Outline

1. Introduction to COIN
2. Overview of Projects
3. Using COIN
   - Optimization Services
   - SYMPHONY
   - CHiPPS
   - DECOMP
4. Conclusion
Brief Bio

- BS and MS in Mathematics from Carnegie Mellon in 1991
- PhD. in Operations Research from Cornell in 1995
- Currently
  - Associate professor at Lehigh University
  - Director of the Laboratory for Computation Optimization Research at Lehigh (COR@L).
  - Chair of the Technical Leadership Council of the COIN-OR Foundation.
- Research interests
  - Computational optimization
  - Discrete/Combinatorial optimization theory
  - Parallel/Grid computing
  - Software development
The Common Optimization Interface for Operations Research Initiative was an initiative launched by IBM at ISMP in 2000.

IBM seeded an open source repository with four initial projects and created a Web site.

The goal was to develop the project and then hand it over to the community.

The project has now grown to be self-sustaining and was spun off as a nonprofit educational foundation in the U.S. several years ago.

The name was also changed to the Computational Infrastructure for Operations Research to reflect a broader mission.
What is COIN-OR Today?

The COIN-OR Foundation

- A non-profit foundation promoting the development and use of interoperable, open-source software for operations research.
- A consortium of researchers in both industry and academia dedicated to improving the state of computational research in OR.
- A venue for developing and maintaining standards.
- A forum for discussion and interaction between practitioners and researchers.

The COIN-OR Repository

- A collection of interoperable software tools for building optimization codes, as well as a few stand alone packages.
- A venue for peer review of OR software tools.
- A development platform for open source projects, including an SVN repository.
The COIN-OR Foundation

- The foundation has been up and running for several years.
- We have two boards.
  - A **strategic board** to set overall direction
  - A **technical board** to advise on technical issues
- The boards are composed of members from both industry and academia, as well as balanced across disciplines.
- Membership in the foundation is available to both individuals and institutions.
- The foundation Web site and repository are hosted by **INFORMS**.
What You Can Do With COIN

- We currently have 30+ projects and more are being added all the time.
- Most projects are now licensed under the EPL (very permissive).
- COIN has solvers for most common optimization problem classes.
  - Linear programming
  - Nonlinear programming
  - Mixed integer linear programming
  - Mixed integer nonlinear programming (convex and nonconvex)
  - Stochastic linear programming
  - Semidefinite programming
  - Graph problems
  - Combinatorial problems (VRP, TSP, SPP, etc.)
- COIN has various utilities for reading/building/manipulating/preprocessing optimization models and getting them into solvers.
- COIN has overarching frameworks that support implementation of broad algorithm classes.
  - Parallel search
  - Branch and cut (and price)
  - Decomposition-based algorithms
**Clp**: COIN LP Solver
   
   Project Manager: John Forrest

**Cbc**: COIN Branch and Cut

   Project Manager: John Forrest

**SYMPHONY**: a flexible integer programming package that supports shared and distributed memory parallel processing, biobjective optimization, warm starting, sensitivity analysis, application development, etc.

   Project Manager: Ted Ralphs

**BLIS**: Parallel IP solver built to test the scalability of the CHiPPS framework.

   Project Manager: Ted Ralphs
**Ipopt**: Interior Point OPTimizer implements interior point methods for solving nonlinear optimization problems.

**Project Manager**: Andreas Wächter

**Bonmin**: Basic Open-source Nonlinear Mixed INteger programming is for (convex) nonlinear integer programming.

**Project Manager**: Pierre Bonami

**Couenne**: Solver for nonconvex nonlinear integer programming problems.

**Project Manager**: Pietro Belloti
**COIN-OR Projects Overview: Modeling and Interfaces**

- **Osi**: Open solver interface is a generic API for linear and mixed integer linear programs.
  
  **Project Manager**: Matthew Saltzman

- **GAMSlinks**: Allows you to use the GAMS algebraic modeling language and call COIN-OR solvers.
  
  **Project Manager**: Stefan Vigerske

- **FLOPC++**: An open-source modeling system.
  
  **Project Manager**: Tim Hultberg

- **CoinMP**: A callable library that wraps around CLP and CBC, providing an API similar to CPLEX, XPRESS, Gurobi, etc.
  
  **Project Manager**: Bjarni Kristjansson

- **Optimization Services**: A framework defining data interchange formats and providing tools for calling solvers locally and remotely through Web services.
  
  **Project Managers**: Jun Ma, Gus Gassmann, and Kipp Martin
COIN-OR Projects Overview: Frameworks

- **Bcp**: A generic framework for implementing branch, cut, and price algorithms.
  
  **Project Manager**: Laci Ladanyi

- **CHiPPS**: A framework for developing parallel tree search algorithms.
  
  **Project Manager**: Ted Ralphs

- **DECOMP (coming very soon)**: A framework for implementing decomposition-based algorithms for integer programming, including Dantzig-Wolfe, Lagrangian relaxation, cutting plane, and combinations.
  
  **Project Manager**: Ted Ralphs
- **CppAD**: a package for doing algorithmic differentiation, a key ingredient in modern nonlinear optimization codes.
  
  **Project Manager**: Brad Bell

- **CSDP**: A solver for semi-definite programs
  
  **Project Manager**: Brian Borchers

- **DFO**: An algorithm for derivative free optimization.
  
  **Project Manager**: Katya Scheinburg
Many of the tools mentioned interoperate by using the configuration and build utilities provided by the **BuildTools** project.

The **BuildTools** includes autoconf macros and scripts that allow PMs to smoothly integrate code from other projects into their own.

The **CoinAll** project is an über-project that includes a set of mutually interoperable projects and specifies specific sets of versions that are compatible.

The **TestTools** project is the focal point for testing of COIN code.

The **CoinBinary** project is a long-term effort to provide pre-built binaries for popular platforms.

  * Installers for Windows
  * RPMs for Linux
  * .debs for Linux

You can download **CoinAll** (source and/or binaries) here: here:

http://projects.coin-or.org/svn/CoinBinary/CoinAll/

http://www.coin-or.org/download/binary/CoinAll
Building CoinAll

- For MSVC++, there are project files provided.
- In *nix environments (Linux, Solaris, AIX, CYGWIN, MSys, etc.)

**Installing CoinAll**

```
svn co http://projects.coin-or.org/svn/CoinBinary/CoinAll/releases/1.3.0 CoinAll-1.3.0
cd CoinAll-1.3.0
./get.AllThirdParty
mkdir build
cd build
../configure --enable-gnu-packages -C [--prefix=/path/to/install/location]
make -j 2
make test
make install
```
Optimization Services (OS) integrates numerous COIN-OR projects. The OS project provides:

- A set of **XML based standards** for representing optimization instances (OSiL), optimization results (OSrL), and optimization solver options (OSoL).
- A **uniform API** for constructing optimization problems (linear, nonlinear, discrete) and passing them to solvers.
- A command line executable **OSSolverService** for reading problem instances in several formats and calling a solver either locally or remotely.
- Utilities that convert AMPL nl and MPS files into the OSiL format.
- Client side software for creating **Web Services** SOAP packages with OSiL instances and contact a server for solution.
- Standards that facilitate the communication between clients and solvers using Web Services.
- **Server software** that works with Apache Tomcat.
Solving a Problem on the Command Line

- The OS project provides an single executable `OSSolverService` that can be used to call most COIN solvers.
- To solve a problem in MPS format

  ```
  OSSolverService -mps ../../data/mpsFiles/parinc.mps
  ```

- The solver also accepts AMPL nl and OSiL formats.
- You can display the results in raw XML, but it’s better to print to a file to be parsed.

  ```
  OSSolverService -osil ../../data/osilFiles/parincLinear.osil -osrl result.xml
  ```

- You can then in a browser using XSLT.
  - Copy the stylesheets to your output directory.
  - Open in your browser
Specifying a Solver

OSSolverService -osil ../../data/osilFiles/p0033.osil -solver cbc

To solve a **linear program** set the solver options to:

- clp
- dylp

To solve a **mixed-integer linear program** set the solver options to:

- cbc
- symphony

To solve a **continuous nonlinear program** set the solver options to:

- ipopt

To solve a **mixed-integer nonlinear program** set the solver options to:

- bonmin
- couenne
What is the point of the OSiL format?

- Provides a single interchange standard for all classes of mathematical programs.
- Makes it easy to use existing tools for defining Web services, etc.
- Generally, however, one would not build an OSiL file directly.

To construct a model and pass it to a COIN solver, there are several routes.

- Use a modeling language—AMPL, GAMS, and MPL work with COIN-OR solvers.
- Use FlopC++.
- Build the instance in memory using COIN-OR utilities.
To use OS to call solvers in AMPL, you specify the `OSAmplClient` as the solver.

```ampl
model hs71.mod;
# tell AMPL that the solver is OSAmplClient
option solver OSAmplClient;

# now tell OSAmplClient to use Ipopt
option OSAmplClient_options "solver ipopt";

# now solve the problem
solve;
```

In order to call a remote solver service, set the solver `service` option to the address of the remote solver service.

```ampl
option ipopt_options "service http://gsbkip.chicagogsb.edu/os/OSSolverService.jws";
```
Building a Model in Memory using OS

**Step 1:** Construct an instance in a solver-independent format using the OS API.

**Step 2:** Create a solver object

```cpp
CoinSolver *solver = new CoinSolver();
solver->sSolverName = "clp";
```

**Step 3:** Feed the solver object the instance created in Step 1.

```cpp
solver->osinstance = osinstance;
```

**Step 4:** Build solver-specific model instance

```cpp
solver->buildSolverInstance();
```

**Step 5:** Solve the problem.

```cpp
solver->solve();
```
Building an OS Instance

The **OSInstance** class provides an API for constructing models and getting those models into solvers.

- **set()** and **add()** methods for creating models.
- **get()** methods for getting information about a problem.
- **calculate()** methods for finding gradient and Hessians using algorithmic differentiation.
Create an `OSInstance` object.

```cpp
OSInstance *osinstance = new OSInstance();
```

Put some variables in

```cpp
osinstance->setVariableNumber( 2);
osinstance->addVariable(0, "x0", 0, OSDBL_MAX, 'C', OSNAN, "");
osinstance->addVariable(1, "x1", 0, OSDBL_MAX, 'C', OSNAN, "");
```

There are methods for constructing

- the objective function
- constraints with all linear terms
- quadratic constraints
- constraints with general nonlinear terms
Other Options for Linear Problems

- **CoinUtils** has a number of utilities for constructing instances.
  - **PackedMatrix** and **PackedVector** classes.
  - **CoinBuild**
  - **CoinModel**

- **Osi** provides an interface for building models and getting them into solvers for linear probes.
Quick Summary of SYMPHONY Features

Using SYMPHONY
- C Library API
- OSI C++ interface
- Interactive shell
- AMPL/GMPL, GAMS, FLOPC++
- Framework for customization

Advanced Features
- Shared and distributed memory parallel MIP (since 1994)
- Biobjective MIP
- Warm starting for MIP
- Sensitivity analysis for MIP

SYMPhONY Applications
- TSP/VRP
- Set Partitioning Problem
- Mixed Postman Problem
- Capacitated Node Routing
- Multicriteria Knapsack
Quick Introduction to CHiPPS

- CHiPPS stands for COIN-OR High Performance Parallel Search.
- CHiPPS is a set of C++ class libraries for implementing tree search algorithms for both sequential and parallel environments.

### CHiPPS Components (Current)

**ALPS (Abstract Library for Parallel Search)**
- is the search-handling layer (parallel and sequential).
- provides various search strategies based on node priorities.

**BiCePS (Branch, Constrain, and Price Software)**
- is the data-handling layer for relaxation-based optimization.
- adds notion of variables and constraints.
- assumes iterative bounding process.

**BLIS (BiCePS Linear Integer Solver)**
- is a concretization of BiCePS.
- specific to models with linear constraints and objective function.
ALPS: Design Goals

- Intuitive object-oriented class structure.
  - AlpsModel
  - AlpsTreeNode
  - AlpsNodeDesc
  - AlpsSolution
  - AlpsParameterSet

- Minimal algorithmic assumptions in the base class.
  - Support for a wide range of problem classes and algorithms.
  - Support for constraint programming.

- Easy for user to develop a custom solver.

- Design for parallel scalability, but operate effective in a sequential environment.

- Explicit support for memory compression techniques (packing/differencing) important for implementing optimization algorithms.
ALPS: Overview of Features

- The design is based on a very general concept of knowledge.
- Knowledge is shared asynchronously through pools and brokers.
- Management overhead is reduced with the master-hub-worker paradigm.
- Overhead is decreased using dynamic task granularity.
- Two static load balancing techniques are used.
- Three dynamic load balancing techniques are employed.
- Uses asynchronous messaging to the highest extent possible.
- A scheduler on each process manages tasks like
  - node processing,
  - load balancing,
  - update search states, and
  - termination checking, etc.
Knowledge Sharing

- All knowledge to be shared is derived from a single base class and has an associated *encoded form*.
- Encoded form is used for *identification, storage, and communication*.
- Knowledge is maintained by one or more *knowledge pools*.
- The knowledge pools communicate through *knowledge brokers*.
Master-Hub-Worker Paradigm

Master Workers Hubs Workers
The formulation of the binary knapsack problem is

\[
\text{max} \left\{ \sum_{i=1}^{m} p_i x_i : \sum_{i=1}^{m} s_i x_i \leq c, x_i \in \{0, 1\}, i = 1, 2, \ldots, m \right\},
\]  

(1)

We derive the following classes:

- **KnapModel** (from AlpsModel) : Stores the data used to describe the knapsack problem and implements `readInstance()`
- **KnapTreeNode** (from AlpsTreeNode) : Implements `process()` (bound) and `branch()`
- **KnapNodeDesc** (from AlpsNodeDesc) : Stores information about which variables/items have been fixed by branching and which are still free.
- **KnapSolution** (from AlpsSolution) Stores a solution (which items are in the knapsack).
Then, supply the main function.

```c
int main(int argc, char* argv[]) {
    KnapModel model;

    #if defined(SERIAL)
        AlpsKnowledgeBrokerSerial broker(argc, argv, model);
    #elif defined(PARALLEL_MPI)
        AlpsKnowledgeBrokerMPI broker(argc, argv, model);
    #endif

    broker.search();
    broker.printResult();
    return 0;
}
```
BiCePS: Support for Relaxation-based Optimization

- Adds notion of *modeling objects* (variables and constraints).
- Models are built from sets of such objects.
- Bounding is an iterative process that produces new objects.
- A differencing scheme is used to store the difference between the descriptions of a child node and its parent.

```cpp
struct BcpsObjectListMod<
{
    int numRemove;
    int* posRemove;
    int numAdd;
    BcpsObject **objects;
    BcpsFieldListMod<
        lbHard;
    BcpsFieldListMod<
        ubHard;
    BcpsFieldListMod<
        lbSoft;
    BcpsFieldListMod<
        ubSoft;
};

template<class T>
struct BcpsFieldListMod<
{
    bool relative;
    int numModify;
    int* posModify;
    T* entries;
};
```
BLIS: A Generic Solver for MILP

### MILP

\[ \begin{align*}
\text{min} & \quad c^T x \\
\text{s.t.} & \quad Ax \leq b \\
& \quad x_i \in \mathbb{Z} \quad \forall i \in I
\end{align*} \]  

where \((A, b) \in \mathbb{R}^{m \times (n+1)}, c \in \mathbb{R}^n\).

### Basic Algorithmic Components

- Bounding method.
- Branching scheme.
- Object generators.
- Heuristics.
BLIS Branching scheme comprise three components:

- **Object**: has feasible region and can be branched on.
- **Branching Object**:
  - is created from objects that do not lie in their feasible regions or objects that will be beneficial to the search if branching on them.
  - contains instructions for how to conduct branching.
- **Branching method**:
  - specifies how to create a set of candidate branching objects.
  - has the method to compare objects and choose the best one.
BLIS constraint generator:

- provides an interface between BLIS and the algorithms in COIN/Cgl.
- provides a base class for deriving specific generators.
- has the ability to specify rules to control generator:
  - where to call: root, leaf?
  - how many to generate?
  - when to activate or disable?
- contains the statistics to guide generating.
BLIS: Heuristics

BLIS primal heuristic:

- defines the functionality to search for solutions.
- has the ability to specify rules to control heuristics.
  - where to call: before root, after bounding, at solution?
  - how often to call?
  - when to activate or disable?
- collects statistics to guide the heuristic.
- provides a base class for deriving specific heuristics.
BLIS can be customized easily by deriving the base C++ classes.

Sample Applications (Scott DeNegre, Ted Ralphs, Yan Xu, and others)

- Vehicle Routing Problem (VRP)
- Traveling Salesman Problem (TSP)
- Mixed Integer Bilevel Programming (MiBS)
BLIS Applications: VRP Formulation

\[
\begin{align*}
\min & \quad \sum_{e \in E} c_e x_e \\
\sum_{e = \{0, j\} \in E} x_e &= 2k, \quad (5) \\
\sum_{e = \{i, j\} \in E} x_e &= 2 \quad \forall i \in N, \quad (6) \\
\sum_{e = \{i, j\} \in E} x_e &\geq 2b(S) \quad \forall S \subseteq N, \quad |S| > 1, \quad (7) \\
0 \leq x_e &\leq 1 \quad \forall e = \{i, j\} \in E, \quad i, j \neq 0, \quad (8) \\
0 \leq x_e &\leq 2 \quad \forall e = \{i, j\} \in E, \quad (9) \\
x_e &\in \mathbb{Z} \quad \forall e \in E. \quad (10)
\end{align*}
\]
First, derive a few subclasses to specify the algorithm and model

- VrpModel (from BlisModel),
- VrpSolution (from BlisSolution),
- VrpCutGenerator (from BlisConGenerator),
- VrpHeurTSP (from BlisHeuristic),
- VrpVariable (from BlisVariable), and
- VrpParameterSet (from AlpsParameterSet).
int main(int argc, char* argv[]) {
    OsiClpSolverInterface lpSolver;
    VrpModel model;
    model.setSolver(&lpSolver);
#ifdef COIN_HAS_MPI
    AlpsKnowledgeBrokerMPI broker(argc, argv, model);
#else
    AlpsKnowledgeBrokerSerial broker(argc, argv, model);
#endif
broker.search(&model);
broker.printBestSolution();
return 0;
}

Shameless Self-Promotion

In October, 2007, the VRP/TSP solver won the Open Contest of Parallel Programming at the 19th International Symposium on Computer Architecture and High Performance Computing.
DECOMP Framework: Motivation

DECOMP is a software framework that provides a virtual sandbox for testing and comparing various decomposition-based bounding methods.

- It’s difficult to compare variants of decomposition-based algorithms.
- The method for separation/optimization over $\mathcal{P}'$ is the primary custom component of any of these algorithms.
- DECOMP abstracts the common, generic elements of these methods.
  - **Key:** The user defines methods in the space of the compact formulation.
  - The framework takes care of reformulation and implementation for all variants.
DECOMP Framework: Motivation

DECOMP is a software framework that provides a virtual sandbox for testing and comparing various decomposition-based bounding methods.

- It’s difficult to compare variants of decomposition-based algorithms.
- The method for separation/optimization over $\mathcal{P}'$ is the primary custom component of any of these algorithms.
- DECOMP abstracts the common, generic elements of these methods.
  - Key: The user defines methods in the space of the compact formulation.
  - The framework takes care of reformulation and implementation for all variants.
DECOMP Framework: Motivation

**DECOMP** is a software framework that provides a virtual sandbox for testing and comparing various decomposition-based bounding methods.

- It’s difficult to compare variants of decomposition-based algorithms.
- The method for separation/optimization over $\mathcal{P}'$ is the primary custom component of any of these algorithms.
- **DECOMP** abstracts the common, generic elements of these methods.
  - **Key:** The user defines methods in the space of the compact formulation.
  - The framework takes care of reformulation and implementation for all variants.
The **Cutting Plane Method (CP)** iteratively builds an *outer* approximation of $\mathcal{P}'$ by solving a *cutting plane* generation subproblem.
The **Cutting Plane Method (CP)** iteratively builds an *outer* approximation of $\mathcal{P}'$ by solving a *cutting plane generation subproblem.*

![Diagram of Cutting Plane Method](image1)

The **Dantzig-Wolfe Method (DW)** iteratively builds an *inner* approximation of $\mathcal{P}'$ by solving a *column generation subproblem.*

![Diagram of Dantzig-Wolfe Method](image2)
The **Cutting Plane Method (CP)** iteratively builds an *outer* approximation of $\mathcal{P}'$ by solving a cutting plane generation subproblem.

The **Dantzig-Wolfe Method (DW)** iteratively builds an *inner* approximation of $\mathcal{P}'$ by solving a column generation subproblem.

The **Lagrangian Method (LD)** iteratively solves a Lagrangian relaxation subproblem.
The **LP bound** is obtained by optimizing over the intersection of two explicitly defined polyhedra.

\[ z_{LP} = \min_{x \in \mathbb{R}^n} \{ c^T x \mid x \in Q' \cap Q'' \} \]

The **decomposition bound** is obtained by optimizing over the intersection of one explicitly defined polyhedron and one implicitly defined polyhedron.

\[ z_{CP} = z_{DW} = z_{LD} = z_D = \min_{x \in \mathbb{R}^n} \{ c^T x \mid x \in P' \cap Q'' \} \geq z_{LP} \]

Traditional decomposition-based bounding methods contain two primary steps

- **Master Problem**: Update the primal/dual solution information.
- **Subproblem**: Update the approximation of \( P' \): \( SEP(x, P') \) or \( OPT(c, P') \).

**Integrated decomposition methods** further improve the bound by considering two implicitly defined polyhedra whose descriptions are iteratively refined.

- **Price and Cut (PC)**
- **Relax and Cut (RC)**
- **Decompose and Cut (DC)**
Common Threads

- The **LP bound** is obtained by optimizing over the intersection of two explicitly defined polyhedra.

\[
    z_{LP} = \min_{x \in \mathbb{R}^n} \{ c^T x \mid x \in Q' \cap Q'' \}
\]

- The **decomposition bound** is obtained by optimizing over the intersection of one explicitly defined polyhedron and one implicitly defined polyhedron.

\[
    z_{CP} = z_{DW} = z_{LD} = z_D = \min_{x \in \mathbb{R}^n} \{ c^T x \mid x \in P' \cap Q'' \} \geq z_{LP}
\]

- Traditional decomposition-based bounding methods contain two primary steps
  - **Master Problem:** Update the primal/dual solution information.
  - **Subproblem:** Update the approximation of \( P' \): \( SEP(x, P') \) or \( OPT(c, P') \).

- Integrated decomposition methods further improve the bound by considering two implicitly defined polyhedra whose descriptions are iteratively refined.
  - Price and Cut (PC)
  - Relax and Cut (RC)
  - Decompose and Cut (DC)
The **LP bound** is obtained by optimizing over the intersection of two explicitly defined polyhedra.

$$z_{LP} = \min_{x \in \mathbb{R}^n} \{ c^T x \mid x \in Q' \cap Q'' \}$$

The **decomposition bound** is obtained by optimizing over the intersection of one explicitly defined polyhedron and one implicitly defined polyhedron.

$$z_{CP} = z_{DW} = z_{LD} = z_{D} = \min_{x \in \mathbb{R}^n} \{ c^T x \mid x \in P' \cap Q'' \} \geq z_{LP}$$

Traditional decomposition-based bounding methods contain two primary steps

- **Master Problem**: Update the primal/dual solution information.
- **Subproblem**: Update the approximation of $P'$: $SEP(x, P')$ or $OPT(c, P')$.

**Integrated decomposition methods** further improve the bound by considering two implicitly defined polyhedra whose descriptions are iteratively refined.

- **Price and Cut (PC)**
- **Relax and Cut (RC)**
- **Decompose and Cut (DC)**
The DECOMP framework, written in C++, is accessed through two user interfaces:

- **Applications Interface**: DecompApp
- **Algorithms Interface**: DecompAlgo

DECOMP provides the bounding method for branch and bound.

ALPS (Abstract Library for Parallel Search) provides the framework for parallel tree search.

- **AlpsDecompModel** : public AlpsModel
  - a wrapper class that calls (data access) methods from DecompApp
- **AlpsDecompTreeNode** : public AlpsTreeNode
  - a wrapper class that calls (algorithmic) methods from DecompAlgo
DECOMP Framework

- The **DECOMP** framework, written in C++, is accessed through two user interfaces:
  - **Applications Interface**: `DecompApp`
  - **Algorithms Interface**: `DecompAlgo`

- **DECOMP** provides the bounding method for branch and bound.

- **ALPS** (Abstract Library for Parallel Search) provides the framework for parallel tree search.
  - `AlpsDecompModel` : public `AlpsModel`
    - a wrapper class that calls (data access) methods from `DecompApp`
  - `AlpsDecompTreeNode` : public `AlpsTreeNode`
    - a wrapper class that calls (algorithmic) methods from `DecompAlgo`
COIN needs your help!

- Contribute a project
- Help develop an existing project
- Use projects and report bugs
- Volunteer to review new projects
- Develop documentation
- Develop Web site
- Chair a committee

Questions? & Thank You!