A Graph-Based Computational Framework for Simulation and Optimization of Coupled Infrastructure Networks

Jordan Jalving[¶], Shrirang Abhyankar[*], Kibaek Kim[‡], Mark Hereld[‡], Victor M. Zavala[¶]

[¶]Department of Chemical and Biological Engineering, University of Wisconsin-Madison, 1415 Engineering Dr. Madison, WI 53706, USA
[*]Energy Sciences Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL, USA
[‡]Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL, USA

Abstract: We present a computational framework that facilitates the construction, instantiation, and analysis of large-scale optimization and simulation applications of coupled energy networks. The framework integrates the optimization modeling package `PLASMO` and the simulation package `DMNetwork` (built around `PETSc`). These tools use a common graph-based abstraction that enables us to achieve compatibility between data structures and to build applications that use network models of different physical fidelity. We also describe how to embed these tools within complex computational workflows using `SWIFT`, which is a tool that facilitates parallel execution of multiple simulation runs and management of input and output data. We discuss how to use these capabilities to target coupled natural gas and electricity systems.

Keywords: large-scale; optimization; graph; simulation; parallel; instantiation; workflows

1. Motivation

There is an increasing need to create optimization and simulation models to design, operate, and analyze interactions between infrastructure networks. This is technically challenging, as different infrastructures present different physical phenomena and constraints at different scales. The nature of the equations describing such phenomena and numerical techniques to address them can be drastically different and complicates the creation of computational tools capable of handling coupled systems. For instance, optimization of natural gas networks often needs to capture slow transient effects using partial differential equations [4, 17, 6]. Such dynamics result, from instance, from sudden withdrawals of natural gas that propagate throughout the network and affect gas delivery to power plants and grid operations [11].

In this work, we present a computational framework that integrates powerful optimiza-

tion, simulation, and workflow management tools: PLASMO , DMNetwork, and SWIFT. PLASMO is a graph-based algebraic modeling language that facilitates the construction of structured optimization models and provides interfaces to large-scale parallel optimization solvers DSP [9] and PIPS-NLP [5], as well as off-the-shelf solvers (e.g., IPOPT [12]). DMNetwork is a modeling package that facilitates the implementation of network models and uses PETSc libraries to perform high-resolution dynamic simulations on parallel computers [1]. SWIFT is a scripting language that facilitates deployment and management of computational tasks on high-performance computing environments [16]. By interfacing these capabilities, our framework allows us to construct sophisticated applications for coupled infrastructure networks and to leverage high-performance computing architectures. We demonstrate the capabilities by using cases studies arising in the optimization and simulation of coupled natural gas and electric networks.

We highlight that, seamless integration of different optimization and simulation tools is achieved by using a common *graph-based abstraction* in which infrastructure components are represented as nodes and edges. Such an abstraction facilitates the construction of complex hierarchical networks models (networks of networks) coupled at different levels, facilitates collaboration by enabling model and data sharing and re-use, facilitates input-output data management and transfer, and facilitates the construction of workflows to explore algorithmic performance and to perform model validation and verification.

2. PLASMO

PLASMO (Platform for Scalable Modeling and Optimization) is a Julia-based modeling framework that facilitates the construction and analysis of structured optimization models. To do this, it leverages a hierarchical graph abstraction wherein nodes and edges can be associated with mathematical models and connectivity constraints (physical or logical). Given a graph structure with models and connections, PLASMO can produce either a pure (flattened) optimization model to be solved using any off-the-shelf optimization solvers such as IPOPT [12], or it can communicate structures to parallel solvers such as DSP [9] or PIPS [5]. By using a graph abstraction, a model can be created in collaborative form where different modelers develop different components. The hierarchical graph abstraction also induces hierarchical data structures, which makes input and output data easier to parse and analyze.

2.1. Graph Abstraction

PLASMO incorporates a graph-based abstraction for model representation and interaction that facilitates coupling of submodels without requiring underlying changes to those models. To do this, the PLASMO graph associates individual models with nodes and edges and uses the graph topology to create coupling constraints among edges and neighbors to enforce physical or logical connections. Figure 1 illustrates the basic graph concept for a simple power grid system. Within the figure, we present a graph topology of four nodes with as-

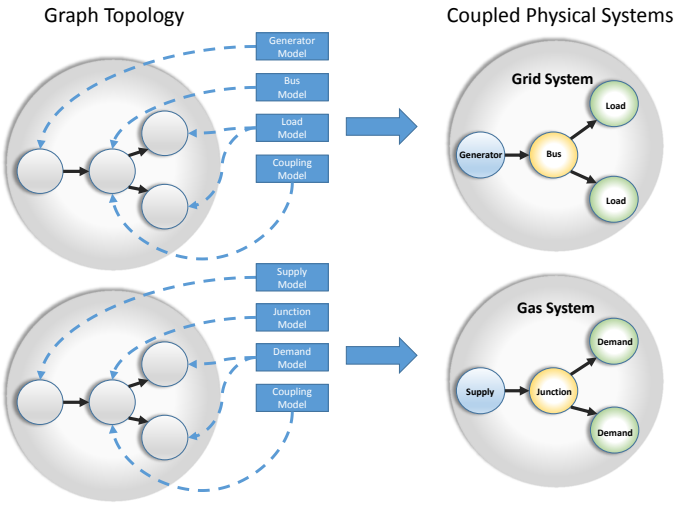

Fig. 1: Adding model components to a graph topology

sociated physical models for a power grid bus, generator, and load. The bus is coupled to its neighbors through a coupling statement (explained later), which aggregates the load and generator connected to the bus. In more practical applications, it is useful to organize a model into a hierarchy of such nodes themselves to create large hierarchical network systems. This is straightforward given the `PLASMO` hierarchical graph object. Sub-systems can be completely specified as illustrated in Figure 1, and then encapsulated into a node within a larger graph. Figure 2 illustrates a hierarchical graph containing two separate grid systems (graphs) connected by a transmission line in the context of a network (parent graph). Graph hierarchy is accomplished while retaining the overall connectivity structure in subnodes. This means that an equivalent graph can be constructed without the use of hierarchies, but it would not benefit from the coupling context across layers. The hierarchical structure automatically differentiates between edges and nodes connected within a local graph and those connected between subgraphs. Referencing Figure 2 again, the total degree of bus one is equal to four, but at the regional level its total degree is only one (the edge with the power line model). This abstraction makes it straightforward to develop models for smaller systems, and then connect nodes containing such smaller systems to larger systems at the topology level of interest. Such a setting can be used to create networks that span local, regional, and national scales. In our simple example, for instance, bus one is coupled to a generator and two loads where the coupling defines total bus generation and load. It is then coupled at the regional level to bus two, which defines the power balance across the bus. For massive problems, the graph abstraction can be used directly to perform partitioning and aggregation tasks that can in turn be used to develop sophisticated decentralized solution schemes (e.g., neighbor-to-neighbor, price coordination). Partitioning, in particular, induces problem structure that can be exploited by parallel solvers such as `DSP`[9] and `PIPS-NLP`[5].

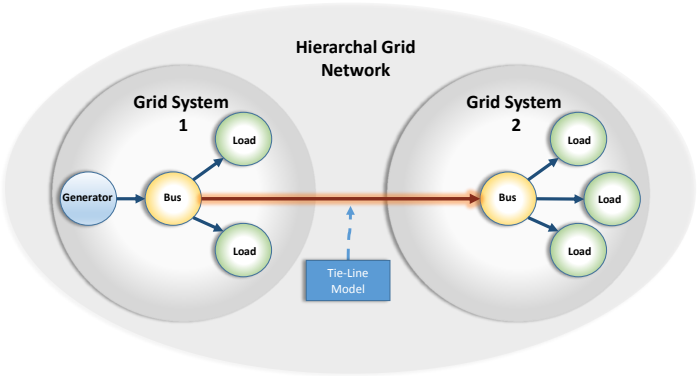

Fig. 2: Connecting components across subgraphs

2.2. Model Construction

PLASMO models are constructed in the Julia [3] language building upon the JuMP (Julia for Mathematical Programming)[7] framework. Julia is a high-level, high-performance programming language for technical computing with speeds comparable to pure C/C++. JuMP is an optimization package written in Julia that contains an abstract model object for modeling general mathematical programs with interfaces to off-the-shelf optimization solvers such as Ipopt and Gurobi. PLASMO builds on the core JuMP model object and associated processing tools (e.g., automatic differentiation) for the construction of high-level and structured models. This also allows PLASMO to manage input and solution data for individual modeling objects.

PLASMO Function	Description
graph = Graph()	Creates a PLASMO graph
node = Node(g::Graph) or node = Node(g::Graph,m::Model)	Adds a node to Graph g. Sets the node model if provided.
edge = Edge(g::Graph) or edge = Edge(g::Graph,m::Model)	Adds an edge to Graph g. Sets the edge model if provided.
node = addgraph(g::Graph,sg::Graph)	Create a new node in graph containing a subgraph
edges_in(g::Graph,node::Node)	Retrieves the edges into node that are at the level of graph
edges_out(g::Graph,node::Node)	Retrieves the edges out of node that are at the level of graph
neighbors_in(g::Graph,node::Node)	Retrieves the neighbors with edges into node that are at the level of graph
neighbors_out(g::Graph,node::Node)	Retrieves the neighbors with edges out of node that are at the level of graph
setmodel(ne::NodeOrEdge)	Set the model for a node or edge
setcouplingfunction(ne::NodeOrEdge)	Set the coupling function for a node or edge
@couple()	macro to perform quick node and edge couplings using expressions
model = generatemodel(g::Graph)	Create a flat optimization model from the graph
storecurrentsolution(g::Graph)	Hold the current solution from the most recent model
setcurrentsolution(g::Graph)	Initialize node and edge models with current solution

Table 1 Core PLASMO functions

To demonstrate PLASMO model construction features, we develop component models for both power grid and natural gas systems, and then combine them using functionality from

Table 1 to create hierarchical (networks of networks) systems representative of the physical infrastructure.

2.2.1. Example: Power Grid Model: For the power grid model, we focus on the well-known economic dispatch problem. This problem is solved by ISOs to balance electric supply and demand and to price electricity in intraday operations. Consider the following basic continuous-time economic dispatch problem:

$$\min \ \varphi^{grid} := \int_0^T \sum_{g \in \mathcal{G}} \alpha_i^g g_i(\tau) d\tau \tag{2.1a}$$

$$\text{s.t.} \ \frac{dg_i(\tau)}{d\tau} = r_i(\tau), \ i \in \mathcal{G} \tag{2.1b}$$

$$\sum_{i \in \mathcal{G}_n} g_i(\tau) - \sum_{j \in \mathcal{D}_n} d_j(\tau) + \sum_{\ell \in \mathcal{L}_n^{rec}} f_\ell(\tau) - \sum_{\ell \in \mathcal{L}_n^{snd}} f_\ell(\tau) = 0, \ n \in \mathcal{N} \tag{2.1c}$$

Here, $\tau \in [0, T]$ denotes the time dimension where T is the final time. We define the sets of electricity generators as $\mathcal{G}$, the set of electrical loads as $\mathcal{D}$, the set of network nodes as $\mathcal{N}$, and the set of transmission lines (links) as $\mathcal{L}$. For each link $\ell \in \mathcal{T}$ we denote $snd(t) \in \mathcal{N}$ as the sending node and $rec(t) \in \mathcal{N}$ as the receiving node. Using such notation we can construct the sets $\mathcal{L}_n^{rec}$ and $\mathcal{L}_n^{snd}$, which denote the set of flows entering and leaving node $n \in \mathcal{N}$. To simplify the discussion, we do not show capacity, ramping, or DC flow constraints.

In the context of the `PLASMO` graph, this formulation can be represented with four types of model *components*: generators, loads, buses (nodes), and transmission lines (edges) between buses. Each component has its own attributes (variables, objectives, and constraints). We can abstract the topology into a hierarchical graph as shown in Figure 2. Each bus is a central node connected to generation and load nodes within its own low-level graph and connected to other buses through tie lines within a higher level graph. Snippet 1 demonstrates the individual `Julia` functions for the different components of the economic dispatch problem. For simple cases, models can be coupled without directly instantiating node and edge data structures using the `@couple` macro. Snippet 1 shows one way a bus could be coupled to its generators and loads.

Snippet 1: Building a Grid Model using PLASMO Graph and Modeling Components

```
# Build grid
time = 1:N # define a time discretization to model dynamics
grid = Graph()
bus = busmodel()
gen = genmodel()
load1 = loadmodel()
load2 = loadmodel()
@couple(grid, [t=time], bus.busgen[t] == gen.Pgen[t])
@couple(grid, [t=time], bus.busgen[t] == load1.Pload[t] + load2.Pload[t])

# A bus with generation and load
function busmodel()
    m = Model()
    @variable(m,busgen[time]>= 0)  #bus generation
    @variable(m,busload[time]>= 0) #bus load
    return m
end

# A generator model
function genmodel()
    m = Model()
    @variable(m,0 <= Pgen[time] <= capacity)  #power generation
    @variable(m,genCost)
    @constraint(m,genCost== sum{cost*Pgen[t],t = time}) #generation cost
    @constraint(m,ramp_ub[t = time[1:end-1]],Pgen[t+1]-Pgen[t] <=  +max_ramp)  #ramp limit
    @constraint(m,ramp_lb[t = time[1:end-1]],Pgen[t]-Pgen[t+1] >= -max_ramp)   #ramp limit
    @objective(m,Min, genCost)
    return m
end

# A load model
function loadmodel()
    m = Model()
    @variable(m,Pload[time] >= 0)
    return m
end
```

We produce a complete grid system model from Figure 2 by coupling bus nodes on a higher level graph (e.g., representing a regional grid) wherein the nodes are individual grid systems (e.g., representing a local grid). Coupling expressions given directly between two models (or nodes) facilitates quick model building, but more complex models typically require user defined coupling functions around nodes and edges. Snippet 2 simply defines a coupling function around the buses, and we provide the higher level graph to query the edges that connect across individual grid systems. By using a coupling function, a bus can be coupled to an arbitrary number of generators and loads. This also facilitates mixing various generator and load models around a bus. For example, it would be possible to model multiple types of generators subject to different sets of constraints. Similarly, we could also have defined different types of buses with different generators and loads. PLASMO provides flexibility to consider all these alternatives.

Snippet 2: Building a Grid Model using PLASMO Graph and Coupling Function

```
function couplegridnode(m::JuMP.Model,node::Node,graph::Graph)
    busgen = getvariable(node,:busgen)
    busload = getvariable(node,:busload)
    links_in = edges_in(graph,node)
    links_out = edges_out(graph,node)
    @constraint(m,powerbalance[t = time],
    0 == sum{getvariable(links_in[i],:P)[t],i = 1:n_edges_in(graph,node)}
    - sum{getvariable(links_out[i],:P)[t], i = 1:n_edges_out(graph,node)}
    + busgen[t] - busload[t])
end
grid_network = Graph()
addgraph(grid_region,grid_system)   #add grid system as a node
addgraph(grid_region,another_grid_system)   #add another grid system with a bus and loads
edge = Edge(grid_region,busnode,another_busnode)    # define edge between systems
setmodel(edge,tielinemodel())   #set the edge model to a tie-line
setcouplingfunction(grid_region,busnode,couplegridnode)
setcouplingfunction(grid_region,another_busnode,couplegridnode)
```

2.2.2. Example: Coupled Power Grid and Natural Gas Networks: The gas network can be abstracted in the same way as the power grid model. Junction nodes are analogous to the power grid bus in that they receive gas from suppliers (generators) and send it to their respective demands (loads). At a higher level, junctions are connected to other junctions over long distances by links (pipelines) which may or may not include compressor stations (active or passive). Mathematically, the gas network delivery problem can be summarized by the set of equations shown in (2.2). The nonlinear transport equations (2.2b)-(2.2c) capture the spatiotemporal dynamics of flow and pressure. The boundary conditions for the transport equations are given by (2.2e)-(2.2f), and the balance at each node is given by (2.2h). The junction node pressures are given by $\theta_n(\cdot)$, $n \in \mathcal{N}$. Symbols $\Delta\theta_\ell(\cdot)$, $\ell \in \mathcal{L}_a$ denote the compressor pressure increments in the case of active links. $\theta_{snd(\ell)}(\cdot)$ and $\theta_{snd(\ell)}(\cdot) + \Delta\theta_\ell(\cdot)$ are hence, the inlet and outlet pressures for the compressors. The total compression power for the active links is given by (2.2i) and the costs of compression are α_ℓ^P. The gas supply flows are denoted as $s_i(\cdot)$, the delivered gas demands are $d_j(\cdot)$, the gas demand targets are $d_j^{target}(\cdot)$, and the actual delivered gas demands are denoted as $d_j(\cdot)$. Symbol α_j^d denotes the value of the delivered gas and α_ℓ^P is the cost of compression. The system seeks to maximize the amount of gas delivered at the multiple demand locations while minimizing the total compression cost. We note that the variable names and notation used in the natural gas model conflict with those used in the power grid model. Such conflicts can be dealt with in a straightforward manner in PLASMO by modularizing (compartmentalizing) models. With this we also allow users to reuse existing modeling components. This is a useful feature of structured modeling as opposed to using off-the-shelf modeling languages that need to

embed all variables and constraints into a single model.

$$\min \varphi := \int_0^T \left(\sum_{\ell \in \mathcal{L}_a} \alpha_\ell^P P_\ell(\tau) - \sum_{j \in \mathcal{D}} \alpha_j^d d_j(\tau) \right) d\tau \tag{2.2a}$$

$$\text{s.t. } \frac{\partial p_\ell(x,\tau)}{\partial \tau} + \frac{1}{A_\ell} \frac{p_\ell(x,\tau)}{\rho_\ell(x,\tau)} \frac{\partial f_\ell(x,\tau)}{\partial x} = 0, \quad \ell \in \mathcal{L},\ x \in \mathcal{X}_\ell \tag{2.2b}$$

$$\frac{1}{A_\ell} \frac{\partial f_\ell(x,\tau)}{\partial \tau} + \frac{\partial p_\ell(x,\tau)}{\partial x} + \frac{8\lambda_\ell}{\pi^2 D_\ell^5} \frac{f_\ell(x,\tau)|f_\ell(x,\tau)|}{\rho_\ell(x,\tau)} = 0, \quad \ell \in \mathcal{L},\ x \in \mathcal{X}_\ell \tag{2.2c}$$

$$p_\ell(x,\tau) = c^2 \cdot \rho_\ell(x,\tau), \quad \ell \in \mathcal{L},\ x \in \mathcal{X}_\ell \tag{2.2d}$$

$$p_\ell(L_\ell,\tau) = \theta_{rec(\ell)}(\tau),\ \ell \in \mathcal{L} \tag{2.2e}$$

$$p_\ell(0,\tau) = \theta_{snd(\ell)}(\tau),\ \ell \in \mathcal{L}_p \tag{2.2f}$$

$$p_\ell(0,\tau) = \theta_{snd(\ell)}(\tau) + \Delta\theta_\ell(\tau),\ \ell \in \mathcal{L}_a \tag{2.2g}$$

$$\sum_{\ell \in \mathcal{L}_n^{rec}} f_\ell(L_\ell,\tau) - \sum_{\ell \in \mathcal{L}_n^{snd}} f_\ell(0,\tau) + \sum_{i \in \mathcal{S}_n} s_i(\tau) - \sum_{j \in \mathcal{D}_n} d_j(\tau) = 0,\ n \in \mathcal{N} \tag{2.2h}$$

$$P_\ell(\tau) = c_p \cdot T \cdot f_\ell(0,\tau) \left(\left(\frac{\theta_{snd(\ell)}(\tau) + \Delta\theta_\ell(\tau)}{\theta_{snd(\ell)}(\tau)} \right)^{\frac{\gamma-1}{\gamma}} - 1 \right),\ \ell \in \mathcal{L}_a \tag{2.2i}$$

$$0 \le d_j(\tau) \le d_j^{target}(\tau),\ j \in \mathcal{D}. \tag{2.2j}$$

The implementation of the gas dynamics equations is relatively straightforward. Gas systems are connected across graphs as given in Figure 1. In the interest of modeling gas networks, we can use *the same graph* but embed different physical models to nodes to create different types of problem formulations. For instance, we can create steady-state and dynamic versions of pipelines. This gives us the ability to solve easier versions or subcomponents of a model to initialize (warm-start) more complicated ones. This again allows the user to re-use a certain existing graph structure and just modify the nature of the physical component associated to a given node.

2.2.3. Example: Coupling Power Grid and Natural Gas Networks: Finally, the graph hierarchy facilitates coupling of interdependent network models. Extending the multi-level system presented in Figure 2 for the electric grid, Snippet 3 shows how to couple the grid and gas networks using physical associations between grid generators and gas demands. The complete coupled grid-gas system is shown in Figure 3.

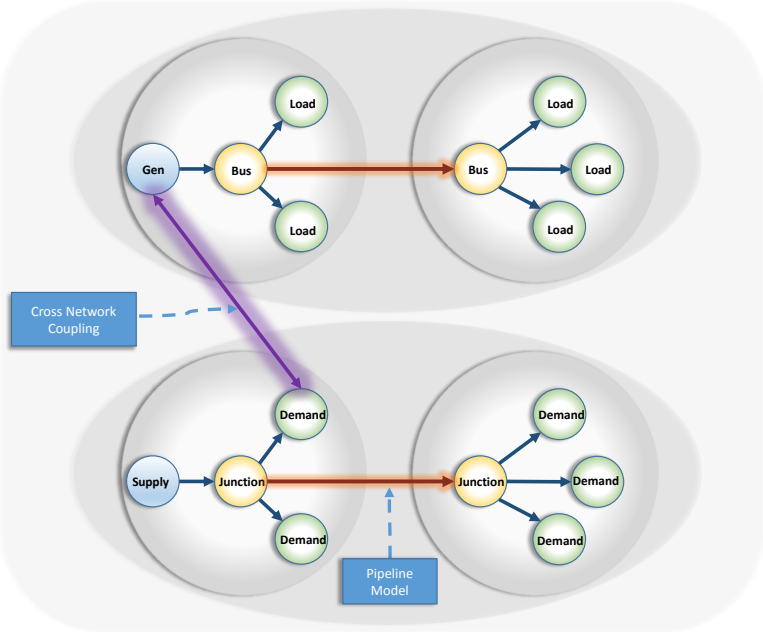

Fig. 3: Coupling grid and gas networks across PLASMO graph hierarchy

Snippet 3: Coupling Power Grid and Gas Network

```
function couple_gas_demand(m::JuMP.Model,edge::Edge,graph::Graph)
    gen_node = getconnectedfrom(edge)
    Pgend = getvariable(gen_node,:Pgend) #gas-fired generator requested gas
    demand_node = getconnectedto(edge)
    fdemand = getvariable(demand_node,:fdemand) #total gas demand
    fdeliver = getvariable(demand_node,:fdeliver) #gas delivered to generator
    @constraint(m,couple[t = time],fdemand[t] == Pgend[t] + eps)
    @constraint(m,limit[t = time], Pgend[t] <= fdeliver[t] - eps)
end
# construct grid-gas model
grid_gas_network = Graph() #create graph
addgraph(grid_gas_network,grid_network) #add gas network as a node
addgraph(grid_gas_network_network) #add grid network as a node
edge = Edge(grid_gas_network,gennode,demandnode) #create an edge at the grid-gas level
setcouplingfunction(grid_gas_network,edge,couple_gas_demand)  #set coupling on tie line
```

2.3. Solving Models, Warm-Starting, and Navigating Results

PLASMO provides interfaces to the same solvers accessible through JuMP. From any given PLASMO graph, it is possible to retrieve a flattened JuMP model that can be directly solved with off-the-shelf solvers. For example, Snippet 4 refers to building a gas network model containing gas supplies, junctions, demands, and transport equations and solves it with Ipopt. Furthermore, querying model results is as simple as querying nodes and edges by traversing the hierarchical graph. Snippet 4 shows how node and edge variables can be directly queried from the solver solution.

Snippet 4: Solving a Gas Network Optimization Model

```
using Ipopt  #use Ipopt to solve nonlinear program
model = getmodel(gas_network)  #builds and retrieves optimization model
model.solver = IpoptSolver()
solve(model)   #gas_network stores references to solution values
query_edge = getedge(gas_network,1)  #get the first edge in a gas_network
pressures = getvalue(query_edge,:px) #retrieves the pressure profile for the edge
```

Because solution data is organized within the graph structure itself, it is possible to exploit the structure of the graph and develop solution strategies such as initializing (warm-starting) highly no-linear problems with more computationally tractable approximations. Figure 4 captures a possible workflow for initializing gas transport dynamics using a steady state solution. Snippet 5 demonstrates how this is achieved in PLASMO using a single graph and swapping out link component models on the same edge (exchanging steady-state and dynamic transport equations).

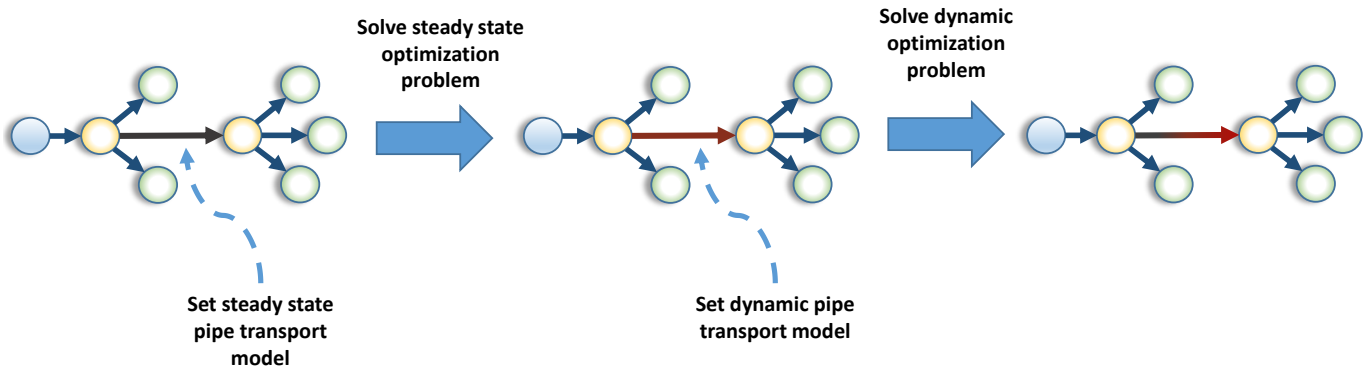

Fig. 4: Using PLASMO to implement a warm-starting optimization strategy

Snippet 5: Developing Warm-Starting Strategies with PLASMO

```
using PlasMO
query_edge = get_edges(gas_network,1)  #get  edge in a gas_network
setmodel(query_edge,ssactivelink()) #append steady-state transport equations
ss_model = getmodel(gas_network) #create steady-state model
solve(model) #solve steady-state model
storecurrentsolution(gas_network) #store solution
setmodel(query_edge,activelink())#append dynamic transport equations
setcurrentsolution(query_edge)#use steady-state solution to initialize model
dynamic_model = generatemodel(gas_network)#set dynamic model
solve(dynamic_model)#solve dynamic model
```

3. DMNetwork

Developing scalable simulation software for large-scale infrastructure networks is challenging due to the underlying unstructured and irregular geometry of the problem. DMNetwork is a native programming framework in the PETSc [2] library that facilities expression of network problems and thereby reduces the application development time.

PETSc is an open source package for the numerical solution of large-scale applications and provides the building blocks for the implementation of large-scale application codes on parallel (and serial) computers. The wide range of sequential and parallel linear and nonlinear solvers, time-stepping methods, preconditioners, reordering strategies, flexible runtime options, ease of code implementation, debugging options, and a comprehensive source code profiler have made PETSc an attractive experimentation platform for developing scientific applications. Along with the numerical solvers, PETSc also provides abstractions through the DM class for managing the application geometry and data.

DMNetwork is a *graph-based* modeling framework. This is a subclass of the DM class that provides for managing geometry and data for unstructured grids, particularly suited for network applications. Its built on top of the DMPlex subclass, a rewritten version of the Sieve [10] framework. Delving on three basic elements of any network: nodes, edges, and data, the framework provides abstractions for easily creating the network layout, partitioning, data movement, and utility routines for extracting connectivity information.

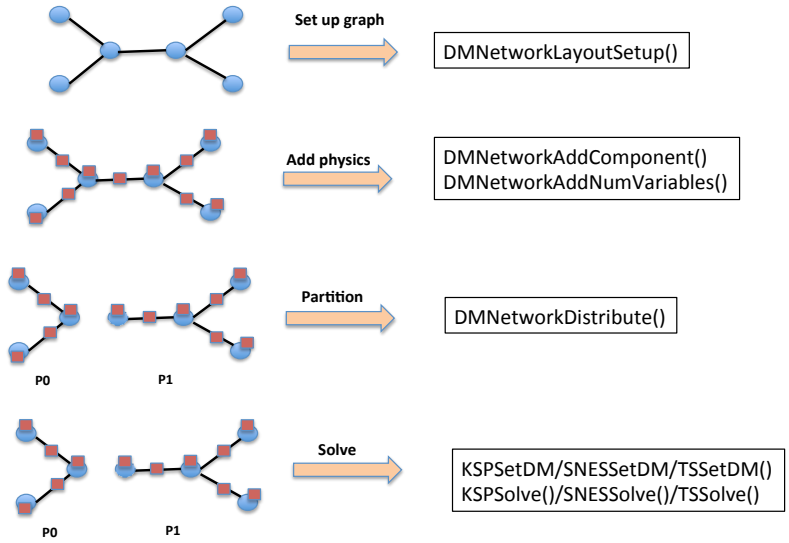

Fig. 5: Steps in creating, partitioning, and solving a network using DMNetwork

A key feature of this framework is that the user only needs to work with higher level application specific abstractions while PETSc takes care of the underlying data management; a feature consistent with the PETSc philosophy. We now discuss the salient features of the DMNetwork framework next and list some of the utility routines. The core features of DMNetwork are:

- Support for assigning different numbers of variables for any node or edge. This is particularly important for networks that comprise of sub-networks having heterogenous characteristics.
- Any data, 'component' as we term it, can be attached with a node or edge. For example, the component could be edge weights or vertex weights for graph problems, or equations describing the physical behavior of the component. Multiple components can be

attached to an edge or node. Note the same abstraction is used in PLASMO .

- Support for partitioning (called edge distribution) of the network graph using ParMetis or Chaco partitioners. Components associated with nodes/edges are also distributed to the appropriate processor when the network is partitioned.
- The framework can create the linear operator or compute derivatives to construct the Jacobian for the network. Global (parallel) vectors and local vectors for residual evaluation can be created by DMNetwork.
- DMNetwork also keeps track of the global and local offsets for use in function evaluation or matrix assembly. It also stores information of the 'ghost' nodes (nodes that need to perform communication with other processors). In other words, once a network is partitioned, DMNetwork automatically redefines local and global variables.
- While doing a calculation, most network applications require information about the edges connected to a node, and/or the nodes covering an edge. The framework provides API routines to extract this information.
- Full compatibility with all PETSc's linear (KSP), nonlinear (SNES), and time-stepping (TS) solvers. This allows simulation of both steady-state and dynamic models. Time-steppers are adaptive, so high-resolution simulations are possible. This is an advantage over PLASMO , in which a fixed time discretization scheme is often used to create coarse but tractable optimization formulations.

Snippet 6 shows the main steps in creating a network object using DMNetwork for the gas network simulation example. Once the network layout has been set up, the components are registered with the network object via the function DMNetworkRegisterComponent() as shown in Snippet 7. This register mechanism provides an "inventory" of the components incident on the network. As with PLASMO , such inventories can be re-used to create applications. The characteristics or data of each component is defined by a struct. The gas network DM can then be used with any of the PETSc's linear (KSP), nonlinear (SNES), or time-stepping (TS) solvers.

Snippet 6: Gas network creation

```
1  /* Create an empty network object */
2  DMNetworkCreate(PETSC_COMM_WORLD,&gasdm);
3  /* Set number of nodes/links */
4  DMNetworkSetSizes(gasdm,gasnet.nnode,gasnet.nlink,PETSC_DETERMINE,PETSC_DETERMINE);
5  /* Add link connectivity */
6  DMNetworkSetEdgeList(gasdm,links);
7  /* Set up the network layout (no components added yet) */
8  DMNetworkLayoutSetUp(gasdm);
```

Snippet 7: Gas network component addition

```
1
2  /* Register the components in the network */
3  DMNetworkRegisterComponent(gasdm,''nodeinfo'',sizeof(struct _p_NODE),&componentkey[0]);
4  DMNetworkRegisterComponent(gasdm,''linkinfo'',sizeof(struct _p_LINK),&componentkey[1]);
5  DMNetworkRegisterComponent(gasdm,''supinfo'',sizeof(struct _p_SUPPLY),&componentkey[2]);
6  DMNetworkRegisterComponent(gasdm,''deminfo'',sizeof(struct _p_DEMAND),&componentkey[3]);
7
8  /* Add network components (node data, link data, supply data, demand data ) */
9  int eStart, eEnd, vStart, vEnd,j;
10
11 DMNetworkGetEdgeRange(gasdm,&eStart,&eEnd);
12 for(i = eStart; i < eEnd; i++) {
13   /* Add the component to this edge */
14   DMNetworkAddComponent(gasdm,i,componentkey[1],&gasnet.links[i-eStart]);
15   /* Add the number of variables */
16   DMNetworkAddNumVariables(gasdm,i,gasnet.links[i-eStart].Nx*gasnet.links[i-eStart].dof);
17 }
18
19 DMNetworkGetVertexRange(gasdm,&vStart,&vEnd);
20 for(i = vStart; i < vEnd; i++) {
21   DMNetworkAddComponent(gasdm,i,componentkey[0],&gasnet.nodes[i-vStart]);
22   DMNetworkAddNumVariables(gasdm,i,2);CHKERRQ(ierr);
23   /* Add supply component if the node has a supply */
24   if (gasnet.nodes[i-vStart].nsup) {
25     for (j=0; j < gasnet.nodes[i-vStart].nsup; j++) {
26       DMNetworkAddComponent(gasdm,i,componentkey[2],&gasnet.supplies[gasnet.nodes[i-vStart].sup[j]]);
27     }
28   }
29   /* Add demand component if the node has a demand */
30   if (gasnet.nodes[i-vStart].ndem) {
31     for (j=0; j < gasnet.nodes[i-vStart].ndem; j++) {
32       DMNetworkAddComponent(gasdm,i,componentkey[3],&gasnet.demands[gasnet.nodes[i-vStart].dem[j]]);
33     }
34   }
35 }
36 DMNetworkSetUp(gasdm);
```

4. Workflows

Computational analysis often requires the execution of optimization and simulation models that are dependent. For instance, the outputs from a model become inputs to another model. In some cases, models are often marginally different and instantiated with different sets of data, variables, constraints, or algorithmic parameters. For example in a recent power system study [8], several optimization models were solved with thousands of different wind power generation scenarios for a Western Electricity Coordinating Council (WECC) test system to evaluate the performance of a given network design. Efficient mechanisms are needed to instantiate and manage multiple computational tasks.

As we have illustrated with the warm-starting example, `PLASMO` can use the graph structure to efficiently instantiate models with different sets of variables, constraints, and parameters. However, `PLASMO` does not enable task management in parallel computing environments. Achieving efficient task management in large computing clusters is challenging because it is necessary to communicate data and allocate appropriate processors to tasks. To enable this, we interface `PLASMO` and `DMNetwork` with `Swift`. `Swift` [13] is a scripting language that creates and manages workflows automatically by specifying how a series of

computational tasks are executed. This workflow management tool also automates collection and distribution of data to computing nodes. By using `Swift`, we can run large parallel jobs without modifying or adjusting `PLASMO` models and scripts.

Snippet 8: Swift script to run PLASMO Gas-Electric instances

```
1  type file;
2  # ------ Inputs ------ #
3  file runGas <"runGas.jl">;
4  file runGrid <"runGrid.jl">;
5  file runGasGrid <"runGasGrid.jl">;
6  file initialGasInput <"initialGasInput.txt">;
7  int n = 1; # Number of iterations for the decoupled systems
8  # ------ Outputs ------ #
9  file gasOutputs[];
10 file gridOutputs[];
11 file coupled <"output/coupled.out">;
12 file comparison <"output/comparison.out">;
13 # ------ Applications ------ #
14 app (file _out) PlaSMO_coupled (file _runfile) {
15   julia @_runfile;
16 }
17 app (file _out) PlaSMO_decoupled (file _runfile, file _input) {
18   julia @_runfile @_input;
19 }
20 app (file _out) CompareSystems (file _gas, file _grid, file _coupled) {
21   julia @_gas, @_grid, @_coupled;
22 }
23 # ------ Workflow Elements ------ #
24 # Iterate decoupled systems
25 foreach i in [1:n] {
26   # run gas model
27   if i == 1 {
28     gasInput = initialGasInput;
29   } else {
30     gasInput = gridOutputs[i-2];
31   }
32   # run grid model
33   gasOutputs[i-1] = PlaSMO_decoupled(runGas, gasInput;
34   gridOutputs[i-1] = PlaSMO_decoupled(runGrid, gasOutput);
35 }
36 # Run the coupled system
37 coupled = PlaSMO_coupled(runGasGrid);
38 # Compare the results
39 comparison = CompareSystem(gasOutputs[n], gridOutputs[n], coupled);
```

Snippet 8 shows the `Swift` script lines to run a number of `PLASMO` models for gas and electricity systems and analyze the results from the runs. `Swift` takes care of workflow management by taking input files and parameters (lines 3-7), executing the runs (lines 25-39), and writing output files. In particular, if this runs on a cluster, the input files and the output files are collected and distributed to computing clusters required for running the `PLASMO` models. `PLASMO` models and a post-processing Julia script are modularized as applications in lines 14-22. Applications `PlaSMO_coupled` and `PlaSMO_decoupled` instantiate and solve optimization models for the coupled gas-electricity system and decoupled systems, respectively. Application `CompareSystems` runs a Julia script to visualize the results as a post-processing step. We highlight that `Swift` directly identifies which tasks of the workflow are parallelizable and which ones are not; and automatically designs a suitable workflow to be executed in a parallel cluster. The `Swift` workflow for the decoupled setting is shown in Figure 6.

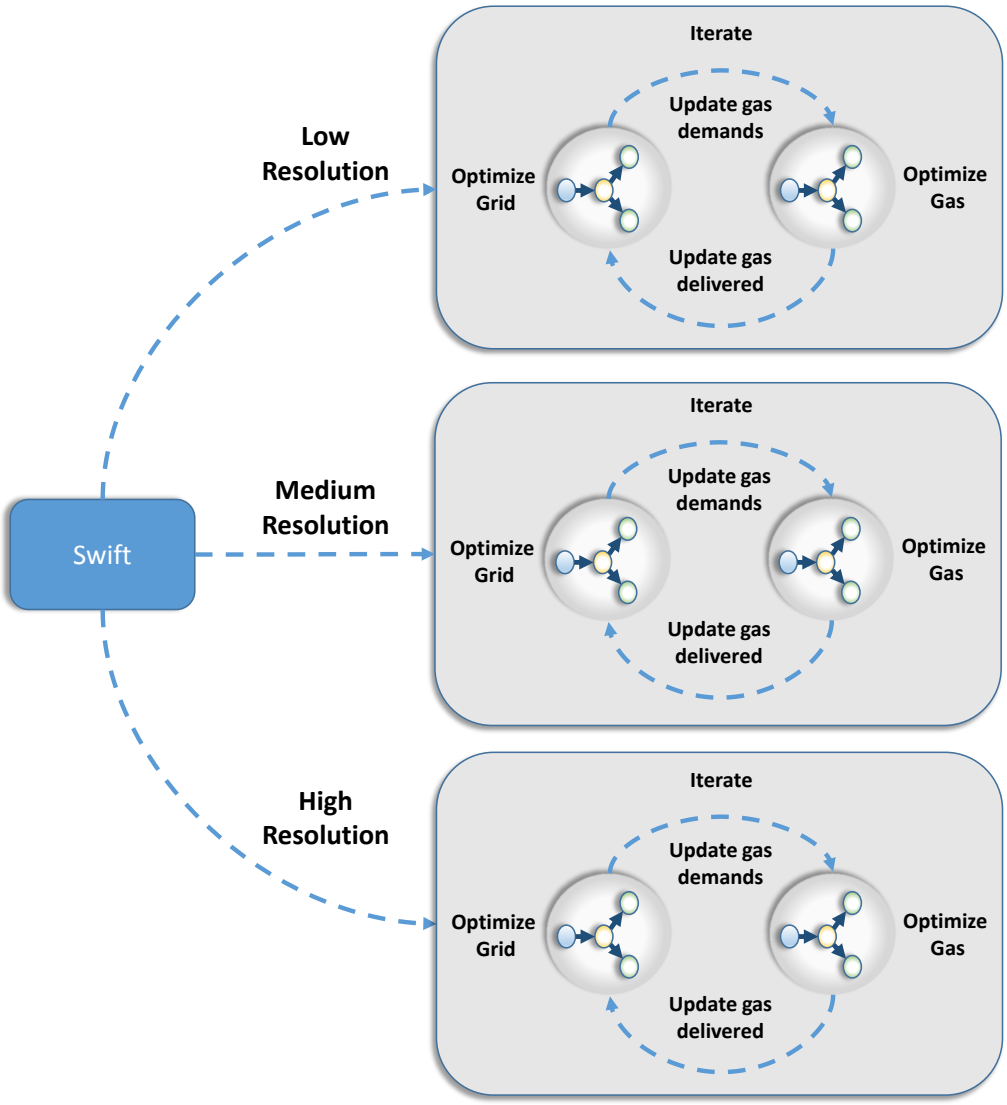

Fig. 6: Integration of `PLASMO` and `Swift` for instantiating a decoupled gas-electric instance.

5. Application Examples

We now demonstrate the application of `PLASMO` , `DMNetwork`, and `Swift` to two model cases. We first illustrate how `PLASMO` can be used to construct and optimize a coarse gas pipeline compression optimization problem, and then validate the solution using a fine resolution simulation in `DMNetwork`. In the second case, we use `PLASMO` to build an interconnected model of the Illinois gas and electrical networks. We implement a `Swift` workflow in which we simulate sequential (decoupled) and coordinated operations.

5.1. Gas Pipeline Optimization

We consider the gas compressor system from [14] sketched in Figure 7. This system is comprised of 13 junctions, 12 pipelines, 10 compressors, one supply at the first junction, and one demand at the last junction. The pressure at the supply is fixed at 34 bar, and the target

pressure at the demand node is between 39 and 41 bar. Suction pressures should be maintained at 34 bar, and the nominal demand flow is 10x10^6 SCM/day. We build the model in PLASMO using (2.2) and model components. Our objective is to develop an optimal gas compression policy subject to power and pressure constraints, and verify that the resulting solution is physically feasible using a high resolution simulation. Each link is discretized using three spatial points to produce a coarse grid, and the full optimization model is produced from the PLASMO graph. We solve the resulting nonlinear optimization problem using IPOPT. The compression policies are then converted into DMNetwork data structures for high-fidelity simulation with PETSc. We use DMNetwork to run the simulation at a higher spatial resolution with both 10 and 50 grid points per link. The validation results are illustrated in Figure 8, where we see that the PLASMO pressure profile is consistent with the verified profile. This shows that, in practice, it is possible to construct a PLASMO graph to quickly solve a coarsened decision problem, and use the same model to verify physical feasibility.

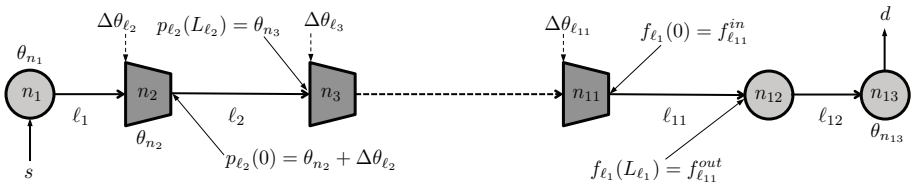

Fig. 7: Representation of a simple gas pipeline system

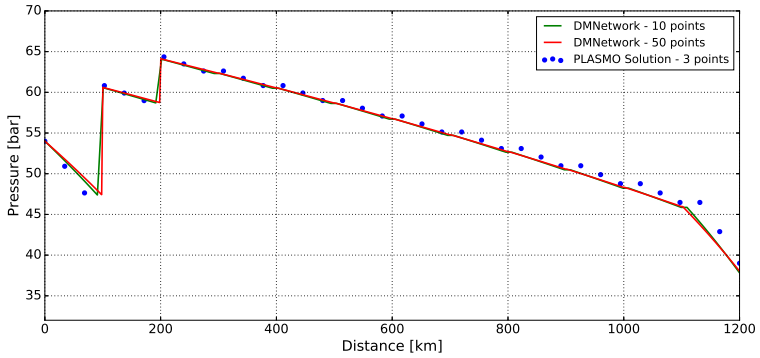

Fig. 8: Coarse PLASMO optimization solution and verification using DMNetwork

Table 2 shows the computational improvement of using high fidelity simulation to verify our optimization problem. Using a high resolution of N_x =200 spatial mesh points and 24 time points over a 24 hour horizon, the optimization problem takes over fifteen minutes to solve on an Intel(R) Core i7 CPU at 2.40GHz. The same optimization problem solves in under three seconds using a coarse resolution of N_x =3 points per pipeline segment at the same time resolution. For comparison, running a dynamic simulation using a N_x =200 point discretization mesh in addition to a 96 point time resolution takes about 20 seconds with over a factor of 2 improvement using DMNetwork's parallel simulation capabilities.

Table 2 Scalability of PLASMO and DMNetwork for high-resolution pipeline system.

PLASMO with IPOPT (Nx=200)	
# CPUs	Solution Time [sec]
1	1006.16
DMNetwork (Nx=200)	
# CPUs	Solution Time [sec]
1	22.49
2	17.79
4	12.85
8	9.88

Table 3 Dispatch costs and computational times for Illinois system with different spatial resolutions (scm= standard cubic meters and M$=million U.S. dollars).

	φ^{grid} [M$]	φ^{gas} [M$]	Solution Time [sec]
Nx = 3	35.32	-13.12	251
Nx = 10	35.32	-13.20	2787
Nx = 20	35.31	-13.15	9321

5.2. Coupled Gas-Electric Networks

We finally demonstrate the extensibility of PLASMO for modeling large coupled gas networks with a case study on the Illinois power grid and gas distribution systems. The power grid system is modeled based on the data set from [15] comprising 2,522 transmission lines, 1,908 buses, 870 electric loads, and 225 generators of which 153 are gas-fired. Similarly, the natural gas network is constructed using the basic topology reported in [6]. The gas network contains 215 pipelines, 157 gas junctions, 12 compression stations, and 4 gas supply points. The coupled topology is shown in Figure 10.

Using PLASMO we developed separate topologies for gas and electric systems, and simply couple the networks at gas-fired generators using the syntax from Snippet 3. Because the entire model is presented in a graph abstraction, we reuse the same graph to solve both the coupled system, as well as implement an iterative (decoupled) strategy. In the case of the iterative strategy, we design the complete coupled topology, and solve grid and gas systems sequentially. Using this approach, we first optimize the Illinois grid dispatch problem defined in equations (2.1), and pass the generator gas demands to the gas network system which optimizes its own problem subject to equations (2.2). Figure 9 shows time profiles for requested and delivered gas demands for two generators under coordinated and decoupled settings. By the second iteration of the decoupled setting, we see both generators have adjusted their gas demands and re-optimized their operation based on actual realized gas delivery. The power grid system thus finds a new feasible dispatch once the delivered demands become known. We then use the same PLASMO graph to solve the more computationally intensive coordinated problem by creating a direct coupling function between generators and

demands. We warm-start the new nonlinear problem using the solution from the decoupled problem. We have found this to be key to solve the coupled problem robustly. This is another benefit of using PLASMO to generate warm-starting strategies. Figure 10 shows the spatial flow profiles throughout the gas system for the coupled and uncoupled problems at peak time. Brighter links correspond to higher flows. We see that the coordinated setting (middle) exhibits more homogeneous profiles compared to the decoupled setting (right). This is because coordination enables better balancing of pressures in the system, as discussed in [6].

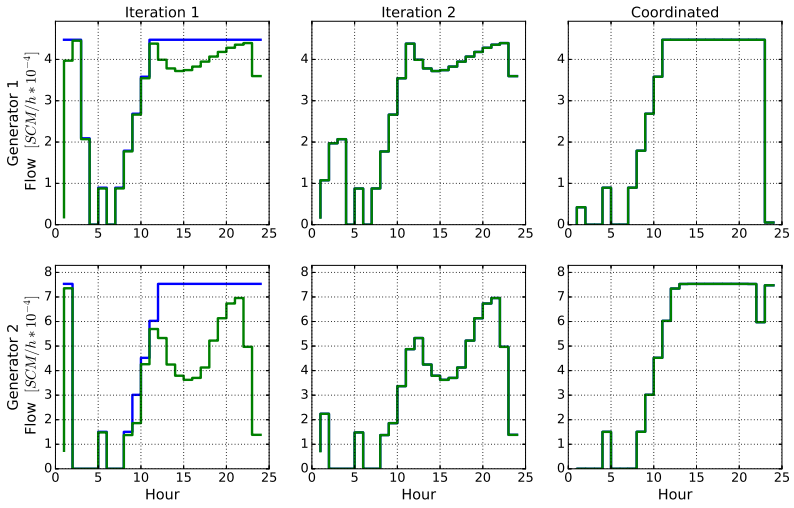

Fig. 9: Time profiles for requested and realized gas delivered for two Illinois power plants.

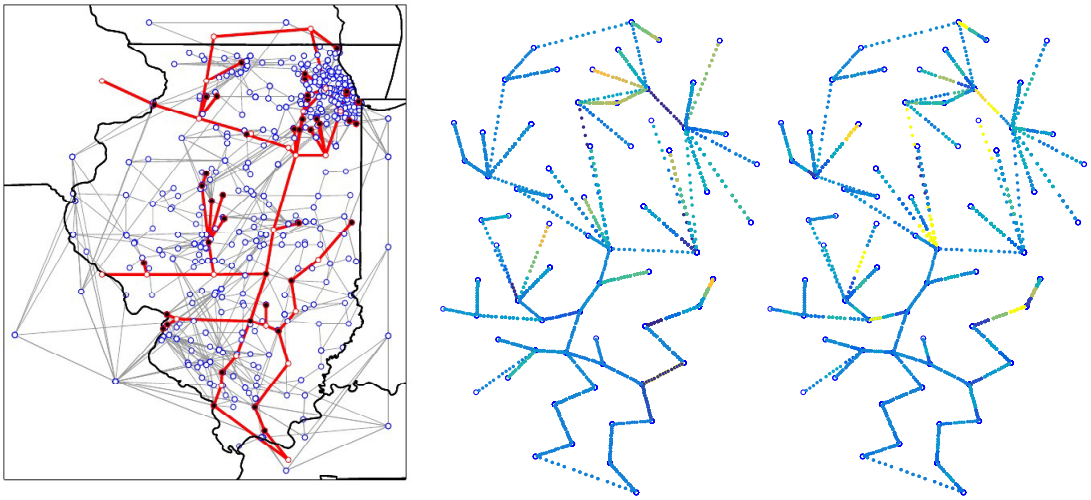

Fig. 10: Illinois grid and gas network (left), spatial flow profiles for coordinated dispatch (middle), and decoupled dispatch (right).

6. Conclusions

We have presented a computational framework that facilitates the construction, instantiation, and analysis of large-scale optimization and simulation applications of coupled energy networks. The framework integrates the optimization modeling package `PLASMO` and the simulation package `DMNetwork` (built around `PETSc`). We also describe how to embed these tools within complex computational workflows using `SWIFT`, which is a tool that facilitates parallel execution of multiple simulation runs and management of input and output data. We have found that the use of a common graph abstraction allows for seamless integration of these tools. In addition, such an abstraction enables the creation of complex models in collaborative environments and facilitates the design of warm-starting and infrastructure coordination strategies.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357.

7. References

[1] S. Abhyankar, B. Smith, H. Zhang, and A. Flueck. Using petsc to develop scalable applications for next-generation power grid. In *Proceedings of the first international workshop on High performance computing, networking and analytics for the power grid*, pages 67–74. ACM, 2011.

[2] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page. http://www.mcs.anl.gov/petsc, 2013.

[3] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *arXiv*, pages 1–37, 2015.

[4] R. G. Carter et al. Pipeline optimization: Dynamic programming after 30 years. In *PSIG Annual Meeting*. Pipeline Simulation Interest Group, 1998.

[5] N. Chiang, C. G. Petra, and V. M. Zavala. Structured nonconvex optimization of large-scale energy systems using PIPS-NLP. In *Proc. of the 18th Power Systems Computation Conference (PSCC), Wroclaw, Poland*, 2014.

[6] N.-Y. Chiang and V. M. Zavala. Large-scale optimal control of interconnected natural gas and electrical transmission systems. *Applied Energy*, 168:226–235, 2016.

[7] I. Dunning, J. Huchette, and M. Lubin. JuMP: A modeling language for mathematical optimization. *arXiv:1508.01982 [math.OC]*, 2015.

[8] K. Kim, F. Yang, V. M. Zavala, and A. A. Chien. Data centers as dispatchable loads to harness stranded power. *arXiv preprint arXiv:1606.00350*, 2016.

[9] K. Kim and V. M. Zavala. Algorithmic innovations and software for the dual decomposition method applied to stochastic mixed-integer programs. *Optimization Online*, 2015.

[10] M. G. Knepley and D. A. Karpeev. Mesh algorithms for PDE with Sieve I: Mesh distribution. Technical Report ANL/MCS-P1455-0907, Argonne National Laboratory, February 2007. ftp://info.mcs.anl.gov/pub/tech_reports/reports/P1455.pdf.

[11] M. Shahidehpour, Y. Fu, and T. Wiedman. Impact of natural gas infrastructure on electric power systems. *Proceedings of the IEEE*, 93(5):1042–1056, 2005.

[12] A. Wächter and L. T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106:25–57, 2006.

[13] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[14] V. M. Zavala. Stochastic optimal control model for natural gas networks. *Computers & Chemical Engineering*, 64:103–113, 2014.

[15] V. M. Zavala, A. Botterud, E. Constantinescu, and J. Wang. Computational and economic limitations of dispatch operations in the next-generation power grid. *2010 IEEE Conference on Innovative Technologies for an Efficient and Reliable Electricity Supply, CITRES 2010*, pages 401–406, 2010.

[16] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Services, 2007 IEEE Congress on*, pages 199–206. IEEE, 2007.

[17] A. Zlotnik, M. Chertkov, and S. Backhaus. Optimal control of transient flow in natural gas networks. In *2015 54th IEEE Conference on Decision and Control (CDC)*, pages 4563–4570. IEEE, 2015.