Scalable Parallel Nonlinear Optimization with PyNumero and Parapint

Jose S. Rodriguez[1], Robert Parker[2], Carl D. Laird[2], Bethany Nicholson[3], John D. Siirola[3], and Michael Bynum[3]

[1] Davidson School of Chemical Engineering, Purdue University, West Lafayette, IN, 47907 , Email: santiagoropb@gmail.com
[2] Carnegie Mellon University, Department of Chemical Engineering, Pittsburgh, PA 15213 , Email: robbybparker@gmail.com, claird@andrew.cmu.edu
[3] Center for Computing Research, Sandia National Laboratories, Albuquerque NM 87185 , Email: blnicho@sandia.gov, jdsiiro@sandia.gov, mlbynum@sandia.gov

September 22, 2021

Abstract

We describe PyNumero, an open-source, object-oriented programming framework in Python that supports rapid development of performant parallel algorithms for structured nonlinear programming problems (NLP's) using the Message Passing Interface (MPI). PyNumero provides three fundamental building blocks for developing NLP algorithms: a fast interface for calculating first and second derivatives with the AMPL Solver Library (ASL), a number of interfaces to efficient linear solvers, and block-structured vectors and matrices based on NumPy, SciPy, and MPI that support distributed parallel storage and computation. PyNumero's design enables efficient, parallel algorithm development using high-level Python syntax while keeping expensive numerical calculations in fast, compiled implementations based on languages like C and Fortran. To demonstrate the utility of PyNumero, we also present Parapint, a Python package built on PyNumero for parallel solution of dynamic optimization problems. Parapint includes a parallel interior-point solver based on Schur-Complement decomposition. We illustrate the effectiveness of PyNumero for developing parallel algorithms with both code examples and scalability analyses for parallel matrix-vector dot products, parallel solution of structured systems of linear equations using Schur-Complement decomposition, and the parallel solution of a 2-dimensional PDE optimal control problem. Our numerical results show nearly perfect scaling to over 1000 cores for large matrix-vector dot products and structured linear systems. Moreover, we obtain over 360 times speedup for the optimal control example.

Contents

1 Introduction

Recent needs for efficient solution of large-scale, structured nonlinear optimization problems have led to the development of tailored solution algorithms for exploiting problem structure. Special structure arises in many applications, including dynamic optimization, stochastic programming, infrastructure applications with natural network structure (e.g., power transmission systems), parameter estimation, and many others. All of these applications have characteristics that result in large-scale optimization problems. For example, dynamic optimization problems can become quite large due to discretization of time and space in differential equations. Adequate sampling of large uncertainty spaces can also result in large-scale stochastic programming problems with many scenarios.

Although these applications often involve the solution of very large nonlinear programs (NLPs), many of the problems have an inherent structure that can be exploited using decomposition and even parallel solution algorithms. This paper discusses the packages PyNumero and Parapint, which are designed to support rapid development of performant parallel algorithms for structured NLP problems. To frame the discussion of these tools, we will focus on an example formulation addressed by algorithms within Parapint.

Many of the optimization problems discussed above can be structurally partitioned as in Problem (1).

$$\min_{x_s,d} \sum_{s\in\mathcal{S}} f_s(x_s) \tag{1a}$$

s.t.

$$c_s(x_s) = 0 \qquad \forall s \in \mathcal{S} \tag{1b}$$

$$\underline{x_s} \leq x_s \leq \overline{x_s} \qquad \forall s \in \mathcal{S} \tag{1c}$$

$$P_s x_s - P_s^d d = 0 \qquad \forall s \in \mathcal{S} \tag{1d}$$

Here, the set $\mathcal{S}$ denotes a set of partitions. These partitions could be formed by finite elements in dynamic optimization, scenarios in stochastic programming, or data sets in parameter estimation. The variables $x_s \in \mathbb{R}^{N_s}$ are only involved in the constraints associated with partition s ($c_s \in \mathbb{R}^{M_s}$). However, there is often a set of coupling variables, $d \in \mathbb{R}^D$, that link two or more of the partitions, enforced by Equation (1d). Here, $P_s \in \mathbb{R}^{C_s \times N_s}$, $P_s^d \in \mathbb{R}^{C_s \times D}$, and C_s is the number of coupling variables in partition s. For example, the coupling variables in two-stage stochastic programming are the first stage variables, whereas the coupling variables in dynamic optimization are the differential variables at finite element boundaries.

Many NLP algorithms have been developed to exploit the structure of Problem (1) (Gondzio and Grothey 2009, Chiang et al. 2014, Shin et al. 2020a). Nonlinear interior-point methods have been a popular choice with algorithms that exploit the structure in Problem (1) by parallelizing the solution of the KKT system at every iteration of the NLP algorithm. It is important to notice that the solution of the KKT system comprises the majority of the computational effort of each iteration of interior-point algorithms. For this reason a significant amount of research has focused on accelerating this step. Schur-Complement decomposition has been used for parallel solution of parameter estimation problems, stochastic programming problems, and dynamic optimization problems (Zavala et al. 2008, Kang et al. 2014, Petra et al. 2014, Word et al. 2014). Zavala et al. (2008) and Word et al. (2014) form the Schur-Complement explicitly using repeated backsolves with a single factorization of each diagonal block in the KKT system. Petra et al. (2014)

also forms the Schur-Complement explicitly, but with a custom factorization routine that produces the Schur-Complement as a by-product of the factorization of each of the diagonal blocks in the KKT system. On the other hand, Kang et al. (2014) avoids formation of the Schur-Complement entirely with a preconditioned conjugate gradient method. Other algorithms have been utilized for exploiting the structure of interior-point KKT systems, including Cyclic Reduction (Wan et al. 2019) and overlapping Schwarz (Shin et al. 2020a). Alternatives also include decomposition approaches that solve sequences of NLP subproblems to exploit the structure of Problem (1). For example, the alternating-direction method of multipliers (ADMM) and Progressive Hedging (PH) (Eckstein and Bertsekas 1992, Rodriguez et al. 2018, Rockafellar and Wets 1991, Word et al. 2012) have both been used for structured NLPs. More recently, Rodriguez et al. (2020) proposed a hybrid approach with an ADMM-based preconditioner for iterative solution of structured KKT systems.

Several of these algorithms have been implemented in existing software. OOPS (the Object-Oriented Parallel Solver) and PIPS-NLP are both parallel interior-point solvers written in C++ that utilize Schur-Complement decomposition for parallel solution of the KKT system (Gondzio and Grothey 2009, Chiang et al. 2014). MadNLP is a Julia package that contains a parallel interior-point solver and has been utilized with an iterative method which uses a Restrictive-Additive Schwarz (RAS) preconditioner (Shin et al. 2020b). The RAS method was tested using a multi-threaded implementation. ParNMPC is a Matlab-based package for parallel nonlinear model-predictive control and uses C and C++ code generation for parallelization with OpenMP (Deng and Ohtsuka 2019).

While the above solvers all demonstrate impressive performance improvements over serial algorithms, developing new, distributed memory algorithms is a challenging and time-consuming task. The vast majority of both serial and parallel NLP solvers are implemented in low-level languages such as C, and they are difficult to modify or extend. Significant software engineering expertise is required to prototype even minor modifications to existing solvers. This impedes the exploration, development, and testing of new ideas slowing research progress in this area.

To address and mitigate these challenges, we present PyNumero, a Python package for numerical optimization that provides a high-level programming framework for rapidly developing efficient, parallel, and scalable solution algorithms for structured NLP's. PyNumero has been designed with computational performance in mind and utilizes Python-C interfaces internally to ensure that expensive numeric calculations are all performed using compiled kernels. While computations performed directly in Python can be slow, efficient algorithms can be developed in Python by utilizing interfaces to low-level languages for computationally intensive tasks, as demonstrated by NumPy, a widely used package for scientific computing (Virtanen et al. 2020a).

PyNumero supports a variety of algorithmic building blocks that allow researchers to quickly explore new parallel algorithms. In particular, PyNumero provides an interface to the AMPL Solver Library (ASL) for automatic differentiation, interfaces to several linear solvers, and parallel implementations of block-structured matrices and vectors. Parallel linear algebra routines are based on the Message Passing Interface (MPI) and can be used on shared or distributed memory machines. The intent is to enable more practitioners and researchers in nonlinear optimization to write numerical algorithms and rapidly implement new ideas in a high-level language with little to no sacrifice in computational performance. Furthermore, PyNumero can be used alongside Pyomo to provide a unified Python platform for both modeling and solving optimization problems. This platform allows PyNumero to directly exploit Pyomo model structure and facilitates rapid implementation of structure-

exploiting optimization algorithms.

We demonstrate the effectiveness and computational performance of PyNumero with code examples and scalability analyses for parallel matrix-vector dot products and parallel solution of structured systems of linear equations using Schur-Complement decomposition. Our results show nearly perfect scaling to over 1000 cores. Moreover, we present Parapint, a Python package built on top of PyNumero for parallel solution of both stochastic and dynamic optimization problems. We present numerical results for a 2-dimensional partial differential equation (PDE)-constrained optimal control problem with over 360 times speedup over a serial interior-point algorithm.

The remainder of this paper is organized as follows. In Section 2, we present an overview of PyNumero, describing the MPI-based block-structured matrices and vectors, interfaces for linear solvers, and interactions with the NLP problem definition. This section closes with a short example implementation of an equality-constrained sequential quadratic programming (SQP) algorithm. In Section 3, we present the Parapint package, describe the Schur-complement decomposition implementation, and illustrate parallel performance on a 2-D PDE optimal control case study. We provide distribution details for PyNumero and Parapint in Section 4. Finally, we summarize our results and provide conclusions and future research directions in Section 5.

2 PyNumero Overview

As shown in Figure 1, PyNumero provides three fundamental components for developing parallel NLP algorithms. First, PyNumero implements block-based vector matrix classes with both serial and parallel distributed implementations. These classes conveniently facilitate development of optimization algorithms and decomposition strategies for structured problems. Second, PyNumero provides interfaces to several linear solvers, including the HSL solvers MA27 and MA57, MUMPS, and any SciPy solver (Duff and Reid 1983, Amestoy et al. 2000, Virtanen et al. 2020b). These linear solvers form the core computational kernel for many nonlinear algorithms. Third, PyNumero provides a set of NLP interfaces for function and derivative evaluations. The current interfaces, including the Pyomo interface (`PyomoNLP`), perform all derivative calculations in C by calling the ASL (Gay 1997) from Python. As Figure 1 illustrates, PyNumero provides a high-level Python API but performs all computationally intensive operations via interfaces to efficient, compiled kernels. In this section, we provide an overview of these components, illustrate the parallel performance of the block-based matrix and vector classes, and show how these building blocks can be used and integrated for NLP algorithm development. More detailed documentation can be found in the online documentation at `https://pyomo.readthedocs.io/`.

2.1 Block Vectors and Matrices

Matrix-vector operations are fundamental for the development of any numerical algorithm. NumPy is a popular Python package that provides functionality to store and manipulate n-dimensional arrays of data, with most of the operations being performed in compiled code. The efficiency and flexibility of NumPy has made the package an essential library for most of today's scientific/mathematical Python-based software. SciPy is another popular Python package for scientific computing that builds on NumPy to provide a collection of common numerical routines (also compiled) and sparse matrix storage schemes. To exploit the capabilities of the NumPy/SciPy ecosystem, PyNumero's vector and matrix classes are built

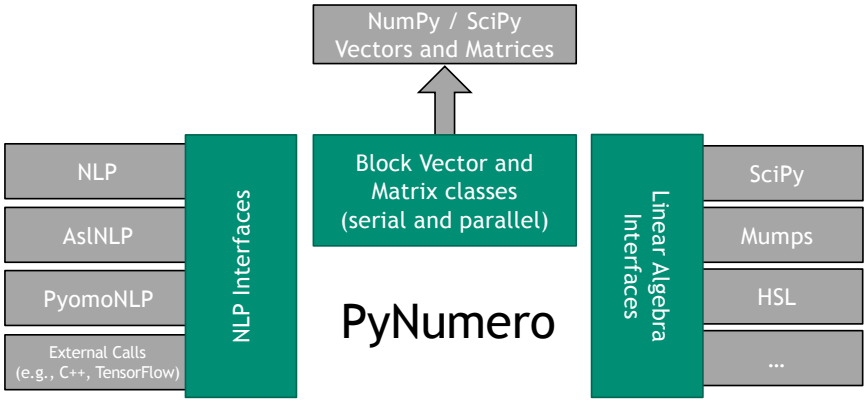

Figure 1: PyNumero building blocks for algorithmic development

on top of NumPy arrays and SciPy sparse matrices. PyNumero therefore benefits from the fast compiled implementations within NumPy (e.g. vectorization and broadcasting), makes all subroutines in NumPy/SciPy available for implementing algorithms, and minimizes the burden on users to learn additional syntax besides what is offered in NumPy/SciPy.

PyNumero extends these implementations and provides classes for working with block-structured matrices and vectors. These classes facilitate optimization algorithm development and support distributed parallel storage, computation, and interrogation. In a KKT system for an equality-constrained NLP, for example, the KKT matrix is composed of the Hessian of the Lagrangian and the Jacobian of the constraints. Additionally, the KKT right-hand-side (rhs) is composed of the gradient of the Lagrangian and the residuals of the constraints. PyNumero supports construction of these composite matrices efficiently without copying the underlying numerical data. These implementations are designed to work seamlessly with NumPy arrays and SciPy sparse matrices.

Figure 2 shows a class diagram for PyNumero's `BlockVector` and `MPIBlockVector`, the serial and parallel implementations of block structured vectors, respectively. As the figure shows, both of these classes inherit from NumPy's `ndarray` class, and their application programming interfaces (APIs) mirror that of NumPy's `ndarray`. This allows algorithm developers to write intuitive code while still exploiting structure for parallel computing.

Most parallel NLP algorithms work by exploiting problem structure. For example, certain classes of problems, such as stochastic programming and dynamic optimization, impose certain structures on the KKT system. An example of the KKT system of a stochastic programming problem is presented in Figure 3 where the plotted points represent non-zero entries in the matrix. To this end, the PyNumero implementations of `BlockVector` and `BlockMatrix` support arbitrary hierarchical representations of numerical data for structured linear algebra operations. The `MPIBlockVector` and `MPIBlockMatrix` extend this functionality to support parallel, distributed data structures where individual blocks of numerical data can be owned by different processes. While the `BlockVector` and `MPIBlockVector` classes inherit from `numpy.ndarray`, they represent an ordered list of sub-vectors or *blocks* that are either additional `BlockVector` objects or NumPy `ndarray` objects. The leaves within these structures are all NumPy `ndarray` objects that support efficient elementary

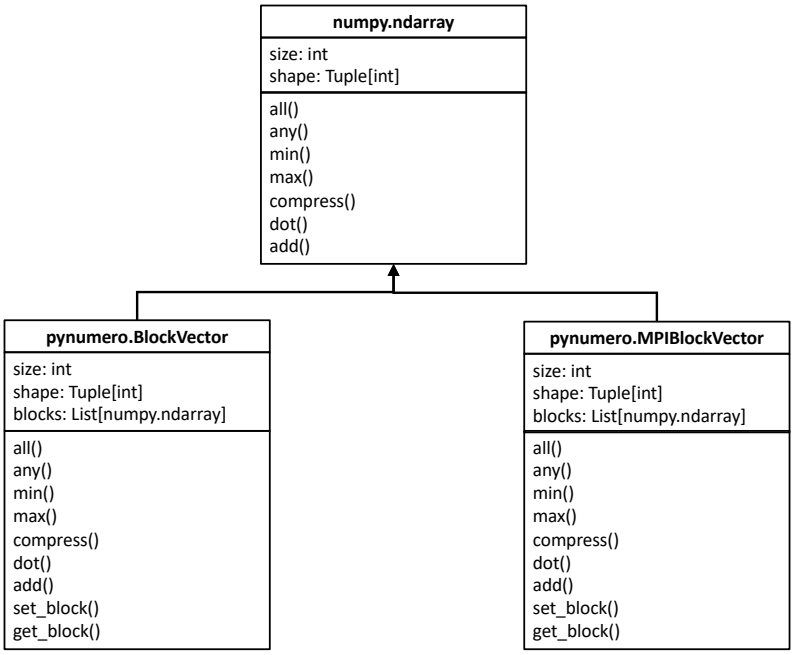

Figure 2: Class diagram for PyNumero's block vectors. The list of attributes and methods is incomplete.

linear algebra operations. This design allows us to utilize an API similar to that of NumPy, while supporting composition through block structures and also utilizing NumPy for efficient computation. The `BlockMatrix` and `MPIBlockMatrix` are similarly structured, containing a set of sub-matrices (i.e., *blocks*) indexed by block row and column. Each of these blocks can either be additional `BlockMatrix` objects or SciPy sparse matrices. As with their vector counterparts, the leaves within these structures are SciPy objects that support efficient computation. This design is important to represent parallel, distributed matrices with arbitrary block structures. In addition to the benefits of distributed data ownership of the parallel versions, simply being able to represent and manipulate the KKT system using its block sub-matrices greatly simplifies the implementation of both general optimization algorithms and tailored decomposition strategies.

We now illustrate the use of block-based vector classes in PyNumero. Listing 1 demonstrates how to perform vector-vector addition with `BlockVector` and how NumPy arrays are utilized. Lines 5-11 create two instances of `BlockVector`: x and y, each with two blocks. Line 14 performs vector-vector addition with these two `BlockVectors`:

$$z_1 = x + y \tag{2}$$

Many standard operations and NumPy methods, in addition to vector-vector addition, are supported, and PyNumero provides the underlying block-based implementations.

The `MPIBlockVector` automatically takes care of parallelization for all of the basic operations needed for algorithm development. The only major difference in the API is in construction. When constructing instances of `MPIBlockVector`, users must specify which blocks are owned by which processes (i.e., the MPI *rank*). Listing 2 demonstrates how to

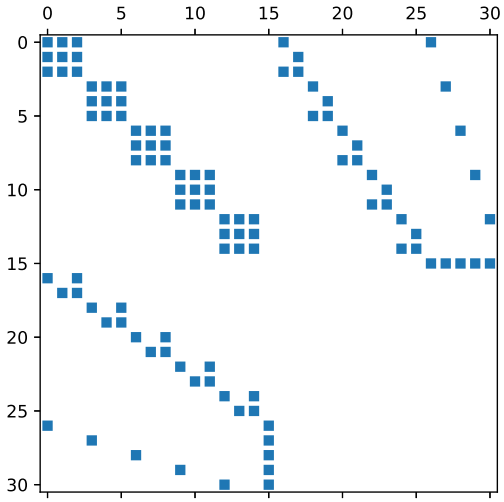

Figure 3: KKT system of a stochastic programming problem

perform vector-vector addition with `MPIBlockVectors`. On line 9, we create an instance of `MPIBlockVector` for x. The `rank_owner` argument is a Python `list` of integers specifying the particular MPI process (identified by the rank) that owns each block index. In this example, block 0 is owned by the process with rank 2, block 1 is owned by the process with rank 0, and block 2 is owned by the process with rank 1. Line 10 creates a random vector of values and assigns it to the correct index for the current MPI rank. Note that each process only needs to create data and specify the blocks for which it is responsible. Lines 12-13 perform a similar construction for y, and line 15 performs the vector-vector addition. Note that this addition operation occurs in parallel. Lines 16-17 show two more example operations with the `MPIBlockVector` objects. `BlockMatrix` and `MPIBlockMatrix` have a class diagram similar to that of `BlockVector` and `MPIBlockVector` with an API similar to that of SciPy sparse matrices.

2.2 Performance of MPI-based block matrices and vectors

In this section, we discuss the performance of PyNumero's parallel linear algebra routines. We present scalability results for a parallel matrix-vector dot product all the way up to 1024 cores. The results in this section demonstrate that modern, high-level languages can be used for performant parallel algorithm development and that PyNumero provides a viable framework for this development.

We perform a weak scaling analysis of PyNumero's parallel matrix-vector dot product using a block-structured matrix with the number of block-columns equal to the number of cores used and the number of block-rows equal to the number of cores used plus one. Each diagonal block and each block in the last row contain non-zeros. All other blocks have no non-zeros. This is a similar structure to the Jacobian of the constraints for a dynamic

```
from pyomo.contrib.pynumero.sparse import BlockVector
import numpy as np


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

y = BlockVector(2)
y.set_block(0, np.random.normal(size=3))
y.set_block(1, np.random.normal(size=3))

# add x and y
z1 = x + y
```

Listing 1: `BlockVector` Addition

```
from pyomo.contrib.pynumero.sparse.mpi_block_vector import MPIBlockVector
import numpy as np
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

owners = [2, 0, 1]
x = MPIBlockVector(3, rank_owner=owners, mpi_comm=comm)
x.set_block(owners.index(rank), np.random.normal(size=3))

y = MPIBlockVector(3, rank_owner=owners, mpi_comm=comm)
y.set_block(owners.index(rank), np.random.normal(size=3))

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

Listing 2: `MPIBlockVector` Addition

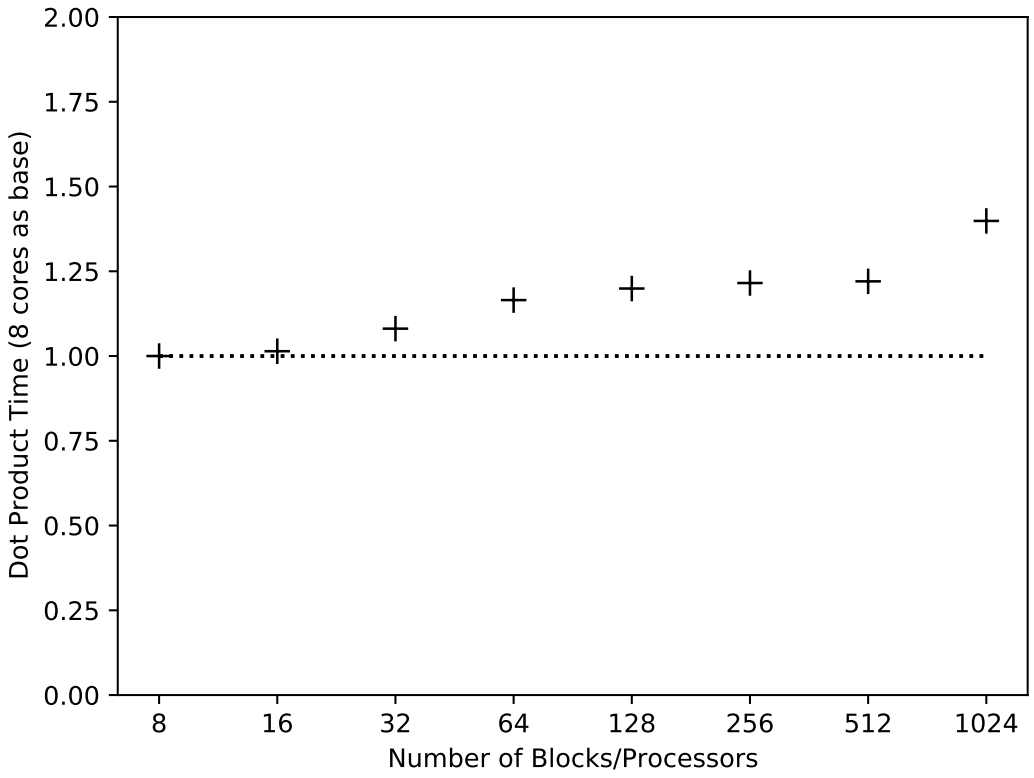

Figure 4: Weak scaling for PyNumero's parallel matrix-vector dot product.

optimization problem with the constraints and variables ordered for Schur-Complement decomposition (Word et al. 2014). Each nonzero block contains a square 100,000 by 100,000 matrix with a sparsity of 0.1% (each nonzero block contained 10 million non-zeros). Figure 4 shows the weak scaling results where the problem size is increased from 8 to 1024 blocks while the cores utilized are simultaneously increased from 8 to 1024. At 1024 cores, this represents a matrix with over 10 billion non-zeros. This is a challenging parallel problem to test scalability since the computational effort required by each process is low and the overall performance on many cores is dominated by communication. The dot product performed with 1024 cores is 128 times larger than that performed with 8 cores. Nevertheless, this distributed matrix-vector product is performed in less than 0.3 seconds and takes only 1.4 times longer than the smaller problem over 8 cores. Furthermore, if the work per processor was larger, we would expect improved scaling results (as illustrated in other examples later). As the figure shows, PyNumero's parallel scalability is very good.

2.3 Linear Solver Interfaces

Efficient implementations of nonlinear optimization algorithms require fast and reliable linear solvers. PyNumero provides access to several libraries for the solution of the sparse linear systems that arise in nonlinear programming. Because PyNumero stores matrices in

```
1  from scipy.sparse import tril
2  from pyomo.contrib.pynumero.linalg.ma27 import MA27Interface
3
4
5  A = get_coo_matrix()
6  rhs = get_rhs()
7  solver = MA27Interface()
8  solver.set_cntl(1, 1e-6)  # set the pivot tolerance
9  A_tril = tril(A)  # extract lower triangular portion of A
10 status = solver.do_symbolic_factorization(dim=5,
11                                           irn=A_tril.row,
12                                           icn=A_tril.col)
13 status = solver.do_numeric_factorization(dim=5,
14                                          irn=A_tril.row,
15                                          icn=A_tril.col,
16                                          entries=A_tril.data)
17 x = solver.do_backsolve(rhs)
```

Listing 3: PyNumero interface to HSL's MA27

NumPy/SciPy objects, subroutines available in these packages can be used when writing algorithms in PyNumero. This includes the SciPy direct and iterative solvers as well as any other Python package based on NumPy such as PyTrilinos (Sala et al. 2008), Petsc4py (Dalcin et al. 2011), Cysparse, Krypy, and PyMumps. PyNumero also provides interfaces for the HSL linear solvers MA27 and MA57 to solve sparse, symmetric linear systems. These latter solvers are important in interior-point algorithms as they also provide the inertia (number of positive and negative eigenvalues) of the factorized matrix.

Listing 3 illustrates how to use PyNumero's interface to MA27 to solve a symmetric linear system of equations. Line 2 imports the `MA27Interface` from PyNumero. Lines 5–6 call functions to construct the matrix and right-hand-side. Lines 7–8 construct an instance of `MA27Interface` and set the pivot tolerance to 10^{-6}. Line 9 extracts the lower triangular portion of the matrix (the matrix is symmetric). Lines 10–17 perform the symbolic factorization, numeric factorization, and back-solve. These methods enable the use of a single symbolic factorization for multiple matrices of the same nonzero structure or a single factorization for multiple back-solves. This example highlights the ease of using an efficient linear solver through a Python interface.

2.4 NLP Interfaces

When interfacing to the NLP model, PyNumero considers general nonlinear programming problems of the form:

$$\begin{aligned} \min \quad & f(x) \\ & g_L \leq g(x) \leq g_U \\ & x_L \leq x \leq x_U \end{aligned} \tag{3}$$

where $x \in \mathbb{R}^n$ are the primal variables with lower and upper bounds $x_L \in \mathbb{R}^n$ and $x_U \in \mathbb{R}^n$ respectively. The inequality constraints $g : \mathbb{R}^n \to \mathbb{R}^m$ are bounded by $g_L \in \mathbb{R}^m$ and $g_U \in \mathbb{R}^m$. PyNumero also provides an interface with explicit distinction between the equality

(where $g_L = g_U$) and inequality constraints (where $g_L < g_U$) to facilitate the implementation of algorithms that require such distinction,

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & c(x) = 0 \\ & d_L \leq d(x) \leq d_U \\ & x_L \leq x \leq x_U \end{aligned} \tag{4}$$

The equality constraints are represented by $c : \mathbb{R}^n \to \mathbb{R}^{m_c}$ and $d : \mathbb{R}^n \to \mathbb{R}^{m_d}$ denotes the inequality constraints with bounds $d_L \in \mathbb{R}^{m_d}$ and $d_U \in \mathbb{R}^{m_d}$ and $m = m_c + m_d$.

Gradient-based optimization algorithms have been proven to be among the most effective algorithms for solving nonlinear optimization problems. The development of fast automatic differentiation tools (Andersson 2013, Griewank et al. 1996, Fourer et al. 1993) enables efficient computation of both first- and second-order derivatives. PyNumero uses the AMPL Solver Library (ASL) to compute derivative information, and the Ctypes Python package to call the underlying ASL subroutines from Python.

The Pynumero `AslNLP` class takes the problem definition in the form of an `.nl` file (Gay 2005), maps this to the form of Equations (3) or (4), and provides an API for evaluating the model and its derivatives. The `PyomoNLP` class inherits from AslNLP, providing the same API for evaluating the model, while also giving access to the associated Pyomo components for the constraints and variables. These interfaces return derivative values from the ASL using NumPy arrays and SciPy sparse matrices (Harris et al. 2020, Virtanen et al. 2020a). This leverages the capabilities within the NumPy ecosystem to avoid marshalling of data between the C and Python environments and enables performant Python implementations of gradient-based nonlinear optimization algorithms. Listings 4 and 5 show a small example of how a `PyomoNLP` instance can be used for function and derivative evaluations.

2.5 An Equality-Constrained SQP Example

In this section, we pull together material discussed earlier and implement a simple numerical optimization algorithm. This example illustrates the compactness, simplicity, and efficiency of PyNumero for developing optimization algorithms. Implemented in Python and complementing the algebraic modeling features of Pyomo, PyNumero's design has focused on maximizing code readability while minimizing performance bottlenecks. In this section we demonstrate these features by presenting the implementation of a fundamental optimization algorithm.

Consider Problem (5),

$$\min \quad f(x) \tag{5a}$$

$$\text{s.t.} \quad c(x) = 0 \tag{5b}$$

with the Lagrangian defined as

$$\mathcal{L}(x, \lambda) = f(x) + \lambda^T c(x) \tag{6}$$

Algorithm 1 presents the pseudo-code for an Equality-Constrained Sequential Quadratic Programming (SQP) algorithm for solving Problem (5). Code Listing 6 shows a PyNumero implementation of this algorithm. Lines 1-5 import the relevant modules, classes, and

```
from pyomo.contrib.pynumero.interfaces.pyomo_nlp import PyomoNLP
import pyomo.environ as pyo
import numpy as np

# define optimization model
m = pyo.ConcreteModel()
m.x = pyo.Var([1, 2, 3], bounds=(0.0, None), initialize=3.0)
m.c = pyo.Constraint(expr=m.x[3]**2 + m.x[1] == 25)
m.d = pyo.Constraint(expr=m.x[2]**2 + m.x[1] <= 18.0)
m.o = pyo.Objective(expr=m.x[1]**4 - 3*m.x[1]*m.x[2]**3 + m.x[3]**2 - 8.0)

# create NLP
nlp = PyomoNLP(m)

# Set values of variables
nlp.set_primals(np.array([4, -1, 3]))

# accessing variable values
primals = nlp.get_primals()
print("Values of primal variables:\n", primals)
duals = nlp.get_duals()
print("Values of dual variables:\n", duals)

# variable bounds
primals_lb = nlp.primals_lb()
primals_ub = nlp.primals_ub()
print("Variable lower bounds:\n", primals_lb)
print("Variable upper bounds:\n", primals_ub)

# NLP function evaluations
f = nlp.evaluate_objective()
print("Objective Function\n", f)
g = nlp.evaluate_constraints()
print("Constraints\n", g)
c = nlp.evaluate_eq_constraints()
print("Equality Constraints\n", c)
d = nlp.evaluate_ineq_constraints()
print("Inequality Constraints\n", d)

# NLP first and second-order derivatives
df = nlp.evaluate_grad_objective()
print("Gradient of Objective Function:\n", df)
jac_g = nlp.evaluate_jacobian()
print("Jacobian of Constraints:\n", jac_g)
jac_c = nlp.evaluate_jacobian_eq()
print("Jacobian of Equality Constraints:\n", jac_c)
jac_d = nlp.evaluate_jacobian_ineq()
print("Jacobian of Inequality Constraints:\n", jac_d)
hess_lag = nlp.evaluate_hessian_lag()
print("Hessian of Lagrangian\n", hess_lag)
```

Listing 4: Using `PyomoNLP` for function evaluations

```
Values of primal variables:
 [ 4. -1.  3.]
Values of dual variables:
 [0. 0.]
Variable lower bounds:
 [0. 0. 0.]
Variable upper bounds:
 [inf inf inf]
Objective Function
 -502.0
Constraints
 [-21.  19.]
Equality Constraints
 [-21.]
Inequality Constraints
 [19.]
Gradient of Objective Function:
 [-432.   -2.  -84.]
Jacobian of Constraints:
   (1, 0) 8.0
  (0, 1)  -2.0
  (0, 2)  1.0
  (1, 2)  1.0
Jacobian of Equality Constraints:
   (0, 1) -2.0
  (0, 2)  1.0
Jacobian of Inequality Constraints:
   (0, 0) 8.0
  (0, 2)  1.0
Hessian of Lagrangian
   (0, 0) -216.0
  (1, 1)  2.0
  (2, 0)  -144.0
  (2, 2)  108.0
  (0, 2)  -144.0
```

Listing 5: Output of Listing 4

1. **Initialize the algorithm**
   Given initial guess for primals x^0 and duals λ^0; tolerance $\epsilon > 0$
   Set the iteration index $k \leftarrow 0$
2. **Check convergence**
   if $||\nabla_x \mathcal{L}(x^k, \lambda^k)||_\infty \leq \epsilon$ and $||c(x^k)||_\infty \leq \epsilon$ **then exit, solution found.**
3. **Compute step direction**

$$\begin{bmatrix} \nabla_x^2 \mathcal{L}(x^k, \lambda^k) & \nabla_x c(x^k)^T \\ \nabla_x c(x^k) & \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} \nabla_x \mathcal{L}(x^k, \lambda^k) \\ c(x^k) \end{bmatrix}$$

5. **Update variables**
   $x^{k+1} = x^k + \Delta x$
   $\lambda^{k+1} = \lambda^k + \Delta \lambda$
6. **Update iteration index**
   Set the iteration index $k \leftarrow k + 1$
   Return to step **2**

Algorithm 1: An Equality-Constrained SQP Algorithm

```python
from pyomo.contrib.pynumero.interfaces.nlp import NLP
from pyomo.contrib.pynumero.sparse import BlockVector, BlockMatrix
from pyomo.contrib.pynumero.linalg.ma27 import MA27Interface
import numpy as np
from scipy.sparse import tril


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

    # create and initialize the iteration vector
    z = BlockVector(2)
    z.set_block(0, nlp.get_primals())
    z.set_block(1, nlp.get_duals())

    # 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(z.get_block(0))
        nlp.set_duals(z.get_block(1))

        grad_lag = (nlp.evaluate_grad_objective() +
                    nlp.evaluate_jacobian_eq().transpose() * z.get_block(1))
        residuals = nlp.evaluate_eq_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_eq())
        kkt.set_block(0, 1, nlp.evaluate_jacobian_eq().transpose())

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

        _kkt = tril(kkt.tocoo())
        linear_solver.do_symbolic_factorization(_kkt.shape[0], _kkt.row,
                                                _kkt.col)
        linear_solver.do_numeric_factorization(_kkt.row, _kkt.col,
                                               _kkt.shape[0], _kkt.data)
        delta = linear_solver.do_backsolve(-rhs.flatten())
        z += delta
```

Listing 6: Example implementation of an equality-constrained SQP algorithm

```
from pyomo.contrib.pynumero.interfaces.pyomo_nlp import PyomoNLP

# Create a Pyomo model
m = build_burgers_model()

# Create a PyNumero PyomoNLP
nlp = PyomoNLP(m)

# Solve the problem
sqp(nlp)
```

Listing 7: Solving a Pyomo model with an example SQP implementation

functions. The `sqp` function takes an instance of a NLP object as an argument along with optional termination criteria. On line 10, we define the KKT matrix as a `BlockMatrix` with 2 block-rows and 2 block-columns (4 blocks total). On line 11, we define the KKT right-hand-side as a `BlockVector` with 2 blocks. Each block in a `BlockMatrix` can contain a SciPy sparse matrix or another `BlockMatrix`. Alternatively, a block can be empty, indicating that there are no non-zeros in that block. Each block in a `BlockVector` can contain a NumPy array or another `BlockVector`. Line 14 constructs a `BlockVector` for the iteration variables that includes the primal and dual variables. The loop for computing steps begins on line 23. On lines 24-25, we update the values of the primals and duals within the NLP object. On lines 27-33, we compute the gradient of the Lagrangian and the residuals of the constraints, check the norms for convergence, and terminate the algorithm if the tolerance has been satisfied. If not converged, we need to compute a step in the iteration variables. On lines 35-40, we build the KKT matrix and right-hand-side (`rhs`) with the Hessian of the Lagrangian, the Jacobian of the constraints, the gradient of the Lagrangian and the residuals of the constraints. On lines 42-47, we factorize and solve the KKT system. Finally, on line 48, we update the primals and duals using the computed step.

Code Listing 7 demonstrates the use of the `sqp` function in Code Listing 6 to solve a Pyomo model for a 2D PDE-constrained optimal control problem with Burgers' Equation:

$$\min \int_0^1 \int_0^1 \left[(y-\hat{y})^2 + \alpha u^2\right] dx dt \tag{7a}$$

s.t.

$$\frac{\partial y}{\partial t} - v\frac{\partial^2 y}{\partial x^2} + \frac{\partial y}{\partial x} y = u \tag{7b}$$

$$y(x=0) = 0 \tag{7c}$$

$$y(x=1) = 0 \tag{7d}$$

$$u(x=0) = 0 \tag{7e}$$

$$u(x=1) = 0 \tag{7f}$$

$$y(0 < x < 1, t = 0) = \hat{y} \tag{7g}$$

$$u(0 < x < 1, t = 0) = 0 \tag{7h}$$

$$\hat{y} = \begin{cases} 1 & x <= 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{7i}$$

The PDE-constrained optimal control problem is first discretized spatially using central finite difference, followed by backward finite difference (Biegler 2010) using Pyomo.DAE (Nicholson et al. 2018). The resulting Pyomo model returned from `build_burgers_model` (not shown here, but available online at https://github.com/Parapint/parapint) has 100,600 variables and 80,800 constraints. We construct a `PyomoNLP` instance on line 7 and pass it to the `sqp` function on line 10. On a 2.9 GHz Quad-Core Intel Core i7 MacBook Pro, IPOPT solves this problem in 1.27 seconds. The `sqp` function in Code Listing 6 solves the problem in 1.4 seconds, only 11% slower than IPOPT. This demonstrates that there is very little overhead introduced by writing the algorithm in Python.

Because PyNumero provides fast automatic differentiation capabilities for Pyomo expressions using ASL, and since all data is stored in NumPy arrays, users can write efficient implementations like this SQP algorithm in very few lines of code using standard functions within the NumPy/SciPy ecosystem.

3 Parapint

The primary goal of PyNumero is to facilitate research on decomposition algorithms for nonlinear optimization. Here we utilize the `MPIBlockVector` and `MPIBlockMatrix` classes described in Section 2.1 to develop a Schur-Complement based interior-point algorithm for parallel solution of structured NLPs. The algorithm is available in Parapint (https://github.com/Parapint/parapint), an open-source Python package built on PyNumero. In this section, we present Parapint both as an example of how PyNumero can be utilized for parallel NLP algorithm development and as a framework for future research.

As shown in Figure 5, Parapint builds on PyNumero in three ways. First, Parapint

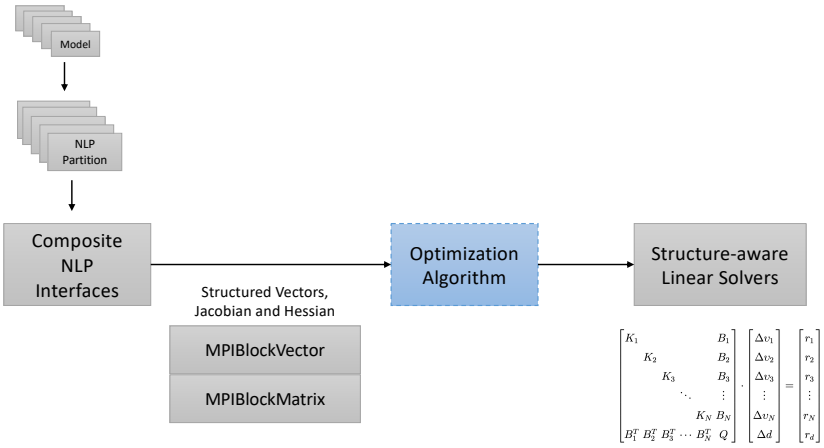

Figure 5: Parapint design

implements an interior-point algorithm for solution of NLP problems described by Problem (4). Second, Parapint extends the NLP interfaces described in Section 2.4 for stochastic programming problems and dynamic optimization problems. These interfaces expose structure in the Jacobian and Hessian computations and support parallel evaluation of model functions and derivatives. We denote these specialized NLP interfaces as *composite* NLP

interfaces since they construct a large NLP from a number of smaller partitions. Third, Parapint implements linear solvers that can recognize the structure imposed by the composite interfaces and exploit this structure for parallel solution of the linear system in the step computation. The composite NLP interfaces and the Schur-Complement approach for solving structured linear systems are discussed in Sections 3.1 and 3.2, respectively, including a scalability analysis of Parapint's Schur-Complement implementation. In Section 3.3, we briefly describe Parapint's interior-point algorithm. In Section 3.4, we show an example of how a dynamic optimization problem can be solved in parallel using Parapint, along with computational results.

3.1 Parapint Composite NLPs

Parapint currently contains two types of composite NLP interfaces – one for stochastic programming problems and one for dynamic optimization problems. Consider again the structured problem (1). These interfaces capture the problem structure using a separate Pyomo model (and NLP) for each partition, while providing a composite API that combines these partitions into a single representation for the solver. The purpose of these interfaces is threefold. First, the composite NLP interfaces provide a convenient mechanism for constructing and evaluating multiple Pyomo models in parallel (i.e., one for each partition). Second, the composite NLP interfaces automatically construct linking constraints for coupling variables between different partitions (e.g., non-anticipative variables or differential variables at finite element boundaries). Finally, they construct instances of `MPIBlockVector` and `MPIBlockMatrix` to hold function and derivative information appropriate for use with both the interior-point algorithm discussed in Section 3.3 and the structured linear solvers discussed in Section 3.2. For example, Parapint's `MPIDynamicSchurComplementInteriorPointInterface` class has a method, `evaluate_primal_dual_kkt_matrix()`, which returns a `MPIBlockMatrix` with a block-bordered-diagonal structure appropriate for Schur-Complement decomposition. An example using the composite NLP interface for dynamic optimization problems is presented in Section 3.4.

3.2 Schur-Complement Decomposition

The dominant computational cost of the interior-point method reviewed in Section 3.3 is the solution of the KKT system. In the case of stochastic or dynamic optimization problems, the structure of the KKT system can be exploited for parallel solution. With the correct permutation of rows and columns, the KKT matrix for such problems can be rearranged into a block-bordered diagonal matrix (Gondzio and Grothey 2009, Chiang et al. 2014, Kang et al. 2014).

$$\begin{bmatrix} K_1 & & & & & B_1 \\ & K_2 & & & & B_2 \\ & & K_3 & & & B_3 \\ & & & \ddots & & \vdots \\ & & & & K_N & B_N \\ B_1^T & B_2^T & B_3^T & \cdots & B_N^T & Q \end{bmatrix} \cdot \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \Delta v_3 \\ \vdots \\ \Delta v_N \\ \Delta d \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_N \\ r_d \end{bmatrix}, \tag{8}$$

Block-Gaussian elimination brings (8) to a block-upper triangular form allowing for decomposition. This decomposition is known as the Schur-complement decomposition and algorithm details are shown in Word et al. (2014).

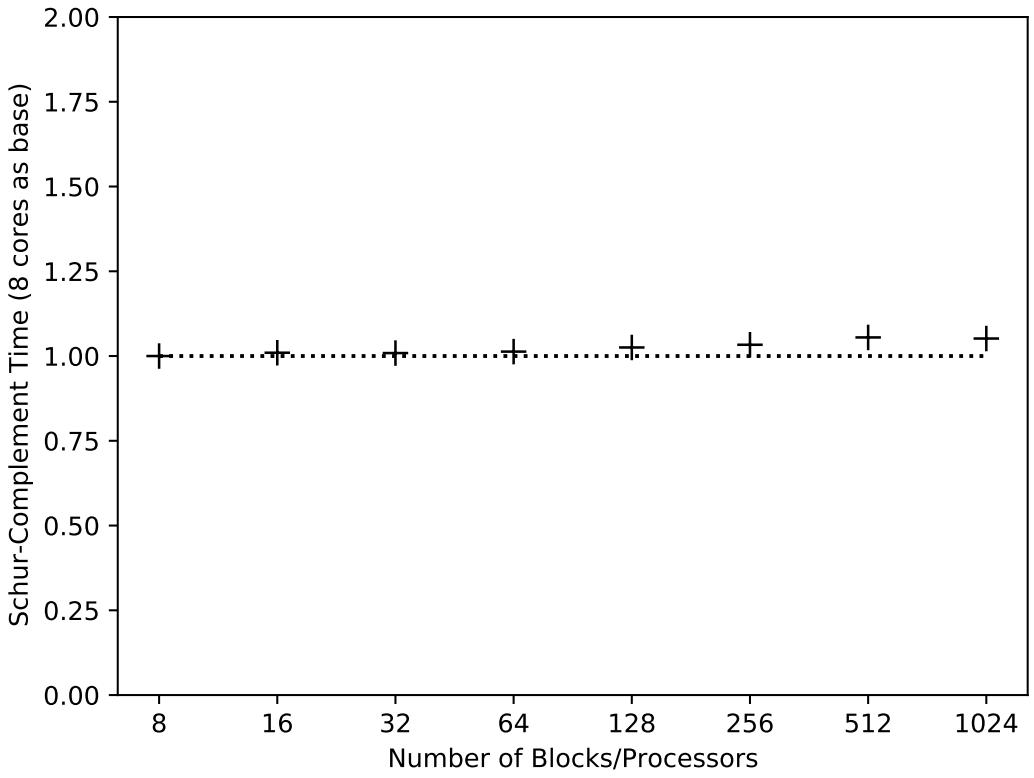

Figure 6: Weak scaling for Schur-Complement decomposition.

We implement this Schur-Complement decomposition algorithm in Parapint for parallel solution of structured linear systems of the form shown in Equation (8). We test the algorithm on a quadratic programming test problem similar to that used by Kang et al. (2014). The problem is based on least-squares parameter estimation and is presented below.

$$\min_{y,q,\theta} \sum_{l \in \mathcal{N}} ||y_l - y_l^*||_2^2 \tag{9}$$

s.t.

$$y_l - A q_l = 0 \qquad \forall l \in \mathcal{N} \tag{10}$$

$$P_l q_l - P_l^d \theta = 0 \qquad \forall l \in \mathcal{N} \tag{11}$$

Here, $\mathcal{N}$ is the set of blocks or partitions, y_l^* are vectors of data, q_l are the parameters being estimated, and θ are parameters common to all blocks. The relevant dimensions are: $q_l \in \mathbb{R}^{5,000}$, $y_l \in \mathbb{R}^{600,000}$, $A \in \mathbb{R}^{600,000\text{x}5,000}$, $\theta \in \mathbb{R}^{10}$, $P_l \in \mathbb{R}^{10\text{x}5,000}$, and $P_l^d \in \mathbb{R}^{10\text{x}10}$. Weak scaling results are presented in Figure 6. As the figure shows, when the number of coupling variables is small, the algorithm scales nearly perfectly to over 1,000 cores. The largest problem solved has over 600,000,000 variables and constraints and is solved in under 5 seconds. As shown by Kang et al. (2014), the parallel efficiency of the (explicit) Schur-Complement method degrades as the number of coupling variables increases due to the time

required to factorize a large, dense Schur-Complement. However, Figure 6 demonstrates that PyNumero and Parapint provide a viable framework for parallel NLP algorithm development and that the Python overhead is not significant for large problems.

3.3 Interior-Point Algorithm

In this section, we briefly describe the interior-point algorithm included in Parapint. Parapint's interior-point algorithm is based largely on IPOPT (Wächter and Biegler 2006), including full support for equality constraints, variable bounds, and general inequalities. The algorithm introduces slacks to transform inequality constraints to equality constraints and uses a log-barrier penalty for variable and slack bounds. The algorithm includes a fraction-to-the-boundary approach to ensure the log-barrier penalty is always well defined and uses an inertia correction to ensure a descent direction.

Parapint's interior-point algorithm is designed to be independent of the problem structure and the linear solver used for the KKT system as long as the linear solver is compatible with the structure presented. This enables the use of a single implementation for full-space solution of general NLPs, serial solution of structured problems (stochastic or dynamic), and parallel solution of structured problems (stochastic or dynamic). This design is enabled both by a well-defined API for the interior-point NLP interface (`BaseInteriorPointInterface`) and by the interoperability between NumPy arrays, SciPy sparse matrices, and PyNumero block vectors and matrices.

3.4 Parallel solution of dynamic optimization problems

In this section, we present computational results for the parallel solution of a 2-dimensional PDE-constrained optimal control problem using Parapint. The test problem is a small variant of Problem (7),

$$\min \int_0^{t_f} \int_0^1 \left[(y-\hat{y})^2 + \alpha u^2\right] dx dt \tag{12a}$$

s.t.

$$\frac{\partial y}{\partial t} - v\frac{\partial^2 y}{\partial x^2} + \frac{\partial y}{\partial x} y = u \tag{12b}$$

$$y(x=0) = 0 \tag{12c}$$

$$y(x=1) = 0 \tag{12d}$$

$$u(x=0) = 0 \tag{12e}$$

$$u(x=1) = 0 \tag{12f}$$

$$y(0 < x < 1, t = 0) = \hat{y} \tag{12g}$$

$$u(0 < x < 1, t = 0) = 0 \tag{12h}$$

$$\hat{y} = \begin{cases} \lfloor \cos(2\pi t) \rceil & x <= 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{12i}$$

where $\lfloor w \rceil$ rounds w to the nearest integer. We model the problem with Pyomo.DAE (Nicholson et al. 2018, Bynum et al. 2021) and discretize the problem with central finite difference with respect to x and backward finite difference with respect to t. We used 30

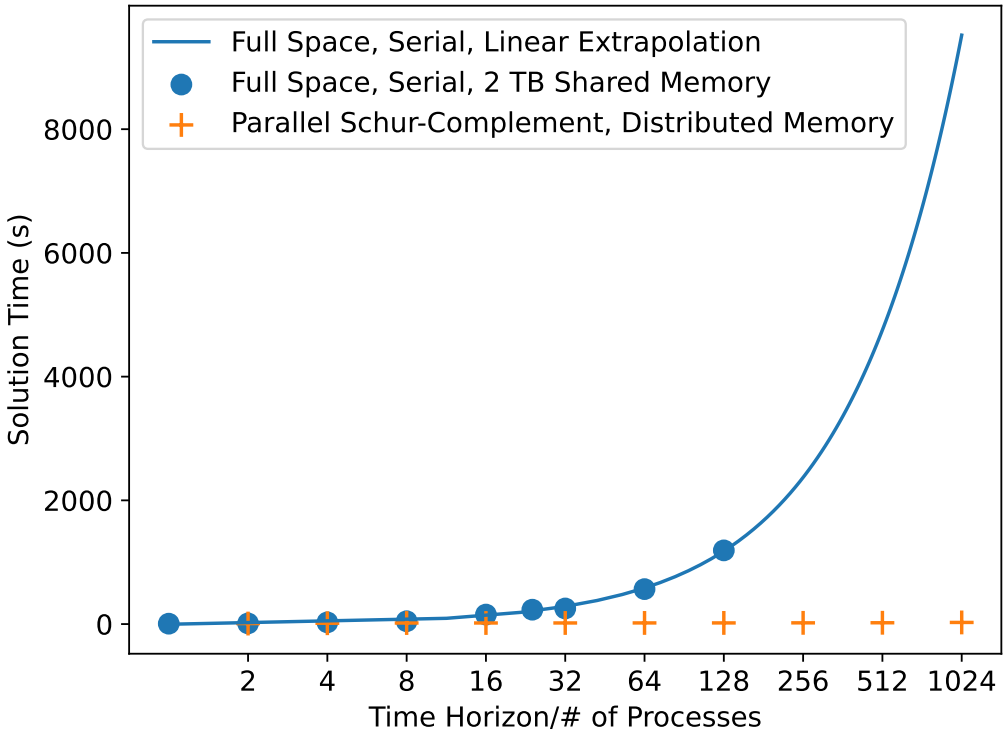

Figure 7: Scaling results for Parapint's interior-point algorithm applied to Problem (12)

.

finite elements in x and 1600 finite elements per unit time. For scalability studies, the problem size was increased by increasing t_f.

Scaling results are presented in Figure 7. We solve Problem (12) with Parapint's interior-point algorithm both in serial with a full-space method (no decomposition) and in parallel with Schur-Complement decomposition. In the full-space method, a single, large KKT system is solved with a sparse symmetric linear solver (MA27) at each iteration. The full-space method is performed on a 2-TB, shared memory machine with 40 2.8 GHz Intel Xeon CPU E7-8891 v3 cores (2 threads per core) while the Schur-Complement method is performed on a distributed memory machine with 64 GB and 8 2.6 GHz Intel Xeon CPU E5-2670 cores per node (2 threads per core). We utilize up to 8 processes per node. The x-axis shows the time horizon of the problem (t_f), which is equal to the number of processes used for the Schur-Complement method, on a logarithmic scale. The y-axis shows the solution time. We are able to solve the full-space method up to t_f=128. However, the full-space method scales very closely to linearly with t_f, and we projected the full-space solution time to t_f=1024 using linear extrapolation. As the figure shows, Parapint's Schur-Complement based interior-point algorithm scales well to over 1024 cores, achieving a projected speedup factor of approximately 360 on 1024 cores.

The largest problem solved with the Schur-Complement method has approximately 250,000,000 variables and converges in under 30 seconds. The Schur-Complement for this

```python
import pyomo.environ as pe
from pyomo import dae
import parapint
import numpy as np
from mpi4py import MPI
import logging
import argparse
import math
from pyomo.common.timing import HierarchicalTimer
import csv
import psutil
from pyomo.contrib.pynumero.sparse.mpi_block_matrix import MPIBlockMatrix


comm: MPI.Comm = MPI.COMM_WORLD  # MPI Communicator
rank = comm.Get_rank()
size = comm.Get_size()           # Number of processes

logger = logging.getLogger(__name__)
```

Listing 8: Import statements for Problem (12)

problem is $59,334 \times 59,334$ with 3,439,690 non-zeros. In order to achieve good scalability, Parapint's Schur-Complement linear solver exploits sparsity of the Schur-Complement matrix for efficient communication and factorization. The script used to generate these results is presented in Listings 8 – 11. Listing 8 shows the required import statements along with a few statements setting up the MPI communicator and the logger. Listing 9 shows the function used to build a Pyomo model for Problem (12) given a time window. Listing 10 shows how to setup the composite NLP interface used by the interior-point algorithm. Note that the `build_model_for_time_block` method returns the state variables at the start and end of the time window and the base class automatically introduces the coupling variables and sets up the linking constraints. Finally, Listing 11 shows how to solve the problem and record the results.

4 Distribution

The PyNumero package can be obtained with Pyomo. All Python files are distributed under the Pyomo umbrella available at https://github.com/Pyomo. Instructions for compiling the extensions can be found at https://pyomo.readthedocs.io/en/stable/contributed_packages/pynumero/installation.html.

Parapint can be installed with pip through PyPI (https://pypi.org/project/parapint/).

5 Conclusions and Future Directions

We discussed the design and implementation of a Python package called PyNumero. The package extends the modeling capabilities of Pyomo providing building blocks for writing numerical algorithms for nonlinear optimization from Python. PyNumero combines the modeling features of Pyomo with efficient libraries like ASL and NumPy/SciPy. With this combination, PyNumero performs all linear algebra operations in compiled code, and is designed to avoid marshalling of data between the C and Python environments, allowing for high-level development of algorithms without a significant sacrifice in performance. These features are demonstrated with a variety of applications presented in the paper. Timing results together with code snippets are also included. Among these applications we have shown that an implementation of an equality-constrained SQP algorithm in PyNumero has comparable solution times with the state of the art solver Ipopt when solving a dynamic

```
def build_burgers_model(nfe_x=50, nfe_t=50, start_t=0, end_t=1, add_init_conditions=True):
    dt = (end_t - start_t) / float(nfe_t)  # finite element size (time)
    start_x = 0
    end_x = 1
    dx = (end_x - start_x) / float(nfe_x)  # finite element size (space)

    m = pe.Block(concrete=True)
    m.omega = pe.Param(initialize=0.02)
    m.v = pe.Param(initialize=0.01)
    m.r = pe.Param(initialize=0)
    m.x = dae.ContinuousSet(bounds=(start_x, end_x))
    m.t = dae.ContinuousSet(bounds=(start_t, end_t))
    m.y = pe.Var(m.x, m.t)
    m.dydt = dae.DerivativeVar(m.y, wrt=m.t)
    m.dydx = dae.DerivativeVar(m.y, wrt=m.x)
    m.dydx2 = dae.DerivativeVar(m.y, wrt=(m.x, m.x))
    m.u = pe.Var(m.x, m.t)

    def _y_init_rule(m, x, t):  # desired state profile
        if x <= 0.5 * end_x:
            return 1 * round(math.cos(2*math.pi*t))
        return 0
    m.y0 = pe.Param(m.x, m.t, default=_y_init_rule)

    def _upper_x_bound(m, t):  # boundary conditions
        return m.y[end_x, t] == 0
    m.upper_x_bound = pe.Constraint(m.t, rule=_upper_x_bound)

    def _lower_x_bound(m, t):  # boundary conditions
        return m.y[start_x, t] == 0
    m.lower_x_bound = pe.Constraint(m.t, rule=_lower_x_bound)

    def _upper_x_ubound(m, t):  # no control at bounary
        return m.u[end_x, t] == 0
    m.upper_x_ubound = pe.Constraint(m.t, rule=_upper_x_ubound)

    def _lower_x_ubound(m, t):  # no control at boundary
        return m.u[start_x, t] == 0
    m.lower_x_ubound = pe.Constraint(m.t, rule=_lower_x_ubound)

    def _lower_t_bound(m, x):  # initial conditions
        if x == start_x or x == end_x:
            return pe.Constraint.Skip
        return m.y[x, start_t] == m.y0[x, start_t]

    def _lower_t_ubound(m, x):  # initial control
        if x == start_x or x == end_x:
            return pe.Constraint.Skip
        return m.u[x, start_t] == 0

    if add_init_conditions:
        m.lower_t_bound = pe.Constraint(m.x, rule=_lower_t_bound)
        m.lower_t_ubound = pe.Constraint(m.x, rule=_lower_t_ubound)

    def _pde(m, x, t):  # The governing PDE
        if t == start_t or x == end_x or x == start_x:
            e = pe.Constraint.Skip
        else:
            e = m.dydt[x, t] - m.v*m.dydx2[x, t] + m.dydx[x, t]*m.y[x, t] == m.r + m.u[x, m.t.
     prev(t)]
        return e
    m.pde = pe.Constraint(m.x, m.t, rule=_pde)

    disc = pe.TransformationFactory('dae.finite_difference')
    disc.apply_to(m, nfe=nfe_t, wrt=m.t, scheme='BACKWARD')  # discretize space
    disc.apply_to(m, nfe=nfe_x, wrt=m.x, scheme='CENTRAL')   # discretize time

    def _intX(m, x, t):  # Objective integral (space)
        return (m.y[x, t] - m.y0[x, t]) ** 2 + m.omega * m.u[x, t] ** 2
    m.intX = dae.Integral(m.x, m.t, wrt=m.x, rule=_intX)

    def _intT(m, t):  # Objective integral (time)
        return m.intX[t]
    m.intT = dae.Integral(m.t, wrt=m.t, rule=_intT)

    def _obj(m):  # minor correction to integral at block-boundaries
        e = 0.5 * m.intT
        for x in sorted(m.x):
            if x != start_x and x != end_x:
                e += 0.5 * 0.5 * dx * dt * m.omega * m.u[x, start_t] ** 2
        return e
    m.obj = pe.Objective(rule=_obj)

    return m
```

Listing 9: A Pyomo model for Problem (12)

```
class BurgersInterface(parapint.interfaces.MPIDynamicSchurComplementInteriorPointInterface):
    def __init__(self, start_t, end_t, num_time_blocks, nfe_t, nfe_x):
        self.nfe_x = nfe_x
        self.dt = (end_t - start_t) / float(nfe_t)
        super(BurgersInterface, self).__init__(start_t=start_t, end_t=end_t,
                                               num_time_blocks=num_time_blocks, comm=comm)

    def build_model_for_time_block(self, ndx, start_t, end_t, add_init_conditions):
        nfe_t = math.ceil((end_t - start_t) / self.dt)
        m = build_burgers_model(nfe_x=self.nfe_x, nfe_t=nfe_t, start_t=start_t, end_t=end_t,
                                add_init_conditions=add_init_conditions)

        return (m, ([m.y[x, start_t] for x in sorted(m.x) if x not in {0, 1}]),
                   ([m.y[x, end_t] for x in sorted(m.x) if x not in {0, 1}]))
```

Listing 10: Setting up the composite NLP interface for Problem (12)

optimization problem with 100K variables and 80K constraints. The overhead from the Python interface to ASL and HSL only increases the solution time by 11% for large-scale instances.

PyNumero uses object-oriented principles comprehensively, applying them to algorithms and problem formulations that exploit block-structures via polymorphism and inheritance mechanisms. Since block-structured problems result from real-life optimization problems, we expect the design to promote research of decomposition algorithms. Of special interest are stochastic programming problems and dynamic optimization problems. Current developments in Pyomo to model dynamics and uncertainty in optimization problems (Watson et al. 2012, Nicholson et al. 2018) can be combined with features offered in PyNumero to prototype and explore new decomposition approaches.

PyNumero's parallel, block-based linear algebra tools make it possible to write efficient and scalable parallel NLP algorithms in Python. As an example, we presented Parapint, a Python package built on PyNumero for parallel solution of stochastic and dynamic optimization problems. Parapint currently includes a Schur-Complement based interior-point algorithm, and computational results were presented illustrating excellent performance to at least 1024 cores.

As part of future work we plan to include interfaces to different automatic differentiation packages. Currently, PyNumero relies on ASL to compute first and second derivatives. Efficient packages available in Python like CasADi and PyAdolC would be excellent extensions to PyNumero. Additionally, Parapint will be extended to include additional methods for parallel solution of stochastic and dynamic optimization problems, including cyclic reduction (Wan et al. 2019), overlapping Schwarz (Shin et al. 2020a), and implicit methods (Kang et al. 2014).

6 Acknowledgements

The authors would like to thank V. Zavala and D. Ridzal for their valuable inputs.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and 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-NA-0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. This work was funded in part by the Institute for the Design of Advanced Energy Systems (IDAES) with funding from the Office of Fossil Energy,

```python
def setup_logging(args):
    if rank == 1:
        logging.basicConfig(level=logging.INFO)


class Args(object):
    def __init__(self):
        self.nfe_x = 50  # number of finite elements in space
        self.nfe_t = 200 # number of finite elements in time per unit time
        self.end_t = 1   # time horizon
        self.nblocks = 4 # number of blocks for decomposition

    def parse_arguments(self):
        parser = argparse.ArgumentParser()
        parser.add_argument('--nfe_x', type=int, required=True, help='number of finite elements for x')
        parser.add_argument('--end_t', type=int, required=True, help='end time')
        parser.add_argument('--nfe_t_per_t', type=int, required=False, default=100, help='number of finite elements for t per unit time')
        parser.add_argument('--nblocks', type=int, required=True, help='number of time blocks for schur complement')
        args = parser.parse_args()
        self.nfe_x = args.nfe_x
        self.end_t = args.end_t
        self.nfe_t = args.nfe_t_per_t * args.end_t
        self.nblocks = args.nblocks


def main(args, subproblem_solver_class, subproblem_solver_options):
    # construct the composite NLP interface
    interface = BurgersInterface(start_t=0,
                                 end_t=args.end_t,
                                 num_time_blocks=args.nblocks,
                                 nfe_t=args.nfe_t,
                                 nfe_x=args.nfe_x)
    # construct the Schur-Complement linear solver
    linear_solver = parapint.linalg.MPISchurComplementLinearSolver(
        subproblem_solvers={ndx: subproblem_solver_class(**subproblem_solver_options) for ndx in range(args.nblocks)},
        schur_complement_solver=subproblem_solver_class(**subproblem_solver_options))
    # Specify options for the interior point algorithm
    options = parapint.algorithms.IPOptions()
    options.linalg.solver = linear_solver
    # construct a timer for reporting stats on computational performance
    timer = HierarchicalTimer()
    comm.Barrier()
    # Solve the problem with the interior point algorithm
    status = parapint.algorithms.ip_solve(interface=interface, options=options, timer=timer)
    assert status == parapint.algorithms.InteriorPointStatus.optimal
    # Store the results in a csv file
    n_primals = interface.n_primals()
    logger.info('\n' + str(timer))
    if rank == 1:
        f = open('burgers_' + str(args.end_t) + '_' + str(args.nfe_x) + '_' + str(args.nfe_t) + '_' + str(size) + '.csv', 'w')
        fieldnames = ['end_t', 'nfe_x', 'nfe_t', 'size', 'n_blocks', 'n_primals', 'sc_nnz', 'sc_dim', 'virt_mem', 'cpu_percent']
        timer_identifiers = timer.get_timers()
        fieldnames.extend(timer_identifiers)
        writer = csv.writer(f)
        writer.writerow(fieldnames)
        row = [args.end_t, args.nfe_x, args.nfe_t, size, args.nblocks, n_primals,
               linear_solver.schur_complement.data.size,
               linear_solver.schur_complement.shape[0],
               psutil.virtual_memory().percent, psutil.cpu_percent()]
        row.extend(timer.get_total_time(name) for name in timer_identifiers)
        writer.writerow(row)
        f.close()


if __name__ == '__main__':
    args = Args()
    args.parse_arguments()
    setup_logging(args)
    # cntl[1] is the MA27 pivot tolerance
    main(args=args,
         subproblem_solver_class=parapint.linalg.InteriorPointMA27Interface,
         subproblem_solver_options={'cntl_options': {1: 1e-6}})
```

Listing 11: Solving Problem (12)

Cross-Cutting Research, U.S. Department of Energy. This work was also funded by Sandia National Laboratories Laboratory Directed Research and Development (LDRD) program.

References

Patrick R Amestoy, Iain S Duff, Jean-Yves L'Excellent, and Jacko Koster. Mumps: a general purpose distributed memory sparse solver. In *International Workshop on Applied Parallel Computing*, pages 121–130. Springer, 2000.

J. Andersson. *A General-Purpose Software Framework for Dynamic Optimization*. PhD thesis, Arenberg Doctoral School, KU Leuven, Department of Electrical Engineering (ESAT/SCD) and Optimization in Engineering Center, Kasteelpark Arenberg 10, 3001-Heverlee, Belgium, October 2013.

L T. Biegler. *Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes*. SIAM, 2010.

Michael L Bynum, Gabriel A Hackebeil, William E Hart, Carl D Laird, Bethany L Nicholson, John D Siirola, Jean-Paul Watson, and David L Woodruff. *Pyomo—Optimization Modeling in Python*, volume 67. Springer Nature, 2021.

Naiyuan Chiang, Cosmin G Petra, and Victor 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.

Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. Parallel distributed computing using Python. *Advances in Water Resources*, 34(9):1124–1139, September 2011.

Haoyang Deng and Toshiyuki Ohtsuka. A parallel newton-type method for nonlinear model predictive control. *Automatica*, 109:108560, 2019.

Iain S Duff and John K Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software (TOMS)*, 9(3):302–325, 1983.

Jonathan Eckstein and Dimitri P Bertsekas. On the Douglas-Rachford splitting method and the proximal point algorithm for maximal monotone operators. *Mathematical Programming*, 55 (1-3):293–318, 1992.

R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Scientific Press, 1993. ISBN 9780894262333. URL https://books.google.com/books?id=8vJQAAAAMAAJ.

David M Gay. Hooking your solver to ampl. Technical report, Citeseer, 1997.

David M Gay. Writing `.nl` files. Technical report, Sandia National Laboratories, 2005.

J. Gondzio and A. Grothey. Exploiting structure in parallel implementation of interior point methods for optimization. *Computational Management Science*, 6(2):135–160, May 2009.

Andreas Griewank, David Juedes, and Jean Utke. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in c/c++. *ACM Trans. Math. Softw.*, 22(2): 131–167, June 1996. ISSN 0098-3500.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

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*, 71:563–573, 2014.

Bethany Nicholson, John D Siirola, Jean-Paul Watson, Victor M Zavala, and Lorenz T Biegler. pyomo.dae: a modeling and automatic discretization framework for optimization with differential and algebraic equations. *Mathematical Programming Computation*, 10(2):187–223, 2018.

Cosmin G. Petra, Olaf Schenk, Miles Lubin, and Klaus Gäertner. An augmented incomplete factorization approach for computing the schur complement in stochastic optimization. *SIAM Journal on Scientific Computing*, 36(2):C139–C162, 2014. doi: 10.1137/130908737. URL https://doi.org/10.1137/130908737.

R Tyrrell Rockafellar and Roger J-B Wets. Scenarios and policy aggregation in optimization under uncertainty. *Mathematics of operations research*, 16(1):119–147, 1991.

Jose S Rodriguez, Bethany Nicholson, Carl Laird, and Victor M Zavala. Benchmarking ADMM in nonconvex NLPs. *Computers & Chemical Engineering*, 119:315–325, 2018.

Jose S Rodriguez, Carl D Laird, and Victor M Zavala. Scalable preconditioning of block-structured linear algebra systems using ADMM. *Computers & Chemical Engineering*, 133:106478, 2020.

M. Sala, W. Spotz, and M. Heroux. PyTrilinos: High-performance distributed-memory solvers for Python. *ACM Transactions on Mathematical Software (TOMS)*, 34, March 2008.

Sungho Shin, Mihai Anitescu, and Victor M Zavala. Overlapping schwarz decomposition for constrained quadratic programs. In *2020 59th IEEE Conference on Decision and Control (CDC)*, pages 3004–3009. IEEE, 2020a.

Sungho Shin, Carleton Coffrin, Kaarthik Sundar, and Victor M Zavala. Graph-based modeling and decomposition of energy infrastructures. *arXiv preprint arXiv:2010.02404*, 2020b.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020a. doi: 10.1038/s41592-019-0686-2.

Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020b.

Andreas Wächter and Lorenz 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.

Wei Wan, John P Eason, Bethany Nicholson, and Lorenz T Biegler. Parallel cyclic reduction decomposition for dynamic optimization problems. *Computers & Chemical Engineering*, 120: 54–69, 2019.

Jean-Paul Watson, David L Woodruff, and William E Hart. PySP: modeling and solving stochastic programs in Python. *Mathematical Programming Computation*, 4(2):109–149, 2012.

Daniel P Word, Jean-Paul Watson, David L Woodruff, and Carl D Laird. A progressive hedging approach for parameter estimation via stochastic nonlinear programming. In *Computer Aided Chemical Engineering*, volume 31, pages 1507–1511. Elsevier, 2012.

Daniel P Word, Jia Kang, Johan Akesson, and Carl D Laird. Efficient parallel solution of large-scale nonlinear dynamic optimization problems. *Computational Optimization and Applications*, 59 (3):667–688, 2014.

Victor M Zavala, Carl D Laird, and Lorenz T Biegler. Interior-point decomposition approaches for parallel solution of large-scale nonlinear parameter estimation problems. *Chemical Engineering Science*, 63(19):4834–4845, 2008.