A Framework for Scalable Parallel Tree Search

Yan Xu
SAS Institute

Ted Ralphs
Lehigh University

Laszlo Ladányi
IBM T.J. Watson Research Center

Matt Saltzman
Clemson University

CORS/INFORMS Joint Int’l Meeting, Banff, Alberta, Canada, Sunday, May 16, 2004
Outline

• Overview of parallel tree search
  – Tree search
  – Scalability
  – Knowledge sharing

• The Abstract Library for Parallel Search (ALPS)
  – Design overview
  – Class hierarchy
  – Knapsack solver
  – Generic MIP solver

• Future work
Tree Search

• Tree search algorithms systematically search the nodes of a directed, acyclic graph for one or more goal nodes.

• Generally, the graph is not known a priori, but is constructed as the algorithm progresses.

• A generic tree search algorithm consists of the following elements:
  – Processing method: Is goal achieved?
  – Search strategy: What is node priority?
  – Fathoming rule: Can node can be fathomed?
  – Branching method: What are the successors?

• Each node has an associated description and a status.
  – candidate: Ready to be processed.
  – active: Currently being processed.
  – processed: Successors generated (not a leaf node).
  – fathomed: No successors generated (a leaf node).

• The algorithm consists of choosing a candidate node, processing it, and either fathoming or branching.
Parallelizing Tree Search

- Tree search is a “divide and conquer” approach, so it is conceptually easy to parallelize.
- As the number of processors is increased, it becomes increasingly difficult to manage the solution process.
- *Scalability* measures how well a parallel system takes advantage of increased computing resources.

- **Terms**
  
  - **Sequential runtime** $T_s$
  - **Parallel runtime** $T_p$ when using $p$ processes
  - **Parallel overhead** $T_o = pT_p - T_s$
  - **Speedup** $S = T_s / T_p$
  - **Efficiency** $E = S / p$

- Good scalability is the goal of any parallel algorithm.
- Achieving it involves a number of tradeoffs.
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 a fundamental *tradeoff* between centralization and decentralization of knowledge.
Parallel Overhead

- The main contributors to parallel overhead in tree search are
  - Communication Overhead (cost of sharing knowledge)
  - Idle Time
    * Handshaking/Synchronization (cost of sharing knowledge)
    * Task Starvation (cost of *not* sharing knowledge)
    * Ramp Up Time (cost of sharing knowledge)
    * 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.
Load Balancing

• The most fundamental knowledge generated during the search are the node descriptions.

• The node descriptions represent work in the system.

• It is critical to keep this work evenly distributed, both in terms of quantity and of quality.

• 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.
ALPS Project

• ALPS is a framework for implementing parallel tree search developed in partnership with the COIN-OR project, IBM, and NSF.

• A number of such frameworks and solvers already exist.
  – Commercial: CPLEX, Lindo, Xpress, OSL, and SAS/OR.
  – Serial: ABACUS, MINTO, MIPO, bc-opt, MPSARX, COIN/SBB, bonsaiG.
  – Parallel
    * MIP: SYMPHONY, COIN/BCP, PARINO, FATCOP, PICO
    * Branch and Bound: BoB, PPBB-Lib, and PUBB.

• Why another one?

• Our goal is to build upon and improve previous work.
  – Provide a general framework for parallel tree search.
  – Provide a base layer upon which to build more specialized frameworks.
  – Provide support for data-intensive algorithms.
  – Provide improved scalability.
  – Operate effectively in sequential environments.
Global Picture

ALPS

- search handling layer.
- prioritizes nodes based on quality.

BiCePS  Branch, Constrain, and Price Software

- data handling layer for optimization.
- adds notion of variables and constraints.
- assumes iterative bounding process.

BLIS  BiCePS Linear Integer Solver

- concretization of BiCePS.
- constraints are linear functions.
ALPS: Knowledge Sharing Scheme

- All knowledge to be shared is considered as **AlpsKnowledge**, and has its own type, e.g., **AlpsSubTree**, **AlpsTreeNode**, **AlpsSolution**, and **AlpsModel**.

- **AlpsKnowledge** is managed by one or more **AlpsKnowledgePools**.

- The knowledge pools communicate through **AlpsKnowledgeBrokers**.
ALPS: Knowledge Handling

- Need to deal with potentially **HUGE** amounts of knowledge.
- **Duplication** and **efficient storage** are a big issue.
- All knowledge has an **encoded form** that contains only the data from the class in the form of a string.
  - This representation is memory-efficient for storage.
  - This form is also appropriate for message-passing.
  - Provides a type-independent representation.
  - Allows easy detection of duplication.
- **Detecting duplicate knowledge:**
  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.
ALPS: Increased Granularity

• One easy way to decrease communication overhead is to increase granularity.

• In ALPS, the basic unit of work is a subtree.

• Subtrees can be stored efficiently using differencing.

• ALPS tried to keep subtrees together as a unit.

• Pros:
  – less communication.
  – more compact storage via differencing.
  – Easy to use local valid knowledge, like locally valid cuts.

• Cons:
  – more possibility of redundant work being done.
ALPS: Master-Hub-Worker paradigm

The easiest load balancing approach is a single central node pool, but this does not scale. Instead, we use the following architecture:

Master

- has global information about the system status.
- balances load between hubs (quantity and quality).

Hub

- manages a cluster of workers. Has information of its cluster status.
- balances load between its workers.

Worker

- explores subtrees.
- hub can interrupt.
- sends workload information to its hub periodically.
ALPS: Master-Hubs-Workers Paradigm
ALPS: Master-Hubs-Workers Paradigm

- Processes are grouped into different *clusters*.
- Each cluster has one *hub* and at least one *worker*.
- Cluster size $S$ is the minimal integer satisfying

$$\text{hubNum} \times S \geq \text{processNum}$$
ALPS: Static and Dynamic Load Balancing

- Load balancing is used to distribute and redistribute tasks.

- Static load balancing (mapping):
  - ALPS uses *two-level root initialization*.
  - The master generates nodes for the hubs.
  - The hubs generate nodes for the workers.
  - This helps reduce ramp-up time.
  - However, it’s difficult to predict the work associated with a given node.

- Dynamic load balancing:
  - Periodically redistribute the high priority work.
  - *Inter-cluster* dynamic load balancing
  - *Intra-cluster* dynamic load balancing
  - ALPS tries to pass groups of nodes together as subtrees.
ALPS: Threads and Scheduler

• Each processor hosts a knowledge broker (KB) and several knowledge pools (KPs), so there must be a scheme for multi-tasking.

• ALPS uses a simple version of threads (ala PICO).

• ALPS processes are message-driven, so the knowledge broker resident at each processor controls the scheduling after initialization.

• The KB receives external messages and forwards them to the appropriate knowledge pool.

• It also receives internal messages from the local knowledge pools and forwards them to the appropriate KB.

• In between execution of the communication threads, the KB schedules computational tasks.

• The granularity of computational tasks is controlled by parameter settings.
ALPS: Class Hierarchy

- AlpsKnowledgePool and AlpsKnowledgeBroker related classes.
ALPS: Class Hierarchy

- **AlpsKnowledge** related classes.
Mainly comprises two parts:

- deriving the required and auxiliary problem-specific classes
  - \texttt{AlpsModel}
  - \texttt{AlpsTreeNode}
  - \texttt{AlpsNodeDesc}
  - \texttt{AlpsSolution}

- writing the \texttt{main} function
ALPS: Example Application

```c
int main(int argc, char* argv[]) {
    UserModel model;
    UserParams userPar;

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

    broker.registerClass("MODEL", new UserModel);
    broker.registerClass("SOLUTION", new UserSolution);
    broker.registerClass("NODE", new UserTreeNode);

    broker.search();
    broker.printResult();
    return 0;
}
```
Simple Knapsack Solver

- Four classes are derived from ALPS's base classes:
  - KnapModel,
  - KnapTreeNode,
  - KnapNodeDesc,
  - KnapSolution, and
  - KnapParameterSet

- It is a pure branch and bound algorithm without advanced techniques such as cutting planes, strong branching, reduce cost fixing, etc.

- Used to test how ALPS scales when node processing times are short.
Knapsack: Preliminary Computational Results

- **Test Environment:**
  
  Machine: Beowulf cluster with 48 dual-processor nodes  
  Processor: 1.0 GHz Pentium III  
  Memory: 512M, 4 nodes has 2G  
  Operating System: Red Hat Linux 7.2  
  Message Passing: LAM/MPI

- **Experiment Design:**
  
  - Randomly select four hard knapsack instances that generated based on the rule proposed in Martello(’90)  
  - Run five trials for each instance, and take the average  
  - Use COIN/Sbb (without cuts, heuristics) to produce serial result.
Knapsack: Preliminary Computational Results

- The four instances had similar behavior, so we present summary results.

<table>
<thead>
<tr>
<th>p</th>
<th>Wallclock</th>
<th>Ramp-up</th>
<th>Idle</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22m15s</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>254,151</td>
</tr>
<tr>
<td>4</td>
<td>4m56s</td>
<td>0%</td>
<td>2.9%</td>
<td>4.5</td>
<td>1.13</td>
<td>85 m</td>
</tr>
<tr>
<td>8</td>
<td>2m40s</td>
<td>0%</td>
<td>2.6%</td>
<td>8.3</td>
<td>1.04</td>
<td>85 m</td>
</tr>
<tr>
<td>16</td>
<td>1m34s</td>
<td>0%</td>
<td>7.8%</td>
<td>14.2</td>
<td>0.89</td>
<td>85 m</td>
</tr>
<tr>
<td>32</td>
<td>53s</td>
<td>0%</td>
<td>7.9%</td>
<td>26.3</td>
<td>0.83</td>
<td>85 m</td>
</tr>
</tbody>
</table>

- These indicate reasonable scalability, but for a very limited test.
- For up to 32 processors, 1 hub generally has better results than using 2 or more hubs.
Knapsack: Preliminary Computational Results

Scalability

Wallclock (Seconds) vs. Number of Processes

- Sum
- Ideal
For an application in which node processing times are much larger, we implemented **ALPS Branch and Cut** (ABC).

- **Consists of the classes:**
  - `AbcModel`,
  - `AbcTreeNode`,
  - `AbcNodeDesc`,
  - `AbcSolution`,
  - `AbcParameterSet`, and
  - other auxiliary classes

- Use COIN/Cgl cut generators.

- Use COIN/Sbb rounding heuristic as a primal heuristic.
ABC: Preliminary Computational Results

- Tested ABC using four MIPLIB problems: gesa3, blend2, fixnet6, cap6000

<table>
<thead>
<tr>
<th>Problem</th>
<th>Rows</th>
<th>Cols</th>
<th>Int Vars</th>
<th>Cont Vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>gesa3</td>
<td>1368</td>
<td>1152</td>
<td>720</td>
<td>504</td>
</tr>
<tr>
<td>blend2</td>
<td>274</td>
<td>353</td>
<td>264</td>
<td>89</td>
</tr>
<tr>
<td>fixnet8</td>
<td>478</td>
<td>878</td>
<td>378</td>
<td>500</td>
</tr>
<tr>
<td>cap6000</td>
<td>2176</td>
<td>6000</td>
<td>6000</td>
<td>0</td>
</tr>
</tbody>
</table>

- Search strategy: best-first
- Node selection: strong branching
- Test environment is the same as the previous.
ABC: Preliminary Computational Results

<table>
<thead>
<tr>
<th>Problem</th>
<th>p</th>
<th>Wallclock</th>
<th>Ramp-up</th>
<th>Idle</th>
<th>Speedup</th>
<th>$E$</th>
<th>nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>gesa3</td>
<td>1</td>
<td>27m6s</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>403</td>
</tr>
<tr>
<td>gesa3</td>
<td>4</td>
<td>10m14s</td>
<td>9.8%</td>
<td>0</td>
<td>2.6</td>
<td>0.66</td>
<td>445</td>
</tr>
<tr>
<td>gesa3</td>
<td>8</td>
<td>4m29s</td>
<td>35.1%</td>
<td>0.2%</td>
<td>6.0</td>
<td>0.76</td>
<td>337</td>
</tr>
<tr>
<td>gesa3</td>
<td>16</td>
<td>2m41s</td>
<td>49.1%</td>
<td>0.1%</td>
<td>10.1</td>
<td>0.63</td>
<td>247</td>
</tr>
<tr>
<td>blend2</td>
<td>1</td>
<td>26m5s</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>2339</td>
</tr>
<tr>
<td>blend2</td>
<td>4</td>
<td>4m18s</td>
<td>12.8%</td>
<td>0</td>
<td>6.1</td>
<td>1.53</td>
<td>1019</td>
</tr>
<tr>
<td>blend2</td>
<td>8</td>
<td>3m33s</td>
<td>14.0%</td>
<td>0.2%</td>
<td>7.3</td>
<td>0.92</td>
<td>717</td>
</tr>
<tr>
<td>blend2</td>
<td>16</td>
<td>2m9s</td>
<td>34.1%</td>
<td>0</td>
<td>12.1</td>
<td>0.76</td>
<td>980</td>
</tr>
<tr>
<td>fixnet6</td>
<td>1</td>
<td>45m16s</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>2729</td>
</tr>
<tr>
<td>fixnet6</td>
<td>4</td>
<td>11m43s</td>
<td>1.0%</td>
<td>0</td>
<td>3.9</td>
<td>0.98</td>
<td>3598</td>
</tr>
<tr>
<td>fixnet6</td>
<td>8</td>
<td>10m26s</td>
<td>3.0%</td>
<td>0.2%</td>
<td>4.3</td>
<td>0.54</td>
<td>4703</td>
</tr>
<tr>
<td>fixnet6</td>
<td>16</td>
<td>6m16s</td>
<td>4.6%</td>
<td>0</td>
<td>7.2</td>
<td>0.45</td>
<td>6570</td>
</tr>
<tr>
<td>cap6000</td>
<td>1</td>
<td>71m27s</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>6129</td>
</tr>
<tr>
<td>cap6000</td>
<td>4</td>
<td>22m24s</td>
<td>0.2%</td>
<td>0</td>
<td>3.2</td>
<td>0.80</td>
<td>9551</td>
</tr>
<tr>
<td>cap6000</td>
<td>8</td>
<td>16m52s</td>
<td>0.3%</td>
<td>0</td>
<td>4.2</td>
<td>0.53</td>
<td>12363</td>
</tr>
<tr>
<td>cap6000</td>
<td>16</td>
<td>10m40</td>
<td>1.2%</td>
<td>0.2%</td>
<td>6.7</td>
<td>0.42</td>
<td>14121</td>
</tr>
</tbody>
</table>
Future work

• Improving ALPS:
  – Reduction of ramp-up time, and
  – Elimination of redundant work.

• Develop the Branch, Constrain, and Price Software (BiCePS) library
  – Data handling layer
  – Support the implementation of parallel branch and bound algorithms
  – Integrate with OSI 2.

• Develop the BiCePS Linear Integer Solver (BLIS)
  – Add customization features akin to COIN/BCP