The SYMPHONY Framework for Mixed-Integer Linear Programming: Basic Features

TED RALPHS

ISE Department
COR@L Lab
Lehigh University
tkralphs@lehigh.edu

COR@L
COMPUTATIONAL OPTIMIZATION RESEARCH AT LEHIGH

DIMACS Workshop on COIN-OR

July 19, 2006

Outline

Brief Overview of SYMPHONY

- SYMPHONY is an open-source software package for solving and analyzing mixed-integer linear programs (MILPs).
- SYMPHONY can be used in three distinct modes.
  - Black box solver: From the command line or shell.
  - Callable library: From a C/C++ code.
  - Framework: Develop a customized solver or callable library.
- Advanced features
  - Warm starting
  - Sensitivity analysis
  - Bicriteria solve
  - Parallel execution

Basic Algorithm

- Core solution methodology is branch and cut.
- Default search strategy is a hybrid depth-first/best-first strategy.
- Default branching scheme is strong branching.
- Uses two of Cbc's simple primal heuristics to generate new solutions.
- Cuts can be generated using the Cut Generator Library.

- Clique
- Flow Cover
- Gomory
- Knapsack Cover
- Lift and Project
- Mixed Integer Rounding
- Odd Hole
- Probing
- Simple Rounding
- Reduce and Split
- Two-slope MIR

Basic Design

To enable parallel execution, SYMPHONY is functionally divided into five independent modules that communicate through shared or distributed memory.

SYMPHONY Modules

- Master Maintains static data between solves, spawns parallel processes, performs I/O.
- Tree Manager (TM): Controls overall execution by tracking growth of the tree and dispatching subproblems to the LP solvers.
- Node Processors (NP): Perform processing and branching operations.
- Cut Generator (CG): Generates cuts.
- Cut Pool (CP): Acts as an auxiliary cut generator by maintaining a list of the "most effective" cuts found so far.

SYMPHONY Configurations

- Each module can be compiled as an independent executable for parallel execution.
- The modules can also be combined in any number of different ways to yield other parallel configurations.
- If all modules are combined together, we get either
  - A sequential executable (with a standard C++ compiler).
  - A shared-memory parallel executable (with an OpenMP-aware C++ compiler).
- The most common distributed-memory parallel configuration is to have two executables.
  - Combined NP/CG executable:
  - Combined Master/TM/CP executable: Storing and distributing generated data (subproblem descriptions and cuts).

What's Available

- Packaged releases from `www.branchandcut.org` (old),
- Current source at SVN on `projects.coin-or.org`.
- An extensive user's manual on-line and in PDF.
- A tutorial illustrating the development of a custom solver.
- Configuration and compilation files
- Examples and Applications

SYMPHONY Solvers

- Generic MILP
- Multicriteria MILP
- Multicriteria Knapsack
- Traveling Salesman Problem
- Vehicle Routing Problem
- Mixed Postman Problem
- Set Partitioning Problem
- Matching Problem
- Network Routing

Downloading SYMPHONY

- Download packaged source releases from `www.branchandcut.org` (out of date).
- Download a source tarball from `www.coin-or.org/Tarballs`
- Download source using SVN
  - Windows (GUI): Use TortoiseSVN
  - Unix/Linux/CYGWIN/MinGW: Use `svn` from the command line

```
https://projects.coin-or.org/svn/SYMPHONY
```

- Pre-compiled binaries coming soon!
- Note that you may also consider checking out

```
https://projects.coin-or.org/svn/CoinAll
```

- The rest of this presentation is based on the current version, SYMPHONY 5.1.

Top Level Directory Structure

After checking out the code or unpacking the archive, you should see the following at the top level:

Directories

- `Cgl/`
- `Clp/`
- `CoinUtils/`
- `Osi/`
- `SYMPHONY/`
- `configure` script and other related files.

This also applies if you check out `CoinAll`, except that there will be additional top-level directories.

The `SYMPHONY/` Directory

`Applications/`

- `CNRP/`
- `MATCH/`
- `MCKP/`
- `MPP/`
- `SPP/`
- `SPP+CUTS/`
- `USER/`
- `VRP/`

`src/`

- `Common/`
- `CutGen/`
- `CutPool/`
- `LP/`
- `Master/`
- `TreeManager/`
- `WIN32/`

Other Directories

- `Datasets/`
- `Examples/`
- `include/`
- `scripts/`

Building the Default Solver and Library

- SYMPHONY does not yet build with the GNU autotools (soon!)

Building in Unix/Linux/CYGWIN/MinGW

The following commands should build everything on most architectures

```
./configure
make
make install
cd SYMPHONY
make
```

- The result will be the `symphony` (sequential) executable and the callable library `libsym.*`.
- To customize the build process, edit the `SYMPHONY/config` file (see next slide).

Customizing the Configuration

Primary Configuration Variables

- ARCH: Architecture (usually detected automatically)
- COINROOT: If other than COIN-SYMPHONY
- LIBTYPE: SHARED or STATIC
- LP_SOLVER: Any Osi solver
- HAS_READLINE: TRUE or FALSE
- USE_GLPMPL: TRUE or FALSE
- COMM_PROTOCOL: NONE or PVM
- CC: Compiler name
- OPT: Optimization level
- SYM_COMPILE_IN_*: Which modules to compile together
- SENSITIVITY_ANALYSIS: TRUE or FALSE

Using an IDE in Windows

Windows IDEs

- MSVC++ 6.0
  - Open the workspace file `SYMPHONY\WIN32\symphony.dsw`.
  - Build other COIN libraries and set the paths to them.
  - Build the `symphony` project.
- Eclipse
  - Install CYGWIN or MinGW.
  - Download and build the code as above.
  - Create an Eclipse Project.
- Dev-C++

Using an IDE in Linux

Linux IDEs

- Eclipse
  - Download and build the code as above.
  - Create an Eclipse Project.
- KDevelop

Using the Interactive Shell

```
Woody: ~/COIN/SYMPHONY> bin/CYGWIN/OSI_CLP/symphony.exe

*******************************************************
*   This is SYMPHONY Version 5.1alpha                 *
*   Copyright 2000-2005 Ted Ralphs                    *
*   All Rights Reserved.                              *
*   Distributed under the Common Public License 1.0   *
*******************************************************

***** WELCOME TO SYMPHONY INTERACTIVE MIP SOLVER ******

Please type 'help'/'?' to see the main commands!

SYMPHONY: help

List of main commands:

load      : read a problem in mps or ampl format
solve     : solve the problem
lpsolve   : solve the lp relaxation of the problem
set       : set a parameter
display   : display optimization results and stats
reset     : restart the optimizer
help      : show the available commands/params/options

quit/exit : leave the optimizer

SYMPHONY: load Datasets/sample.mps
```

Calling from the Command Line

- Read and solve a model in MPS format

Linux/Unix/CYGWIN/MinGW Shell

```
SYMPHONY/bin/$(ARCH)/$(LP_SOLVER)/symphony -F Datasets/sample.mps
```

DOS Shell

```
SYMPHONY\WIN32\Debug\symphony.exe -F Datasets/sample.mps
```

- Read and solve a model in GMPL/AMPL format:

Linux/Unix/CYGWIN/MinGW Shell

```
SYMPHONY/bin/$(ARCH)/$(LP_SOLVER)/symphony -F Datasets/sample.mod -D Datasets/sample.dat
```

DOS Shell

```
SYMPHONY\WIN32\Debug\symphony.exe -F Datasets/sample.mod -D Datasets/sample.dat
```

Setting Parameters

- Command-line parameters are set Unix style. (to get a list, invoke SYMPHONY with `-h`.

Example

```
symphony -t 1800 -v 3 -u 100 -F sample.mps
```

- To set other parameters specify a parameter file with `-f par.par`.
- The lines of the parameter file are pairs of keys and values.
- Parameters are listed in the user's manual.

Example Parameter File

```
time_limit 1800
strong_branching_cand_num_max 5
max_presolve_iter 50
```

Linking to the Callable Library

The SYMPHONY library is built along with the executable.

Unix/Linux/CYWIN/MinGW

- Library is in the `SYMPHONY/lib/$(ARCH)/$(LP_SOLVER)/` directory.
- To link to it, just include `-L''PATH''` and `-lsym` at link time.
- There is a sample Makefile in the Examples directory.

MSVC++

- Library is in the `SYMPHONY\WIN32\Debug` directory.
- To link, just add the symphony library to your project.

API Overview

Primary subroutines

- `sym_open_environment()`
- `sym_parse_command_line()`
- `sym_load_problem()`
- `sym_find_initial_bounds()`
- `sym_solve()`
- `sym_mc_solve()`
- `sym_resolve()`
- `sym_close_environment()`

API Overview (cont.)

Auxiliary subroutines

- Accessing and modifying problem data
  - `sym_set_xxx`
  - `sym_get_xxx`
- Accessing and modifying parameters:
  - `sym_set_xxx_param`
  - `sym_get_xxx_param`
- Accessing results and solver status
  - `sym_get_sol_solution`
  - `sym_get_obj_val`
- Advanced functions

Implementing a Basic MILP Solver

- We only need a few lines to implement a basic solver.
- The default command line parser can be invoked.
- The code is exactly the same for all architectures, even parallel.
- Command line would be `symphony -F model.mps`

```
#include "symphony_api.h"

int main(int argc, char **argv)
{
   sym_environment *env = sym_open_environment();
   sym_parse_command_line(env, argc, argv);
   sym_load_problem(env);
   sym_solve(env);
   sym_close_environment(env);
}
```

The SYMPHONY OSI Interface

- The COIN-OR Open Solver Interface is a standard C++ class for accessing solvers for mathematical programs.
- Each solver has its own derived class that translates OSI calls into those of the solver's library.
- For each method in OSI, SYMPHONY has a corresponding method.
- The OSI interface is implemented as wrapped C calls.
- The constructor calls `sym_open_environment()` and the destructor calls `sym_close_environment()`.
- The OSI `initialSolve()` method calls `sym_solve()`.
- The OSI `resolve()` method calls `sym_resolve()`.
- To use the SYMPHONY OSI interface, simply make the SYMPHONY OSI library.

Implementation with the OSI Interface

- Below is the implementation of a simple solver using the SYMPHONY OSI interface.
- Again, the code is the same for any configuration or architecture, sequential or parallel.

```
#include "OsiSolverInterface.hpp"
#include "OsiSymSolverInterface.hpp"

int main(int argc, char **argv)
{
   OsiSymSolverInterface si;
   si.parseCommandLine(argc, argv);
   si.loadProblem();
   si.branchAndBound();
}
```

Solving the Matching Problem

- Given an undirected graph $G = (V, E)$, the *Matching Problem* is that of selecting a minimum weight set of disjoint edges.
- The problem can be formulated as follows:

The Matching Problem

$$\min \sum_{e \in E} c_e x_e$$

$$\begin{aligned} \sum_{j \in V : e = \{i,j\} \in E} x_e &= 1 && \forall i \in V, && (1) \\ x_e &\geq 0 && \forall e \in E, && (2) \\ x_e &\in \mathbb{Z} && \forall e \in E, \end{aligned}$$

- x_e is a binary variable that takes value 1 if edge e is selected and 0 otherwise.

Data Structure

First, we need a data structure for storing the problem data.

User Data for Matching Solver

```
typedef struct MATCH_DATA{
   int numnodes;
   int cost[MAXNODES][MAXNODES];
   int endpoint1[MAXNODES*(MAXNODES-1)/2];
   int endpoint2[MAXNODES*(MAXNODES-1)/2];
   int index[MAXNODES][MAXNODES];
}match_data;
```

Reading in the Data

Next, we need to read in the data

```
int match_read_data(user_problem *prob, char *infile)
{
   int i, j;
   FILE *f = NULL;

   if ((f = fopen(infile, "r")) == NULL){
      printf("match_read_data(): user file %s can't be opened\n", infile);
      return(ERROR__USER);
   }

   /* Read in the costs */
   fscanf(f,"%d",&(prob->numnodes));
   for (i = 0; i < prob->numnodes; i++)
      for (j = 0; j < prob->numnodes; j++)
         fscanf(f, "%d", &(prob->cost[i][j]));

   return (FUNCTION_TERMINATED_NORMALLY);
}
```

Note that this could be implemented within a callback, but it is unnecessary.

Setting up Arrays

Finally, we load the problem. Here, we are declaring the arrays.

```
int match_load_problem(sym_environment *env, user_problem *prob){

   int i, j, index, n, m, nz, *matbeg, *matind;
   double *matval, *lb, *ub, *obj, *rhs, *rngval;
   char *sense, *is_int;

   /* set up the initial LP data */
   n = prob->numnodes*(prob->numnodes-1)/2;
   m = 2 * prob->numnodes;
   nz = 2 * n;

   /* Allocate the arrays */
   matbeg  = (int *) malloc((n + 1) * ISIZE);
   matind  = (int *) malloc((nz) * ISIZE);
   matval  = (double *) malloc((nz) * DSIZE);
   obj     = (double *) malloc(n * DSIZE);
   lb      = (double *) calloc(n, DSIZE);
   ub      = (double *) malloc(n * DSIZE);
   rhs     = (double *) malloc(m * DSIZE);
   sense   = (char *) malloc(m * CSIZE);
   rngval  = (double *) calloc(m, DSIZE);
   is_int  = (char *) malloc(n * CSIZE);
```

Constructing the Problem Description

Here we construct the problem description and load it.

```
   for (i = 0; i < prob->numnodes; i++) {
      for (j = i+1; j < prob->numnodes; j++) {
         prob->match1[index] = i; /*The first component of assignment 'index'*/
         prob->match2[index] = j; /*The second component of assignment 'index'*/
         prob->index[i][j] = prob->index[j][i] = index; /* To recover the index later */
         obj[index] = prob->cost[i][j]; /* Cost of assignment (i, j) */
         is_int[index] = TRUE;
         matbeg[index] = 2*index;
         matval[2*index] = 1;
         matval[2*index+1] = 1;
         matind[2*index] = i;
         matind[2*index+1] = j;
         ub[index] = 1.0;
         index++;
      }
   }
   matbeg[n] = 2 * n;
   for (i = 0; i < m; i++) {
      rhs[i] = 1;
      sense[i] = 'E';
   }
   sym_explicit_load_problem(env, n, m, matbeg, matind, matval, lb, ub,
                             is_int, obj, 0, sense, rhs, rngval, true);
   return (FUNCTION_TERMINATED_NORMALLY);
}
```

The Main Function

Here, we put it all together to get a basic solver.

```
int main(int argc, char **argv)
{
   int termcode;
   char * infile;

   /* Create a SYMPHONY environment */
   sym_environment *env = sym_open_environment();

   /* Create the data structure for storing the problem instance.*/
   user_problem *prob = (user_problem *)calloc(1, sizeof(user_problem));

   CALL_FUNCTION( sym_set_user_data(env, (void *)prob) );
   CALL_FUNCTION( sym_parse_command_line(env, argc, argv) );
   CALL_FUNCTION( sym_get_str_param(env, "infile_name", &infile));
   CALL_FUNCTION( match_read_data(prob, infile) );
   CALL_FUNCTION( match_load_problem(env, prob) );
   CALL_FUNCTION( sym_solve(env) );
   CALL_FUNCTION( sym_close_environment(env) );
   return(0);
}
```

In the talk on advanced features, we will show how to customize the solver with callbacks.