SAND2022-16044C

Sandia National Laboratories

Parallel Solution of Optimal Gas Network Control Under Uncertainty

Michael Bynum, Larry Biegler, Carl Laird, Sakshi Naik, Robert Parker, and John Siirola

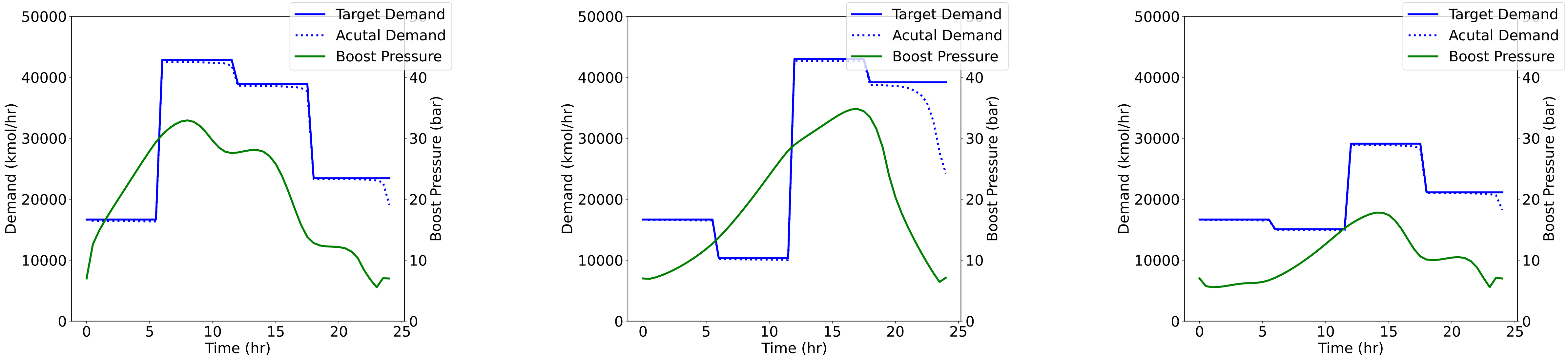
U.S. DEPARTMENT OF ENERGY NNSA National Nuclear Security Administration

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Motivation

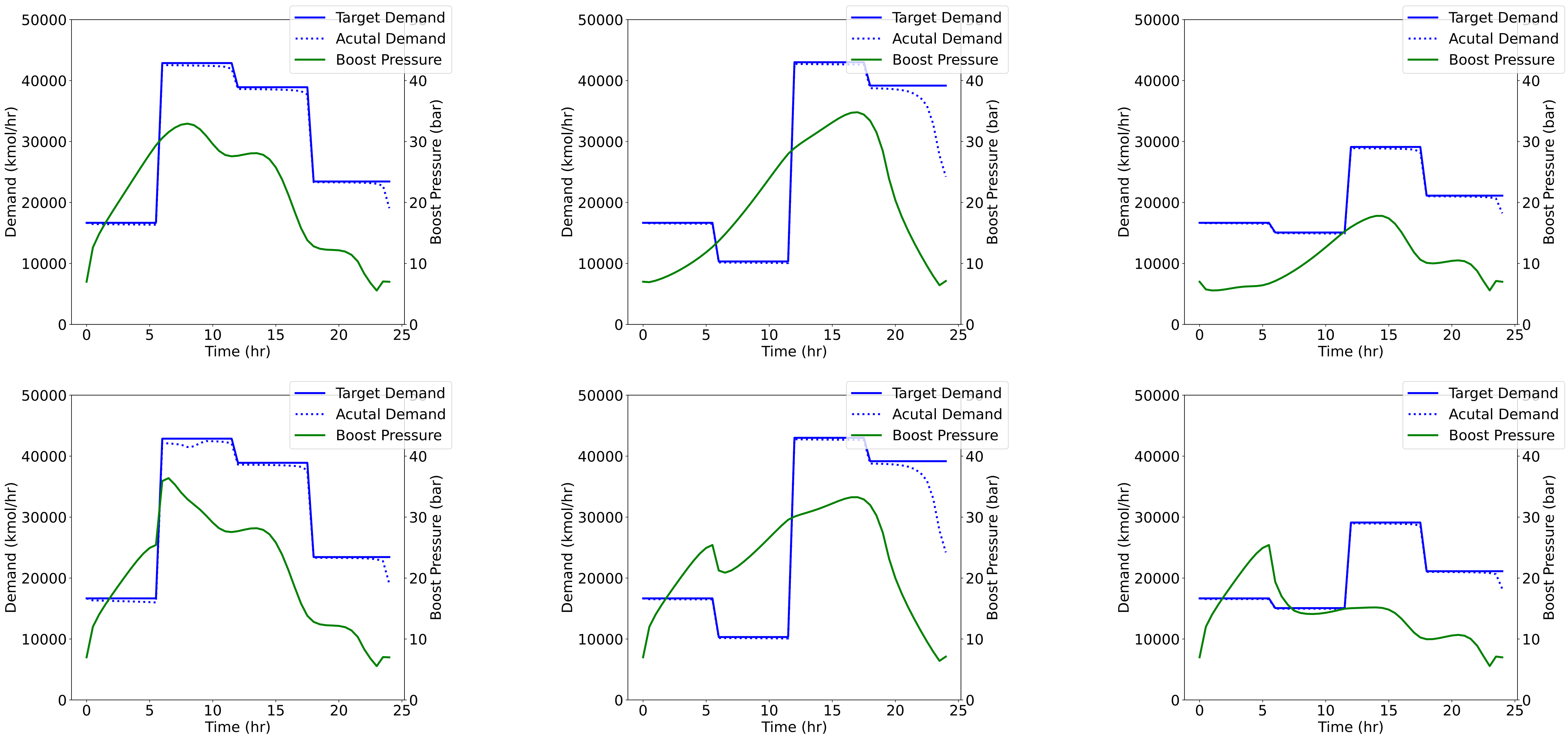
Goal: Minimize compression costs of a natural gas distribution system while satisfying demands

Optimal Compressor Boost Pressure



Optimal compressor boost pressures depend heavily on demands.

A Stochastic Programming Solution

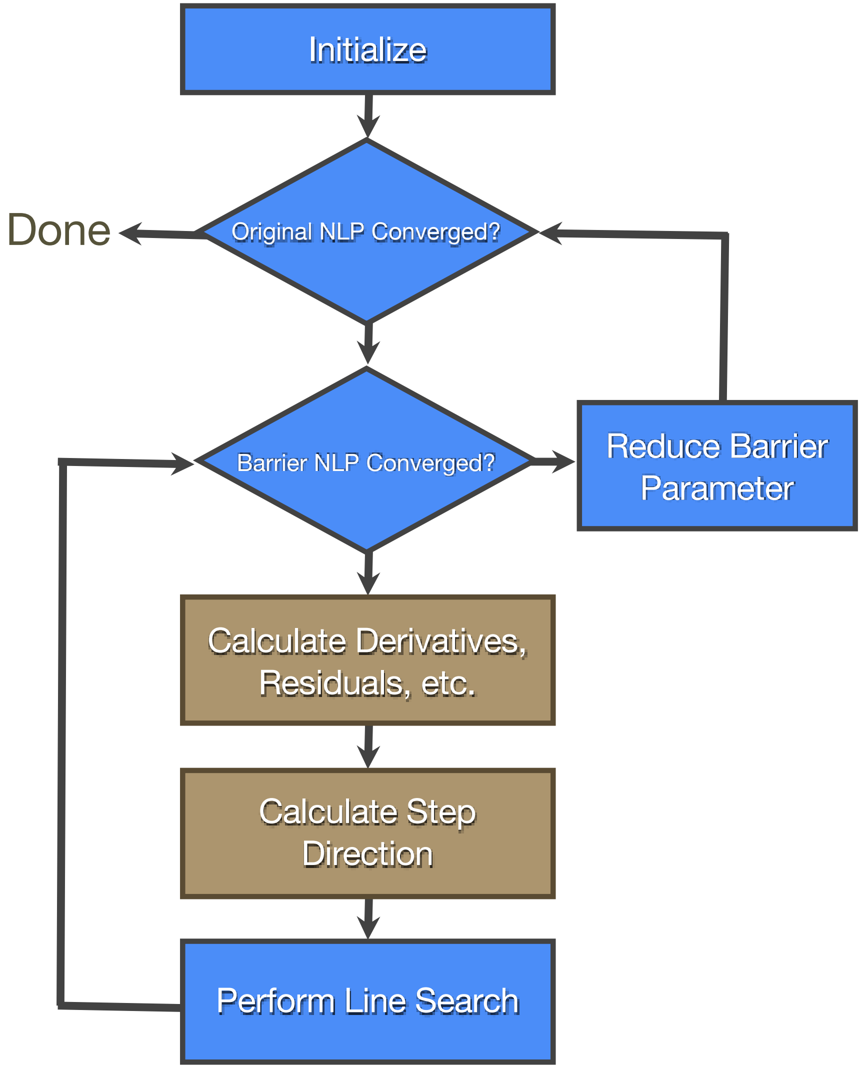


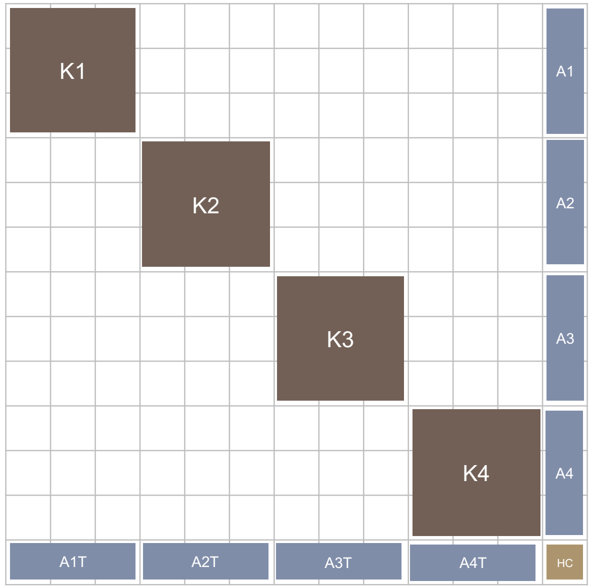
Zavala, V. M. (2014). Stochastic optimal control model for natural gas networks. Computers & Chemical Engineering, 64, 103-113.

But…

A 2-node network with 32 scenarios takes 1.7 minutes to solve with Ipopt

Turn to Parallel Computing





Parallel approaches have demonstrated tremendous success on many problems when tailored to the problem class

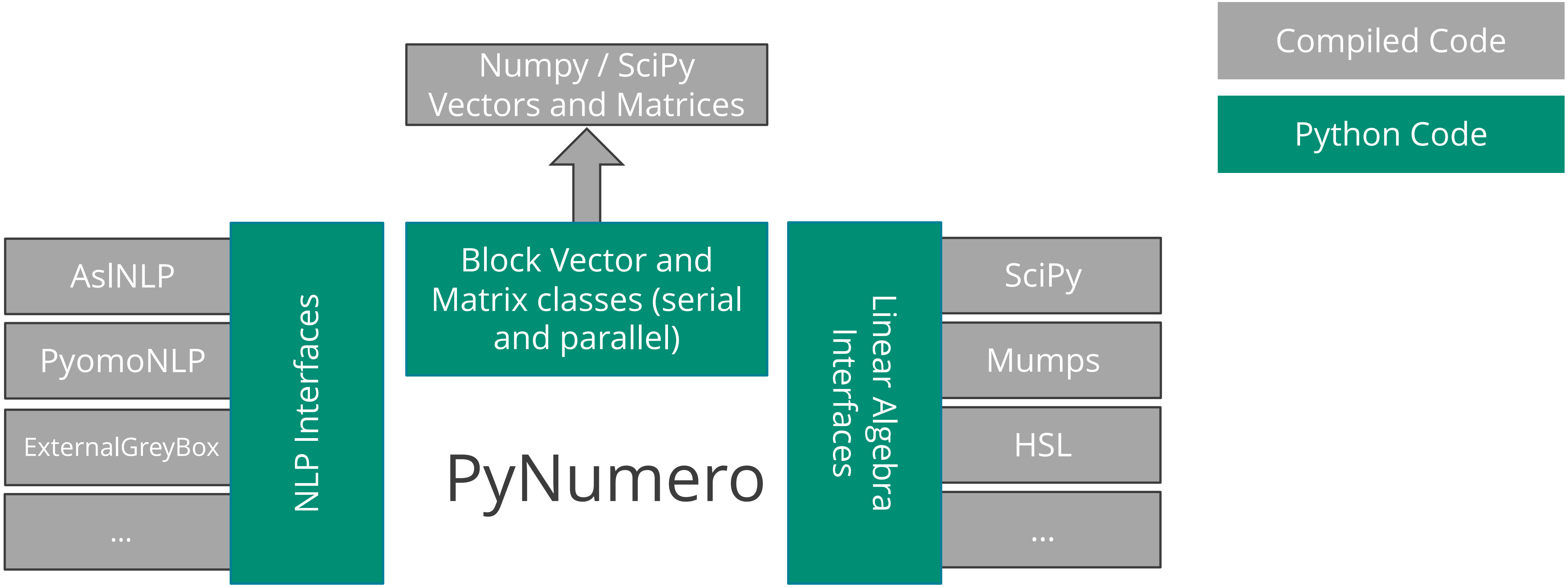
Need software frameworks to support rapid innovation in serial and parallel algorithm development

- Rapid development in high-level languages
- Support for block-based representations
- Strong serial and parallel computational performance

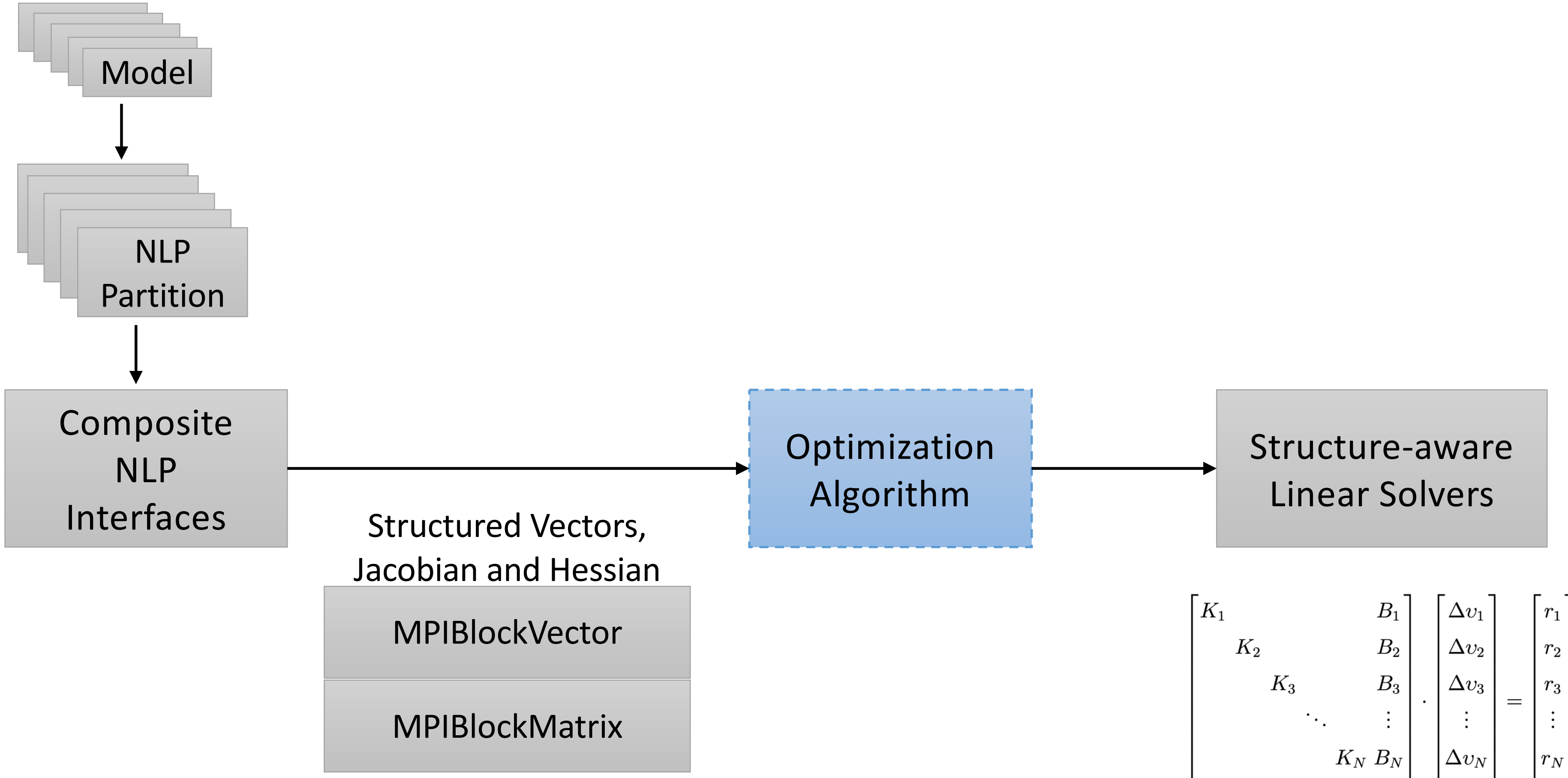
Zavala, V. M., Laird, C. D., & Biegler, L. T. (2008). Interior-point decomposition approaches for parallel solution of large-scale nonlinear parameter estimation problems. *Chemical Engineering Science*, *63*(19), 4834-4845.

PyNumero Overview

PyNumero: A high-level Python framework for rapid development of truly performant serial and parallel optimization algorithms on distributed HPC

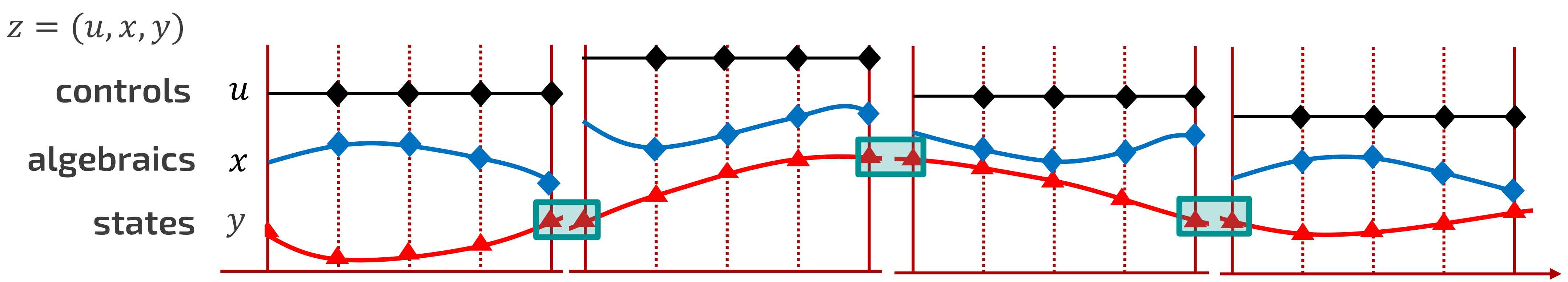


Parapint Overview



$$\begin{bmatrix} K_1 & & & & & B_1 \\ & K_2 & & & & B_2 \\ & & K_3 & & & B_3 \\ & & & \ddots & & \vdots \\ & & & & K_N & B_N \\ B_1^T & B_2^T & B_3^T & \cdots & B_N^T & Q \end{bmatrix} \cdot \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \Delta v_3 \\ \vdots \\ \Delta v_N \\ \Delta d \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_N \\ r_d \end{bmatrix}$$

Originally Developed for Dynamic Optimization



After full-space discretization:

$$\min_z \quad f(z)$$
$$\text{s.t.} \quad c(z,q) = 0$$
$$\underline{z} \le z \le \overline{z}$$

$$c(z,q) = \begin{bmatrix} \overline{G}z_1 - x_0 \\ R(z_1) \\ \underline{G}z_1 + q_1 \\ \overline{G}z_2 - q_1 \\ R(z_2) \\ \underline{G}z_2 + q_2 \\ \vdots \\ \overline{G}z_{n_e} - q_{n_e-1} \\ R(z_{n_e}) \end{bmatrix}.$$

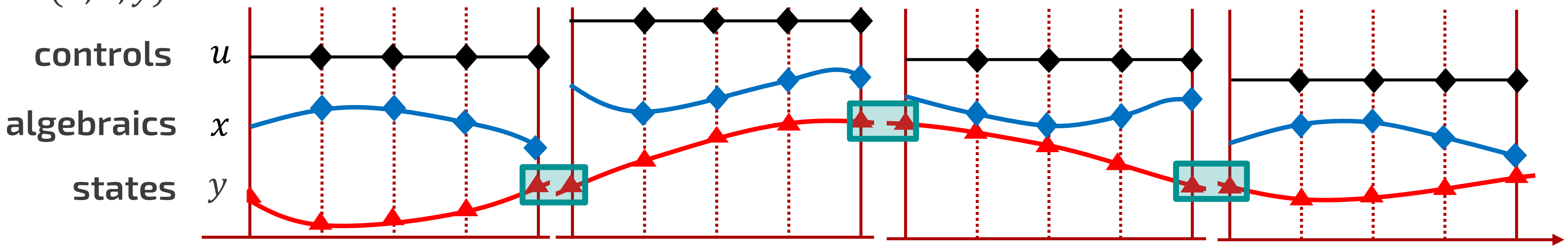
Collocation equations

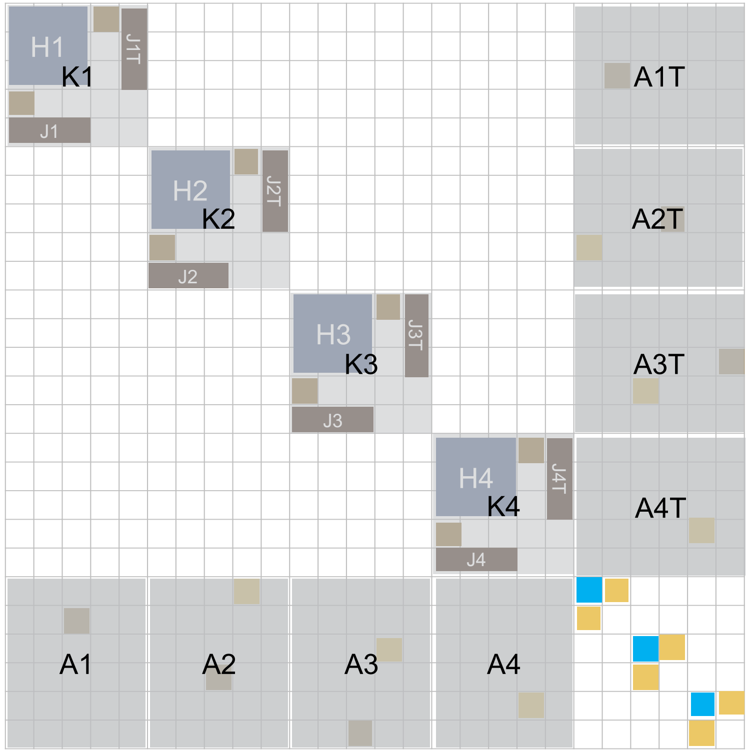
Linking constraints

[Adapted from L.T. Biegler (2007)]

Schur Complements for PinT Optimization

$z = (u, x, y)$





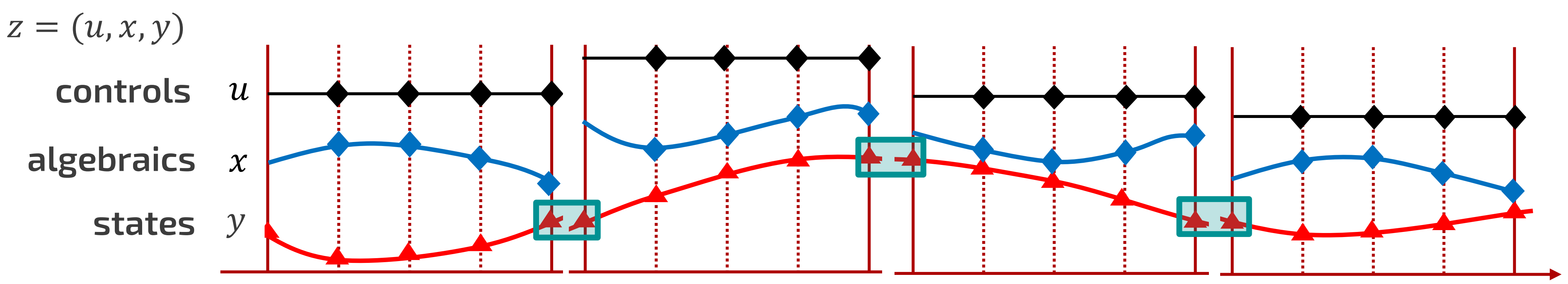
$$c(z, q) = \begin{bmatrix} \overline{G} z_1 - x_0 \\ R(z_1) \\ \underline{G} z_1 + q_1 \\ \overline{G} z_2 - q_1 \\ R(z_2) \\ \underline{G} z_2 + q_2 \\ \vdots \\ \overline{G} z_{n_e} - q_{n_e - 1} \\ R(z_{n_e}) \end{bmatrix}.$$

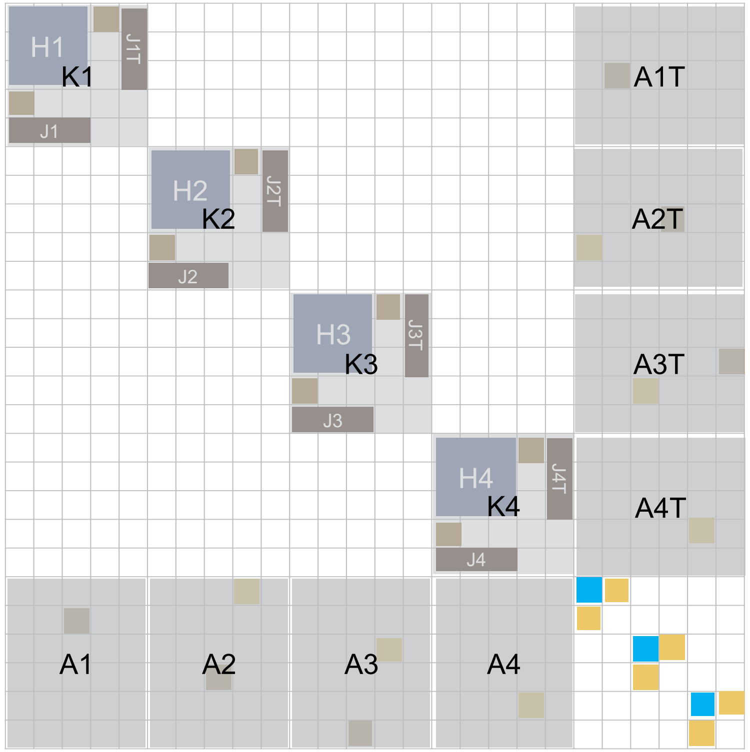
Collocation equations

Linking constraints

[Adapted from L.T. Biegler (2007)]

Schur Complements for PinT Optimization



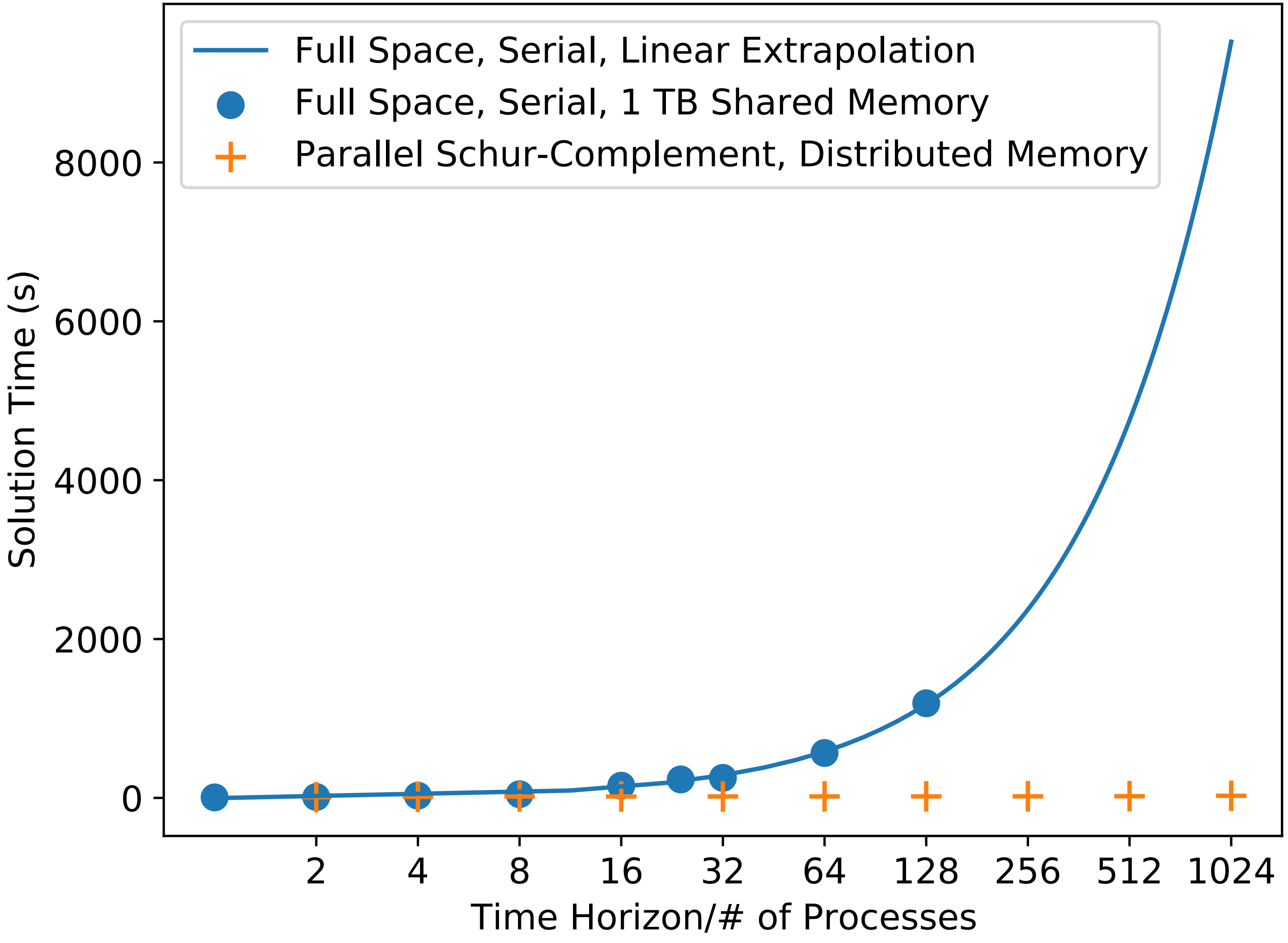


- Factor all K_i matrices ($\downarrow$)
- Form Schur-complement ($\approx$ or $\downarrow$)
  - Backsolve of each K_i for each nonzero column in A_i to form local Schur-complement S_i
- Compute $S = \sum_i S_i$ (MPI Reduction) ($\uparrow$ communication)
- Solve $S\Delta q = r_s$ ($\uparrow$)
- Solve $K_i \Delta z_i = r_i$ ($\downarrow$)

Based on **Kang, Cao, Word and Laird** (2014).

[Adapted from L.T. Biegler (2007)]

Scalability



Approximately 360x speedup on 1024 cores!

A Note on Communication

	Dynamic Optimization	Stochastic Optimization	Multi-stage stochastic optimization	Network-Based Decomposition
Schur-Complement Structure	Sparse	Dense	Sparse	Sparse

Analyze the structure once during symbolic factorization, and exploit it for performant communication during numeric factorization.

BlockVector/Matrix, MPIBlockVector/Matrix

$$\begin{bmatrix} W_k + \Sigma_k + \delta_w I & \nabla c(x_k) \\ \nabla c(x_k)^T & -\delta_c I \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix}$$

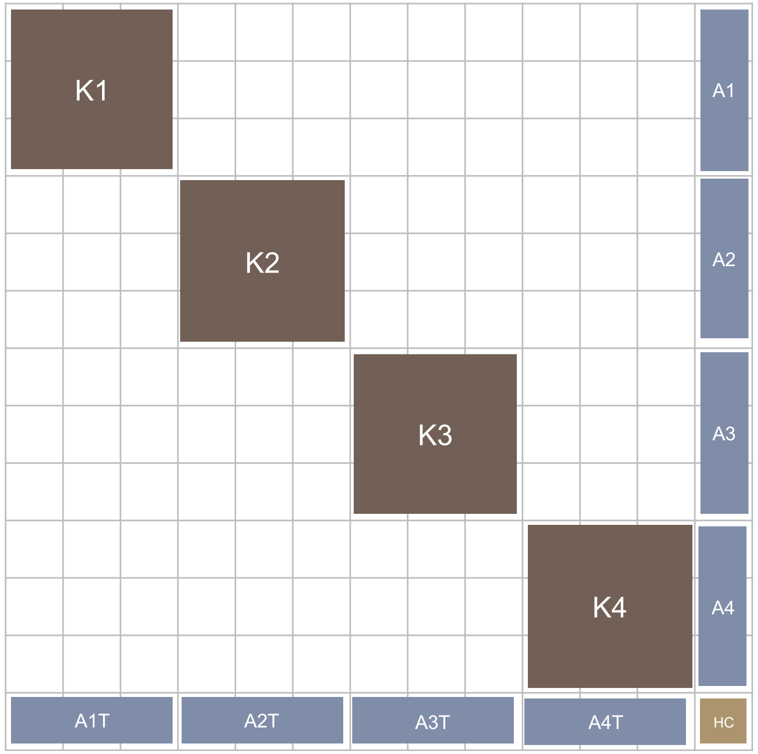
$$Kv$$

Efficient block structures are core to PyNumero

- Many algorithms intuitively represented with blocks
- Critical for parallel, block decomposition approaches

PyNumero BlockVector and BlockMatrix classes

- Support creation of block-based vectors and matrices without data duplication
- BlockVector and BlockMatrix store C++ pointers to underlying data structures
- Convenient for performing block operations (e.g., formation and solution of KKT system)
- Support (almost all) standard operations for Numpy vectors and matrices
- Full support to build and interrogate these structures in both serial and parallel



What does the code look like…

Working with matrices and vectors looks just like Numpy, and (almost all) Numpy / SciPy operations are supported – including solvers

```
# ...
# working with matrices and vectors
y = A.dot(x)          # dot product
z = x + y             # addition
zm = abs(z).max()     # infinity norm

# this code looks the same even if these are
# a dense numpy vector or Pynumero BlockVector
```

```
# ...
# working with nlp interfaces
nlp.set_primals(x)
nlp.set_duals(lam)

grad_lag = (nlp.evaluate_grad_objective() +
            nlp.evaluate_jacobian().transpose() * lam)

residuals = nlp.evaluate_constraints()

jacobian = nlp.evaluate_jacobian()
hessian_lag = nlp.evaluate_hessian_lag()
```

Interfaces to the NLP classes are clean, easy to use, and easy to integrate with Numpy and SciPy

What does parallel code look like…

BlockVector and BlockMatrix are intuitive and performant – references stored internally

```
# create vector x
x = BlockVector(3)
x.set_block(0, np.random.normal(size=3))
x.set_block(1, np.random.normal(size=3))
x.set_block(2, np.random.normal(size=3))

# create vector y
y = BlockVector(3)
y.set_block(0, np.random.normal(size=3))
y.set_block(1, np.random.normal(size=3))
y.set_block(3, np.random.normal(size=3))

# perform operations
z1 = x + y              # add x and y
z2 = x.dot(y)           # dot product
z3 = np.abs(x).max()    # infinity norm
```

```
# get communicator and process #
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# specify the parallel ownership
owners = [2, 0, 1]

# create vector x (each processor does local block)
x = MPIBlockVector(3, rank_owner=owners, mpi_comm=comm)
x.set_block(owners.index(rank), np.random.normal(size=3))

# create vector y (each processor does local block)
y = MPIBlockVector(3, rank_owner=owners, mpi_comm=comm)
y.set_block(owners.index(rank), np.random.normal(size=3))

# perform parallel operations
z1 = x + y              # add x and y
z2 = x.dot(y)           # dot product
z3 = np.abs(x).max()    # infinity norm
```

Parallel code can be written at a high level with mpi4py and the PyNumero MPIBlockVector and MPIBlockMatrix classes

Writing Algorithms: Equality-Constrained SQP Example

```python
def sqp(nlp, max_iter=100, tol=1e-8):
    # setup KKT matrix
    kkt = BlockMatrix(2, 2)
    rhs = BlockVector(2)

    # create and initialize the iteration vector
    x = BlockVector(2)
    x.set_block(0, nlp.init_primals().copy())
    x.set_block(1, nlp.init_duals().copy())

    # create the linear solver
    linear_solver = MA27Interface()
    linear_solver.set_cntl(1, 1e-6) # pivot tolerance

    # main iteration loop
    for _iter in range(max_iter):
        nlp.set_primals(x.get_block(0))
        nlp.set_duals(x.get_block(1))

        grad_lag = (nlp.evaluate_grad_objective() +
                    nlp.evaluate_jacobian().transpose() * x.get_block(1))
        residuals = nlp.evaluate_constraints()

        if np.abs(grad_lag).max() <= tol and np.abs(residuals).max() <= tol:
            break

        kkt.set_block(0, 0, nlp.evaluate_hessian_lag())
        kkt.set_block(1, 0, nlp.evaluate_jacobian())
        kkt.set_block(0, 1, nlp.evaluate_jacobian().transpose())

        rhs.set_block(0, grad_lag)
        rhs.set_block(1, residuals)

        linear_solver.do_symbolic_factorization(kkt)
        linear_solver.do_numeric_factorization(kkt)
        delta = linear_solver.do_back_solve(-rhs)
        x += delta
```

Equality-constrained SQP method

- Makes use of BlockVector / BlockMatrix to form KKT, RHS, and iteration vector
- BlockMatrix sent directly to the linear solver
- BlockMatrix efficiently stores pointers to sub-blocks

What about performance?

```python
def sqp(nlp, max_iter=100, tol=1e-8):
    # setup KKT matrix
    kkt = BlockMatrix(2, 2)
    rhs = BlockVector(2)

    # create and initialize the iteration vector
    x = BlockVector(2)
    x.set_block(0, nlp.init_primals().copy())
    x.set_block(1, nlp.init_duals().copy())

    # create the linear solver
    linear_solver = MA27Interface()
    linear_solver.set_cntl(1, 1e-6) # pivot tolerance

    # main iteration loop
    for _iter in range(max_iter):
        nlp.set_primals(x.get_block(0))
        nlp.set_duals(x.get_block(1))

        grad_lag = (nlp.evaluate_grad_objective() +
                    nlp.evaluate_jacobian().transpose() * x.get_block(1))
        residuals = nlp.evaluate_constraints()

        if np.abs(grad_lag).max() <= tol and np.abs(residuals).max() <= tol:
            break

        kkt.set_block(0, 0, nlp.evaluate_hessian_lag())
        kkt.set_block(1, 0, nlp.evaluate_jacobian())
        kkt.set_block(0, 1, nlp.evaluate_jacobian().transpose())

        rhs.set_block(0, grad_lag)
        rhs.set_block(1, residuals)

        linear_solver.do_symbolic_factorization(kkt)
        linear_solver.do_numeric_factorization(kkt)
        delta = linear_solver.do_back_solve(-rhs)
        x += delta
```

Equality-constrained SQP method

- Makes use of BlockVector / BlockMatrix to form KKT, RHS, and iteration vector
- BlockMatrix sent directly to the linear solver
- BlockMatrix efficiently stores pointers to sub-blocks

Performance compared with IPOPT (fully compiled C++)

- Discretized PDE control problem (Burgers)
- PyNumero ESQP within 15% of IPOPT on reasonable problems (100,000 vars, ~2 sec)
- PyNumero directly comparable to compiled C++ when the problem is sufficiently large.

With Parapint...

A 2-node network with 32 scenarios takes 6.7 seconds to solve with Parapint and 16 processes

Remarks and Future Work

Remarks:

- Schur-Complement decomposition provides a scalable solution approach for stochastic optimal control of gas pipeline networks
- PyNumero provides a flexible, high-level Python framework for creating nonlinear optimization solvers
- PyNumero designed to facilitate rapid innovation in parallel algorithm development
- Full support for block representations of linear algebra
- Intuitive interface for NLP and algorithm creation with support for Numpy and SciPy
- Interfaces to commonly used linear solvers for NLP algorithms (SciPy, Mumps, HSL)
- Computationally expensive operations performed with compiled code
- Great scalability to over 1000 cores!

Future work

- Improved robustness of Parapint’s interior point algorithm
- More algorithms!
- Combined scenario and time decomposition for stochastic optimal control via nested Schur-Complement

Acknowledgements

This work was conducted as part of the Institute for the Design of Advanced Energy Systems (IDAES) with funding from the Office of Fossil Energy, Cross-Cutting Research, U.S. Department of Energy.

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.