A New Framework for Scalable Parallel Tree Search

Ted Ralphs and Yan Xu
Industrial and Systems Engineering
Lehigh University
http://www.lehigh.edu/~tkr2

Laszlo Ladanyi
IBM T.J. Watson Research Center

Matt Saltzman
Clemson University

International Symposium on Mathematical Programming, TU Denmark, Monday, August 18, 2003

Outline of Talk

- Introduction
  - Parallel computing concepts
  - Tree search
  - Knowledge sharing
- New framework
  - Abstract Library for Parallel Search (ALPS)
  - Branch, Constrain, and Price Software (BiCePS)
- Conclusions

Motivation and Objectives

- Tree search algorithms are widely used in the areas such as Mathematical Programming, Artificial Intelligence, Theorem Proving, etc.
- These areas are rich with difficult, but important problems.
- Numerous specialized frameworks and solvers already exist.
- Why another one?
- Our goal is a framework that
  - operates effectively in both sequential and parallel environments;
  - is general enough for a wide range of settings, but has the tools needed for specific applications;
  - provides scalability for applications with highly dynamic and irregular search trees; and
  - provides the functionality needed for management of large amounts of dynamically generated data.

Parallel Systems and Scalability

- 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	$E = S/N$

Parallel Overhead

- The key to scalability is reducing parallel overhead.
- Contributors to parallel overhead
  - Communication Overhead
  - Idle Time due to
    - Handshaking/Synchronization
    - Task Starvation
    - Ramp Up/Down
  - Performance of Redundant Work
- Redundant work is work that would not have been performed in the sequential algorithm.

Tree Search Algorithms

- The *search space* consists of a global set of *states*.
- The *solution space* consists of the states that are *feasible*.
- In addition, each state may have an associated *cost*.
- The goal is to discover a feasible state, possibly having least cost.
- The approach is to divide and conquer.
- Elements of a Tree Search Algorithms
  - Evaluation method (determines the priority).
  - Pruning method.
  - Splitting or branching method (divides the search space into subproblems).
- We will use the terms node and subproblem interchangeably.

Knowledge Generation and Sharing

- *Knowledge* is information generated during the course of the search that guides the search.
  - Knowledge generation changes the shape of the tree dynamically.
  - The primary way in which parallel tree search algorithms differ is the way in which knowledge is shared (Trienekens '92).
- Sharing knowledge helps reduce overhead by guiding the search.
  - If all processes have "perfect knowledge," then no process will have an empty task queue and no redundant work will be performed.
  - The goal is for the parallel search to be executed in roughly the same manner as the sequential search.
- However, knowledge sharing also increases communication overhead and idle time.
- This is the fundamental tradeoff of knowledge sharing.

Parallel Overhead in Tree Search

Main contributors to parallel overhead

- Communication Overhead (cost of sharing knowledge)
- Idle Time
  - Handshaking/Synchronization (cost of sharing knowledge)
  - Task Starvation (cost of *not* sharing knowledge)
  - Ramp Up/Down Time (cost of generating initial knowledge).
- Performance of Redundant Work (cost of *not* sharing knowledge).

Knowledge Pools

- Knowledge is shared through *knowledge pools*.
- Methods for disseminating knowledge
  - Pull: Process requests information from a knowledge pool (asynchronously or synchronously).
  - Push: Knowledge pool broadcasts knowledge to other pools.
- Basic examples of knowledge to be shared.
  - Solutions
  - Node/Subproblem Priorities
  - Node/Subproblem Descriptions

Load Balancing

- *Load balancing* is the process by which tasks are distributed or redistributed.
- Load balancing is a type of knowledge sharing.
- Static load balancing
  - Determines the initial task distribution.
  - In dynamic search algorithms, this can be difficult.
  - The main source of ramp-up time.
- Dynamic load balancing
  - Used periodically to redistribute the tasks.
  - Critical in dynamic search algorithms.

The ALPS Project

- In partnership with IBM and the COIN-OR project.
- Multi-layered C++ class library for implementing scalable, parallel tree search algorithms.
- Design is fully generic and portable.
  - Support for implementing general tree search algorithms.
  - Support for any evaluation/bounding scheme.
  - No assumptions on problem/algorithm type.
  - No dependence on architecture/operating system.
  - No dependence on third-party software (communications, solvers).
- Emphasis on parallel scalability.
- Support for large-scale, data-intensive applications (such as BCP).
- Support for advanced methods not available in commercial codes.

Previous Work

- Previously developed frameworks for implementing parallel BCP algorithms:
  - SYMPHONY is written in C.
  - COIN/BCP is written in C++.
- Both frameworks implement a *single-pool* algorithm, in which there is a central knowledge pool for node descriptions.
- Computational experience
  - The central node pool has perfect knowledge of the search tree and effectively eliminates the performance of redundant work.
  - The most serious scalability issues are ramp-up/ramp-down and bottlenecks at the knowledge pools.
  - Surprisingly, the cut pool is a bigger bottleneck than the central node pool.
  - Ramp-up time can be a very serious issue for settings in which the search tree is relatively small.

ALPS Library Hierarchy

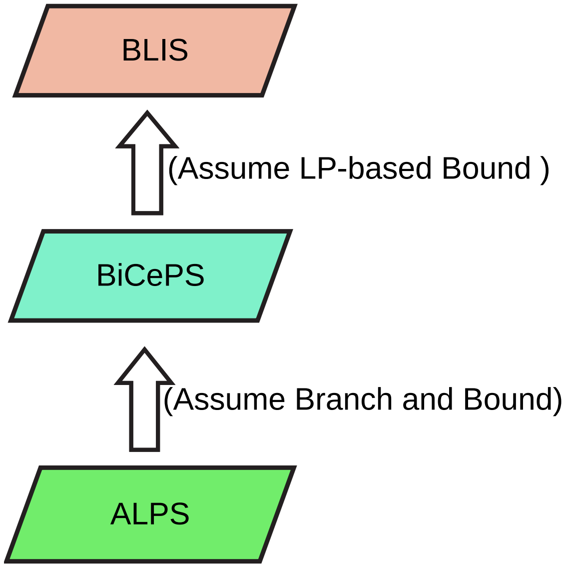
Modular library design with minimal assumptions in each layer.

ALPS Abstract Library for Parallel Search

- search handling layer.
- prioritizes based on quality.

BiCePS Branch, Constrain, and Price Software

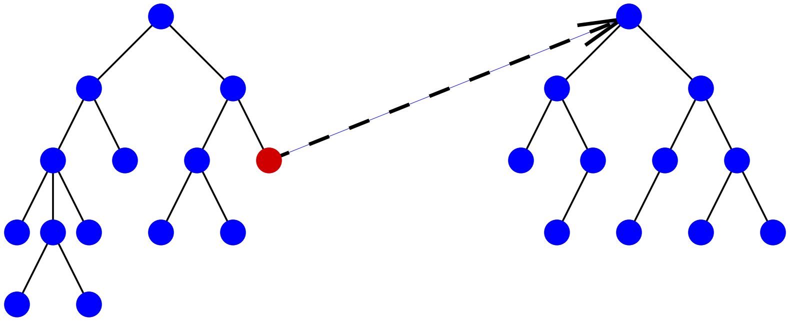
- data handling layer for mathematical programming.
- adds notion of variables and constraints.
- assumes iterative bounding process.
- built using OSI modeling library (under development).



BLIS BiCePS Linear Integer Solver

- Concretization of BiCePS.
- Constraints are linear functions.

ALPS: Overall Design



- The basic unit task is to search an `AlpsSubTree`.
- Searches are performed asynchronously, but can be halted and resumed.
- Subtrees can be broken apart for load balancing purposes.
- Searching a subtree
  - Place the root `AlpsTreeNode` on the queue.
  - While the queue is not empty, take an `AlpsTreeNode` from the queue and process it.
- Processing a node consists of changing its status, although partial processing is possible.
- Node stati: `candidate`, `evaluated`, `fathomed`, `branched`.

ALPS: Knowledge Management

- All information to be shared is considered `AlpsKnowledge` and has a type.
- `AlpsKnowledge` is stored locally in one or more `AlpsKnowledgePool`s.
- `AlpsKnowledgePool` functions
  - Receive and store knowledge from other knowledge pools.
  - Field requests for knowledge from other knowledge pools.
  - Generate new knowledge.
- The knowledge pools communicate through `AlpsKnowledgeBroker`s, which contain routing information.
- ALPS knowledge types
  - `AlpsSubTree`
  - `AlpsTreeNode`
  - `AlpsSolution`
  - `AlpsModel`
- The user can also define new types of knowledge.

ALPS: Knowledge Handling

- Need to deal with potentially HUGE amounts of knowledge.
- Duplication may be a big issue.
- Goal is to avoid such duplication in generation and storage.
- All knowledge types have an *encoded form* that allows it to be sent over the network.
- Detecting duplicate knowledge:
  1. Obtain a hash value from the encoded form.
  2. Object is looked up in hash map.
  3. If it does not exist, then it is inserted.
  4. A pointer to the unique copy in the hash map is added to the list.

ALPS: Load Balancing

ALPS uses a Master-Hubs-Workers task allocation paradigm.

Master

- has global information about workload.
- balances load between hubs (quantity *and* quality).

Hub

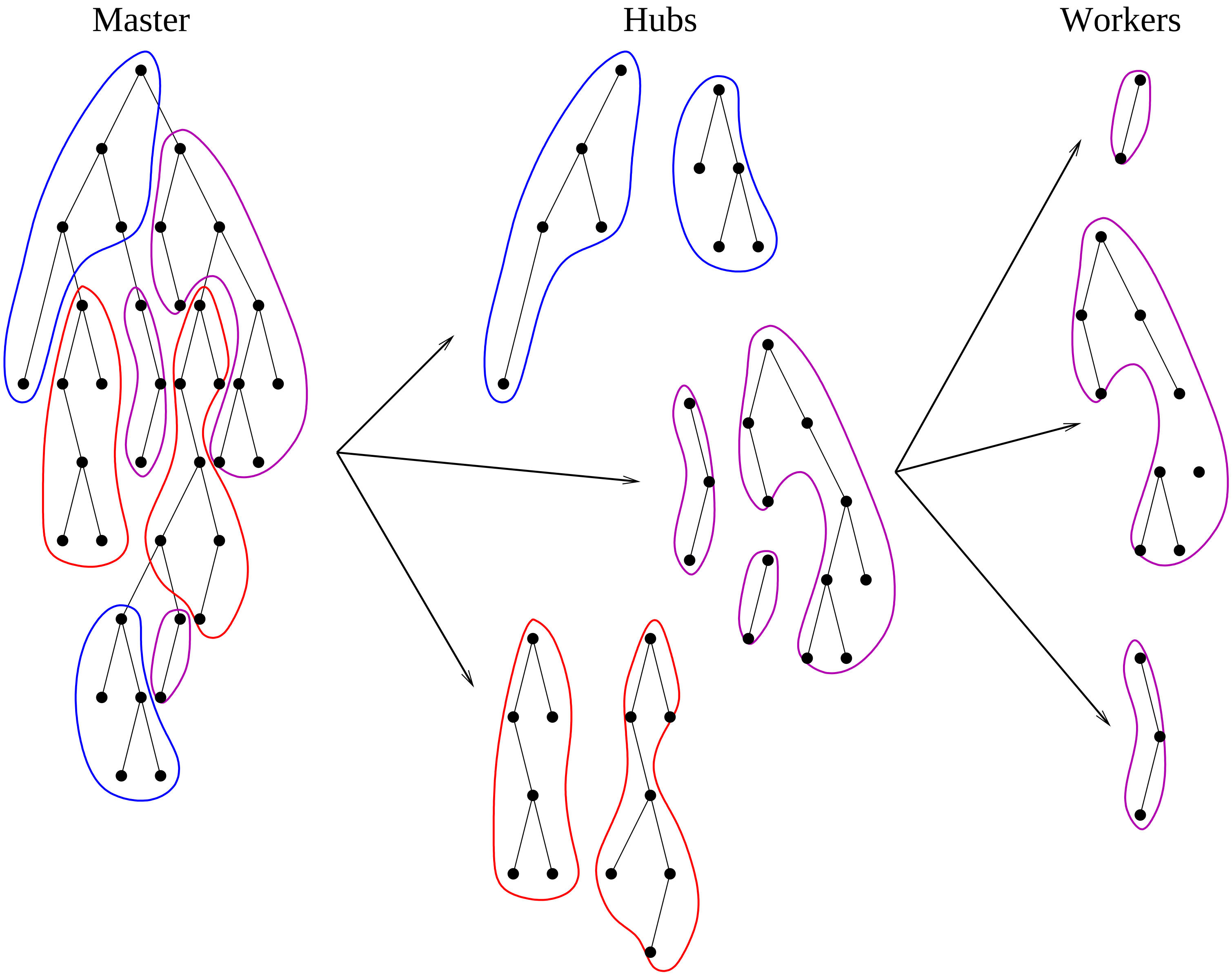
- manages collection of subtrees.
- balances load between workers

Worker

- searches subtrees.
- hub can interrupt.
- sends workload information to hub.

For data-intensive algorithms, load balancing presents additional challenges.

ALPS: Master - Hubs - Workers Paradigm



ALPS: Ramp-up/Ramp-down

- Ramp-up time: Time until all processors have useful work to do.
- Ramp-down time: Time during which there is not enough work for all processors.
- Ramp-up time is a difficult scalability issue for branch and bound when node evaluation is computationally intensive.
- Reducing Ramp-up
  - Branch more quickly.
  - Use different branching rules (produce more children).
  - Perform other useful work.

ALPS: Scheduler

- ALPS uses threads to perform task management (similar to PICO).
- Each knowledge broker has its own scheduler that listens for messages and prioritizes tasks.
- Most tasks are triggered by the arrival of a particular type of knowledge.
- Each task type has its own thread, which has three possible states:
  - waiting
  - active, and
  - blocked
- Tasks are divided up into those performed by the master, hub, and worker.
- A process can play one or more of these roles.

BiCePS: Knowledge Management

- In BCP algorithms, knowledge discovery consists of finding the constraints and variables that form the relaxations.
- Generating these objects can be difficult, so we want to share them.
- Hence we have a new types of knowledge that must be shared.
- BiCePS knowledge types
  - `BcpsTreeNode`
  - `BcpsSolution`
  - `BcpsModel`
  - `BcpsConstraint`
  - `BcpsVariable`
- The `BcpsModel`, `BcpsVariable`, and `BcpsConstraint` classes are derived from OSI classes.
- To conserve memory, subtrees are stored using *differencing*.
- This presents assitional challenges for load balancing.

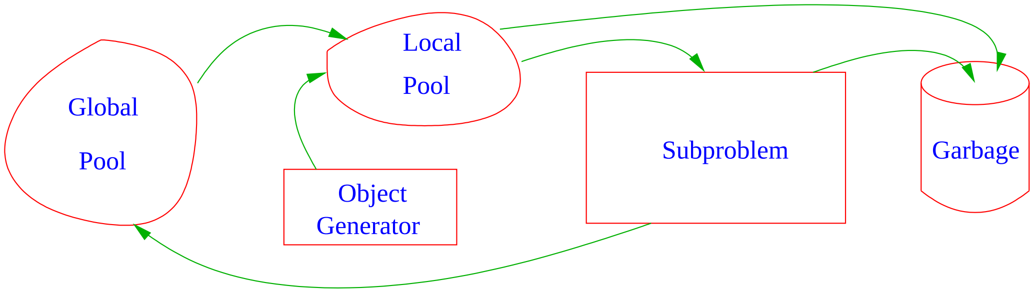
BiCePS: Overall Design

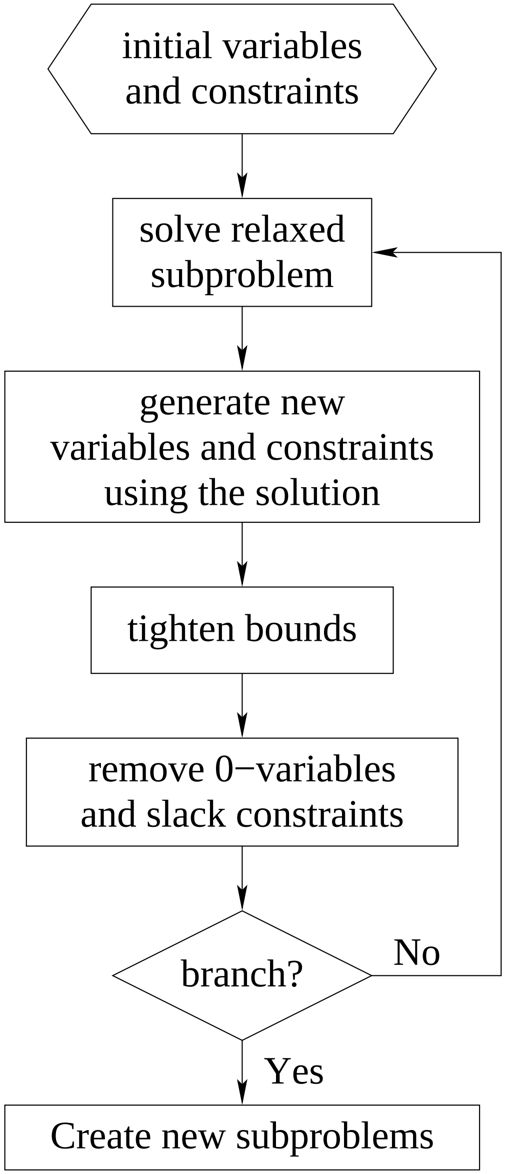
A `BcpsModel` is a set of `BcpsVariables` and `BcpsConstraints`.

Evaluating a subproblem

- solve a relaxation.
- generate new variables and constraints.
- tighten bounds.
- remove 0-variables and slack constraints.

If all else fails or when desired, branch.





Example: Simple Knapsack Solver

Required classes:

- `KnapModel`
- `KnapTreeNode`
- `KnapSolution`

Example: Simple Knapsack Solver

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

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

    KnapModel().registerClass();
    KnapSolution().registerClass();
    KnapTreeNode().registerClass();

    AlpsTreeNode* root = new KnapTreeNode(&model);

    broker.search(root);

    broker.printResult();
}
```

(Very) Preliminary Computational Results

Preliminary computational results with the ALPS knapsack solver on a Beowulf cluster using six processors.

Instance	Sequential	Parallel	Speedup	nodes (million)
60_1	28m34s	53s	31.2	15
75_1	10m47s	1m02s	10.2	21
75_2	8m12s	1m04s	7.7	20
75_3	14m46s	1m33s	9.3	29
75_4	4m48s	49s	5.6	15

Current Status

- A preliminary version of ALPS is almost completed and will be released in the near future as part of the COIN-OR repository.
- We have just begun testing with the knapsack solver and results are promising.
- In the near future, we would like to
  - move to a much larger number of processors, and
  - begin work with more data-intensive algorithms, such as BCP.
- We are currently focusing on
  - developing BiCePS,
  - improving load balancing, and
  - reducing ramp-up.

What's Currently Available

- SYMPHONY: C library for implementing BCP
  - User fills in stub functions.
  - Supports shared or distributed memory.
  - Documentation and source code available `www.BranchAndCut.org`.
- COIN/BCP: C++ library for implementing BCP
  - User derives classes from library.
  - Documentation and source code available `www.coin-or.org`.
- ALPS/BiCePS/BLIS
  - In development and available soon.
  - Will be distributed from CVS at `www.coin-or.org`.
- The COIN-OR repository `www.coin-or.org`

The COIN-OR Project

- Supports the development of interoperable, open source software for operations research.
- Maintains a CVS repository for open source projects.
- Promotes peer review of open source software as a supplement to the open literature.
- Software and documentation is freely downloadable from `www.coin-or.org`.
- For more information, please attend the sessions sponsored by COIN-OR.

Scalability Issues: Motivation

Results solving VRP instances with SYMPHONY 2.8.2 (single node pool, multiple cut pools) and OSL 3.0 on a 48-node Beowulf cluster

Instance	Tree Size	Ramp Up	Ramp Down	Idle (Nodes)	Idle (Cuts)	CPU sec	Wallclock
A − n37 − k6	14305	1.70	2.02	12.31	40.06	1067.49	286.37
A − n39 − k5	483	0.81	0.05	0.35	1.30	54.17	14.49
A − n39 − k6	739	0.90	0.06	0.45	1.10	37.45	10.25
A − n44 − k6	3733	1.58	0.55	3.62	11.64	453.45	119.35
A − n45 − k6	493	0.59	0.05	0.42	1.06	65.09	17.10
A − n46 − k7	176	0.96	0.01	0.15	0.79	25.69	7.02
A − n48 − k7	4243	1.14	0.77	4.31	15.54	593.36	155.05
A − n53 − k7	2808	1.32	0.48	2.95	9.44	385.68	100.98
A − n55 − k9	6960	2.07	1.46	8.12	15.31	913.35	237.30
A − n65 − k9	18165	1.41	5.83	25.89	105.84	5190.83	1335.60
B − n45 − k6	1635	0.72	0.21	1.39	2.09	131.13	34.92
B − n51 − k7	348	0.36	0.03	0.32	0.37	25.35	6.88
B − n57 − k7	4036	0.76	0.39	3.21	5.52	494.13	131.87
B − n64 − k9	100	0.58	0.01	0.08	0.19	15.49	4.22
B − n67 − k10	16224	2.95	2.54	17.85	64.88	2351.30	618.73
4 NP$'$s	74451	17.87	14.45	81.42	275.11	11803.97	3080.12
Per Node		0.0002	0.0002	0.0011	0.0037	0.1585	0.1655
8 NP$'$s	82488	67.12	17.07	89.54	370.96	11834.68	1569.27
Per Node		0.0008	0.0002	0.0011	0.0045	0.1435	0.1522
16 NP$'$s	97078	203.54	41.19	110.36	1045.95	12881.44	908.68
Per Node		0.0021	0.0004	0.0011	0.0108	0.1327	0.1498
32 NP$'$s	98991	640.74	49.09	135.74	3320.88	13044.33	545.73
Per Node		0.0065	0.0005	0.0014	0.0335	0.1318	0.1764