Computational Experience with Parallel Integer Programming using the CHiPPS Framework

TED RALPHS
Lehigh University
YAN XU
SAS Institute
LASZLO LADÁNYI
IBM ILOG
MATTHEW SALTZMAN
Clemson University

INFORMS Annual Conference, San Diego, CA
October 12, 2009

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

1. Introduction

2. The CHiPPS Framework
   - ALPS: Abstract Library for Parallel Search
   - BiCePS: Branch, Constrain, and Price Software
   - BLIS: BiCePS Linear Integer Solver

3. Computational Experiments
   - Measuring Performance
   - Computational Experiments
   - Conclusions
Quick Introduction to CHiPPS

- CHiPPS stands for COIN-OR High Performance Parallel Search.
- CHiPPS is a set of C++ class libraries for implementing tree search algorithms for both sequential and parallel environments.
- The generic algorithm appears very easy to parallelize.

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

ALPS (Abstract Library for Parallel Search)
- is the search-handling layer (parallel and sequential).
- provides various search strategies based on node priorities.

BiCePS (Branch, Constrain, and Price Software)
- is the data-handling layer for relaxation-based optimization.
- adds notion of variables and constraints.
- assumes iterative bounding process.

BLIS (BiCePS Linear Integer Solver)
- is a concretization of BiCePS.
- specific to models with linear constraints and objective function.
Previous Work

Previous tree search codes:

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

Parallelization of tree search have become an increasingly important and mainstream undertaking.

*All tree search codes will have to be parallelized to stay competitive.*
ALPS: 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.
ALPS: 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.
Knowledge Sharing

- All knowledge to be shared is 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.
Master-Hub-Worker Paradigm

Master

Hubs

Workers

Ralphs, Xu, Ladányi, & Saltzman
Parallel MIP with CHiPPS
Alps Class Hierarchy

The CHiPPS Framework
Computational Experiments

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

Ralphs, Xu, Ladányi, & Saltzman
Parallel MIP with CHiPPS
Using ALPS: A Knapsack Solver

The formulation of the binary knapsack problem is

\[
\text{max}\left\{ \sum_{i=1}^{m} p_i x_i : \sum_{i=1}^{m} s_i x_i \leq c, x_i \in \{0, 1\}, i = 1, 2, \ldots, m \right\},
\]

We derive the following classes:

- **KnapModel** (from AlpsModel): Stores the data used to describe the knapsack problem and implements `readInstance()`
- **KnapTreeNode** (from AlpsTreeNode): Implements `process()` (bound) and `branch()`
- **KnapNodeDesc** (from AlpsNodeDesc): Stores information about which variables/items have been fixed by branching and which are still free.
- **KnapSolution** (from AlpsSolution): Stores a solution (which items are in the knapsack).
BiCePS: Support for Relaxation-based Optimization

- Adds notion of **modeling objects** (variables and constraints).
- Models are built from sets of such objects.
- Bounding is an iterative process that produces new objects.
- A differencing scheme is used to store the difference between the descriptions of a child node and its parent.

```c++
struct BcpsObjectListMod {
    int numRemove;
    int* posRemove;
    int numAdd;
    BcpsObject **objects;
    BcpsFieldListMod<double> lbHard;
    BcpsFieldListMod<double> ubHard;
    BcpsFieldListMod<double> lbSoft;
    BcpsFieldListMod<double> ubSoft;
};
```

```c++
template<class T>
struct BcpsFieldListMod {
    bool relative;
    int numModify;
    int *posModify;
    T *entries;
};
```
BLIS: A Generic Solver for MILP

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

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

Basic Algorithmic Components

- Bounding method.
- Branching scheme.
- Object generators.
- Heuristics.
BLIS: Branching Scheme

BLIS Branching scheme comprise three components:

- **Object**: has feasible region and can be branched on.

- **Branching Object**:
  - is created from objects that do not lie in their feasible regions or objects that will be beneficial to the search if branching on them.
  - contains instructions for how to conduct branching.

- **Branching method**:
  - specifies how to create a set of candidate branching objects.
  - has the method to compare objects and choose the best one.
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 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.
Traditional Measures of Performance

- **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**: $E = S / N$

- Standard analysis considers change in efficiency on a fixed test set as number of processors is increased.
- **Isoefficiency analysis** considers the increase in problem size to maintain a fixed efficiency as number of processors is increased.
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.
Challenges in Measuring Performance

- Traditional measures are not appropriate.
  - The interesting problems are the ones that take too long to solve sequentially.
  - Need to account for the possibility of failure.

- It’s exceedingly difficult to construct a test set
  - Scalability varies substantially by instance.
  - Hard to know what test problems are appropriate.
  - A fixed test set will almost surely fail to measure what you want.

- Results are highly dependent on architecture
  - Difficult to make comparisons
  - Difficult to tune parameters

- Hard to get enough time on large-scale platforms for tuning and testing.

- Results are non-deterministic!
  - Determinism can be a false sense of security.
  - Lack of determinism requires more extensive testing.
## 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 for Moderately Difficult Instances

- Tested the moderately difficult instances at Clemson.
- The default algorithm was used except that
  - the static load balancing scheme was the two-level root initialization,
  - the number of nodes generated by the master was 3000, and
  - the size of a unit work was 300 nodes.

<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>4</td>
<td>193057493</td>
<td>0.28%</td>
<td>0.02%</td>
<td>0.01%</td>
<td>586.90</td>
<td>1.00</td>
</tr>
<tr>
<td>8</td>
<td>192831731</td>
<td>0.58%</td>
<td>0.08%</td>
<td>0.09%</td>
<td>245.42</td>
<td>1.20</td>
</tr>
<tr>
<td>16</td>
<td>192255612</td>
<td>1.20%</td>
<td>0.26%</td>
<td>0.37%</td>
<td>113.43</td>
<td>1.29</td>
</tr>
<tr>
<td>32</td>
<td>191967386</td>
<td>2.34%</td>
<td>0.71%</td>
<td>1.47%</td>
<td>56.39</td>
<td>1.30</td>
</tr>
<tr>
<td>64</td>
<td>190343944</td>
<td>4.37%</td>
<td>2.27%</td>
<td>5.49%</td>
<td>30.44</td>
<td>1.21</td>
</tr>
</tbody>
</table>

- Super-linear speedup observed.
- Ramp-up, ramp-down, and idle time overhead remains low.
Tested difficult instances on SDSC Blue Gene,
The default algorithm was used except that
- the static load balancing scheme was the two-level root initialization,
- nodes generated by the master varies from 10K to 30K.
- nodes generated by hub varies from 10K to 20K.
- the size of a unit of work was 3K nodes; and
- multiple hubs were used.

<table>
<thead>
<tr>
<th></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>

Note the inevitable increase in ramp-up and ramp-down.
KNAP scales well even when using several thousand processors.
A Simple Test

- 38 MILP instances from Lehigh/CORAL and MIPLIB3.
- Solved to optimality by using BLIS in 10 minutes.
- PC, 2.8 GHz Pentium, 2G RAM, Linux, COIN/Clp.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Time(No)</th>
<th>Memory(No)</th>
<th>Time(Yes)</th>
<th>Memory(Yes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcmulti</td>
<td>4.20 s</td>
<td>17.4 MB</td>
<td>4.19 s</td>
<td>1.4 MB</td>
</tr>
<tr>
<td>dsbmipt</td>
<td>33.28 s</td>
<td>34.1 MB</td>
<td>33.16 s</td>
<td>2.4 MB</td>
</tr>
<tr>
<td>egout</td>
<td>0.18 s</td>
<td>0.3 MB</td>
<td>0.18 s</td>
<td>0.2 MB</td>
</tr>
<tr>
<td>enigma</td>
<td>6.41 s</td>
<td>13.1 MB</td>
<td>6.16 s</td>
<td>2.3 MB</td>
</tr>
<tr>
<td>fiber</td>
<td>6.60 s</td>
<td>17.6 MB</td>
<td>6.56 s</td>
<td>2.1 MB</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### 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</td>
<td>11809956</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>33820.53</td>
<td>0.00286</td>
</tr>
<tr>
<td>Per Node</td>
<td></td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.00286</td>
<td></td>
</tr>
<tr>
<td>4P</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>0.00386</td>
</tr>
<tr>
<td>Per Node</td>
<td></td>
<td>0.03%</td>
<td>4.66%</td>
<td>0.00%</td>
<td>16.34%</td>
<td>0.00386</td>
<td></td>
</tr>
<tr>
<td>8P</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>0.00376</td>
</tr>
<tr>
<td>Per Node</td>
<td></td>
<td>0.10%</td>
<td>4.52%</td>
<td>0.53%</td>
<td>16.95%</td>
<td>0.00376</td>
<td></td>
</tr>
<tr>
<td>16P</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>0.00371</td>
</tr>
<tr>
<td>Per Node</td>
<td></td>
<td>0.27%</td>
<td>5.66%</td>
<td>1.62%</td>
<td>17.45%</td>
<td>0.00371</td>
<td></td>
</tr>
<tr>
<td>32P</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>0.00410</td>
</tr>
<tr>
<td>Per Node</td>
<td></td>
<td>1.22%</td>
<td>8.78%</td>
<td>2.93%</td>
<td>21.07%</td>
<td>0.00410</td>
<td></td>
</tr>
<tr>
<td>64P</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>0.00506</td>
</tr>
<tr>
<td>Per Node</td>
<td></td>
<td>1.38%</td>
<td>11.46%</td>
<td>6.72%</td>
<td>34.44%</td>
<td>0.00506</td>
<td></td>
</tr>
</tbody>
</table>

Ralphs, Xu, Ladányi, & Saltzman

Parallel MIP with CHiPPS
Impact of Problem Properties

- **Instance input150_1** is a knapsack instance. When using 128 processors, BLIS achieved super-linear speedup mainly to the decrease of the tree size.
- **Instance fc_30_50_2** is a fixed-charge network flow instance. It exhibits very significant increases in the size of its search tree.
- **Instance pk1** is a small integer program with 86 variables and 45 constraints. It is relatively easy to solve.

<table>
<thead>
<tr>
<th>Instance</th>
<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>input150_1</td>
<td></td>
<td>64</td>
<td>0.44%</td>
<td>3.38%</td>
<td>1.45%</td>
<td>1257.82</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>128</td>
<td>1.18%</td>
<td>6.90%</td>
<td>2.88%</td>
<td>559.80</td>
<td>1.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>256</td>
<td>1.62%</td>
<td>5.53%</td>
<td>7.02%</td>
<td>380.95</td>
<td>0.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td>512</td>
<td>3.81%</td>
<td>10.26%</td>
<td>10.57%</td>
<td>179.48</td>
<td>0.88</td>
</tr>
<tr>
<td>fc_30_50_2</td>
<td></td>
<td>64</td>
<td>0.15%</td>
<td>31.46%</td>
<td>9.18%</td>
<td>564.20</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>128</td>
<td>0.22%</td>
<td>33.25%</td>
<td>21.71%</td>
<td>399.60</td>
<td>0.71</td>
</tr>
<tr>
<td></td>
<td></td>
<td>256</td>
<td>0.23%</td>
<td>29.99%</td>
<td>28.99%</td>
<td>390.12</td>
<td>0.36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>512</td>
<td>0.27%</td>
<td>23.54%</td>
<td>29.00%</td>
<td>337.85</td>
<td>0.21</td>
</tr>
<tr>
<td>pk1</td>
<td></td>
<td>64</td>
<td>3.97%</td>
<td>12.00%</td>
<td>5.86%</td>
<td>103.55</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>128</td>
<td>11.66%</td>
<td>12.38%</td>
<td>10.47%</td>
<td>61.31</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td>256</td>
<td>11.55%</td>
<td>13.93%</td>
<td>20.19%</td>
<td>41.04</td>
<td>0.63</td>
</tr>
<tr>
<td></td>
<td></td>
<td>512</td>
<td>19.14%</td>
<td>9.07%</td>
<td>26.71%</td>
<td>36.43</td>
<td>0.36</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>6.05</td>
<td>1.12</td>
<td></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>
In October, 2007, the VRP/TSP solver won the Open Contest of Parallel Programming at the 19th International Symposium on Computer Architecture and High Performance Computing.
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.
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?