Graph-Based Modeling and Optimization using Plasmo.jl

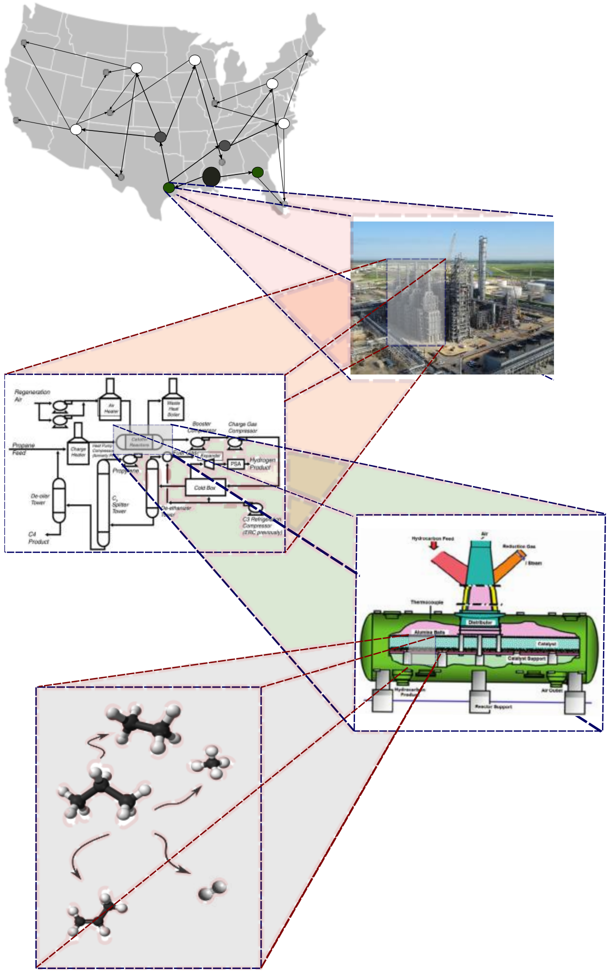
Jordan Jalving & Victor M. Zavala
Department of Chemical and Biological Engineering
University of Wisconsin-Madison

Third Annual JuMP Development Workshop
March 14th, 2019

Motivation: Cyber-Physical Systems

Physical Aspects

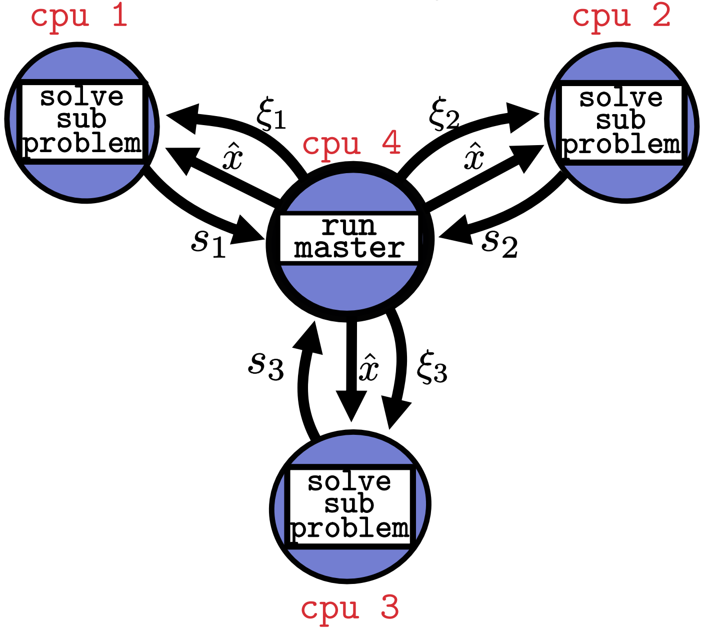
Physical Models and Connections



Challenges: Large-scale optimization problems

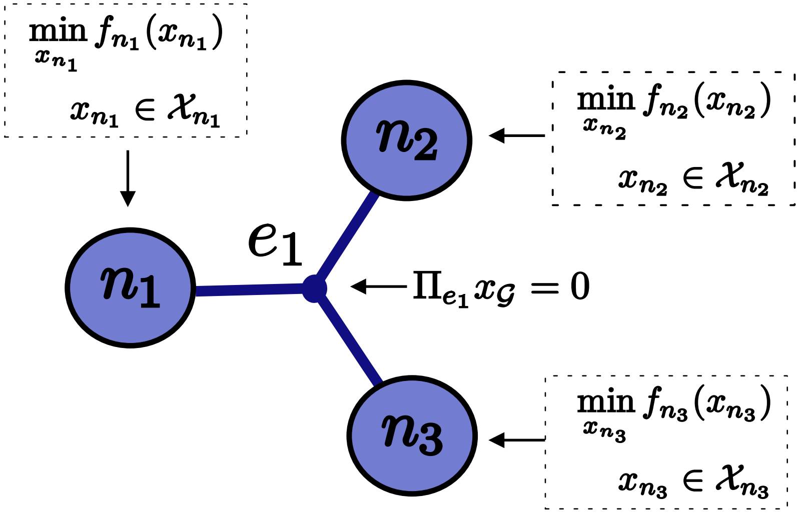
Computing Aspects

Communication/Cyber Connections



Challenges: Simulating real-time systems

Algebraic Graphs (Model Graphs)



$\min_{x_{n_1}} f_{n_1}(x_{n_1})$

$x_{n_1} \in \mathcal{X}_{n_1}$

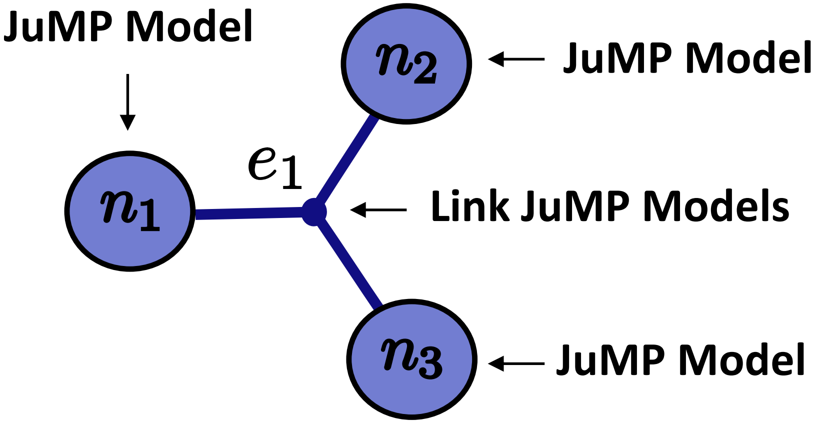
$\min_{x_{n_2}} f_{n_2}(x_{n_2})$

$x_{n_2} \in \mathcal{X}_{n_2}$

$\min_{x_{n_3}} f_{n_3}(x_{n_3})$

$x_{n_3} \in \mathcal{X}_{n_3}$

$\Pi_{e_1} x_{\mathcal{G}} = 0$



JuMP Model

JuMP Model

Link JuMP Models

JuMP Model

Model Graph Formulation

$$\min_{x_{\mathcal{MG}}} \sum_{n \in \mathcal{N}(\mathcal{MG})} f_n(x_n)$$

$$\text{s.t.} \quad x_n \in \mathcal{X}_n, \quad n \in \mathcal{N}(\mathcal{MG})$$

$$\Pi_{\mathcal{MG}} x_{\mathcal{MG}} = 0$$

Connectivity Matrix

$$\begin{pmatrix} \Pi_{e_1,n_1} & \Pi_{e_1,n_2} & \cdots & \Pi_{e_1,n_{|\mathcal{N}|}} \\ \Pi_{e_2,n_1} & \Pi_{e_2,n_2} & \cdots & \Pi_{e_2,n_{|\mathcal{N}|}} \\ \vdots & \vdots & \vdots & \vdots \\ \Pi_{e_{|\mathcal{E}|},n_1} & \Pi_{e_{|\mathcal{E}|},n_2} & \cdots & \Pi_{e_{|\mathcal{E}|},n_{|\mathcal{N}|}} \end{pmatrix}$$

Algebraic Graph Example

Load Packages

```julia
using Plasmo
using Ipopt
```

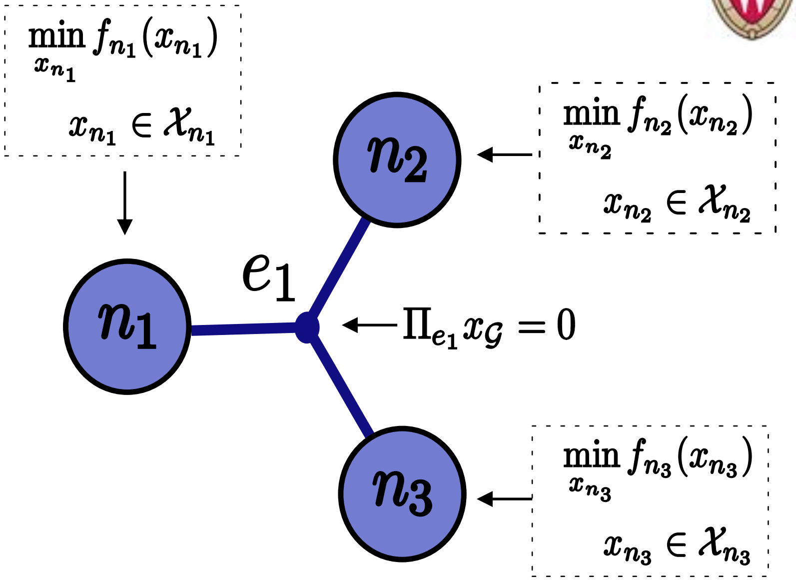
Create a Model-Graph (Algebraic Graph)

```julia
#Create a model graph
mg = ModelGraph(solver = IpoptSolver())

#Add nodes to the model graph
n1 = addnode!(mg)
n2 = addnode!(mg)
n3 = addnode!(mg)

#Associate models with the nodes
setmodel(n1,simple_model1())
setmodel(n2,simple_model2())
setmodel(n3,simple_model3())

#Link models
@linkconstraint(mg,n1[:x] + n2[:x] + n3[:y] == 2)
```

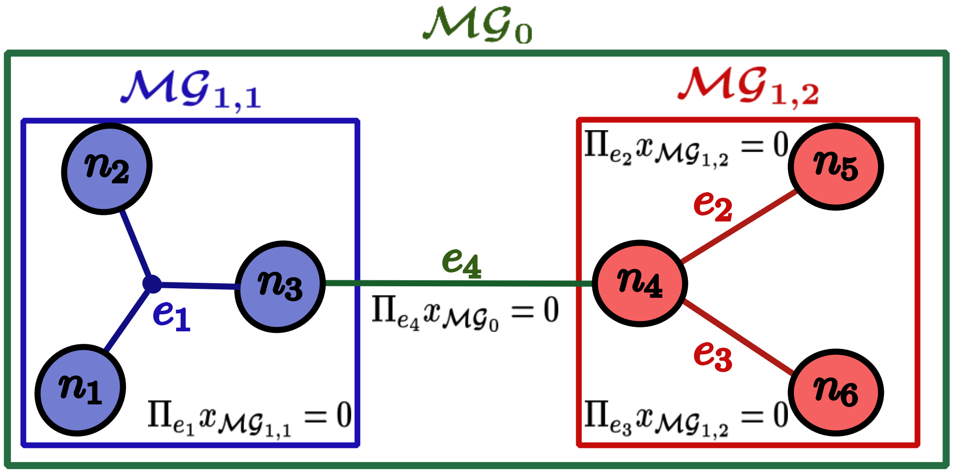
← **JuMP Models**

Solve and Query Results

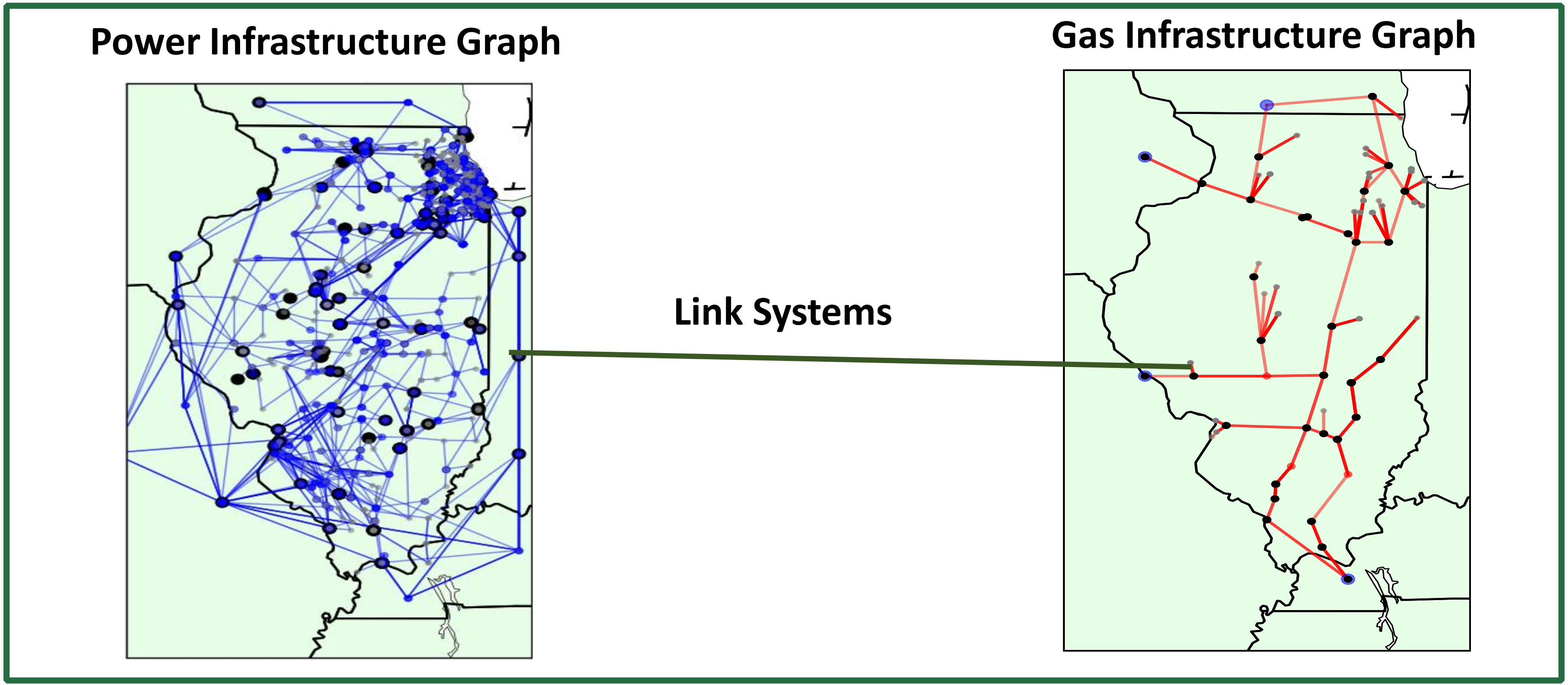
```julia
solve(mg)
result1 = getvalue(n1[:x])
result2 = getvalue(n2[:x])
```



Hierarchical Algebraic Graphs



$$
\begin{aligned}
\min_{x_{\mathcal{MG}_0}} \quad & \sum_{n\in\mathcal{N}(\mathcal{MG}_0)} f_n(x_n) \\
\text{s.t.} \quad & x_n \in \mathcal{X}_n, \quad n \in \mathcal{N}(\mathcal{MG}_0) \\
& \Pi_{\mathcal{MG}_0} x_{\mathcal{MG}_0} = 0 \\
& \Pi_{\mathcal{MG}_{1,1}} x_{\mathcal{MG}_{1,1}} = 0 \\
& \Pi_{\mathcal{MG}_{1,2}} x_{\mathcal{MG}_{1,2}} = 0
\end{aligned}
$$



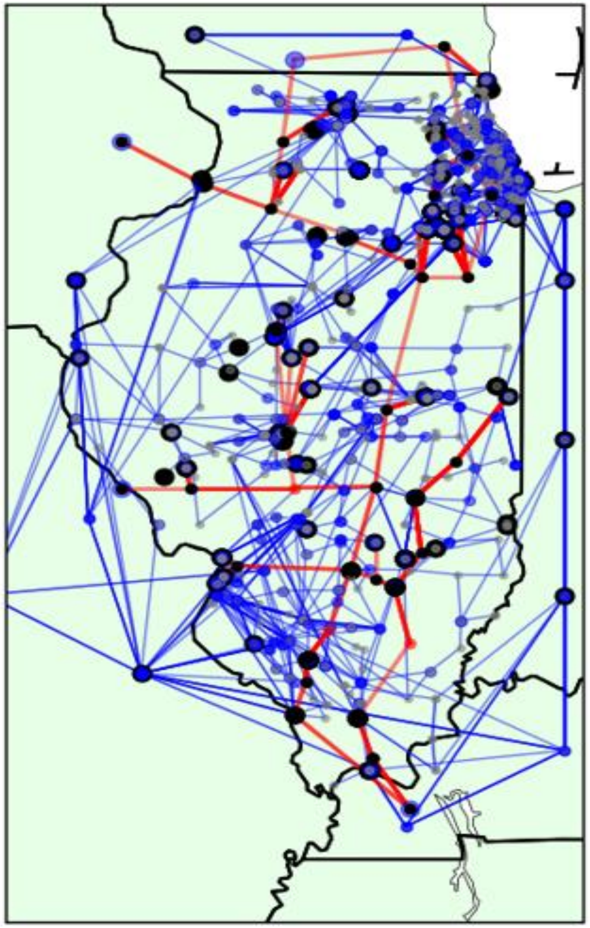
Power Infrastructure Graph

Gas Infrastructure Graph

Link Systems

Hierarchical Modeling Example

```
using Plasmo
using Ipopt

#
# include model functions
#

graph1 = create_illinois_gas_system()
graph2 = create_illinois_grid_system()

combined_system = ModelGraph()
setsolver(combined_system,IpoptSolver())

addsubgraph!(combined_system,graph1)
addsubgraph!(combined_system,graph2)

#Query for nodes
generator_node = getnode(graph1,1)
gas_demand = getnode(graph2,1)

@linkconstraint(combined_system, [t in times],
generator_node[:Pgend][t] == gas_demand[:demand][t])

solution = solve(graph)
```



Decomposition Algorithms

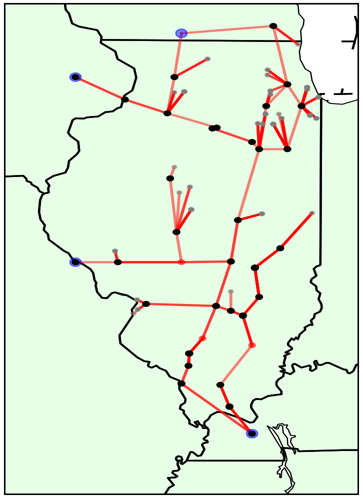
Braulio Brunaud

PLASMO ALGORITHMS

```
using JuMP
using GLPKMathProgInterface
using Plasmo

m1 = Model(solver=GLPKSolverMIP())
#...construct m1

m2 = Model(solver=GLPKSolverMIP())
#.. construct m2

graph = ModelGraph()
setsolver(graph, LagrangeSolver(update_method=:subgradient,
    max_iterations=30))
n1 = addnode!(graph,m1)
n2 = addnode!(graph,m2)

@linkconstraint(graph, [i in 1:2], n1[:xm][i] == n2[:xs][i])

solution = solve(graph)
```

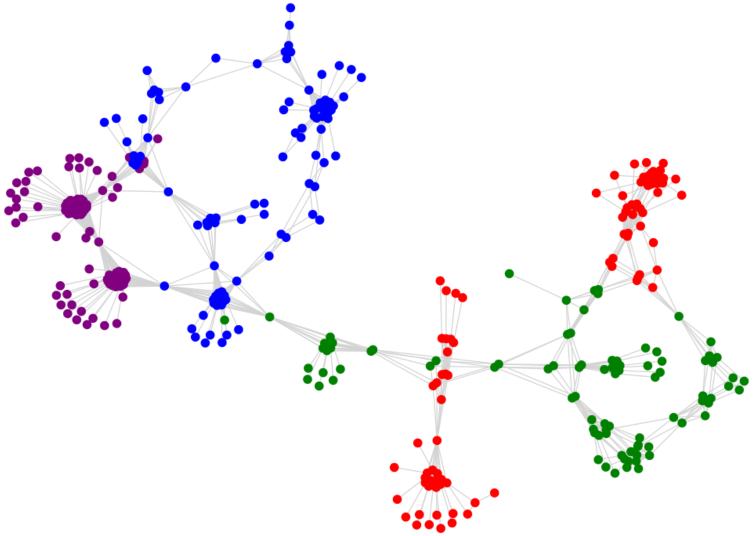
Graph Decomposition

Modeled System



$$
\begin{aligned}
\min_{x_{\mathcal{MG}}} \quad & \sum_{n\in\mathcal{N}(\mathcal{MG})} f_n(x_n) \\
\text{s.t.} \quad & x_n \in \mathcal{X}_n, \quad n \in \mathcal{N}(\mathcal{MG}) \\
& \Pi_{\mathcal{MG}} x_{\mathcal{MG}} = 0
\end{aligned}
$$

Graph Partitions



Linear Algebra Decomposition (e.g., PIPS-NLP)

$$
\left[\begin{array}{cccc|c}
K_{n_1} & & & & \Pi_{n_1}^T \\
& K_{n_2} & & & \Pi_{n_2}^T \\
& & K_{n_3} & & \Pi_{n_3} \\
& & & K_{n_4} & \Pi_{n_4}^T \\
\hline
\Pi_{n_1} & \Pi_{n_2} & \Pi_{n_3} & \Pi_{n_4} &
\end{array}\right]
$$

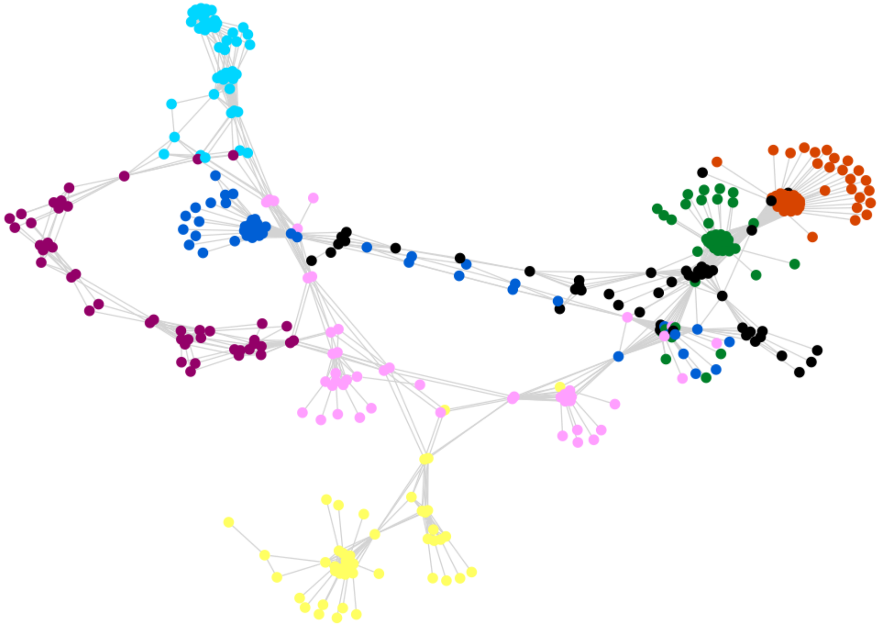
Lagrangean Decomposition

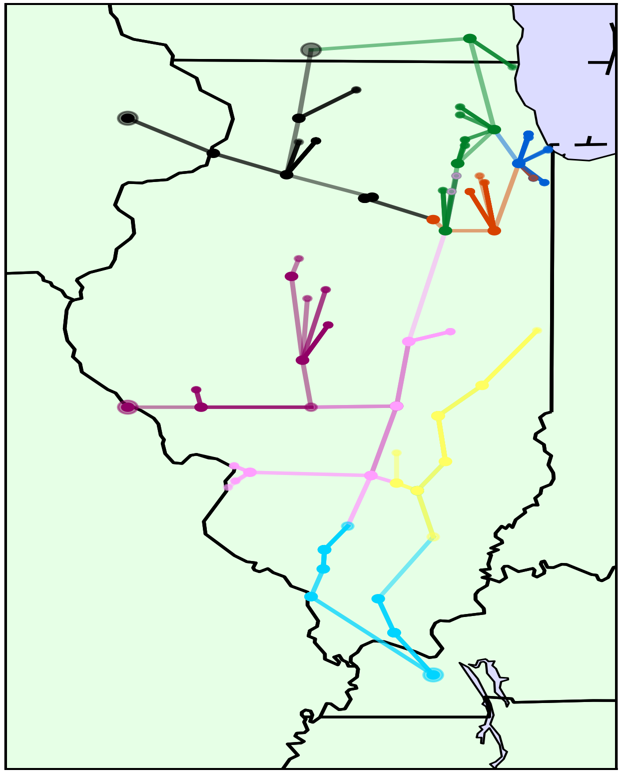
$$
\begin{aligned}
{x_{n_1}}^{+} &= \min_{x_{n_1}} f_{n_1} + \lambda_{\mathcal{MG}}\Pi_{\mathcal{MG}}x_{\mathcal{MG}} \\
{x_{n_2}}^{+} &= \min_{x_{n_2}} f_{n_2} + \lambda_{\mathcal{MG}}\Pi_{\mathcal{MG}}x_{\mathcal{MG}} \\
{x_{n_3}}^{+} &= \min_{x_{n_3}} f_{n_3} + \lambda_{\mathcal{MG}}\Pi_{\mathcal{MG}}x_{\mathcal{MG}} \\
{x_{n_4}}^{+} &= \min_{x_{n_4}} f_{n_4} + \lambda_{\mathcal{MG}}\Pi_{\mathcal{MG}}x_{\mathcal{MG}}
\end{aligned}
$$

Model Graph Partitioning

```julia
using Plasmo
using PlasmoSolverInterface #Contains PipsSolver
using Metis #graph partitioning package
mg = create_illinois_gas_system()

#Obtain a k-way partition of the graph
partitions = Metis.partition(mg,8,method = :KWAY)
setsolver(mg,PipsSolver(n_workers = 8,partitions = partitions))
solve(mg)
```

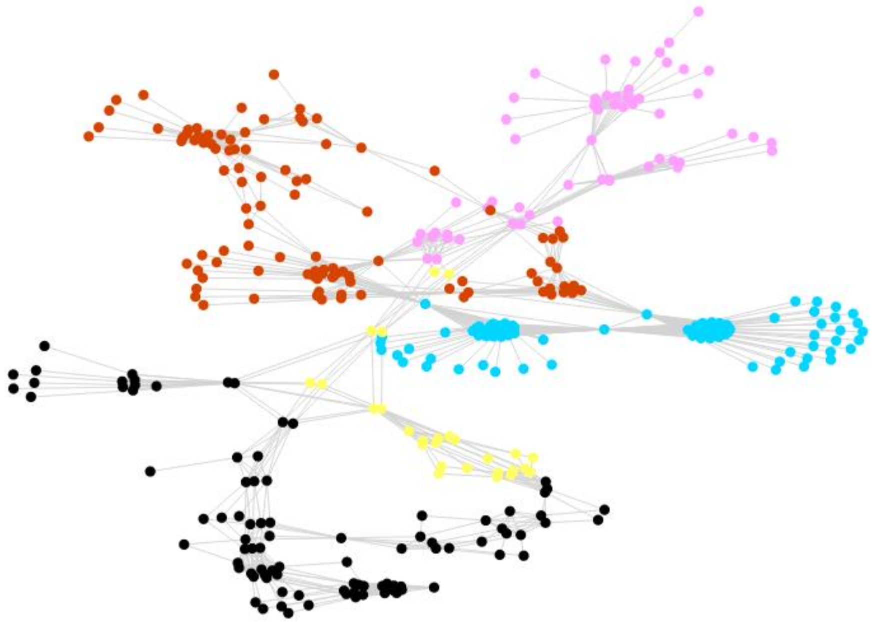
- 1 million variable nonlinear programming problem
- Solves with PIPS-NLP ~40 minutes

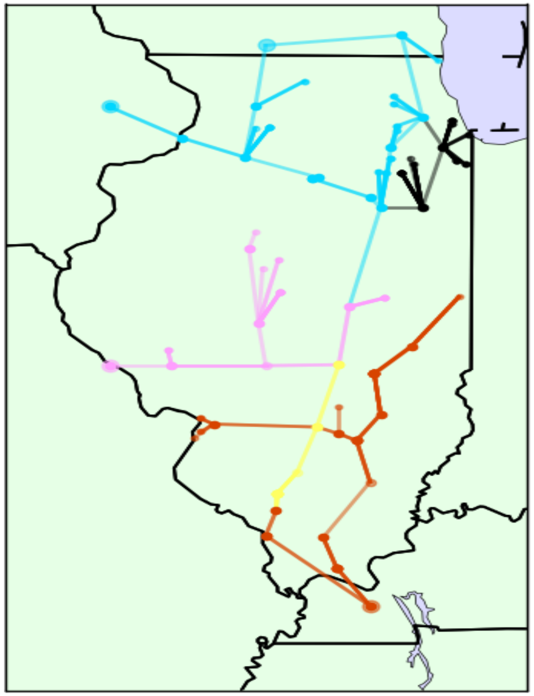




Model Graph Community Detection

```julia
using Plasmo
using PlasmoSolverInterface
using CommunityDetection #Use the community detection package
mg = create_illinois_gas_system()

#Obtain communities through modularity maximization
partitions = community_detection_louvain(mg)
n_partitions = length(partitions)
setsolver(mg,PipsSolver(n_workers = n_partitions,partitions =
    partitions))
solve(mg)
```

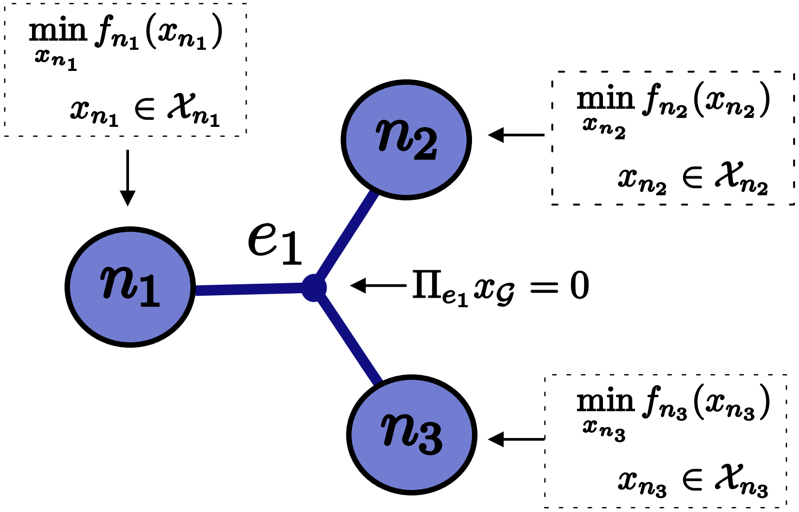




Graph-Based Modeling Abstractions

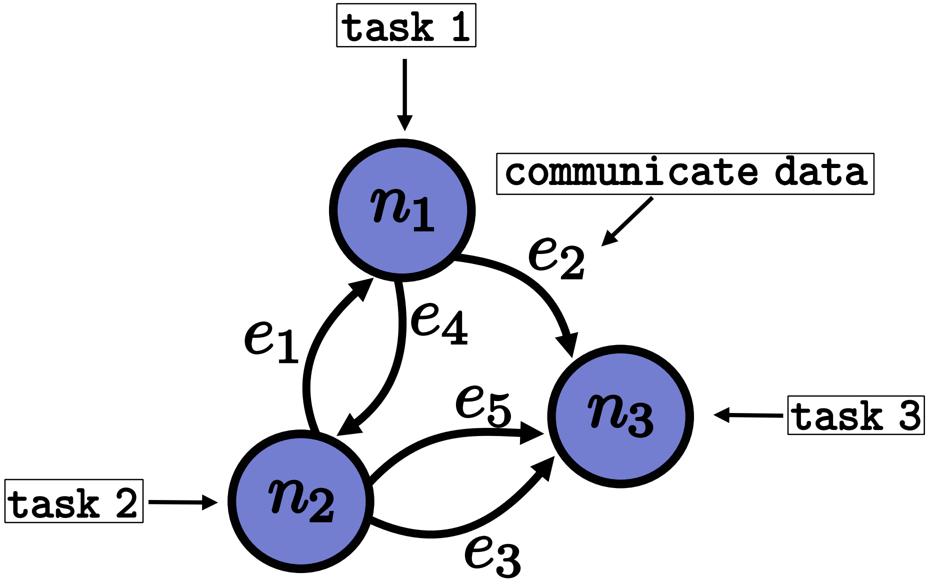
Algebraic Graphs

- Exploit **physical** topology
- **Nodes**=Models, **Edges**=Static Connections
- Exploit topology to **decompose** large-scale optimization problems



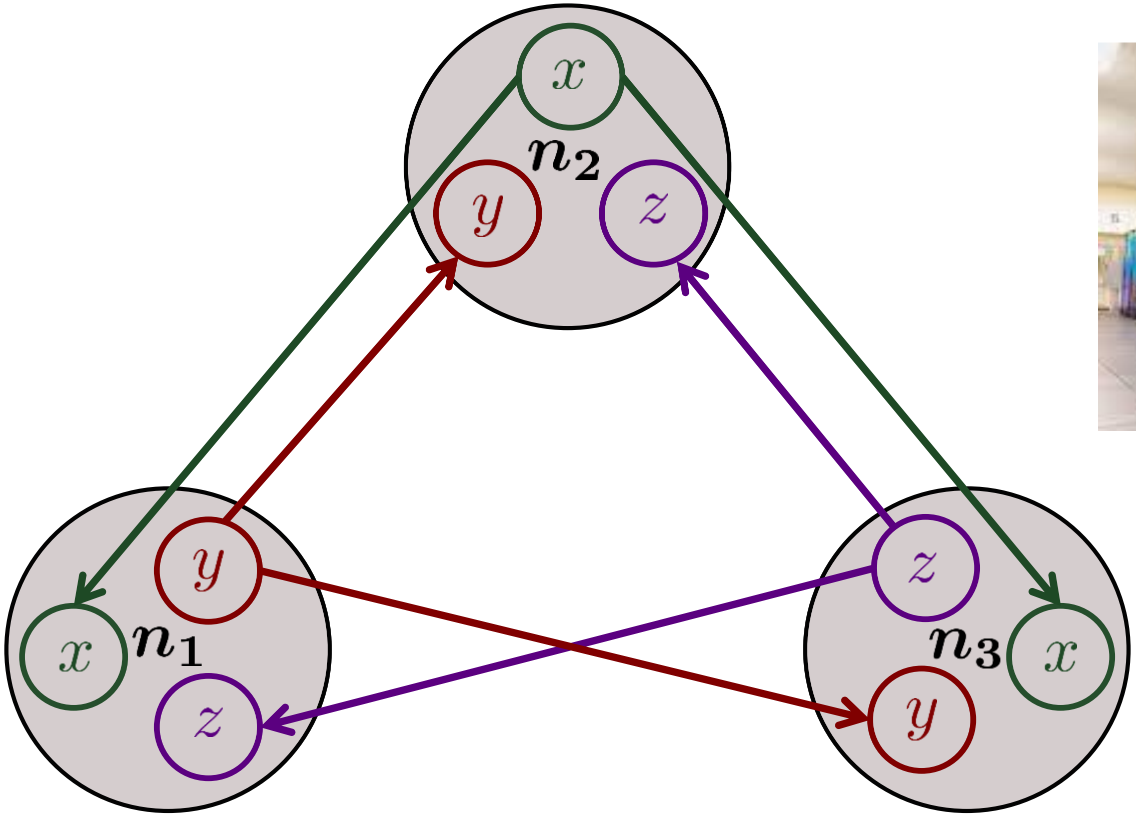
Computing Graphs

- Exploit communication topology
- **Nodes**=Tasks, **Edges**=Dynamic Connections
- Exploit topology to **simulate behavior** of algorithms and computing architectures



Computing Graphs

Challenge: Capture computing aspects (e.g., Asynchronicity, Delays, Latency) of a real-time system



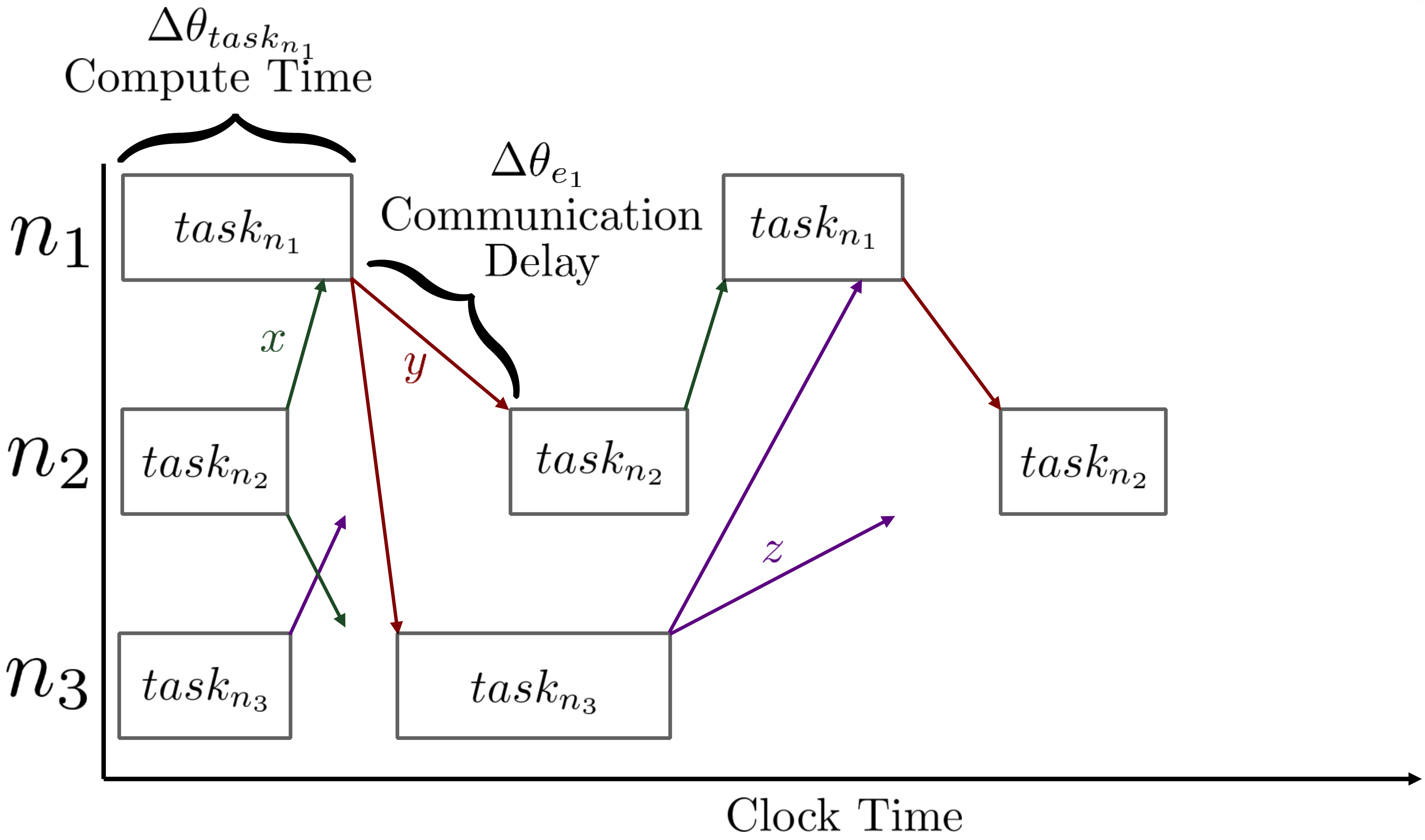
Key Elements

Nodes: Tasks and Attributes (data)
Tasks: Computing time
Edges: Communication
Clock: Scheduling & Management

State-Space Description

$$x_n^+ = f(x_n, u)$$
$$\eta_n^+ = g(\eta)$$

Computing Graphs



- Compute tasks and communication each require time
- A discrete-event queue coordinates simulation timings (the clock)

Simulation of Distributed Optimization Algorithms

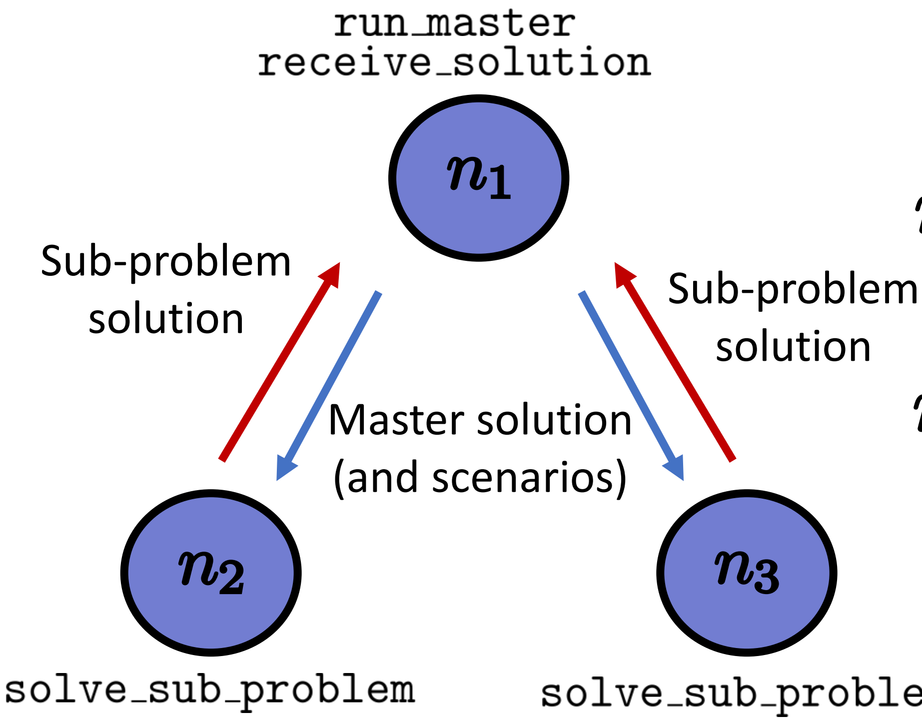
Example: Benders Decomposition

Simulate parallel algorithm variants (synchronous & asynchronous)

$task\ \ 1$: `run_master`

$task\ \ 2$: `receive_solution`

$task\ \ 3$: `solve_sub_problem`



`run_master`
`receive_solution`

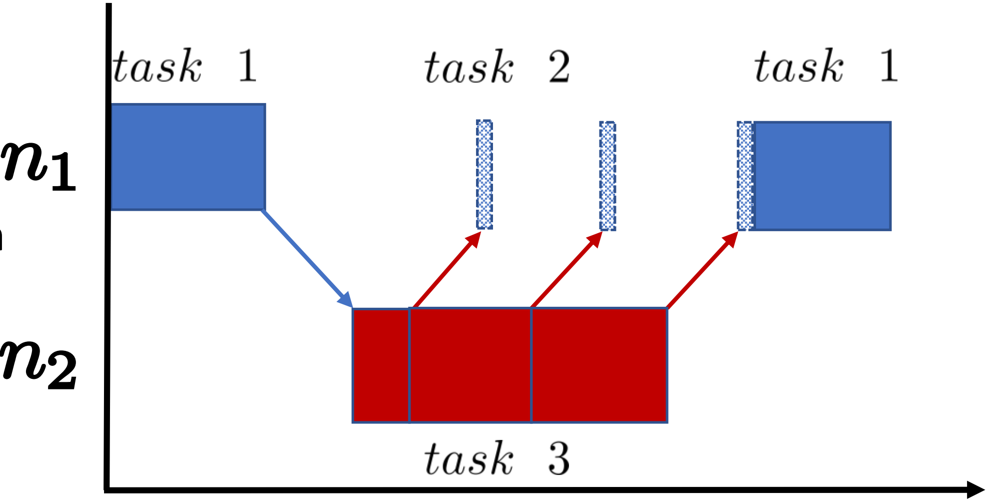
Sub-problem solution

Master solution (and scenarios)

Sub-problem solution

`solve_sub_problem`

`solve_sub_problem`



Idea: Predict Effects of Computing/Communication Delays and Failures

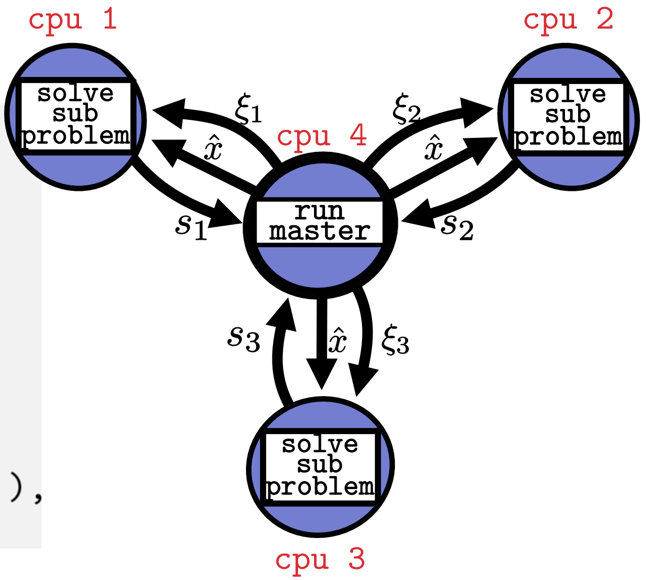
Plasmo.jl Implementation

1. Create Computing Graph

```
graph = ComputingGraph()
```

2. Add Master Node with Attributes (Data) and Tasks

```
#Node
master = addnode!(graph)

#Attributes
@attributes(master,x,C,S,r,ξ[1:N],s[1:N])

#Tasks
@nodetask(graph,master,run_master(master),compute_time = :
    walltime,triggered_by = Updated(r))
@nodetask(graph,master,receive_solution[i = 1:N](master,s[i]),
    compute_time = 0, triggered_by = Received(s[i]))
```



3. Initialize Graph

```
schedulesignal(graph,master,signal_execute(run_master),time = 0)
```

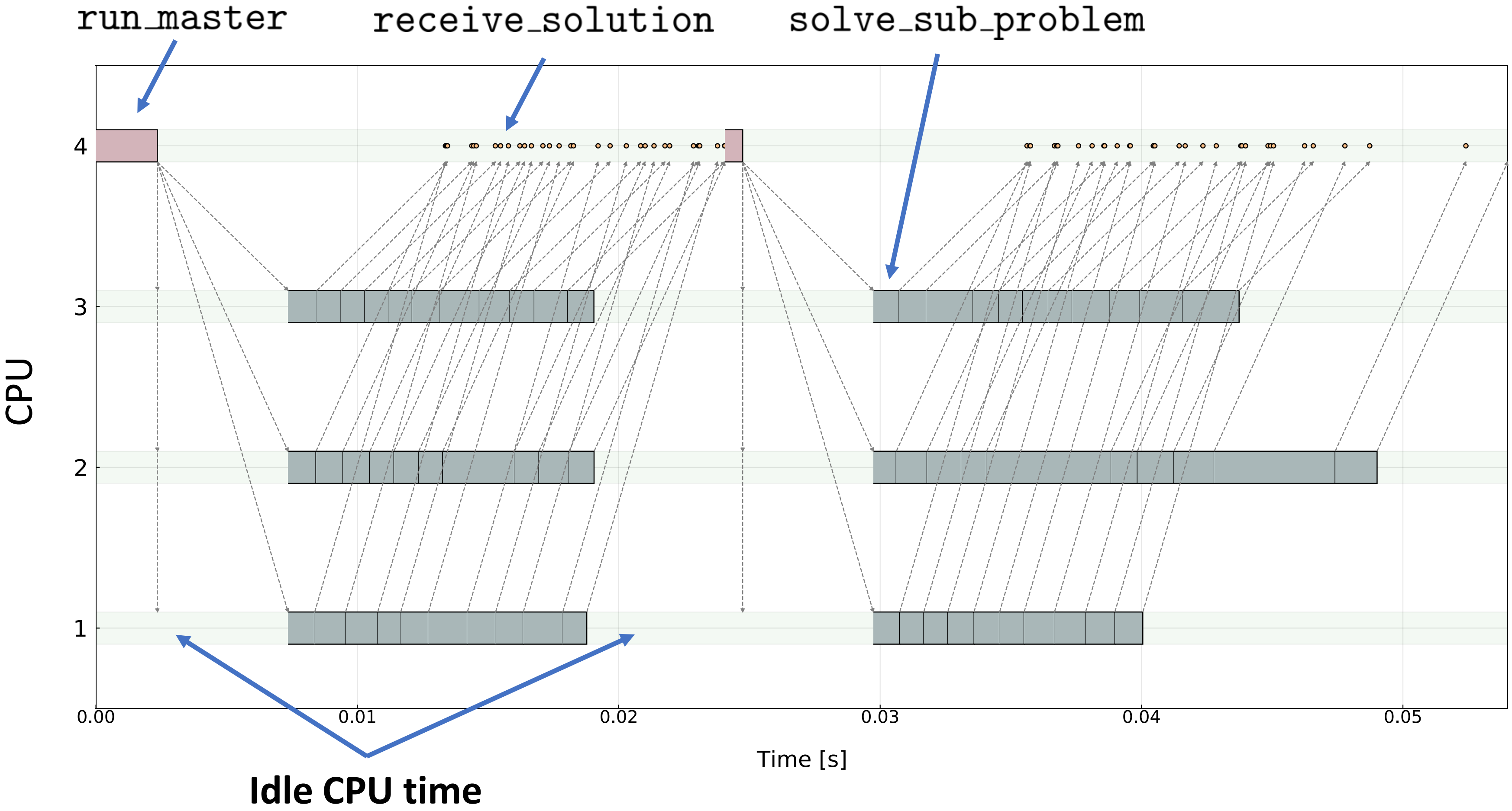
4. Add Sub-nodes and Connections

```
N = 3
for i = 1:N
    subnode = addnode!(graph)
    @attributes(subnode,x,ξ,s)
    @nodetask(graph,subnode,solve_subproblem,triggered_by =
        Received(ξ) , compute_time = :walltime)
    #Connections
    @connect(graph,master[:x] => subnode[:x],send_on = Updated(x))
    @connect(graph,master[:ξ][i] => subnode[:ξ],delay = 0.005)
    @connect(graph,subnode[:s] => master[:s][i],delay = 0.005)
end
```

5. Execute Computing Graph

```
execute!(graph)
```

Synchronous Benders Algorithm

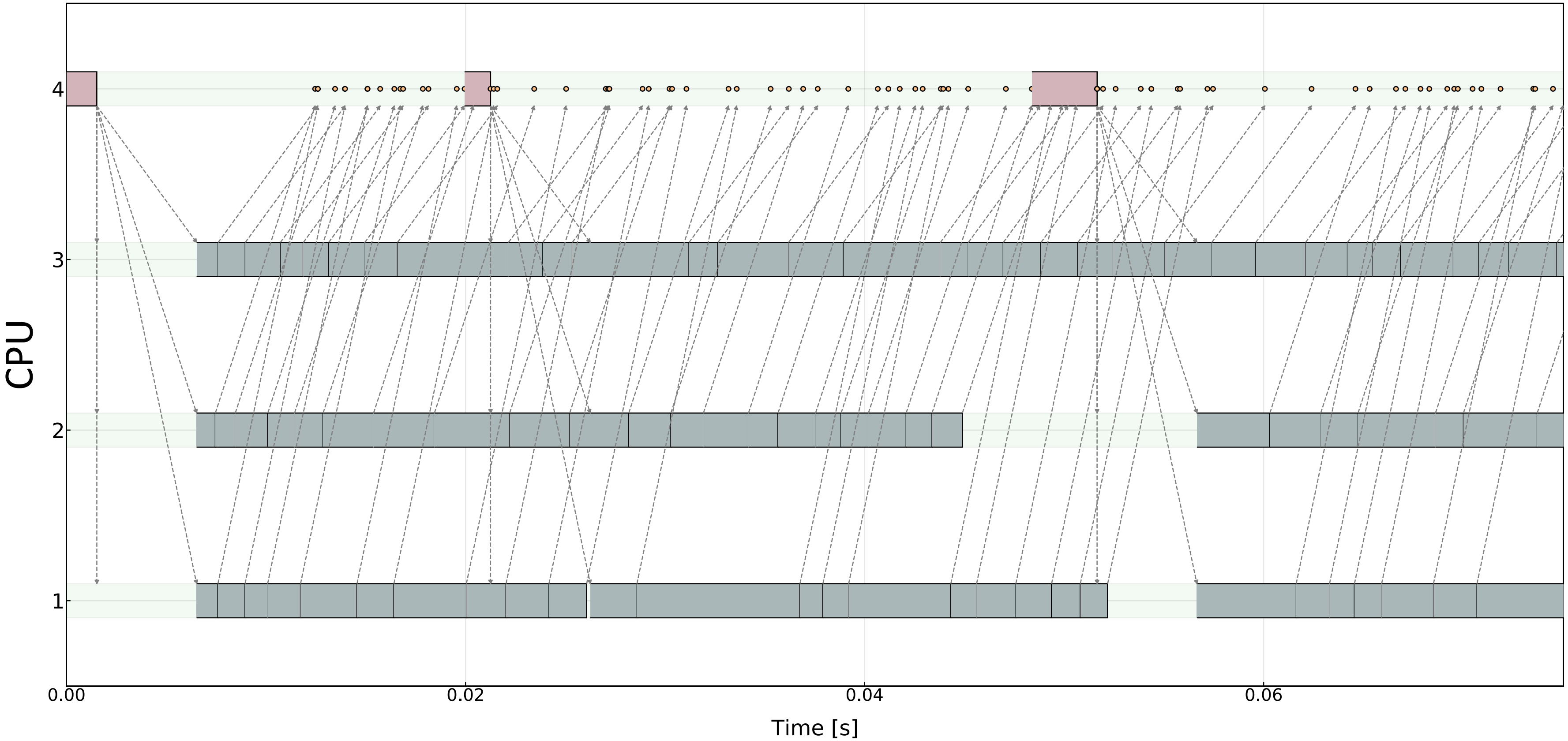


`run_master` `receive_solution` `solve_sub_problem`

Idle CPU time

- Simulation Predicts **Poor Parallel Efficiency** (Idle Processors)

Asynchronous Benders Algorithm



- Simulation predicts much **higher parallel efficiency** (but longer solution time)

Thank You

Plasmo.jl

https://github.com/zavalab/Plasmo.jl