Computational Experience with Parallel Integer Programming using the CHiPPS Framework

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

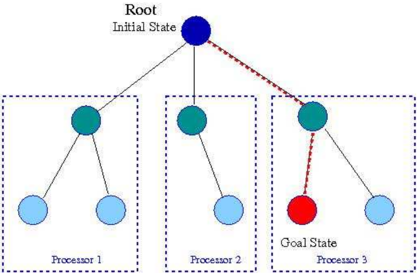
INFORMS Annual Conference, San Diego, CA
October 12, 2009

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

Outline

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.
- 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.

CHiPPS Components

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.

Previous Work

Previous tree search codes:

- Game tree search: ZUGZWANG and APHID
- Constraint programming: ECLiPSe, etc.
- Optimization:
  - Commercial: CPLEX, Lindo, Mosek, SAS/OR, Xpress, Gurobi, etc.
  - Serial: ABACUS, bc-opt, COIN/CBC, GLPK, MINTO, SCIP, etc.
  - Parallel: COIN/BCP, FATCOP, PARINO, PICO, SYMPHONY, CPLEX, 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*.

ALPS: Design Goals

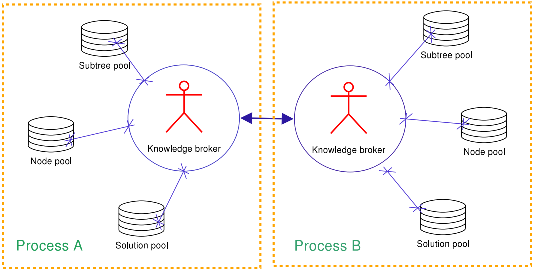
- Intuitive object-oriented class structure.
  - `AlpsModel`
  - `AlpsTreeNode`
  - `AlpsNodeDesc`
  - `AlpsSolution`
  - `AlpsParameterSet`
- Minimal algorithmic assumptions in the base class.
  - Support for a wide range of problem classes and algorithms.
  - Support for constraint programming.
- Easy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

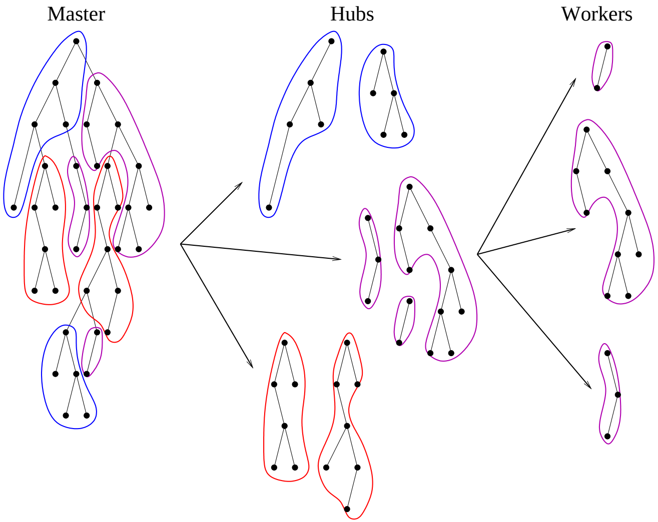
- The design is based on a very general concept of *knowledge*.
- Knowledge is shared asynchronously through *pools* and *brokers*.
- Management overhead is reduced with the *master-hub-worker* paradigm.
- Overhead is decreased using dynamic task granularity.
- Two static load balancing techniques are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

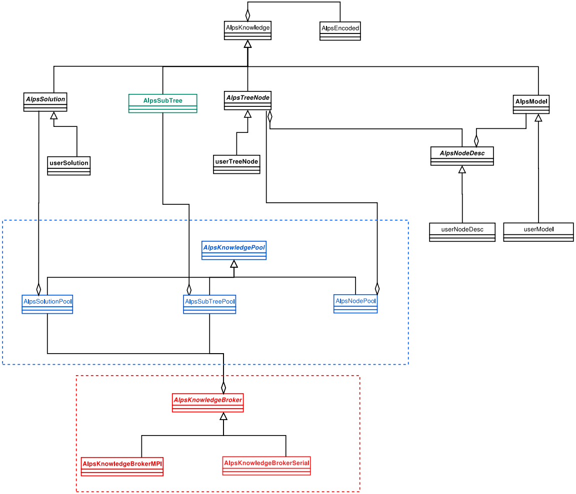
- All knowledge to be shared is derived from a single base class and has an associated *encoded form*.
- Encoded form is used for identification, storage, and communication.
- Knowledge is maintained by one or more *knowledge pools*.
- The knowledge pools communicate through *knowledge brokers*.



Master-Hub-Worker Paradigm



Alps Class Hierarchy



Using ALPS: A Knapsack Solver

The formulation of the binary knapsack problem is

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

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).

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

$$\min \quad c^T x \tag{2}$$
$$s.t. \quad Ax \leq b \tag{3}$$
$$x_i \in \mathbb{Z} \quad \forall\, i \in I \tag{4}$$

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

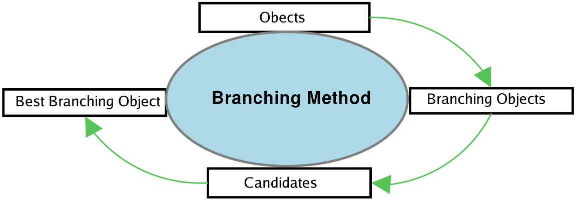
Basic Algorithmic Components

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

BLIS: Branching Scheme

BLIS Branching scheme comprise three components:

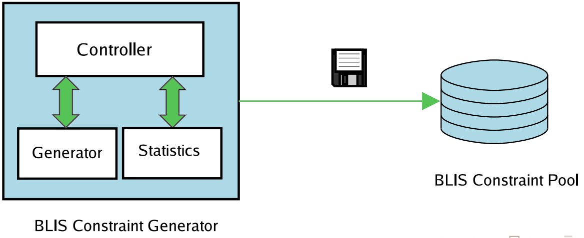
- Object: has feasible region and can be branched on.
- Branching Object:
  - is created from objects that do not lie in they feasible regions or objects that will be beneficial to the search if branching on them.
  - contains instructions for how to conduct branching.
- Branching method:
  - specifies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

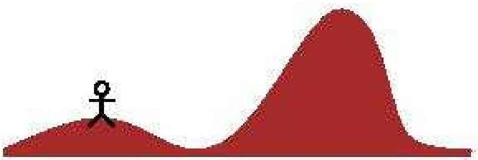
- provides an interface between BLIS and the algorithms in COIN/Cgl.
- provides a base class for deriving specific generators.
- has the ability to specify rules to control generator:
  - where to call: root, leaf?
  - how many to generate?
  - when to activate or disable?
- contains the statistics to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



Traditional Measures of Performance

- *Parallel System*: Parallel algorithm + parallel architecture.
- *Scalability*: How well a parallel system takes advantage of increased computing resources.

Terms

- Sequential runtime: T_s
- Parallel runtime: T_p
- Parallel overhead: $T_o = NT_p - T_s$
- Speedup: $S = T_s/T_p$
- Efficiency: $E = S/N$

- Standard analysis considers change in efficiency on a fixed test set as number of processors is increased.
- *Isoefficiency analysis* considers the increase in problem size to maintain a fixed efficiency as number of processors is increased.

Parallel Overhead

- The amount of *parallel overhead* determines the scalability.

Major 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.

Challenges in Measuring Performance

- Traditional measures are not appropriate.
  - The interesting problems are the ones that take too long to solve sequentially.
  - Need to account for the possibility of failure.
- It's exceedingly difficult to construct a test set
  - Scalability varies substantially by instance.
  - Hard to know what test problems are appropriate.
  - A fixed test set will almost surely fail to measure what you want.
- Results are highly dependent on architecture
  - Difficult to make comparisons
  - Difficult to tune parameters
- Hard to get enough time on large-scale platforms for tuning and testing.
- Results are non-deterministic!
  - Determinism can be a false sense of security.
  - Lack of determinism requires more extensive testing.

Computational Results: Platforms

Clemson Cluster

Machine:	Beowulf cluster with 52 nodes
Node:	dual core PPC, speed 1654 MHz
Memory:	4G RAM each node
Operating System:	Linux
Message Passing:	MPICH

SDSC Blue Gene System

Machine:	IBM Blue Gene with 3,072 compute nodes
Node:	dual processor, speed 700 MHz
Memory:	512 MB RAM each node
Operating System:	Linux
Message Passing:	MPICH

KNAP Scalability for Moderately Difficult Instances

- Tested the moderately difficult instances at Clemson.
- The default algorithm was used except that
  - the static load balancing scheme was the two-level root initialization,
  - the number of nodes generated by the master was 3000, and
  - the size of a unit work was 300 nodes.

P	Node	Ramp-up	Idle	Ramp-down	Wallclock	Eff
4	193057493	0.28%	0.02%	0.01%	586.90	1.00
8	192831731	0.58%	0.08%	0.09%	245.42	1.20
16	192255612	1.20%	0.26%	0.37%	113.43	1.29
32	191967386	2.34%	0.71%	1.47%	56.39	1.30
64	190343944	4.37%	2.27%	5.49%	30.44	1.21

- Super-linear speedup observed.
- Ramp-up, ramp-down, and idle time overhead remains low.

KNAP Scalability for Very Difficult Instances

- Tested difficult instances on SDSC Blue Gene,
- The default algorithm was used except that
  - the static load balancing scheme was the two-level root initialization,
  - nodes generated by the master varies from 10K to 30K.
  - nodes generated by hub varies from 10K to 20K.
  - the size of a unit of work was 3K nodes; and
  - multiple hubs were used.

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	8.33%	3.41%	16.14%	469.78	0.84
2048	14045428590	9.59%	3.54%	22.00%	256.22	0.77

- Note the inevitable increase in ramp-up and ramp-down.
- KNAP scales well even when using several thousand processors.

Does Differencing Make a Difference?

A Simple Test

- 38 MILP instances from Lehigh/CORAL and MIPLIB3.
- Solved to optimality by using BLIS in 10 minutes.
- PC, 2.8 GHz Pentium, 2G RAM, Linux, COIN/Clp.

	No	Yes	Geometric
Total Time	2016 seconds	1907 seconds	1.0
Total Peak Memory	1412 MB	286 MB	4.3

Problem	Time(No)	Memory(No)	Time(Yes)	Memory(Yes)
dcmulti	4.20 s	17.4 MB	4.19 s	1.4 MB
dsbmip	33.28 s	34.1 MB	33.16 s	2.4 MB
egout	0.18 s	0.3 MB	0.18 s	0.2 MB
enigma	6.41 s	13.1 MB	6.16 s	2.3 MB
fiber	6.60 s	17.6 MB	6.56 s	2.1 MB
...	...	...	...	...

BLIS Scalability for Moderately Difficult Instances

- Selected 18 MILP instances from Lehigh/CORAL, MIPLIB 3.0, MIPLIB 2003, BCOL, and markshare.
- Tested on the Clemson cluster.

Instance	Nodes	Ramp -up	Idle	Ramp -down	Comm Overhead	Wallclock	Eff
1 P Per Node	11809956	– –	– –	– –	– –	33820.53 0.00286	1.00
4P Per Node	11069710	0.03% 0.03%	4.62% 4.66%	0.02% 0.00%	16.33% 16.34%	10698.69 0.00386	0.79
8P Per Node	11547210	0.11% 0.10%	4.53% 4.52%	0.41% 0.53%	16.95% 16.95%	5428.47 0.00376	0.78
16P Per Node	12082266	0.33% 0.27%	5.61% 5.66%	1.60% 1.62%	17.46% 17.45%	2803.84 0.00371	0.75
32P Per Node	12411902	1.15% 1.22%	8.69% 8.78%	2.95% 2.93%	21.21% 21.07%	1591.22 0.00410	0.66
64P Per Node	14616292	1.33% 1.38%	11.40% 11.46%	6.70% 6.72%	34.57% 34.44%	1155.31 0.00506	0.46

Impact of Problem Properties

- Instance `input150_1` is a knapsack instance. When using 128 processors, BLIS achieved super-linear speedup mainly to the decrease of the tree size
- Instance `fc_30_50_2` is a fixed-charge network flow instance. It exhibits very significant increases in the size of its search tree.
- Instance `pk1` is a small integer program with 86 variables and 45 constraints. It is relatively easy to solve.

Instance	P	Node	Ramp-up	Idle	Ramp-down	Wallclock	Eff
input150_1	64	75723835	0.44%	3.38%	1.45%	1257.82	1.00
	128	64257131	1.18%	6.90%	2.88%	559.80	1.12
	256	84342537	1.62%	5.53%	7.02%	380.95	0.83
	512	71779511	3.81%	10.26%	10.57%	179.48	0.88
fc_30_50_2	64	3494056	0.15%	31.46%	9.18%	564.20	1.00
	128	3733703	0.22%	33.25%	21.71%	399.60	0.71
	256	6523893	0.23%	29.99%	28.99%	390.12	0.36
	512	13358819	0.27%	23.54%	29.00%	337.85	0.21
pk1	64	2329865	3.97%	12.00%	5.86%	103.55	1.00
	128	2336213	11.66%	12.38%	10.47%	61.31	0.84
	256	2605461	11.55%	13.93%	20.19%	41.04	0.63
	512	3805593	19.14%	9.07%	26.71%	36.43	0.36

BLIS Scalability for Very Difficult Instances

- Tests on Clemson's palmetto cluster (60 on the Top 500 list, 11/2008, Linux, MPICH, 8-core 2.33GHz Xeon/Opteron mix, 12-16GB RAM).
- Tests use one processor per node.

Raw Computational Results

Name	256	128	64	1
mcf2	926	1373	2091	43059
neos-1126860	2184	1830	2540	39856
neos-1122047	1676	1125	1532	NS
neos-1413153	4230	3500	2990	20980
neos-1456979		78.06%	NS	NS
neos-1461051	396	1082	536	NS
neos-1599274		1500	8108	9075
neos-548047		137.29%	376.48%	482%
neos-570431	1034	1255	1308	21873
neos-611838	712	956	886	8005
neos-612143	565	1716	1315	4837
neos-693347		1.28%	1.70%	NS
neos-912015	538	438	275	10674
neos-933364		6.67%	6.79%	11.80%
neos-933815		6.54%	8.77%	32.85%
neos-934184		6.67%	6.76%	9.15%
neos18		30.78%	30.78%	79344

Speedups

Name	256	128	64
mcf2	46.5	31.36	20.59
neos-1126860	18.25	21.78	15.69
neos-1413153	4.96	5.99	7.02
neos-1599274		6.05	1.12
neos-570431	21.15	17.43	16.72
neos-611838	11.24	8.37	9.03
neos-612143	8.56	2.82	3.68
neos-912015	19.84	24.37	38.81

Efficiency

Name	256	128	64
mcf2	0.18	0.25	0.32
neos-1126860	0.07	0.17	0.25
neos-1413153	0.02	0.05	0.11
neos-1599274		0.05	0.02
neos-570431	0.08	0.14	0.26
neos-611838	0.04	0.07	0.14
neos-612143	0.03	0.02	0.06
neos-912015	0.08	0.19	0.61

Shameless 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.

ALPS

- Our methods implemented in ALPS seem effective in improving scalability.
- The framework is useful for implementing serial or parallel tree search applications.
- The KNAP application achieves very good scalability.
- There is still much room for improvement
  - load balancing,
  - fault tolerance,
  - hybrid architectures,
  - grid enable.

BLIS

- The performance of BLIS in serial mode is favorable when compared to state of the art non-commercial solvers.
- The scalability for solving generic MILPs is highly dependent on properties of individual instances.
- Based on BLIS, applications like VRP/TSP can be implemented in a straightforward way.
- Much work is still needed
  - Callable library API
  - Support for column generation
  - Enhanced heuristics
  - Additional capabilities

Obtaining CHiPPS

The CHiPPS framework is available for download at

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

Thank You!

Questions?