Parallel Integer Programming with ALPS

YAN XU
SAS INSTITUTE
TED RALPHS
LEHIGH UNIVERSITY
LASZLO LADÁNYI
IBM T.J. WATSON RESEARCH CENTER
MATTHEW SALTZMAN
CLEMSON UNIVERSITY

COR@L
COMPUTATIONAL OPTIMIZATION RESEARCH AT LEHIGH

ISMP 2006, Rio de Janeiro, Brazil

August 3, 2006

Outline

Overview

- ALPS is a C++ class library for implementing parallel tree search.
- ALPS is an on-going open source software project originally developed in partnership with the COIN-OR Foundation, IBM, and NSF.
- ALPS is an attempt to further improve on existing solvers and frameworks in a number of ways.

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 *parallel scalability*.
- Explicitly supports *data-intensive* algorithms.
- Operates effectively in both *parallel* and *sequential* environments.

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, but the graph may not be known a priori.
- A generic tree search algorithm consists of the following elements:

Generic Tree Search Algorithm

- 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*.

Components of Parallel Overhead in Tree Search

- Communication Overhead (cost of sharing knowledge)
- Idle Time
  - Handshaking/Synchronization (cost of sharing knowledge)
  - Task Starvation (cost of *not* sharing knowledge)
  - Ramp Up Time
  - 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.

Illustrating Overhead and Scalability

Results solving MIPLIB instances with SYMPHONY 5.1

	Tree Size	Ramp Up	Ramp Down	Comm	Hand−shake	CPU	Wallclock	Eff
1 NP	220846	0.00	0.00	163.45	465.32	18280.70	19173.94	1.00
Per Node		0.0000	0.0000	0.0007	0.0021	0.0828	0.0868	
2 NP′s	224266	18.41	0.01	166.45	484.70	18357.87	9697.61	0.99
Per Node		0.0001	0.0000	0.0007	0.0022	0.0819	0.0865	
4 NP′s	222446	60.29	2.46	161.70	529.36	17822.92	4737.28	1.02
Per Node		0.0003	0.0000	0.0007	0.0024	0.0801	0.0852	
8 NP′s	213954	163.26	127.87	139.33	608.71	17433.65	2395.28	1.01
Per Node		0.0008	0.0006	0.0007	0.0029	0.0815	0.0896	
16 NP′s	215597	393.70	605.28	128.37	1006.22	17127.50	1277.09	0.94
Per Node		0.0018	0.0028	0.0006	0.0047	0.0794	0.0948	
32 NP′s	212672	911.73	2282.13	148.22	4235.01	16723.89	794.87	0.75
Per Node		0.0043	0.0107	0.0007	0.0200	0.0786	0.1196	

ALPS: Feature Overview

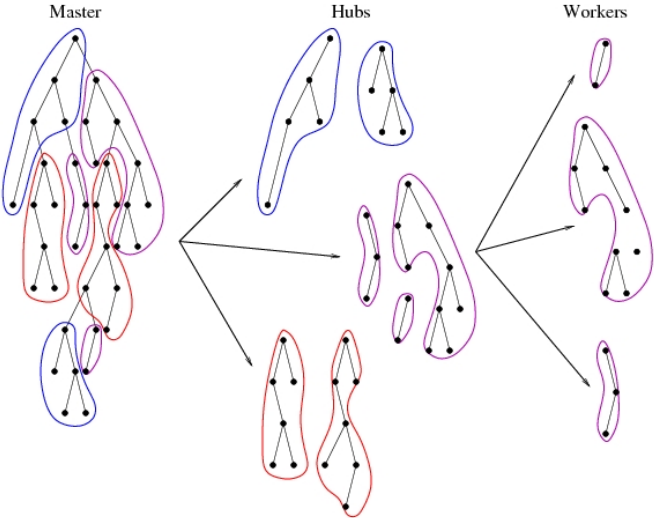
Generality

- ALPS only assumes that the graph to be searched is acyclic.
- The implementation is based on a very general concept of knowledge.

Scalability

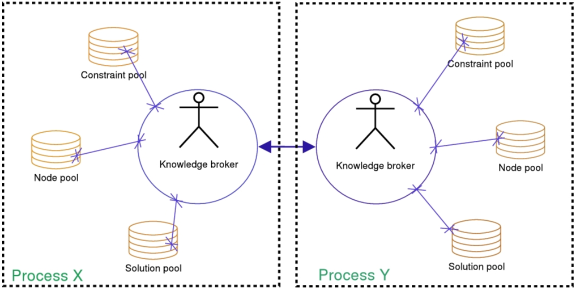
- Management overhead is reduced with the *master-hub-worker* paradigm.
- Knowledge is shared asynchronously through *pools* and *brokers*.
- Overhead is decreased using dynamic task granularity.
- Static and dynamic load balancing techniques are employed.
- Tasks are managed locally by a task scheduler.

ALPS: Master-Hub-Worker Paradigm



ALPS: Knowledge Sharing

- All knowledge to be shared is derived from the base class `AlpsKnowledge` and has an associated *encoded form*.
- The encoded form is used for identification, storage, and communication.
- `AlpsKnowledge` is maintained by one or more `AlpsKnowledgePool`s.
- The knowledge pools communicate through `AlpsKnowledgeBroker`s.



ALPS: Asynchronism

- A primary design goal for ALPS is to eliminate handshaking and synchronicity.
- Knowledge brokers can work completely asynchronously, as long as their local node pool is not empty.
- This asynchronism can result in an increase in the performance of redundant work.
- The task granularity also affects the degree of synchronism and is adjusted dynamically.
- To combat this, we need good load balancing.

ALPS: Load Balancing

Load balancing is a specific type of knowledge sharing in which tasks are distributed or redistributed.

Static load balancing

- Determines the initial task distribution.
- In dynamic search algorithms, this can be difficult.
- This is the main source of ramp up overhead.

Dynamic load balancing

- Used periodically to redistribute the useful work.
- Work must be balanced both by quantity and quality.
- Donors and receivers are matched at both the hub and master level.
- A unique aspect of load balancing in ALPS is that subtrees are kept together in order to enable differencing.

ALPS: How to Develop an Application

- The first step is deriving a few classes to specify the algorithm.

User-written Classes

- `AlpsModel`
- `AlpsTreeNode`
- `AlpsNodeDesc`
- `AlpsSolution`

- Once the classes have been implemented, the user writes a `main` function.

Sample Applications

- Knapsack problem
- ALPS branch and cut (ABC)
- Mixed-integer Stackelberg games (LBMZZSSPIC)

ALPS: Sample main() Function

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

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

    broker.registerClass("MODEL", new UserModel);
    broker.registerClass("SOLUTION", new UserSolution);
    broker.registerClass("NODE", new UserTreeNode);

    broker.search();
    broker.printResult();
    return 0;
}
```

ALPS: Preliminary Experiments

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

Experimental Design

- We generated *ten hard knapsack instances* based on the rule proposed by Martello ('90).
- We ran three trials for each instance, and take the average.
- Some default parameters:
  - Two hubs were used when for runs with 16 processes or more.
  - Dynamic load balancing was activated.
  - The hubs did not process subproblems.

ALPS: Computational Results

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

P	Nodes	Time	Starvation	Ramp-up	Ramp-down	Eff
1	$>$190m	$>$7200.00	–	–	–	–
4	190m	1401.16	0.00	0.01	53.00	1.00
8	190m	639.19	0.00	0.01	46.27	1.10
16	190m	361.80	0.00	0.01	23.57	0.97
32	190m	191.50	0.00	0.01	15.00	0.92

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

- Serial code could not solve several instance due to lack of memory.
- These indicate reasonable scalability.
- Starvation and ramp-up overhead is negligible.
- Ramp-down overhead has room to decrease.

Library Hierarchy for Optimization

ALPS (Abstract Library for Parallel Search)

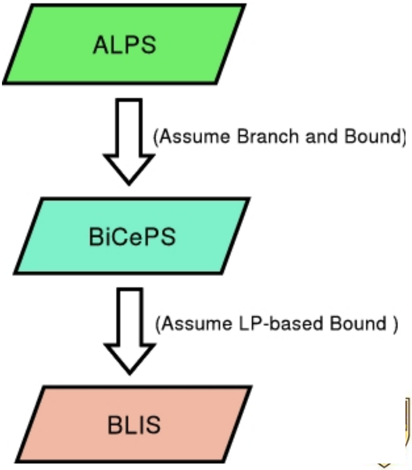
- is the search-handling layer.
- prioritizes nodes based on quality.

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.

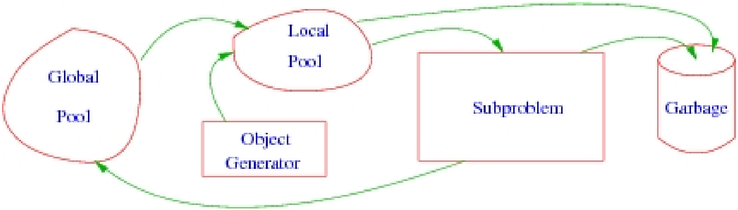


BiCePS: Data-intensive Applications

- 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 domain 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 are generated and either stored in pools or used to augment the current relaxation.
- The number of objects can be huge, so duplicate and weak objects can be removed based on their hash keys and their effectiveness.
- Periodically, invalid and ineffective objects are purged.
- Effectively sharing objects is a challenge.



BLIS: Branch, Cut, and Price

Problem P

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

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

Basic Algorithmic Elements

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

BLIS: Branching scheme

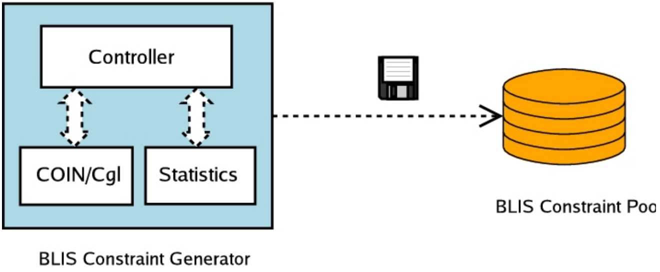
BLIS Branching scheme comprise three components:

- 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 primal 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 various heuristics.

BLIS: Preliminary Experiments

To benchmark, we compare BLIS with COIN/Cbc.

Test Bed

- Test Machine: PC, 2.8 GHz Pentium, 2.0G RAM, Linux
- Test instances: Selected 33 instances from Lehigh/CORAL and MIPLIB 3, which both solvers can solve in 10 minutes.

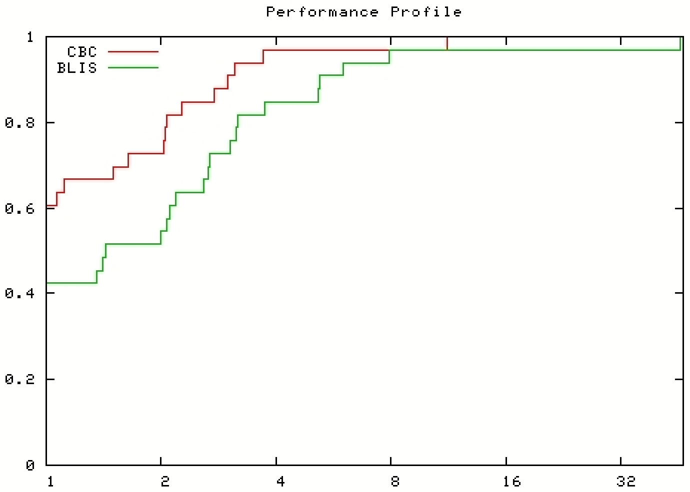
Solver settings

- 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: 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: Comparing with Cbc



What's Available

- ALPS itself is available for download at

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

- There is a user's manual, but it is slightly out of date.
- There are several sample applications that come with ALPS.
- The base library will build and run on most platforms with the GNU autotools.
- MPI is required for parallel execution.
- BiCePS and BLIS will be publicly available in a week or two.

Case Study: Mixed-Integer Stackelberg Games

- In this research project, we are studying *mixed-integer Stacklerberg games*.
- This effort required implementation of a solver for bilevel programs.
- We were able to implement such a solver using ALPS and a variety of other COIN-OR projects.

COIN-OR Components Used

- The Abstract Library for Parallel Search (ALPS) to perform the branch and bound.
- The COIN Branch and Cut (CBC) framework for solving the MILPs.
- The COIN LP Solver (CLP) framework for solving the LPs arising in the branch and cut.
- The Cut Generation Library (CGL) for generating cutting planes within CBC.
- The Open Solver Interface (OSI) for interfacing with CBC and CLP.

Future work

Improve ALPS

- Reduce ramp-up/down time when node processing times are long.
- More effectively adjust parameters dynamically based on problem structure and search 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.
- Large-scale testing!