# Implementing Custom Applications with CHiPPS

TED RALPHS
LEHIGH UNIVERSITY
YAN XU
SAS INSTITUTE
LASZLO LADÁNYI
IBM ILOG
MATTHEW SALTZMAN
CLEMSON UNIVERSITY

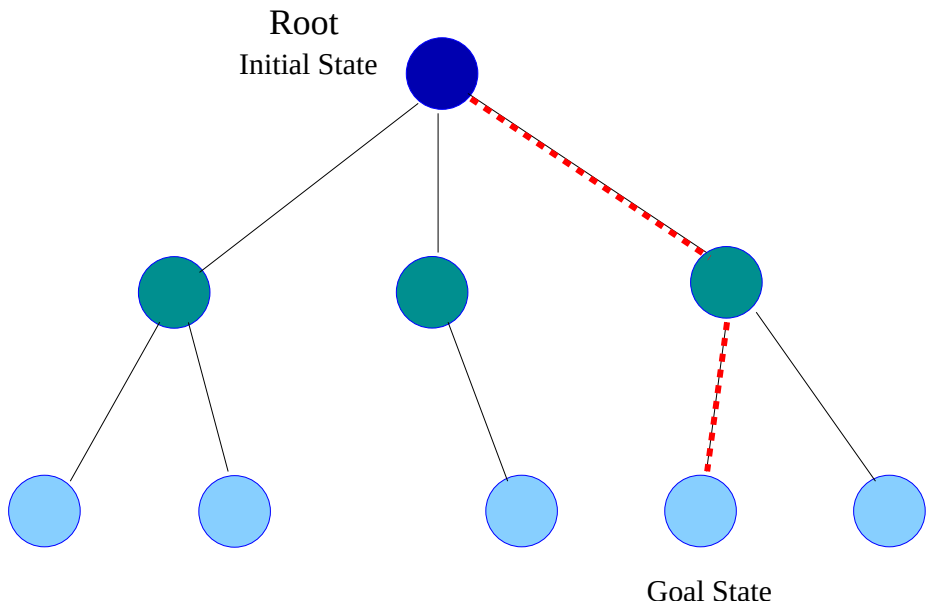

INFORMS Annual Conference, San Diego, CA
October 12, 2009

Thanks: Work supported in part by the National Science Foundation and IBM

# Outline

# Tree Search Algorithms

- Tree search algorithms systematically search the nodes of an acyclic graph for certain *goal nodes*.




- Tree search algorithms have been applied in many areas such as
  - Constraint satisfaction,
  - Game search,
  - Artificial intelligence, and
  - Mathematical programming.

# Elements of Tree Search Algorithms

- A generic tree search algorithm consists of the following elements:

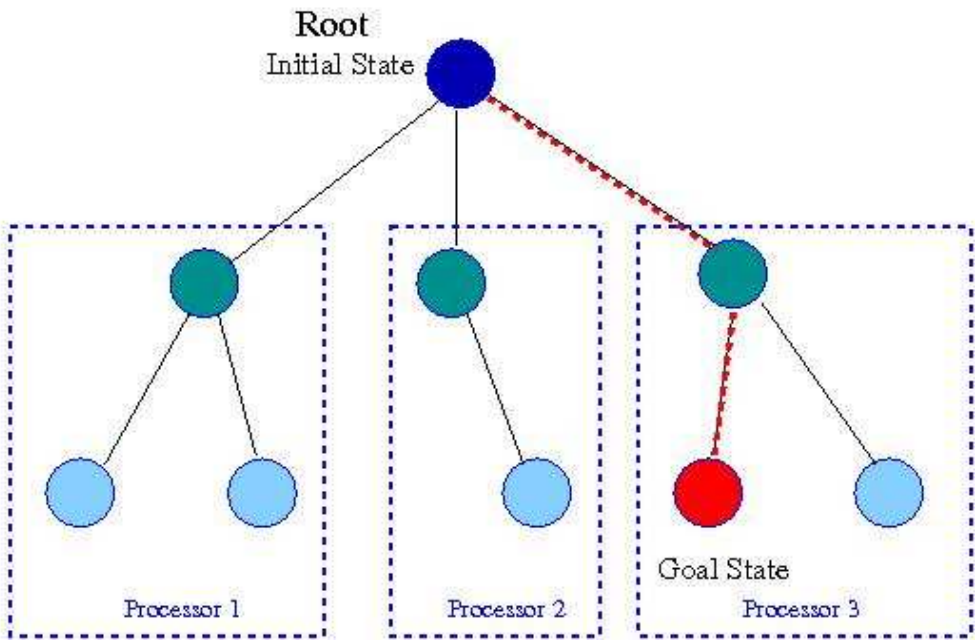
**Generic Tree Search Algorithm**

- Processing method: Is this a goal node?
- Fathoming rule: Can node can be fathomed?
- Branching method: What are the successors of this node?
- Search strategy: What should we work on next?

- 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 can be used to guide the search.

# Parallelizing Tree Search Algorithms

- In general, the search tree can be very large.
- Fortunately, the generic algorithm appears very easy to parallelize.




- The appearance is deceiving, as the search graph is not generally known a priori and naïve parallelization strategies are not generally effective.

# Previous Work

Previous tree search codes:

- Game tree search: ZUGZWANG and APHID
- Constraint programming: ECLiPSe, etc.
- Optimization:
  - Commercial: CPLEX, Lindo, Mosek, SAS/OR, Xpress, etc.
  - Serial: ABACUS, bc-opt, COIN/CBC, GLPK, MINTO, SCIP, etc.
  - Parallel: COIN/BCP, FATCOP, PARINO, PICO, SYMPHONY, Gurobi, XPress, etc.
- Parallelization of tree search have become an increasingly important and mainstream undertaking.
- *All tree search codes will have to be parallelized to stay competitive*.

# 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

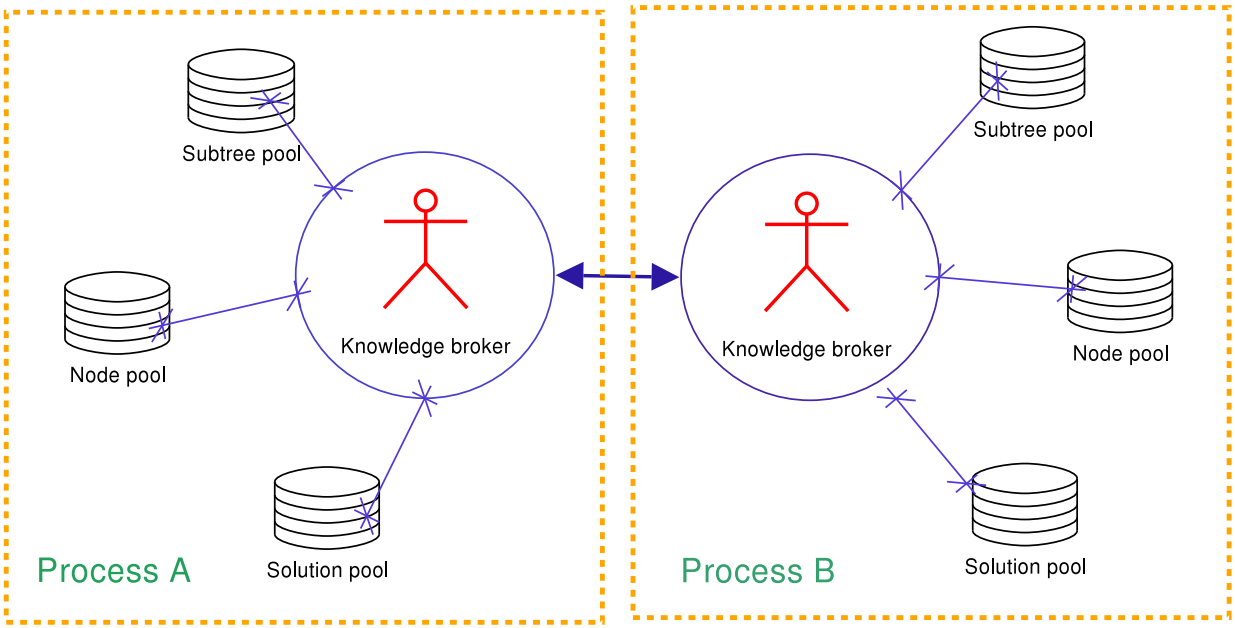
- Intuitive object-oriented class structure.
  - `AlpsModel`
  - `AlpsTreeNode`
  - `AlpsNodeDesc`
  - `AlpsSolution`
  - `AlpsParameterSet`
- Minimal algorithmic assumptions in the base class.
  - Support for a wide range of problem classes and algorithms.
  - Support for constraint programming.
- Easy 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

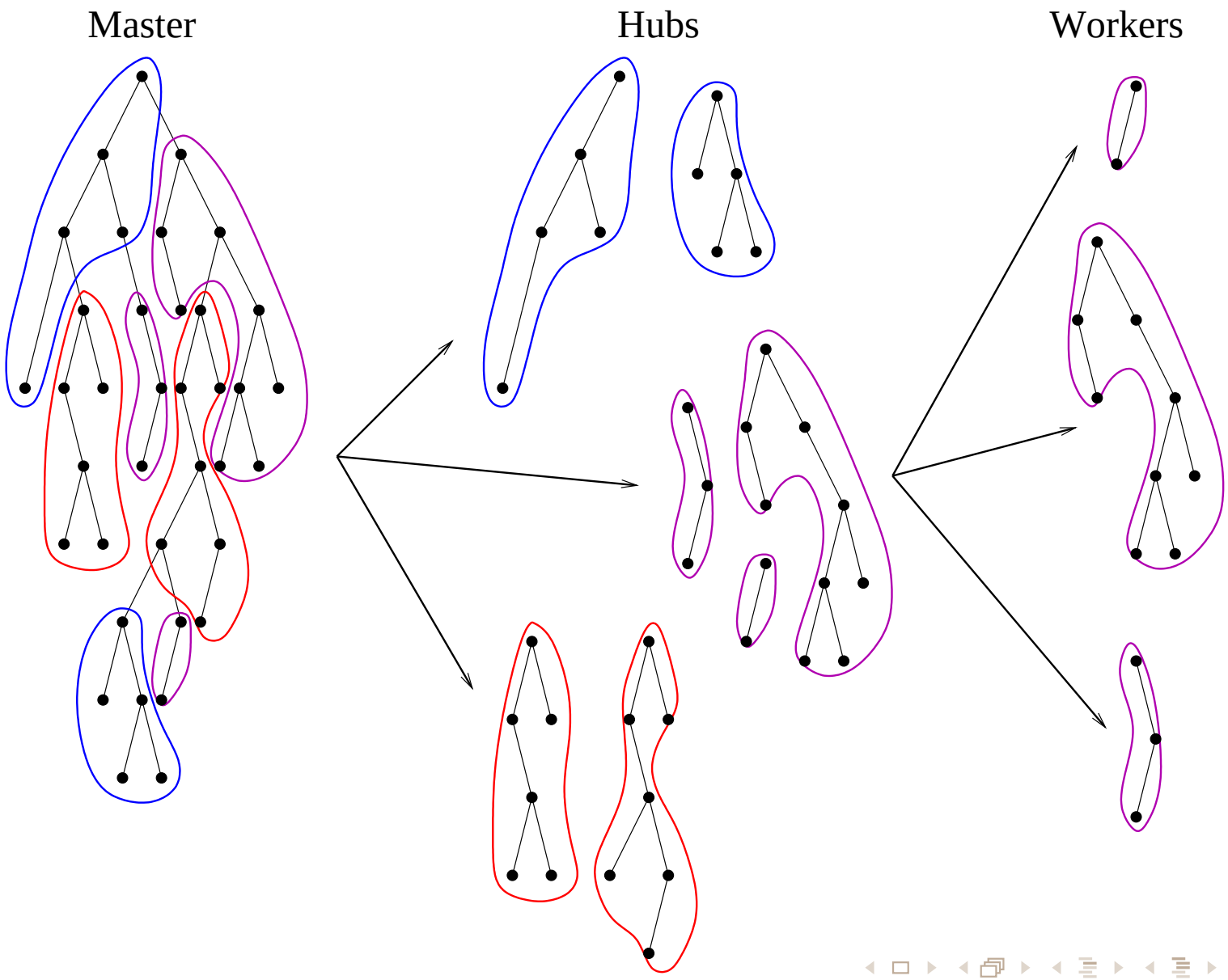
- The design is based on a very general concept of *knowledge*.
- Knowledge is shared asynchronously through *pools* and *brokers*.
- Management overhead is reduced with the *master-hub-worker* paradigm.
- Overhead is decreased using dynamic task granularity.
- Two static load balancing techniques are 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.

# ALPS: Knowledge Sharing

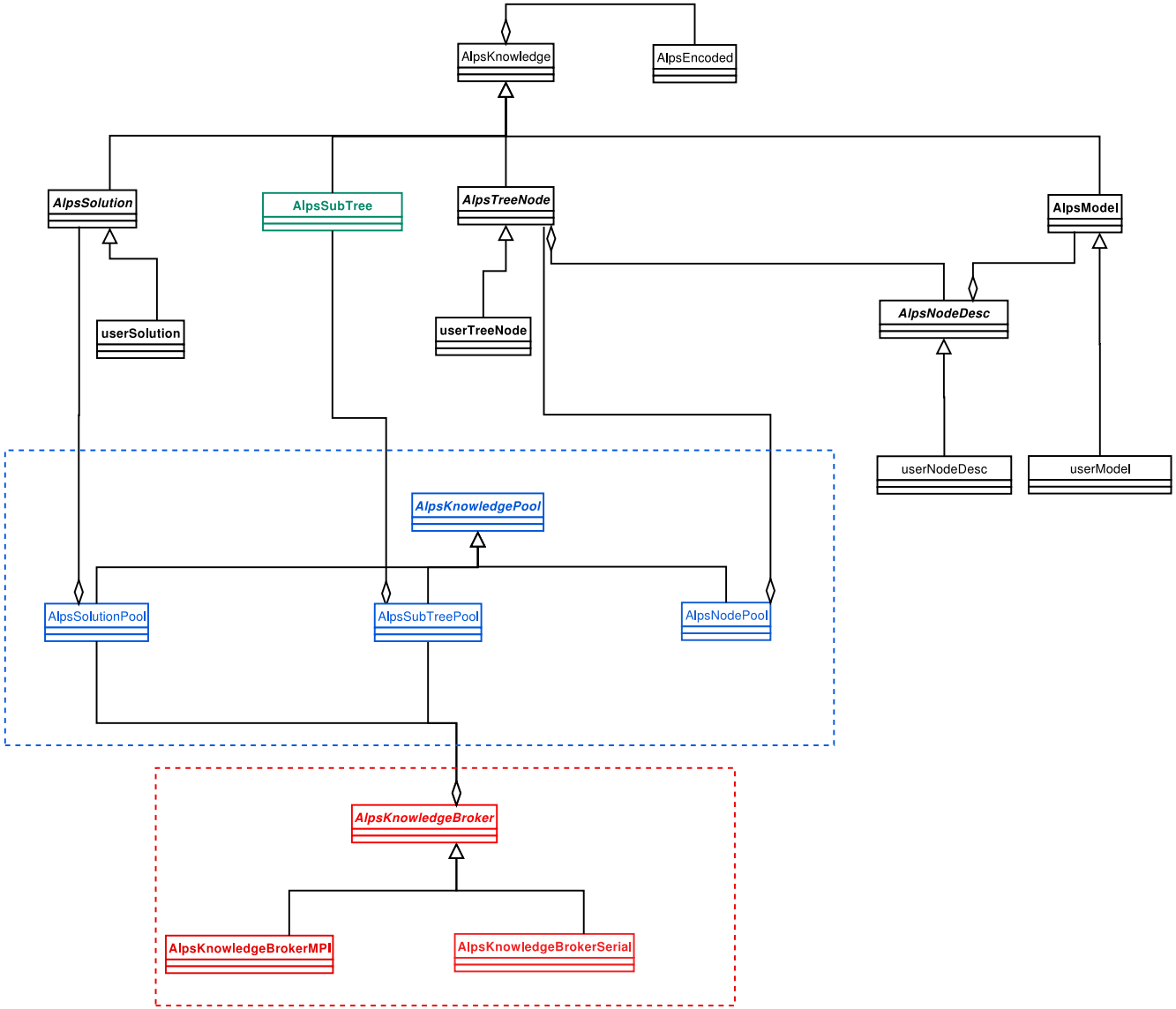
- All knowledge to be shared is derived from a single base class and has an associated *encoded form*.
- Encoded form is used for identification, storage, and communication.
- Knowledge is maintained by one or more *knowledge pools*.
- The knowledge pools communicate through *knowledge brokers*.



# ALPS: Master-Hub-Worker Paradigm



# ALPS: Class Hierarchy



# 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.

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

```
template<class T>
struct BcpsFieldListMod
{
    bool relative;
    int  numModify;
    int  *posModify;
    T    *entries;
};
```

# BLIS: A Generic Solver for MILP

## MILP

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

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

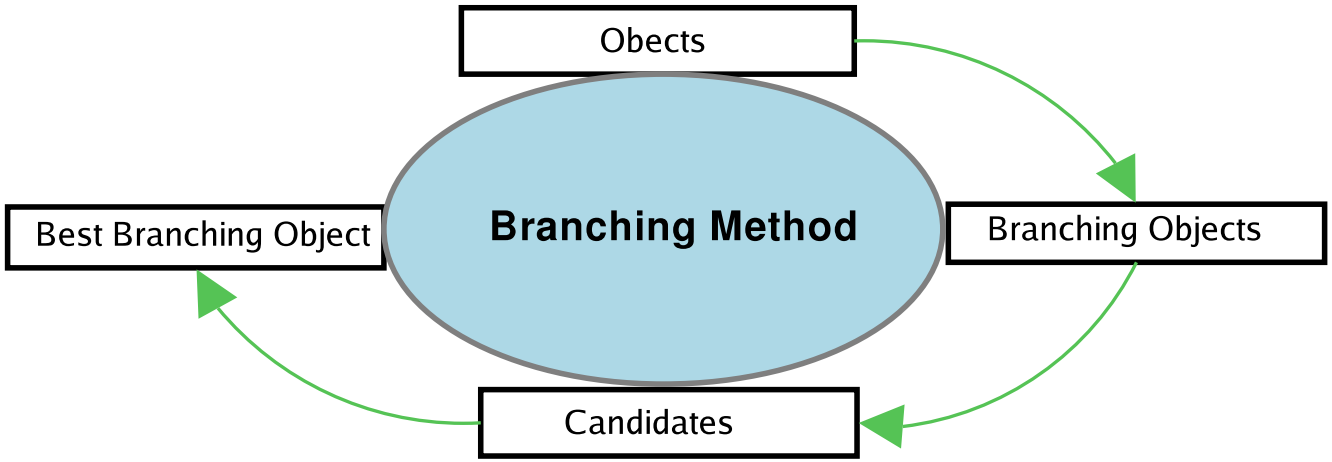
## Basic Algorithmic Components

- Bounding method.
- Branching scheme.
- Object generators.
- Heuristics.

# BLIS: Branching Scheme

BLIS Branching scheme comprise three components:

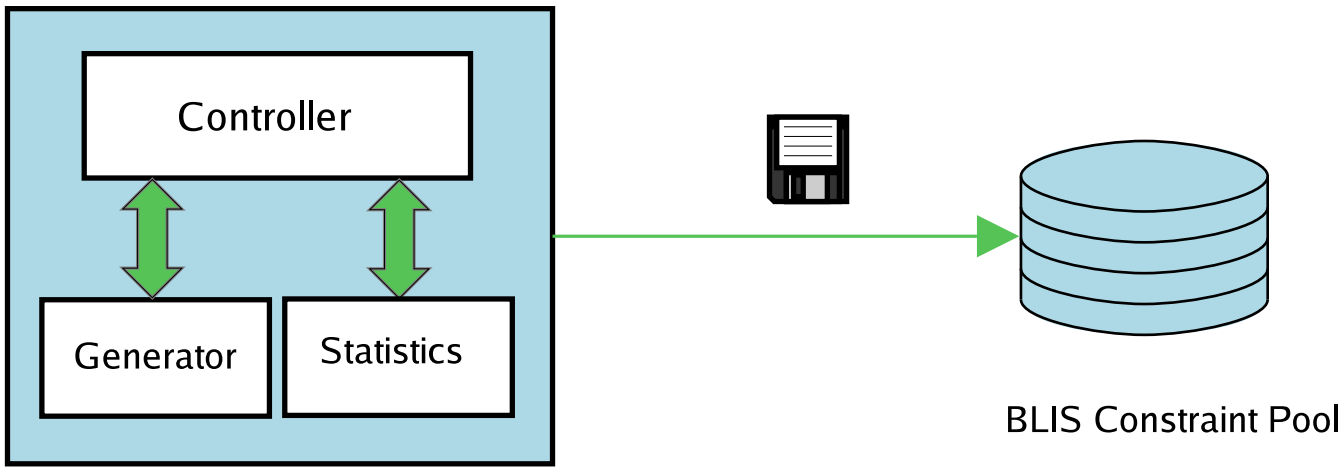
- Object: has feasible region and can be branched on.
- Branching Object:
  - is created from objects that do not lie in they feasible regions or objects that will be beneficial to the search if branching on them.
  - contains instructions for how to conduct branching.
- Branching method:
  - specifies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



# BLIS: Constraint Generators

BLIS constraint generator:

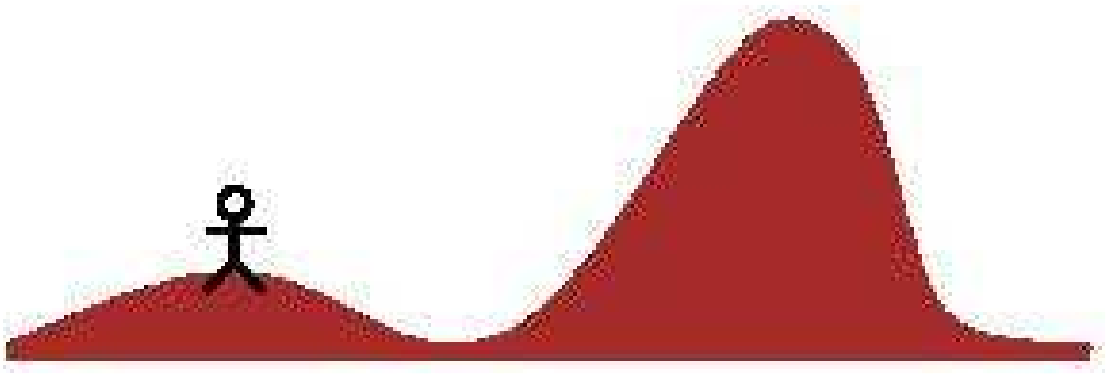
- provides an interface between BLIS and the algorithms in COIN/Cgl.
- provides a base class for deriving specific generators.
- has the ability to specify rules to control generator:
  - where to call: root, leaf?
  - how many to generate?
  - when to activate or disable?
- contains the statistics to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



# ALPS Applications

- Relatively simple tree search algorithms can be implemented directly from ALPS.
- The user must derive a few classes to specify the algorithm and model.
  - `AlpsModel`: Implement `readInstance()` and store problem data.
  - `AlpsTreeNode`: Implement `process()` and `branch()`.
  - `AlpsNodeDesc`: Store branching information.
  - `AlpsSolution`: Strore solution data.
  - `AlpsParameterSet`: Define parameters (optional)
- To use the parallel mode, the user must also define `pack()` and `unpack()` methods for each knowledge type (easy with provided utilities).
- After deriving classes, use the resulting libray to implement an application.

# Using ALPS: A Knapsack Solver

The formulation of the binary knapsack problem is

$$\max\{\sum_{i=1}^{m} p_i x_i : \sum_{i=1}^{m} s_i x_i \leq c, x_i \in \{0,1\}, i = 1,2,\ldots,m\}, \tag{4}$$

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).

# Using ALPS: Knapsack Solver Main Function

Then, supply the main function.

```
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.registerClass(AlpsKnowledgeTypeModel, new KnapModel);
    broker.registerClass(AlpsKnowledgeTypeSolution,
                         new KnapSolution(&model));
    broker.registerClass(AlpsKnowledgeTypeNode,
                         new KnapTreeNode(&model));
    AlpsTreeNode* root = new KnapTreeNode(&model);
    broker.search(root);
    broker.printResult();
    return 0;
}
```

# Computational Experiments with KNAP

- Randomly generated 26 *difficult* Knapsack instances based on the rule proposed by Martello ('90).
- Tested on the SDSC Blue Gene System (Linux, MPICH, 700MHz Dual Processor, 512 MB RAM).

Table: Scalability for solving Difficult Knapsack Instances

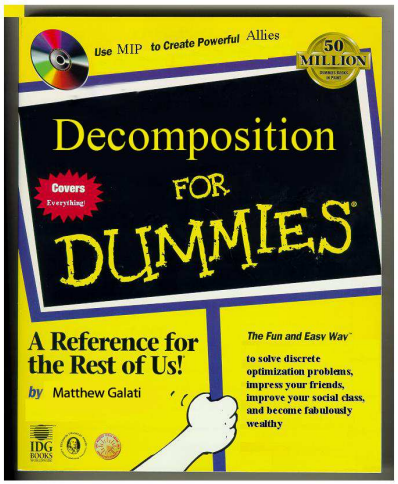
| P | Node | Ramp-up | Idle | Ramp-down | Wallclock | Eff |
|---|---|---|---|---|---|---|
| 64 | 14733745123 | 0.69% | 4.78% | 2.65% | 6296.49 | 1.00 |
| 128 | 14776745744 | 1.37% | 6.57% | 5.26% | 3290.56 | 0.95 |
| 256 | 14039728320 | 2.50% | 7.14% | 9.97% | 1672.85 | 0.94 |
| 512 | 13533948496 | 7.38% | 4.30% | 14.83% | 877.54 | 0.90 |
| 1024 | 13596979694 | 13.33% | 3.41% | 16.14% | 469.78 | 0.84 |
| 2048 | 14045428590 | 9.59% | 3.54% | 22.00% | 256.22 | 0.77 |

# ALPS Application: DECOMP

## DECOMP Framework

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.

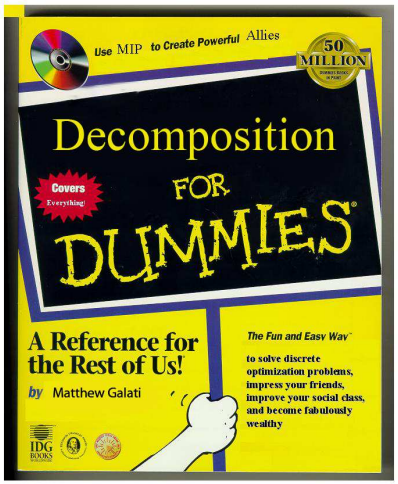


# ALPS Application: DECOMP

## DECOMP Framework

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.

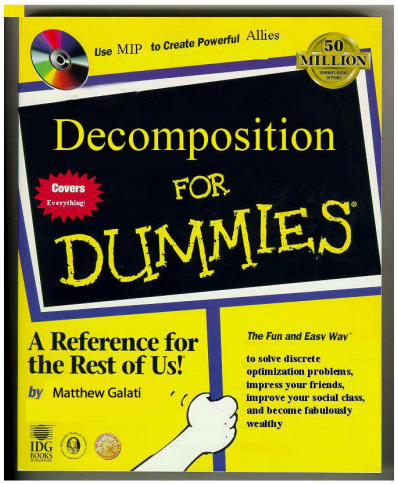


# ALPS Application: DECOMP

## DECOMP Framework

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: Bounding Methods

- Decomposition methods
  - Cutting Plane Method (CPM) dynamically builds an *outer* approximation of the feasible region of a given relaxation. $\mathcal{Q}''$.
  - Dantzig-Wolfe Method (DW) builds an *inner* approximation of the feasible region of a given relaxation.
  - Lagrangian Method (LD) obtains a bound by solving a penalized version of the relaxation.
- These three methods can be seen as equivalent.
- The user has only to define the relaxation to get implementations of all three, plus hybrids.
- DECOMP can be inserted as a bounding method to drive a branch-and-bound built in ALPS.

# DECOMP: Implementation Overview

- 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: Implementation Overview

- 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`

# BLIS Applications: VRP Formulation

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

$$\sum_{e=\{0,j\} \in E} x_e = 2k, \tag{5}$$

$$\sum_{e=\{i,j\} \in E} x_e = 2 \ \ \forall i \in N, \tag{6}$$

$$\sum_{\substack{e=\{i,j\} \in E \\ i \in S, j \notin S}} x_e \geq 2b(S) \ \ \forall S \subset N, \ |S| > 1, \tag{7}$$

$$0 \leq x_e \leq 1 \ \ \forall e = \{i,j\} \in E, \ i,j \neq 0, \tag{8}$$

$$0 \leq x_e \leq 2 \ \ \forall e = \{i,j\} \in E, \tag{9}$$

$$x_e \in \mathbb{Z} \ \ \forall e \in E. \tag{10}$$

# BLIS Applications: VRP Implementation

- 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`).
- The cut generator in this case is for the Generalized Subtour Elimination Constraints.
- Note that we derive our own variable class so that we can store a decription of the variables, i.e., the locations the edge connects.
- We now also have our own heuristic.
- Then, we write a `main` function as before.

# Computational Results with VRP

- Select 16 VRP instances from public sources (Ralphs,'03)
- Tested on a Clemson Cluster (Linux, MPICH, 1654 MHz Dual Core, 4G RAM).

| P | Nodes | Ramp-up | Idle | Ramp-down | Wallclock | Eff |
|---|---|---|---|---|---|---|
| 1 | 40250 | – | – | – | 19543.46 | 1.00 |
| 4 | 36200 | 7.06% | 7.96% | 0.39% | 5402.95 | 0.90 |
| 8 | 52709 | 9.88% | 6.15% | 1.29% | 4389.62 | 0.56 |
| 16 | 70865 | 14.16% | 8.81% | 3.76% | 3332.52 | 0.37 |
| 32 | 96160 | 15.85% | 10.75% | 16.91% | 3092.20 | 0.20 |
| 64 | 163545 | 18.19% | 10.65% | 19.02% | 2767.83 | 0.11 |

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.

# BLIS Applications: MiBS

The Mixed Integer Bilevel Solver (MibS) has been developed to solve bilevel integer programming problems.

## COIN-OR Components Used

- The COIN High Performance Parallel Search Framework (CHiPPS) to manage the branch and bound.
- The COIN Branch and Cut (CBC) framework for solving MILPs.
- The COIN LP Solver (CLP) framework for solving LPs.
- The Cut Generation Library (CGL) for generating valid inequalities.
- The Open Solver Interface (OSI) for interfacing with CBC and CLP.

# BLIS Applications: MiBS Implementation

- As before, the main effort is in deriving a few classes to specify the algorithm.
  - `MibSModel`
  - `MibSSolution`
  - `MibSCutGenerator`
  - `MibSVariable`
  - `MibSParams`
- Once the classes have been implemented, the user writes a `main` function as before.
- Note that all the power of the generic elements of the MIP solver are available.

# Computational Results with MiBS: Knapsack Interdiction

- Below are results for knapsack interdiction problems.
- These use a special class of cuts that applies only to interdiction problems.
- There is also a heuristic.
- Instances come from the MCDM library.

| | Maximum Infeasibility | | | Strong Branching | | |
|---|---|---|---|---|---|---|
| $2n$ | Avg Nodes | Avg Depth | Avg CPU (s) | Avg Nodes | Avg Depth | Avg CPU (s) |
| 20 | 359.30 | 8.65 | 9.32 | 358.30 | 8.65 | 11.07 |
| 22 | 658.40 | 9.85 | 18.50 | 658.20 | 9.85 | 18.92 |
| 24 | 1414.80 | 10.85 | 46.03 | 1410.80 | 10.75 | 46.46 |
| 26 | 2725.00 | 12.05 | 97.55 | 2723.50 | 12.05 | 100.17 |
| 28 | 5326.40 | 12.90 | 214.97 | 5328.60 | 12.95 | 220.26 |
| 30 | 10625.00 | 14.05 | 482.70 | 10638.00 | 14.10 | 538.32 |

# Computational Results with MiBS: Assignment Interdiction

The setup here is similar, but we have assignment interdiction instead of knapsack.

| Instance | Nodes | Depth | CPU (s) | Instance | Nodes | Depth | CPU (s) |
|---|---|---|---|---|---|---|---|
| 2AP05-1 | 6203 | 33 | 290.25 | 2AP05-14 | 3173 | 28 | 261.08 |
| 2AP05-2 | 3881 | 32 | 384.97 | 2AP05-15 | 2509 | 32 | 127.05 |
| 2AP05-3 | 3909 | 32 | 205.93 | 2AP05-16 | 1699 | 29 | 44.61 |
| 2AP05-4 | 2441 | 36 | 102.66 | 2AP05-17 | 5417 | 29 | 201.34 |
| 2AP05-5 | 3505 | 33 | 119.18 | 2AP05-18 | 5785 | 32 | 176.67 |
| 2AP05-6 | 2031 | 35 | 80.31 | 2AP05-19 | 2259 | 32 | 79.70 |
| 2AP05-7 | 2957 | 29 | 153.02 | 2AP05-20 | 2585 | 31 | 77.35 |
| 2AP05-8 | 3549 | 32 | 224.77 | 2AP05-21 | 6039 | 33 | 161.44 |
| 2AP05-9 | 2271 | 33 | 111.13 | 2AP05-22 | 2479 | 29 | 48.06 |
| 2AP05-10 | 3299 | 31 | 211.07 | 2AP05-23 | 1519 | 25 | 49.40 |
| 2AP05-11 | 707 | 33 | 35.13 | 2AP05-24 | 15 | 5 | 1.32 |
| 2AP05-12 | 407 | 18 | 29.51 | 2AP05-25 | 3857 | 31 | 115.97 |
| 2AP05-13 | 391 | 18 | 23.80 | | | | |

The CHiPPS framework is available at

`https://projects.coin-or.org/CHiPPS`

# Questions? & Thank You!