A New Framework for Scalable Parallel Tree Search

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

Laszlo Ladanyi
IBM T.J. Watson Research Center

Matt Saltzman
Clemson University

INFORMS Annual Conference, Sun Nov 17, 2002

Outline of Talk

- Overview of parallel tree search
    - Knowledge sharing
    - Data-intensive applications
- The Abstract Library for Parallel Search (ALPS)
    - Scalability
    - Data handling
- Conclusions

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	$E = S/N$

Tree Search Algorithms

- Application Areas
  - Discrete Optimization
  - Artificial Intelligence
  - Game Playing
  - Theorem Proving
  - Expert Systems
- Elements of Tree Search Algorithms
  - Node splitting method (branching)
  - Search order
    - Depth-first search
    - Iterative Deepening
    - Best-first search
  - Pruning rules
    - Feasibility
    - Cost
  - Bounding method (optimization only)

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, which makes *load balancing* difficult.
  - The primary way in which parallel tree search algorithms differ is the way in which knowledge is shared (Trienekens '92).
- Sharing knowledge helps eliminate the performance of *redundant work* by guiding the search.
  - If all processes have "perfect knowledge," then 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.
- Knowledge sharing increases communication overhead and idle time.
- This is the fundamental tradeoff.

Knowledge Bases

- Knowledge is shared through *knowledge bases*.
- Methods for disseminating knowledge
  - Pull: Process requests information from the knowledge base (asynchronously or synchronously).
  - Push: Knowledge base broadcasts knowledge to processes.
  - An important parameter to consider is whether the current task is interrupted when knowledge is received or not.
- Basic examples of knowledge to be shared.
  - Bounds
    - Upper (single global bound)
    - Lower (need knowledge of distribution of bounds in tree)
  - Node Descriptions

Parallel Overhead in Tree Search

- Main contributors to parallel overhead
  - Communication Overhead (cost of sharing knowledge)
  - Idle Time
    - Handshaking (cost of sharing knowledge)
    - Ramp Up/Down Time (cost of generating initial knowledge).
  - Performance of Redundant Work (cost of *not* sharing knowledge).
- Redundant work is work that would not have been performed in the sequential algorithm.
- Note again the fundamental tradeoff of knowledge sharing.

Data-intensive Applications

- In applications such as *branch, cut, and price*, 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.
- Abstractly, we can think of each node as being described by a list of *objects*.
- In our case, the objects are the cuts and variables.
- These objects can be generated throughout the search process.
- In BCP, the list of objects does not change much from parent to child.
- We can therefore store the description of an entire subtree very compactly using *differencing*.

Knowledge Sharing in BCP

- In BCP, knowledge discovery consists of finding the cuts and variables that form the LP relaxations.
- Generating these objects can be time consuming, so we want to share them when they are found.
- Hence we have a new kind of knowledge that must be shared.
- Knowledge bases in BCP
  - Node Pools
    - Node descriptions
    - Lower bounds
  - Object Pools
- Note that the sharing of lower bounds is important in enforcing the search order and limiting redundant work.

Previous Work

- Existing 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 base 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 bases.
  - Surprisingly, the object pools are 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.

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 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.

The Library Hierarchy

Modular library design with minimal assumptions in each layer.

ALPS Abstract Library for Parallel Search

- manages the search tree.
- prioritizes based on quality.

BiCePS Branch, Constrain, and Price Software

- manages the data.
- adds notion of objects and differencing.
- assumes iterative bounding process.

BLIS BiCePS Linear Integer Solver

- implementation of BCP algorithm.
- objects are the cuts and variables.

ALPS Design Overview

- Each processor hosts one or more knowledge bases.
- Knowledge base functions
  - Receive and store knowledge from other knowledge bases.
  - Field requests for knowledge from other knowledge bases.
  - Generate new knowledge.
  - May request knowledge from other knowledge bases.
- The knowledge bases communicate through *knowledge brokers*, which contain routing information.
- Knowledge bases in BiCePS
  - Node Pools
  - Object Pools
  - Node Processors

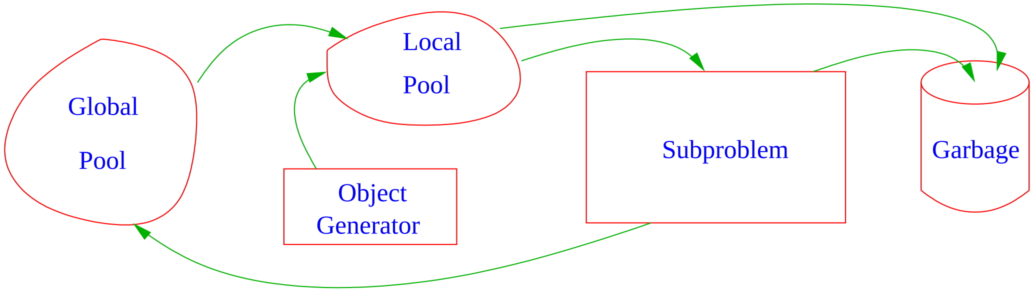
BiCePS Design Overview

A subproblem is a set of objects with an objective.

Processing a subproblem

- solve a relaxation.
- generate new objects.
- tighten bounds.
- remove objects with value 0.

If all else fails or when desired, branch.



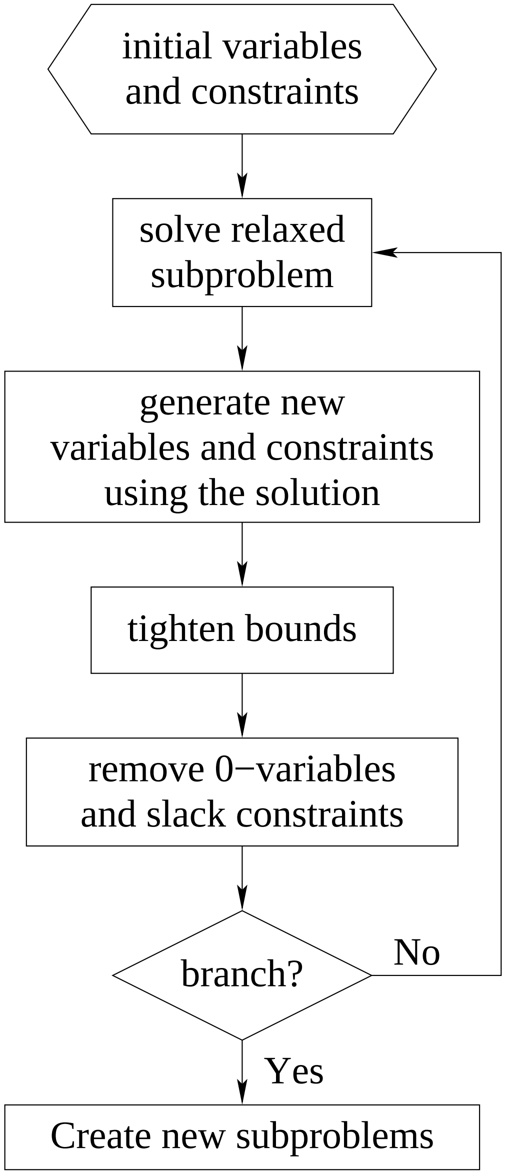
Global Pool

Local Pool

Object Generator

Subproblem

Garbage



initial variables and constraints

solve relaxed subproblem

generate new variables and constraints using the solution

tighten bounds

remove 0−variables and slack constraints

branch?

No

Yes

Create new subproblems

Data Handling in BiCePS

- Need to deal with HUGE numbers of objects.
- Duplication is a big issue.
- Goal is to avoid such duplication in generation and storage.
- Objects have an *encoded form* containing information about how to add the object to a relaxation.
- Object pools allow generated objects to be shared.
- Implementation:

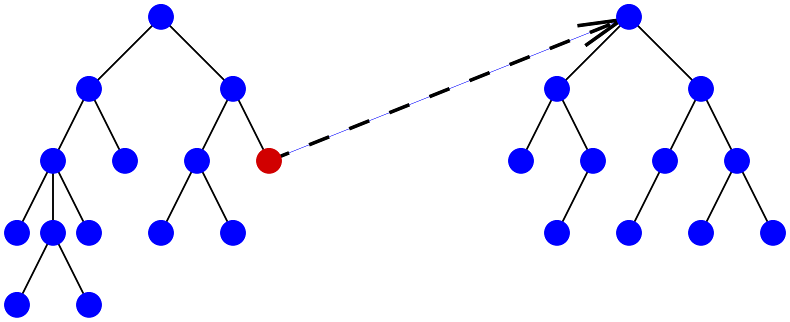
1. From encoded form, obtain a hash value.
2. Object is looked up in hash map.
3. If it does not exist, then it is inserted.
4. A pointer to the unique copy in the hash map is added to the list.

Improving Scalability

- Dynamic granularity
- Decentralization of knowledge
- Elimination of synchronous messaging
- Reduction in ramp-up/ramp-down time

Improving Scalability: Granularity

Work unit is a subtree.



Advantages:

- less communication.
- more compact storage via differencing.

Disadvantage:

- more possibility of redundant work being done.

Improving Scalability: Master - Hubs - Workers Paradigm

Master

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

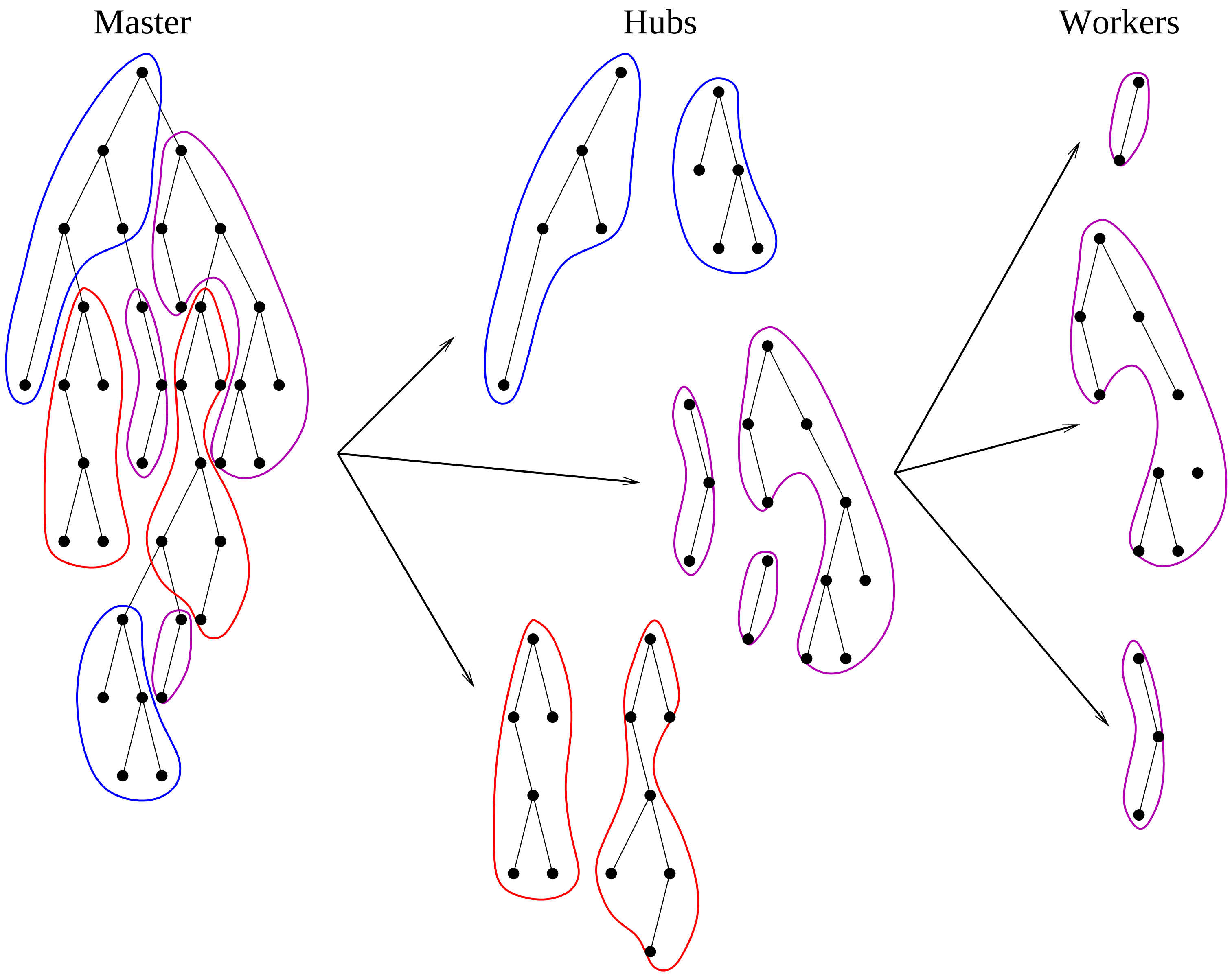
Hub

- manages collections of subtrees.
- balances load between workers

Worker

- processes one subtree.
- hub can interrupt.
- sends branch and quality information to hub.

Improving Scalability: Master - Hubs - Workers Paradigm



Improved Scalability: Asynchronous Messaging

Possible communication bottlenecks:

- Too many messages.
  - avoided by the increased task granularity.
  - master-hub-worker paradigm also contributes.
- Too much synchronization (handshaking)
  - almost no handshaking.
  - must take place when a worker finishes exploring a subtree.

Improving Scalability: 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 perhaps the most important scalability issue for branch and bound when the bounding is computationally intensive.
- Controlling Ramp-up/ramp-down
  - Branch more quickly.
  - Use different branching rules (produce more children).
  - Hub instructs workers when to change rules.

Preliminary Conclusions

- We can achieve close to linear speedup with up to 32 processors using a single-pool approach.
- However, there is still significant parallel overhead and this is not a scalable solution.
- Performance of redundant work is not a problem with a single node pool, but may be with multiple pools.
- Efficient knowledge sharing is the key challenge.
- Synchronous requests for information and ramp-up time are the primary scalability issue for BCP algorithms.
  - For BCP, the object pools are the biggest bottleneck.
  - We can try to control this by scaling the number of pools.
  - Ramp up time is more difficult to control.

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`

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