A Library Hierarchy 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

Parallel Processing in Operations Research, Tuesday May 21, 2002
Outline of Talk

• Overview of parallel tree search
  – Knowledge sharing
  – Data-intensive applications

• Overview of branch, cut, and price algorithms

• The Abstract Library for Parallel Search (ALPS)
  – Scalability
  – Data handling

• Computational results

• 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 Search Algorithms
  – Node splitting method (branching)
  – Search order
    * Depth-first search
    * Iterative Deepening
    * Best-first search
  – Pruning rule
    * Feasibility
    * Cost
  – Bounding method (optimization only)
Parallel Tree Search

• Main contributors to parallel overhead
  – Communication Overhead
  – Idle Time (Ramp Up/Down Time)
  – Idle Time (Handshaking)
  – Performance of Redundant Work

• Redundant work is work that would not have been performed in the sequential algorithm.

• The primary way in which tree search algorithms differ is the way in which knowledge is shared (Trienekens '92).

• Sharing knowledge helps eliminate the performance of redundant work.

• If all processes have “perfect knowledge,” then no redundant work will be performed.

• However, knowledge sharing may also increase communication overhead and idle time.
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**
LP-based Branch and Bound

• Consider problem $P$:

$$\begin{align*}
\min & \quad c^T x \\
\text{s.t.} & \quad Ax \leq b \\
& \quad x_i \in \mathbb{Z} \forall i \in I
\end{align*}$$

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

• Let $\mathcal{P} = \text{conv}\{ x \in \mathbb{R}^n : Ax \leq b, x_i \in \mathbb{Z} \forall i \in I \}$.

• Basic Algorithmic Approach
  – Use LP relaxations to produce lower bounds.
  – Branch using hyperplanes.

• Basic Algorithmic Elements
  – A method for producing and tightening the LP relaxations.
  – A method for branching.
Branch, Cut, and Price

- Weyl-Minkowski
  - \( \exists (\bar{A}, \bar{b}) \in \mathbb{R}^{\bar{m} \times n+1} \) s.t. \( \mathcal{P} = \{x \in \mathbb{R}^n : \bar{A}x \leq \bar{b}\} \)
  - We want the solution to \( \min \{c^T x : \bar{A}x \leq \bar{b}\} \).
  - Solving this LP isn’t practical (or necessary).

- BCP Approach
  - Form LP relaxations using submatrices of \( \bar{A} \).
  - The submatrices are defined by sets \( \mathcal{V} \subseteq [1..n] \) and \( \mathcal{C} \subseteq [1..\bar{m}] \).
  - Forming/managing these relaxations efficiently is one of the primary challenge of BCP.
Data-intensive Applications

• In applications such as 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.

• 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.
## 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>
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.
**ALPS Project**

**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.
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.
Improving Scalability

- Unit of work (granularity)
- Number and location of knowledge bases
- Synchronous vs. asynchronous messaging
- 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 (node quality and distribution).
• balances load between hubs.
• balances quantity and quality.

Hubs

• manage collections of subtrees (may not have full descriptions)
• balances load between workers

Workers

• 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.
Data Handling

- 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.
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.

• **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 not yet freely available.
  – 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](http://www.coin-or.org)