The COIN-OR High-Performance Parallel Search Framework (CHiPPS)

TED RALPHS
LEHIGH UNIVERSITY
YAN XU
SAS INSTITUTE

CPAIOR, June 15, 2010

Thanks: Work supported in part by the National Science Foundation and IBM
Introduction

Tree Search Algorithms
Parallel Computing
Previous Work

The CHiPPS Framework

Introduction
ALPS: Abstract Library For Parallel Search
BiCePS: Branch, Constrain, and Price Software
BLIS: BiCePS Linear Integer Solver

Applications
Knapsack Problem
Vehicle Routing

Results and Conclusions
Tree Search Algorithms

- Tree search algorithms systematically search the nodes of an acyclic graph for certain goal nodes.

- Tree search algorithms have been applied in many areas such as
  - Constraint satisfaction,
  - Game search,
  - Constraint Programming, and
  - Mathematical programming.
Elements of Tree Search Algorithms

- A generic tree search algorithm consists of the following elements:
  - **Processing method**: Is this a goal node?
  - **Fathoming rule**: Can node can be fathomed?
  - **Branching method**: What are the successors of this node?
  - **Search strategy**: What should we work on next?

- The algorithm consists of choosing a candidate node, processing it, and either fathoming or branching.

- During the course of the search, various information (knowledge) is generated and can be used to guide the search.
In general, the search tree can be very large.
The generic algorithm appears very easy to parallelize, however.

The appearance is deceiving, as the search graph is not generally known a priori and naïve parallelization strategies are not generally effective.
Parallel Overhead

- The amount of *parallel overhead* determines the scalability.

### Major Components of Parallel Overhead in Tree Search

- **Communication Overhead** (cost of sharing knowledge)
- **Idle Time**
  - Handshaking/Synchronization (cost of sharing knowledge)
  - Task Starvation (cost of *not* sharing knowledge)
  - Ramp Up Time
  - 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.
Previous Work

Previous tree search codes:

- Game tree search: **ZUGZWANG** and **APHID**
- Constraint programming: **ECLiPSe**, G12, etc.
- Optimization:
  - Commercial: **CPLEX**, Lindo, Mosek, SAS/OR, Xpress, etc.
  - Serial: **ABACUS**, bc-opt, COIN/CBC, GLPK, MINTO, SCIP, etc.
  - Parallel: **COIN/BCP**, **FATCOP**, **PARINO**, **PICO**, SYMPHONY, etc.

However, to our knowledge:

- Few studies of general tree search algorithms, and only one framework (**PIGSeL**).
- No study has emphasized scalability for *data-intensive* applications.
- Many packages are not open source or not easy to specialize for particular problem classes.
The COIN-OR High-Performance Parallel Search Framework

- CHiPPS has been under development since 2000 in partnership with IBM, NSF, and the COIN-OR Foundation.
- The broad goal was to develop a successor to SYMPHONY and BCP, two previous parallel MIP solvers.
- It consists of a hierarchy of C++ class libraries for implementing general parallel tree search algorithms.
- It is an open source project hosted by COIN-OR.
- Design goals
  - Scalability
  - Usability
The software discussed in this talk is available for free download from the Computational Infrastructure for Operations Research Web site

projects.coin-or.org/CHiPPS

The COIN-OR Foundation (www.coin-or.org)

- An non-profit educational foundation promoting the development and use of interoperable, open-source software for operations research.
- A consortium of researchers in both industry and academia dedicated to improving the state of computational research in OR.

The COIN-OR Repository

- A library of interoperable software tools for building optimization codes, as well as some stand-alone packages.
- A venue for peer review of OR software tools.
- A development platform for open source projects, including an SVN repository, project management tools, etc.
CHiPPS: Design Goals

- Intuitive object-oriented class structure.
  - AlpsModel
  - AlpsTreeNode
  - AlpsNodeDesc
  - AlpsSolution
  - AlpsParameterSet

- Minimal algorithmic assumptions in the base class.
  - Support for a wide range of problem classes and algorithms.
  - Support for constraint programming.

- Easy for user to develop a custom solver.

- Design for *parallel scalability*, but operate effective in a sequential environment.

- Explicit support for *memory compression* techniques (packing/differencing) important for implementing optimization algorithms.
CHiPPS: Overview of Features

- The design is based on a very general concept of *knowledge*.
- Knowledge is shared *asynchronously* through *pools* and *brokers*.
- Management overhead is reduced with the *master-hub-worker* paradigm.
- Overhead is decreased using *dynamic task granularity*.
- Two *static load balancing* techniques are used.
- Three *dynamic load balancing* techniques are employed.
- Uses *asynchronous* messaging to the highest extent possible.
- A scheduler on each process manages tasks like
  - node processing,
  - load balancing,
  - update search states, and
  - termination checking, etc.
**CHiPPS Library Hierarchy**

**ALPS** (Abstract Library for Parallel Search)
- search-handling layer
- prioritizes based on quality

**BiCePS** (Branch, Constrain, and Price Software)
- data-handling layer for relaxation-based optimization
- variables and constraints
- iterative bounding procedure

**BLIS** (BiCePS Linear Integer Solver)
- concretization of BiCePS
- linear constraints and objective

---

Ralphs & Xu

CHiPPS
**ALPS: Knowledge Sharing**

- All knowledge to be shared is stored in classes derived from a single base class and has an associated **encoded form**.
- Encoded form is used for **identification**, **storage**, and **communication**.
- Knowledge is maintained by one or more **knowledge pools**.
- The knowledge pools communicate through **knowledge brokers**.
ALPS: Master-Hub-Worker Paradigm

Master

Hubs

Workers
ALPS: Task Granularity

- Task granularity is a crucial element of parallel efficiency.
- In CHiPPS, each worker is capable of exploring an entire subtree autonomously.
- By stopping the search prematurely, the task granularity can be adjusted dynamically.
- As granularity increases, communication overhead decreases, but other sources of overhead increase.
ALPS: Synchronization

- As much as possible, we have eliminated handshaking and synchronization.
- A knowledge broker can work completely asynchronously, as long as its local node pool is not empty.
- This asynchronism can result in an **increase in the performance of redundant work**.
- To overcome this, we need good **load balancing**.
ALPS: Load Balancing

Static
- Performed at startup
- Two types
  - Two-level root initialization.
  - Spiral initialization.

Dynamic
- Performed periodically and as needed.
- Balance by quantity and quality.
- Keep subtrees together to enable differencing.
- Three types
  - Inter-cluster dynamic load balancing,
  - Intra-cluster dynamic load balancing,
  - Worker-initiated dynamic load balancing.
- Workers do not know each others’ workloads.
- Donors and receivers are matched at both the hub and master level.
- Three schemes work together to ensure workload is balanced.
ALPS: Class Hierarchy

ALPS: Abstract Library For Parallel Search
BiCePS: Branch, Constrain, and Price Software
BLIS: BiCePS Linear Integer Solver
BiCePS: Basic Notions

- BiCePS introduces the notion of \textit{variables} and \textit{constraints} (generically referred to as \textit{objects}).
- Objects are abstract entities with \textit{values} and \textit{bounds}.
- They are used to build mathematical programming \textit{models}.
- Search tree nodes consist of subproblems described by sets of variables and constraints.
- Key assumptions
  - Algorithm is relaxation-based branch-and-bound.
  - Bounding is an iterative procedure involving generation of variables and constraints.
BiCePS: Differencing Scheme

- Descriptions of search tree nodes can be extremely large.
- For this reason, subtrees are stored using a *differencing scheme*.
- Nodes are described using differences from the parent is this description is smaller.
- Again, there is a tradeoff between memory savings and additional computation.
- This approach requires keeping subtrees whole as much as possible.
- This impacts load balancing significantly.
BLIS: A Generic Solver for MILP

MILP

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

where \( A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, c \in \mathbb{R}^n, I \subseteq \{1, 2, \ldots, n\} \).

Basic Algorithmic Elements

- Search strategy.
- Branching scheme.
- Object generators.
- Heuristics.
BLIS: Branching Scheme

BLIS Branching scheme comprises three components:

- **Branching object**: has feasible region and can be branched on.
- **Branching candidate**:
  - created from objects not in their feasible regions or
  - contains instructions for how to conduct branching.
- **Branching method**:
  - specifies how to create a set of branching candidates.
  - has the method to compare objects and choose the best one.
BLIS: Constraint Generators

BLIS constraint generator:
- provides an interface between BLIS and the algorithms in COIN/Cgl.
- provides a base class for deriving specific generators.
- has the ability to specify rules to control generator:
  - where to call: root, leaf?
  - how many to generate?
  - when to activate or disable?
- contains the statistics to guide generating.
BLIS: Heuristics

BLIS primal heuristic:
- defines the functionality to heuristically search for solutions.
- has the ability to specify rules to control heuristics.
  - where to call: before root, after bounding, at solution?
  - how often to call?
  - when to activate or disable?
- collects statistics to guide the heuristic.
- provides a base class for deriving specific heuristics.
As a demonstration application, we implemented a solver for the knapsack problem using ALPS.

The solver uses the closed form solution of the LP relaxation as a bound.

Branching is on the fractional variable.

Implementation consists of deriving a few classes to specify the algorithm.

- **KnapModel**
- **KnapTreeNode**
- **KnapSolution**
- **KnapParams**

Once the classes have been implemented, the user writes a main function.

The only difference between parallel and serial code is the knowledge broker class that is used.
int main(int argc, char* argv[]) {
    KnapModel model;
#if defined(SERIAL)
    AlpsKnowledgeBrokerSerial knap(argc, argv, model);
#elif defined(PARALLEL_MPI)
    AlpsKnowledgeBrokerMPI knap(argc, argv, model);
#endif
    knap.registerClass("MODEL", new KnapModel);
    knap.registerClass("SOLUTION", new KnapSolution);
    knap.registerClass("NODE", new KnapTreeNode);
    knap.search();
    knap.printResult();
    return 0;
}
The Vehicle Routing Problem

The **VRP** is a combinatorial problem whose *ground set* is the edges of a graph \( G(V, E) \). Notation:

- \( V \) is the set of customers and the depot (0).
- \( d \) is a vector of the customer *demands*.
- \( k \) is the number of *routes*.
- \( C \) is the *capacity* of a truck.

A **feasible solution** is composed of:

- a partition \( \{R_1, \ldots, R_k\} \) of \( V \) such that \( \sum_{j \in R_i} d_j \leq C, \ 1 \leq i \leq k \);
- a permutation \( \sigma_i \) of \( R_i \cup \{0\} \) specifying the order of the customers on route \( i \).
Standard IP Formulation for the VRP

**VRP Formulation**

\[
\begin{align*}
\sum_{j=1}^{n} x_{0j} &= 2k \\
\sum_{j=1}^{n} x_{ij} &= 2 \quad \forall i \in V \setminus \{0\} \\
\sum_{i \in S} \sum_{j \not\in S} x_{ij} &\geq 2b(S) \quad \forall S \subset V \setminus \{0\}, \quad |S| > 1.
\end{align*}
\]

- \(b(S) = \text{lower bound}\) on the number of trucks required to service \(S\) (normally \(\lceil (\sum_{i \in S} d_i)/C \rceil\)).

- The number of constraints in this formulation is exponential.

- We must therefore generate the constraints dynamically.

- A solver can be implemented in BLIS by deriving just a few classes.
Implementing the VRP Solver

- The algorithm is defined by deriving the following classes.
  - VrpModel
  - VrpSolution
  - VrpCutGenerator
  - VrpHeuristic
  - VrpVariable
  - VrpsParams

- Once the classes have been implemented, the user writes a main function as before.
## Computational Results: Platforms

### Clemson Cluster
- **Machine:** Beowulf cluster with 52 nodes
- **Node:** dual core PPC, speed 1654 MHz
- **Memory:** 4G RAM each node
- **Operating System:** Linux
- **Message Passing:** MPICH

### SDSC Blue Gene System
- **Machine:** IBM Blue Gene with 3,072 compute nodes
- **Node:** dual processor, speed 700 MHz
- **Memory:** 512 MB RAM each node
- **Operating System:** Linux
- **Message Passing:** MPICH
KNAP Scalability on Difficult Instances

- Tested the 26 instances on the SDSC Blue Gene.
- The default algorithm was used except that
  - the static load balancing scheme is the two-level root initialization,
  - the number of nodes generated by the master varies from 10000 to 30000 depends on individual instance,
  - the number of nodes generated by a hub varies from 10000 to 20000 depends on individual instance,
  - the size a unit work is 300 nodes; and
  - multiple hubs were used.

<table>
<thead>
<tr>
<th>P</th>
<th>Node</th>
<th>Ramp-up</th>
<th>Idle</th>
<th>Ramp-down</th>
<th>Wallclock</th>
<th>Eff</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>14733745123</td>
<td>0.69%</td>
<td>4.78%</td>
<td>2.65%</td>
<td>6296.49</td>
<td>1.00</td>
</tr>
<tr>
<td>128</td>
<td>14776745744</td>
<td>1.37%</td>
<td>6.57%</td>
<td>5.26%</td>
<td>3290.56</td>
<td>0.95</td>
</tr>
<tr>
<td>256</td>
<td>14039728320</td>
<td>2.50%</td>
<td>7.14%</td>
<td>9.97%</td>
<td>1672.85</td>
<td>0.94</td>
</tr>
<tr>
<td>512</td>
<td>13533948496</td>
<td>7.38%</td>
<td>4.30%</td>
<td>14.83%</td>
<td>877.54</td>
<td>0.90</td>
</tr>
<tr>
<td>1024</td>
<td>13596979694</td>
<td>8.33%</td>
<td>3.41%</td>
<td>16.14%</td>
<td>469.78</td>
<td>0.84</td>
</tr>
<tr>
<td>2048</td>
<td>14045428590</td>
<td>9.59%</td>
<td>3.54%</td>
<td>22.00%</td>
<td>256.22</td>
<td>0.77</td>
</tr>
</tbody>
</table>

- KNAP scales well even when using several thousand processors.
- Ramp-up and ramp-down overhead inevitably increase.
### BLIS Scalability for Moderately Difficult Instances

- Selected 18 MILP instances from Lehigh/CORAL, MIPLIB 3.0, MIPLIB 2003, BCOL, and markshare.
- Tested on the Clemson cluster.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Nodes</th>
<th>Ramp-up</th>
<th>Idle</th>
<th>Ramp-down</th>
<th>Comm Overhead</th>
<th>Wallclock</th>
<th>Eff</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 P Per Node</td>
<td>11809956</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>33820.53</td>
<td>1.00</td>
</tr>
<tr>
<td>4P Per Node</td>
<td>11069710</td>
<td>0.03%</td>
<td>4.62%</td>
<td>0.02%</td>
<td>16.33%</td>
<td>10698.69</td>
<td>0.79</td>
</tr>
<tr>
<td>8P Per Node</td>
<td>11547210</td>
<td>0.11%</td>
<td>4.53%</td>
<td>0.41%</td>
<td>16.95%</td>
<td>5428.47</td>
<td>0.78</td>
</tr>
<tr>
<td>16P Per Node</td>
<td>12082266</td>
<td>0.33%</td>
<td>5.61%</td>
<td>1.60%</td>
<td>17.46%</td>
<td>2803.84</td>
<td>0.75</td>
</tr>
<tr>
<td>32P Per Node</td>
<td>12411902</td>
<td>1.15%</td>
<td>8.69%</td>
<td>2.95%</td>
<td>21.21%</td>
<td>1591.22</td>
<td>0.66</td>
</tr>
<tr>
<td>64P Per Node</td>
<td>14616292</td>
<td>1.33%</td>
<td>11.40%</td>
<td>6.70%</td>
<td>34.57%</td>
<td>1155.31</td>
<td>0.46</td>
</tr>
</tbody>
</table>
BLIS Scalability for Very Difficult Instances

- Tests on Clemson’s palmetto cluster (60 on the Top 500 list, 11/2008, Linux, MPICH, 8-core 2.33GHz Xeon/Opteron mix, 12-16GB RAM).
- Tests use one processor per node.
## Raw Computational Results

<table>
<thead>
<tr>
<th>Name</th>
<th>256</th>
<th>128</th>
<th>64</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcf2</td>
<td>926</td>
<td>1373</td>
<td>2091</td>
<td>43059</td>
</tr>
<tr>
<td>neos-1126860</td>
<td>2184</td>
<td>1830</td>
<td>2540</td>
<td>39856</td>
</tr>
<tr>
<td>neos-1122047</td>
<td>1676</td>
<td>1125</td>
<td>1532</td>
<td>NS</td>
</tr>
<tr>
<td>neos-1413153</td>
<td>4230</td>
<td>3500</td>
<td>2990</td>
<td>20980</td>
</tr>
<tr>
<td>neos-1456979</td>
<td>78.06%</td>
<td>NS</td>
<td>NS</td>
<td></td>
</tr>
<tr>
<td>neos-1461051</td>
<td>396</td>
<td>1082</td>
<td>536</td>
<td>NS</td>
</tr>
<tr>
<td>neos-1599274</td>
<td>1500</td>
<td>8108</td>
<td>9075</td>
<td></td>
</tr>
<tr>
<td>neos-548047</td>
<td>137.29%</td>
<td>376.48%</td>
<td>482%</td>
<td></td>
</tr>
<tr>
<td>neos-570431</td>
<td>1034</td>
<td>1255</td>
<td>1308</td>
<td>21873</td>
</tr>
<tr>
<td>neos-611838</td>
<td>712</td>
<td>956</td>
<td>886</td>
<td>8005</td>
</tr>
<tr>
<td>neos-612143</td>
<td>565</td>
<td>1716</td>
<td>1315</td>
<td>4837</td>
</tr>
<tr>
<td>neos-693347</td>
<td>1.28%</td>
<td>1.70%</td>
<td>NS</td>
<td></td>
</tr>
<tr>
<td>neos-912015</td>
<td>538</td>
<td>438</td>
<td>275</td>
<td>10674</td>
</tr>
<tr>
<td>neos-933364</td>
<td>6.67%</td>
<td>6.79%</td>
<td>11.80%</td>
<td></td>
</tr>
<tr>
<td>neos-933815</td>
<td>6.54%</td>
<td>8.77%</td>
<td>32.85%</td>
<td></td>
</tr>
<tr>
<td>neos-934184</td>
<td>6.67%</td>
<td>6.76%</td>
<td>9.15%</td>
<td></td>
</tr>
<tr>
<td>neos18</td>
<td>30.78%</td>
<td>30.78%</td>
<td>79344</td>
<td></td>
</tr>
</tbody>
</table>
## Speedups

<table>
<thead>
<tr>
<th>Name</th>
<th>256</th>
<th>128</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcf2</td>
<td>46.5</td>
<td>31.36</td>
<td>20.59</td>
</tr>
<tr>
<td>neos-1126860</td>
<td>18.25</td>
<td>21.78</td>
<td>15.69</td>
</tr>
<tr>
<td>neos-1413153</td>
<td>4.96</td>
<td>5.99</td>
<td>7.02</td>
</tr>
<tr>
<td>neos-1599274</td>
<td></td>
<td>6.05</td>
<td>1.12</td>
</tr>
<tr>
<td>neos-570431</td>
<td>21.15</td>
<td>17.43</td>
<td>16.72</td>
</tr>
<tr>
<td>neos-611838</td>
<td>11.24</td>
<td>8.37</td>
<td>9.03</td>
</tr>
<tr>
<td>neos-612143</td>
<td>8.56</td>
<td>2.82</td>
<td>3.68</td>
</tr>
<tr>
<td>neos-912015</td>
<td>19.84</td>
<td>24.37</td>
<td>38.81</td>
</tr>
</tbody>
</table>
## Efficiency

<table>
<thead>
<tr>
<th>Name</th>
<th>256</th>
<th>128</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcf2</td>
<td>0.18</td>
<td>0.25</td>
<td>0.32</td>
</tr>
<tr>
<td>neos-1126860</td>
<td>0.07</td>
<td>0.17</td>
<td>0.25</td>
</tr>
<tr>
<td>neos-1413153</td>
<td>0.02</td>
<td>0.05</td>
<td>0.11</td>
</tr>
<tr>
<td>neos-1599274</td>
<td></td>
<td>0.05</td>
<td>0.02</td>
</tr>
<tr>
<td>neos-570431</td>
<td>0.08</td>
<td>0.14</td>
<td>0.26</td>
</tr>
<tr>
<td>neos-611838</td>
<td>0.04</td>
<td>0.07</td>
<td>0.14</td>
</tr>
<tr>
<td>neos-612143</td>
<td>0.03</td>
<td>0.02</td>
<td>0.06</td>
</tr>
<tr>
<td>neos-912015</td>
<td>0.08</td>
<td>0.19</td>
<td>0.61</td>
</tr>
</tbody>
</table>
Our methods implemented in ALPS seem effective in improving scalability.

The framework is useful for implementing serial or parallel tree search applications.

The KNAP application achieves very good scalability.

There is still much room for improvement:
- load balancing,
- fault tolerance,
- hybrid architectures,
- grid enable.
BLIS

- The performance of BLIS in serial mode is favorable when compared to state of the art non-commercial solvers.
- The scalability for solving generic MILPs is highly dependent on properties of individual instances.
- Based on BLIS, applications like VRP/TSP can be implemented in a straightforward way.
- Much work is still needed
  - Callable library API
  - Support for column generation
  - Enhanced heuristics
  - Additional capabilities
Obtaining CHiPPS

The CHiPPS framework is available for download at

https://projects.coin-or.org/CHiPPS
Thank You!

Questions?