Doing It in Parallel (DIP) with The COIN-OR High-Performance Parallel Search Framework (CHiPPS)

TED RALPHS
Lehigh University

YAN XU
SAS Institute

LASZLO LADÁNYI
IBM T.J. Watson Research Center

MATTHEW SALTZMAN
Clemson University

Penn State University
April 10, 2008

Thanks: Work supported in part by the National Science Foundation
**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,
  - Artificial intelligence, 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.

Fortunately, 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.
Parallel Architectures

A **parallel computer** is a collection of processing elements that can cooperate to perform a task. Two broad architecture classes are:

- **Shared memory**
  - Each processor can access any memory module.
  - Software scales, hardware doesn’t.
  - A typical example is a *symmetric multiprocessor* (SMP).

- **Distributed memory**
  - Each processing unit has its own local memory and can only access its own memory directly.
  - Processing units communicate with each other via a network.
  - Hardware scales, software doesn’t.
  - Typical examples are *massively parallel processors* (MPP), *cluster*, and *computational grids*.

Recently, hybrids of these two architectures have become common.
Parallel Programming Tools

- **Low-level Tools**: *Sockets, threads, remote procedure calls.*
- **Parallelizing Compilers**: Compilers that automatically parallelize sequential programs.
- **APIs**: Standard programming interface for threading (primarily *OpenMP* on shared memory computers).
- **Parallel Languages**: Languages with parallel constructs such as *High Performance Fortran*.
- **Message Passing Libraries**: *MPI, PVM*, etc.
- **Grid Tools**: Tools for coordinating remote jobs across networks such as *Condor*.
A **programming paradigm** is a class of algorithms that have generally the same control structure. Common paradigms include:

- Task-Farming/Master-Worker
- Single-Program Multiple-Data
- Data Pipelining
Measuring Performance of a Parallel System

- **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

- **Parallel overhead** is a major cause of poor scalability.

### 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, 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.
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.
Algorithm Design Elements (Scalability)

For scalability, the main **objective** is to control overhead. Design issues fall into three broad categories:

- **Task Management**
  - Task granularity
  - Ramp up/Ramp down
  - Termination detection

- **Knowledge Management**
  - Sharing
  - Storage
  - Searching

- **Load balancing**
  - Static (mapping)
  - Dynamic
Algorithm Design Elements (Knowledge Management)

Knowledge is information generated during the search.
- Knowledge generation changes the shape of the tree dynamically.
- This makes load balancing difficult.
- Parallel tree search algorithms differ primarily in the way knowledge is shared (Trienekens ’92).

Sharing knowledge can improve efficiency.
- Helps eliminate the performance of redundant work.
- Helps avoid “task starvation.”
- Goal is for parallel search to mirror sequential search.

Sharing knowledge also increases communication overhead.
- This is the fundamental tradeoff.
Algorithm Design Elements (Usability)

- **Ease of use**
  - Intuitive class structure.
  - No need to understand implementation.

- **Generality**
  - Minimal algorithmic assumptions in base layer.
  - Specialized methods implemented in derived classes.

- **Extensibility**
  - Mechanism for defining new knowledge types.
  - Ability to develop custom applications.

- **Portability**
  - Coded in ANSI/ISO C++.
  - No dependance on architecture, operating system, or third-party software.
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, Ladányi, & Saltzman
DIP with CHiPPS
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**.

![Diagram showing knowledge sharing between processes A and B](image)
Master-Hub-Worker Paradigm

Master

Hubs

Workers
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 to be adjusted dynamically. As granularity increases, communication overhead decreases, but other sources of overhead increase.
Synchronization

- As much as possible, we have eliminated handshaking and synchronization.
- Knowledge brokers 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.
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, and
    - 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.
BiCePS: Basic Notions

- BiCePS introduces the notion of variables and constraints (generically referred to as objects).
- Objects are abstract entities with values and bounds.
- They are used to build mathematical programming 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: Branch, Cut, and Price

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

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.
Implementing a Knapsack Solver

- 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 $i$. 

Ralphs, Xu, Ladányi, & Saltzman

DIP with CHiPPS
Standard IP Formulation for the VRP

\[ \sum_{j=1}^{n} x_{0j} = 2k \]
\[ \sum_{j=1}^{n} x_{ij} = 2 \quad \forall i \in V \setminus \{0\} \]
\[ \sum_{i \in S} x_{ij} \geq 2b(S) \quad \forall S \subset V \setminus \{0\}, |S| > 1. \]

- \( b(S) \) = 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.
Bilevel Programming

- Real-world decision problems involve multiple decision-makers (DMs)
  - Independent
  - Conflicting objectives
- Assumption of single DM makes standard mathematical programs inadequate for analysis of hierarchical systems
- *Multilevel programming* overcomes this limitation
  - Variables divided into groups, controlled by separate DMs
  - Assumption of DM rationality allows hierarchy to be modeled as single optimization problem, from top-level perspective
Integer Bilevel Linear Programming

Let

- $x \in \mathbb{Z}^{n_1}$ represent the *upper-level variables*
- $y \in \mathbb{Z}^{n_2}$ represent the *lower-level variables*

**Integer Bilevel Linear Program**

$$
\max \{ c^1 x + d^1 y \mid x \in \mathcal{P}_U^I, y \in \arg\max\{d^2 y \mid y \in \mathcal{P}_L^I(x)\}\}
$$

The *upper- and lower-level feasible regions* are:

$$
\mathcal{P}_U^I = \{ x \in \mathbb{Z}^{n_1} \mid A^1 x \leq b^1, x \geq 0 \} \quad \text{and} \\
\mathcal{P}_L^I(x) = \{ y \in \mathbb{Z}^{n_2} \mid G^2 y \leq b^2 - A^2 x, y \geq 0 \} ,
$$

with respect to a given $x \in \mathbb{Z}^{n_1}$. 
Interdiction Problems

A special case of interest in this research is the *integer interdiction problem* (IPINT)

- Existing literature focuses on variants of network interdiction problem
- IPINT allows for lower-level systems described by ILPs

**Integer Interdiction**

\[
\min_{x \in \mathcal{P}_U} \max_{y \in \mathcal{P}_L(x)} \, dy
\]

where

\[
\mathcal{P}_U = \{ x \in \mathbb{B}^n \mid A^1 x \leq b^1 \}
\]

\[
\mathcal{P}_L(x) = \{ y \in \mathbb{Z}_+^n \mid G^2 y \leq b^2, y \leq u(e - x) \}
\]
Biobjective IPINT

Let
- $A^1_i$ denote the $i$th row of $A^1$
- $A^1_{-i}$ and $b^1_{-i}$ denote constraint matrix and RHS, after removing the $i$th row.

We can define a biobjective version of IPINT ($i = 1, \ldots, m_1$):

$$\min_{(x,y) \in \mathcal{F}^\text{INT}} [dy, A^1_i x]$$

(BIPINT$_i$)

where

$$\mathcal{F}^\text{INT} = \{(x,y) \mid \Omega^\text{INT}_{\text{proj}}, y \in M^\text{INT}(x)\}$$,

and

$$\mathcal{P}^\text{INT}_U = \{x \in \mathbb{B}^n \mid A^1_{-i} x \leq b^1_{-i}\}$$

$$\mathcal{P}^\text{INT}_L(x) = \{y \in \mathbb{Z}^p_+ \times \mathbb{R}^{n-p}_+ \mid G^2 y \leq b^2, y \leq u(e-x)\}$$

Applying a single-objective reformulation yields an IBLP.
Implementation

The Mixed Integer Bilevel Solver (MibS) has been developed to solve bilevel integer programming problems.

**COIN-OR Components Used**

- The **COIN High Performance Parallel Search Framework** (CHiPPS) to manage the branch and bound.
- The **COIN Branch and Cut** (CBC) framework for solving MILPs.
- The **COIN LP Solver** (CLP) framework for solving LPs.
- The **Cut Generation Library** (CGL) for generating valid inequalities.
- The **Open Solver Interface** (OSI) for interfacing with CBC and CLP.
Implementing MibS

As before, the main effort is in deriving a few classes to specify the algorithm.

- MibSModel
- MibSSolution
- MibSCutGenerator
- MibSVariable
- MibSParams

Once the classes have been implemented, the user writes a `main` function as before.
Sample Results

We can see the role that resources play in the effectiveness of an interdiction effort...

- Areas with steep slope suggest small increases in budget will yield substantial increases in effectiveness
- Areas with more gradual slope suggest resources may be better used elsewhere
DECOMP provides a flexible software framework for testing and extending various decomposition-based bounding methods with minimal user responsibility.

With most frameworks (BCP, ABACUS, MINTO, ...), the learning curve is steep since each model class requires a custom algorithm.

DECOMP breaks the usual paradigm of algorithm/model dependence.

Key: The user defines the decomposition in terms of the original model.

The framework takes care of the rest.
Traditional Decomposition Methods

- Decomposition methods
  - Cutting Plane Method (CPM) dynamically builds an outer approximation of the feasible region of a given relaxation. $Q''$.
  - Dantzig-Wolfe Method (DW) builds an inner approximation of the feasible region of a given relaxation.
  - Lagrangian Method (LD) obtains a bound by solving a penalized version of the relaxation.

- These three methods can be seen as equivalent.
- The user has only to define the relaxation to get implementations of all three, plus hybrids.
- DECOMP can be inserted as a bounding method to drive a branch-and-bound built in ALPS.
Test Machines

**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
Scalability of KNAP for Moderately Difficult Instances

- Tested the 10 instances in the moderately difficult set on the Clemson cluster.
- 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 is 3000, and
  - the size of a unit work is 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.
Scalability of KNAP for Very Difficult Instances

- Tested the 26 instances in the difficult set on the Blue Gene system.
- 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.
The Performance of Serial BLIS on Generic MILPs

Test Bed

- **Test Machine**: PC, 2.8 GHz Pentium, 2.0G RAM, Linux
- **Test instances**: Selected 33 instances from Lehigh/CORAL and MIPLIB 3 that both solvers can solve in 10 minutes.

- **BLIS** (serial version)
  - Branching strategy: Pseudocost branching.
  - Cuts generators: Gomory, Knapsack, Flow Cover, MIR, Probing, and Clique.
  - Heuristics: Rounding.

- **COIN/Cbc**
  - Branching strategy: Strong branching.
  - Cut generators: Gomory, Knapsack, Flow Cover, MIR, Probing, and Clique.
  - Heuristics: Rounding and Local search.
## Running Times

<table>
<thead>
<tr>
<th>Problem</th>
<th>Row</th>
<th>Column</th>
<th>Nonzero</th>
<th>Time-BLIS</th>
<th>Time-CBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>22433</td>
<td>198</td>
<td>429</td>
<td>3408</td>
<td>95.00</td>
<td>37.59</td>
</tr>
<tr>
<td>23588</td>
<td>137</td>
<td>368</td>
<td>3701</td>
<td>118.75</td>
<td>108.75</td>
</tr>
<tr>
<td>air03</td>
<td>124</td>
<td>10757</td>
<td>91028</td>
<td>36.45</td>
<td>7.84</td>
</tr>
<tr>
<td>aligninq</td>
<td>340</td>
<td>1831</td>
<td>15734</td>
<td>356.76</td>
<td>181.22</td>
</tr>
<tr>
<td>bell3a</td>
<td>123</td>
<td>133</td>
<td>347</td>
<td>49.21</td>
<td>38.49</td>
</tr>
<tr>
<td>dcmulti</td>
<td>290</td>
<td>548</td>
<td>1315</td>
<td>18.68</td>
<td>13.84</td>
</tr>
<tr>
<td>dsbmip</td>
<td>1182</td>
<td>1886</td>
<td>7366</td>
<td>55.30</td>
<td>38.66</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>qnet1</td>
<td>503</td>
<td>1541</td>
<td>4622</td>
<td>20.17</td>
<td>41.80</td>
</tr>
<tr>
<td>rgn</td>
<td>24</td>
<td>180</td>
<td>460</td>
<td>20.06</td>
<td>62.61</td>
</tr>
<tr>
<td>roy</td>
<td>162</td>
<td>149</td>
<td>411</td>
<td>23.79</td>
<td>7.56</td>
</tr>
<tr>
<td>stein27</td>
<td>118</td>
<td>27</td>
<td>378</td>
<td>25.15</td>
<td>9.78</td>
</tr>
<tr>
<td>vpm1</td>
<td>234</td>
<td>378</td>
<td>749</td>
<td>1.45</td>
<td>16.24</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>1642.61</strong></td>
<td><strong>1238.32</strong></td>
</tr>
</tbody>
</table>
Performance Profile

The Performance Profile chart compares the performance of CBC and BLIS algorithms. The x-axis represents the number of processors, ranging from 1 to 32, and the y-axis shows the performance ratio. CBC consistently outperforms BLIS across all processor counts, with CBC reaching the maximum performance ratio closer to the y-axis than BLIS. This indicates that CBC is more efficient in parallel computing environments.
Does Differencing Make a Difference?

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>dsbmip</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 (Generic MILPs)

- **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>Per Node</td>
<td></td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>0.00286</td>
<td></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>Per Node</td>
<td></td>
<td>0.00%</td>
<td>4.66%</td>
<td>0.00%</td>
<td>16.34%</td>
<td>0.00386</td>
<td></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>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 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>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 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>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 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>
<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>
BLIS Scalability (VRP instances)

- Performed this experiment on the Clemson Cluster.
- The default setting was used except that
  - search strategy was best-first, and
  - branching method was strong branching.

<table>
<thead>
<tr>
<th>P</th>
<th>Nodes</th>
<th>Ramp-up</th>
<th>Idle</th>
<th>Ramp-down</th>
<th>Wallclock</th>
<th>Eff</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>40250</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>19543.46</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>36200</td>
<td>7.06%</td>
<td>7.96%</td>
<td>0.39%</td>
<td>5402.95</td>
<td>0.90</td>
</tr>
<tr>
<td>8</td>
<td>52709</td>
<td>9.88%</td>
<td>6.15%</td>
<td>1.29%</td>
<td>4389.62</td>
<td>0.56</td>
</tr>
<tr>
<td>16</td>
<td>70865</td>
<td>14.16%</td>
<td>8.81%</td>
<td>3.76%</td>
<td>3332.52</td>
<td>0.37</td>
</tr>
<tr>
<td>32</td>
<td>96160</td>
<td>15.85%</td>
<td>10.75%</td>
<td>16.91%</td>
<td>3092.20</td>
<td>0.20</td>
</tr>
<tr>
<td>64</td>
<td>163545</td>
<td>18.19%</td>
<td>10.65%</td>
<td>19.02%</td>
<td>2767.83</td>
<td>0.11</td>
</tr>
</tbody>
</table>

- As the number of processes increases, the number of nodes increases significantly.
- Usually, good solutions were found very late. Should have some heuristics.
Cheering Up

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.
**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?