Benchmarking and Performance Analysis of Optimization Software

TED RALPHS
ISE Department
COR@L Lab
Lehigh University
ted@lehigh.edu

CPAIOR, Bologna, 15 June 2010
Outline

1. Introduction

2. Benchmarking
   - Purpose
   - Sequential Codes
   - Parallel Codes

3. Performance Analysis

4. Conclusions
My Hats

- Research Scientist
- Lab Director (COR@L)
- Software Developer (SYMPHONY, CHiPPS, DIP, CBC, MiBS, ...)
- Open Source Project Leader (COIN-OR)
- Educator
- Thesis Advisor
- Industry Consultant
Caveats

- This talk is heavily biased towards LP-based branch-and-bound algorithms for solving *mathematical programming problems*.
- In such a setting, results can be “messy.”
- Important aspects of this setting are that we have to account for
  - numerical error
  - failure of the algorithm to converge
- This talk contains a lot more questions than answers!
Background: COIN-OR

- The **Common Optimization Interface for Operations Research Initiative** was an initiative launched by IBM at ISMP in 2000.
- IBM seeded an open source repository with four initial projects and created a Web site.
- The goal was to develop the project and then hand it over to the community.
- The project has now grown to be self-sustaining and was spun off as a nonprofit educational foundation in the U.S. several years ago.
- The name was also changed to the **Computational Infrastructure for Operations Research** to reflect a broader mission.
What is COIN-OR Today?

The COIN-OR Foundation

- A non-profit 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.
- A venue for developing and maintaining standards.
- A forum for interaction and discussion of OR software.

The COIN-OR Repository

- A collection of interoperable software tools for building optimization codes, as well as a few stand-alone packages.
- A venue for peer review of OR software tools.
- A development platform for open source projects, including an SVN repository.
The COIN-OR Foundation

- The foundation has been up and running for more than five years.
- We have two boards.
  - A strategic board to set overall direction
  - A technical board to advise on technical issues
- The boards are composed of members from both industry and academia, as well as balanced across disciplines.
- Membership in the foundation is available to both individuals and institutions.
- The foundation Web site and repository are hosted by INFORMS.
My Hats: COIN-OR

- Member of Strategic Leadership Board
- Chair of Technical Leadership Council
- Project Manager
  - CoinBinary/CoinAll
  - SYMPHONY
  - CHiPPS
    - ALPS
    - BiCePS
    - BLIS
  - DIP
  - CBC
  - MiBS
What You Can Do With COIN

- We currently have 50+ projects and more are being added all the time.
- Most projects are now licensed under the EPL (very permissive).
- COIN has solvers for most common optimization problem classes.
  - Linear programming
  - Nonlinear programming
  - Mixed integer linear programming
  - Mixed integer nonlinear programming (convex and nonconvex)
  - Stochastic linear programming
  - Semidefinite programming
  - Graph problems
  - Combinatorial problems (VRP, TSP, SPP, etc.)
- COIN has various utilities for reading, building, and manipulating optimization models and feeding them to solvers.
- COIN has overarching frameworks that support implementation of broad algorithm classes.
  - Parallel search
  - Branch and cut (and price)
  - Decomposition-based algorithms
COIN-OR Projects Overview: Linear Optimization

- **Clp**: COIN LP Solver
  
  **Project Manager**: Julian Hall

- **Cbc**: COIN Branch and Cut
  
  **Project Manager**: T.R.

- **SYMPHONY**: a flexible integer programming package that supports shared and distributed memory parallel processing, biobjective optimization, warm starting, sensitivity analysis, application development, etc.
  
  **Project Manager**: T.R.

- **BLIS**: Parallel IP solver built to test the scalability of the CHiPPS framework.
  
  **Project Manager**: T.R.
COIN-OR Projects Overview: Nonlinear Optimization

- **Ipopt**: Interior Point OPTimizer implements interior point methods for solving nonlinear optimization problems.
  
  **Project Manager**: Andreas Wächter

- **Bonmin**: Basic Open-source Nonlinear Mixed INteger programming is for (convex) nonlinear integer programming.
  
  **Project Manager**: Pierre Bonami

- **Couenne**: Solver for nonconvex nonlinear integer programming problems.
  
  **Project Manager**: Pietro Belloti
COIN-OR Projects Overview: Modeling

- **FLOPC++**: An open-source modeling system.
  
  **Project Manager**: Tim Hultberg

- **PuLP**: Python-based modeling language for linear mathematical programs.
  
  **Project Manager**: Stu Mitchell

- **Pyomo**: Python-based modeling language for linear mathematical programs.
  
  **Project Manager**: Bill Hart
COIN-OR Projects Overview: Interfaces

- **Osi**: Open solver interface is a generic API for linear and mixed integer linear programs.
  
  **Project Manager**: Matthew Saltzman

- **GAMSlinks**: Allows you to use the GAMS algebraic modeling language and call COIN-OR solvers.
  
  **Project Manager**: Stefan Vigerske

- **CoinMP**: A callable library that wraps around CLP and CBC, providing an API similar to CPLEX, XPRESS, Gurobi, etc.
  
  **Project Manager**: Bjarni Kristjansson

- **Optimization Services**: A framework defining data interchange formats and providing tools for calling solvers locally and remotely through Web services.
  
  **Project Managers**: Jun Ma, Gus Gassmann, and Kipp Martin
COIN-OR Projects Overview: Frameworks

- **Bcp**: A generic framework for implementing branch, cut, and price algorithms.
  
  **Project Manager**: Laci Ladanyi

- **CHiPPS**: A framework for developing parallel tree search algorithms.
  
  **Project Manager**: T.R./Yan Xu

- **DIP**: A framework for implementing decomposition-based algorithms for integer programming, including Dantzig-Wolfe, Lagrangian relaxation, cutting plane, and combinations.
  
  **Project Manager**: T.R./Matthew Galati
COIN-OR Projects Overview: Miscellaneous

- **CppAD**: a package for doing algorithmic differentiation, a key ingredient in modern nonlinear optimization codes.
  
  **Project Manager**: Brad Bell

- **CSDP**: A solver for semi-definite programs
  
  **Project Manager**: Brian Borchers

- **DFO**: An algorithm for derivative free optimization.
  
  **Project Manager**: Katya Scheinburg
CoinAll, CoinBinary, BuildTools, and TestTools

- Many of the tools mentioned interoperate by using the configuration and build utilities provided by the BuildTools project.
- The BuildTools includes autoconf macros and scripts that allow PMs to smoothly integrate code from other projects into their own.
- The CoinAll project is an über-project that includes a set of mutually interoperable projects and specifies specific sets of versions that are compatible.
- The TestTools project is the focal point for testing of COIN code.
- The CoinBinary project is a long-term effort to provide pre-built binaries for popular platforms.
  - Installers for Windows
  - RPMs for Linux
  - .debs for Linux
- You can download CoinAll (source and/or binaries) here:
  - http://www.coin-or.org/download/source/CoinAll
  - http://www.coin-or.org/download/binary/CoinAll
Outline

1. Introduction
2. Benchmarking
   - Purpose
   - Sequential Codes
   - Parallel Codes
3. Performance Analysis
4. Conclusions
The Different Roles of Benchmarking

- Comparing performance of different codes
- Comparing performance of different versions of the same code
- Debugging software
- Setting a direction/goal for future research
- Tuning software
Academy versus Industry

- The role of benchmarking in academia is different than in the commercial sector.
  - **Commercial codes**: Primary goal is to **satisfy users**.
  - **Academic codes**: Primary goal is to **test ideas** and **generate papers**.

- The importance of software to the progress of academic research is evident.
- However, academic research is (unfortunately) still driven primarily by publication in archival journals.
- Software is difficult to evaluate as an intellectual product on its own merits.
- Developers are forced to publish papers in archival journals about software instead of publishing the software itself.
- Publications about software necessitate the use of benchmarks.
Developing/Maintaining Benchmarks

- Many academic test sets are developed in an ad hoc fashion specifically to support findings reported in a paper.
- Hence, they are essentially only vetted by the referees of the paper who may not even examine the test set closely.
- Once cited in a paper, the test set is established and may drive the research agenda.
- Many codes become tuned to the benchmark.
- This introduces undesirable biases into the literature.
- Fortunately, there are some exceptions.
The Role of Open Source

- Open source projects can play an important role in benchmarking.
- Reference implementations released in open source provide a well-understood baseline for comparison.
- Without such implementations, it is virtually impossible to do a properly designed and controlled experiment.
- Comparisons against black-box software are often not very meaningful.
- This was one of the central motivation for the founding of COIN-OR.
Within open source projects, benchmarking plays a role somewhere between academia and industry.

Since development is decentralized, benchmarking can provide an “early warning system” for problems.

As in industry, they can also make it easier to track progress.

There may still be a tendency to “develop to the benchmark” that has to be guarded against.

COIN-OR uses nightly builds and standard benchmarks to track development.
Issue 1: What is Really Being Tested?

- In general, the challenge is to test only a particular aspect of a given algorithm.
- To do so, we want to hold all other aspects of the algorithm constant.
- This is most easily accomplished when all experiments are done within a common software framework on a common experimental platform.
- Even in the most ideal circumstances, it can be difficult to draw conclusions.
  - Should the values of parameters be re-tuned?
  - Should “unrelated” parameter settings be held constant?
- How do you show that a new technique will be effective within a state-of-the-art implementation without access to the implementation?
Issue 2: How To Measure Performance?

Most papers in mathematical programming use measures such as

<table>
<thead>
<tr>
<th>Without time limit</th>
<th>With time limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Running time (wallclock or CPU?)</td>
<td>- Fraction solved (tolerance?)</td>
</tr>
<tr>
<td>- Tree size (which nodes to count?)</td>
<td>- Final gap (how measured?)</td>
</tr>
<tr>
<td></td>
<td>- Quality of solution (what is optimum?)</td>
</tr>
<tr>
<td></td>
<td>- Cost</td>
</tr>
<tr>
<td></td>
<td>- Feasibility</td>
</tr>
<tr>
<td></td>
<td>- Time to first solution (quality?)</td>
</tr>
</tbody>
</table>

Are these good choices? Probably not.
Issue 3: What Is a Fair Comparison?

- How do we really compare two different codes “fairly”?
- Codes may have inconsistent default parameters
  - Error tolerances
  - Gap tolerances
- Two codes claiming to have found an optimal solution may nevertheless produce a different optimal value.
- In the case of nonlinear optimization, we may also have to deal with the fact that codes can produce local optima.
- Details of implementation
  - Who implemented the code and how well is it optimized?
  - Are there differences in the implementation of common elements that are tangential to what is being tested?
For the foreseeable future, increases in computing power will come in the form of additional cores rather than improvements in clock speeds.

For this reason, most codes will need to be parallelized in some way to remain competitive.

All of the previously mentioned issues are brought into even greater contrast when benchmarking such codes.

In addition to traditional performance measures, we must also consider \textit{scalability}.

- What is it?
- What are the tradeoffs?
Parallel Scalability

- **Parallel scalability** measures how well an algorithm is able to take advantage of increased resources (primarily cores/processors).
- Generally, this is measured by executing the algorithm with different levels of available resources and observing the change in performance.
- The most clear-cut and often-cited measure is speedup, which measures time to optimality for different numbers of processors.
- This is not necessarily a relevant measure for real-world performance.
Traditional Measures of Performance

- **Parallel System**: Parallel algorithm + parallel architecture.
- **Scalability**: How well a parallel system takes advantage of increased computing resources.

<table>
<thead>
<tr>
<th>Terms</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential runtime</td>
<td>$T_s$</td>
</tr>
<tr>
<td>Parallel runtime</td>
<td>$T_p$</td>
</tr>
<tr>
<td>Parallel overhead</td>
<td>$T_o = NT_p - T_s$</td>
</tr>
<tr>
<td>Speedup</td>
<td>$S = T_s/T_p$</td>
</tr>
<tr>
<td>Efficiency</td>
<td>$E = S/N$</td>
</tr>
</tbody>
</table>

- Standard analysis considers change in efficiency on a fixed test set as number of processors is increased.
- This analysis is purely “compute-centric,” and does not take into account the effects of limitations on memory and storage.
Amdahl’s Law

- Amdahl’s Law postulates a theoretical limit on speed-up based on the amount of *inherently sequential* work to be done.
- If $s$ is the fraction of work to be done that is sequential, then efficiency on $p$ processors is limited to $s + (1 - s)/p$.
- In other words, efficiency is bounded by the sequential fraction $s$.
- In reality, there is no well-defined “sequential fraction.”
- The analysis also assumes a single, fixed test set.
- *Isoefficiency analysis* considers the increase in problem size to maintain a fixed efficiency as number of processors is increased.
- This is perhaps a more reasonable measure.
Parallel Overhead

- In practice, the amount of *parallel overhead* essentially determines the scalability.

**Major Components of Parallel Overhead in Tree Search**

- **Communication Overhead** (cost of sharing information)
- **Idle Time**
  - Handshaking/Synchronization (cost of sharing information)
  - Task Starvation (cost of *not* sharing information)
  - Ramp Up Time
  - Ramp Down Time
- **Performance of Redundant Work** (cost of *not* sharing information)

- Information sharing is the main driver of efficiency.
- There is a fundamental tradeoff between centralized and decentralized information storage and decision-making.
Architectures are getting more complex and each has its own bottlenecks.

- “Traditional” architectures are fast becoming extinct.
- Multi-core desktops are now common.
- Clusters of multi-core machines are becoming a standard.
- GPUs are still a bit unknown.

Performance is affected by

- Memory
- Bandwidth
- Latency

Ultimately, one can think of the architecture primarily in terms of an extended memory hierarchy.

Performance measures are only really valid for practically identical architectures.

It’s extremely difficult to extrapolate.
Challenges in Measuring Performance

- Traditional measures may not be 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 probably 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.
Sample Scalability Analysis

Solved difficult knapsack instances by branch and bound on SDSC Blue Gene,

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

<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>

Note the increase in ramp-up and ramp-down.
Scalability for Generic MILPs

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

<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>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</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>0.03%</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>0.78</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</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</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</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>
**Impact of Instance 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><code>input150_1</code></td>
<td>64</td>
<td>75723835</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>128</td>
<td>64257131</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>256</td>
<td>84342537</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>512</td>
<td>71779511</td>
<td>3.81%</td>
<td>10.26%</td>
<td>10.57%</td>
<td>179.48</td>
<td>0.88</td>
</tr>
<tr>
<td><code>fc_30_50_2</code></td>
<td>64</td>
<td>3494056</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>128</td>
<td>3733703</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>256</td>
<td>6523893</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>512</td>
<td>13358819</td>
<td>0.27%</td>
<td>23.54%</td>
<td>29.00%</td>
<td>337.85</td>
<td>0.21</td>
</tr>
<tr>
<td><code>pk1</code></td>
<td>64</td>
<td>2329865</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>128</td>
<td>2336213</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>256</td>
<td>2605461</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>512</td>
<td>3805593</td>
<td>19.14%</td>
<td>9.07%</td>
<td>26.71%</td>
<td>36.43</td>
<td>0.36</td>
</tr>
</tbody>
</table>
Properties Affecting Scalability

- Shape of search tree (balanced or not)
- Time to process a node
- Number/distribution of feasible solutions
- Relative strength of upper/lower bound (proving optimality)
- Sizes of node descriptions
Benchmarking Tests

Scalability can be tested separately from sequential performance.

**Scalability Tests**

- Test set with known optima (prove optimality)
- Instances known to have balanced trees
- Instances with small node processing times and large trees
- Instances with large node processing times and small trees
- Instances with large node descriptions
Alternative Measures of Parallel Performance

- Time to optimality may not be the most appropriate measure.
- Most interesting problems cannot be solved easily with small numbers of processors.

### Alternative Measures

- Final gap in fixed time
- Time to *prove* optimality (post facto)
- Time to target gap
- Time to target solution quality
- Time to target upper/lower bound
Tradeoffs

- How important is scalability versus sequential performance?
- The answer depends on the availability of computing resources.
- With large numbers of processors available, good scalability may overcome sub-standard performance.
- Keep in mind, however, that going on level deeper in a balanced tree doubles the size.
- Hence, parallelism is unlikely to be much of a silver bullet.
Outline

1. Introduction

2. Benchmarking
   - Purpose
   - Sequential Codes
   - Parallel Codes

3. Performance Analysis

4. Conclusions
Performance Tuning

- One of the goals of benchmarks is performance tuning.
- Does the information used to benchmark help us to tune?
- Not really, we need more in-depth analysis.
- This section focuses on branch and bound algorithms generally.
Assessing the Performance of B&B

- Benchmarking focuses on aggregate measures, but these measures are not very useful for performance tuning.
- Most commercial and open-source solvers report:
  - optimality gap (global lower and upper bound)
  - number of candidate nodes
  - statistics to indicate use/effectiveness of various components of the algorithm
    - Preprocessing
    - Cutting plane generators
    - Primal heuristics
- These are ultimately not very useful in identifying strategies for performance improvement.
Optimality gap

- **Strength**: Gives indication of quality of solution
- **Strength**: Nonincreasing measure
- **Weakness**: may remain constant for long periods, then drop suddenly
Number of active nodes

- **Strength:** Indicates “work done” and “work remaining.”
- **Weakness:** may go up and down
- **Weakness:** each active node counts equally
Deeper Analysis

In principle, there is a wealth of additional information available that can be used to visualize performance.

- Number of nodes of different statuses
  - Candidate
  - Infeasible
  - Branched
  - Fathomed

- For each “feasible” node:
  - LP relaxation value
  - Integer infeasibility
  - History/position in tree (e.g., depth and parent)
  - Statistics about methods applied

How can we use this information to better assess performance?
The Branch and Bound Analysis Kit (BAK)

- Works with any instrumented solver (currently open-source solvers GLPK, SYMPHONY, and CBC).
- Solver must be modified to provide output when nodes are added and processed.
- A processing script creates visual representations of the data by parsing the output file
  - Output file can be processed at any point during the solving process
  - Parsing is done in Python, images are created with Gnuplot
- Available for download at http://www.rosemaryroad.org/brady/software/index.html
Example of output from solver

# CBC
0.040003 heuristic -28.000000
2.692169 branched 0 -1 N -39.248099 16 0.169729
2.692169 pregnant 2 0 R -39.248063 14 105.991922
2.708170 pregnant 3 0 L -38.939929 6 0.105246
2.764173 pregnant 5 2 R -39.244862 12 49.115388
2.764173 branched 2 0 R -39.248063 14 105.991922
Visual Representations

- Histogram of active node LP bounds
- Scatter plot of active node LP bounds & integer infeasibility
- Incumbent node history in scatter plot
- B&B trees showing the LP bound of each node
Visualization tools: Histogram of active node LP bounds

- Horizontal axis is the LP bound
- Vertical axis is number of active nodes
- Green vertical line shows the current incumbent value and the blue one
Example histogram series 1: 1152lav
Example histogram series 1: l152lav
Example histogram series 1: 1152lav
Example histogram series 1: 1152lav
Example histogram series 1: 1152lav
Example histogram series 1: l152lav
Example histogram series 1: 1152lav
Example histogram series 1: 1152lav
Example histogram series 1: 1152lav

![Histogram of Objective Values 008](image.png)
Example histogram series 1: l152lav
Example histogram series 1: 1152lav
Example histogram series 1: 1152lav
Example histogram series 1: l152lav
Example histogram series 1: l152lav
Example histogram series 1: l152lav
Example histogram series 1: 1152lav
Example histogram series 1: 1152lav
Example histogram series 2: swath

histogram of objective values 000

T.K. Ralphs  Benchmarking
Example histogram series 2: swath
Example histogram series 2: swath
Example histogram series 2: swath
Example histogram series 2: swath
Example histogram series 2: swath
Example histogram series 2: swath
Example histogram series 2: swath
Example histogram series 2: swath
Example histogram series 2: swath
Example histogram series 2: swath
Example histogram series 2: swath
Example histogram series 2: swath
Example histogram series 2: swath
Example histogram series 2: swath
Example histogram series 2: swath
Visualization tools: Scatter plot

- Horizontal axis is the integer infeasibility
- Vertical axis is the LP bound
- Green horizontal line is the current incumbent value
Example scatter plot series 1: swath
Example scatter plot series 1: swath
Example scatter plot series 1: swath
Example scatter plot series 1: swath
Example scatter plot series 1: swath
Example scatter plot series 1: swath
Example scatter plot series 1: swath
Example scatter plot series 1: swath
Example scatter plot series 1: swath
Example scatter plot series 1: swath
Example scatter plot series 1: swath
Example scatter plot series 1: swath
Example scatter plot series 1: swath
Example scatter plot series 1: swath
Example scatter plot series 1: swath
Example scatter plot series 1: swath
Patterns in integer infeasibility: SYMPHONY
Patterns in integer infeasibility: SYMPHONY

Scatterplot: mod008, SYMPHONY, 42s
Visualization tools: Incumbent node history in scatter plot

- Horizontal axis is the integer infeasibility
- Vertical axis is the LP bound
- Green line shows ancestors of the incumbent node
Example incumbent node history series 1: l152lav
Example incumbent node history series 1: l152lav

path of incumbents 001
Example incumbent node history series 2: liu
Visualization tools: B&B trees

- Vertical axis is the LP bound
- Nodes are horizontally positioned to make the pictures more readable
- Alternatively, horizontal positions may be fixed based on position in the tree
Visualization tools: B&B trees

Node color legend:
- green: branched
- yellow: candidate or pregnant
- red: fathomed
- blue: infeasible
Example B&B trees
Example B&B trees
Example B&B trees
Example B&B trees

B&B tree (dataset2per8inv10.dat 11s )
Example B&B trees

B&B tree (dataset2per8inv10.dat 3s )
Example B&B trees

B&B tree (dataset2per8inv10.dat 3s)
Example B&B trees
Example B&B trees
Example B&B tree series 1: 1152lav
Example B&B tree series 1: 1152lav
Example B&B tree series 1: l152lav
Example B&B tree series 1: l152lav
Example B&B tree series 1: l152lav
Example B&B tree series 1: l152lav
Example B&B tree series 1: 1152lav
Example B&B tree series 1: 1152lav
Example B&B tree series 1: 1152lav
Example B&B tree series 1: l152lav
Example B&B tree series 1: 1152lav
Example B&B tree series 1: 1152lav
Example B&B tree series 1: 1152lav
Example B&B tree series 1: 1152lav
Example B&B tree series 1: l152lav
Example B&B tree series 1: 1152lav
Example B&B tree series 1: 1152lav
Example B&B tree series 2: liu
Example B&B tree series 2: liu
Example B&B tree series 2: liu
Example B&B tree series 2: liu
Example B&B tree series 2: liu
Example B&B tree series 2: liu
Example B&B tree series 2: liu
Example B&B tree series 2: liu
Example B&B tree series 3
Example B&B tree series 3

B&B tree (dataset2per8inv10.dat 3s )
Example B&B tree series 3

B&B tree (dataset2per8inv10.dat 7s )
Example B&B tree series 3

B&B tree (dataset2per8inv10.dat 11s)

-38
-36
-34
-32
-30
-28

obj. value
Example B&B tree series 3

B&B tree (dataset2per8inv10.dat 11s)
Example B&B tree series 3
Example B&B tree series 3

B&B tree (dataset2per8inv10.dat 19s)
Example B&B tree series 3
Example B&B tree series 3
Example B&B tree series 3
Example B&B tree series 3
Outline

1. Introduction
2. Benchmarking
   - Purpose
   - Sequential Codes
   - Parallel Codes
3. Performance Analysis
4. Conclusions

T.K. Ralphs Benchmarking
Other Tools

- Performance profiles
- Hudson (https://software.sandia.gov/hudson/)
- Hans Mittelman’s Optimization Benchmarks (http://plato.asu.edu/bench.html)
- STOP (http://www.rosemaryroad.org/brady/software/index.html)
Final Remarks

- Benchmarking must be done with extreme care, especially with parallel codes.
- Open source can play a critical role in allowing researchers to carry out properly designed and controlled experiments.
- Please consider putting your codes into the COIN-OR repository or elsewhere for others to build on.