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: $T_o = N T_p - T_s$
  - 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 \textit{not} sharing knowledge)
  - Ramp Up/Down Time (cost of generating initial knowledge).
- Performance of Redundant Work (cost of \textit{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.
• 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 AlpsKnowledgePools.

• 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 AlpsKnowledgeBrokers, 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

```c
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.

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

- **COIN/BCP**: C++ library for implementing BCP
  - User derives classes from library.
  - Documentation and source code available [www.coin-or.org](http://www.coin-or.org).

- **ALPS/BiCePS/BLIS**
  - In development and available soon.
  - Will be distributed from CVS at [www.coin-or.org](http://www.coin-or.org).

- The **COIN-OR** repository [www.coin-or.org](http://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

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