Modelling and solving SC-ACOPF in parallel

Cosmin G. Petra

Argonne National Laboratory

Joint work with

Feng Qiang (ANL), Joey Huchette (MIT), Miles Lubin (MIT), Mihai Anitescu (ANL)

FERC Technical Conference on Increasing Market and Planning Efficiency through Improved Software

June 28, 2016

Washington DC

Outline

- Our background is in HPC optimization solvers
  - Targeting very large-scale optimization problems (billions of decisions variables and constraints)
  - Making the best use of the problem structure to parallelize optimization and computations
- PIPS parallel solvers suite as the result of several years of development
  - Structured problems such as stochastic optimization
  - Security-constrained AC OPF has similar structure
- StructJuMP is the front-end for problem specification, a.k.a. algebraic modelling
  - Easy-to-use yet fully parallel and HPC ready

Goal: provide a complete modeling+solving HPC framework for solving large-scale optimization problems.

Block-angular optimization problems

Example: stochastic optimization problems

$$
\begin{array}{lllllll}
\min & c_0^T x + \sum_{i=1}^N c_i^T y_i & & & & & \\
\text{s.t.} & Ax & & & & & = b_0, \\
 & T_1 x + & W_1 y_1 & & & & = b_1, \\
 & T_2 x + & & W_2 y_2 & & & = b_2, \\
 & \vdots & & & \ddots & & \vdots \\
 & T_N x + & & & & W_N y_N & = b_N, \\
 & x \geq 0, & y_1 \geq 0, & y_2 \geq 0, & \ldots, & y_N \geq 0. &
\end{array}
$$

Large instances with 1000s of scenarios could have billions of variables and constraints, requiring memory distributed parallel computing.

Parallel optimization solvers

PIPS suite of solvers for (continuous) structured optimization
PIPS-IPM, **PIPS-NLP** (interior point), and **PIPS-S** (simplex)

https://github.com/Argonne-National-Laboratory/PIPS/

Structured optimization problems result in structured linear systems

$$\begin{bmatrix} K_1 & & & B_1 \\ & \ddots & & \vdots \\ & & K_N & B_N \\ B_1^T & \dots & B_N^T & K_0 \end{bmatrix} \begin{bmatrix} \Delta z_1 \\ \vdots \\ \Delta z_N \\ \Delta z_0 \end{bmatrix} = \begin{bmatrix} r_1 \\ \vdots \\ r_N \\ r_0 \end{bmatrix}$$

Schur complement decomposition of linear algebra

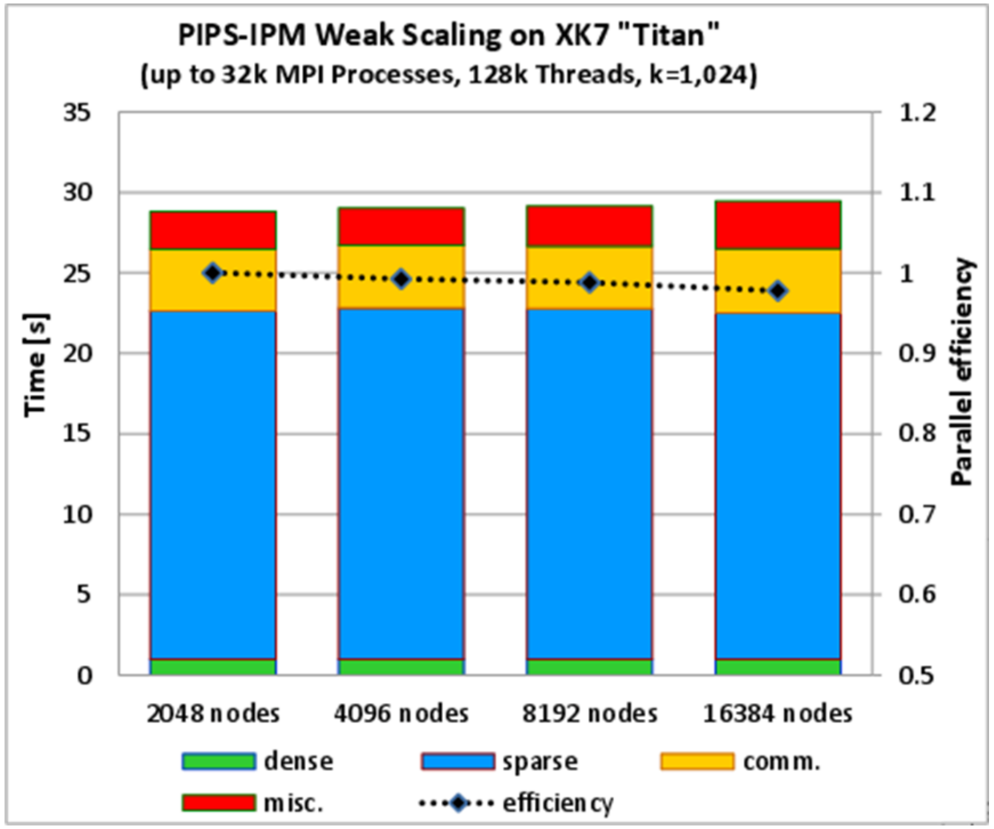
$$\begin{bmatrix} K_1 & & & B_1 \\ & \ddots & & \vdots \\ & & K_N & B_N \\ B_1^T & \dots & B_N^T & K_0 \end{bmatrix} \begin{bmatrix} \Delta z_1 \\ \vdots \\ \Delta z_N \\ \Delta z_0 \end{bmatrix} = \begin{bmatrix} r_1 \\ \vdots \\ r_N \\ r_0 \end{bmatrix}$$

Block elimination

$$\left(K_0 - \sum_{i=1}^{N} B_i^T K_i^{-1} B_i \right) \Delta z_0 = r_0 - \sum_{i=1}^{N} B_i^T K_i^{-1} r_i$$

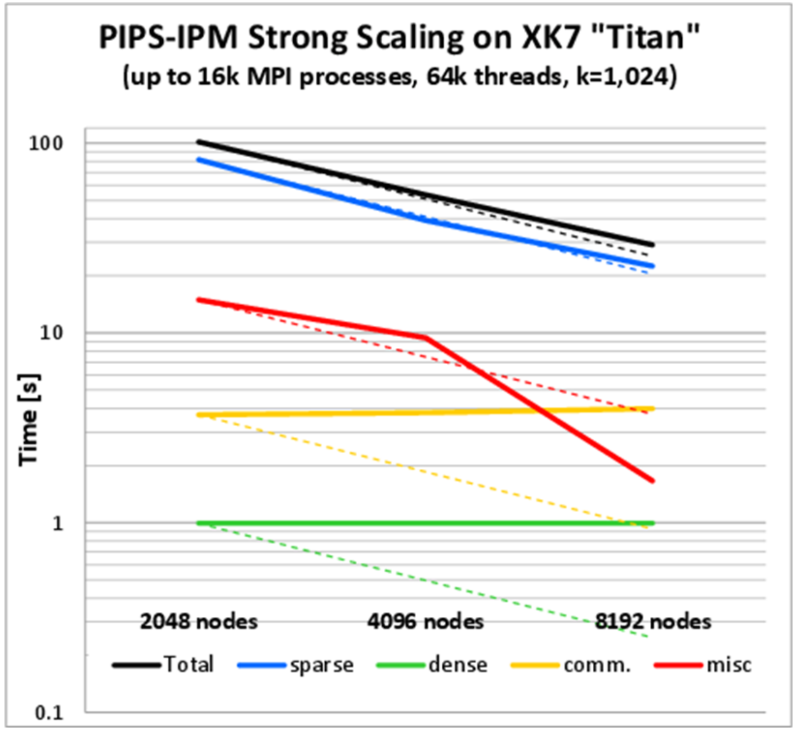
The matrix $C := K_0 - \sum_{i=1}^{N} B_i^T K_i^{-1} B_i$ is the Schur-complement of the diagonal $K_1, \dots, K_N$ block.

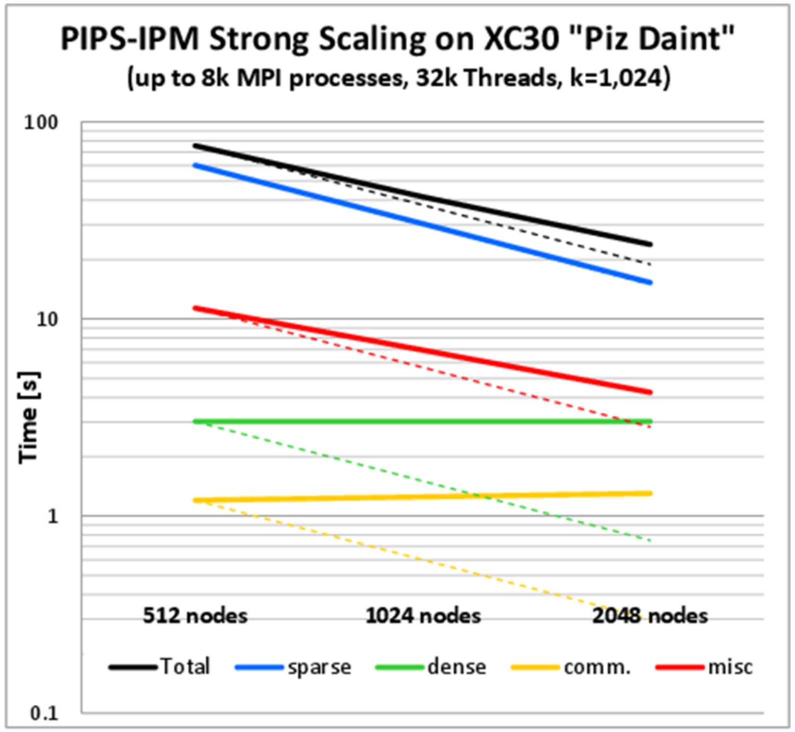
Weak scaling efficiency – Titan @ Oak Ridge National Lab



Largest instance has 4.08 billion decision variables and 4.12 billion constraints.
16K nodes (128K cores) used in the largest run.

Strong scaling – Titan and "Piz Daint" (@ Swiss National Computing Center)





The instance used in the XK7 runs has 4.08 billion decision variables and 4.12 billion constraints.

Structure-exploiting solvers generally scale.

Does the modelling scale?

Modelling structured optimization problems efficiently in parallel – a wish list

- Algebraic modelling language/framework
  - easy-to-express syntax, similar to the mathematical abstractions
  - “high performance”
    - scalable and efficient models generation in parallel (data distributed and localized)
    - code speed – ideally C/Fortran speed
    - minimum I/O
  - transparently passes structure to the optimization solver, yet solver agnostic
  - quick development; easy to specialize and/or extend
  - plug-and-play with optimization solvers (generally Fortran, C, C++ codes)

- Existing modelling frameworks with parallel capabilities: SML (Grothey et al., 2009), PySP (Watson et al, 2012), PSMG (Qiang and Grothey, 2014)

What is an algebraic modeling language (AML) for optimization?

- Aimed at quickly specifying the optimization problem by domain specialists with no knowledge about optimization algorithms/software and computing, and minimal programming skills.
- Offers concise, mathematical-like syntax to allow closed-form (algebraic) expressions for constraints and objective
- Separates the mathematical model from the input data
- Automatic computations of objective, constraints and their derivatives needed by the optimization solver
- Solver-agnostic, can switch between different solver codes
  - achieved based on generic solver interface
- Examples: Ampl, Gams, Xpress, etc.

JuMP – an Algebraic Modeling Language in Julia

- Mathematically natural syntax for problem specification

$$\begin{aligned} \max \quad & \sum_{i=1}^{N} p_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{N} w_i x_i \leq C \\ & 0 \leq x \leq 1 \end{aligned}$$

```
#maximize revenue subject to prescribed capacity
m = Model(:Max)
@variable(m, 0 <= x[1:N] <= 1)
@objective(m, sum{profit[j] * x[j], j=1:N})
@constraint(m, sum{weight[j]* x[j], j=1:N} <= C)

solve(m) #solving using Ipopt
```

- Open source, C/C++ like performance

- Allows using different optimization solvers (via MathProgBase.jl) and can be easily embedded in domain-specific applications

- Developed by collaborators at MIT (Miles Lubin, Iain Dunning, Joey Huchette)

Julia

- Fresh approach for technical computing (http://julialang.org/)
- User friendly and syntax is similar to Matlab
  - Dynamic language with interactive command-line Read-eval-print loop
- C-like performance.
  - Just-In-Time compilation and generate native assembly code
- Open source with a large and fast growing community behind
- Runs on workstations, clusters, cloud and HPC platforms

- JuMP performance

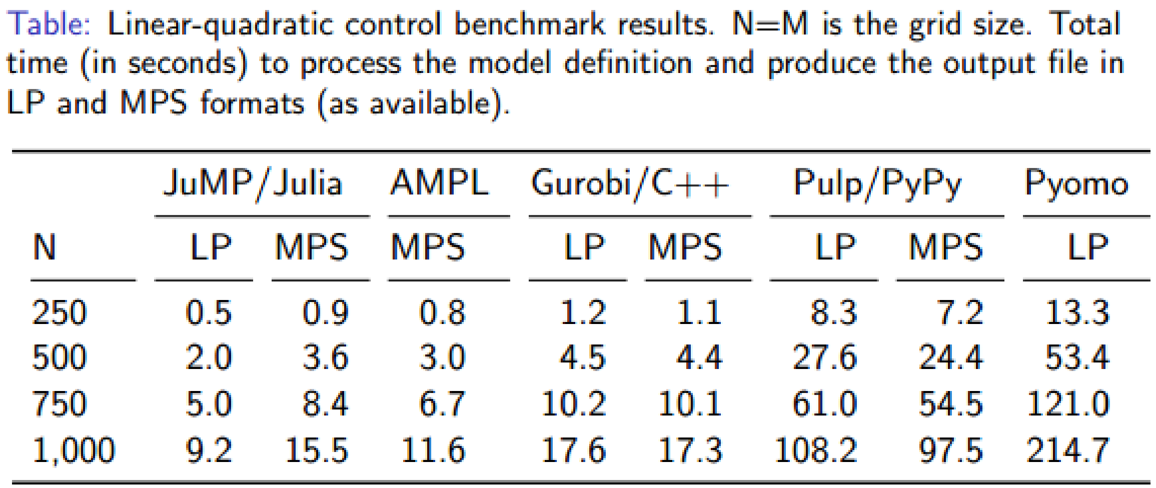


Table: Linear-quadratic control benchmark results. N=M is the grid size. Total time (in seconds) to process the model definition and produce the output file in LP and MPS formats (as available).

	JuMP/Julia		AMPL	Gurobi/C++		Pulp/PyPy		Pyomo
N	LP	MPS	MPS	LP	MPS	LP	MPS	LP
250	0.5	0.9	0.8	1.2	1.1	8.3	7.2	13.3
500	2.0	3.6	3.0	4.5	4.4	27.6	24.4	53.4
750	5.0	8.4	6.7	10.2	10.1	61.0	54.5	121.0
1,000	9.2	15.5	11.6	17.6	17.3	108.2	97.5	214.7

StructJuMP – Parallel AML for Structured Optimization Problems

- Uses full syntax features from JuMP:
  - Eg. @variable, @constraint, @NLconstraint, etc.
- Minimal additional syntax
- Parallel model manipulation and function/derivatives evaluation using MPI
- Targeted at block angular structures, both LP and NLP
  - Stochastic optimization is one such example

$$\begin{array}{lllllll}
\min & c_0^T x + \sum_{i=1}^N c_i^T y_i & & & & & \\
\text{s.t.} & Ax & & & & & = b_0, \\
& T_1 x + & W_1 y_1 & & & & = b_1, \\
& T_2 x + & & W_2 y_2 & & & = b_2, \\
& \vdots & & & \ddots & & \vdots \\
& T_N x + & & & & W_N y_N & = b_N, \\
& x \geq 0, & y_1 \geq 0, & y_2 \geq 0, & \ldots, & y_N \geq 0. &
\end{array}$$

$$\begin{array}{lll}
\min & f_0(x) + \sum_{i=1}^{N} f_i(x, y_i) & \\
\text{s.t.} & g_0(x) = b_0 & \\
& g_i(x, y_i) = b_i, & i = 1, \cdots, N \\
& x \geq 0, y_i \geq 0, & i = 1, \cdots, N
\end{array}$$

Availability

StructJuMP.jl:
https://github.com/StructJuMP/StructJuMP.jl

StructJuMPSolverInterface.jl:
https://github.com/StructJuMP/StructJuMPSolverInterface.jl

SC-ACOPF Model – variables definitions

```
opfdata = opf_loaddata(casename)
...
opfmodel = StructuredModel(num_scenarios=nscen)
…
nbus = length(buses); nline = length(lines); ngen = length(generators);

YffR,YffI,YttR,YttI,YftR,YftI,YtfR,YtfI,YshR,YshI = computeAdmitances(lines,
buses, baseMVA)

@variable(opfmodel, gens[i].Pmin <= Pg[i=1:ngen] <= gens[i].Pmax)
@variable(opfmodel, -0.05*gen[i].Pmax <=extra[i=1:ngen]<=0.05*gen[i].Pmax)
@variable(opfmodel, gens[i].Qmin <= Qg[i=1:ngen] <= gens[i].Qmax)
@variable(opfmodel, buses[i].Vmin <= Vm[i=1:nbus] <= buses[i].Vmax)
@variable(opfmodel,Va[1:nbus])
#fix the voltage angle at the reference bus
setlowerbound(Va[opfdata.bus_ref], buses[opfdata.bus_ref].Va)
setupperbound(Va[opfdata.bus_ref], buses[opfdata.bus_ref].Va)
#objective function
@NLobjective(opfmodel, Min, (1/(nscen+1))*
          sum{gens[i].coeff[gens[i].n-2]*(baseMVA*(Pg[i] + extra[i]))^2
         +gens[i].coeff[gens[i].n-1]*(baseMVA*(Pg[i]+extra[i]))
         +gens[i].coeff[gens[i].n ], i=1:ngen})
 # generator min and max output
@constraint(opfmodel, mmo[i=1:ngen], gens[i].Pmin <= Pg[i]+extra[i] <=
gens[i].Pmax)
```

SC-ACOPF Model – power flow balance

```
# power flow balance
for b in 1:nbus
#real part
@NLconstraint( opfmodel,
   (sum{ YffR[l], l in FromLines[b]} + sum{ YttR[l], l in ToLines[b]} + YshR[b] )*Vm[b]^2
   +sum{ Vm[b]*Vm[busIdx[lines[l].to]] *( YftR[l]*cos(Va[b]-Va[busIdx[lines[l].to]] )
                 + YftI[l]*sin(Va[b]-Va[busIdx[lines[l].to]] )), l in FromLines[b] }
   +sum{ Vm[b]*Vm[busIdx[lines[l].from]]*( YtfR[l]*cos(Va[b]-Va[busIdx[lines[l].from]])
                 + YtfI[l]*sin(Va[b]-Va[busIdx[lines[l].from]])), l in ToLines[b] }
   -(sum{baseMVA*(Pg[g]+extra[g]), g in BusGeners[b]} - buses[b].Pd ) / baseMVA
   ==0)
#imaginary part
@NLconstraint( opfmodel,
   (sum{-YffI[l], l in FromLines[b]} + sum{-YttI[l], l in ToLines[b]} - YshI[b] )*Vm[b]^2
   +sum{ Vm[b]*Vm[busIdx[lines[l].to]] *(-YftI[l]*cos(Va[b]-Va[busIdx[lines[l].to]] )
                 + YftR[l]*sin(Va[b]-Va[busIdx[lines[l].to]] )), l in FromLines[b] }
   +sum{ Vm[b]*Vm[busIdx[lines[l].from]]*(-YtfI[l]*cos(Va[b]-Va[busIdx[lines[l].from]])
                 + YtfR[l]*sin(Va[b]-Va[busIdx[lines[l].from]])), l in ToLines[b] }
   -(sum{baseMVA*Qg[g], g in BusGeners[b]} - buses[b].Qd ) / baseMVA
   ==0)
end
```

SC-ACOPF Model – branch flow limits

```
# branch/lines flow limits
nlinelim=0
for l in 1:nline
  if lines[l].rateA!=0 && lines[l].rateA<1.0e10
   nlinelim += 1
   flowmax=(lines[l].rateA/baseMVA)^2
   Yff_abs2=YffR[l]^2+YffI[l]^2; Yft_abs2=YftR[l]^2+YftI[l]^2
   Yre=YffR[l]*YftR[l]+YffI[l]*YftI[l]; Yim=-YffR[l]*YftI[l]+YffI[l]*YftR[l]
   @NLconstraint(opfmodel,
       Vm[busIdx[lines[l].from]]^2 *
          (Yff_abs2*Vm[busIdx[lines[l].from]]^2 + Yft_abs2*Vm[busIdx[lines[l].to]]^2
           +2*Vm[busIdx[lines[l].from]]*Vm[busIdx[lines[l].to]]*
                ( Yre*cos(Va[busIdx[lines[l].from]]-Va[busIdx[lines[l].to]])
                 -Yim*sin(Va[busIdx[lines[l].from]]-Va[busIdx[lines[l].to]]))
          )
        - flowmax <=0)

   Ytf_abs2=YtfR[l]^2+YtfI[l]^2;
   Ytt_abs2=YttR[l]^2+YttI[l]^2 Yre=YtfR[l]*YttR[l]+YtfI[l]*YttI[l];
   Yim=-YtfR[l]*YttI[l]+YtfI[l]*YttR[l]
   @NLconstraint( opfmodel,
       Vm[busIdx[lines[l].to]]^2 *
           (Ytf_abs2*Vm[busIdx[lines[l].from]]^2 + Ytt_abs2*Vm[busIdx[lines[l].to]]^2
            +2*Vm[busIdx[lines[l].from]]*Vm[busIdx[lines[l].to]]*
                 ( Yre*cos(Va[busIdx[lines[l].from]]-Va[busIdx[lines[l].to]])
                  -Yim*sin(Va[busIdx[lines[l].from]]-Va[busIdx[lines[l].to]]))
           )
       - flowmax <=0)
     end
end
```

SC-ACOPF Model – security constrained part

```
for cont_num in getLocalChildrenIds(opfmodel)
  #set-up the system with on line off
  opfdata_cont = scopf_loaddata(opfdata, sd.lines_off[cont_num])
  ...
  #compute second stage system parameters
  ...
  opfmodel_cont = StructuredModel(parent=opfmodel,id=cont_num)
  #variables and constraints for the system with the line off
  ...
  #power flow constraints as on the previous slides
  ...
  #line limits constraints as on the previous slides
  ...
end
```

- getLocalChildrenIds(opfmodel) returns a list of scenario ids assigned on the local MPI processes/rank
- The SC-OPF model is a collection of AC-OPF (sub)models

Parallel Computational Paradigm

- Each process has a subset of scenarios (ie. getLocalChildrenIds)
- Objective and constraints, derivatives are computed for the local scenarios only.
  - MPI-based implementation.
- This matches solver distribution of the scenarios across nodes.

- No parallel computing knowledge is needed from the user
  - Everything is implemented under the hood
- Solving with PIPS-NLP in parallel
  - **mpiexec –np 48 julia scopf.jl**

Model data

- MatPower ACOPF test cases used as the base for the SC-ACOPF problems

- Dataloader in Julia loads directly “.m” MatPower files
  - Other ACOPF formats can also be supported.
  - No change of model is required

Generating SC-ACOPF problems

- SC problems are setup for **line contingencies** and target N-1 SC-ACOPF

- Can solve all the AC-OPF problems (including the 9,241-bus pegase system)

- Difficult to find a list of contingencies for which Matpower systems are feasible
- Even with a corrective formulation (allows for contingency-related redispatch)
  - References: Monticelli et al., Capitanescu et al.
- None of the Matpower cases are feasible under N-1 line contingency

- For the 300 bus system we found about 50 lines by trial-and-error for which the SC-ACOPF model remains feasible

Preliminary Results – 300-bus system with 48 contingencies on up to 48 nodes on "Blues" cluster @ANL

#procs	Model initiation (seconds)	Structure building (seconds)	Function & derivative evaluation (seconds)	Total time (seconds)
1	7.56	14.02	1002.90	1367.55
2	6.44	7.26	468.77	770.80
4	5.82	4.22	285.36	604.64
8	5.65	2.65	136.76	407.40
16	7.72	1.91	69.79	329.95
24	8.48	1.58	51.21	315.53
48	6.06	1.32	28.96	216.85

300 buses, 411 lines, 69 generators

Future work

- Further code optimization
- Support for linking constraints (already supported by PIPS)

- Realistic, larger power systems

- Support for dynamics (*e.g.*, transient stability)
  - rapid and scalable specification of transient constraints
  - streamlined integration with state-of-the-art time integrators and optimization solvers