}^{ref},\ t \in \mathcal{T} \qquad \text{(Reference Buses)} \tag{4.2e}$$

$$\sum_{q\in\Omega_i} P_{q,t} + \sum_{\ell\in\mathcal{L}_{rec}(i)} P_{\ell,t} - \sum_{\ell\in\mathcal{L}_{snd}(i)} P_{\ell,t} = P_{i,t}^L,\ i \in \mathcal{V},\ t \in \mathcal{T} \qquad \text{(Bus Conservation)} \tag{4.2f}$$

$$v_{\ell,j,t} = v_{src(\ell),t}, v_{\ell,k,t} = v_{dst(\ell),t}, \quad \ell \in \mathcal{L}_g,\ t \in \mathcal{T} \qquad \text{(Line Connections)} \tag{4.2g}$$

This problem is also called the direct-current (DC) optimal power flow problem and is almost equivalent to the formulation in Section 3.6 except we have added the time periods $t \in \mathcal{T}$ and use a simpler objective function. In this problem, we simply seek to minimize the total generation cost over the time horizon subject to meeting all of the power loads by manipulating power generation and voltage angles $P_{q,t}$ and $v_{i,t}$.

The two problems described by (4.1) and (4.2) can be coupled at gas generators with the equations given by (4.3).

$$f_{d,t}^{target} = f_{d(q)}^{grid}, \quad d \in \mathcal{D}_{gas}, \quad t \in \mathcal{T} \qquad \text{(Couple Demands)} \tag{4.3a}$$

$$f_{d(q)}^{grid} \le f_{d,t}, \quad d \in \mathcal{D}_{gas}, \quad t \in \mathcal{T} \qquad \text{(Limit Generator Output)} \tag{4.3b}$$

Here, $\mathcal{D}_{gas}$ is the set of gas-fired generators in the power grid network and $f_{d(q)}^{grid}$ is the generator demand target for demand d. Hence, (4.3a) couples the demand targets, and (4.3b) limits the generator capacity based on the actual delivered demand flows. Figure 4.2 visually depicts the complexity that is involved in coordinating the two systems. In the left figure we show how tightly integrated the coupled networks are, and in the right figure we show the information that is exchanged between two potential network

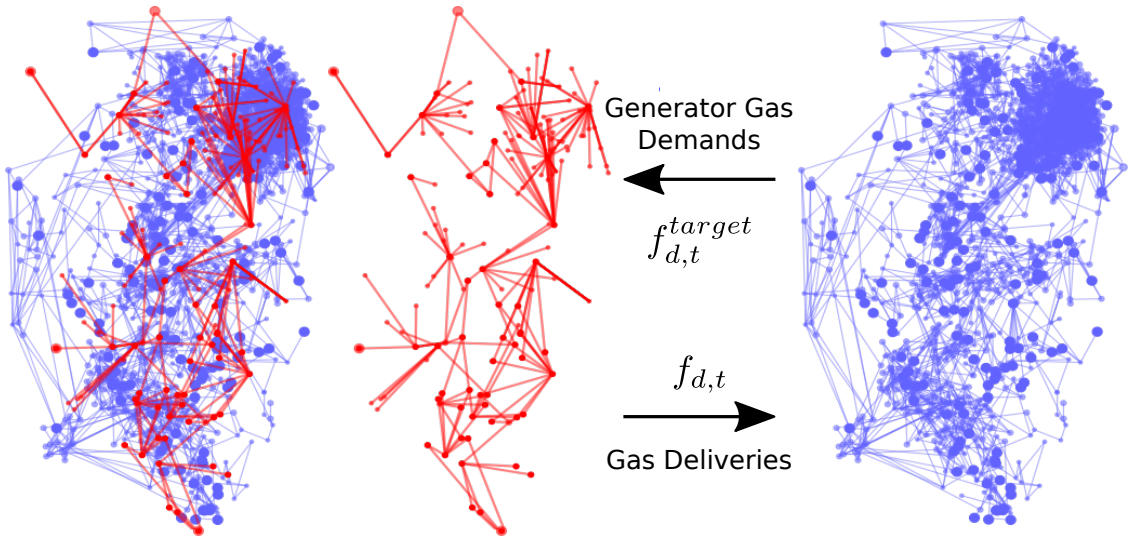
operators.

For this study, we consider three primary settings to evaluate coordination strategies between the two systems. These are:

- (I) **Uncoupled**: The power system solves (4.2) and communicates gas generator demands ($\{f_{d,t}^{target}\}_{d\in\mathcal{D}_{gas},t\in\mathcal{T}}$) to the natural gas operator. The natural gas operator seeks to make deliveries and balance system line-pack by solving (4.1).

- (II) **Data Exchange**: We obtain the solution of setting (I), but after we solve (4.1), we communicate the deliveries that the gas operator is able to make to gas generators ($\{f_{d,t}\}_{d\in\mathcal{D}_{gas},t\in\mathcal{T}}$) back to the power operator. The power operator resolves (4.2) given the available gas.

- (III) **Fully Coupled**: We solve the linked formulation given by coupling (4.2) and (4.1) with (4.3). In this formulation, the natural gas generators are effectively treated as dispatchable loads to the gas operator.



(a) Coupled system (b) Data exchange between systems

Figure 4.2: Coupled gas and electric infrastructure systems. The red network corresponds to the natural gas system and blue network corresponds to the power grid system.

4.3.2 *Implementation*

To model the coupled system we use an `OptiGraph` wherein we represent each individual infrastructure system as a *self-contained* subgraph. Code Snippet 4.1 shows how we can use `Plasmo.jl` to create the coupled system corresponding to setting (III). In fact, a key benefit of using `Plasmo.jl` is that it naturally facilitates coupling different infrastructure systems [67]. In this snippet, Line 2 creates an `OptiGraph` which we call `combined_system` and we then add separate subgraphs that separately model the natural gas and grid networks. On Line 7 we create a new expression called `Pgend` in the `grid_network` `OptiGraphs` which converts generator power generation (in mega-watts) to natural gas demand (in mass flow units). Lines 10 through 16 use `OptiGraph` linking constraints to couple the two systems at the higher level `combined_system` graph. We note that we use some helpful expressions defined for each system subgraph (e.g. `fdemands` and `fdeliver`).

Code Snippet 4.1: Creating a fully coupled infrastructure problem in `Plasmo.jl`

```
1  #Create an OptiGraph for the combined natural gas and electric systems
2  combined_system = OptiGraph()
3  add_subgraph!(combined_system,gas_network)
4  add_subgraph!(combined_system,grid_network)
5
6  #Add expression to the natural gas demands for power grid generators
7  @expression(grid_network,Pgend[t=1:nt,q=1:n_gens],pow_to_gas*grid_network[:Pgen][q,t])
8
9  #Link generator gas demands to the natural gas system
10 for d in gas_network[:demands]
11     q = demand_map[d]
12     @linkconstraint(combined_system,[t = 1:nt], gas_network[:fdemands][d,t]
13                                                 == grid_network[:Pgend][q,t])
14     @linkconstraint(combined_system,[t = 1:nt], grid_network[:Pgend][q,t]
15                                                 <= gas_network[:fdeliver][d,t])
16 end
17 #Optimize the combined system
18 optimize!(combined_system,ipopt)
```

4.3.3 *Results*

Figure 4.3 shows time profiles for the requested and delivered gas demands for two generators under the uncoupled (I) setting (left figure), the data exchange (II) setting (middle figure), and the fully coupled (III) setting (right figure). In setting (I) we observe

that the natural gas demands from the grid system system (shown by the blue line) cannot be met by the gas deliveries (shown by the green line). However, if the delivery shortfalls are communicated back to the grid operator, we see that the gas-fired generators have adjusted their gas demands and re-optimized their operation based on actual realized gas delivery. We lastly solve the more computationally intensive fully coupled problem (according to Code Snippet 4.1) and observe that all of the original gas demands are met as shown in the right side of Figure 4.3. In fact, the fully coupled setting results in no gas shortfalls. We highlight that the fully coupled problem can be warm-started using the solution from the decoupled problem following the approaches given in Chapter 2. and this can be used to solve the coupled problem robustly.

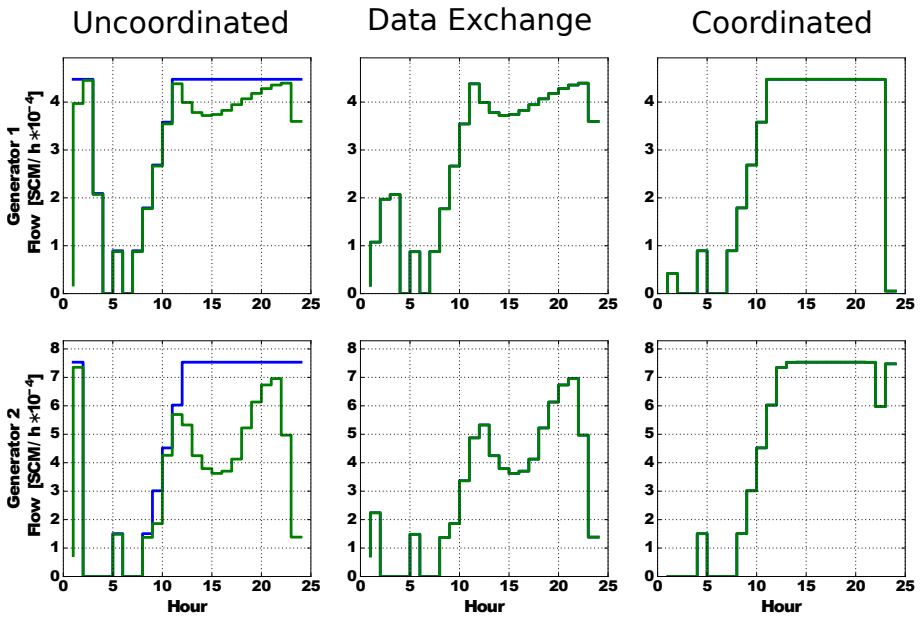


Figure 4.3: Sample generator gas demands (blue) and realized gas delivery (green) for uncoupled setting (I), data exchange setting (II), and fully coupled setting (III).

4.4 Case Study: Space-Time Decomposition of a Large-Scale Natural Gas Network

A key aspect of complex infrastructure models (such as the coupled system in the previous study) is that they exhibit the *spatiotemporal structures* we discussed in Chapter 3. Such structures can effectively be exploited in computation, but the main challenge is obtaining a suitable decomposition that can be communicated to a parallel solver. This case study extends upon the study presented in Section 3.5 and more rigorously explores partitioning strategies for the optimal control problem over the natural gas system in Figure 4.1.

4.4.1 *Problem Overview*

Now that we have constructed an `OptiGraph` representation of the natural-gas network optimal control problem (described by Equation (4.1)) we can use the hypergraph partitioning approaches presented in Section 3.2 to explore various decomposition strategies and their effect on computational performance. We formulate the optimal control problem to track time-varying demand withdrawal and minimize compression costs over a 24-hour time horizon. The optimization problem we produce contains 432048 variables, 427512 equality constraints, and 3887 inequality constraints. Capturing the space-time structure of Formulation (4.1) is seemingly complex but it is straightforward to do so with an `OptiGraph` because each pipeline can be treated independently.

Figure 3.2b depicts the graph structure induced by the space-time nature of this problem. Our goal is to identify efficient partitions that traverse space-time to efficiently solve the problem using Schur decomposition in `PIPS-NLP` (as described in Section 3.3.1). For our implementation, each component of the system is modeled as a stand-alone `OptiGraph` with `OptiNodes` distributed over time and we connect these in a high-level graph to form the complete problem. We use the capabilities of `Plasmo.jl` to experiment

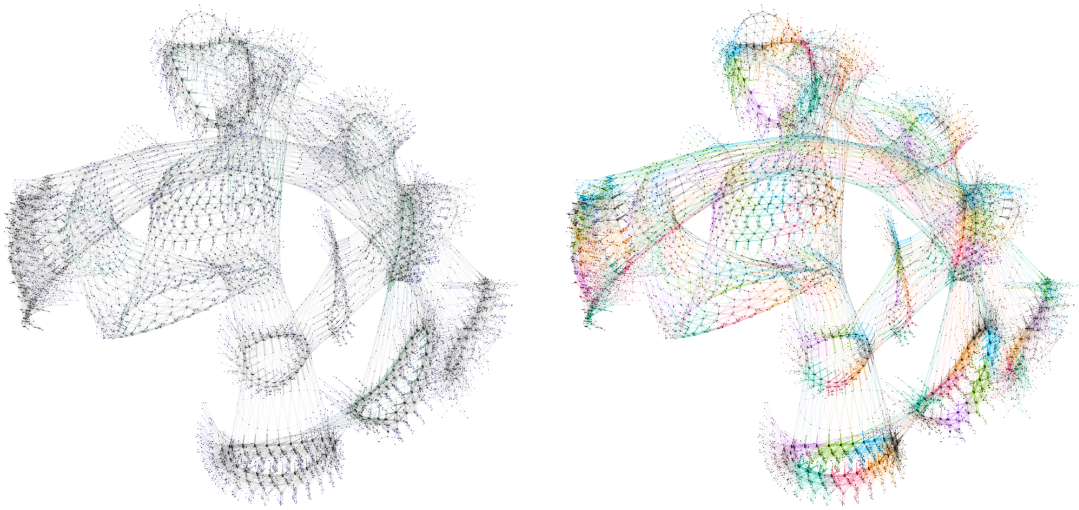
with different partitioning strategies and with this analyze trade-offs between coupling, imbalance, and memory use.

4.4.2 *Implementation*

For our implementation, each component of the system is modeled as a stand-alone `OptiGraph` with `OptiNodes` distributed over time and we connect these in a high-level graph to form the complete problem. We refer to the Code Snippets A.1, A.2, and A.3, and A.4 in Appendix A which detail how each component model is constructed and connected.

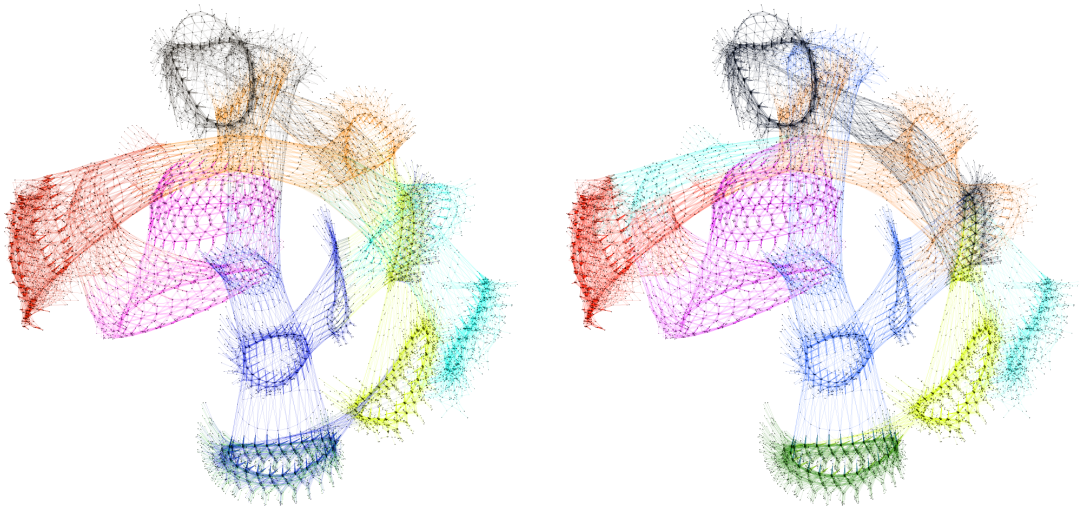
Figure 4.4 visualizes the graph structure of the optimal control problem and shows different partitions that can be obtained. Figure 4.4a shows the `OptiGraph` components we assembled in the above code snippets with pipeline nodes depicted in grey, compressor nodes in green, and junctions in blue. Figure 4.4b depicts a pure time partition of the problem with 8 partitions (each with a different color). Partitioning in time is a reasonable approach to the optimal control problem, but we will see that the produced partition is intractable to solve using a general Schur complement approach. This problem can also be partitioned purely as a network wherein we consider the partition of the network components themselves (as opposed to the structure of the optimal control problem) which is shown in Figure 4.4c. The network partition is physically intuitive but it does not capture the true mathematical structure of the problem nor consider computational aspects.

We highlight that both time and network partitioning approaches can be performed using the partitioning framework (by manually defining a partition vector), but the value of the `OptiGraph` is that we can efficiently obtain space-time partitions to produce the partition shown in Figure 4.4d. While visually similar to the network partition, the space-time partition produces considerably less coupling (748 linkconstraints versus 1800) which is advantageous since the number of linkconstraints corresponds to the size of the matrix in



(a) Modeled Components

(b) Time Partitions



(c) Network Partitions

(d) Space-Time Partitions

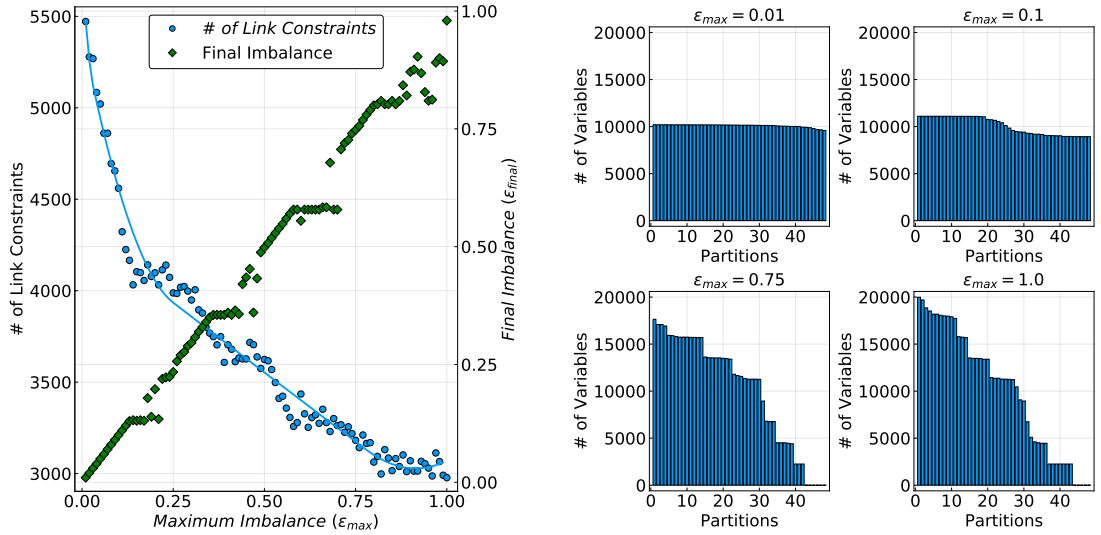
Figure 4.4: Graph depictions of the natural gas network optimal control problem. The graph is colored by the physical network components (top left), by 8 time partitions (top right), by 8 network partitions (bottom left), and by 8 space-time partitions (bottom right).

Equation (3.4b). Moreover, the pure time partition produces considerable coupling (over 80000 links) which makes Schur decomposition infeasible.

We perform partitioning in the same way we did in Code Snippet 3.6 to produce new `OptiGraphs` with partitions we can distribute and solve with `PIPS-NLP`. To solve with `PIPS-NLP` we follow the exact same setup used in Snippet 3.7 using `PipsSolver` and distribute the `OptiGraph` between workers to solve in parallel. We experiment with different numbers of partitions and imbalance values and explore how the `PIPS-NLP` algorithm performs. Table 4.1 details the results obtained where we vary the number of partitions $|\mathcal{P}|$ (8, 24, and 48) and the maximum imbalance value ϵ_{max} (between 0.01 and 1.0). Figure 4.5 shows the effect of increasing the maximum imbalance for the 48 partition case on the true final imbalance the partitioner produced (ϵ_{final}) and the total number of linkconstraints. For the most part the maximum and final imbalances ϵ_{final} display a one-to-one relationship but there are distinct intervals wherein the final imbalance flattens out. We also see that nominal values of maximum imbalance (less than 25%) reduce the number of linkconstraints considerably after which greater values produce diminished decreases. We also show the distribution of subproblem sizes for a few select maximum imbalance values in Figure 4.5b for reference.

4.4.3 *Results*

For each run we note the true imbalance `KaHyPar` produced ϵ_{final}, the sum of the edge weights $sum(w(\mathcal{E}))$ (which corresponds to the number of linkconstraints), the maximum node size $max(\{s(n)\}_{n\in\mathcal{N}(\mathcal{OG})})$ (which corresponds to the node with the most variables), as well as the average and maximum number of node connections (i.e. the number of linkconstraints incident to a node). For timing results we observe the time spent building the KKT system t_{build} (i.e. the time to build Equation (3.4b)), the time spent factorizing the Schur Complement matrix t_{fac} (i.e. the time to solve Equation (3.4b)) and the total time spent inside `PIPS-NLP` t_{pips}.



(a) (b)

Figure 4.5: Partitioning results with 48 partitions. Imbalance versus the number of linkconstraints and final imbalance (left) and the distribution of subproblem sizes for select imbalance parameters (right).

The first rows of Table 4.1 show results with 8 partitions and also include the cases for the time and network partitions corresponding to Figures 4.4b and 4.4c. As expected, the high degree of coupling in these partitions is computationally prohibitive: the time partition can not solved (with Schur complement decomposition) and the network partition requires over 2 hours. Significant improvement is achieved using the `KaHyPar` partitioner where even a 1% imbalance is over twice as fast as the network partition. Increasing the imbalance parameter to 50% results in better performance (the average and maximum number of columns in B_n decreases), but increasing it to 60% results in diminished speed, despite producing a similar partition. Interestingly, allowing too much maximum imbalance can result in ill-conditioned subproblems and motivates investigating other graph metrics that might inform the partitioner.

The second and third sets of rows present the results using 24 and 48 partitions with 24 CPUs. Increasing the maximum imbalance for the 24 partition case results in diminished algorithm performance (due to the bottleneck building the KKT system). This is because either the node connectivity increases (for the 10% imbalance), the maximum

Partition	ϵ_{max}	ϵ_{final}	(# Links)	(Largest Node)	(# Incident Links) mean , max	t_{build} (sec)	t_{fac} (sec)	t_{pips} (sec)
$\mid \mathcal{P} \mid = 8$ # Proc = 8	time	0.0	87505	60567	21877 , 24940	-	-	-
	network	0.41	1800	85248	459 , 1368	7236	56	7505
	0.01	0.0073	748	61008	205 , 316	2944	6.3	3115
	0.5	0.37	434	83202	121 , 218	2269	1.7	2475
	0.6	0.37	458	83202	124 , 218	2862	1.9	3069
$\mid \mathcal{P} \mid = 24$ # Proc = 24	0.01	0.01	2889	20390	259 , 431	1746	273	2117
	0.1	0.1	2748	22200	256 , 529	1934	236	2270
	1.0	0.99	1572	40080	127 , 362	2279	54	2447
$\mid \mathcal{P} \mid = 48$ # Proc = 24	0.01	0.01	5472	10194	245 , 482	2029	1769	3954
	0.1	0.1	4560	11104	210 , 440	1871	1031	3054
	0.75	0.75	3182	17642	151 , 387	2126	368	2670
	1.0	0.98	2978	19985	143 , 480	1826	298	2247

Table 4.1: PIPS-NLP results for different problem partitions

subproblem size increases (for the 100% imbalance), or there is some ill-conditioning of the subproblems. In contrast, increasing the imbalance for the 48 partition case results in computational improvement. This is because more linking constraints (more overall coupling) shifts the bottleneck step to factorizing the Schur complement and drastic speedups result from reducing the degree of coupling.

4.5 Comparison with Simulation-Based Approaches

This section addresses some of the limitations that arise using the presented optimization models in this chapter. Thus far, we demonstrated `OptiGraph` capabilities to create coupled infrastructure models and to experiment with different partitioning approaches. The gas network models we have formulated adhere to what is called *direct-transcription* (DT) optimization. Within this paradigm, the physical dynamics (e.g. our PDE equations for mass and momentum) are encoded as constraints within the optimization problem which is solved with a general optimization solver (such as `Ipopt` or `PIPS-NLP`). To make our DT models more tractable from a computational standpoint, we made considerable physical assumptions (e.g. ideal gas assumptions, constant friction factors, no flow reversals). Such assumptions are common in DT approaches, but they can lead to uncertainty about model fidelity. In contrast, typical industry approaches in infrastructure modeling (and

specifically in gas network modeling) use high fidelity models using simulation-based (SB) approaches. SB approaches however, can suffer from long computation times and creating SB models is often a nontrivial and intensive task.

The rest of this section provides a summary of DT and SB approaches and introduces *hybrid* strategies that interface our presented optimization approaches with a high-fidelity simulation-based framework. We discuss the key benefits of hybrid optimization strategies and how they enable experimentation with diverse model formulations (e.g. using the `OptiGraph`) that can be refined using high fidelity simulation-based tools.

4.5.1 *Simulation-Based Optimization*

Industrial optimization approaches tend to target what is called the *simulation-based (SB)* paradigm. within this paradigm, successive simulations are coupled with optimization strategies to find optimal operating plans. A key aspect of this approach is that simulation tasks can harness high-fidelity simulators and perform optimization separately. SB approaches also build upon existing simulation packages that contain rich feature sets (and that often devote considerable effort in describing the functionality needed for accurate simulation with real equipment).

The key concept behind SB approaches is to perform repeated simulations of a dynamic model with different decision/input values (e.g. controller values) to identify a set of decisions that minimizes/maximizes a cost function (e.g. minimize energy useage or maximize system revenue) where the trial inputs are updated using a derivative-based or derivative-free optimization solver. The SB approach is intuitive, but repeated solutions of large complex dynamic systems can become computationally expensive and simulations (using time-stepping solvers) can fail at poor trial control values (e.g., that lead to non-physical states). In addition, techniques to compute derivatives are often limited in practice to computation of single gradients rather than full constraint Jacobians.

Equation (4.4) represents the general SB formulation, also depicted in Figure 4.6. In

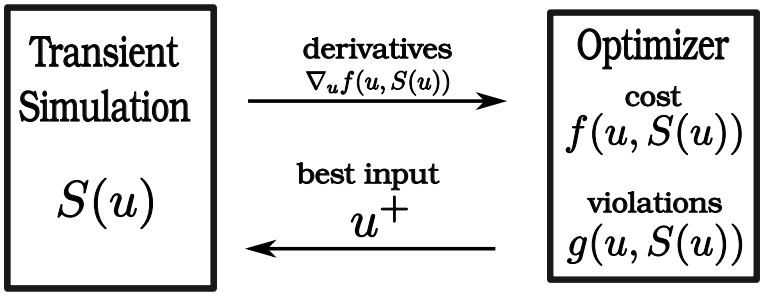


Figure 4.6: Representation of information exchanged in one iteration of a simulation-based optimizer.

this formulation $S(u)$ represents the result of a simulation (e.g. the physical state of the system x) given inputs u, $f(u, S(u))$ represents the cost associated with the simulation result, and $g(u, S(u)$ represents resulting violations. The simulation results are communicated to an optimizer which can use derivative information to calculate new inputs u^+ that it communicates back to the simulator.

$$\min_u f(u, S(u)) \qquad \text{(Cost)} \tag{4.4a}$$

$$s.t. \quad lb \leq g(u, S(u)) \leq ub \qquad \text{(Limits)} \tag{4.4b}$$

4.5.2 *Direct Transcription Optimization*

The `OptiGraph` models we present in this chapter adhere to the DT optimization paradigm. In this approach, we seek to find the inputs and physical states *simultaneously* that minimize/maximize a cost function according to Formulation (4.5).

$$\min_{x,u} f(x, u) \qquad \text{(Cost)} \tag{4.5a}$$

$$s.t. \quad lb \leq g(x, u) \leq ub \qquad \text{(Limits)} \tag{4.5b}$$

$$h(x, u) = 0 \qquad \text{(Dynamic equations)} \tag{4.5c}$$

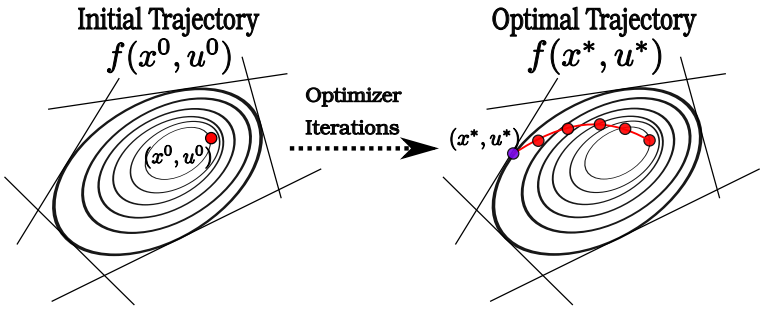


Figure 4.7: Representation of iterations performed using direct transcription with an interior-point solver such as `Ipopt`.

Here, u represents inputs/decisions and the states x (physical states of the system) are now included directly within the optimization formulation. The term (4.5a) evaluates the system cost given the state and input trajectories, the expression (4.5b) represents limits on the inputs and states (e.g. physical limits of equipment), and the expression (4.5c) explicitly encodes the discretized simulation dynamics as a constraint within the optimization problem. The DT approach can be described as a series of internal iterations from an initial state (possibly non-physical) and iterates to find an optimal solution that minimizes the cost and that satisfies the dynamics and constraints all-at-once. This procedure is summarized in Figure 4.7. Our `OptiGraph` formulations of our infrastructure models in this chapter adhere to the description of Formulation (4.5).

In contrast to the SB approach, DT approaches (especially using an `OptiGraph`) offer more modeling flexibility and the potential to add otherwise intractable features (such as integer decision variables) to nonlinear physical equations. However, as discussed, DT approaches usually require the use of simplified physics models, fixed time steps, or ideal thermodynamic properties. Such fidelity limitations tend to be considered unacceptable in many simulation contexts (e.g. when trying to capture fast transient effects).

4.5.3 *Hybrid SB-DT Optimization*

SB and DT approaches can be used to solve much the same problems. For instance, both approaches can be used to compute policies for operational plans over long horizons and

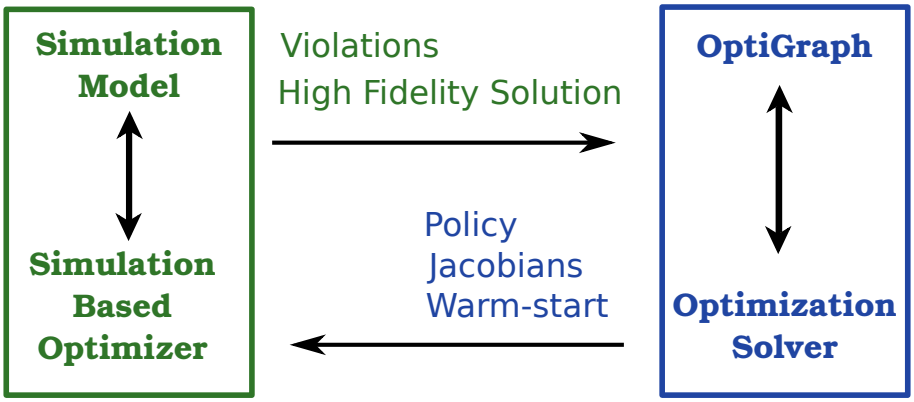


Figure 4.8: Depiction of information exchange between an `OptiGraph` and a simulation-based optimizer.

can include time-varying effects. While SB approaches can obtain higher fidelity, they can take considerable time to find solutions that satisfy operational constraints. In contrast, DT approaches are more generalizable in the types of constraints and variables that they can handle (e.g. integer decisions).

The proposed hybrid optimization strategies combine SB and DT approaches and specifically make use of the `OptiGraph` model management capabilities to facilitate complex data workflows. For example, solutions obtained from DT can be communicated to a SB optimizer for high-fidelity verification and refinement to reduce constraint violations that result from simplifications. A SB optimizer can provide violation information to a DT model to guide the selection of constraint back-off terms, or it can provide high fidelity solution information to perform linearization strategies on a per-component basis (this is greatly facilitated using an `OptiGraph`). Constraint Jacobians can also be computed for specific sets of equipment and given to the SB optimizer to aid in obtaining tighter constraint satisfaction in fewer iterations. Figure 4.8 depicts the types of information an `OptiGraph` (or a general DT framework) might exchange with a simulation-based framework.

4.6 Case Study: Hybrid Optimization for a Large-Scale Natural Gas Network

This section concludes the chapter by showing how DT optimization approaches (using the `OptiGraph`) can benefit in a hybrid optimization setting. We show how the hybrid setting benefits from both the generalizable qualities inherent to DT approaches and the high-fidelity afforded by SB.

4.6.1 *Problem Overview*

We use the optimal control model described by (A.21) for our natural gas network (Figure 4.1) for which we refer to as the *DT-NLP* model for this study. To make the problem more interesting from an operational standpoint we increase the optimization horizon to 72 hours ($\mathcal{T} := \{1, ..., 72\}$) which creates a multi-day operational problem.

We additionally explore a discrete-decision optimization model that incorporates binary decision variables (compressors can be on or off). Discrete-decision models have garnered considerable interest in gas network optimization, but they require solving a mixed-integer nonlinear program (MINLP) which is intractable for large-scale problems. As such, we develop a reduced linearized natural gas model which we call the *DT-MILP* model described by Formulation (4.6). The details of the linearization for this model can be found in Section A.4 in Appendix A. Within the hybrid setting, we seek to quantify the performance of the DT-MILP model to implement discrete decisions.

$$\min_{\substack{\{\eta_{\ell,t}, f_{d,t}^{target}\} \\ \ell \in \mathcal{L}_c, d \in \mathcal{D}, t \in \mathcal{T}}} \quad \phi_{gas} := \sum_{\substack{\ell \in \mathcal{L}_c \\ t \in \mathcal{T}}} \alpha_\ell P_{\ell,t} - \sum_{\substack{d \in \mathcal{D} \\ t \in \mathcal{T}}} \alpha_d f_{d,t}^{target} \tag{4.6a}$$

s.t.	Junction Limits	(A.1)	(4.6b)
	Supply Pressure Ramping	(A.2)	(4.6c)
	Discrete Compressor Equations	(A.4)(A.5)(A.8)	(4.6d)
	Compressor Ramping	(A.7)(A.5)	(4.6e)
	Linearized Pipeline Equations	(A.27)	(4.6f)
	Boundary Conditions	(A.29)	(4.6g)
	Junction Conservation	(A.28)	(4.6h)

4.6.2 *Implementation*

For our implementation we create both DT-NLP and DT-MILP models using the `OptiGraph` with approximate (coarse) demand forecasts (blue lines in Figure 4.9) and solve with `Ipopt` and `Gurobi` respectively. We use coarsened forecast data to pose computational tractable models over the long time horizon. For the pure SB strategy, we fix the gas deliveries based on true (fine) demand forecasts (green lines in Figure 4.9) and seek to minimize the compressor work subject to our optimization constraints (junction pressures and line-pack targets) and solve with the `Synergi Gas GTO` commercial SB optimizer. The SB optimizer makes consecutive simulation runs and calculates new compressor controls at each iteration (as depicted by Figure 4.6). For each hybrid strategy, we pass compressor discharge setpoints from the corresponding DT solution (which again uses coarsened demand forecasts) to the SB optimizer (which uses fine demand forecasts and higher fidelity modeling with more realistic physics). Using these setpoints, the SB optimizer first computes a physically accurate solution that corrects all the modeling simplifications of the DT model, and then computes any remaining operational violations. In this case study

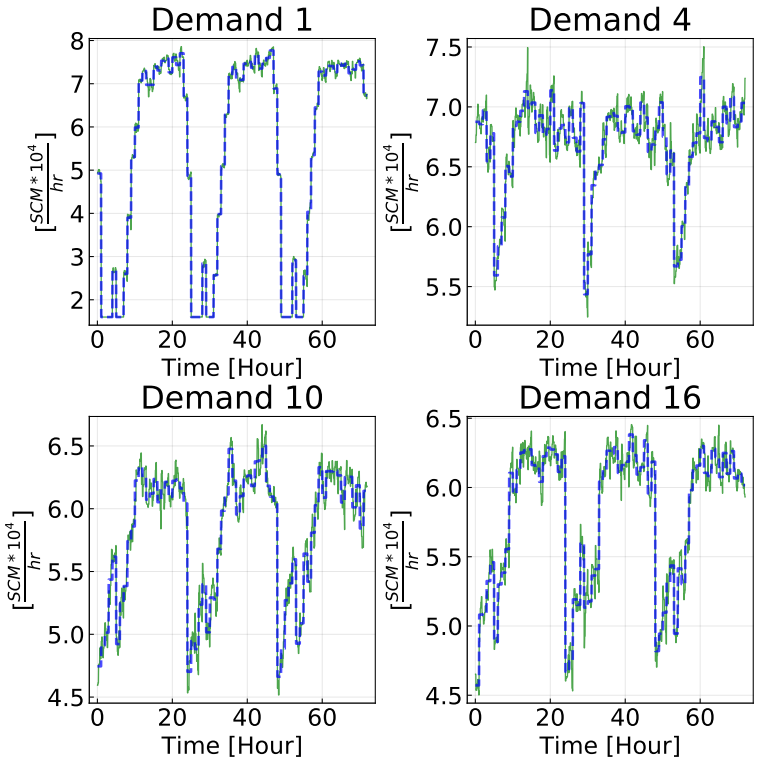


Figure 4.9: Select forecasted demand profiles used for hybrid optimization study. The true (high-resolution) demands (solid green lines) and approximate (coarsened) demands (dashed blue lines).

the hybrid strategy ends at this point, but the SB optimizer could be allowed to further adjust compressor controls to improve the solution with respect to any violations found.

To create the linearized pipeline model (for the DT-MILP setting) we use a steady-state `OptiGraph` model (A.26) with added time-coupling, and we use Jacobian information to linearize the model around each space-time discretization point to retain as much model accuracy as possible (again, the details are given by (A.21)).

4.6.3 *Results*

An overview of the simulation results are given in Table 4.2. Figure 4.10 also compares the produced pressure violations (for all 172 junctions over the time horizon) for each optimization strategy and shows that no setting produced extreme violations. We first

Strategy	Total Work (kW-hr)	Line-Pack Change (%)	Max Violation (bar)	Demand Shortfall (%)	Total Time (minutes)
SB	4.48E5	+0.58	2.41	0	172 (GTO)
DT-NLP	7.23E5	+1.26	-	0	8.1 (Ipopt)
Hybrid-NLP	6.44E5	+1.40	1.03	0	9.9 (8.1 Ipopt + 1.8 GTO)
DT-MILP	6.56E5	+2.10	-	1.93	38 (Gurobi)
Hybrid-MILP	8.05E5	-0.44	7.58	0	41.8 (38 Gurobi) + (3.8 GTO)

Table 4.2: Overview of Hybrid Optimization Results

draw attention to the purely SB result which produces the lowest total work (4.48E5 kW-hr), but it requires considerable computation time because the `GTO` optimizer has to perform many time-consuming high-fidelity simulations. The SB result also produces noticeable pressure violations. To achieve the least overall work the pure SB strategy exploits compressor bypass (reverse flow through compressors) which the DT strategies do not incorporate (flow reversal is challenging to model in the DT setting). However, the SB optimizer does not support minimum power limits (this is a challenging constraint to implement in SB optimization) which are helpful from an operational standpoint and the DT strategies incorporate this constraint almost trivially.

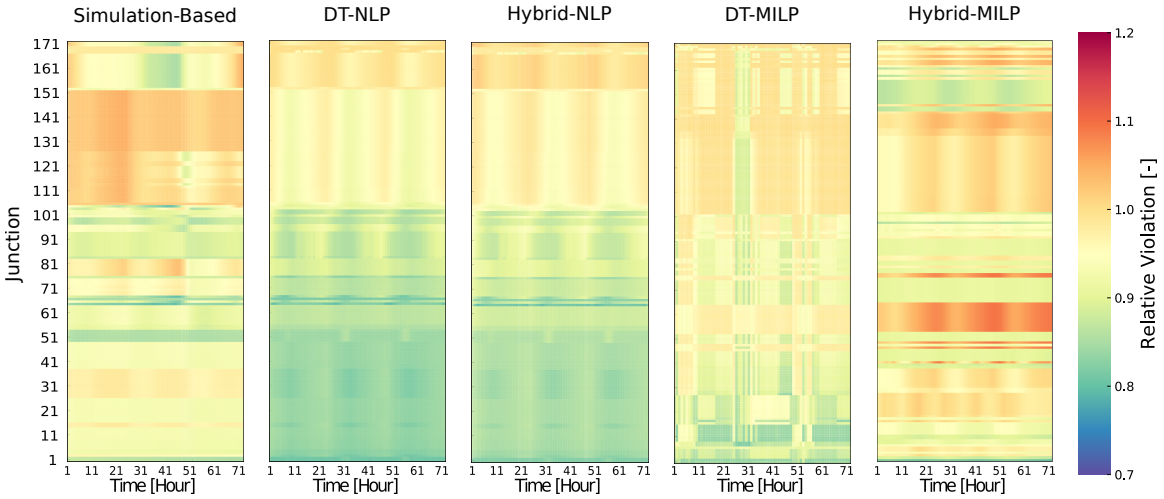


Figure 4.10: Comparison of pressure violations.

The DT-NLP result requires drastically less computation time (about 8 minutes with `Ipopt`), and produces no violations by design (constraints have to be satisfied in a DT optimization problem). The Hybrid-NLP verifies the DT-NLP solution using the SB op-

timizer which only requires an additional two minutes. Notably, the hybrid strategy produces very small violations (about 1 bar at the worst) and the compressor operation is remarkably comparable to the DT-NLP result as depicted in Figure 4.11. The agreement demonstrates benefits of the hybrid strategy. The DT-NLP model can be easily modified, and it's assumptions are reasonable enough such that the SB optimizer can find a high fidelity solution fairly quickly.

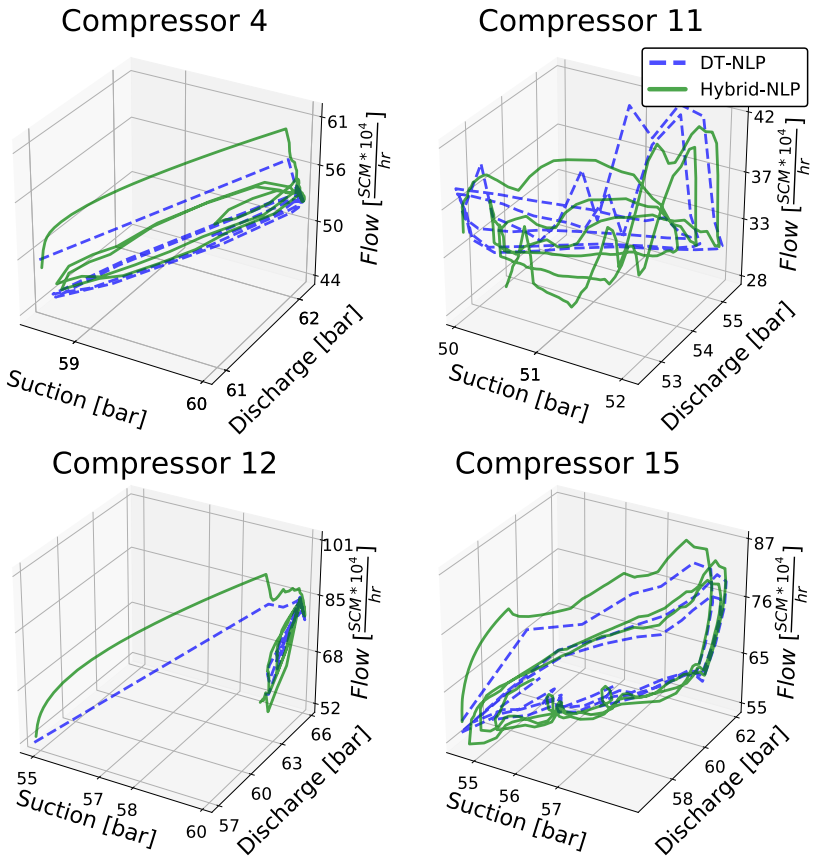


Figure 4.11: Comparison of path profiles for DT-NLP (dashed blue lines) and Hybrid-NLP (solid green lines) strategies for select compressors.

We finally look at the Hybrid-MILP strategy and see that the results are quite encouraging. The DT-MILP model produces the desirable discrete behavior which is tracked by the corresponding Hybrid-MILP strategy as shown in Figure 4.12. The DT-MILP produces demand shortfalls (failed gas deliveries) but Hybrid-MILP is able to leverage system line-pack to up the deliveries. The Hybrid-MILP requires considerable computation

time (it took `Gurobi` 38 minutes to close to a 1% gap), and reveals noticeable violations, but enhancements could certainly be made to the DT model to improve these results.

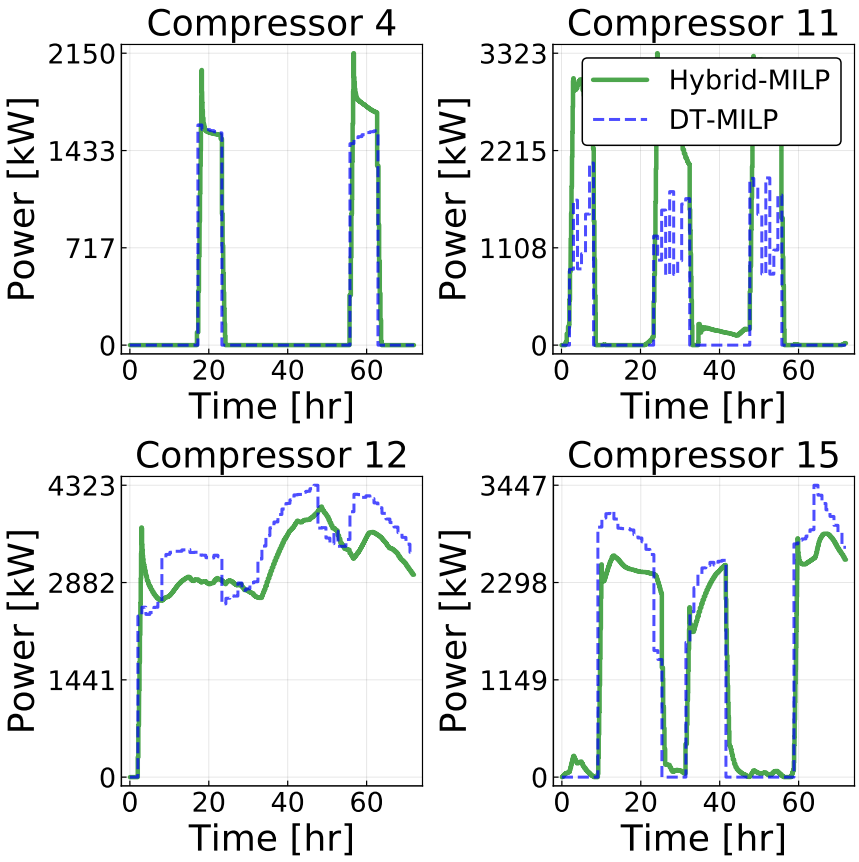


Figure 4.12: Verification of discrete compressor decisions using hybrid strategy for select compressors.

Chapter 5

GRAPH-BASED MODELING FOR CYBER SYSTEMS

Thus far, Chapters 2, 3, and 4 have focused on modeling the physical aspects (specifically modeling optimization problems) that arise in cyber-physical systems. This chapter introduces an abstraction to model and simulate *cyber systems* which we call the `ComputingGraph`.

5.1 Introduction

Simulating cyber systems requires capturing dynamic and logical aspects that arise in real-time decision-making such as time delays, computing/processing latency and failures, and asynchronicity. For example, we might be interested in predicting how a given control system will perform on an architecture that contains sensors and computing devices with limited processing capabilities (and thus long delays) or how a distributed optimization algorithm will perform on a distributed-memory computing cluster compared to a single central processing unit (CPU). Communication aspects are particularly challenging to capture in cyber systems, since they often involve complex topologies and communication frequencies (call back to Figure 1.3).

As mentioned in Chapter 1, the most widely used tools to simulate cyber-physical aspects are `Simulink` and agent-based modeling platforms. The `ComputingGraph` offers

advantages over `Simulink` and agent-based tools in that it handles cyber features such as communication delays, latency, and synchronous/asynchronous computing and information exchange in a more coherent fashion. This chapter presents `ComputingGraph` concepts and is organized as following. Section 5.2 introduces the `ComputingGraph`, discusses its underlying representation, and shows how the `ComputingGraph` captures the cyber aspects in cyber-physical systems. Section 5.3 presents the corresponding software framework called `PlasmoCompute.jl` and provides a helpful example to demonstrate its syntax. Section 5.4 provides an interesting case study in distributed model predictive control to illustrate the `ComputingGraph` capabilities, and Section 5.5 provides extra case study details.

5.2 Computing Graphs

This section presents the basic elements of the `ComputingGraph` and shows how the abstraction facilitates the modeling and simulation of cyber systems.

5.2.1 *Representation*

A `ComputingGraph` is a *directed* multigraph (remember Figure 1.4c) that we denote as $\mathcal{CG}(\mathcal{N}, \mathcal{E})$ and that contains a set of `ComputeNodes` $\mathcal{N}(\mathcal{CG})$ (which perform computing tasks) and `CommunicationEdges` $\mathcal{E}(\mathcal{CG})$ (which communicate attributes between nodes).

A `ComputeNode` $n \in \mathcal{N}(\mathcal{CG})$ contains a set of *attributes* $\mathcal{A}_n$ and computing *tasks* $\mathcal{T}_n$. The attributes $\mathcal{A}_n$ represent node data and tasks $\mathcal{T}_n$ are computational procedures that operate on and/or change attributes. In other words, *a computing task maps attributes* (a task takes attribute data and processes it to create other attribute data). This interpretation resembles that of a manufacturing process, which takes raw material to generate products. Each task $task \in \mathcal{T}_n$ in the `ComputingGraph` requires a given execution time $\Delta\theta_{task}$. A `CommunicationEdge` $e \in \mathcal{E}(\mathcal{CG})$ contains a set of attributes $\mathcal{A}_e$ associated with its support nodes $\mathcal{N}(e)$. Communicating attributes between nodes involves a communication delay

$\Delta\theta_e$. The collection of computing and communication tasks comprises an *algorithm* (also known as a computing workflow in the computer science community). Consequently, a `ComputingGraph` seeks to facilitate the *creation and simulation of algorithms*.

The `ComputingGraph` contains a *global clock* t_{CG} and each node has an internal *local clock* t_n. The clocks are used to manage and schedule computing tasks and communication. For any *task* executed at time t, its attributes become updated at clock time $t + \Delta\theta_{task}$. Likewise, for any edge e that communicates its attribute at time t, the destination attribute value is updated with the source attribute value at time $t + \Delta\theta_e$. Under the proposed abstraction, computing and communication tasks can be *synchronous* (a task is not executed until all attributes are received) or *asynchronous* (a task is executed with current values). This enables capturing a wide range of behaviors seen in applications.

The computing and communication times $\Delta\theta_{task}$ and $\Delta\theta_e$ can represent *true* times (times required by the computing devices executing the tasks) or *virtual* times (times required by hypothetical devices executing tasks). In other words, the proposed abstraction allows the simulation of algorithms on virtual (hypothetical) computing architectures. This is beneficial when we lack access to actual sophisticated computing architectures (such as a large-scale parallel computer or an industrial control system) but we wish predict how an algorithm will execute under such architecture. Moreover, this allows us to analyze the behavior of algorithms under extreme events that might involve communication or computation failures and with this test their resilience.

Both nodes and edges use the concept of *state managers* to manage task behavior (e.g., determining when a task has been completed) and to manage communication (e.g., determining when data is sent or received). This representation resembles that used in process scheduling and has interesting connections with automata theory and discrete event simulation [4, 2]. These connections can be exploited to derive a coherent *state-space* representation (we discuss this further in Section 5.2.3).

Figure 5.1 depicts an example `ComputingGraph` containing three nodes and six edges. Each node contains a single task which takes local attribute values x, y, and z as input and

updates one of their values. For example, $task_{n_1}$ processes its attributes and updates the value of attribute y. The nodes communicate attribute values with each other using the six edges. For instance, attribute y is communicated to both nodes n_2 and n_3 which updates the value of y on these respective nodes. We use the superscript $+$ to denote that the attribute may be updated after a given time (to capture computing and communication delays).

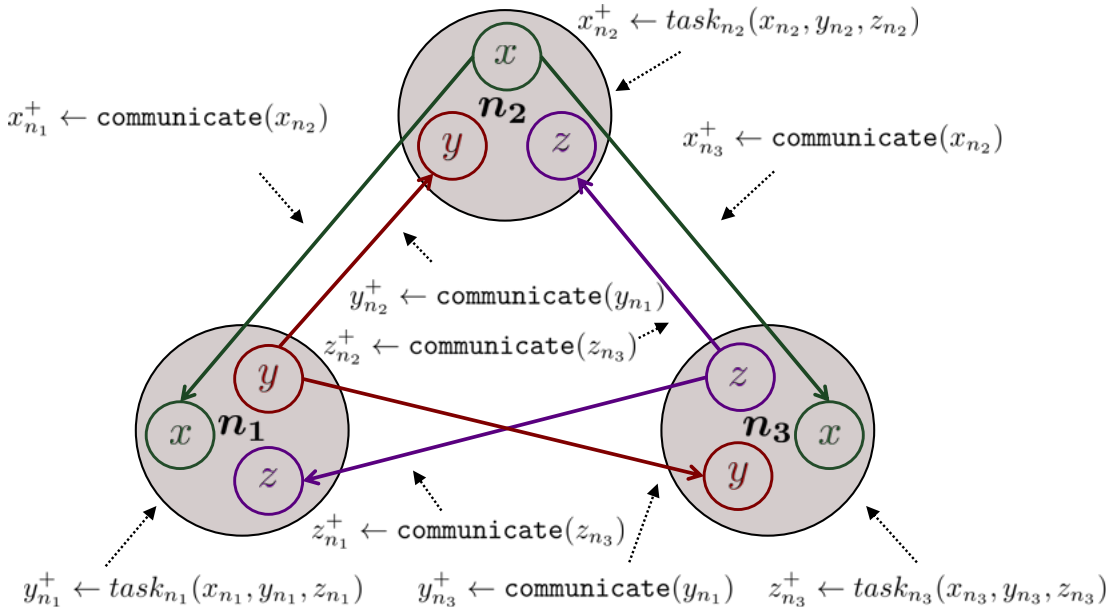


Figure 5.1: Depiction of a `ComputingGraph` with three nodes and six edges. Node n_1 computes $task_{n_1}$ using the data attributes (x, y, and z) and updates the value of attribute y. Similarly, node n_2 computes $task_{n_2}$ and updates attribute x, and node n_3 computes $task_{n_3}$ and updates attribute z. Attribute values are communicated between nodes using edges.

5.2.2 *Connections with OptiGraphs*

It is helpful to highlight the differences and synergies between a `ComputingGraph` and the `OptiGraph` presented in Chapters 2 and 3. In a `ComputingGraph`, nodes contain a dynamic components (computing tasks) while in an `OptiGraph`, nodes contain a static components (algebraic models). Moreover, in a `ComputingGraph` an edge connects attributes (dynamically) while in an `OptiGraph` an edge connects algebraic variables (statically). Under a

ComputingGraph abstraction, the *solution of an* OptiGraph is considered a computing task. Consequently, a ComputeNode might use an OptiGraph to perform an optimization task or a ComputingGraph might be an algorithm for solving a given OptiGraph. For instance, for the former, we might create a ComputingGraph that executes a control algorithm and use an OptiGraph to model the physical system under the actions of the control system. For the latter, we might create a ComputingGraph that executes a solution algorithm (e.g., Bender decomposition) to solve OptiGraph. These capabilities enable the simulation of complex cyber-physical systems. We also highlight that computing tasks are general and might involve procedures that go beyond the solution of OptiGraphs such as forecasting, data analysis, learning, solving optimization problems (that are not expressed as graphs), and so on.

5.2.3 *State-Space Description*

The computation and communication logic provided by the ComputingGraph is driven by an underlying state manager abstraction wherein transitions in task states are triggered by input signals. This representation conveniently captures computing and communication latency and is flexible and extensible. In this section, we provide details on how state managers facilitate the implementation of ComputingGraphs.

Node and edge managers use *states* and *signals* to manage task computation and attribute communication. The use of managers is motivated by state machine abstractions from automata theory [21, 18], which are often used in control-logic applications [39]. State machines are also used to manage logical behaviors in Simulink [22] and agent-based simulation frameworks [98]. State machines can be used to represent actions that trigger transitions in task states. Figure 5.2 presents a simple state machine with three states, three signals, and five possible transitions between states.

In a ComputingGraph, a node manager M_n oversees the state of node tasks. Such tasks are specified by the user in the form of *functions*. Each node has an associated tuple

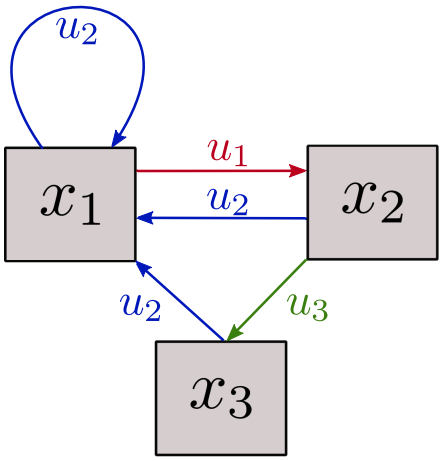


Figure 5.2: A simple state machine with three states (x_1, x_2, x_3), three action signals (u_1, u_2, u_3), and five possible state transitions.

$(\mathcal{X}_n, \mathcal{U}_n, \mathcal{Y}_n, f_n, g_n)$ where $\mathcal{X}_n$ is the set of states, $\mathcal{U}_n$ is the set of action signals, $\mathcal{Y}_n \subseteq \mathcal{A}_n$ is the set of broadcast (output) attributes, $f_n : \mathcal{X}_n \times \mathcal{U}_n \to \mathcal{X}_n$ is the state transition mapping, and $g_n : \mathcal{Y}_n \to \mathcal{Y}_n$ is the attribute update mapping. The node dynamic evolution is represented as a system of the form:

$$x_n^+ = f_n(x_n, u_n) \tag{5.1a}$$

$$\eta_n^+ = g_n(\eta_n). \tag{5.1b}$$

Here, the next state $x_n^+ = f_n(x_n, u_n)$ is the result of the mapping from the current state x_n and a given action signal u_n. Every state transition $(u_n, x_n) \to x_n^+$ also triggers a transition in the attribute values $\eta_n \to \eta_n^+$ (i.e., tasks update attributes). Action signals can also be sent to the state managers of other nodes in the form of attributes.

The proposed abstraction can incorporate an arbitrary number of states and actions but here we provide an example of a basic set of states and actions that can be considered

in an actual implementation:

$$\mathcal{X}_n := (\texttt{idle}, \texttt{executing\_task}, \texttt{finalized\_task}) \tag{5.2a}$$

$$\mathcal{U}_n := (\texttt{execute\_task}, \texttt{finalize\_task}, \texttt{attribute\_updated}, \texttt{attribute\_received}). \tag{5.2b}$$

At any given point in time, a node manager can be in one of the states in $\mathcal{X}_n$. The set of signals recognized by the manager are $\mathcal{U}_n$ and these will trigger (depending on the current state) a transition between states. Such signals include, for instance, `execute_task` or `attribute_received`. The set of broadcast targets (that receive created signals) are the node itself and all of its outgoing edges $\mathcal{E}(n)$. Hence, a node can send signals (in the form of attributes) to itself or its outgoing edges.

Using the sets defined in (5.2), we can define a transition mapping $f_n(\cdot)$ as:

$$\texttt{executing\_task} \leftarrow (\texttt{idle}, \texttt{execute\_task}) \tag{5.3a}$$

$$\texttt{finalized\_task} \leftarrow (\texttt{executing\_task}, \texttt{finalize\_task}) \tag{5.3b}$$

$$\texttt{idle} \leftarrow (\texttt{finalized\_task}, \texttt{back\_to\_idle}). \tag{5.3c}$$

In (5.3) we can see, for instance, that a task transitions to the `executing_task` state when it receives the corresponding `execute_task` signal and it transitions to the `finalized_task` state when it is executing a task and receives a signal to finalize such task. The signals to execute or finalize a task are generated by user-defined attributes. For instance, a user-defined attribute consisting of a flag such as `convergence` or `max_iterations` can generate a finalize task signal that in turn triggers a state transition.

An edge manager M_e can be defined in the same way as a node manager with associated states and signals for communication. A possible implementation of an edge

manager includes the following states and actions:

$$\mathcal{X}_e := (\texttt{idle}, \texttt{communicating}, \texttt{all\_received}) \tag{5.4a}$$

$$\mathcal{U}_e := (\texttt{attribute\_updated}, \texttt{communicate}, \texttt{attribute\_sent}, \texttt{attribute\_received}). \tag{5.4b}$$

An edge can send signals to itself or its supporting nodes $\mathcal{N}(e)$ in the form of its attributes. Using the sets defined in (5.4), we can define a transition mapping $f_e(\cdot)$ of the form:

$$\texttt{communicating} \leftarrow (\texttt{idle}, \texttt{communicate}) \tag{5.5a}$$

$$\texttt{all\_received} \leftarrow (\texttt{communicating}, \texttt{all\_received}) \tag{5.5b}$$

$$\texttt{idle} \leftarrow (\texttt{all\_received}, \texttt{back\_to\_idle}). \tag{5.5c}$$

The mappings in (5.5) closely reflect the node transition mapping in (5.3). An edge transitions to the communicating state when it receives a `communicate` signal and transitions to the `all_received` state when it receives the `all_received` signal (indicating that all sent attributes were received).

Figure 5.3 depicts the node and edge manager transition mappings (5.3) and (5.5) with additional transitions. This figure highlights that action signals can trigger self-transitions wherein nodes or edges loop back to their original state. For instance, a node can transition back to the `executing_task` state when it receives the signal `attribute_updated` and an edge can transition back to the `communicating` state when it receives either `attribute_sent` or `attribute_received` signals. Self-transitions allow attribute updates to occur during task execution or edge communication.

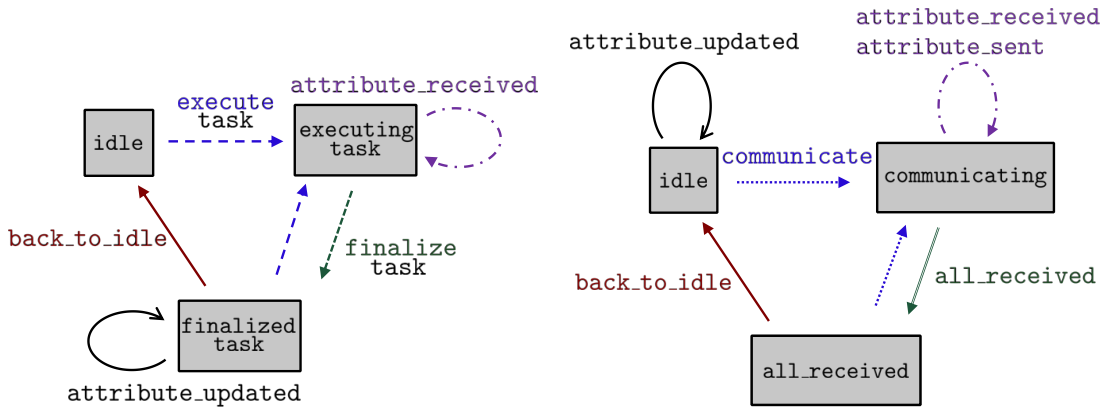


Figure 5.3: Implementation of a node manager M_n (left) and an edge manager M_e (right). Action signals trigger transitions between states and can include transitions that return to the same state.

5.2.4 *Task Scheduling and Timing*

A `ComputingGraph` implementation needs to capture timings and order of task execution and attribute communication. These timings can be managed using a discrete-event queue wherein items in the queue are evaluated in an order based on their scheduled time [44]. The `ComputingGraph` specifically uses a *signal-queue* wherein action signals are evaluated based on their scheduled evaluation time.

Figure 5.4 illustrates an example execution of the `ComputingGraph` from Figure 5.1. Node n_1 computes $task_{n_1}$ (which requires compute time $\Delta\theta_{task_{n_1}}$) after which the value of attribute y is sent to the corresponding attribute y on nodes n_2 and n_3 (which each requires communication time). The compute and communication times are expressed using signals. For instance, when $task_{n_1}$ completes its execution, it produces a `finalize` $task_{n_1}$ signal with a delay $\Delta\theta_{task_{n_1}}$ to capture the computation time. Equivalently, when edge e_1 that connects node n_1 to node n_2 communicates attribute y, it produces the y`_received` signal with a delay $\Delta\theta_{e_1}$.

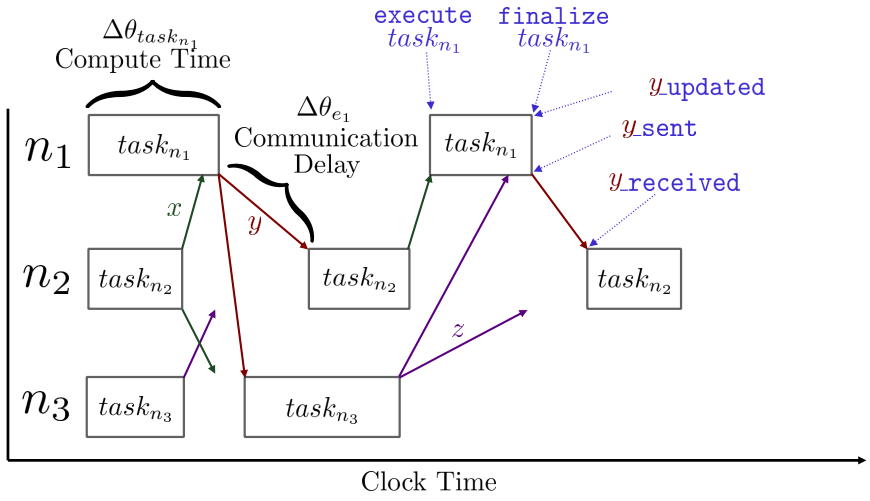


Figure 5.4: An example execution of the `ComputingGraph` in Figure 5.1. Computing times and communication delays are captured using action signals.

5.3 Software Framework: PlasmoCompute.jl

This section introduces `PlasmoCompute.jl`, a `Julia`-based software package that implements the `ComputingGraph` abstraction to simulate cyber system behaviors. We introduce the basic syntax and functions required to create a `ComputingGraph` simulation and provide a simple example that simulates a centralized model predictive control system.

5.3.1 *Overview of Modeling Functions*

Table 5.1 highlights the key functions and macros used used to implement `ComputingGraph` capabilities. After creating a new `ComputingGraph` object (using a `ComputingGraph()` constructor), the provided macros can be used to setup topology, computational tasks, and communication. The `@computenode` macro is used to create `ComputeNodes` after which we can assign attributes and computing tasks to the node using the `@attributes` and `@computetask` macros. The communication topology can created using the `@connect` macro which creates `CommunicationEdges` between attributes.

Other functions are used to manage a `ComputingGraph` simulation. For instance `schedule!`

Function	Description
`cg = ComputingGraph()`	Create a new `ComputingGraph` object.
`@computenode(cg::ComputingGraph,expr::Expr)`	Create a `ComputeNode` using an expression
`@attributes(node::ComputeNode,expr::Expr)`	Create attributes on `node` using an expression
`@computetask(node::ComputeNode,expr::Expr)`	Create a `ComputeTask` on `node` using an expression
`@connect(node::ComputeNode,expr::Expr)`	Create a `CommunicationEdge` between attributes
`getcomputenodes(cg::ComputingGraph)`	Retrieve `ComputeNodes` in `cg`
`getcommedges(cg::ComputingGraph)`	Retrieve CommunicationEdges in `cg`
`value(attribute::ComputeAttribute)`	Retrieve current value of `attribute`
`getcurrenttime(cg::ComputingGraph)`	Retrieve the current clock time of `cg`
`schedule!(cg::ComputingGraph,task::Computetask,time)`	Schedule an execute signal on `task` at `time`
`step!(cg::ComputingGraph)`	Evaluate the next signal in `cg`
`execute!(cg::ComputingGraph,stop_time::Float64)`	Execute `cg` until `stop_time`
`plot_trace(cg::ComputingGraph)`	Plot the trace output of `cg`

Table 5.1: Overview of *ComputingGraph* construction and management functions in *PlasmoCompute.jl*.

queues an `execute` signal to occur at a prescribed global clock time, `step!` advances the simulation by evaluating the next signal in the queue, and `execute!` executes the simulation to the prescribed `stop_time`.

5.3.2 *Example: Simulating Centralized Control of a Reactor System*

To demonstrate basic `ComputingGraph` syntax using `PlasmoCompute.jl` we pose a simple model predictive control example for a two-reactor system with a separator. The reactor setup is depicted by Figure 5.5 [125] and the details of the system are given in Sections 5.4 and 5.5. For this problem, we seek to simulate the *real-time* performance of a centralized controller that attempts to stabilize the system after changing plant-wide operating setpoints. We denote the key behavior of our simulation setup with Computing Graph 5.1 which is presented in the same spirit we would do so for an algorithm (i.e. a `ComputingGraph` is something that can be *executed* like an algorithm).

For our setup we denote the `ComputeNode` *pl* (the plant) which contains the tasks and attributes to simulate the reactor system. The attributes consist of the plant state x and the injected control input u_{pl}. The node executes a single task `run_plant` which runs continuously as a result of the trigger: Updated(x) (i.e. the task reschedules every time it advances the reactor state). We also denote a single node called *mpc* which contains tasks

and attributes corresponding to a single plant-wide MPC controller. The controller node contains the attributes for the control action (u_{mpc}) and the current measurement (y_{mpc}) as well as a single task that computes the control action `control_action` which is triggered every time it receives a new measurement.

The `CommunicationEdges` e_1 and e_2 represent the attribute connections between the two nodes. Edge e_1 connects the plant state x_{pl} to the controller measurement y_{mpc} where communication is triggered continuously (every time the edge evaluates the Sent(x_{pl}) signal and waits θ_{wait}). Edge e_2 connects the controller input attribute u_{mpc} to the plant input u_{pl} which is triggered every time the controller updates u_{mpc}.

Computing Graph 5.1 Centralized Control

Plant Node (pl)
Attributes: $\mathcal{A}_{pl} := (u_{pl}, x)$
Tasks: $\mathcal{T}_{pl} :=$ (`run_plant`)
  `run_plant`: Task 5.3, triggered by: Updated(x)

MPC Controller (mpc)
Attributes: $\mathcal{A}_{mpc} := (u_{mpc}, y_{mpc})$
Tasks: $\mathcal{T}_{mpc} :=$ (`control_action`, `receive_policy`)
  `control_action`: Task 5.4, triggered by: Received(y)

Edge $e_1 := x_{pl} \rightarrow y_{mpc}$ send on: Sent(x_{pl}), wait: θ_{wait}
Edge $e_2 := u_{mpc} \rightarrow u_{pl}$, send on: Updated($u_{mpc}$)

Computing Graph 5.1 can be implemented in `PlasmoCompute.jl` as depicted in Code Snippet 5.1. Line 2 creates the `ComputingGraph` object (`graph`), Lines 5 through 8 setup the plant node (pl) with attributes and its compute task, and Lines 11 through 13 setup the mpc node and add attributes and tasks. Lines 16 through 19 connect the attributes between the two nodes and setup activation triggers, Line 22 executes the `ComputingGraph`, and Line 25 plots a trace of the first 300 seconds of execution.

Code Snippet 5.1: Simulating Centralized Control with `PlasmoCompute.jl`

```
#Create a computing graph
graph = ComputingGraph()

#Create plant simulation node
@computenode(graph,pl)
@attributes(pl,x,u)
@computetask(pl,run_plant,triggered_by=Updated(x))
schedule!(graph,run_plant,time = 0) #initialize computing graph

#Create MPC controller node
@computenode(graph,mpc)
@attributes(mpc,u_inject,y)
@computetask(mpc,control_action,triggered_by=Received(y))

#Connect plant to MPC controller
@connect(graph,pl[:x] => mpc[:y], delay=30, send_on = Sent(pl[:x]), send_wait=60,
    start=5)

#Connect MPC controllers to plant
@connect(graph,mpc[:u_inject] => pl[:u], delay=30, send_on=Updated([mpc[:u]))

#Execute computing graph
execute!(graph,5000)

#Plot the trace of the execution
plt = plot_trace(graph,0:300)
```

The results of the centralized controller setup are depicted in the next section (Section 5.4). Figure 5.7a shows the first 300 seconds of execution and 5.7b shows that the temperature of both reactors and the separator is successfully stabilized despite the real-time communication and computational delays.

5.4 Case Study: Simulating Cooperative Control

This case study demonstrates how a `ComputingGraph` can be used to simulate the behavior of a distributed control architecture that includes computation and communication delays. The key objective of this study is to simulate how such delays impact the actual behavior of the physical reactor system as well as how communication and controller *failures* might realize in the real-time system.

5.4.1 *Problem Setup*

We again consider the reactor-separator system in Figure 5.5 which is a standard application for evaluating distributed model predictive control (MPC) algorithms. The system consists of two reactors in series where the reaction $A \rightarrow B \rightarrow C$ takes place and a separator which separates out a product stream and recycle. The system is described by twelve states: the weight fractions of A and B in each unit, the unit heights, and the unit temperatures. The manipulated inputs are the flow rates F_{f1}, F_{f2}, F_1, F_2, F_3, F_R, and heat exchange rates Q_1, Q_2, and Q_3. Once again, the details of the system are given in Section 5.5

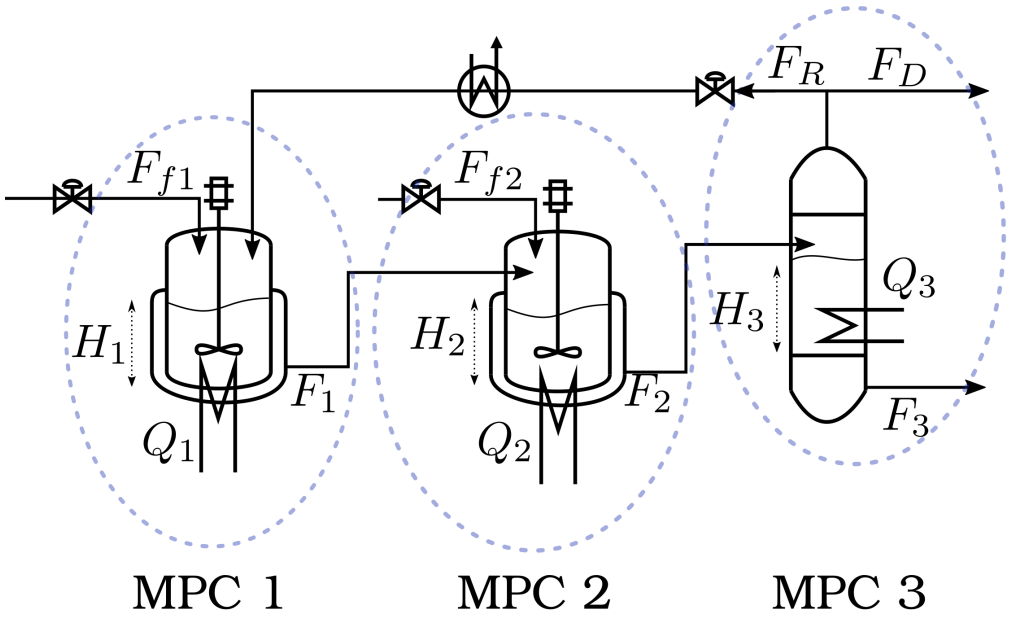
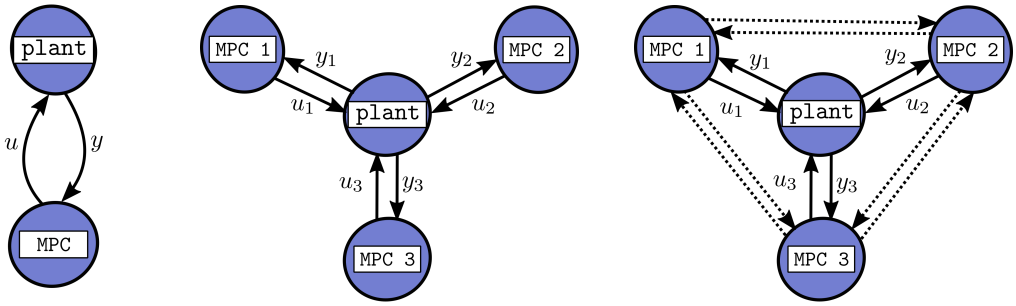


Figure 5.5: Reactor separator process and partitioning into MPC controllers.

The behavior of the physical reactor system (the plant) shown in Figure 5.5 is simulated under three different MPC architectures. We first consider the *centralized* MPC architecture (Figure 5.6a) which we evaluated in Section 5.3.2. As discussed earlier, in this architecture every output is sent to a central MPC controller which computes all control actions for the plant. We also consider a *decentralized* control architecture that consists of three MPC controllers (one for each unit) and simulate their behavior when they do

not communicate (Figure 5.6b) and when they *cooperate* by communicating their state and intended control actions (Figure 5.6c). Complex performance, computation, and communication trade-offs arise under the three MPC architectures studied. In particular, the centralized scheme achieves best performance when the computing and communication delays are short (which might not be achievable in large systems). On the other hand, the performance of decentralized schemes might be worse than centralized but computing delays are expected to be shorter as well. Analyzing such trade offs is facilitated by the `ComputingGraph`, since this captures communication and computing times while simultaneously advancing the plant simulation. The proposed framework also captures *asynchronous* behavior of the decentralized and cooperative schemes. In particular, the controllers might inject their control actions as soon as they complete their computing tasks (as opposed to waiting when all of them are done).



(a) Centralized (b) Decentralized (c) Cooperative

Figure 5.6: Simulated MPC architectures: centralized (left), decentralized (middle) and cooperative (right).

5.4.2 *Implementation*

Computing Graph 5.2 specifies the setup for the cooperative MPC algorithm. The plant node *pl* contains the task `run_plant` which advances the state of the system from the current clock time to the time of the next action signal. The plant node includes attributes u_1, u_2, and u_3 (which are the control actions received from the MPC controllers) and x (which are the plant states). The plant node is self-triggered by updating x_{pl}, which allows the

task to run continuously (i.e., the task `run_plant` constantly updates the attribute x_{pl}). The plant state is communicated to each MPC node at a *constant sampling time*. The edges that connect the plant state attribute x_{pl} to each MPC node will trigger when sending its source attribute (with a given waiting time θ_{wait}). The MPC nodes $n_{mpc,1}$, $n_{mpc,2}$, and $n_{mpc,3}$ each execute their task `control_action` which computes a control action by solving an optimization problem using attributes from the rest of the MPC controllers and from the plant. If they have completed enough coordination iterations, they each update their attribute u_{inject}, which triggers communication into the plant. Otherwise, they each update their control trajectory u_{p_n} and exchange their attributes. This triggers the MPC node task `receive_policy`, which manages the MPC trajectory exchange. Additionally, we specify logic to handle the case when the attributes $\{u_{p_i}, (i \neq n)\}$ are received while a node n is busy executing a task. When this occurs, the triggered task is queued and triggered once the controller finishes its current task (or every task queued before it).

Computing Graph 5.2 Cooperative Control

Plant Node (pl)
Attributes: $\boldsymbol{\mathcal{A}_{pl}} := (u_1, u_2, u_3, x)$
Tasks: $\boldsymbol{\mathcal{T}_{pl}} := (\texttt{run\_plant})$
  `run_plant`: Task 5.3, triggered by: Updated(x)

MPC Nodes $(n = \{1,2,3\})$
Attributes: $\boldsymbol{\mathcal{A}_n} := (u_{inject}, y, u_{p_1}, u_{p_2}, u_{p_3}, iter, flag)$
Tasks: $\boldsymbol{\mathcal{T}_n} := (\texttt{control\_action}, \texttt{receive\_policy})$
  `control_action`: Task 5.4, triggered by: Received(y) or Updated($flag$)
  `receive_policy`: Task 5.5, triggered by: Received(u_{p_i}), $i \neq n$

Edges $\boldsymbol{\mathcal{E}_1} := x_{pl} \rightarrow y_n,\ n = \{1,2,3\}$, send on: Sent($x_{pl}$), wait: θ_{wait}
Edges $\boldsymbol{\mathcal{E}_2} := u_{n,inject} \rightarrow u_{pl,n},\ n = \{1,2,3\}$, send on: Updated($u_{n,inject}$)
Edges $\boldsymbol{\mathcal{E}_3} := u_{n,p_n} \rightarrow u_{i,p_n},\ n = \{1,2,3\}, i \neq n$, send on: Updated($u_{n,p_n}$)

Code Snippet 5.2 details the implementation of Computing Graph 5.2 for cooperative control (in much the same way Snippet 5.1 is constructed besides the increased complexity). Lines 2 through 26 create a `ComputingGraph`, add `ComputeNodes`, attributes, and tasks, and setup the appropriate triggers. Lines 29 through 31 make connections between the

plant and the controllers, and Lines 34 through 36 connect the controllers to each other (to perform cooperative communication). Finally, Line 39 executes the cooperative control simulation.

Code Snippet 5.2: Simulating Cooperative Control with `PlasmoCompute.jl`

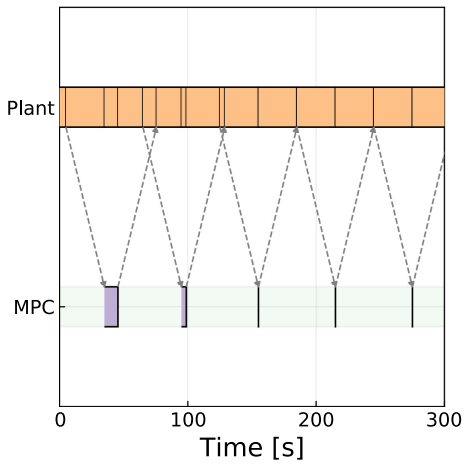
```
1  #Create a computing graph
2  graph = ComputingGraph()
3
4  #Create plant simulation node
5  @computenode(graph,pl)
6  @attributes(pl,x,u1,u2,u3)
7  @computetask(graph,pl,run_plant,triggered_by=Updated(x))
8  schedule_task(graph,run_plant,time = 0) #initialize graph
9
10 #Reactor 1 MPC
11 @computenode(graph,n1)
12 @attributes(n1,u_inject,y,u_p1,u_p2,u_p3,iter,flag)
13 @computetask(graph,n1,control_action_r1,triggered_by=[Received(y),Updated(flag)])
14 @computetask(graph,n1,receive_policy,triggered_by=Received(u_p2,u_p3),
       trigger_during_busy=:queue_task)
15
16 #Reactor 2 MPC
17 @computenode(graph,n2)
18 @attributes(n2,u_inject,y,u_p1,u_p2,u_p3,iter,flag)
19 @computetask(graph,n2,control_action_r2,triggered_by=[Received(y),Updated(flag)])
20 @computetask(graph,n2,receive_policy,triggered_by=Received(u_p1,u_p3),
       trigger_during_busy=:queue_task)
21
22 #Separator MPC
23 @computenode(graph,n3)
24 @attributes(n3,u_inject,y,u_p1,u_p2,u_p3,iter,flag)
25 @computetask(graph,n3,control_action_sep,triggered_by = [Received(y),Updated(flag)])
26 @computetask(graph,n3,receive_policy,triggered_by=Received(u_p1,u_p2),
       trigger_during_busy=:queue_task)
27
28 #Connect plant to MPC controllers
29 @connect(graph,pl[:x]=>[n1[:y],n2[:y],n3[:y]],delay=30,send_on=Sent(pl[:x]),
       send_wait=60,start=5)
30 #Connect MPC controllers to plant
31 @connect(graph,[n1[:u_inject],n2[:u_inject],n3[:u_inject]]=>[pl[:u1],pl[:u2],pl[:u3
       ]],delay=30,send_on=Updated([n1[:u_inject],n2[:u_inject],n3[:u_inject]]))
32
33 #Connect MPC controllers to perform cooperation
34 @connect(graph,n1[:u_p1]=>[n2[:u_p1],n3[:u_p1]],send_on=Updated(n1[:u_p1]))
35 @connect(graph,n2[:u_p2]=>[n1[:u_p2],n3[:u_p2]],send_on=Updated(n2[:u_p2]))
36 @connect(graph,n3[:u_p3]=>[n1[:u_p3],n2[:u_p3]],send_on=Updated(n3[:u_p3]))
37
38 #Execute computing graph
39 execute!(graph,5000)
```

5.4.3 *Results*

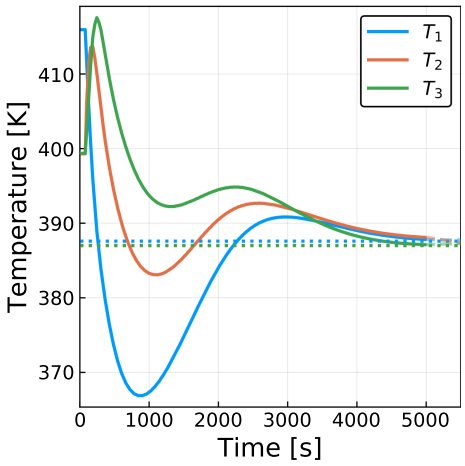
Figure 5.7 presents the simulation results for each MPC algorithm. The centralized MPC communication pattern (5.7a) shows the communication delays between the plant and the controller (grey arrows), the time required to compute the control action (the purple

bar), and highlights how the plant state advances continuously while computation and communication tasks execute. Despite the delays enforced for the controller, centralized MPC is able to drive the state to the set-point (Figure 5.7b) as we discussed in Section 5.3.2. Decentralized MPC does not require communication between controllers and computing times are decreased (Figure 5.7c) but we observe that the set-point cannot be reached (Figure 5.7d). This is because this approach does not adequately capture multi-variable interactions. Finally, cooperative MPC has a more sophisticated communication strategy (Figure 5.7e) but we observe that this helps mimic the performance of centralized MPC (Figure 5.7f).

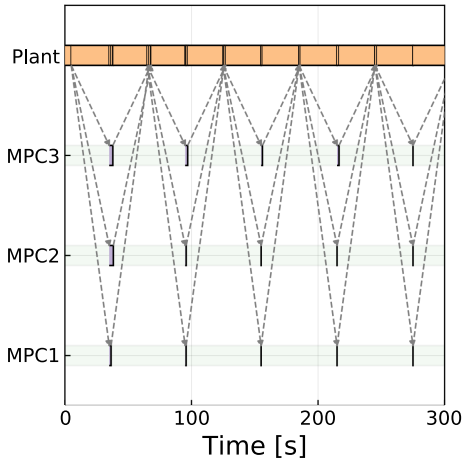
We also use the `ComputingGraph` to simulate the effects of control system failures. Particularly, we look at the cooperative control case and simulate three distinct scenarios where we (i) artificially increase controller computation times, (ii) shut down the `CommunicationEdges` between `MPC1` and the other controllers at time 250, and (iii), shut down controller `MPC1` at time 250. Figure 5.8 depicts the results for each scenario. For the case with long computation times the system still stabilizes (albeit producing a less smooth state profile) as shown in Figure 5.8a and 5.8b. The cooperative system also handles the communication failure despite an initially non-smooth controller response as depicted in Figures 5.8c and 5.8d. Most surprisingly, we see that the system appears to recover despite the failure of *MPC*3 which highlights the robustness of cooperative control policies (the other controllers save the system). This highlights a powerful aspect of using the `ComputingGraph` for simulations. With a modest amount of effort, we were able to simulate various policies and failures subject to limitations that would arise in the real-time system.



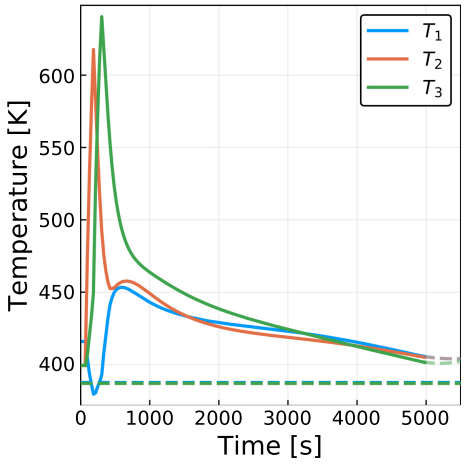
(a) Centralized communication



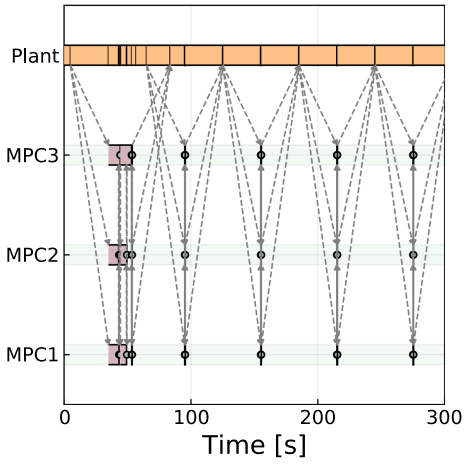
(b) Centralized state profile



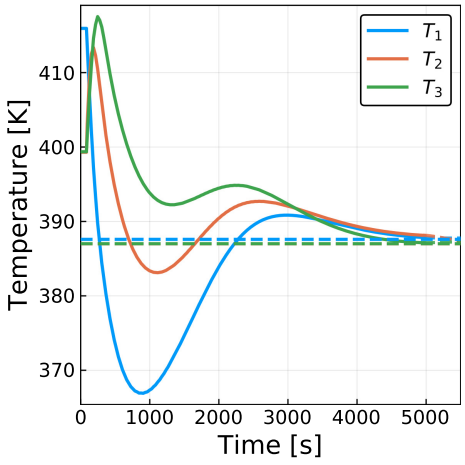
(c) Decentralized communication



(d) Decentralized state profile

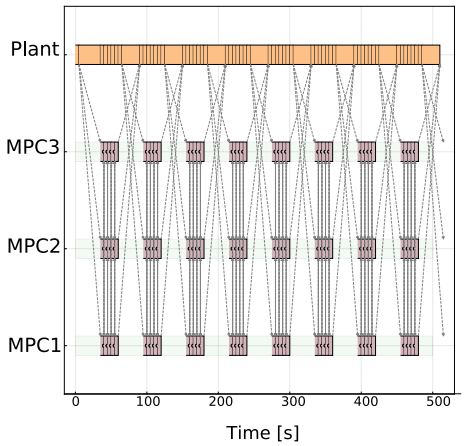


(e) Cooperative communication

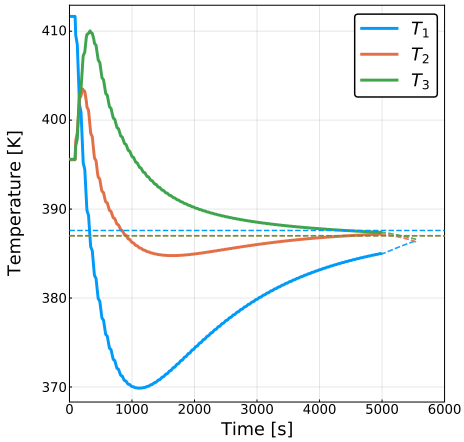


(f) Cooperative state profile

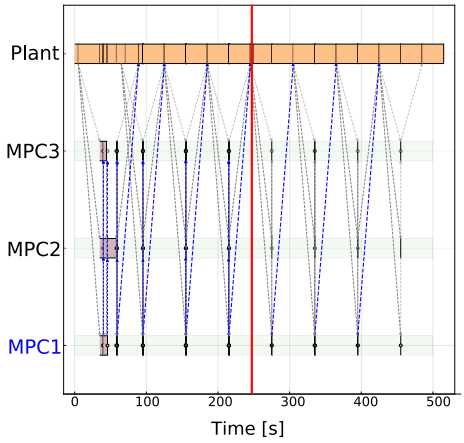
Figure 5.7: Simulation results for MPC architectures. Centralized MPC converges to the set-point despite the computing delays (top panels). Decentralized MPC does not converge to the set-point (middle panels). Cooperative MPC exhibits communication complexity but converges to the centralized MPC solution (bottom panels).



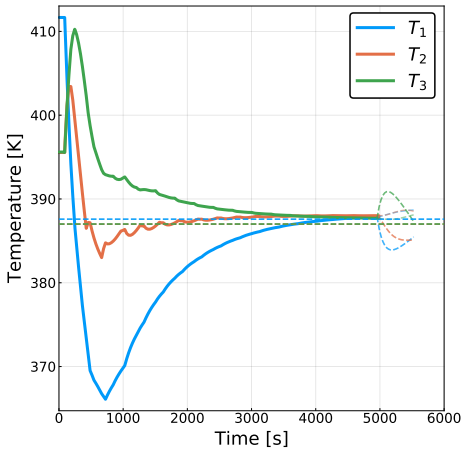
(a) Long computation



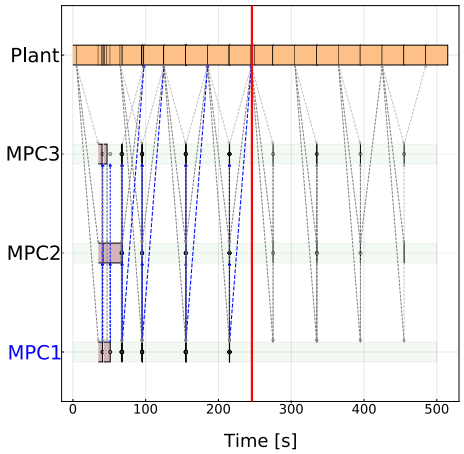
(b) Long computation state profile



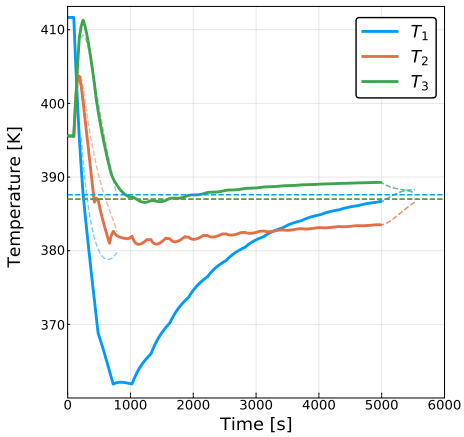
(c) Failed communication



(d) Failed communication state profile



(e) Failed controller



(f) Failed controller state profile

Figure 5.8: Simulation results for cooperative MPC failures. Cooperative MPC converges to the set-point despite long computing delays (top panels). Cooperative MPC stabilizes the system after MPC1 loses communication with MPC2 and MPC 3 (middle panels). Cooperative MPC stabilizes the system after MPC1 fails (bottom panels).

5.5 Appendix: Reactor System Study Model

5.5.1 *Model Equations*

The model for the plant is given by the following set of differential equations:

$$
\begin{aligned}
\frac{dH_1}{dt} &= \frac{1}{\rho A_1}(F_{f1} + F_R - F_1) \\
\frac{dx_{A1}}{dt} &= \frac{1}{\rho A_1 H_1}(F_{f1}x_{A0} + F_R x_{AR} - F_1 x_{A1}) - k_{A1}x_{A1} \\
\frac{dx_{B1}}{dt} &= \frac{1}{\rho A_1 H_1}(F_R x_{BR} - F_1 x_{B1}) + k_{A1}x_{A1} - k_{B1}x_{B1} \\
\frac{dT_1}{dt} &= \frac{1}{\rho A_1 H_1}(F_{f1}T_0 + F_R T_R - F_1 T_1) - \frac{1}{C_p}(k_{A1}x_{A1}\Delta H_A + k_{B1}x_{B1}\Delta H_B) + \frac{Q_1}{\rho A_1 C_p H_1} \\
\frac{dH_2}{dt} &= \frac{1}{\rho A_2}(F_{f2} + F_1 - F_2) \\
\frac{dx_{A2}}{dt} &= \frac{1}{\rho A_2 H_2}(F_{f2}x_{A0} + F_1 x_{A1} - F_2 x_{A2}) - k_{A2}x_{A2} \\
\frac{dx_{B2}}{dt} &= \frac{1}{\rho A_2 H_2}(F_1 x_{B1} - F_2 x_{B2}) + k_{A2}x_{A2} - k_{B2}x_{B2} \\
\frac{dT_2}{dt} &= \frac{1}{\rho A_2 H_2}(F_{f2}T_0 + F_1 T_1 - F_2 T_2) - \frac{1}{C_p}(k_{A2}x_{A2}\Delta H_A + k_{B2}x_{B2}\Delta H_B) + \frac{Q_2}{\rho A_2 C_p H_2} \\
\frac{dH_3}{dt} &= \frac{1}{\rho A_3}(F_2 - F_D - F_R - F_3) \\
\frac{dx_{A3}}{dt} &= \frac{1}{\rho A_3 H_3}(F_2 x_{A2} - (F_D + F_R)x_{AR} - F_3 x_{A3}) \\
\frac{dx_{B3}}{dt} &= \frac{1}{\rho A_3 H_3}(F_2 x_{B2} - (F_D + F_R)x_{BR}) - F_3 x_{B3}) \\
\frac{dT_3}{dt} &= \frac{1}{\rho A_3 H_3}(F_2 T_2 - (F_D + F_R)T_R - F_3 T_3) + \frac{Q_3}{\rho A_3 C_p H_3},
\end{aligned}
\tag{5.6}
$$

where for $i = \{1, 2\}$ we have:

$$
k_{Ai} = k_A \exp\Big(-\frac{E_A}{RT_i}\Big), \quad k_{Bi} = k_B \exp\Big(-\frac{E_B}{RT_i}\Big).
$$

The recycle and weight fractions are given by:

$$F_D = 0.01F_R, \quad x_{AR} = \frac{\alpha_A x_{A3}}{\overline{x}_3}, \quad x_{BR} = \frac{\alpha_B x_{B3}}{\overline{x}_3}$$

$$\overline{x}_3 = \alpha_A x_{A3} + \alpha_B x_{B3} + \alpha_C x_{C3}, \quad x_{C3} = (1 - x_{A3} - x_{B3}).$$

The target steady-state (set-point) is described by the parameters in Table 5.2 and the initial operating condition is given in Table 5.3.

Table 5.2: Target steady-state and parameters for reactor-separator system

Parameter	Value	Units	Parameter	Value	Units
H_1	16.1475	m	A_1	1.0	m^2
x_{A1}	0.6291	wt	A_2	1.0	m^2
x_{B1}	0.3593	wt	A_3	0.5	m_2
T_1	387.594	K	ρ	1000	kg/m^3
H_2	12.3137	m	C_p	4.2	kJ/kgK
x_{A2}	0.6102	wt	x_{A0}	0.98	wt
x_{B2}	0.3760	wt	T_0	359.1	K
T_2	386.993	K	k_A	2769.44	$1/s$
H_3	15.0	m	k_B	2500.0	$1/s$
x_{A3}	0.2928	wt	E_A/R	6013.95	kJ/kg
x_{B3}	0.67	wt	E_B/R	7216.74	kJ/kg
T_3	387.01	K	ΔH_A	-167.4	kJ/kg
F_{f1}	6.3778	kg/s	ΔH_B	-139.5	kJ/kg
Q_1	26.0601	kJ/s	α_A	5.0	–
F_{f2}	6.8126	kg/s	α_B	1.0	–
Q_2	5.0382	kJ/s	α_C	0.5	–
F_R	56.7989	kg/s			
Q_3	5.0347	kJ/s			
F_1	63.1766	kg/s			
F_2	69.9892	kg/s			
F_3	12.6224	kg/s			

For the 3 MPC controllers, we have the following outputs and inputs:

Table 5.3: Initial conditions for simulation of reactor-separator system

STATE	Value	Units	Input	Value	Units
H_1	25.4702	m	F_{f1}	1.1866	kg/s
x_{A1}	0.1428	wt	Q_1	29.0597	kJ/s
x_{B1}	0.7045	wt	F_{f2}	7.0263	kg/s
T_1	415.944	K	Q_2	5.1067	kJ/s
H_2	5.4703	m	F_R	11.6962	kg/s
x_{A2}	0.3653	wt	Q_3	5.09834	kJ/s
x_{B2}	0.5307	wt	F_1	12.8828	kg/s
T_2	399.303	K	F_2	19.9091	kg/s
H_3	15.0	m	F_3	8.0960	kg/s
x_{A3}	0.1565	wt			
x_{B3}	0.67	wt			
T_3	399.364	K			

$$y_1 = [H_1,\ x_{A1},\ x_{B1},\ T_1], \quad u_1 = [F_{f1},\ Q_1,\ F_1]$$
$$y_2 = [H_2,\ x_{A2},\ x_{B2},\ T_2], \quad u_2 = [F_{f2},\ Q_2,\ F_2]$$
$$y_3 = [H_3,\ x_{A3},\ x_{B3},\ T_3], \quad u_3 = [F_R,\ Q_3,\ F_3].$$

Each MPC controller uses a quadratic cost function with weights:

$$Q_{y1} = \text{diag}(100, 10, 100, 0.1) \quad Q_{y2} = \text{diag}(10, 10, 100, 0.1) \quad Q_{y3} = \text{diag}(1, 10, 10^5, 0.1)$$
$$R_{yi} = \text{diag}(100, 100, 100),\ i = \{1, 2, 3\}.$$

The differential equations are discretized using an Euler scheme with a time horizon of $N = 20$ and a time step $\Delta\, t = 30$.

5.5.2 *Simulation Tasks*

The cooperative MPC computation tasks are defined in Tasks 5.3, 5.4, and 5.5. Task 5.3 simulates the plant forward in time from the current computing graph time to the time

of the next signal in the computing graph queue and updates the attribute x. Task 5.4 computes the open-loop control trajectory for MPC controller n and updates its control injection u_{inject} if it has completed enough iterations and updates its control policy u_{p_n} if it has not. Task 5.5 checks whether the MPC controller has received updates from the other MPC controllers and updates the flag indicator $flag$ if it has received both inputs.

Task 5.3 `run_plant`(computing graph $\mathcal{G}$,node n)

1: Get graph clock value t_{now}: Current clock time
2: Get graph clock value t_{next}: Next signal time
3: Get node attributes $\{u_1, u_2, u_3\}$: Received MPC controller actions
4: Get node attribute x: Plant state
5: Simulate plant (5.6) `update attribute` $x \leftarrow$ `simulate_plant`$(u_1, u_2, u_3, x, t_{now}, t_{next})$

Task 5.4 `control_action`(node n)

1: Get node attribute $iter_{max}$
2: Get node attribute y: Received plant measurement
3: Get node attribute u_{inject}: Current injected control from MPC controller
4: Get node attributes $\{u_{p_i}, i \neq n\}$: Received control actions from other MPC controllers
5: Get node attribute $iter$: Current cooperative iteration
6: Calculate control action: $u_{calc} \leftarrow$ `compute_control_action`$(y_n, u_{p_i}, i \neq n)$
7: Update iteration count: $iter \leftarrow iter + 1$
8: **if** $iter = iter_{max}$ **then**
9: Update injected control: `update attribute` $u_{inject} \leftarrow u_{calc}$
10: $iter \leftarrow 1$
11: **else**
12: Update control: `update attribute` $u_{p_n} \leftarrow u_{calc}$
13: **end if**

Task 5.5 `receive_policy`(node n, attribute u_{p_i})

1: Get node argument u_{p_i} : Received neighbor controller actions
2: Get node attribute $flag$: Flag that control action is ready to compute
3: **if** Received both policies($\{u_{p_i}, i \neq n\}$) **then**
4: `update attribute` $flag$
5: **end if**

Chapter 6

DISTRIBUTED ALGORITHM SIMULATION

In Chapter 5 we presented the `ComputingGraph` abstraction and demonstrated how it facilitates the simulation of real-time cyber-physical systems over distributed communication topologies. This chapter further explores the computing aspects afforded by the `ComputingGraph`. Particularly, we explore how it can be used to develop and simulate *distributed algorithms* for optimization and machine learning applications.

6.1 Introduction

Continuous software and hardware advances have made it possible to deploy massively parallel scientific applications over increasingly larger sets of distributed computing resources. A key underlying aspect of many parallel applications is the development and implementation of *algorithms* that need to utilize increasingly complex distributed dynamic and heterogeneous computing environments. Consequently, there is an increasing need to have the capabilities to *simulate algorithms* that run on high-performance computing (HPC) architectures to quantify their performance and to identify bottlenecks that arise due to load-imbalance and communication latency.

Realistically simulating algorithm performance on real distributed architectures is, however, extremely nontrivial [5, 93]. Communication aspects are particularly challeng-

ing to capture in computing (i.e. cyber) systems since they often involve complex topologies and frequencies (see Figure 1.3 in Chapter 1). Research in HPC simulation tends to target very specific applications such as simulating computing task scheduling techniques [11] or improving communication in large distributed networks [134]. Throughout this chapter, we intend to show that the `ComputingGraph` presented in Chapter 5 naturally captures the topology and timing aspects needed to simulate distributed algorithms. For instance, we show how it can be used to develop and benchmark distributed optimization algorithms under multi-core architectures compared to using a single central processing unit (CPU). We also show how the `ComputingGraph` facilitates the development of asynchronous algorithm variants which form the basis of many machine learning applications.

The rest of the chapter is organized as follows. Section6.2 provides an overview of the challenges in implementing distributed algorithms for optimization and machine learning applications. Specifically, Section 6.2 discusses distributed Benders decomposition for solving two-stage stochastic optimization problems and introduces the distributed stochastic gradient descent (SGD) algorithm for training machine learning classification models. Section 6.3 presents a case study where we use a `ComputingGraph` to simulate distributed Benders decomposition on a multi-core system and Section 6.4 uses a `ComputingGraph` to experiment with different distributed SGD variants.

6.2 Distributed Optimization and Machine Learning

The increasing size and complexity of modern data sets has motivated efforts to solve challenging problems in the optimization and machine learning fields that harness massive amounts of data or that exhibit extremely high dimensionality. Examples of such problems include solving large-scale stochastic optimization problems [16] and training data intensive machine learning classification models [20]. Developing efficient distributed algorithms offers a means to tackle such challenging problems by exploiting massively parallel computing architectures [14]. Consequently, the development of efficient

distributed algorithms that better harness parallel architectures will have key implications towards tackling optimization and machine learning problems that involve large-scale distributed data sets.

The remainder of this section presents an overview of distributed Benders decomposition for stochastic optimization and highlights the challenges that *still* arise in the *distributed implementation* of this classic decomposition algorithm. We then present a summary of distributed stochastic gradient descent (SGD) and discuss the challenges in executing its distributed algorithmic variants.

6.2.1 *Distributed Benders Decomposition*

Benders decomposition is one of the most widely used approaches to solve two-stage stochastic optimization problems. The main idea behind the two-stage problem is to optimize decisions (variables) subject uncertainty that may realize in the future. We can make decisions both before this uncertainty realizes (called a first stage problem), and after the realization of uncertainty (called the second stage, or recourse function). The typical practice in the two-stage paradigm is to approximate uncertainty with a probability distribution of random variables (e.g. using a forecast of the future) and sample a finite number of scenarios (which enter into the optimization problem as data). In this section we provide a somewhat high level overview of Benders decomposition for two-stage problems and then discuss its distributed implementation.

Overview of Benders Decomposition

Here we provide a quick overview of the Benders decomposition algorithm. This overview omits much of the algorithm details and so we refer the interested reader to [104] for a recent comprehensive review and construction of the algorithm. For our overview we use a simple two-stage optimization problem where we denote a finite set of scenarios (data) as $\xi \in \Xi$ that capture uncertainty where each ξ is a specific realized scenario. For ease of presentation we also consider a simple *linear* two-stage stochastic program described by

(6.1a).

$$\phi = \min_{x} c^T x + \sum_{\xi \in \Xi} p_\xi Q_\xi(x) \tag{6.1a}$$

$$s.t. \quad Ax = b, x \in \mathbb{R}^{n_1}_+ \tag{6.1b}$$

In this formulation, c is a vector of coefficients, x is a vector of first-stage decision variables, and p_ξ is the probability that each specific scenario ξ realizes. This formulation also uses a standard recourse function $Q_\xi(x)$ given by:

$$Q_\xi(x) = \min_{y} q_\xi^T y \tag{6.2}$$

$$s.t. \quad W_\xi y = h_\xi - T_\xi x, y \in \mathbb{R}^{n_2}_+ \tag{6.3}$$

where the scenario data ξ are captured in $\{q_\xi, W_\xi, h_\xi, T_\xi\}$ which describe the second stage objective function and constraints. In Benders decomposition, we iterate between the solution of a master problem (corresponding to the first stage) and its subproblems (the second stage) where we denote the iteration counter as k. The Benders master problem corresponding to (6.1a) can be given by (6.4a)

$$(MP) : \phi^k = \min_{x, \{\theta_\xi\}_{\xi \in \Xi}} c^T x + \sum_{\xi \in \Xi} p_\xi \theta_\xi \tag{6.4a}$$

$$s.t. \quad Ax = b, x \in \mathbb{R}^{n_1}_+ \tag{6.4b}$$

$$\theta_\xi \geq (\pi^\xi)^T (h_\xi - T_\xi x), \quad \pi^\xi \in \hat{V}^{\xi,k}, \quad \xi \in \Xi \tag{6.4c}$$

where θ_ξ represents cuts added to the master problem for scenario ξ for subproblem constraints that are violated (i.e. scenarios where we have the solution $\hat{\theta}_\xi \leq Q_\xi(\hat{x}^k)$). Here, π^ξ comes from the subproblem dual solution and $\hat{V}^{\xi,k}$ represents the history of dual solutions received by the master problem up to iteration k.

Each subproblem (SP) of the Benders decomposition algorithm can be solved according to: $Q_\xi(\hat{x}^k) = \max_\pi \{\pi^T (h_\xi - T_\xi \hat{x}^k) : \pi^T W_\xi = q_\xi\}$ which can be formulated into the

linear program given by (6.5a).

$$(SP) : Q_{\xi}(\hat{x}^k) = \min_{y} q_{\xi}^T y \tag{6.5a}$$

$$s.t. \quad W_{\xi} y = h_{\xi} - T_{\xi}\hat{x}^k, y \in \mathbb{R}_{+}^{n_2} \tag{6.5b}$$

Here, the dual solution $\hat{\pi}^{\xi,k}$ is obtained from the subproblem and is used to update the master problem (i.e. $\hat{V}^{\xi,k} = \{\hat{V}^{\xi,k-1}, \hat{\pi}^{\xi,k}\}$) if any constraints in (6.5a) are violated. The Benders algorithm iterates between the master and subproblems until a convergence criteria is achieved.

Parallel Benders Decomposition

The most prominent parallel variant of Benders decomposition is based on a synchronous master-subproblem paradigm in which a master processor coordinates with sub-processors to solve the two-stage optimization problem [82]. An example of such a communication topology is depicted in Figure 6.1. Here, we show a hypothetical computing architecture consisting of four CPUs where CPUs 1,2, and 3 solve subproblem solutions (i.e. solve (6.5a)) given scenarios (data) ξ_1, ξ_2, ξ_3 and a master problem solution $\hat{x}$, and return solution data s_1, s_2, s_3 to CPU 4 which creates new cuts and solves the master problem (i.e. solves (6.4a)).

The synchronous variant of Benders decomposition using a master processor and set of $\mathcal{N} := \{1, ..., N\}$ worker CPUs that solve scenario subproblems is described in Algorithm 6.1. The algorithm uses computing tasks implemented in the functions `solve_subproblem`, `receive_solution`, and `solve_master`. The function `solve_subproblem` computes a subproblem solution s for a given scenario data sample $\xi \in \Xi$ using the current master solution $\hat{x}$. Here, the master solution $\hat{x}$, the scenario data ξ, and the subproblem solution s are data attributes that are communicated between the master and the worker CPUs. The `solve_subproblem` function activates the `receive_solution` task on the master node, which stores subproblem solutions into the

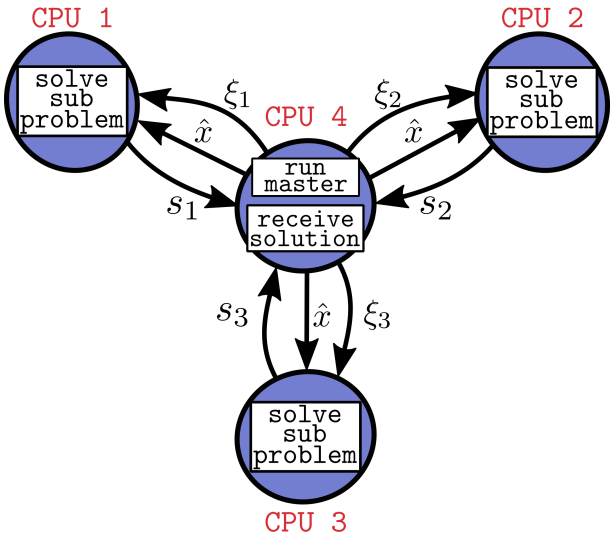


Figure 6.1: Hypothetical computing architecture executing Benders decomposition. CPU 4 executes the solution of the Benders master problem and receives solutions from the subproblems. CPUs 1, 2, and 3 execute the solution of the scenario subproblems.

set S, updates the set of cuts C, and checks how many scenario subproblems have been solved. If all subproblems are completed, it triggers the execution of the `solve_master` task using the current set of solution data S and cuts C. Otherwise, it triggers the execution of the `solve_subproblem` task on a worker processor. When `solve_master` is executed, it takes subproblem solution attribute s to update the master attribute $\hat{x}$ and stops the computing graph if convergence is achieved. If not converged, it empties the solution set S and activates the `solve_subproblem` tasks for each worker again, which will then obtain new subproblem attributes.

While seemingly straight-forward, maximizing the parallel efficiency of Benders decomposition is not trivial. The synchronous algorithm given by Algorithm 6.1 can suffer from significant load imbalance (particularly when an iteration of the master problem takes considerable computation time). To alleviate this, communication between the master and subproblems can be performed *asynchronously* such that the master problem resolves before all sub-processors have communicated their solutions. There should also be consideration for the *allocation* of subproblems which can be done in a static or dynamic

Algorithm 6.1 (Synchronous Benders)

function `solve_subproblem` ($\hat{x}$, ξ)
Given the master solution $\hat{x}$ and scenario ξ, evaluate sub-problem solution s
Activate `receive_solution`(s) on the master node
end function

---

function `receive_solution`(s)
Given subproblem solution s, store solution $S \leftarrow s$, and update cuts C
if All subproblems complete **then**
  Activate `solve_master`(S, C) on the master node
else
  Get new scenario ξ from scenario set Ξ
  Activate `solve_subproblem`($\hat{x}$, ξ) on worker node
end if
end function

---

function `solve_master`(S, C)
Given solution data S and cuts C, update master solution $\hat{x}$
Empty solution data $S \leftarrow \{\}$
if converged **then**
  STOP
else
  for $n = 1, ..., N$ **do**
    Get scenario ξ_n from set Ξ
    Activate `solve_subproblem`($\hat{x}$, ξ_n) on worker node
  end for
end if
end function

Initialize variable values and scenario set Ξ
Activate `solve_master` on master node

fashion. Specifically, batches of subproblems can be assigned to CPUs before algorithm execution (static allocation), or they can be assigned to CPUs on the fly (dynamic allocation) based on their availability.

6.2.2 *Distributed Stochastic Gradient Descent*

Stochastic gradient descent (SGD) [19] is the backbone of most state-of-the-art machine learning algorithms. The SGD algorithm has frequently demonstrated its capability to

solve large convex learning problems with lots of data [135], but it faces the same challenges as discussed with Benders decomposition with respect to its distributed implementation [103]. Traditional SGD algorithm implementations are run in serial using a single CPU, but such implementations are prohibitively slow when dealing with massive data sets. A solution that has proved successful in recent years is to parallelize the training across many learners (e.g. CPUs). This method was first used at a large-scale in the `Google DistBelief` project [34] which used a central parameter server (PS) to aggregate gradients computed by learner nodes (see Figure 6.2). While parallelism dramatically speeds up training, distributed machine learning frameworks face several challenges such as straggling learners and gradient staleness [38]. For example, distributed SGD when run in a synchronous manner, suffers from delays in waiting for the slowest learners (i.e. stragglers). Asynchronous methods can alleviate stragglers, but cause gradient staleness (gradients evaluated with old PS parameters) that can adversely affect convergence.

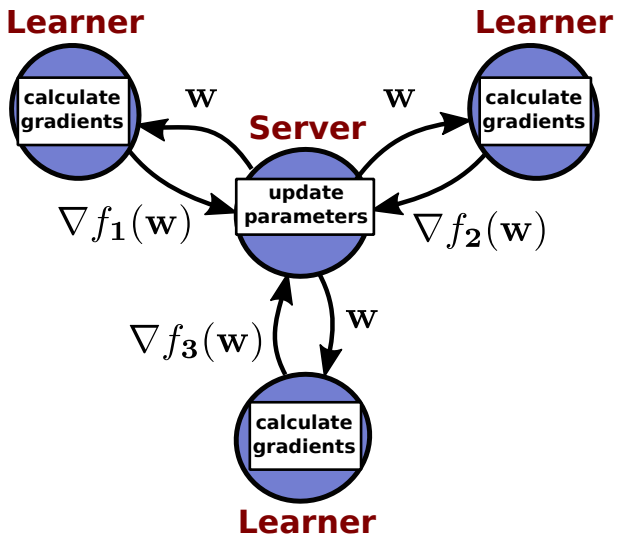


Figure 6.2: Topology of simulated architecture for stochastic gradient descent. The server node updates parameters given evaluated gradients. The learner nodes compute gradients given a current set of parameters and data.

SGD typically solves the minimization problem of an empirical risk function given by

(6.6)

$$\min_{w} \phi(w) := \frac{1}{N_s} \sum_{j=1}^{N_s} f(w, \xi_j) \tag{6.6}$$

where ξ_j denotes the j^{th} data point with $j \in \{1, ..., N_s\}$ where N_s is the total number of samples. In this function, w is a vector of parameters, and $f(w, \xi_n)$ is a loss function (e.g. such as mean squared error or cross entropy). The SGD algorithm iteratively minimizes this objective function by updating the parameter vector w in the opposite direction of the gradient of $\phi(w)$ at each iteration k of the algorithm according to (6.7)

$$w^{k+1} = w^k - \eta \nabla \phi(w^k) = w^k - \frac{\eta}{N_s} \sum_{j=1}^{N_s} \nabla f(w^k, \xi_j) \tag{6.7}$$

where η in this case is the learning rate. In practice, the calculation of $\nabla f(w^k, \xi_j)$ is often expensive, and so it is common to use batches of samples and compute the average gradient. We can specify the set of *batches* as $\xi_b \in \{\xi_{b1}, \xi_{b2}..., \xi_{bB}\}$ where B is the total number of batches. We assume a constant batch size m for all batches. Thus, in the case for $b = 1$, we would have $\xi_{b1} := \{\xi_1, ..., \xi_m\}$ if we ordered the samples sequentially.

Fully Synchronous Variant

With our definitions so far, we can pose the *synchronous* parameter update for SGD given by (6.8)

$$w^{k+1} = w^k - \frac{\eta}{B} \sum_{j=1}^{B} g(w^k, \xi_{bj}^k) \tag{6.8}$$

where $g(w^k, \xi_{bj}^k) = \frac{1}{m} \sum_{\xi \in \xi_{bj}^k} \nabla f(w^k, \xi)$ is the average gradient for the batch of samples ξ_{bj}^k for the given parameter vector w at iteration k. A description of the synchronous algorithm is given by Algorithm 6.2. In essence, this algorithm has the parameter server (PS) allocate batches to the learners and then activates the `compute_gradient` function on each learner with the initial parameter vector w. Each learner starts computing gradients

for its set of batches and activates the `update_gradient` function on the PS after each batch is computed. The learners compute gradients until they run through all of their allocated batches of samples. Once the PS receives every gradient, it updates the parameters w with `update_parameters` and then re-activates the `compute_gradient` function on each learner.

Fully Asynchronous Variant

The fully synchronous update described by Equation (6.8) and Algorithm 6.2 is prone to stragglers (gradients that require long computation times) and communication bottlenecks (waiting for gradients to arrive at the PS). To alleviate bottlenecks, the fully *asynchronous* variant can be used to improve parallel efficiency by using the update rule given by (6.9).

$$w^{k+1} = w^k - \eta g(w^{\tau(k)}, \xi_b^k) \tag{6.9}$$

With this update rule, the parameter server updates the parameters *every time* a gradient is received. Here, ξ_b^k simply denotes the batch that is used for the k^{th} update, and $\tau(k)$ denotes the iteration index for the parameters that were taken from central PS to calculate the gradient. Thus we have $\tau(k) \leq k$ which means that the parameter update may use old (stale) gradients.

K-Batch Asynchronous Variant

Asynchronous SGD can achieve almost perfect parallel efficiency, but typically at the cost of instability or a considerable number of iterations. The purely asynchronous variant is also known to achieve higher error-floors compared to the synchronous variant (i.e. it cannot reduce the loss as much as synchronous SGD). The K-batch asynchronous update rule given by (6.10) provides a trade-off between the two extreme variants.

$$w^{k+1} = w^k - \frac{\eta}{K} \sum_{j=1}^{K} g(w^{\tau(j,k)}, \xi_{bj}^k) \tag{6.10}$$

Algorithm 6.2 (Synchronous Distributed SGD)

function `compute_gradient`(w,ξ_b)
Given parameter vector w, retrieve batch ξ_b, store current parameter w and compute average gradient g
Activate `update_gradients`(g) on PS
if not all batches complete **then**
  Increment batch counter $count+ = 1$ on learner n
  Activate `compute_gradient`(w) on learner n
else
  Reset batch counter $count = 1$ on learner n
end if
end function

---

function `update_gradients`(g)
Given average gradient g, store gradient $G \leftarrow g$
if All gradients received **then**
  Activate `update_parameters`(w,G) on PS
end if
end function

---

function `update_parameters`(w,G)
Given current parameters w and gradients G, update w according to (6.8)
Empty gradients $G \leftarrow \{\}$
Evaluate loss function with new w
if tolerance reached **then**
  STOP
else
  for $n = \{1, ..., N\}$ **do**
    Retrieve first batch ξ_b on each learner n
    Activate `compute_gradient`(w) on each learner n
  end for
end if
end function

Initialize parameter vector w and set of gradients $G \rightarrow \{\}$
Allocate batches $\{\xi_{bj}\}_{j \in \{1,...,B\}}$ to learners $n = \{1, ..., N\}$
for $n = \{1, ..., N\}$ **do**
  Activate `compute_gradient`(w) on each learner n
end for

For this update rule, we have the PS wait for the first K batches of gradients to return before updating the parameters w. We again denote ξ_{bj}^k as the $j-th$ batch used at the $k-th$ iteration (such as in the synchronous case), but we now have $\tau(j,k)$ which denotes

the iteration index of the parameters w that were used to compute gradients on the $j-th$ batch where $\tau(j,k)k$. We thus have that $g(w_{\tau(j,k)}, \xi_{bj}^k) = \frac{1}{m}\sum_{\xi \in \xi_{bj}^k} \nabla f(w_{\tau(j,k)}, \xi)$ is the average gradient of the loss function evaluated over the mini-batch ξ_{bj}^k based on the stale value of the parameter $w_\tau(j,k)$. For $K = 1$, the K-Batch asynchronous variant is exactly equivalent to fully asynchronous SGD variant given by (6.9) and for $K = B$ this variant becomes the purely synchronous variant given by (6.8)

6.3 Case Study: Simulating Distributed Benders Decomposition

This section presents a case study that demonstrates how to use a `ComputingGraph` to simulate distributed Benders decomposition on different computing architectures (i.e. different numbers or types of CPUs). As such, we show how the `ComputingGraph` allows us to experiment with algorithms that are subject to different CPU architectures or to random computing loads from other jobs that increase computing latency in the CPUs.

6.3.1 *Problem Overview*

The problem under study is a resource allocation stochastic program given by (6.14) in Section 6.5. This stochastic program can be decomposed with Benders decomposition into a master problem (6.15) and a set of subproblems (6.16) defined for a set of sampled scenarios $\xi \in \Xi$. Algorithm 6.1 can be modeled and simulated using a `ComputingGraph` which allows us to simulate the effect of computing and communication delays on the performance of the Benders scheme. on a *hypothetical* parallel computer (such as depicted by Figure 6.1). Under this abstraction, the master node solves the master problem and a set of $n \in \{1, ...N\}$ subnodes (CPUs) solve the set of Ξ of scenario subproblems. For our setup a parent `ComputeNode` allocates scenarios to the available children nodes dynamically (by keeping track of which ones are available and which ones are busy solving another scenario subproblem). The children communicate their solutions and cutting plane information to the parent node once they are done solving their subproblem.

6.3.2 *Implementation*

The simulation of the Benders algorithm (Algorithm 6.1) can be expressed in terms of nodes, tasks, and attributes following the setup provided in `ComputingGraph` 6.1. We use a master node m with attributes defined in Line 2 which include the solution to the master problem $\hat{x}$ and the scenario data $\{\xi_1 ... \xi_N\}$ which are communicated to available subnodes $n \in \{1, ..., N\}$. The master node also contains tasks (Line 3) analogous to the functions in Algorithm 6.1 to execute the master problem (`solve_master`) and to receive solutions from subnodes (`receive_solution`). The details of each task and what attributes it updates can be found in Section 6.5. The `solve_master` task is triggered by the attribute $flag$ shown in line 4. The task solves the master problem (6.15) using the master node attributes and *updates* the attributes for the master solution $\hat{x}$ and scenarios $\{\xi_1 ... \xi_N\}$. The updated attributes trigger edge communication to the subnodes (lines 10 and 11). The `receive_solution` task is executed when any solution attribute s_n is received from the connected subnode n. This task determines whether the `solve_master` task is ready to run again and either updates the attribute $flag$ (which triggers `solve_master`) or updates the corresponding scenario attribute ξ_n with a new scenario to send back to subnode n. Each subnode n computes its task `solve_subproblem` when it receives its scenario attribute ξ. The `solve_subproblem` task updates s, which triggers communication to the attribute s_n on the master node m (line 12).

The implementation of the `ComputingGraph` in `PlasmoCompute.jl` for the case of three subnodes is shown in Code Snippet 6.1. Lines 2 through 11 create the graph, add the master node with its attributes and tasks (`solve_master` and `receive_solution`). Lines 15 through 24 add subnodes to the graph, each with a `solve_subproblem` task that is executed after receiving scenario data ξ (line 18). Finally, communication edges are created between attributes (lines 21-23) and the graph is executed on Line 26 until it terminates (i.e., a task calls `Stop(graph)` such as shown in Code Snippet 6.1). It is also possible to simulate to a pre-determined time by providing an argument to `execute!()`. Also note

Computing Graph 6.1 (Synchronous Benders)

Master Node (m)
Attributes: $\boldsymbol{\mathcal{A}_m} := (\hat{x}, S, C, flag, \{s_1, ..., s_N\}, \{\xi_1, ..., \xi_N\})$
Tasks: $\boldsymbol{\mathcal{T}_m} := (\texttt{solve\_master}, \texttt{receive\_solution})$
  `solve_master`: runs Task 6.1, triggered by: Updated($flag$)
  `receive_solution`: runs Task 6.2, triggered by: Received(s_n), $n \in \{1, ..., N\}$

Sub Nodes $(n \in \mathcal{N})$
Attributes: $\boldsymbol{\mathcal{A}_n} := (\hat{x}, \xi, s)$
Tasks: $\boldsymbol{\mathcal{T}_n} := (\texttt{solve\_subproblem})$
  `solve_subproblem`: runs Task 6.3, triggered by: Received(ξ)

Edges $\boldsymbol{\mathcal{E}_1} := \hat{x}_m \to \hat{x}_n,\ n \in \{1, ..., N\}$, send on: Updated($\hat{x}_m$)
Edges $\boldsymbol{\mathcal{E}_2} := \xi_{m,n} \to \xi_n,\ n \in \{1, ..., N\}$, send on: Updated($\xi_{m,n}$)
Edges $\boldsymbol{\mathcal{E}_3} := s_n \to s_{m,n},\ n \in \{1, ..., N\}$, send on: Updated($s_n$)

that the `solve_master` and `solve_subproblem` tasks are given default compute times as the *true compute time* of their execution (`compute_time` =: `walltime`). We set a communication delay of 0.005 seconds from the master node to the subnodes (but it is also possible to make delay time a function of the attributes communicated or to experiment with different delays to evaluate the effect of communication overhead).

Code Snippet 6.1: PlasmoCompute.jl snippet for simulating Benders decomposition.

```
#Create a computing graph
graph = ComputingGraph()

N = 3 #number of subnodes
#Add the master node (m)
@computenode(graph,m)
@attributes(m,x,C,flag,ζ[1:N],s[1:N])
@computetask(m,run_master, compute_time = :walltime, triggered_by=Updated(flag))
@computetask(m,receive_solution[i = 1:N], compute_time = 0, triggered_by=Received(s[i]))
#Provide an initial signal to start the algorithm
schedule(graph, m[:run_master], time = 0)

subnodes = @computenode(graph,subnodes[1:N])

for node in subnodes
    #Add subnodes (n = 1:N) to solve sub−problems
    @attributes(subnode,x,ξ,s)
    @computetask(subnode,solve_subproblem, compute_time = :walltime,triggered_by=
        Received(ζ))

    #Connect attributes between master and subnodes
    @connect(graph,m[:x] => n[:x],send_on=Updated(m[:x]))
    @connect(graph,m[:ζ][i] => n[:ζ],send_on=Updated(m[:ζ][i]),delay=0.005)
    @connect(graph,n[:s] => m[:s][i],send_on=Updated(n[:s]))
end
#Execute the computing graph
execute!(graph)
```

6.3.3 *Results*

Figure 6.3 summarizes the simulation results of the Benders algorithm as we increase the number of CPUs available in the computing architecture (we consider cases with $N = 1, 4, 8$, and 16 CPUs). We can see that, with a communication delay of 0.005 seconds from the master to the subnodes, using one CPU has a shorter total solution time than using four CPU nodes (due to the communication overhead). Executing the algorithm on 8 and 16 CPU nodes, however, results in algorithm speed up (reduction in computing latency overcomes communication latency). This illustrates how `ComputingGraph` can help to predict trade-offs between computing and communication latency. For instance, the results predict that the proposed Benders scheme only benefits from parallelization when the number of CPUs is sufficiently large. We again highlight that the parallel architectures evaluated are *hypothetical* (the actual simulation of the algorithm was executed on a single CPU). In other words, `PlasmoCompute.jl` simulates the behavior of the Benders algorithm

on a virtual computing environment.

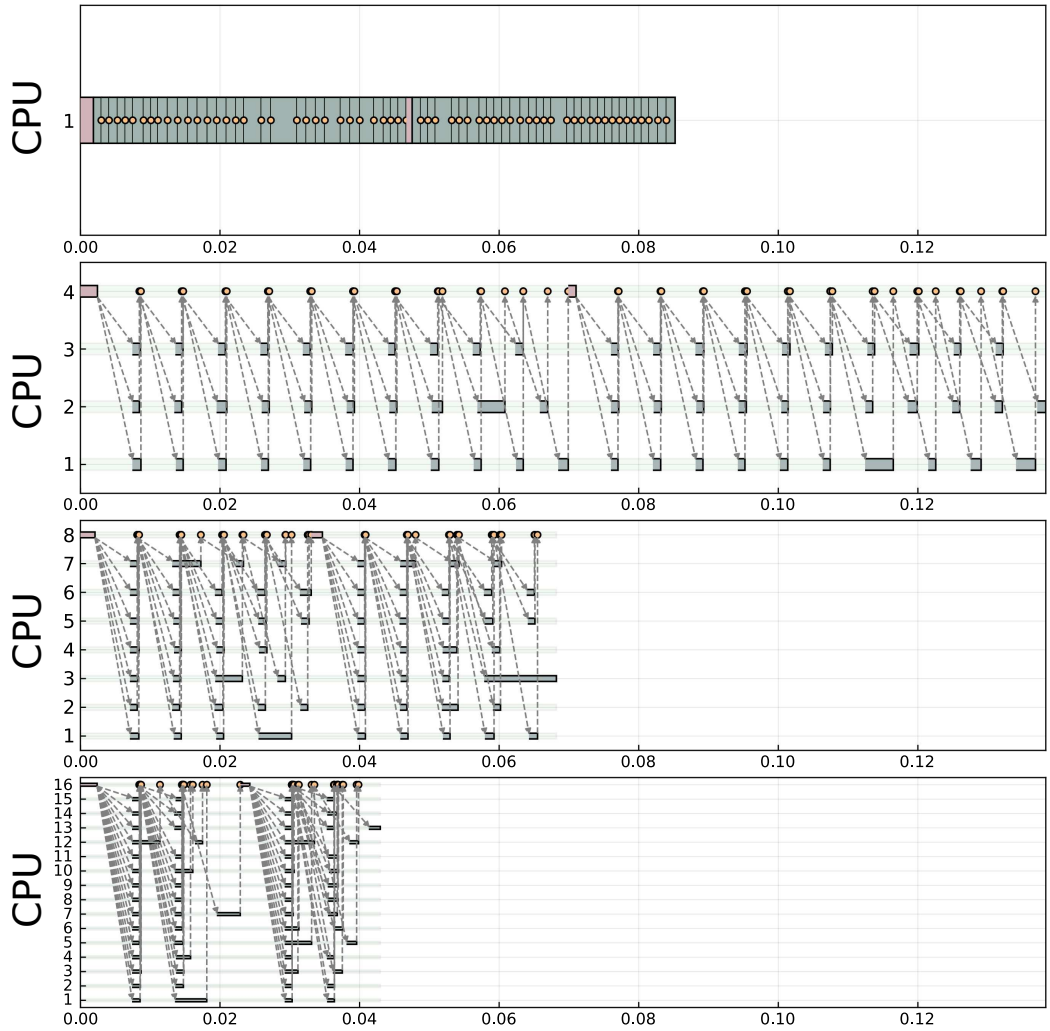


Figure 6.3: Simulation of Benders decomposition using different numbers of CPUs (top panel shows one CPU while bottom panel shows sixteen CPUs). Red tasks correspond to the master problem execution time, grey tasks represent subproblem execution time and orange dots represent the `receive_solution` task which is simulated with zero computing time.

6.4 Case Study: Simulating Stochastic Gradient Descent Variants

For this case study we investigate the trade-offs between the distributed SGD variants presented in Section 6.2.2 to train a regression model on a multi-learner system.

6.4.1 *Problem Overview*

For our problem, we seek to train a set of parameters w (i.e. weights) for a regression model using distributed SGD. We use a standard data set from the MNIST [79] database which contains hand-drawn images of the digits 0-9 (depicted in Figure 6.4) with N_s total samples (i.e. we have $\xi \in \{\xi_1, ..., \xi_{N_s}\}$) and we create 9 batches of samples ($\{\xi_{b1}, ..., \xi_{b9}\}$) which we statically allocate between the learners.

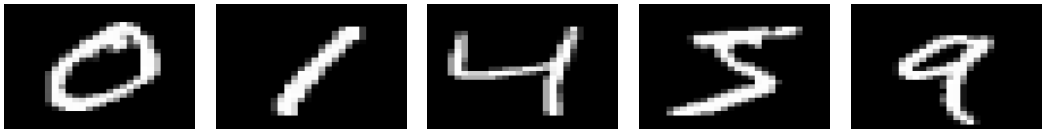


Figure 6.4: Samples of MNIST data set images corresponding to the digits 0,1,4,5, and 9.

For our classification model we use a normalized softmax regression function given by (6.11) to classify the individual hand-drawn images and determine which digit (label) the image corresponds to.

$$m_{w_i}(\xi_j) := \frac{e^{w_i^T \xi_j}}{\sum_{i=1}^{10} e^{w_i^T \xi_j}}, \quad i = \{1, ..., 10\} \tag{6.11}$$

Here, we have that $m_{w_i}(\xi_j)$ returns the normalized probability (between 0 and 1) that the sample (image) ξ_j belongs to label i (i.e. whether the image is a 1,2,3,etc...). It is often typical to include a bias parameter b_i (such that we would have $w_i^T \xi_j + b_i$ in our model) but we assume it is zero to simplify the presentation.

To train the model parameters, we use a standard cross entropy loss function with regularization. The composite cross entropy function is given by (6.12):

$$f(w, \xi_j) := -\sum_{i=1}^{10} y(\xi_j, i) ln(m_{w_i}(\xi_j)) \tag{6.12}$$

where $y(\xi_j, i) \rightarrow \{0, 1\}$ is a binary indicator that returns 1 if label i is correct for sample ξ_n and 0 otherwise. Thus, the complete loss function evaluated for all of the samples is

given by:

$$\phi(w) := \frac{1}{N_s} \sum_{j=1}^{N} f(w, \xi_j) + \beta |w|_2^2 \tag{6.13}$$

where β is a regularization parameter that helps prevent over-fitting.

6.4.2 *Implementation*

We develop a `ComputingGraph` in the same fashion as we did for the Benders case study in Section 6.3. We create the graph with `ComputeNodes` and assign tasks to the nodes as described in Computing Graph 6.2. Line 2 defines attributes for the PS such as its current parameters w_{ps}, a set of gradients G, a trigger attribute $flag_{ps}$, and attributes for gradients it receives from each learner node$\{g_{ps,1}, ..., g_{ps,N}\}$. Lines 3, 4, and 5 define compute tasks for the parameter server that handle the update of parameters and receiving computed gradients. Likewise, Lines 6 through 10 setup attributes and tasks for each learner node. The learners contain attributes for received parameters w_n, calculated gradients g_n, a trigger attribute for calculating said gradients $flag_g$, as well as W_n which represents a queue that contains pairs of parameters and local batches to evaluate. Lastly, Line 11 connects the PS parameter attribute to each learner and Line 12 connects learner gradients to the PS. The tasks described here can be found in Section 6.5.2

We implement Computing Graph 6.2 using `PlasmoCompute.jl` with Code Snippet 6.2. Line 2 create the `ComputingGraph` object, and Lines 4 through 11 setup the parameter server `ps` with its attributes, tasks, and triggers. Lines 13 through 25 setup the learner nodes where we note that we initialize the execution of the graph by scheduling edge communication on Line 24 for each edge that connects `ps` to a learner. We note that we use the machine learning package `Flux.jl` [65] to retrieve the MNIST data set, setup our loss function, and evaluate gradients (as shown Code Snippet 6.4 in Section 6.5.2).

Computing Graph 6.2 (K-Batch Asynchronous Stochastic Gradient Descent)

Parameter Server (ps)
Attributes: $\mathcal{A}_{ps} := (w_{ps}, G, flag_{ps}, \{g_{ps,1}, ..., g_{ps,N}\}$
Tasks: $\mathcal{T}_{ps} :=$ (`update_parameters`, `receive_gradient`)
  `update_parameters`: runs Task 6.5, triggered by: Updated($flag_{ps}$)
  `receive_gradient`: runs Task 6.4, triggered by: Received($g_{ps,n}$), $n \in \{1, ..., N\}$

Learners $(n \in \{1, ..., N\})$
Attributes: $\mathcal{A}_n := (w_n, g_n, flag_n, W_n)$
Tasks: $\mathcal{T}_n :=$ (`calculate_gradient`)
  `receive_parameters`: runs Task 6.6, triggered by: Received(w_n), $n \in \{1, ..., N\}$
  `calculate_gradient`: runs Task 6.7, triggered by: Updated($flag_n$)

Edges $\mathcal{E}_1 := w_{ps} \rightarrow w_n$, $n \in \{1, ..., N\}$, send on: Updated(w_{ps})
Edges $\mathcal{E}_2 := g_n \rightarrow g_{ps,n}$, $n \in \{1, ..., N\}$, send on: Updated(g_n)

Code Snippet 6.2: PlasmoCompute.jl snippet for simulating stochastic gradient descent.

```
#Create a computing graph
graph = ComputingGraph()

N = 3 #number of learners
#Add the parameter sever (ps)
@computenode(graph,ps)
@attributes(ps,w,G,flag,g[1:N])
@computetask(ps,update_parameters, compute_time = :walltime,
triggered_by=Updated(flag))
@computetask(ps,receive_gradient[i = 1:N], compute_time = 0,
triggered_by=Received(g[i]))

@computenode(graph,learners[1:N])
for (i,node) in enumerate(learners)
    #Add subnodes (n = 1:N) to solve sub-problems
    @attributes(node,w,g,flag)
    @computetask(node,compute_gradient, compute_time = :walltime,
    triggered_by=Updated(flag))

    #Connect attributes between master and subnodes
    edge = @connect(graph,ps[:w] => n[:w], send_on=Updated(ps[:w]), delay = 1)
    @connect(graph,n[:g] => ps[:g][i], send_on=Updated(n[:g]))

    schedule!(graph, edge, time = 0)
end
#Execute the computing graph
execute!(graph)
```

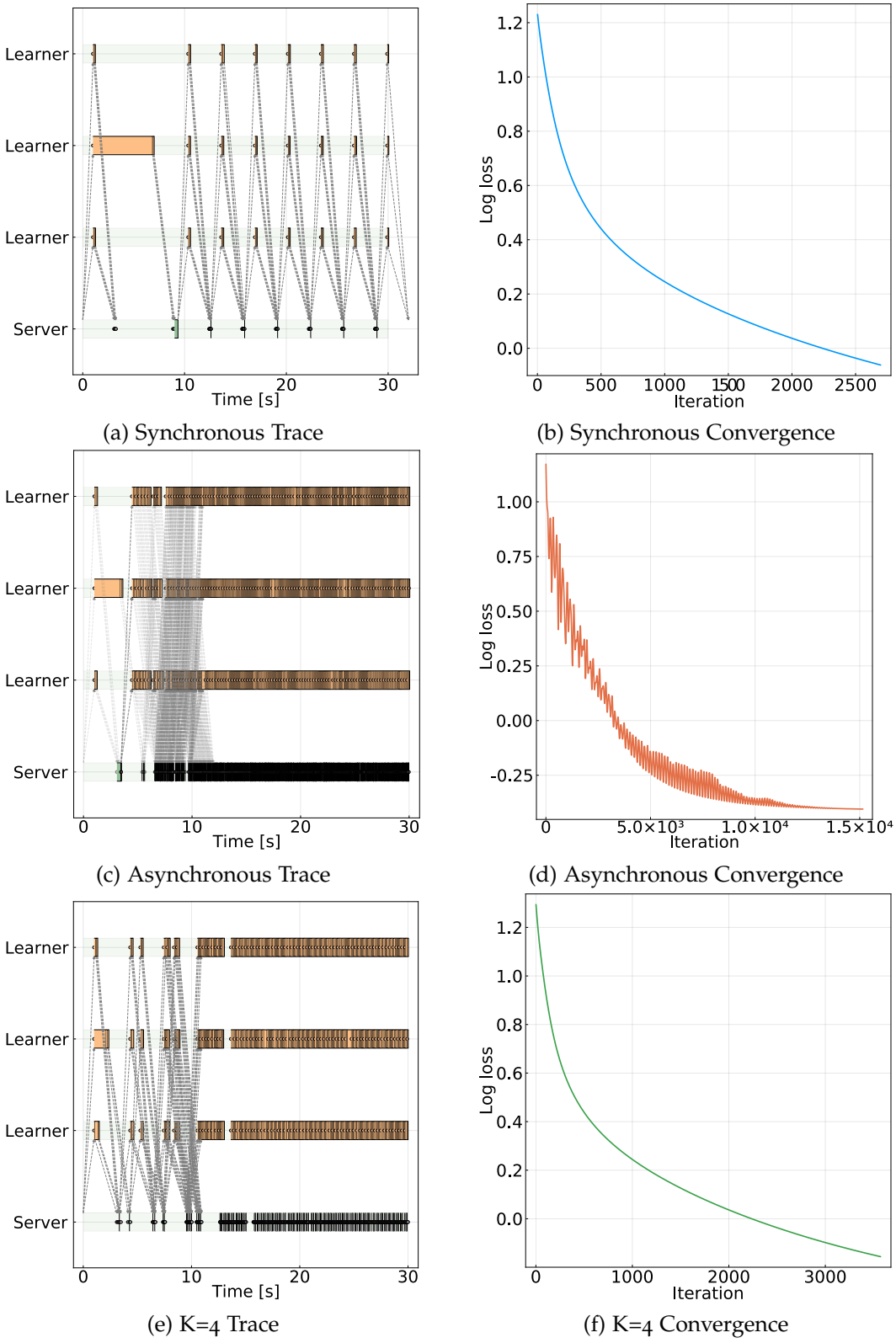
6.4.3 *Results*

Figure 6.5 summarizes the simulation results for the different algorithmic variants. The top figures (6.5a and 6.5b) show the trace plot and convergence results for the syn-

chronous variant. We observe a smooth convergence profile, but the trace plot depicts considerable idle time due to the communication overhead. The middle figures (6.5c and 6.5d) show the results of the asynchronous variant where we see the parallel efficiency is vastly improved (the learners and server are always running their computations). However, we also observe instability with in the convergence profile which suggests the variant could fail to converge without reliable training data. The K-Asynchronous variant (with K = 4) is shown at the bottom (Figures 6.5e and 6.5f) which falls between the purely synchronous and asynchronous variants. We observe a smooth convergence profile akin to the synchronous variant and almost the same parallel efficiency as the asynchronous variant.

Figure 6.6 directly compares the convergence and wall-time performance for the three variants. We see that although the synchronous algorithm demonstrates the fastest possible convergence with respect to iterations, it takes an order of magnitude more computational time (wall-time) to achieve the same loss value as the asynchronous case. To achieve a log loss value of 0, the synchronous variant requires almost 2 hours whereas the asynchronous variant only requires 2 minutes. From a machine learning standpoint, both algorithm variants can be considered undesirable. The long computation time required for the synchronous variant makes training large models prohibitive whereas the instability of purely asynchronous variant could cause issues with convergence. The K-Asynchronous variant strikes a balance between the synchronous and asynchronous extremes. The algorithm convergence is almost indistinguishable from the purely synchronous case and the instability has been eliminated.

We note that this study only scratches the surface of what types of SGD simulations are possible with a `ComputingGraph`. For instance, it would be straight-forward to implement dynamic data allocation and adaptive learning rates. It would also be possible to simulate other SGD variants that use more complex communication topologies such as in [107].



(a) Synchronous Trace

(b) Synchronous Convergence

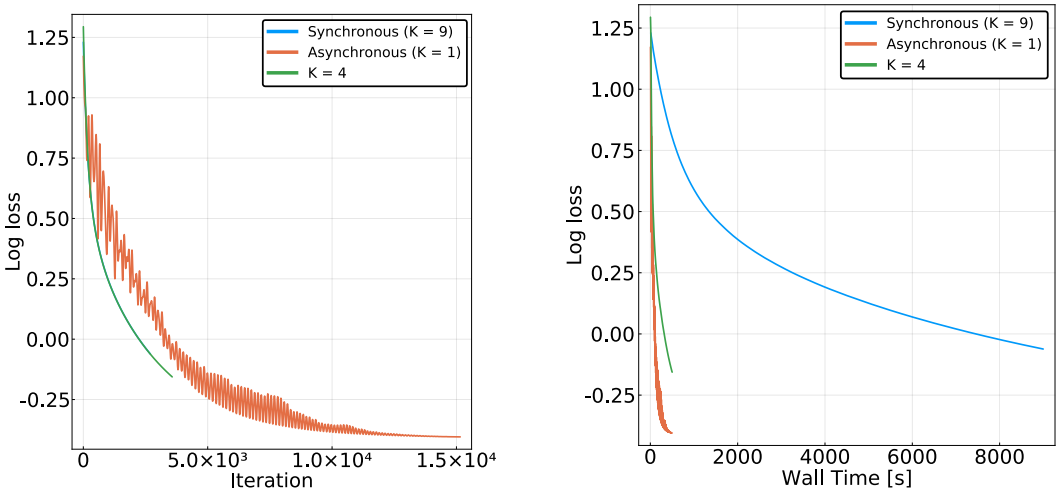
(c) Asynchronous Trace

(d) Asynchronous Convergence

(e) K=4 Trace

(f) K=4 Convergence

Figure 6.5: Communication patterns and convergence results of different stochastic gradient descent algorithmic variants.



Iterations Wall-time

Figure 6.6: Comparison of algorithmic variant performance for stochastic gradient descent.

6.5 Appendix: Case Study Models

This Section provides extra model details for the case studies presented in this Chapter. Section 6.5.1 provides the study model used for the Benders case study, as well as the compute tasks used for its `ComputingGraph`. This section also includes a sample `PlasmoCompute.jl` implementation of a compute task. Section 6.5.2 provides the compute tasks used for the SGD case study as well as an example `PlasmoCompute.jl` implementation.

6.5.1 *Benders Case Study Model*

The model for the two-stage stochastic resource management optimization problem is:

$$
\begin{aligned}
\min_{x,z,y} \quad & \sum_{\xi \in \Xi} \sum_{f \in \mathcal{F}} p_f(\xi) u_f(\xi) && (6.14)\\
\text{s.t.} \quad & \sum_{a \in \mathcal{A}_\mathcal{B}} c_a x_a + \sum_{j \in \mathcal{B}} h_j z_j \leq Budget \\
& v_j = \gamma_j + z_j + \sum_{a \in rec(\mathcal{A}_\mathcal{B}, j)} x_a - \sum_{a \in snd(\mathcal{A}_\mathcal{B}, j)} x_a \quad j \in \mathcal{B} \\
& q_j(\xi) = v_j - \sum_{a \in snd(\mathcal{A}_\mathcal{F}, j)} y_a(\xi) \quad j \in \mathcal{B} \\
& \sum_{a \in snd(\mathcal{A}_\mathcal{F}, f)} y_a(\xi) + u_f(\xi) \geq d_f(\xi) \quad f \in \mathcal{F},
\end{aligned}
$$

where Ξ is a set of realized scenarios, $\mathcal{B}$ is a set of bases each containing resources, $\mathcal{F}$ is a set of districts with resource demands, $\mathcal{A}$ is a set of arcs connecting bases and districts, $\mathcal{A}_\mathcal{B} \subseteq \mathcal{A}$ is the set of arcs between bases, and $\mathcal{A}_\mathcal{F}$ is the set of arcs between bases and districts. Parameter $p_f(\xi)$ is the probability that scenario ξ realizes at district f and variable $u_f(\xi)$ is the unmet demand at district f after dispatch decisions are made for scenario ξ. Variable $x \in \mathbb{R}^{|\mathcal{A}_\mathcal{B}|}$ is a first stage decision to move resources between bases, $z \in \mathbb{R}^{|\mathcal{B}|}$ is a first stage decision to purchase resources at bases, and variable $y \in \mathbb{R}^{|\mathcal{A}_\mathcal{F}|}$ is

a second stage decision to dispatch resources to districts after realizing district demands. Parameter γ_j is the initial amount of resources in base j, v_j is the amount of resources in base j after making transfers, variable $q_j(\xi)$ is the amount of resources in each base after dispatching to districts for scenario ξ, and parameter $d_f(\xi)$ is the resource demand of district f for scenario ξ.

Problem (6.14) is reformulated to conduct Benders decomposition by decomposing it into a master problem and subproblem for each scenario. The master problem is given by:

$$
\begin{aligned}
\min_{x,z,\theta_{cut}} \quad & \theta_{cut} && (6.15)\\
s.t. \quad & \sum_{a\in\mathcal{A}_\mathcal{B}} c_a x_a + \sum_{j\in\mathcal{B}} h_j z_j \leq Budget \\
& v_j = \gamma_j + z_j + \sum_{a\in rec(\mathcal{A}_\mathcal{B},j)} x_a - \sum_{a\in snd(\mathcal{A}_\mathcal{B},j)} x_a, \quad j \in \mathcal{B} \\
& x_a \geq 0, \quad a \in \mathcal{A} \\
& z_j \geq 0, \quad j \in \mathcal{B} \\
& v_j \geq 0, \quad j \in \mathcal{B} \\
& \theta_{cut} \geq 0 \\
& \theta_{cut} \geq c(x), \quad c \in C,
\end{aligned}
$$

where θ_{cut} is a variable to enforce cutting planes and C is the set of cutting planes added to the master problem. The subproblem is a function of the master solution variable $\hat{w}$

and scenario ξ and is given by (6.16).

$$
\begin{aligned}
Q(\hat{v}, \xi) := \min_{y} \quad & \sum_{f \in \mathcal{F}} p_f(\xi) u_f && (6.16)\\
\text{s.t.} \quad & q_j = \hat{w}_j - \sum_{a \in snd(\mathcal{A}_{\mathcal{F}}, j)} y_a, \quad j \in \mathcal{B} \\
& \sum_{a \in snd(\mathcal{A}_{\mathcal{F}}, j)} y_a + u_f \geq d_f(\xi), \quad f \in \mathcal{F} \\
& q_j \geq 0, \quad j \in \mathcal{B} \\
& y_a \geq 0, \quad a \in \mathcal{A}_{\mathcal{F}} \\
& u_f \geq 0, \quad f \in \mathcal{F}.
\end{aligned}
$$

Compute Tasks

For our Benders setup in Computing Graph 6.1, we denote the master node as m which contains two tasks described by Task 6.1 and Task 6.2. Each subnode in the graph consists of a single task described by Task 6.3. Task 6.1 runs the master problem (6.15) and updates the first stage solution $\hat{x}$ (which contains all first-stage variables) and checks whether convergence has been satisfied. If not satisfied, it updates the master node scenario attributes $\{\xi_1, ..., \xi_N\}$ with the first N values from the scenario set Ξ. Task 6.2 runs when the master node receives an update to a solution attribute s_n. The task checks whether every solution has returned, and if true, it updates the flag attribute $flag$, which indicates that the master problem is ready to be solved. If not all subproblem solutions have returned, the task updates the attribute ξ_n with the next scenario in Ξ. Task 6.3 solves the subproblem (6.16) given a first stage solution $\hat{x}$ and a scenario ξ and updates the subnode solution attribute s.

Task 6.1 `solve_master` (computing graph $\mathcal{CG}$,node m)

1: Get node attribute C: Set of current master problem cuts
2: Get node attribute S: Set of solutions received from sub-problems
3: Get node attribute Ξ: Set of sample scenarios
4: Get node attribute $\hat{x}$: Master problem solution attribute
5: Get node attributes $\{\xi_1, ..., \xi_N\}$: Scenarios to subnodes $\mathcal{N}$
6: Solve (6.15): `update attribute` $\hat{x} \leftarrow$ `solve_master_problem`(C)
7: **if** `problem_converged`$(\hat{x}, S)$: **then**
8: Stop Computing Graph $\mathcal{G}$
9: **else**
10: Start sending new scenarios: `update attributes` $\{\xi_1, ..., \xi_N\} \leftarrow \Xi[1:N]$
11: **end if**
12: Empty current solutions: `set` $S \leftarrow \{\}$

Task 6.2 `receive_solution`(node m, attribute s_n)

1: Get node attribute S: Set of solutions received from sub-problems
2: Get node attribute Ξ: Set of sample scenarios
3: Get argument s_n: Solution from subnode n
4: Get node attribute $\xi_n,\ n \in \mathcal{N}$: Subnode n scenario attribute
5: Get node attribute $flag$: Flag that master problem is ready to solve
6: `push to` $S \leftarrow$ `get_objective_value`(s_n)
7: `push to` $C \leftarrow$ `compute_new_cut`(s_n)
8: **if** `all_scenarios_complete`(S) **then**
9: Master problem is ready: `update attribute` $flag$
10: **else**
11: Send new scenario to node n : `update attribute` $\xi_n \leftarrow$ `new_scenario`(Ξ)
12: **end if**

Task 6.3 `solve_subproblem`(node n)

1: Get node attribute $\hat{x}$: Master solution received on subnode n
2: Get node attribute ξ: Scenario received on subnode n
3: Solve (6.16): `update attribute` $s \leftarrow$ `solve_subproblem`$(\hat{x}, \xi)$

PlasmoCompute.jl Implementation

Code Snippet 6.3 depicts how to define the master problem task in `PlasmoCompute.jl`. One argument is typically needed: a reference to a node to retrieve and update attributes, but a reference to the `ComputingGraph` can be provided for access to the graph clock (or global graph attributes) to terminate the computation. For example, line 3 retrieves the attribute value for the set of master problem cuts to solve the master problem, and

updates the master solution in line 6.

Code Snippet 6.3: Creating the `solve_master` compute task in `PlasmoCompute.jl`

```
# function implementing master task.
function solve_master(graph::ComputingGraph,node::ComputeNode)
    C = node[:C]
    S = node[:S]
    solution = solve_master_problem(C)
    node[:solution] = solution

    #Convergence check.
    lower_bound = objective_value(solution) #compute the lower bound given the master solution
    upper_bound = compute_upper_bound(S) #compute the upper bound from sub-problem solutions
    if converged(lower_bound,upper_bound)
        Stop(graph)
    end
    Ξ = node[:Ξ]
    #Start allocating scenarios. Each scenario update will trigger communication to the subnode.
    ζ = node[:ζ] #retrieve array of outgoing scenarios
    for i = 1:N
        ζ[i] = Ξ[i]
    end
end
```

6.5.2 *Stochastic Gradient Descent Implementation*

Here we provide details relating to the SGD case study in Section 6.4.

Compute Tasks

The compute tasks to receive gradients, receive parameters, compute gradients, and update parameters in Section 6.4 are given by Tasks 6.4, 6.5, 6.4, and 6.7. The implemented algorithm allocates sample batches to the learners up front. Thus we use xi_n to denote the batches that belong to learner n. For instance, if we allocated batches 1,2, and 3 to learner n, we would have $\xi_n := \{\xi_{b1}, \xi_{b2}, \xi_{b3}$.

Task 6.4 runs on the parameter server (node ps) which uses node data from ps and stores the received gradient g_n in the set of gradients G. If at least K gradients have been received, the task updates the $flag_{ps}$ which triggers Task 6.5. Task 6.5 uses the current set of gradients G to update the current parameters w_{ps} and evaluates the loss function to check tolerance to decide whether to terminate the `ComputingGraph` execution.

Task 6.6 runs on each learner node $n \in \{1, ..., N\}$. This task queries the attribute w_n

(the most recently received parameters), and adds the parameters along with the batches allocated to the learner ξ_n to the set W_n which contains pairs of batches and parameters to evaluate gradients with. If the queue W_n was originally empty, the task updates the parameter $flag_n$ which triggers Task 6.7. Finally, Task 6.7 computes the gradient of the loss function (6.13) and updates the attribute g_n. If there are still batches to evaluate in the queue W_n, the task updates $flag_n$ which re-triggers the task to evaluates the next gradient.

Task 6.4 `receive_gradient`(node ps, attribute g_n)

1: Get node attribute G: Set of gradients received from learners
2: Get argument g_n: Gradient received from learner n
3: Get node attribute $flag_{ps}$: Flag that master problem is ready to solve
4: `push to` $G \leftarrow$ `value`(g_n)
5: **if** `K_gradients_received`(CG) **then**
6: Ready to update parameters: `update attribute` $flag_{ps}$
7: **end if**

Task 6.5 `update_parameters`(graph $\mathcal{CG}$,node ps)

1: Get node attribute G: Set of gradients received from learners
2: Get node attribute w_{ps}: Current parameters
3: Update parameters: `update attribute` $w_{ps} \leftarrow solutionto$ (6.7)
4: Evaluate loss function (6.13)
5: **if** `within_tolerance`(w, Ξ): **then**
6: Stop $\mathcal{CG}$
7: **end if**
8: Empty current gradients: `set` $G \leftarrow \{\}$

Task 6.6 `receive_parameters`(learner n)

1: Get node attribute w_n: Current received parameter
2: Get node attribute W_n: Queue of batches to evaluate on n
3: `push to` $W_n \leftarrow \{\xi_n, w_n\}$: Add new batches to evaluate with latest parameter w_n
4: **if** `not_started`(n): **then**
5: Get node attribute $flag_n$: Flag that learner should compute next gradient
6: `update attribute` $flag_n$: Update to trigger Task 6.7
7: **end if**

Task 6.7 `compute_gradient`(node n)

1: Pop next batch $\{\xi_b, w\}$ from queue W_n
2: Compute gradient of loss function (6.13) with $\{\xi_b, w\}$: `update attribute` g_n
3: **if** `not_empty`(W): **then**
4: Get node attribute $flag_n$: Flag that learner should compute next gradient
5: `update attribute` $flag_n$: Update to trigger Task 6.7
6: **end if**

PlasmoCompute.jl Implementation

Code Snippet 6.4 shows how the `compute_gradient` task (Task 6.7) is implemented in `PlasmoCompute.jl`. Line 1 imports the `Flux.jl` machine learning package and Line 4 pops the next batch-parameter pair from the queue `node[:W]`. Lines 8 through 13 setup the loss function, Line 16 compute the gradient over the batch, Line 22 updates the `node[:gradient]` attribute, and Lines 25 through 27 update the `node[:flag]` attribute if there are still batches to evaluate in `node[:W]` which re-triggers the task.

Code Snippet 6.4: Creating `compute_gradient` task for SGD case study

```
1  using Flux
2  function compute_gradient(node::ComputeNode)
3      #Get current parameter values and data
4      task_data = popfirst!(node[:W])
5      W,b,X,Y = task_data
6
7      #Create our model to train.
8      W = param(W)
9      b = param(b)
10     layer(x) = W*x .+ b
11     m = Chain(layer,softmax)
12     lambda = 0.01
13     loss(x, y) = crossentropy(m(x), y) + lambda*norm(W)^2
14
15     #Calculate average gradient
16     gradients = Tracker.gradient(() -> loss(X, Y), params(W, b))
17
18     #Collect the gradient and the parameters used to calculate it
19     grad_result = [gradients,W,b]
20
21     #update gradient_result attribute
22     node[:gradient] = grad_result
23
24     #Update attribute to evaluate next batch in the task queue
25     if !isempty(node[:W])
26         trigger!(node[:flag])
27     end
28 end
```

Chapter 7

CONCLUSIONS AND FUTURE DIRECTIONS

We conclude this dissertation by highlighting the key contributions and discuss some new directions that build on the presented research.

7.1 Contributions

The primary contributions of this dissertation stem from newly presented modeling abstractions and their corresponding software implementations to solve challenging problems pertaining to cyber-physical systems.

Modeling and Decomposition with OptiGraphs

In Chapter 2 we presented the `OptiGraph`, a graph-based modeling abstraction that provides flexibility to systematically model optimization problems. A key benefit we showed is that the `OptiGraph` exploits modularity concepts to build up hierarchical optimization models and performs diverse data management tasks. We showed how the `OptiGraph` can be used to model challenging estimation problems and facilitates diverse modeling tasks such as warm-starting and model reduction.

In Chapter 3, we showed how the `OptiGraph` abstraction facilitates *automatic decomposition* strategies. Moreover, we showed how the abstraction naturally facilitates decompo-

sition at both the linear algebra and problem levels and can be used to create interfaces to new decomposition solvers and develop new algorithms.

Fundamentally, the `OptiGraph` is a em new modeling paradigm to create optimization problems. By expressing optimization problems with `OptiGraphs`, it opens up possibilities to create new solvers and to tackle challenging problems by exploiting decomposition.

Cyber Simulation with ComputingGraphs

In Chapter 5 we introduced the `ComputingGraph`, a new abstraction to simulate the real-time computing aspects that arise in cyber systems. The `ComputingGraph` generalizes computing concepts and works at a high-level of abstraction to capture asychronous timing aspects within complex communication topologies. We showed how the `ComputingGraph` can be used to simulate the real-time behavior in a cooperative control system and explore the effects of communication and controller failures.

Chapter 6 further expanded on the simulation aspects afforded by he `ComputingGraph` and showed how it can simulate distributed algorithms on heterogeneous architectures. Specifically, we showed how the `ComputingGraph` can be used to simulate algorithm scaling for distributed Benders decomposition and how it can benchmark different variants of stochastic gradient descent to train machine learning models.

In a similar spirit to the `OptiGraph`, the `ComputingGraph` exploits modular development concepts to develop *algorithms* that target distributed computing architectures. The abstraction facilitates new algorithmic advances and offers a new paradigm to analyze and develop algorithms that target distributed computing systems.

Software: `Plasmo.jl` **and** `PlasmoCompute.jl`

A major contribution of this dissertation is the presentation of new software platforms to model and simulate cyber-physical systems. Developing such software is key to utilizing new abstractions and approaching these problems in a scalable way.

Throughout Chapters 2,3, and 4 we showcased `Plasmo.jl`, a `Julia`-based package that

implements the OptiGraph. Plasmo.jl uses expressive syntax to model hierarchically optimization problems and to perform automatic decomposition. The package enables the use of different graph analysis and visualization tools and provides interfaces to use a broad range graph partitioning tools and decomposition solvers.

In Chapters 5 and 6 we demonstrated the ComputingGraph implementation in PlasmoCompute.jl. We showed how PlasmoCompute.jl uses concise statements to model and simulate real-time cyber systems and provides a powerful framework to study distributed control systems and develop distributed algorithms.

7.2 Future Research Directions

There are broad research opportunities that stem from the ideas presented in this dissertation.

Exploiting Graph Properties for Decomposition

Chapter 3 demonstrated how to partition OptiGraphs using powerful graph partitioning tools such as Metis and KaHyPar. Exploiting such graph partitioning tools creates new possibilities for using decomposition-based solvers, but it is limited to expressing problem characteristics strictly in the form of edge weights and node sizes. The OptiGraph framework motivates the development of more customized graph partitioning algorithms that work directly with optimization problem attributes (such as accounting for integer variables). There is also considerable interest in exploiting descriptive graph properties such as community structures and spanning trees within the partitioning framework. The current OptiGraph framework also motivates the development of partitioning algorithms that calculate overlap directly for overlapping domain decomposition algorithms as opposed to performing subgraph expansion.

Modeling Extensions and Algorithm Development

A key obstacle to the adoption of new state-of-the-art parallel optimization solvers is the lack of a coherent interfaces to use them. This dissertation showed how Plasmo.jl

interfaces to parallel solvers such as PIPS-NLP and facilitates distributed algorithm development with an overlapping Schwarz solver. Consequently, Plasmo.jl presents opportunities to utilize new solvers such as DSP [77] a dual-decomposition solver for stochastic optimization, and SNGO [24], a Julia-based global solver for nonlinear stochastic programs. It also motivates the adoption of new solvers that use advanced Schur decomposition strategies to solve massive nonlinear problems [108]. In this regard, the OptiGraph could be a natural interface to use solvers such as BELTISTOS [74] which exploits specialized time decomposition structures that arise in Schur decomposition.

Another key obstacle in parallel optimization is the lack of *parallel modeling* capabilities. Parallel modeling becomes important in instances when the optimization model itself is too large to fit in memory, such as with stochastic programs with many scenarios. Scenario-based optimization problems can be modeled in parallel using StructJuMP and MPI, but more complex structures (such as those with many linking constraints) are more challenging to implement in a parallel modeling framework. In this context, the Parallel Structured Model Generator (PSMG) [55] is one of the only frameworks that performs parallel model generation which works with structured conveying modeling language (SML) [31]. PSMG focuses on efficient distributed memory design to achieve parallel generation, but it requires solvers to perform subproblem distribution and load balancing based on a well-defined model structure. Parallel modeling extensions in Plasmo.jl would facilitate modeling, partitioning, and parallel generation tasks in a more systematic interactive framework using the OptiGraph.

Simulating High Performance Computing Architectures

There has been considerable research towards designing *energy efficient algorithms* [6] that execute on large-scale data and computing centers. Such research seeks to reduce the considerable electricity costs of large-scale computing centers (which far exceed the costs of computing hardware). Simulating the energy requirements of an algorithm however, requires a highly granular representation of the underlying computing architecture. By far, the most popular framework to simulate HPC systems is SimGrid [27] which is a

simulation toolkit to study algorithm scheduling for distributed computing applications. `SimGrid` has developed considerably over the last 20 years, but it can be considered fairly low-level which is restrictive when experimenting with new algorithmic ideas such as simulating power consumption.

The development of energy efficient algorithms can greatly benefit from new computing abstractions that work at a higher level such as the `ComputingGraph`. In this regard, there is a need to develop a hierarchical `ComputingGraph` in a similar spirit to the hierarchical `OptiGraph`. A hierarchical representation of the `ComputingGraph` could naturally capture *heterogeneous* architectures. For instance, a subgraph within a `ComputingGraph` could represent the elements of a graphical processing unit (GPU) which in turn could connect to other computing elements at a higher level of the graph. Such advances motivate further development of `PlasmoCompute.jl` to capture more specific HPC elements such as shared memory systems and more detailed communication effects such as memory and network bandwidth.

Swarm-Based Systems and Co-Simulation

Lastly, there are interesting opportunities to extend `ComputingGraph` capabilities to tackle the simulation of *swarm-based* systems [83, 115]. Such systems can contain thousands of devices that exhibit complex communication patterns which drive their underlying physical behavior. Consequently, the flexible modeling and simulation of swarm-based systems can open up new interesting research directions in cyber-physical systems. For instance, swarm-based modeling could allow for the simulation of sophisticated hierarchical control architectures that involve combinations of centralized/decentralized subsystems [122]. Swarm-based modeling could also facilitate the high-fidelity simulation of autonomous vehicles [117] which requires capturing both physical and cyber behaviors among many devices in a coherent way. Moreover, devices could also represent biological agents which could allow for the simulation of interactions within large biological communities [129].

In a sense, swarm-based systems can be described as highly distributed cyber-physical

systems, which in turn makes simulating their performance challenging. One possible approach to simulate swarm-based systems is to decompose the underlying system within a *co-simulation* [50] framework, which could perform the simulation in a distributed manner. Co-simulation offers promising tools to model and simulate swarm-based systems in a flexible way (e.g. the dynamics of each device can be simulated independently), but establishing a general co-simulation framework that can capture swarm-based behaviors has proven to be challenging. Such a framework needs to capture complex communication among the participating devices and needs to facilitate the *synchronization* between continuous and discrete dynamic behaviors that arise in swarm-based systems. Further development of the `ComputingGraph` could lead to a more generalizable co-simulation abstraction that enables the scalable simulation of complex swarm-based systems.

Appendix A

NATURAL GAS NETWORK MODELS

This Appendix is organized as follows. Section A.1 summarizes the model nomenclature to describe optimization problems over natural gas networks. Sections A.2 and A.3 describe model equations and implementations of junctions and compressors, and Section A.4 details the equations and implementation for different pipeline model formulations. Lastly, A.5 describes the network connections that couple the presented models.

A.1 Model Nomenclature

Here we introduce model nomenclature used throughout this dissertation to describe natural gas networks.

A.1.1 *Sets*

The relevant sets are described in Table A.1. $\mathcal{J}$ is the set of junctions in the gas network, $\mathcal{D}$ is the set of demands (gas withdrawals), and $\mathcal{S}$ is the set of supplies (gas injections). The set of demands and supplies that belong to a specific junction is denoted $\mathcal{D}_j$ and $\mathcal{S}_j$ respectively. The junctions are connected by links denoted $\mathcal{L}$ throughout the network. The set of links that denote pipelines is given by $\mathcal{L}_p$ and the set of links that denote compressors is denoted as $\mathcal{L}_c$. Lastly, the set of spatial discretization points for each link

is given by $\mathcal{X}$, and set of temporal discretization points is given by $\mathcal{T}$. It is possible to define separate spatial discretization sets for each pipeline (i.e. $\{\mathcal{X}_\ell, \ell \in \mathcal{L}_p\}$), but this dissertation simply applies the single set $\mathcal{X}$ to each pipeline link.

Table A.1: Sets

Set	Description	Elements
$\mathcal{J}$	Set of gas network nodes	$j \in \mathcal{J}$
$\mathcal{S}$	Set of gas supplies	$s \in \mathcal{S}$
$\mathcal{D}$	Set of gas demands	$d \in \mathcal{D}$
$\mathcal{D}_j$	Set of gas demands on junction j	$d \in \mathcal{D}_j$
$\mathcal{S}_j$	Set of gas supplies on junction j	$s \in \mathcal{S}_j$
$\mathcal{L}$	Set of gas network pipelines	$\ell \in \mathcal{L}$
$\mathcal{L}_p$	Set of network pipeline links	$\mathcal{L}_p \subseteq \mathcal{L}$
$\mathcal{L}_c$	Set of network compressor links	$\mathcal{L}_c \subseteq \mathcal{L}$
$\mathcal{X}$	Set of spatial discretization points	$k \in \mathcal{X}$
$\mathcal{T}$	Set of temporal discretization points	$t \in \mathcal{T}$

A.1.2 *Parameters*

Assuming ideal gas behavior, the square of the speed of sound in the gas c^2, the friction factor for each pipeline link λ_ℓ, the isentropic expansion coefficient γ, and the compression coefficient β can be computed from the following

$$
\begin{aligned}
c^2 &= \frac{\gamma z R T_{gas}}{M} \\
\lambda_\ell &= \left(2log_{10}\left(\frac{3.7D_\ell}{\epsilon_\ell}\right)\right)^{-2}, \ell \in \mathcal{L}_p \\
\gamma &= \frac{c_p}{c_v} \\
\beta &= \frac{\gamma - 1}{\gamma}
\end{aligned}
$$

where z is the compressibility factor of the gas, R is the universal gas constant, M is the gas molar mass, T_{gas} is the gas temperature, ϵ_ℓ is the pipe rugosity for each pipeline link, c_v is the gas heat capacity at constant volume, and c_p is the gas heat capacity at

constant pressure. We also have the following auxiliary constants which we use to define the scaled equations in A.13 given by the following

$$c_{1,\ell} = \frac{c^2}{A_\ell}\frac{\alpha_p}{\alpha_f} \qquad c_{2,\ell} = A_\ell\frac{\alpha_f}{\alpha_p} \quad c_{3,\ell} = \frac{8\lambda_\ell c^2 A_\ell}{\pi^2 D_\ell^5}\frac{\alpha_p}{\alpha_f} \qquad c_4 = \frac{1}{\alpha_f}c_p T_{gas}$$

where A_ℓ is pipeline cross sectional area, α_p and α_f are scaling coefficients, and D_ℓ is pipeline diameter. Table A.2 provides an overview of the parameters and units defined for gas network models.

Table A.2: Parameters and units

Parameter	Description	Units
Δx,Δt	Space and time discretization interval length	m, s
c	Speed of sound in gas	m/s
c_p, c_v	Gas heat capacity for constant pressure and volume	$2.34kJ/kgK, 1.85kJ/kgK$
γ, z	Isentropic expansion coefficient and compressibility	$-$
R	Universal gas constant	$8314J/kmolK$
M	Gas molar mass	$18kg/kmol$
ρ_n	Gas density at normal conditions	$0.72kg/m^3$
T	Gas temperature	K
L, D, A	Pipeline length, diameter, and cross sectional area	m, m, m^2
λ, ϵ	Friction factor and pipe rugosity	$-, m$
α_f	Scaling factor for flow	$\frac{3600}{1x10^4\rho_n}[=]\frac{SCMx10^4/h}{kg/s}$
α_p	Scaling factor for pressure	$1x10^{-5}bar/Pa$
c_1	Auxiliary constant	$\frac{bar/s}{SCMx10^4/h\ m}$
c_2	Auxiliary constant	$\frac{SCMx10^4/h-s}{bar/s}$
c_3	Auxiliary constant	$\frac{bar}{SCMx10^4-s/h\ m}$
c_4	Auxiliary constant	$\frac{kW}{SCMx10^4/h}$

A.1.3 *Variables*

All units pertaining to natural gas networks in this dissertation are given in SI units unless otherwise noted. Table A.3 defines the variables and units used throughout the presented gas network models.

Table A.3: Variables and units

Variable	Description	Units
t	Time	s
x	Spatial dimension	m
ρ	Density	kg/m^3
v	Velocity	m/s
V	Volumetric flow rate	m^3/s
p	Pressure	bar
f	Mass flow rate	kg/s
f_ℓ^{in}	Pipe inlet flow	kg/s
f_ℓ^{out}	Pipe outlet flow	kg/s
f_s	Supply flow	kg/s
f_d	Demand flow	kg/s
θ_j	Junction Pressure	bar
$\Delta\theta$	Boost Pressure	bar
η	Boost Ratio	$-$
P	Compressor Power	kW
m	Line-pack	kg

A.2 Junctions

Gas junctions are primarily what connect pipelines and equipment (i.e. compressors) in a natural gas network. In the context of day to day operations, operators seek to maintain junction pressures within prescribed limits.

A.2.1 *Model Equations*

The gas junction model is given in discrete form and is described by (A.1), where $\theta_{j,t}$ is the pressure at junction j and time interval t. $\underline{\theta}_j$ is the lower pressure bound for the junction, $\overline{\theta}_j$ is the upper pressure bound, $f_{j,d,t}^{target}$ is the target gas demand flow for demand d on junction j at time t, and $\overline{f}_{j,s}$ is the available gas generation from supply s on junction j.

$$\underline{\theta}_j \leq \theta_{j,t} \leq \overline{\theta}_j, \quad j \in \mathcal{J}, t \in \mathcal{T} \tag{A.1a}$$

$$0 \leq f_{j,d,t} \leq f_{j,d,t}^{target}, \quad d \in \mathcal{D}_j, \quad j \in \mathcal{J}, t \in \mathcal{T} \tag{A.1b}$$

$$0 \leq f_{s,t} \leq \overline{f}_{j,s}, \quad s \in \mathcal{S}_j, \quad j \in \mathcal{J}, t \in \mathcal{T} \tag{A.1c}$$

For some optimization models it is useful to enforce pressure ramping limits on supply junctions.

$$-\Delta\theta_s \leq \theta_{j(s),t+1} - \theta_{j(s),t} \leq \Delta\theta_s, \quad s \in \mathcal{S}, t \in \mathcal{T} \tag{A.2}$$

where $\Delta\theta_s$ is a pressure ramping limit on supply s and $j(s)$ denotes the junction that corresponds to supply s.

A.2.2 *Junction OptiGraph*

Code Snippet A.1 shows how to create the junction `OptiGraph`. We define the function `create_junction_model` on Line 2 which accepts junction specific `data` and the number of time periods `nt`. We create the `OptiGraph` (`graph`) on Lines 3 through 26 where we add an `OptiNode` for each time interval and then create the variables and constraints for each node in a loop. We also use the `JuMP` specific `@expression` macro to refer to expressions for total gas supplied, total gas delivered, and total cost for convenience. The junction model is returned from the function on Line 29.

Code Snippet A.1: Creating a gas junction model-graph

```
#Define function to create junction model−graph
function create_junction_model(data,nt)
    graph = OptiGraph()

    #Add model−node for each time interval
    @node(graph,nodes[1:nt])

    #query number of supply and demands on the junction
    n_demands = length(data[:demand_values])
    n_supplies = length(data[:supplies])

    #Loop and create variables, constraints, and objective for each model−node
    for (i,node) in enumerate(nodes)
        @variable(node, data[:pmin] <= pressure <= data[:pmax], start = 60)
        @variable(node, 0 <= fgen[1:n_supplies] <= 200, start = 10)
        @variable(node, fdeliver[1:n_demands] >= 0)
        @variable(node, fdemand[1:n_demands] >= 0)

        @constraint(node,[d = 1:n_demands],fdeliver[d] <= fdemand[d])

        @expression(node, total_supplied, sum(fgen[s] for s = 1:n_supplies))
        @expression(node, total_delivered,sum(fdeliver[d] for d = 1:n_demands))
        @expression(node, total_delivercost,sum(1000*fdeliver[d] for d = 1:n_demands))

        @objective(node,Min,total_delivercost)
    end

    #Return the junction OptiGraph
    return graph
end
```

A.3 Compressors

Compressors institute the primary control decisions in a natural gas network. In day to day operations, operators manipulate compressors to boost natural gas pressure to make deliveries and manage total system line-pack.

A.3.1 *Model Equations*

Isentropic Compressor

We use an ideal isentropic compressor model given by (A.3). (A.3a) describes the compressor boost where $\eta_{\ell,t}$,$p^{in}_{\ell,t}$, and $p^{out}_{\ell,t}$ are the compression ratio, suction pressure, and discharge pressure at time t, and (A.3b) denotes the ideal power requirement where $P_{\ell,t}$ is power at time t. (A.3c) defines mass flow rate through the compressor and uses the

dummy variables $f^{in}_{\ell,t}$ and $f^{out}_{\ell,t}$ to be consistent with the pipeline model in Section A.4.

$$p^{out}_{\ell,t} = \eta_\ell p^{in}_{\ell,t}, \quad \ell \in \mathcal{L}_c, \quad t \in \mathcal{T} \tag{A.3a}$$

$$P_{\ell,t} = c_p \cdot T \cdot f_{\ell,t} \left(\left(\frac{p^{out}_{\ell,t}}{p^{in}_{\ell,t}} \right)^{\frac{\gamma-1}{\gamma}} - 1 \right), \quad \ell \in \mathcal{L}_c, \quad t \in \mathcal{T} \tag{A.3b}$$

$$f_{\ell,t} = f^{in}_{\ell,t} = f^{out}_{\ell,t}, \quad \ell \in \mathcal{L}_c, \quad t \in \mathcal{T} \tag{A.3c}$$

Discrete Compressor Decisions

For some problems it is sensible to use a more simplified compressor description that removes the nonlinear terms. For instance, the mixed-integer linear formulation in Section 4.6 uses (A.4) where (A.4a) describes linear compressor boost (as opposed to the more realistic compression ratio) and (A.4b) is the same as in (A.3).

$$p^{out}_{\ell,t} = p^{in}_{\ell,t} + \Delta p_{\ell,t}, \quad \ell \in \mathcal{L}_c, \quad t \in \mathcal{T} \tag{A.4a}$$

$$f_{\ell,t} = f_{\ell,t,in} = f_{\ell,t,out}, \quad \ell \in \mathcal{L}_c, \quad t \in \mathcal{T} \tag{A.4b}$$

We can enforce discrete compressor decisions using the formulation in (A.5). Here we have introduced the binary variables $\{y_{\ell,t}\}_{\ell \in \mathcal{L}_c, t \in \mathcal{T}}$ which denote whether compressor ℓ is on or off at time period t. Equations (A.5b) and (A.5c) force boost pressures to be zero when the compressor is off, and operating between boost limits Δp^L_ℓ and Δp^U_ℓ otherwise.

$$y_{\ell,t} \in \{0,1\} \quad \ell \in \mathcal{L}_c, \quad t \in \mathcal{T} \tag{A.5a}$$

$$\Delta p_{\ell,t} \leq y_{\ell,t} \Delta p^U_\ell, \quad \ell \in \mathcal{L}_c, \quad t \in \mathcal{T} \tag{A.5b}$$

$$\Delta p_{\ell,t} \geq y_{\ell,t} \Delta p^L_\ell, \quad \ell \in \mathcal{L}_c, \quad t \in \mathcal{T} \tag{A.5c}$$

It is also useful to specify compressor ramping limits for the linear case (such limits are helpful in models that do not adequately capture dynamic behavior). Compressor discharge pressure ramp limits are given by (A.6) where $\Delta p^{out}_{\ell,t}$ represents the ramp limit

on the discharge pressure. It is also possible to set ramping limits on suction, boost pressures, or compression ratios which may depend on specific compressor operating modes.

$$-\Delta p^{out}_{\ell,t} \leq p^{out}_{\ell,t+1} - p^{out}_{\ell,t} \leq \Delta p^{out}_{\ell,t}, \quad \ell \in \mathcal{L}_c, \quad t \in \mathcal{T} \tag{A.6}$$

Equation (A.7) represents conditional ramping, such that discharge pressure ramp limits are only active if the compressor is on.

$$y_{\ell,t} \rightarrow (\text{A.6}), \quad \ell \in \mathcal{L}_c, \quad t \in \mathcal{T} \tag{A.7}$$

Lastly, Equation (A.8) represents minimum up and down times which helps to avoid solutions where compressors are turns on and off in quick succession.

$$\sum_{i=t}^{t+UT-1} y_{\ell,i} \geq UT(y_{\ell,t} - y_{\ell,t-1}), \quad \ell \in \mathcal{L}_c, \quad t \in \mathcal{T}_{UT} \tag{A.8a}$$

$$\sum_{i=t}^{t+DT-1} w_{\ell,i} \geq DT(w_{\ell,t} - w_{\ell,t-1}), \quad \ell \in \mathcal{L}_c, \quad t \in \mathcal{T}_{DT} \tag{A.8b}$$

Here, UT is the maximum up time and DT is the maximum down time in terms of number of time intervals. We also define $\mathcal{T}_{UT} = \{1, ..., t_N - UT + 1\}$ and $\mathcal{T}_{DT} = \{1, ..., t_N - DT + 1\}$ for convenience where t_N is the final time interval in $\mathcal{T}$.

A.3.2 *Compressor OptiGraph*

The compressor `OptiGraph` construction is straightforward as shown in Code Snippet A.2 for the isentropic model (A.3). Line 1 defines a function to create a compressor `OptiGraph` given `data` and `nt`. Lines 3 through 16 create the compressor `OptiGraph` by creating nodes, variables, and constraints. Lines 19 and 20 define expressions to refer to flow in and out of the compressor, and Line 23 returns the created `OptiGraph`.

Code Snippet A.2: Creating a compressor `OptiGraph`

```
function create_compressor_model(data,nt)
    #Create compressor model−graph
    graph = OptiGraph()
    @node(graph,nodes[1:nt])

    #Setup variables, constraints, and objective
    for node in nodes
        @variable(node, 1 <= psuction <= 100)
        @variable(node, 1 <= pdischarge <= 100)
        @variable(node, 0 <= power <= 1000)
        @variable(node, flow >= 0)
        @variable(node, 1 <= eta <= 2.5)
        @NLnodeconstraint(node, pdischarge == eta*psuction)
        @NLnodeconstraint(node, power == c4*flow*((pdischarge/psuction)^om-1) )
        @objective(node, Min, cost*power*(dt/3600.0))
    end

    #Create references for flow in and out
    @expression(graph,fin[t=1:nt],nodes[t][:flow])
    @expression(graph,fout[t=1:nt],nodes[t][:flow])

    #Return compressor OptiGraph
    return graph
end
```

A.4 Pipelines

This section presents pipeline equations to describe the dynamic transport throughout gas networks. For all of the pipeline models we assume isothermal flow through horizontal pipeline segments with constant pipe friction and no flow reversals.

A.4.1 *Model Equations*

Full Isothermal Euler

We first present what are called the Euler equations to describe mass and momentum conservation in gas pipelines. The original conservation equations are presented in [97] and take the form given in (A.9)

$$\frac{\partial \rho_\ell(t,x)}{\partial t} + \frac{\partial(\rho_\ell(t,x)v_\ell(t,x))}{\partial x} = 0 \tag{A.9a}$$

$$\frac{\partial(\rho_\ell(t,x)v_\ell(t,x))}{\partial t} + \frac{\partial(\rho_\ell(t,x)v_\ell(t,x)^2 + p_\ell(t,x))}{\partial x} = -\frac{\lambda_\ell}{2D}\rho_\ell(t,x)v_\ell(t,x)\left|v_\ell(t,x)\right|, \tag{A.9b}$$

where notation and units are defined in Tables A.2 and A.3. Here, the link diameter is D_ℓ and the friction coefficient is λ_ℓ. The states of each link vary in space and time and are given by the gas density $\rho_\ell(t,x)$, gas velocity $v_\ell(t,x)$, and gas pressure $p_\ell(t,x)$. From these fundamental quantities we can derive expressions for transversal area A_ℓ, volumetric flow $q_\ell(t,x)$, and mass flow rate $f_\ell(t,x)$ given by (A.10).

$$A_\ell = \frac{1}{4}\pi D_\ell^2 \tag{A.10a}$$

$$q_\ell(t,x) = v_\ell(t,x)A_\ell \tag{A.10b}$$

$$f_\ell(t,x) = \rho_\ell(t,x)v_\ell(t,x)A_\ell. \tag{A.10c}$$

For an ideal gas, the density and pressure are related by the gas speed of sound c (A.11):

$$\frac{p_\ell(t,x)}{\rho_\ell(t,x)} = c^2 \tag{A.11}$$

Typically, gas pipeline operators monitor pressure and mass flow rate since they can be directly measured. To represent the transport equations in terms of these states, we use relations (A.10) and (A.11) to cast (A.9) into (A.12). The final form of the transport equations is given by (A.12)

$$\frac{\partial p_\ell(t,x)}{\partial t} + \frac{c^2}{A_\ell}\frac{\partial f_\ell(t,x)}{\partial x} = 0, \quad \ell \in \mathcal{L}_p \tag{A.12a}$$

$$\begin{aligned}\frac{\partial f_\ell(t,x)}{\partial t} &+ \frac{2c^2}{A_\ell}\frac{f_\ell(t,x)}{p_\ell(t,x)}\frac{\partial f_\ell(t,x)}{\partial x} \\ &- \frac{c^2}{A_\ell}\frac{f_\ell(t,x)^2}{p_\ell(t,x)^2}\frac{\partial p_\ell(t,x)}{\partial x} + A_\ell\frac{\partial p_\ell(t,x)}{\partial x} = -\frac{8c^2\lambda_\ell A_\ell}{\pi^2 D_\ell^5}\frac{f_\ell(t,x)}{p_\ell(t,x)}\left|f_\ell(t,x)\right|, \quad \ell \in \mathcal{L}_p\end{aligned} \tag{A.12b}$$

We can improve the numerical behavior of formulation (A.12) by scaling pressure and flow variables. We use the constants $c_{1,\ell}$, $c_{2,\ell}$ and $c_{3,\ell}$ defined in Section A.1 and cast the

Euler equations into the scaled form given by (A.13).

$$\frac{\partial p_\ell(t,x)}{\partial t} = -c_{1,\ell}\frac{\partial f_\ell(t,x)}{\partial x}, \quad \ell \in \mathcal{L} \tag{A.13a}$$

$$\frac{\partial f_\ell(t,x)}{\partial t} = -2 \cdot c_{1,\ell}\frac{f_\ell(t,x)}{p_\ell(t,x)}\frac{\partial f_\ell(t,x)}{\partial x} + c_{1,\ell}\frac{f_\ell(t,x)^2}{p_\ell(t,x)^2}\frac{\partial p_\ell(t,x)}{\partial x} - c_{2,\ell}\frac{\partial p_\ell(t,x)}{\partial x} - c_{3,\ell}\frac{f_\ell(t,x)}{p_\ell(t,x)}\left|f_\ell(t,x)\right|, \quad \ell \in \mathcal{L}_p. \tag{A.13b}$$

The transport equations are highly nonlinear and computationally challenging to solve, particularly when large networks and long time horizons are considered. Here we present different approximate models proposed in literature [62] which are used in Section 2.4.

Approximate Euler

The euler equations can be approximated by dropping the the momentum term $\partial_x(\rho v^2)$ which has a negligible effect on dynamics, and the states ρ and v can be converted into mass flow rate f and pressure p to produce the formulation in (A.14).

$$\frac{\partial p_\ell(t,x)}{\partial t} + c_{1,\ell}\frac{\partial f_\ell(t,x)}{\partial x} = 0, \ \ell \in \mathcal{L}_p \tag{A.14a}$$

$$\frac{\partial f_\ell(t,x)}{\partial t} + c_{2,\ell}\frac{\partial p_\ell(t,x)}{\partial x} + c_{3,\ell}\frac{f_\ell(t,x)}{p_\ell(t,x)}\left|f_\ell(t,x)\right| = 0. \ \ell \in \mathcal{L}_p \tag{A.14b}$$

This is a coupled set of wave equations for flow and pressure with a nonlinear source term. This system is easier to solve than the Euler equations but it is still a challenging hyperbolic equation. The flow and pressure variables can also be used to define boundary conditions for the link flows. The flow boundaries are given by (A.15)

$$f_\ell(t, L_\ell) = f_\ell^{out}(t), \quad \ell \in \mathcal{L}_p \tag{A.15a}$$

$$f_\ell(t, 0) = f_\ell^{in}(t), \quad \ell \in \mathcal{L}_p. \tag{A.15b}$$

(A.15c)

Pressure boundaries are defined by the following conditions:

$$p_\ell(t, L_\ell) = p_\ell^{out}(t) \quad \ell \in \mathcal{L}_p \tag{A.16}$$

$$p_\ell(t, 0) = p_\ell^{in}(t) \quad \ell \in \mathcal{L}_p \tag{A.17}$$

Each pipeline may also include a steady-state initial condition:

$$c_{1,\ell} \frac{\partial f_\ell(0,x)}{\partial x} = 0 \quad \ell \in \mathcal{L}_p \tag{A.18a}$$

$$c_{2,\ell} \frac{\partial p_\ell(0,x)}{\partial x} + c_{3,\ell} \frac{f_\ell(t,x)}{p_\ell(t,x)} |f_\ell(t,x)| = 0 \quad \ell \in \mathcal{L}_p, \tag{A.18b}$$

and an operational constraint to return its line-pack back to its starting inventory:

$$m_\ell(T) \geq m_\ell(0) \quad \ell \in \mathcal{L}_p. \tag{A.19}$$

Here, the amount of line-pack in any pipeline segment m_ℓ can be computed as:

$$m_\ell(t) = \frac{A_\ell}{c^2} \int_0^{L_\ell} p_\ell(t,x) dx. \tag{A.20}$$

For implementation, these PDEs are discretized in space-time by using finite differences to produce (A.21)

$$\frac{p_{\ell,t+1,k} - p_{\ell,t,k}}{\Delta t} = -c_{1,\ell} \frac{f_{\ell,t+1,k+1} - f_{\ell,t+1,k}}{\Delta x_\ell}, \ell \in \mathcal{L}_p, t \in \mathcal{T}, k \in \mathcal{X}_\ell, \tag{A.21a}$$

$$\frac{f_{\ell,t+1,k} - f_{\ell,t,k}}{\Delta t} = -c_{2,\ell} \frac{p_{\ell,t+1,k+1} - p_{\ell,t+1,k}}{\Delta x_\ell} - c_{3,\ell} \frac{f_{\ell,t+1,k} |f_{\ell,t+1,k}|}{p_{\ell,t+1,k}}, \tag{A.21b}$$

$\ell \in \mathcal{L}_p, t \in \mathcal{T}, k \in \mathcal{X}_\ell$

$$f_{\ell,t,N_x} = f_{\ell,t}^{out}, \quad \ell \in \mathcal{L}_p, \quad t \in \mathcal{T} \tag{A.21c}$$

$$f_{\ell,t,1} = f_{\ell,t}^{in}, \quad \ell \in \mathcal{L}_p, \quad t \in \mathcal{T} \tag{A.21d}$$

$$p_{\ell,t,N_x} = p_{\ell,t}^{out}, \quad \ell \in \mathcal{L}_p, \quad t \in \mathcal{T} \tag{A.21e}$$

$$p_{\ell,t,1} = p_{\ell,t}^{in}, \quad \ell \in \mathcal{L}_p, \quad t \in \mathcal{T} \tag{A.21f}$$

We can also express a steady-state initial condition according to (A.22). Such a condition is typical in natural gas optimal control problems since operators typically seek to return the system to steady-state every day.

$$\frac{f_{\ell,1,k+1} - f_{\ell,1,k}}{\Delta x} = 0, \quad \ell \in \mathcal{L}_p, k \in \mathcal{X}_\ell \tag{A.22a}$$

$$c_{2,\ell}\frac{p_{\ell,1,k} - p_{\ell,1,k}}{\Delta x} + c_3\frac{f_{\ell,1,k}\,|f_{\ell,1,k}|}{p_{\ell,1,k}} = 0, \quad \ell \in \mathcal{L}_p, k \in \mathcal{X}_\ell \tag{A.22b}$$

The line-pack constraint can also be approximated in discrete form by (A.23).

$$m_{\ell,t} = \frac{A_\ell}{c^2}\sum_{k=1}^{N_x} p_{\ell,t,k}\Delta x_\ell, \quad \ell \in \mathcal{L}_p, t \in \mathcal{T} \tag{A.23a}$$

$$m_{\ell,N_t} \geq m_{\ell,1}, \quad \ell \in \mathcal{L}_p. \tag{A.23b}$$

Quasi-Static Approximation

The quasi-static approximation [61] can also be derived from the isothermal Euler equations. We obtain this form by neglecting the momentum $\partial_x(\rho v^2)$ term and the partial derivative $\partial_t(\rho v)$. This gives the following formulation given by (A.24)

$$\frac{\partial p_\ell(t,x)}{\partial t} + c_{1,\ell}\frac{\partial f_\ell(t,x)}{\partial x} = 0 \tag{A.24a}$$

$$c_{2,\ell}\frac{\partial p_\ell(t,x)}{\partial x} + c_{3,\ell}\frac{f_\ell(t,x)}{p_\ell(t,x)}\,|f_\ell(t,x)| = 0 \tag{A.24b}$$

This form of the Euler equations is mostly used as an approximation to capture long-term (planning) behavior and can be used as a proxy to capture situations under which the gas network is under "calm" conditions.

Steady-State Model

The steady-state version of the Euler conditions is given by (A.25). Steady-state models do not capture important dynamic behaviors in natural gas pipelines, and they cannot be used to manipulate line-pack, but they are often useful long term for planning problems

or for incorporating into time-coupled steady-state formulations.

$$c_{1,\ell}\frac{\partial f_\ell(t,x)}{\partial x} = 0 \tag{A.25a}$$

$$c_{2,\ell}\frac{\partial p_\ell(t,x)}{\partial x} + c_{3,\ell}\frac{f_\ell(t,x)}{p_\ell(t,x)}\left|f_\ell(t,x)\right| = 0 \tag{A.25b}$$

The steady-state equations can be presented in discrete form with (A.26).

$$c_{1,\ell}\frac{f_{\ell,t,k+1} - f_{\ell,t,k}}{\Delta x_\ell} = 0, \quad \ell \in \mathcal{L}_p, t \in \mathcal{T}, k \in \overline{\mathcal{X}}_\ell \tag{A.26a}$$

$$c_{2,\ell}\frac{p_{\ell,t,k+1} - p_{\ell,t,k}}{\Delta x_\ell} + c_{3,\ell}\frac{f_{\ell,t,k}}{p_{\ell,t,k}}\left|f_{\ell,t,k}\right| = 0, \quad \ell \in \mathcal{L}_p, t \in \mathcal{T}, k \in \overline{\mathcal{X}}_\ell \tag{A.26b}$$

Linearized Pipeline Model

Using a linearized representation of pipeline dynamics introduces considerable model simplifications, but such assumptions can be incorporated into discrete decision formulations which can be used to pose computationally tractable mixed-integer linear programs. A linearized model based on the steady state model (A.26) is given by (A.27).

$$c_{1,\ell}\frac{f_{\ell,t,k+1} - f_{\ell,t,k}}{\Delta x_\ell} = 0, \quad \ell \in \mathcal{L}_p, t \in \mathcal{T}, k \in \overline{\mathcal{X}}_\ell \tag{A.27a}$$

$$c_{2,\ell}\frac{p_{\ell,t,k+1} - p_{\ell,t,k}}{\Delta x_\ell} + c_{3,\ell}R_{\ell,t,k} = 0, \quad \ell \in \mathcal{L}_p, t \in \mathcal{T}, k \in \overline{\mathcal{X}}_\ell \tag{A.27b}$$

$$R_{\ell,t,k} = \left(\nabla_{f^*_{\ell,t,k}, p^*_{\ell,t,k}} \frac{f^2_{\ell,t,k}}{p^2_{\ell,t,k}}\right)^T \begin{pmatrix} f_{\ell,t,k} - f^*_{\ell,t,k} \\ p_{\ell,t,k} - p^*_{\ell,t,k} \end{pmatrix}, \quad \ell \in \mathcal{L}_p, t \in \mathcal{T}, k \in \mathcal{X}_\ell \tag{A.27c}$$

In this formulation, $R_{\ell,t,k}$ captures the linearized friction factor term in the momentum equation (A.27b). In this formulation, we linearize around the point $\{f^*_{\ell,t,k}, p^*_{\ell,t,k}\}_{\ell \in \mathcal{L}_p, t \in \mathcal{T}, k \in \mathcal{X}_\ell}$ which can be taken from the solution of a higher fidelity physical model.

A.4.2 *Approximate Euler Pipeline OptiGraph*

We express the pipeline model for Approximate Euler case as an `OptiGraph` with `OptiNodes` distributed on a space-time grid. Specifically, the nodes of each pipeline `OptiGraph` form a $N_t \times N_x$ grid wherein pressure and flow variables are assigned to each node. Flow dynamics within pipelines are then expressed with linking constraints that describe the discretized PDE equations for mass and momentum using finite differences. Code Snippet A.3 constructs the pipeline `OptiGraph`. Line 1 defines a pipeline function that takes pipeline specific `data`, the number of time points to use (`nt`), and the number of space points (`nx`). Lines 3 through 25 create the pipeline `OptiGraph` using data and adds `OptiNodes` in the form a space-time grid on Line 10. Lines 28 through 42 define linking constraints that represent the discretized pipeline dynamic equations.

Code Snippet A.3: Creating a pipeline `OptiGraph`

```
function create_pipeline_model(data,nt,nx)
    #Unpack data
    c1 = data[:c1]; c2 = data[:c2]; c3 = data[:c3]
    dx = data[:pipe_length] / (nx - 1)

    #Create pipeline model-graph
    graph = OptiGraph()

    #Create grid of optinodes
    @node(mg,grid[1:nt,1:nx])

    #Create variables on each node in the grid
    for node in grid
        @variable(node, 1 <= px <= 100)
        @variable(node, 0 <= fx <= 100)
        @variable(node,slack >= 0)
        @NLnodeconstraint(node, slack*px - c3*fx*fx == 0)
    end

    #Setup dummy variable references
    @expression(mg,fin[t=1:nt],grid[:,1][t][:fx])
    @expression(mg,fout[t=1:nt],grid[:,end][t][:fx])
    @expression(mg,pin[t=1:nt],grid[:,1][t][:px])
    @expression(mg,pout[t=1:nt],grid[:,end][t][:px])
    @expression(mg,linepack[t=1:nt],c2/A*sum(grid[t,x][:px]*dx for x in 1:nx-1))

    #Finite differencing. Backward difference in time from t, Forward difference in space from x.
    @linkconstraint(mg, press[t=2:nt,x=1:nx-1],
    (grid[t,x][:px]-grid[t-1,x][:px])/dt +
    c1*(grid[t,x+1][:fx] - grid[t,x][:fx])/dx == 0)

    @linkconstraint(mg, flow[t=2:nt,x=1:nx-1],(grid[t,x][:fx] -
    grid[t-1,x][:fx])/dt == -c2*(grid[t,x+1][:px] -
    grid[t,x][:px])/dx - grid[t,x][:slack])

    #Initial steady state
    @linkconstraint(mg,ssflow[x=1:nx-1],grid[1,x+1][:fx] - grid[1,x][:fx] == 0)
    @linkconstraint(mg,sspress[x = 1:nx-1], -c2*(grid[1,x+1][:px] -
    grid[1,x][:px])/dx - grid[1,x][:slack] == 0)

    #Refill pipeline linepack
    @linkconstraint(mg,linepack[end] >= linepack[1])
    return graph
end
```

A.5 Network Connections

The network connections define the topology that connect junctions and equipment links (i.e. pipelines and compressors). Specifically, the network equations express mass conservation around each junction and boundary conditions for pipelines and compressors.

A.5.1 *Model Equations*

Mass conservation around each junction j is given by (A.28a)

$$\sum_{\ell \in \mathcal{L}_{rec}(j)} f^{out}_{\ell,t} - \sum_{\ell \in \mathcal{L}_{snd}(j)} f^{in}_{\ell,t} + \sum_{s \in \mathcal{S}_j} f_{j,s,t} - \sum_{d \in \mathcal{D}_j} f_{j,d,t} = 0, \quad j \in \mathcal{J} \tag{A.28a}$$

where we define $\mathcal{L}_{rec}(j)$ and $\mathcal{L}_{snd}(j)$ as the set of receiving and sending links to each junction j respectively. Equations (A.29a) and (A.29b) define pipeline and compressor link boundary conditions.

$$p^{in}_{\ell,t} = \theta_{rec(\ell),t}, \quad \ell \in \mathcal{L}, t \in \mathcal{T} \tag{A.29a}$$

$$p^{out}_{\ell,t} = \theta_{snd(\ell),t}, \quad \ell \in \mathcal{L}, t \in \mathcal{T} \tag{A.29b}$$

Here, $\theta_{rec(\ell),t}$ and $\theta_{snd(\ell),t}$ are the receiving and sending junction pressure for each link $\ell \in \mathcal{L}$ at time t.

A.5.2 *Network OptiGraph*

The network structure can be induced with an `OptiGraph` by using a higher level graph to capture (A.28) and (A.29). Code Snippet A.4 formulates the complete gas network optimal control model. We create the `OptiGraph` `gas_network` on Line 8 and add the component `OptiGraphs` on Lines 13 through 32. Once we have the multi-level subgraph structure we create linkconstraints at the `gas_network` level on Lines 35 through 60 which impose the junction conservation and boundary conditions for the network links (the pipelines and compressors).

Code Snippet A.4: Formulating the complete gas network `OptiGraph`

```
function create_gas_network(net_data)
    pipe_data = net_data[:pipeline_data]
    comp_data = net_data[:comp_data]
    junc_data = net_data[:junc_data]
    pipe_map = net_data[:pipe_map]; comp_map = net_data[:comp_map]

    #Create OptiGraph for entire gas network
    network = OptiGraph()
    network[:pipelines] = [];network[:compressors] = [];network[:junctions] = []
    j_map = Dict()

    #Create device OptiGraphs and setup data structures
    for j_data in junc_data
        junc= create_junction_optigraph(j_data)
        add_subgraph!(network,junc); push!(network[:junctions],junc)
        j_map[j_data[:id]] = junc
        junc[:devices_in] = []; junc[:devices_out] = []
    end
    for p_data in pipe_data
        pipe = create_pipeline_optigraph(p_data); push!(network[:pipelines],pipe)
        add_subgraph!(network,pipe);
        pipe[:junc_from] = j_map[p_data[:junc_from]]
        pipe[:junc_to] = j_map[p_data[:junc_to]]
        push!(pipe[:junc_from][:devices_out],pipe); push!(pipe[:junc_to][:devices_in],
            pipe)
    end
    for c_data in comp_data
        comp = create_compressor_optigraph(c_data)
        add_subgraph!(gas_network,comp); comp[:data] = c_data
        comp[:junc_from] = j_map[c_data[:junc_from]]
        comp[:junc_to] = j_map[c_data[:junc_to]]
        push!(comp[:junc_from][:devices_out],comp); push!(comp[:junc_to][:devices_in],
            comp)
    end

    #Link pipelines in gas network
    for pipe in network[:pipelines]
        junc_from,junc_to = [pipe[:junc_from],pipe[:junc_to]]
        @linkconstraint(network,[t = 1:nt],pipe[:pin][t] == junc_from[:pressure][t])
        @linkconstraint(gas_network,[t = 1:nt],pipe[:pout][t] == junc_to[:pressure][t])
    end

    #Link compressors in gas network
    for comp in network[:compressors]
        junc_from,junc_to = [comp[:junc_from].comp[:junc_to]]
        @linkconstraint(network,[t = 1:nt],comp[:pin][t] == junc_from[:pressure][t])
        @linkconstraint(network,[t = 1:nt],comp[:pout][t] == junc_to[:pressure][t])
    end

    #Link junctions in gas network
    for junc in network[:junctions]
        devices_in = junc[:devices_in]; devices_out = junc[:devices_out]

        flow_in = [sum(device[:fout][t] for device in devices_in) for t = 1:nt]
        flow_out = [sum(device[:fin][t] for device in devices_out) for t = 1:nt]

        total_supplied = [junction[:total_supplied][t] for t = 1:nt]
        total_delivered = [junction[:total_delivered][t] for t = 1:nt]

        @linkconstraint(gas_network,[t = 1:nt], flow_in[t] - flow_out[t] +
        total_supplied[t] - total_delivered[t] == 0)
    end
    return gas_network
end
```

BIBLIOGRAPHY

[1] Syngeri gas. URL https://www.dnvgl.com/services/hydraulic-modelling-and-simulation-software-synergi-gas-3894.

[2] *Discrete Event Simulation*, pages 519–554. Springer US, Boston, MA, 2006. ISBN 978-0-387-30260-7. doi: 10.1007/0-387-30260-3_11. URL https://doi.org/10.1007/0-387-30260-3_11.

[3] January 2014 FERC Data Request. Technical report, ISO New England Inc, 01 2014.

[4] Ashish Agarwal and Ignacio E Grossmann. Linear coupled component automata for milp modeling of hybrid systems. *Computers & Chemical Engineering*, 33(1):162–175, 2009.

[5] K. Ahmed, J. Liu, A. Badawy, and S. Eidenbenz. A brief history of hpc simulation and future challenges. In *2017 Winter Simulation Conference (WSC)*, pages 419–430, 2017.

[6] Susanne Albers. Energy-efficient algorithms. *Commun. ACM*, 53(5):86âĂŞ96, May 2010. ISSN 0001-0782. doi: 10.1145/1735223.1735245. URL https://doi.org/10.1145/1735223.1735245.

[7] Andrew Allman, Wentao Tang, and Prodromos Daoutidis. Towards a Generic Algorithm for Identifying High-Quality Decompositions of Optimization Problems. In *Computer Aided Chemical Engineering*. 2018. doi: 10.1016/B978-0-444-64241-7.50152-X.

[8] Andrew Allman, Wentao Tang, and Prodromos Daoutidis. Decode: a community-based algorithm for generating high-quality decompositions of optimization problems. *Optimization and Engineering*, 06 2019. doi: 10.1007/s11081-019-09450-5.

[9] Joel A E Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, In Press, 2018.

[10] Sogol Babaeinejadsarookolaee, Adam Birchfield, Richard D. Christie, Carleton Coffrin, Christopher DeMarco, Ruisheng Diao, Michael Ferris, Stephane Fliscounakis, Scott Greene, Renke Huang, Cedric Josz, Roman Korab, Bernard Lesieutre, Jean Maeght, Daniel K. Molzahn, Thomas J. Overbye, Patrick Panciatici, Byungkwon Park, Jonathan Snodgrass, and Ray Zimmerman. The power grid library for benchmarking ac optimal power flow algorithms, 2019.

[11] Olivier Beaumont, Lionel Eyraud-Dubois, and Yihong Gao. Influence of Tasks Duration Variability on Task-Based Runtime Schedulers, February 2018. URL https://hal.inria.fr/hal-01716489. Research report on the Influence of Tasks Duration Variability on Task-Based Runtime Schedulers.

[12] Martin Bergner, Alberto Caprara, Alberto Ceselli, Fabio Furini, Marco E. Lübbecke, Enrico Malaguti, and Emiliano Traversi. Generic dantzig-wolfe reformulation of mixed integer programs. 2011.

[13] Martin Bergner, Alberto Caprara, Alberto Ceselli, Fabio Furini, Marco E Lübbecke, Enrico Malaguti, and Emiliano Traversi. Automatic dantzig–wolfe reformulation of mixed integer programs. *Mathematical Programming*, 149(1):391–424, feb 2015. ISSN 1436-4646. doi: 10.1007/s10107-014-0761-5. URL https://doi.org/10.1007/s10107-014-0761-5.

[14] Dimitri P Bertsekas and John N Tsitsiklis. *Parallel and distributed computation: numerical methods*. 2003.

[15] Martin Biel and Mikael Johansson. Efficient stochastic programming in julia, 2019.

[16] John R. Birge and Franois Louveaux. *Introduction to Stochastic Programming*. Springer Publishing Company, Incorporated, 2nd edition, 2011. ISBN 1461402360.

[17] J. Bisschop. AIMMS - Optimization Modeling, 2006.

[18] Benedikt Bollig, Marie Fortin, and Paul Gastin. Communicating Finite-State Machines and Two-Variable Logic. *arXiv preprint arXiv:1709.09991*, 2017. URL http://arxiv.org/abs/1709.09991.

[19] LÃl'on Bottou. Large-scale machine learning with stochastic gradient descent. In *in COMPSTAT*, 2010.

[20] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.*, 3(1):1âĂŞ122, January 2011. ISSN 1935-8237. doi: 10.1561/2200000016. URL https://doi.org/10.1561/2200000016.

[21] Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM*, 30(2):323–342, 1983. ISSN 00045411. doi: 10.1145/322374.322380. URL http://portal.acm.org/citation.cfm?doid=322374.322380.

[22] M S Branicky and S E Mattsson. Simulation of Hybrid Systems. *Lecture Notes in Computer Science*, 1273:31–56, 1997.

[23] Braulio Brunaud and Ignacio E Grossmann. Perspectives in Multilevel Decision-making in the Process Industry. 4(3):1–34, 2017. ISSN 2095-7513. doi: 10.15302/J-FEM-2017049.

[24] Yankai Cao and Victor M Zavala. A scalable global optimization algorithm for stochastic nonlinear programs. doi: 10.1007/s10898-019-00769-y.

[25] Yankai Cao, Carl D. Laird, and Victor M. Zavala. Clustering-based preconditioning for stochastic programs. *Computational Optimization and Applications*, 64:379–406, 2016.

[26] Richard G Carter. Pipeline optimization: dynamic programming after 30 years. In *PSIG Annual Meeting*. Pipeline Simulation Interest Group, 1998.

[27] H. Casanova. Simgrid: a toolkit for the simulation of application scheduling. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 430–437, 2001.

[28] Ümit Çatalyürek and Cevdet Aykanat. *PaToH (Partitioning Tool for Hypergraphs)*, pages 1479–1487. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4. doi: 10.1007/978-0-387-09766-4_93. URL https://doi.org/10.1007/978-0-387-09766-4{_}93.

[29] Nai Yuan Chiang and Victor M. Zavala. Large-scale optimal control of interconnected natural gas and electrical transmission systems. *Applied Energy*, 168: 226–235, 2016. ISSN 03062619. doi: 10.1016/j.apenergy.2016.01.017. URL http://linkinghub.elsevier.com/retrieve/pii/S0306261916000362.

[30] Nicholson Collier and Michael North. Parallel agent-based simulation with Repast for High Performance Computing. *Simulation*, 89(10):1215–1235, 2013. ISSN 17413133. doi: 10.1177/0037549712462620.

[31] Marco Colombo, Andreas Grothey, Jonathan Hogg, Kristian Woodsend, and Jacek Gondzio. A structure-conveying modelling language for mathematical and stochastic programming. *Mathematical Programming Computation*, 2009. ISSN 18672949. doi: 10.1007/s12532-009-0008-2.

[32] Antonio J. Conejo, Enrique Castillo, Roberto Mínguez, and Raquel García-Bertrand. *Decomposition techniques in mathematical programming: Engineering and science applications*. 2006. ISBN 3540276858. doi: 10.1007/3-540-27686-6.

[33] GAMS Development Corporation. General Algebraic Modeling System (GAMS) Release 24.2.1. Washington, DC, USA, 2013. URL http://www.gams.com/.

[34] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, MarcâĂŹAurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPSâĂŹ12, page 1223âĂŞ1231, Red Hook, NY, USA, 2012. Curran Associates Inc.

[35] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Umit V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *20th International Parallel and Distributed Processing Symposium, IPDPS 2006*, 2006. ISBN 1424400546. doi: 10.1109/IPDPS.2006.1639359.

[36] Alexander W Dowling and Lorenz T Biegler. A framework for efficient large scale equation-oriented flowsheet optimization. *Computers & Chemical Engineering*, 72: 3–20, 2015.

[37] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017. doi: 10.1137/15M1020575.

[38] Sanghamitra Dutta, Gauri Joshi, Soumyadip Ghosh, Parijat Dube, and Priya Nagpurkar. Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd. In *AISTATS*, 2018.

[39] E.W. Endsley and M. Tilbury. Modular finite state machines for logic control. *IFAC Proceedings Volumes*, 37(18):393–398, 2004. ISSN 14746670. doi: 10.1016/S1474-6670(17)30778-4. URL http://linkinghub.elsevier.com/retrieve/pii/S1474667017307784.

[40] Process Systems Enterprise. gPROMS, 1997-2018. URL www.psenterprise.com/gproms.

[41] Francesco Farina, Andrea Camisa, Andrea Testa, Ivano Notarnicola, and Giuseppe Notarstefano. Disropt : a python framework. (638992):1–14.

[42] Michael C Ferris and Jeffrey D Horn. Partitioning mathematical programs for parallel solution. *Mathematical Programming*, 80(1):35–61, jan 1998. ISSN 1436-4646. doi: 10.1007/BF01582130. URL https://doi.org/10.1007/BF01582130.

[43] Marshall L. Fisher. APPLICATIONS ORIENTED GUIDE TO LAGRANGIAN RELAXATION. *Interfaces*, 1985. ISSN 00922102. doi: 10.1287/inte.15.2.10.

[44] George S. Fishman. *Principles of Discrete Event Simulation*. John Wiley & Sons, Inc., New York, NY, USA, 1978. ISBN 0471043958.

[45] S. Folga. Natural Gas Pipeline Technology Overview. Technical Report 4, 2008.

[46] R. Fourer, D. M. Gay, and B. Kernighan. *AMPL: A Mathematical Programming Language*, page 150âĂŞ151. Springer-Verlag, Berlin, Heidelberg, 1989. ISBN 0387508422.

[47] Robert Fourer, David M Gay, Murray Hill, Brian W Kernighan, and T Bell Laboratories. AMPL : A Mathematical Programming Language. 1990.

[48] Peter Fritzson and Peter Bunus. Modelica – a general object-oriented language for continuous and discrete-event system modeling. In *IN PROCEEDINGS OF THE 35TH ANNUAL SIMULATION SYMPOSIUM*, pages 14–18, 2002.

[49] Andreas Frommer and Daniel B. Szyld. An algebraic convergence theory for restricted additive schwarz methods using weighted max norms. *SIAM Journal on Numerical Analysis*, 39(2):463–479, 2002. ISSN 00361429. URL http://www.jstor.org/stable/3062006.

[50] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. Co-simulation: a Survey. *ACM Computing Surveys*, 51(3):Article 49, 2018. ISSN 03600300. doi: 10.1145/3179993. URL http://cordis.europa.eu/project/rcn/194142{%}0Ahttps://doi.org/10.1145/3179993.

[51] Jacek Gondzio and Andreas Grothey. Parallel Interior Point Solver for Structured Quadratic Programs: Application to Financial Planning Problems. *J. Annals of Operations Research*, 152(1):319–339, 2006. doi: 10.1007/s10479-006-0139-z.

[52] Jacek Gondzio and Robert Sarkissian. Parallel interior-point solver for structured linear programs. *Mathematical Programming*, 2003. ISSN 00255610. doi: 10.1007/s10107-003-0379-5.

[53] Ignacio Grossman. *Global Optimization*. Springer, 2013.

[54] Ignacio E. Grossmann. Advances in mathematical programming models for enterprise-wide optimization. *Computers and Chemical Engineering*, 47:2–18, 2012. ISSN 00981354. doi: 10.1016/j.compchemeng.2012.06.038. URL http://dx.doi.org/10.1016/j.compchemeng.2012.06.038.

[55] Andreas Grothey and Feng Qiang. PSMG-A Parallel Structured Model Generator for Mathematical Programming. Workingpaper, Optimization Online, 2014.

[56] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, 1997. ISSN 10459219. doi: 10.1109/71.598277.

[57] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2020. URL http://www.gurobi.com.

[58] David Hallac, Christopher Wong, Steven Diamond, Rok Sosic, Stephen Boyd, and Jure Leskovec. SnapVX: A Network-Based Convex Optimization Solver. *Journal of Machine Learning Research*, 0:1–5, 2017. ISSN 15337928. URL http://arxiv.org/abs/1509.06397.

[59] William E. Hart, Carl D. Laird, Jean-Paul Watson, David L. Woodruff, Gabriel A. Hackebeil, Bethany L. Nicholson, and John D. Siirola. *Pyomo–optimization modeling in python*, volume 67. Springer Science & Business Media, second edition, 2017.

[60] Seongmin Heo, Srinivas Rangarajan, Prodromos Daoutidis, and Sujit S. Jogwar. Graph reduction of complex energy-integrated networks: Process systems applications. *AIChE Journal*, 2014. ISSN 00011541. doi: 10.1002/aic.14341.

[61] M Herty, J. Mohring, and V. Sachers. A new model for gas flow in pipe networks. *Mathematical Methods in the Applied Sciences*, 33(7):845–855, 2010. ISSN 01704214. doi: 10.1002/mma.1197. URL http://doi.wiley.com/10.1002/mma.1197.

[62] Michael Herty. Multi âĂŞ scale modeling and nodal control for gas transportation networks. (Cdc):4585–4590, 2015. ISSN 07431546. doi: 10.1109/CDC.2015.7402935.

[63] Jens Hübner, Martin Schmidt, and Marc C Steinbach. Optimization techniques for tree-structured nonlinear problems. *Computational Management Science*, 2020. ISSN 1619-6988. doi: 10.1007/s10287-020-00362-9. URL https://doi.org/10.1007/s10287-020-00362-9.

[64] Joey Huchette, Miles Lubin, and Cosmin Petra. Parallel algebraic modeling for stochastic optimization. In *Proceedings of HPTCDL 2014: 1st Workshop for High Performance Technical Computing in Dynamic Languages - Held in Conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014. ISBN 9781479970209. doi: 10.1109/HPTCDL.2014.6.

[65] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. Fashionable modelling with flux. *CoRR*, abs/1811.01457, 2018. URL http://arxiv.org/abs/1811.01457.

[66] Jordan Jalving and Victor M Zavala. An optimization-based state estimation framework for large-scale natural gas networks. *Industrial & Engineering Chemistry Research*, 57(17):5966–5979, 2018. ISSN 0888-5885. doi: 10.1021/acs.iecr.7b04124.

[67] Jordan Jalving, Shrirang Abhyankar, Kibaek Kim, Mark Hereld, and Victor M. Zavala. A graph-based computational framework for simulation and optimisa-

tion of coupled infrastructure networks. *IET Generation, Transmission & Distribution*, pages 1–14, 2017. ISSN 1751-8687. doi: 10.1049/iet-gtd.2016.1582. URL http://digital-library.theiet.org/content/journals/10.1049/iet-gtd.2016.1582.

[68] Jordan Jalving, Yankai Cao, and Victor M Zavala. Graph-based modeling and simulation of complex systems. *Computers & Chemical Engineering*, 125:134–154, 2019. doi: https://doi.org/10.1016/j.compchemeng.2019.03.009.

[69] Jordan Jalving, Sungho Shin, and Victor M. Zavala. A graph-based modeling abstraction for optimization: Concepts and implementation in plasmo.jl, 2020.

[70] Wenkai Jiang, Jianzhong Qi, Jeffrey Xu Yu, Jin Huang, and Rui Zhang. HyperX: A Scalable Hypergraph Framework. *IEEE Transactions on Knowledge and Data Engineering*, 2018. ISSN 10414347. doi: 10.1109/TKDE.2018.2848257.

[71] Sujit S. Jogwar, Srinivas Rangarajan, and Prodromos Daoutidis. Reduction of complex energy-integrated process networks using graph theory. *Computers and Chemical Engineering*, 2015. ISSN 00981354. doi: 10.1016/j.compchemeng.2015.04.025.

[72] Jia Kang, Yankai Cao, Daniel P. Word, and C. D. Laird. An interior-point method for efficient solution of block-structured NLP problems using an implicit Schur-complement decomposition. *Computers and Chemical Engineering*, 2014. ISSN 00981354. doi: 10.1016/j.compchemeng.2014.09.013.

[73] Jia Kang, Naiyuan Chiang, Carl D Laird, and Victor M Zavala. Nonlinear programming strategies on high-performance computers. In *Decision and Control (CDC), 2015 IEEE 54th Annual Conference on*, pages 4612–4620. IEEE, 2015.

[74] Juraj Kardos, Drosos Kourounis, and Olaf Schenk. Structure-Exploiting Interior Point Methods. *ArXiv*, 2019.

[75] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392,

1998. ISSN 1064-8275. doi: 10.1137/S1064827595287997. URL http://epubs.siam.org/doi/10.1137/S1064827595287997.

[76] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, pages 343–348, New York, NY, USA, 1999. ACM. ISBN 1-58113-109-7. doi: 10.1145/309847.309954. URL http://doi.acm.org/10.1145/309847.309954.

[77] Kibaek Kim and Victor M Zavala. Algorithmic innovations and software for the dual decomposition method applied to stochastic mixed-integer programs. 2016.

[78] Kibaek Kim, Cosmin G Petra, and Victor M Zavala. An asynchronous bundle-trust-region method for dual decomposition of stochastic mixed-integer programming. *SIAM Journal on Optimization*, 29(1):318–342, 2019.

[79] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[80] Edward A Lee. Cyber physical systems: Design challenges. In *11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, 2008.

[81] Jay Lee, Behrad Bagheri, and Hung-An Kao. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing Letters*, 3:18–23, 2015.

[82] Jeff Linderoth and Stephen Wright. Decomposition algorithms for stochastic programming on a computational grid. *Computational Optimization and Applications*, 2003. ISSN 09266003. doi: 10.1023/A:1021858008222.

[83] Yuri K. Lopes, Stefan M. Trenkwalder, André B. Leal, Tony J. Dodd, and Roderich Groß. Supervisory control theory applied to swarm robotics. *Swarm Intelligence*, 10 (1):65–97, 2016. ISSN 19353820. doi: 10.1007/s11721-016-0119-0.

[84] Miles Lubin, Cosmin G. Petra, and Mihai Anitescu. The parallel solution of dense saddle-point linear systems arising in stochastic programming. *Optimization Methods and Software*, 2012. ISSN 10556788. doi: 10.1080/10556788.2011.602976.

[85] Sean Luke, Claudio Cioffi-revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. MASON: A Multi-Agent Simulation Environment. pages 1–18. doi: 10.1177/0037549705058073.

[86] C M Macal and M J North. Tutorial on agent-based modeling and simulation. *Proceedings of the 2005 Winter Simulation Conference, Vols 1-4*, pages 2–15, 2005. doi: 10.1109/wsc.2005.1574234.

[87] Andrew Makhorin. GNU Linear Programming Kit Version 4.32, 2000–2012. URL http://www.gnu.org/software/glpk/glpk.html.

[88] Christos T. Maravelias. General framework and modeling approach classification for chemical production scheduling. *AIChE Journal*, 2012. ISSN 00011541. doi: 10.1002/aic.13801.

[89] MATLAB Simulink Toolbox. Matlab simulink toolbox, 2018.

[90] Sven Erik Mattsson, Hilding Elmqvist, and Martin Otter. Physical system modeling with modelica. *Control Engineering Practice*, 6(4):501–510, 1998.

[91] Christian Mayer, Ruben Mayer, Sukanya Bhowmik, Lukas Epple, and Kurt Rothermel. Hype: Massive hypergraph partitioning with neighborhood expansion. *2018 IEEE International Conference on Big Data (Big Data)*, pages 458–467, 2018.

[92] Nelson Minar, Roger Burkhart, Chris Langton, Manor Askenazi, et al. The swarm simulation system: A toolkit for building multi-agent simulations. 1996.

[93] Ali Mohammed, Ahmed Eleliemy, Florina M. Ciorba, Franziska Kasielke, and Ioana Banicescu. An approach for realistically simulating the performance of scientific applications on high performance computing systems. *ArXiv*, abs/1910.06844, 2019.

[94] Manjiri Moharir, Lixia Kang, Prodromos Daoutidis, and Ali Almansoori. Graph representation and decomposition of ODE/hyperbolic PDE systems. *Computers and Chemical Engineering*, 2017. ISSN 00981354. doi: 10.1016/j.compchemeng.2017.07.005.

[95] M. E.J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences of the United States of America*, 2006. ISSN 00278424. doi: 10.1073/pnas.0601602103.

[96] Michael J North, Nicholson T Collier, Jonathan Ozik, Eric R Tatara, Charles M Macal, Mark Bragen, and Pam Sydelko. Complex adaptive systems modeling with Repast Simphony. *Complex Adaptive Systems Modeling*, 1(1):3, 2013. ISSN 2194-3206. doi: 10.1186/2194-3206-1-3. URL http://casmodeling.springeropen.com/articles/10.1186/2194-3206-1-3.

[97] A. Osiadacz. Simulation of transient gas flows in networks. *International Journal for Numerical Methods in Fluids*, 4(1):13–24, 1984. ISSN 10970363. doi: 10.1002/fld.1650040103.

[98] Jonathan Ozik, Nick Collier, and Repast Development. Repast statecharts guide. 8 (3):1–41, 1994.

[99] Bruce Palmer, William Perkins, Yousu Chen, Shuangshuang Jin, David Callahan, Kevin Glass, Ruisheng Diao, Mark Rice, Stephen Elbert, Mallikarjuna Vallem, and Zhenyu Huang. Gridpack: A framework for developing power grid simulations on high performance computing platforms. In *Proceedings of the 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC âĂŹ14, page 68âĂŞ77, USA, 2014. IEEE Computer Society. ISBN 9781467367578.

[100] Francois Pellegrini. Distillating knowledge about scotch. In Uwe Naumann, Olaf Schenk, Horst D. Simon, and Sivan Toledo, editors, *Combinatorial Scien-*

tific Computing, number 09061 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. URL http://drops.dagstuhl.de/opus/volltexte/2009/2091.

[101] Peter C Piela. ASCEND : an object-oriented computer environment for modeling and analysis . 1990.

[102] Edgar C. Portante, James A. Kavicky, Brian A. Craig, Leah E. Talaber, and Stephen M. Folga. Modeling electric power and natural gas system interdependencies. *Journal of Infrastructure Systems*, 23(4):04017035, 2017. doi: 10.1061/(ASCE)IS.1943-555X.0000395.

[103] S. Pu, A. Olshevsky, and I. C. Paschalidis. Asymptotic network independence in distributed stochastic optimization for machine learning: Examining distributed and centralized stochastic gradient descent. *IEEE Signal Processing Magazine*, 37(3):114–122, 2020.

[104] Ragheb Rahmaniani, Teodor Gabriel Crainic, Michel Gendreau, and Walter Rei. The benders decomposition algorithm: A literature review. *European Journal of Operational Research*, 259(3):801 – 817, 2017. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2016.12.005. URL http://www.sciencedirect.com/science/article/pii/S0377221716310244.

[105] Christopher V Rao, Stephen J Wright, and James B Rawlings. Application of interior-point methods to model predictive control. *Journal of optimization theory and applications*, 99(3):723–757, 1998.

[106] James B. Rawlings and David Mayne. *Model predictive control: Theory and design*. 2 edition, 2018.

[107] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In

J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011. URL http://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf.

[108] Daniel Rehfeldt, Hannes Hobbie, David Schönheit, Ambros M. Gleixner, Thorsten Koch, and Dominik Möst. A massively parallel interior-point solver for linear energy system models with block structure. 2019.

[109] Jose S Rodriguez, Carl D Laird, and Victor M Zavala. Scalable preconditioning of block-structured linear algebra systems using ADMM. *Computers and Chemical Engineering*, 133:106478, 2020. ISSN 0098-1354. doi: 10.1016/j.compchemeng.2019.06.003. URL https://doi.org/10.1016/j.compchemeng.2019.06.003.

[110] N. V. Sahinidis and I. E. Grossmann. Convergence properties of generalized benders decomposition. *Computers and Chemical Engineering*, 1991. ISSN 00981354. doi: 10.1016/0098-1354(91)85027-R.

[111] Riccardo Scattolini. Architectures for distributed and hierarchical Model Predictive Control - A review, 2009. ISSN 09591524.

[112] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k-way hypergraph partitioning via *n*-level recursive bisection. In *18th Workshop on Algorithm Engineering and Experiments, (ALENEX 2016)*, pages 53–67, 2016.

[113] Kirk Schloegel, George Karypis, and Vipin Kumar. *Graph Partitioning for High-Performance Scientific Simulations*, page 491âĂŞ541. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 1558608710.

[114] Christian Schulz, Sebastian Korbinian Bayer, Christoph Hess, Christian Steiger, Marvin Teichmann, Jan Jacob, Fellipe Bernardes-lima, Robert Hangu, and Sergey

Hayrapetyan. Course notes: Graph partitioning and graph clustering in theory and practice, 2015.

[115] William Lewis Scott. Optimal evasive strategies for groups of interacting agents with motion constraints. *Automatica*, 94:26–34, 2018.

[116] James Fairbanks Seth Bromberger and other contributors. Juliagraphs/lightgraphs.jl: an optimized graphs package for the julia programming language, 2017. URL https://doi.org/10.5281/zenodo.889971.

[117] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In Marco Hutter and Roland Siegwart, editors, *Field and Service Robotics*, pages 621–635, Cham, 2018. Springer International Publishing. ISBN 978-3-319-67361-5.

[118] M. Shahidehpour, Yong Fu, and T. Wiedman. Impact of natural gas infrastructure on electric power systems. *Proceedings of the IEEE*, 93(5):1042–1056, 2005.

[119] S. Shin, T. Faulwasser, M. Zanon, and V. M. Zavala. A parallel decomposition scheme for solving long-horizon optimal control problems. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 5264–5271, 2019.

[120] S. Shin, V. M. Zavala, and M. Anitescu. Decentralized schemes with overlap for solving graph-structured optimization problems. *IEEE Transactions on Control of Network Systems*, 2020.

[121] Sungho Shin and Victor M Zavala. Multi-grid schemes for multi-scale coordination of energy systems. *Energy Markets and Responsive Grids*, 2018.

[122] Sungho Shin and Victor M Zavala. Multi-grid schemes for multi-scale coordination of energy systems. In *Energy Markets and Responsive Grids*, pages 195–222. Springer, 2018.

[123] Sungho Shin, Mihai Anitescu, and Victor M. Zavala. Overlapping schwarz decomposition for constrained quadratic programs, 2020.

[124] Marc C. Steinbach. Tree-sparse convex programs. *Mathematical Methods of Operations Research*, 2003. ISSN 14322994. doi: 10.1007/s001860200227.

[125] Brett T. Stewart, James B. Rawlings, and Stephen J. Wright. Hierarchical cooperative distributed model predictive control. In *Proceedings of the 2010 American Control Conference, ACC 2010*, 2010. ISBN 9781424474264. doi: 10.1109/acc.2010.5530634.

[126] J Sun and Tesfasion L. Dc optimal power flow formulation andsolution using quadprogj, 2010.

[127] Wentao Tang and Prodromos Daoutidis. Network decomposition for distributed control through community detection in inputâĂŞoutput bipartite graphs. *Journal of Process Control*, 2018. ISSN 09591524. doi: 10.1016/j.jprocont.2018.01.009.

[128] Wentao Tang, Andrew Allman, Davood Babaei Pourkargar, and Prodromos Daoutidis. Optimal decomposition for distributed optimization in nonlinear model predictive control through community detection. *Computers & Chemical Engineering*, 111:43–54, 2017. ISSN 00981354. doi: 10.1016/j.compchemeng.2017.12.010. URL http://linkinghub.elsevier.com/retrieve/pii/S0098135417304416.

[129] Ophelia S Venturelli, Alex V Carr, Garth Fisher, Ryan H Hsu, Rebecca Lau, Benjamin P Bowen, Susan Hromada, Trent Northen, and Adam P Arkin. Deciphering microbial interactions in synthetic human gut microbiome communities. *Molecular Systems Biology*, 14(6):e8157, 2018. doi: 10.15252/msb.20178157. URL https://www.embopress.org/doi/abs/10.15252/msb.20178157.

[130] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. 106(1): 25–57, 2006. ISSN 00255610. doi: 10.1007/s10107-004-0559-y.

[131] Jiadong Wang and Ted Ralphs. Computational Experience with Hypergraph-Based Methods for Automatic Decomposition in Discrete Optimization. In Carla Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 394–402, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38171-3.

[132] Jean Paul Watson, David L. Woodruff, and William E. Hart. PySP: Modeling and solving stochastic programs in Python. *Mathematical Programming Computation*, 2012. ISSN 18672949. doi: 10.1007/s12532-012-0036-1.

[133] Arthur W Westerberg and Peter C Piela. Equational-based process modeling. *ASCEND project*, xx:1–76, 1994.

[134] Jeremiah J. Wilke, Joseph P. Kenny, Samuel Knight, and Sebastien Rumley. Compiler-assisted source-to-source skeletonization of application models for system simulation. In Rio Yokota, Michèle Weiland, David Keyes, and Carsten Trinitis, editors, *High Performance Computing*, pages 123–143. Springer International Publishing, 2018.

[135] R. Xin, S. Kar, and U. A. Khan. Decentralized stochastic optimization and machine learning: A unified variance-reduction framework for robust performance and fast convergence. *IEEE Signal Processing Magazine*, 37(3):102–113, 2020.

[136] Victor M. Zavala. New Architectures for Hierarchical Predictive Control. *IFAC-PapersOnLine*, 49(7):43–48, 2016. ISSN 24058963. doi: 10.1016/j.ifacol.2016.07.214. URL http://dx.doi.org/10.1016/j.ifacol.2016.07.214.

[137] Victor M Zavala and Lorenz T Biegler. Optimization-based strategies for the operation of low-density polyethylene tubular reactors: Moving horizon estimation. *Computers & Chemical Engineering*, 33(1):379–390, 2009.

[138] Stavros A Zenios. A distributed algorithm for convex network optimization problems. *Parallel Computing*, 6:45–56, 1988.

[139] Stavros A. Zenios and Mustafa Pinar. Parallel Block-Partitioning of Truncated Newton for Nonlinear Network Optimization. *SIAM Journal on Scientific and Statistical Computing*, 1992. ISSN 0196-5204. doi: 10.1137/0913068.