Coopr:
a COmmon Optimization Python Repository

William E. Hart
Jean-Paul Watson
Sandia National Laboratories
wehart@sandia.gov

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company,
for the United States Department of Energy's National Nuclear Security Administration
under contract DE-AC04-94AL85000.

Overview

GOAL: integrate Python packages related to modeling and optimization

- **coopr.opt**
  - Generic interfaces for optimization solvers
- **coopr.pyomo**
  - A Pythonic math programming modeling tool
- **coopr.pysos**
  - Create optimization applications from heterogeneous models
- **coopr.pysp**
  - Stochastic programming extensions for Pyomo
- **coopr.sucasa**
  - Customizing IP solvers to integrate symbolic information

coopr.sucasa

SUCASA: the Solver Utility for Customization with Automatic Symbol Access

Goal: support customized MILP solvers that can leverage algebraic problem structure

Impact: Enable customization of…

- branching strategies
- incumbent heuristics
- cutting planes
- application I/O
- etc…

Applying SUCASA

Idea: Customize the PICO MILP solver to integrate a class that contains algebraic information that is exported by AMPL

Phase I:

- Parse AMPL model

```
sucasa --acro=<dir> -g pmedian.mod
```

- Generate *.map file that summarizes symbols to be exported
- Generate customized PICO classes

Phase II:

- Build customized PICO solver

```
make
```

Phase III:

- Apply customized PICO solver, using exported symbols

```
sucasa pmedian.mod pmedian.dat
```

Capturing Symbolic Information

Default behavior: capture symbolic information for

- Variables
- Constraints
- Associated sets needed to index these symbols

AMPL comments can be used to expose the symbols for sets and parameters

Example: expose all set and parameter symbols

```
# SUCASA SYMBOLS: *
```

Example: expose the N and Locations symbols

```
# SUCASA SYMBOLS: N Locations
```

Example: Variable and Constraint Indexing

- Variables are indexed by tuples and explicit indices:
  - x(tuple)
  - x(index1,index2)
- These methods return integer indices into the list of variables used by PICO
- Methods can be used to test whether tuples or indices are valid:
  - x_isvalid(tuple)
  - x_isvalid(index1, index2)
- The set of valid indices is returned by the x_valid() method

- Similar methods are available for constraints
  - The indexing functions return the constraint index

coopr.pysos

PYSOS: a Python framework for composing optimization formulations from heterogeneous components

Idea: integrate modeling components like…

- Python classes
- Excel spreadsheets
- MILP formulations
- Etc…

Goal: coordinate the interface between components

- Map outputs from one component to inputs of another
- Cache component input/output values
- Automate the execution of components

coopr.opt

Coopr Opt: a Python framework managing the execution of optimization solvers

Idea: Provide high-level components for

- Problems
- Solvers
- Problem converters
- Solver managers

Note: this capability is complementary to the COIN-OR optimization services (OS) project

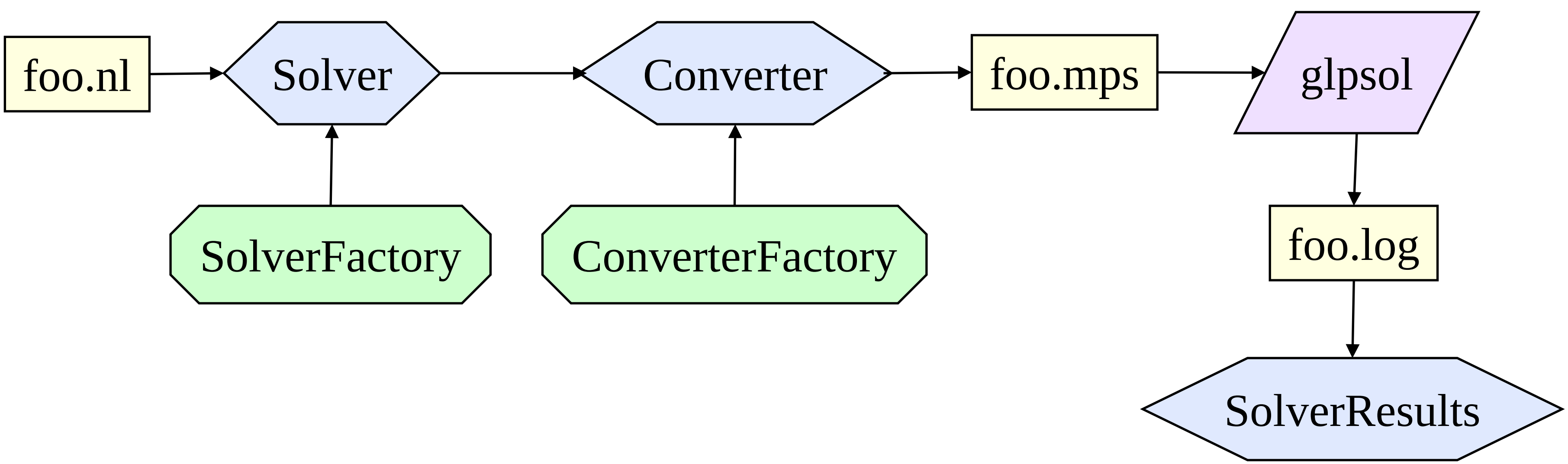
- OS defines XML standards and remote execution API
- Coopr defines an API for managing optimization instances
- TBD: develop a solver manager that uses OS services to apply solvers

A Simple Example

Idea: solve a MIP instance that is defined in a NL input file

```
import coopr.opt

opt = coopr.opt.SolverFactory('glpk')
results = opt.solve('foo.nl', log='foo.log')
results.write('foo.soln')
```



Sandia National Laboratories

Plugin Components

Goal: Support a dynamic, extensible software capability

Idea:

- Decompose software into distinct components
- Components interact through well-defined interfaces
- The plugin framework manages the interaction of components

Traditional Implementation:

- Optimization base class: defines solver API
- Optimization sub-classes: implement API for different solvers
- Solvers are *explicitly* included in software (e.g. import statements)

Plugin Implementation:

- Optimization interface class: defines solver API
- Optimization plugin classes: implement API for different solvers
- Plugin framework manages registration of solvers

coopr.opt Plugins

IProblemWriter – Plugins that write optimization problems

IProblemConverter – Plugins that convert from one optimization problem format to another

IResultsReader – Plugins that read optimization results

IOptSolver – Plugins that apply optimization solvers

Plugins Impact

- Support extensibility by core-developers without risk of destabilizing core functionality
  - **Development of new solver plugins will not impact Coopr core**
- Let third-party developers add value without requiring direct involvement of the core developers
  - **Extensions can be developed and distributed without modifying Coopr's Python distribution**
- Automate activation of external software interfaces, based on user environment
  - **Automatically register optimization solvers that are found on the user's path**
- Support run-time loading of new software capabilities
  - **Load Python EGG files with custom Coopr extensions**

coopr.pyomo

Idea: Support mathematical modeling of integer programs in Python

Goals/Requirements:

- **Flexible Open Source License**
- **Customizable Capability**
  - "Stone Soup" programming model
- **Solver Integration**
  - Support both loosely and tightly coupled solver integration
- **Abstract Model Declarations**
  - Separate modeling and data declarations
- **Flexible Programming Language**
  - A clean syntax, rich set of data types, support for object oriented programming, easily extensible, well-supported, well-documented, standard library, etc.
- **Portability**

Why Python?

- **Flexible Open Source License**
- **Features**
  - A clean syntax, rich set of data types, support for object oriented programming, namespaces, exceptions, etc.
- **Support and Stability**
  - Highly stable and well-supported
- **Documentation**
  - Extensive online documentation and several excellent books
- **Standard Library**
  - Includes a large number of useful modules
- **Extendibility and Customization**
  - Simple model for loading Python code developed by a user
  - Can easily integrate libraries that optimize compute kernels
- **Portability**

AMPL Example: prod.mod

```
set P;

param a {j in P};
param b ;
param c {j in P};
param u {j in P};

var X {j in P};

maximize Total_Profit:
        sum {j in P} c[j] * X[j];

subject to Time:
        sum {j in P} (1/ a[j]) * X[j] <= b;

subject to Limit {j in P}:
        0 <= X[j] <= u[j];
```

AMPL Example: prod.dat

```
data;

set P := bands coils;

param:     a     c     u  :=
  bands   200   25   6000
  coils   140   30   4000 ;

param b := 40;
```

Pyomo Example: prod.py (1)

```
#
# Coopr import
#
from coopr.pyomo import *
#
# Setup the model
#
example = Model(name="Prod Example")
#
# Declare sets, parameters and variables
#
example.P = Set()
example.a = Param(example.P)
example.b = Param()
example.c = Param(example.P)
example.u = Param(example.P)
example.X = Var(example.P)
```

Pyomo Example: prod.py (2)

```
# Declare objective rule and create object
def Objective_rule(instance):
  return summation(instance.c, instance.X)

example.Total_Profit = Objective(rule=Objective_rule,
                                 sense=maximize)

# Declare Time constraint rule and create object
def Time_rule(instance):
  expr = summation(instance.X, denom=instance.a)
  return expr < instance.b

example.Time = Constraint(rule=Time_rule)

# Declare Limit constraint rule and create object
def Limit_rule(j, instance):
  return(0, instance.X[j], instance.u[j])

example.Limit = Constraint(example.P, rule=Limit_rule)
```

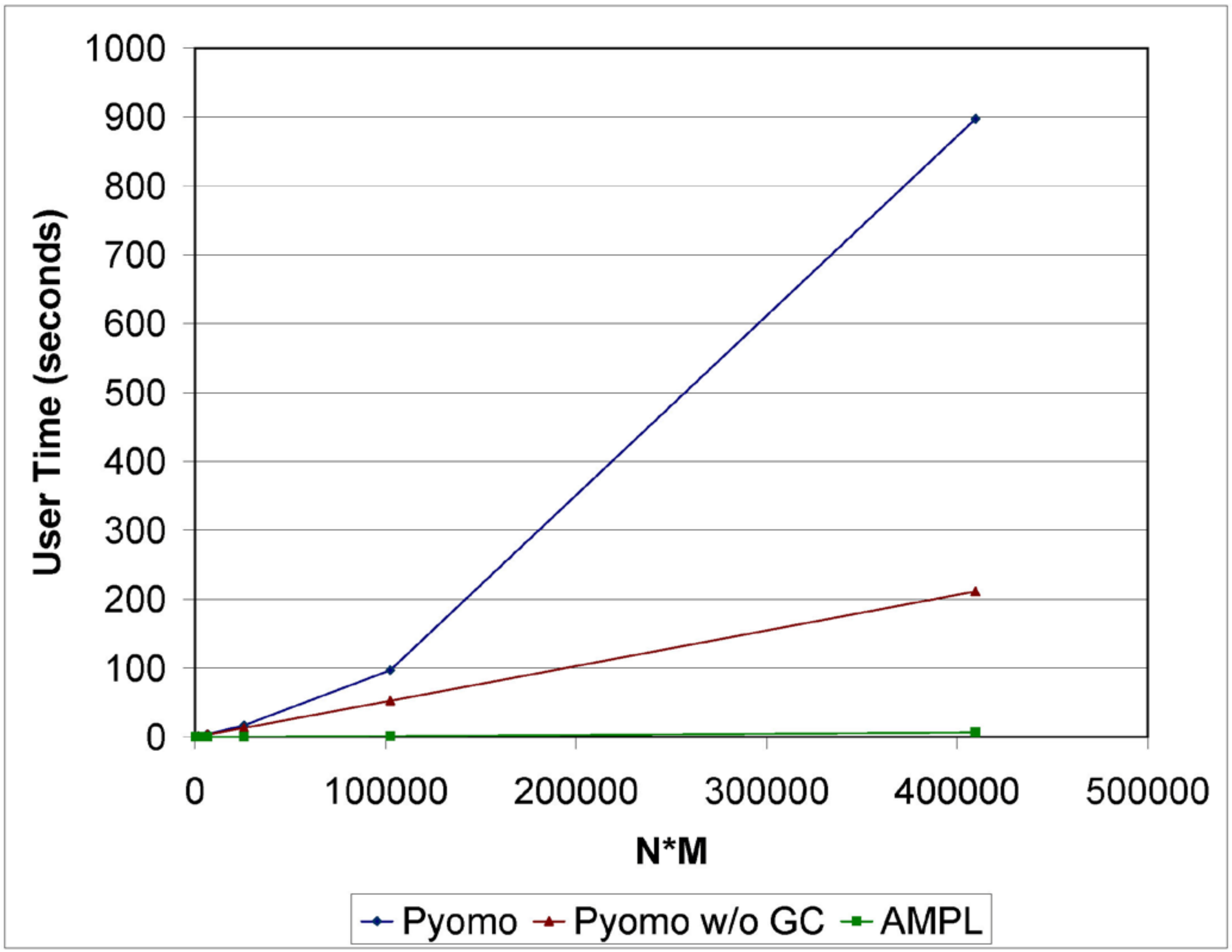
Solving a Pyomo Model within Python

```
#
# Import the prod.py file
from coopr.pyomo import *
import prod
#
# Create the model instance
instance = prod.example.create("prod.dat")
#
# Setup the optimizer
opt = solvers.SolverFactory("glpk")
#
# Optimize
results = opt.solve(instance)
#
# Write the output
results.write(num=1)
```

Pyomo Scalability

Idea: compare Pyomo and AMPL on random p-median instances

– Pyomo is tested without garbage collection, which helps…



coopr.pysp

PYSP: an extension of Pyomo to support stochastic programming

Idea: augment Pyomo to include

- Stochastic programming decomposition techniques
- Progressive hedging
- A Pythonic representation of scenario trees
- Etc…

NOTE: more details in Jean-Paul's talk

Getting Started (1)

- Download coopr_install

```
wget https://software.sandia.gov/trac/coopr/export/1543/trunk/scripts/coopr_install
```

(See the Coopr wiki: https://software.sandia.gov/svn/trac/coopr)

- Run coopr_install

```
./coopr_install coopr
```

- Use sucasa, pyomo, etc, with the virtual Python environment

```
% coopr/bin/python
>>> Import coopr.opt
<etc>
```


Sandia National Laboratories

Getting Started (2)

- Download acro-pico

```
svn checkout https://software.sandia.gov/svn/public/acro/acro-pico/trunk acro
```

(See the Acro wiki: https://software.sandia.gov/svn/trac/acro)

- Build Acro

```
cd acro
./setup configure build
```

- Use sucasa, pyomo, etc, with the virtual Python environment

```
% acro/python/bin/python
>>> Import coopr.opt
<etc>
```

Coopr Releases

Coopr 1.0 – January, 2009

- Initial Release

Coopr 1.1 – September, 2009

- Addition of PYSP
- Major improvements to SUCASA
- Addition of Coopr plugins
- Addition of the coopr_install utility
- Parallel solver manager
- Optimization of Pyomo runtime performance

Online Resources:

- Wiki https://software.sandia.gov/trac/coopr
- Coopr Forum http://code.google.com/p/coopr-forum/