Assessing Performance of Parallel MILP solvers

Ted Ralphs$^1$
Stephen J. Maher$^2$, Yuji Shinano$^3$

$^1$COR@L Lab, Lehigh University, Bethlehem, PA USA $^2$Lancaster University, Lancaster, UK $^3$Zuse Institute Berlin, Berlin, Germany

International Symposium on Mathematical Programming
Bordeaux, France, 5 July 2018
Assessing “Effectiveness”

Fundamental questions we would like to answer.

- How to assess algorithm “effectiveness”?
- How to compare solvers with disparate capabilities?
- How to isolate determinants of “effectiveness”?

These simple questions are surprisingly difficult to answer!

- What do we mean by one solver being “better” than another?
- What is a “fair” way to test?
- What do we even mean by effectiveness?

Can we answer these questions by observation without (much) instrumentation?
What Is “Effectiveness”? 

- We carefully distinguish between two aspects of what we generally (and loosely) refer to as the “effectiveness” of an algorithm.
- The *computing system* is the combination of a software stack and hardware platform and is a crucial element.
- The *resources* are elements of the system that can be dynamically allocated to a computation (e.g., processors, time).

**Performance**
- The raw ability of an algorithm to solve an instance or set of instances on a given computing system with *fixed resources*.
- Measures requirement for one resource if others are held constant.

**Scalability**
- Property of an algorithm deployed in dynamic environment.
- Measures the *tradeoff* between resources (processors and time).
Focus of this talk will be primarily on assessing scalability.
Due to time constraints, I’ll skip many definitions.
There is a full paper that goes into much more detail.
Tree search is easy to parallelize in principle...

- Most straightforwardly, we can parallelize the while loop.
- Naively, this means processing multiple nodes in parallel.
- Applying the successor function turns one task into two!
- This seems to be what is called “embarrassingly parallel”...
- ...but sadly, it’s closer to embarrassingly difficult to parallelize!
- We’re aiming at a moving target...and with conflicting goals.
Nice Trees
Barriers to Scalability: Sophistication of Solvers

- A vast amount of effort has gone into improving the performance of sequential solvers over the past several decades.

- It’s been estimated that overall solver performance has improved by a factor of approximately 2 trillion in past decades.

- Unfortunately, major advances in solver technology have mostly made achieving parallel performance more difficult.
  
  - Solvers are increasingly tightly integrated.
  - Work done at the root node is difficult to parallelize.
  - Algorithmic focus is on reducing the amount of enumeration.
  - Solvers exploit a lot of useful “global” knowledge.

Branch and cut is not nearly as parallelizable as it seems!
We consider the binary knapsack problem:

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

We implemented a naive LP-based branch-and-bound in the Abstract Library for Parallel Search (ALPS).

<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>

Perfect scalability! But terrible performance...
On the Other Hand

- CPLEX output for solving one of these instances...

Root node processing (before b&c):
  Real time = 0.01 sec. (0.76 ticks)
Sequential b&c:
  Real time = 0.00 sec. (0.00 ticks)
--------
Total (root+branch&cut) = 0.01 sec. (0.76 ticks)

Root node processing (before b&c):
  Real time = 0.03 sec. (0.74 ticks)
Parallel b&c, 16 threads:
  Real time = 0.00 sec. (0.00 ticks)
  Sync time (average) = 0.00 sec.
  Wait time (average) = 0.00 sec.
--------
Total (root+branch&cut) = 0.03 sec. (0.74 ticks)

- Parallel slowdown! But great performance...

Ralphs et.al. (COR@L Lab)
Outline

1 Introduction

2 Measures of Effectiveness
   - Performance
   - Scalability

3 Performance Analysis
   - Challenges
   - Alternatives to Classical Analysis
   - Sample Computational Results

4 Conclusions
Measures of Performance

Single-instance measures
- Time to proven optimality
- Number of nodes to proven optimality
- Time to first feasible solution
- Time to fixed gap
- Gap or primal bound after a time limit
- Primal dual integral (PDI)

Summary Measures
- Mean
- Shifted geometric mean
- Performance profile
- Performance plots
- Histograms
Primal Dual Integral [Berthold, 2013]

Figure: Example of a PDI plot

Objective Function Value

Computing Time (sec.)

1: Normalized

Ralphs et.al. (COR@L Lab)
Performance Assessment
The Extended Primal Dual Integral

- The PDI can be inaccurate or nonexistent.
  - No primal (or dual) bounds are known.
  - Primal and dual bounds have different signs.
  - Optimal value is zero.
- Berthold [2013] proposed setting gap to 1 in the first two situations.
  - Note that if the gap stays constant throughout the computation, then it will be normalized to 1 and the PDI will equal the computing time.
  - The method of Berthold [2013] is the same as assuming that the gap is constant through the first part of the computation.
  - This can significantly distort the PDI in some cases.
- The *extended PDI* is our proposal for partially overcoming these issues.
Outline

1. Introduction

2. Measures of Effectiveness
   - Performance
   - Scalability

3. Performance Analysis
   - Challenges
   - Alternatives to Classical Analysis
   - Sample Computational Results

4. Conclusions
Parallel Overhead

- The amount of parallel overhead determines the scalability.
- “Knowledge sharing” is the main driver of efficiency.

### 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)
  - Memory Contention
  - Ramp Up Time
  - Ramp Down Time
- **Performance of Redundant Work** (cost of not sharing knowledge)

This breakdown highlights the tradeoff between centralized and decentralized knowledge storage and decision-making.

Ralphs et.al. (COR@L Lab)  Performance Assessment
Parallel Efficiency

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 cores is increased.
- Isoefficiency analysis considers the increase in problem size to maintain a fixed efficiency as number of cores is increased.
Outline

1. Introduction

2. Measures of Effectiveness
   - Performance
   - Scalability

3. Performance Analysis
   - Challenges
   - Alternatives to Classical Analysis
   - Sample Computational Results

4. Conclusions
Outline

1. Introduction

2. Measures of Effectiveness
   - Performance
   - Scalability

3. Performance Analysis
   - Challenges
   - Alternatives to Classical Analysis
   - Sample Computational Results

4. Conclusions
Challenges in Analyzing Scalability

- It’s exceedingly difficult to construct a test set.
  - Problems need to be solvable by all solvers on single core.
  - Single-core running times should be “long, but not too long”
  - Scalability depends on many factors besides the algorithm itself, including inherent properties of the instances.
  - Different instances scale differently on different solvers.
- It’s not clear what the baseline should be.
  - The best known sequential algorithm,
  - The parallel algorithm running on a single core,
  - Or...?
- It’s difficult to compare solvers with widely varying performance.
- Results are highly dependent on details of the parallel system.
- Variability in execution!
  - Many sources of variability are difficult to control for.
  - Lack of determinism requires extensive testing.
- Scalability numbers alone don’t typically give much insight!

Ralphs et.al. (COR@L Lab)
Outline

1. Introduction

2. Measures of Effectiveness
   - Performance
   - Scalability

3. Performance Analysis
   - Challenges
   - Alternatives to Classical Analysis
   - Sample Computational Results

4. Conclusions
Alternatives to Classical Scalability Analysis

- Direct measures of overhead.
  - Node throughput
  - Ramp-up/Ramp-down time
  - Idle time/Lock time/Wait time
  - Number of nodes

- Performance, scalability, and cumulative profiles.

- Analysis based on measures of progress.
  - Gap
  - Extended PDI

Ralphs et.al. (COR@L Lab)  Performance Assessment
Performance profiles are typically used to compare different algorithms.

They can, however, be used to compare the same algorithm under different conditions.

For scalability, we compare with differing numbers of threads.

A down side is that performance profiles compare to virtual best, whereas scalability compares to single-thread.
Scalability Profiles

- Straight performance profile considers ratios against virtual best.
- An alternative is to consider ratios against single thread.
- In the latter case, we must allow ratios less than one.

Figure: Scalability profile of wallclock running time.
Progress-based Analysis

- Traditional scalability analysis asks **how much time it takes to do a fixed computation.**

  Two simple alternatives

  - How much computation can be done in a **fixed amount of real time** but with **varying numbers of processors**?
  
  - How much computation can be done with **fixed compute time** but with **varying amounts of real time**?

- Allowing partial completion of a fixed computation eliminates many of the problems with finding a test set and comparing solvers.

- Both these alternatives depends on having some reliable “measure of progress,” however.

- It is not enough to just measure the “amount of computation”—this is equivalent to measuring utilization and ignoring other overhead.
Measures of Progress

- A *measure of progress* is an estimate of what fraction of a computation has been completed.
- It may be very difficult to predict how much time remains in a computation.
- However, for computations that have already been performed once, it may be possible.
- Measures of progress can be used to assess the effectiveness of algorithms *even if the computation doesn’t complete* ⇐ Important!

Possible measures for MILP

- Gap
- *Extended PDI*
Gap versus Extended PDI

- **Gap**
  - Final value is always zero
  - Progress can be “irregular”.
  - Current value doesn’t really indicate how “close” the computation is to finishing.

- **Extended PDI**
  - Final value can be anything from 0 to the time required for computation (normalized version).
  - Can be normalized to $[0, 1]$, but the final value is still variable.
  - Progress can be “irregular”.
  - Still, it seems to be a reasonable proxy for wallclock running time.

Ralphs et al. (COR@L Lab)
Performance Assessment
The below figures show the relationship between wallclock running time and extended PDI for different numbers of threads. In general, there is a strong correlation between wallclock and PDI, which is perhaps not very surprising. Extended PDI may thus be a reasonable measure of progress.

(a) ParaSCIP

(b) SYMPHONY

Figure: The relationship between the wall clock time and the extended PDI.
Performance Profiles of Extended PDI and Wallclock

Comparing Primal Dual Integral

Comparing Total Wall Clock Time
Outline

1. Introduction

2. Measures of Effectiveness
   - Performance
   - Scalability

3. Performance Analysis
   - Challenges
   - Alternatives to Classical Analysis
   - Sample Computational Results

4. Conclusions
Sample Computational Results

- We have been experimenting with a number of ways of applying the ideas seen so far.
- In the following, we show results with the following solvers.
  - Gurobi
  - ParaSCIP [Shinano et al., 2013]
  - SYMPHONY [Ralphs and Güzelsoy, 2005]
  - ALPS [Xu et al., 2007]
Performance Profile Using Extended PDI

Figure: Performance profile of PDI for ParaSCIP on MIPLIB2010.
Figure: Scalability profile of the extended PDI
Scalability Profile with Fixed Compute Time

Performance profile comparing the primal-dual integral with limited resources

(a) ParaSCIP

(b) SYMPHONY

Figure: The scalability profile of PDI with fixed compute time.
Node Throughput Scalability Profile

Performance profile comparing NodeThroughput

- **sympathy**
- **parascip**

1 Thread
4 Threads
8 Threads
16 Threads

Ralphs et.al. (COR@L Lab) Performance Assessment
Number of Nodes Scalability Profile

Performance profile comparing NumberOfNodes

Ralphs et.al. (COR@L Lab) Performance Assessment
Number of Nodes at Gap Scalability Profile

Performance profile comparing NodesAtGap_25

- parascip
- symphony

1 Thread
4 Threads
8 Threads
16 Threads

Ratio to best setting

Fraction of instances

Performance profile comparing NodesAtGap_25

- parascip
- symphony

1 Thread
4 Threads
8 Threads
16 Threads
Outline

1 Introduction

2 Measures of Effectiveness
   • Performance
   • Scalability

3 Performance Analysis
   • Challenges
   • Alternatives to Classical Analysis
   • Sample Computational Results

4 Conclusions
Conclusions

- What has been presented here is just a proposal meant to start a discussion within our community.
- These visualizations are not the end of the story, they may just indicate where to dig for more information.
- We are continuing with this long-term project to analyze the differences in the many existing approaches to parallel MIP.
- Feedback appreciated!
- For more details, see
  - Koch et al. [2012]
  - Ralphs et al. [2016]
References I


References II

