A Library Hierarchy for Implementing Scalable Parallel Search Algorithms

Yan Xu
SAS Institute Inc.

Ted Ralphs
Lehigh University

Laszlo Ladányi
IBM T.J. Watson Research Center

Matt Saltzman
Clemson University

November 16, 2006

Outline

- Motivation and background
- The Abstract Library for Parallel Search (ALPS)
  - Design and implementation
  - Preliminary computational results
- The Branch, Constrain, and Price Solver (BiCePS)
  - Design and implementation
- The BiCePS Linear Integer Solver (BLIS)
  - Design and implementation
  - Preliminary computational results
- Future work

Motivation

- ALPS is a C++ class library for implementing parallel tree search.
- ALPS is being developed in partnership with the COIN-OR Foundation, IBM, and NSF.
- A large number of frameworks and solvers already exist.
- What differentiates ALPS?
  - Intuitive interface.and open source implementation.
  - Very general, base classes make minimal algorithmic assumptions.
  - Easy to specialize for particular problem classes .
  - Designed for maximum *parallel scalability*.
  - Explicitly supports *data-intensive* algorithms.

Tree Search Algorithms

- Tree search algorithms systematically search the nodes of a directed, acyclic graph for one or more *goal nodes*.
- This process is ostensibly easy to parallelize.
- However, the graph is not known a priori and is constructed as the algorithm progresses.
- A generic tree search algorithm consists of the following elements:
  - Processing method: Is goal achieved?
  - Search strategy: What should we work on next?
  - Fathoming rule: Can node can be fathomed?
  - Branching method: What are the successors?
- The algorithm consists of choosing a candidate node, processing it, and either fathoming or branching.
- During the course of the search, various information (*knowledge*) is generated and used to guide the search.
- Efficient knowledge sharing is the key to parallelization.

Parallel Computing Concepts

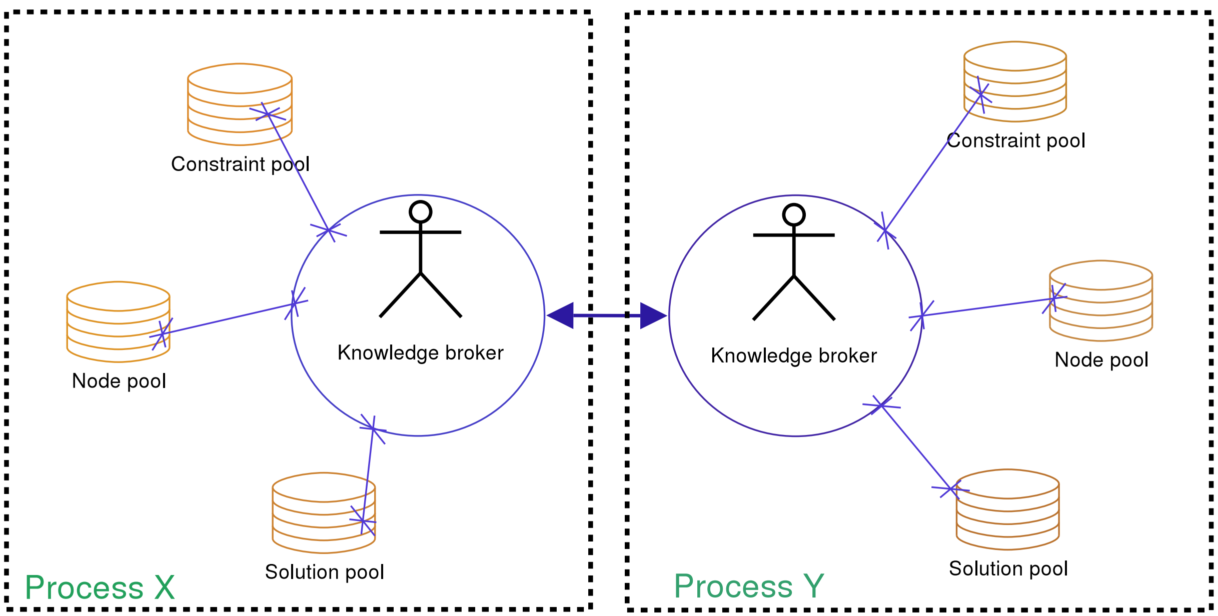
- The goal in parallelizing any algorithm is to minimize *parallel overhead*.
- The main contributors to parallel overhead in tree search are
  - Communication Overhead (cost of sharing knowledge)
  - Idle Time
    * Handshaking/Synchronization (cost of sharing knowledge)
    * Task Starvation (cost of *not* sharing knowledge)
    * Ramp Up Time (cost of sharing knowledge)
    * Ramp Down Time
  - Performance of Redundant Work (cost of *not* sharing knowledge)
- Knowledge sharing is the main driver of efficiency.
- This breakdown highlights the tradeoff between centralized and decentralized knowledge storage and decision-making.

ALPS: Features

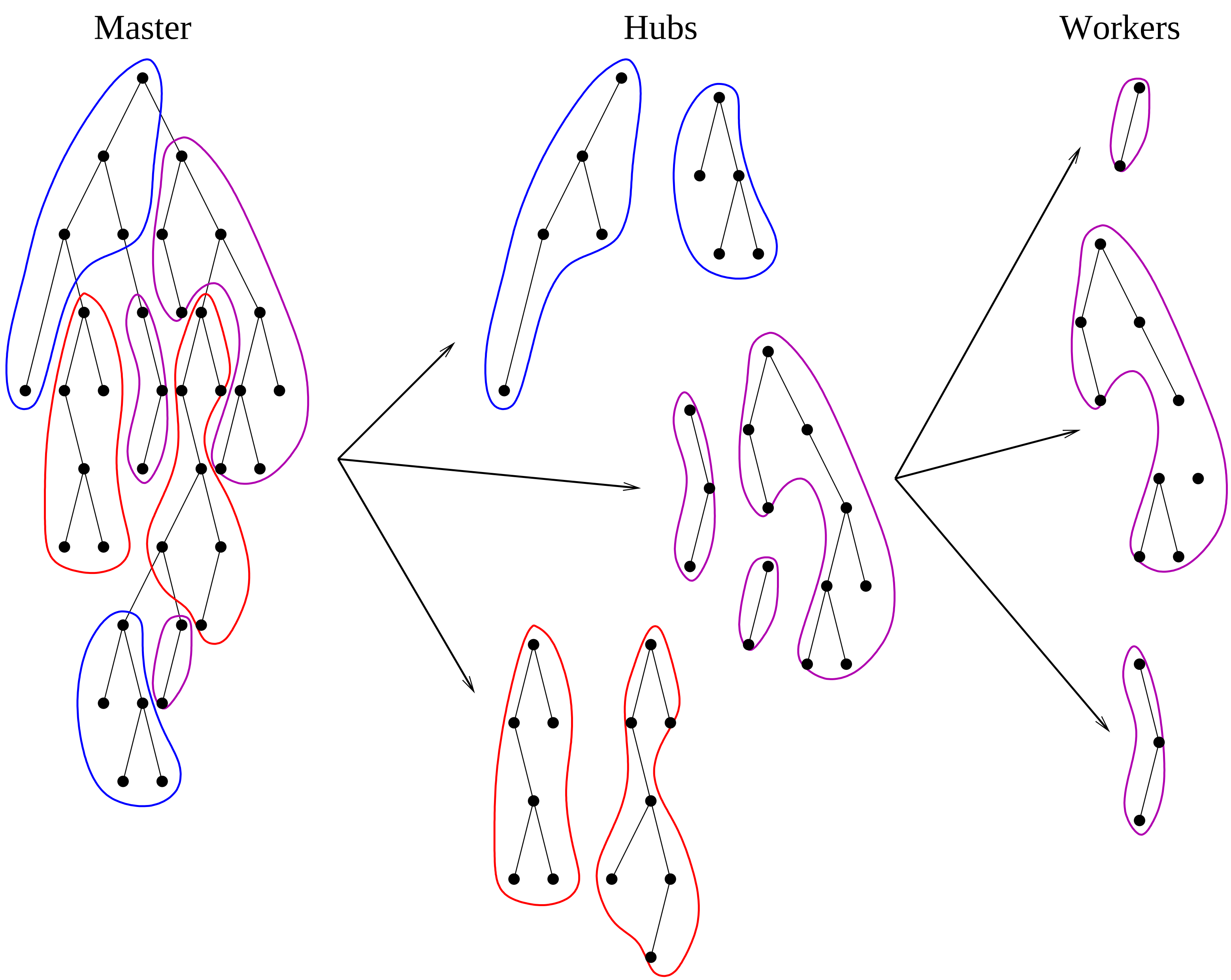
- Generality
  - ALPS only assumes that the graph to be searched is acyclic.
  - The implementation is based on a very general concept of knowledge.
- Scalability
  - Knowledge is shared asynchronously through *pools* and *brokers*.
  - Management overhead is reduced with the *master-hub-worker* paradigm.
  - Overhead is decreased using dynamic task granularity.
  - Static and dynamic load balancing techniques are employed.
  - Tasks are managed locally by a task scheduler.

ALPS: Knowledge Handling

- All knowledge to be shared is considered as `AlpsKnowledge` and has an associated encoded form.
- The encoded form is isued for identification, storage, and communication.
- `AlpsKnowledge` is managed by one or more `AlpsKnowledgePool`s.
- The knowledge pools communicate through `AlpsKnowledgeBroker`s.



ALPS: Master-Hubs-Workers Paradigm



ALPS: How to Develop an Application

Mainly comprises two parts:

- Derive the required and auxiliary problem-specific classes
  - `AlpsModel`
  - `AlpsTreeNode`
  - `AlpsNodeDesc`
  - `AlpsSolution`
- Write the `main` function

Two examples have been developed:

- Knapsack solver
- ALPS Branch and Cut (ABC)

ALPS: Computational Results of Knapsack Solver

- Test Environment:

Machine:	Beowulf cluster with 48 dual-processor nodes
Processor:	1.0 GHz Pentium III
Memory:	512M on 44 nodes, 2G on 4 nodes
Operating System:	Red Hat Linux 7.2
Message Passing:	LAM/MPI

- Experiment Design:
  - Generate *ten hard knapsack instances* based on the rule proposed in Martello ('90).
  - Run three trials for each instance, and take the average.
  - Some default parameters:
    - Two hubs are used when using 16 processes or more.
    - Dynamic load balance is activated.
    - Hubs do not processing subproblems.

ALPS: Computational Results of Knapsack Solver

- The *ten instances* have similar behavior, so we present summary results.

P	Nodes	Time	Starvation	Ramp-up	Ramp-down	Speedup
1	>190m	>7200.00	–	–	–	–
4	190m	1401.16	0.00	0.01	53.00	4.0
8	190m	639.19	0.00	0.01	46.27	8.8
16	190m	361.80	0.00	0.01	23.57	15.5
32	190m	191.50	0.00	0.01	15.00	29.3

(NOTE: m is million, time is in seconds)

- Serial code has difficult to solve several instance due to memory fraction.
- These indicate reasonable scalability.
- Starvation and ramp-up overhead is tiny.
- Ramp-down overhead has room to decrease.

Library Hierarchy for Optimization Engine

ALPS (Abstract Library for parallel search)

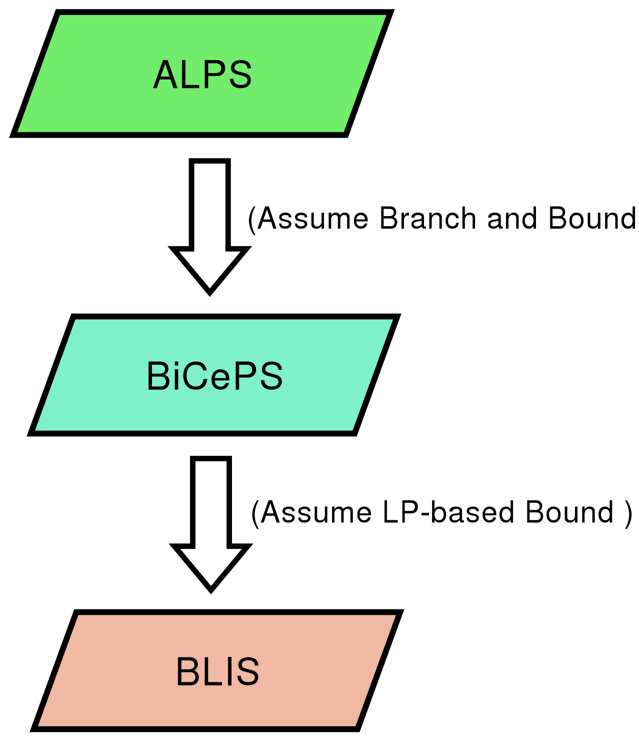
- is the search handling layer.
- prioritizes nodes based on quality.

BiCePS (Branch, Constrain, and Price Software)

- is the data handling layer for relaxation-based optimization.
- adds notion of variables and constraints.
- uses an iterative bounding procedure.

BLIS (BiCePS Linear Integer Solver)

- is a concretization of BiCePS.
- specific to models with linear constraints and objective function.

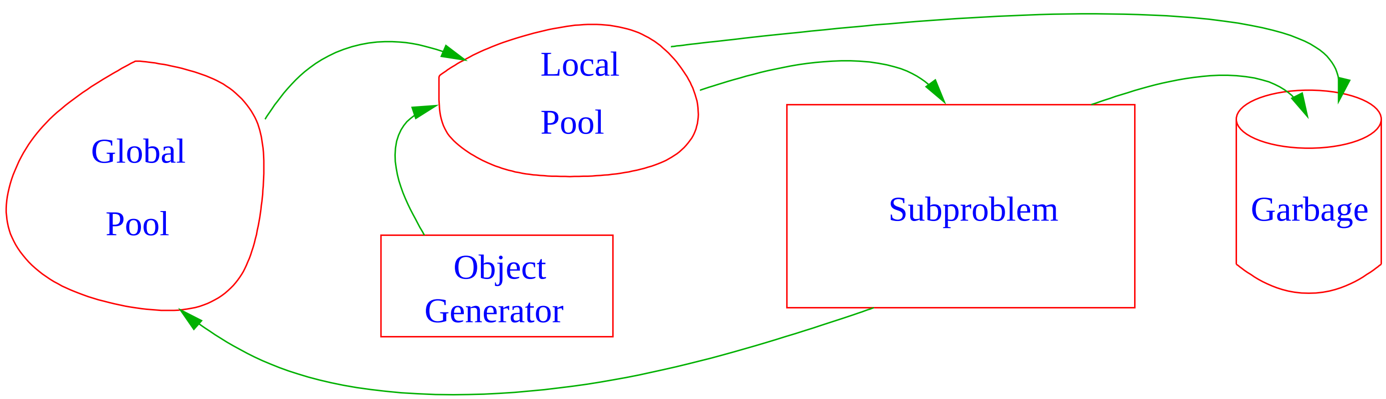


BiCePS: Data-intensive Applications

- In applications such as *branch, cut, and price* (BCP), the amount of information needed to describe each search tree node is very large.
- This can make memory an issue and also increase communication overhead.
- We can think of each node as being described by a list of *objects*, i.e., constraints and variables.
- All objects have a doman and can be treated symmetrically.
- These objects can be generated throughout the search process.
- In BCP, the set of objects may not change much from parent to children.
- We can therefore store the description of an entire subtree very compactly using *differencing*.

BiCePS: Objects

- BiCePS assumes an iterative bounding scheme.
- Each iteration, objects and can be stored in object pools.
- The number of objects can be huge, duplicate and weak objects can be removed based on their hash keys and effectiveness.
- Periodically, invalid and ineffective objects are purged.
- Effectively sharing objects between processes is a challenge.



BLIS: Branch, Cut and Price

- Consider problem P:

$$
\begin{aligned}
min \quad & c^T x && (1)\\
s.t. \quad & Ax \le b && (2)\\
& x_i \in \mathbb{Z} \quad \forall\, i \in I && (3)
\end{aligned}
$$

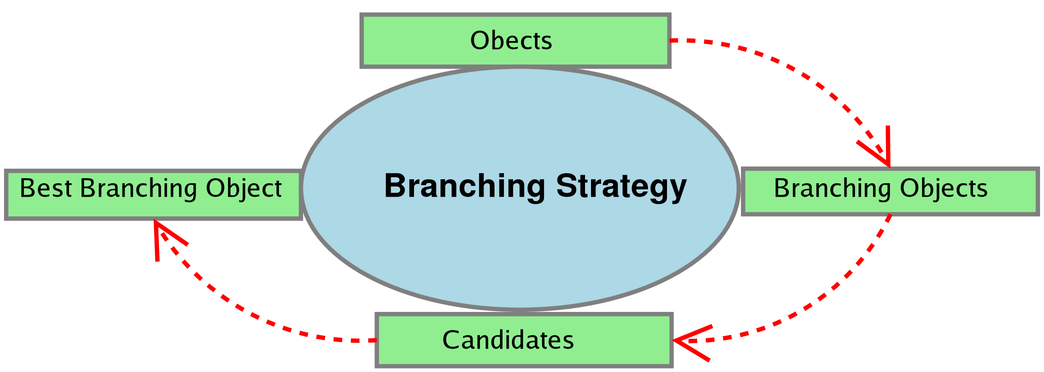
where $(A, b) \in \mathbb{R}^{m\times(n+1)}, c \in \mathbb{R}^n$.

- Basic Algorithmic Elements:
  - bounding method.
  - branching scheme.
  - object generators.
  - heuristics.

BLIS: Branching scheme

BLIS Branching scheme comprise three components:

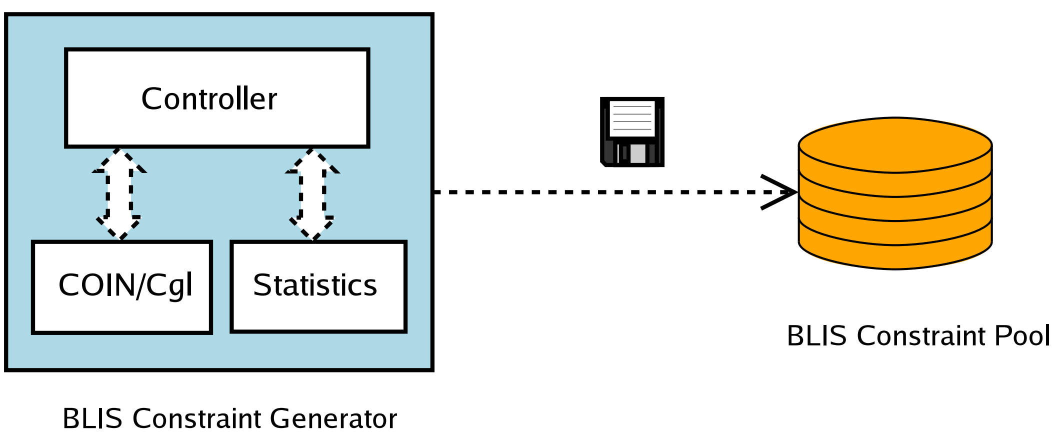
- Object: has feasible region and can be branched on.
- Branching Object:
  - is created from an infeasible object.
  - contains instructions for how to conduct branching.
- Branching strategy:
  - specifies how to create a set of candidate branching objects.
  - has the method to compare objects and choose the best one.



BLIS: Constraint generators

BLIS constraint generator:

- provides an interface between BLIS and the algorithms in COIN/Cgl.
- 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 heuristic:

- Defines the functionality to search for solutions.
- Has the ability to specify rules to control heuristics.
  - where to call: after bounding, at solution?
  - how often to call?
  - when to activate or disable?
- Collects statistics to guide searching.
- Provides a base class for deriving varous heuristics.

BLIS: Prelimiary Computational Results

Test Machine: PC, 2.8 GHz Pentium, 2.0G RAM, Linux

Test instances: Select 33 instances from Lehigh/CORAL and Miplib3, which both solvers can solve in 10 minutes.

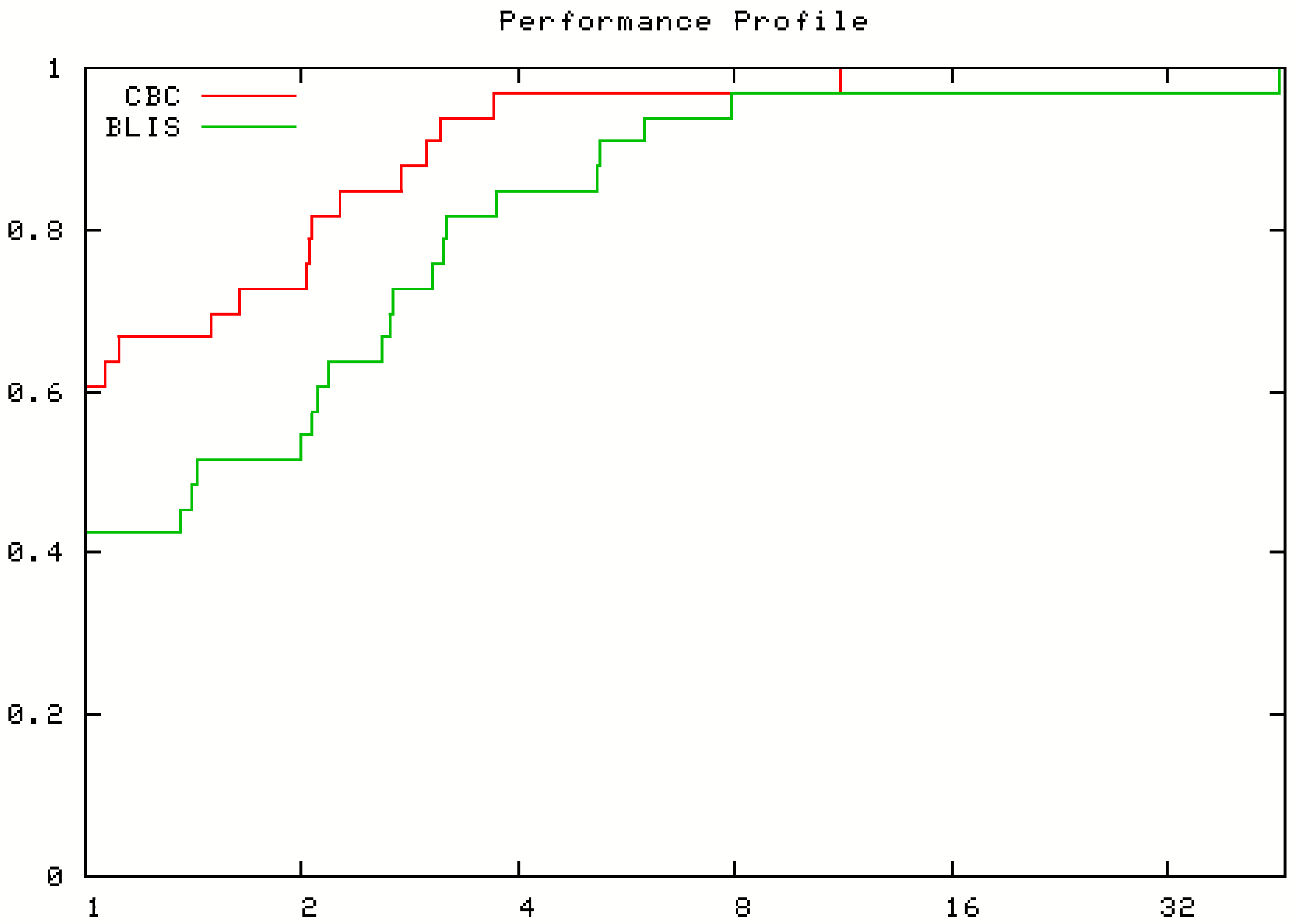
Solver settings:

- BLIS
  - Branching strategy: Pseudocost branching.
  - Cuts generators: Gomory, Knapsack, Flow Cover, MIR, Probing, and Clique.
  - Heuristics: Rounding.
- COIN/Cbc
  - Branching strategy: Strong branching.
  - Cut generators: Gomory, Knapsack, Flow Cover, MIR, Probing, and Clique.
  - Heuristics: Rounding and Local search.

BLIS: Prelimiary Computational Results

Problem	Row	Column	Nonzero	Time-BLIS	Time-CBC
22433	198	429	3408	95	37.59
23588	137	368	3701	118.75	108.75
air03	124	10757	91028	36.45	7.84
aligninq	340	1831	15734	356.76	181.22
bell3a	123	133	347	49.21	38.49
dcmulti	290	548	1315	18.68	13.84
dsbmip	1182	1886	7366	55.3	38.66
...	...	...	...	...	...
qnet1	503	1541	4622	20.17	41.8
rgn	24	180	460	20.06	62.61
roy	162	149	411	23.79	7.56
stein27	118	27	378	25.15	9.78
vpm1	234	378	749	1.45	16.24
TOTAL				1642.61	1238.32

BLIS: Prelimiary Computational Results



Future work

- Improve ALPS:
  - Reduce ramp-up time when node processing time is long.
  - More effectively adjust parameters based on problem stuctures and searching progress.
- Complete the development of BiCePS and BLIS.
  - Finish parallel parts of the code.
  - Find answers to important research questions:
    * How to share objects?
    * How to efficiently avoid duplicated knowledge?
    * How to deal with locally valid knowledge?
  - Add more customization features akin to COIN/BCP:
    * Branch and price.
    * Branch, cut, and price.