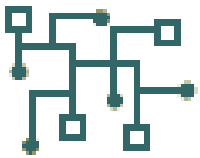
Assessing Performance of Parallel MILP Solvers
How Are We Doing, Really?

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

SIAM Conference on Optimization, 23 May 2017



ISE Industrial and Systems Engineering

COR@L
COMPUTATIONAL OPTIMIZATION RESEARCH AT LEHIGH

Outline

Outline

Assessing Performance

- Fundamental questions we would like to answer

  - **How well are we doing?**
  - **How does solver A compare to solver B?**
  - **What are the main drivers of parallel performance?**

- These questions are surprisingly difficult to answer!
  - What do we mean by one solver being “better” than another?
  - What is a fair way to test?
  - How can we isolate the different factors affecting overall performance?
- Can we answer these questions by observation without (much) instrumentation?

Tree Search

Tree search is an algorithmic framework for exploring an implicitly defined tree to find one or more *goal nodes*. Tree is specified by

- a *root node* and
- a *successor function*.

Algorithm 1: A Generic Tree Search Algorithm

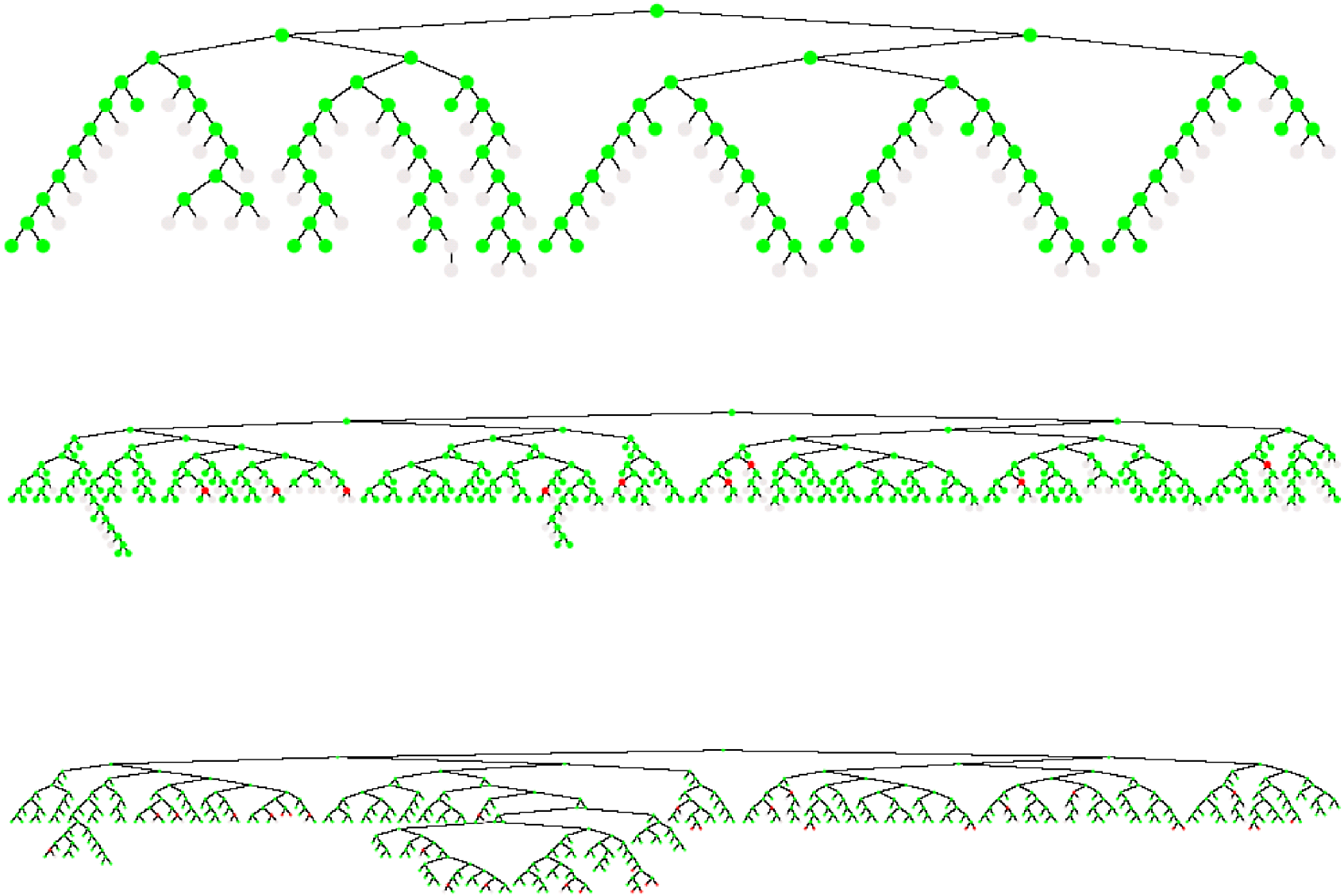
```
1  Add root node r to a priority queue Q.
2  while Q is not empty do
3      Choose a node i from Q.
4      Process the node i.
5      Apply pruning rules (can i or a successor be a goal node?)
6      if Node i can be pruned then
7          Prune (discard) node i (save i if it may be a goal node).
8      else
9          Apply successor function to node i (Branch)
10         Add the successors to Q.
```

Parallelization of Tree Search

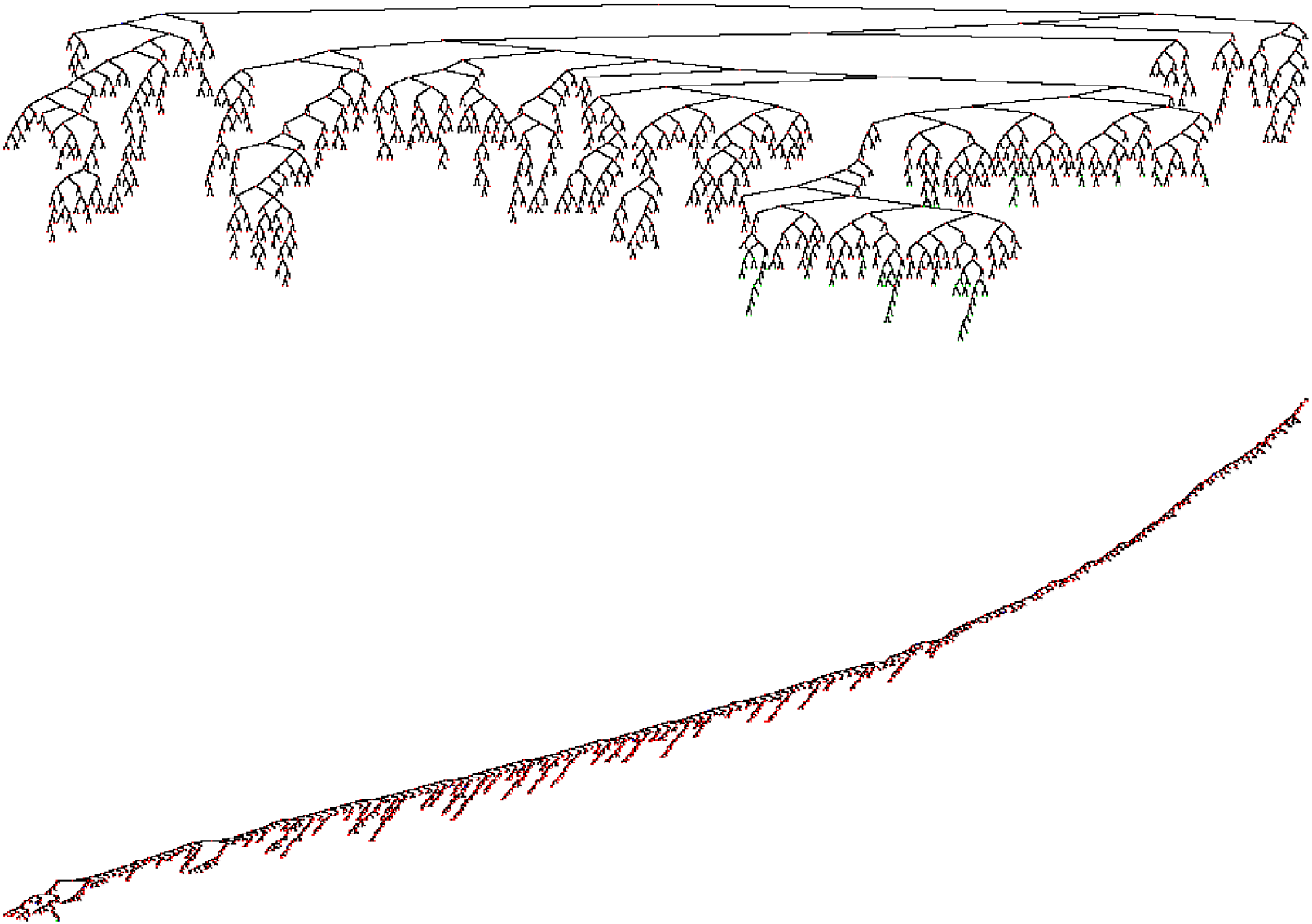
Tree search is easy to parallelize in principle...

- Most straightforwardly, we can parallelize the while loop.
- Naively, this means processing multiple nodes in parallel on line 4.
- Applying the successor function turns one task into two!
- This seems to be what is called “embarassingly parallel”...
- ...but sadly, it’s closer to embarassingly difficult to parallelize!
- We’re aiming at a moving target...and with conflicting goals.

Nice Trees



Ugly Trees



Definitions [Ralphs et al., 2016]

- A *sequential algorithm* is a procedure for solving a given (optimization) problem on a single computing core.
- A *parallel algorithm* is a scheme for performing an equivalent set of computations but using multiple computing cores.
- A parallel algorithm's performance is inherently affected by that of the *underlying sequential algorithm*.
- A *parallel system* is a combination of the
  - Hardware
  - Software
  - OS
  - Toolchain
  - Communication Infrastructure
- We can only measure performance of a parallel system.
- It may be difficult to tell what components are affecting performance.

Outline

What are the Goals?

Sequential Performance

Time (memory) required for a sequential algorithm to perform a fixed computation (as a function of input size).

Scalability (Classical)

Time required for a parallel system to perform a fixed computation as a function of system resources (usually number of cores).

Scalability (Alternative)

Amount of computation that can be done in fixed time as a function of system resources.

Overall Performance

The time required to perform a fixed computation on a parallel system with fixed resources.

Outline

Measures of Sequential Performance for MILP

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]

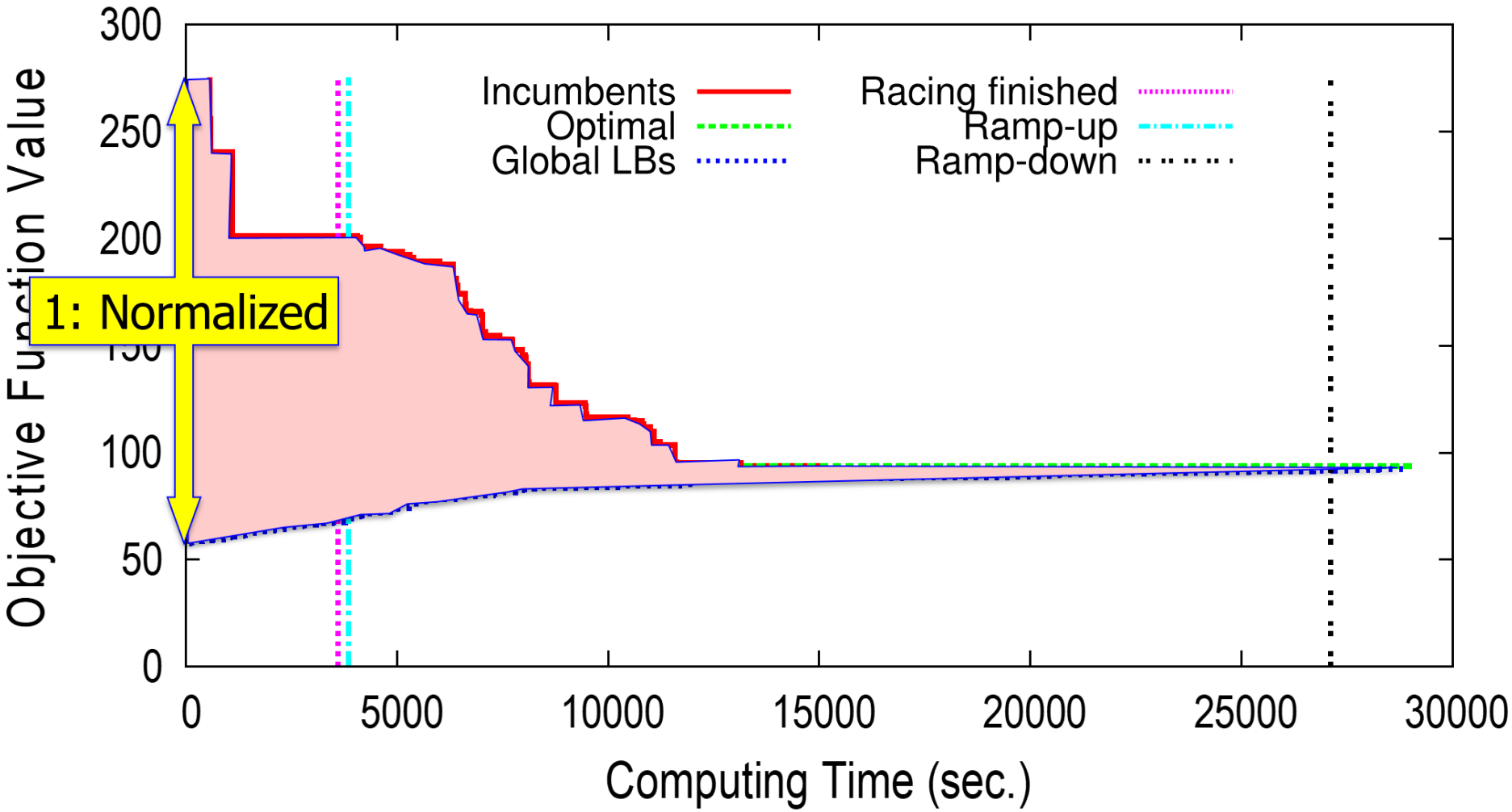


Figure: Example of a PDI plot

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* $\Leftarrow$ Important!
- Possible measures for MILP
  - Gap
  - PDI

Outline

Parallel Performance: Knowledge Sharing

- The goal of parallel computation is to partition a given computation into *equal parts*.
- There are two challenges implicit in achieving this goal.
  - How to partition the computation into *independent* parts.
  - How to ensure the parts are of *equal size*.
- Although partitioning is (ostensibly) easy, the parts are usually not truly independent: *knowledge-sharing* can improve efficiency.
- *Knowledge-sharing* is also necessary in order to "re-balance" when our partition turns not to consist of equal parts.

  - We need *the right data in the right place at the right time*.
  - There is a tradeoff between the *cost incurred in sharing knowledge* versus the *costs incurred by its absence*.
  - The additional cost of navigating this tradeoff is the *parallel overhead* $\Leftarrow$ **This is what we typically try to minimize**

What is “Knowledge” in MILP?

- Descriptions of nodes/subtrees
- Global “knowledge”.
  - Bounds
  - Incumbents
  - Cuts/Conflicts
  - Pseudocosts

Why does it need to be moved?

- It is difficult to know how to partition work equally at the outset, processing units can easily become starved for work.
- Knowledge generated in one part of the tree might be useful for computations in another part of the tree.

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.

Outline

Taking Stock

- Much effort has been poured into developing approaches to parallelizing solvers.
- Many well-developed frameworks taking different approaches exist and are even open source.
- Many computational studies have been done.

Soul-searching Questions

- What have we actually learned?
- What are some best practices and rules of thumb?
- What knowledge can we extract from existing solvers?

Outline

Classical Scalability Analysis

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.

Problems with Classical Analysis

- 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...?
- Scalability numbers alone don't typically give much insight!
- Results are highly dependent on architecture
- Difficult to make comparisons
- Performance variability!
  - Many sources of variability are difficult to control for.
  - Lack of determinism requires extensive testing.

Example: The Knapsack Problem

- We consider the binary knapsack problem:

$$\max\{\sum_{i=1}^{m} p_i x_i : \sum_{i=1}^{m} s_i x_i \leq c, x_i \in \{0,1\}, i = 1,2,\ldots,m\}, \qquad (1)$$

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

P	Node	Ramp-up	Idle	Ramp-down	Wallclock	Eff
4	193057493	0.28%	0.02%	0.01%	586.90	1.00
8	192831731	0.58%	0.08%	0.09%	245.42	1.20
16	192255612	1.20%	0.26%	0.37%	113.43	1.29
32	191967386	2.34%	0.71%	1.47%	56.39	1.30
64	190343944	4.37%	2.27%	5.49%	30.44	1.21

- 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...

Barriers to Scalability: Sophisticated 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!

State-of-the-Art Workflow

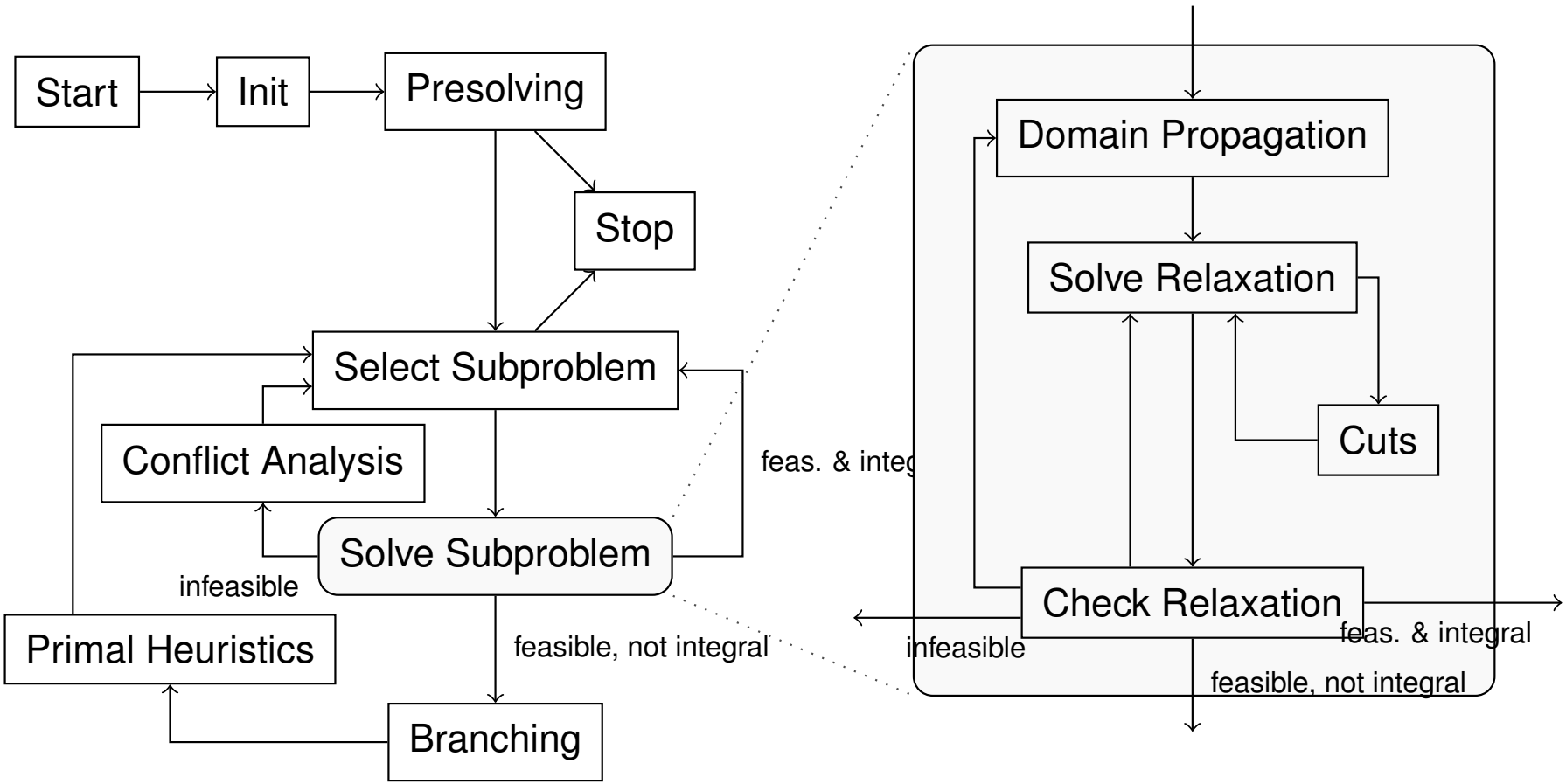


Figure: Flowchart of the main solving loop of a typical MILP solver

Barriers to Scalability: Sophisticated Architectures

- Moore's Law has moved from clock speeds to numbers of cores.
- Current hardware configurations consist of clusters (of clusters) of machines with multiple multi-core chips.
- The result is a memory hierarchy of ever-increasing complexity.

  - Cache memory 1-16x
  - Main memory (local to core) 10-100x
  - Main memory (attached to other cores) 100-700x
  - Co-located distributed memory
  - Remotely located distributed memory >1000x
  - Local disk >3,000,000x

- Such complexity makes it *harder to achieve good parallel performance rather than easier*.
- Tools can help, but to a very limited extent.

Outline

Alternatives to Classical Analysis

- Direct Measures of Overhead
  - Node throughput
  - Ramp-up/Ramp-down time
  - Idle time/Lock time/Wait time
  - Number of nodes
- Analysis based on measures of progress.
  - Gap
  - PDI

Direct Measures of Overhead

- Node throughput [Koch et al., 2012]
  - Easy to measure without instrumentation
  - Not affected by changes in number of nodes
  - Captures the total effect of communication overhead and idle time
  - Hard to interpret with non-constant node processing times (?)
- Ramp-up/Ramp-down time [Xu et al., 2005]
  - May not be that easy to measure.
  - Definitions may differ across solvers
- Idle time/Lock Time/Wait Time
  - Not easy to measure, need instrumentation or proprietary software.
  - Definitions may differ
- Number of nodes
  - Easy to measure
  - Can differ widely due to changes in underlying sequential algorithm

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 resources?
- How much computation can be done with fixed resources but with varying amounts of real time?

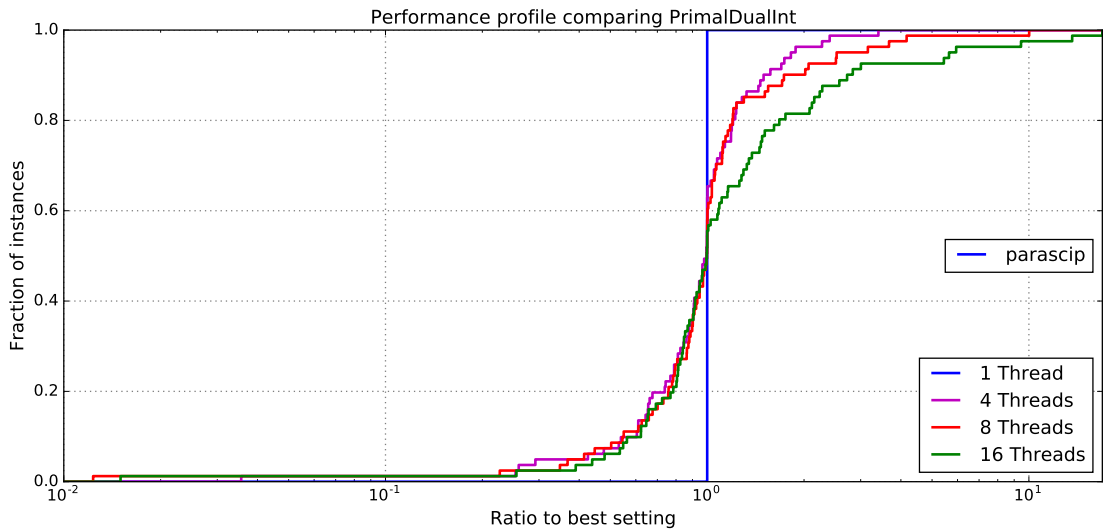
- Allowing partial completion of a fixed computation eliminates many of the problems with finding a test set and comparing solvers.
- Both these alternatives depends on having some reliable "measure of progress," however.
- It is not enough to just measure the "amount of computation"—this is equivalent to measuring utilization and ignoring other overhead.

Gap versus PDI

- Gap
  - Final value is always zero
  - Progress can be “irregular”.
  - Current value doesn’t really indicate now “close” the computation is to finishing.
- PDI
  - Final value can be anything from 0 to the time required for computation (normalized version).
  - Can be normalized to $[0, 1]$., but final value is still variable.
  - Progress can be “irregular”.
  - Still, it seems to be a reasonable proxy for wallclock running time.

Performance Profiles [Dolan and Moré, 2002]

- Simple Idea: Use performance profile to compare performance with different numbers of threads.
- Straight performance profile considers ratios against virtual best.
- Virtual best may not be what is expected.
- An alternative is to consider ratios against single thread.
- In the latter case, we must allow ratios less than one.

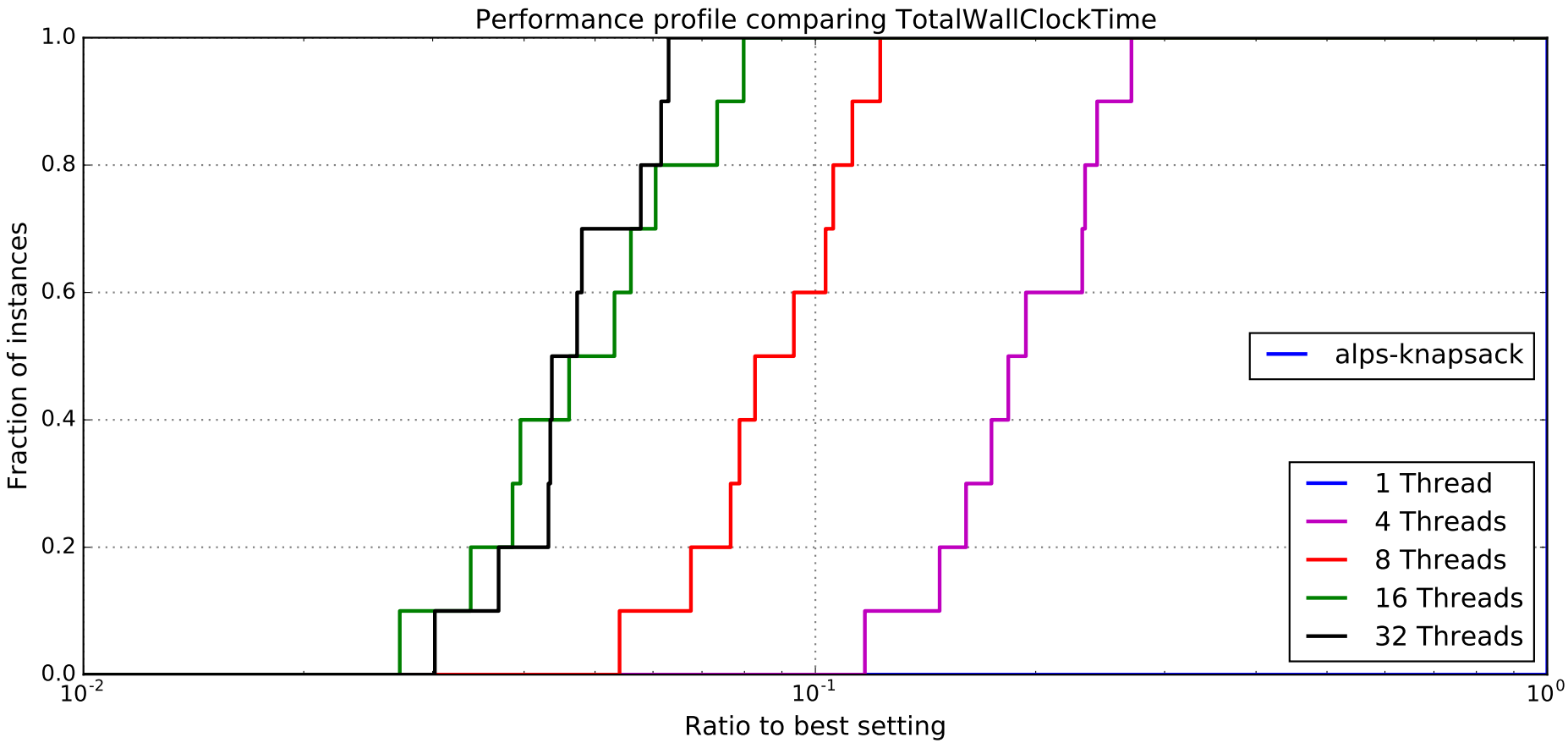


Outline

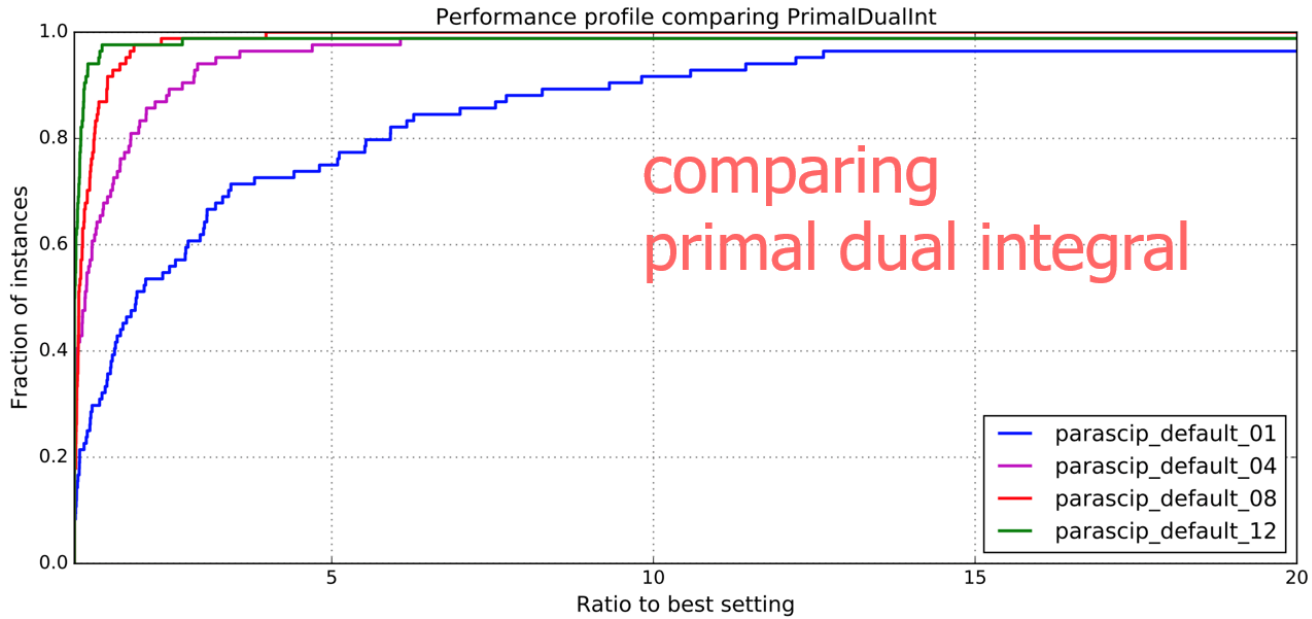
Sample Computational Results

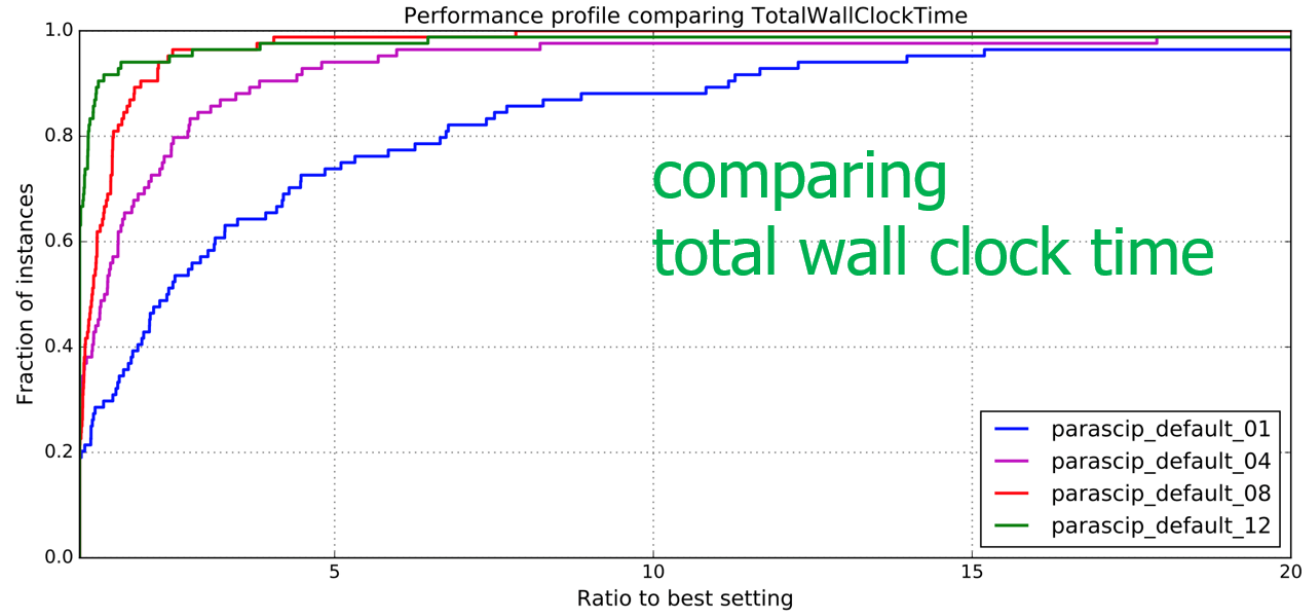
- We have been experimenting with a number of ways of applying the ideas seen so far.
- In the following, we show results with the following solvers.
  - Gurobi
  - ParaSCIP [Shinano et al., 2013]
  - SYMPHONY [Ralphs and Güzelsoy, 2005]
  - ALPS [Xu et al., 2007]

Ideal Scaling: Knapsack Problems

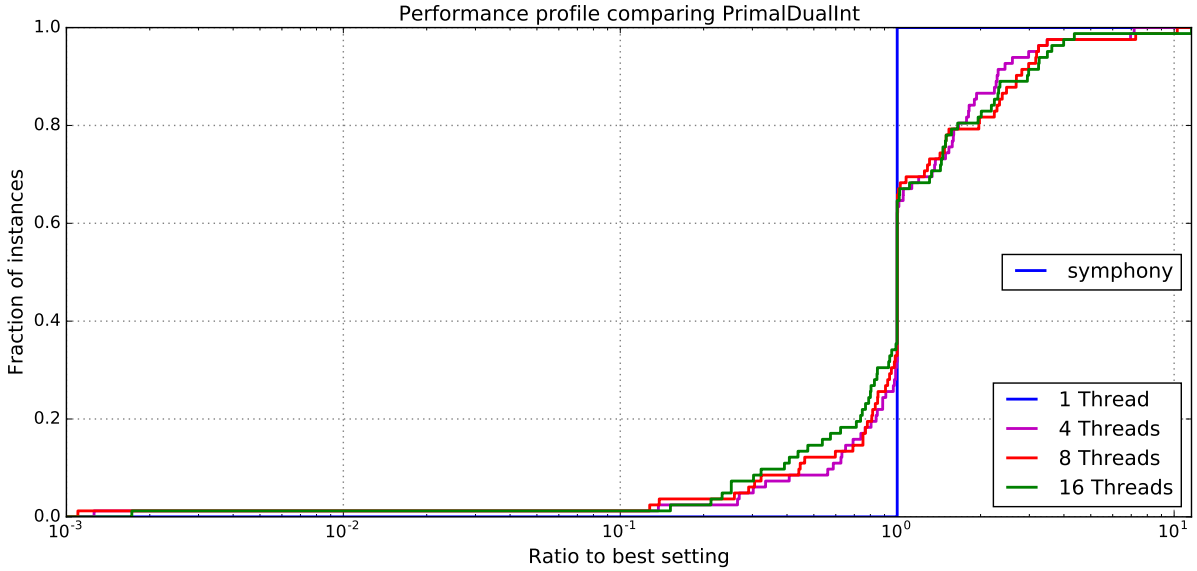


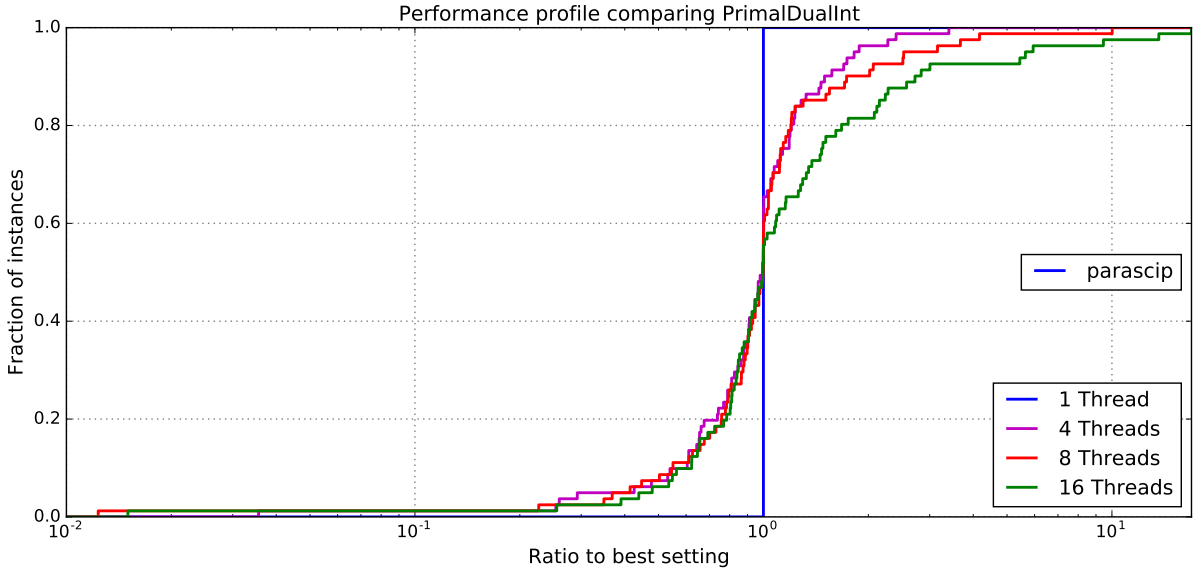
Performance Profile: PDI versus Wallclock





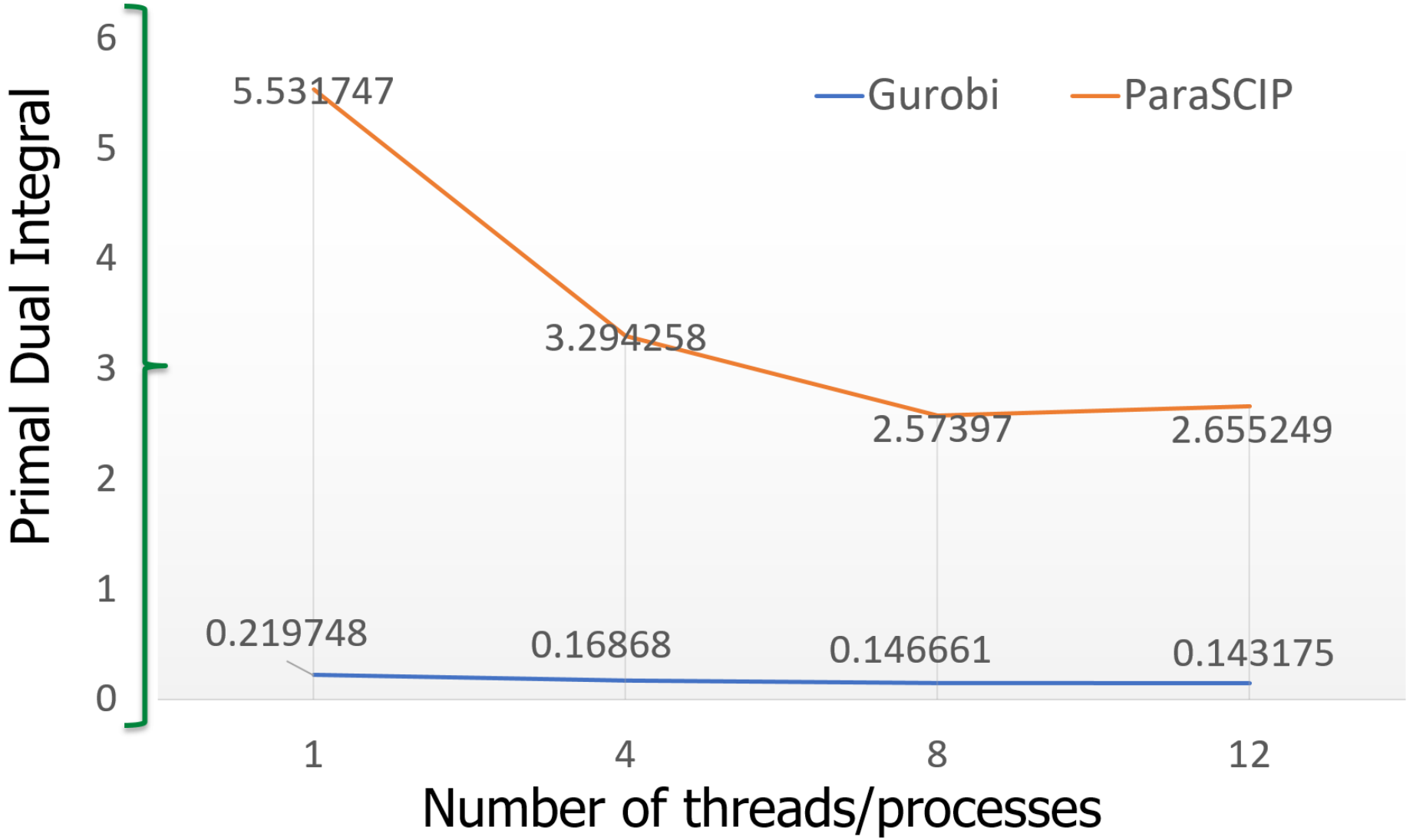
PDI Profile Against Single Thread



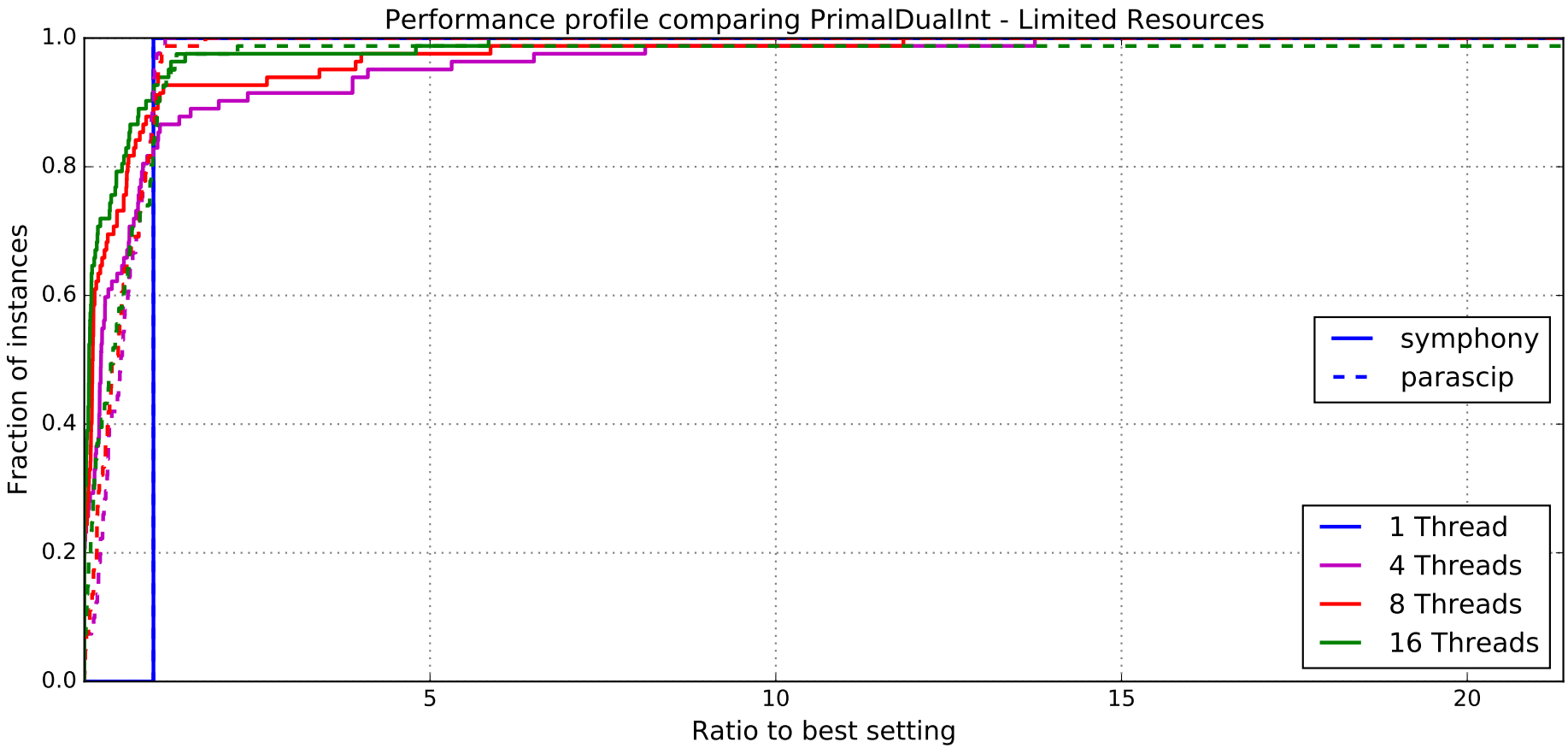


PDI Means

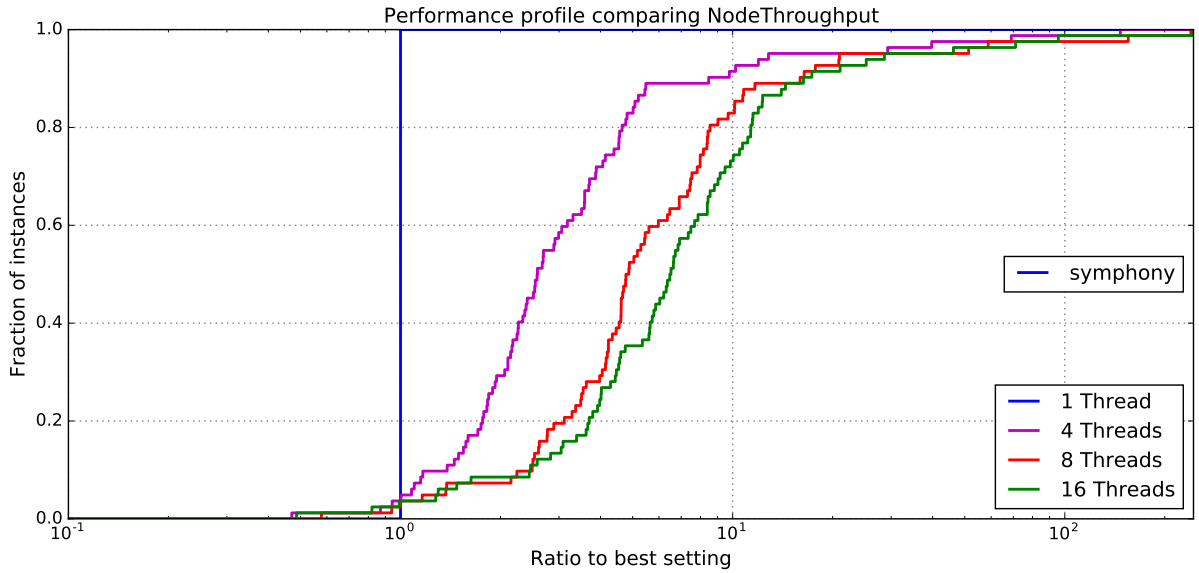
MIPLIB2010 Benchmark

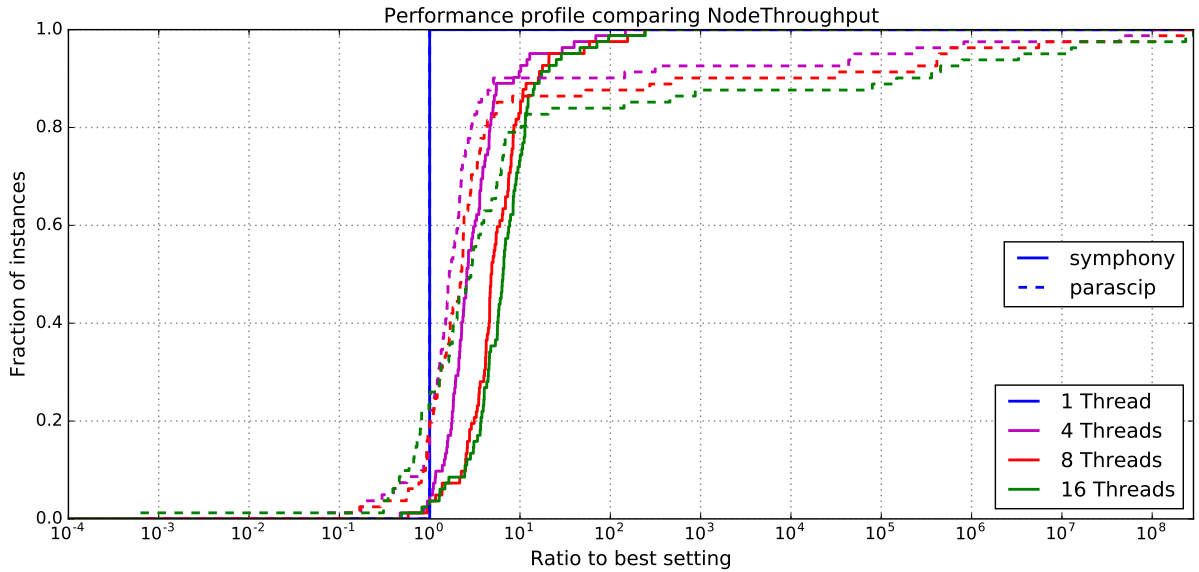


PDI Profile with Equal Core Time

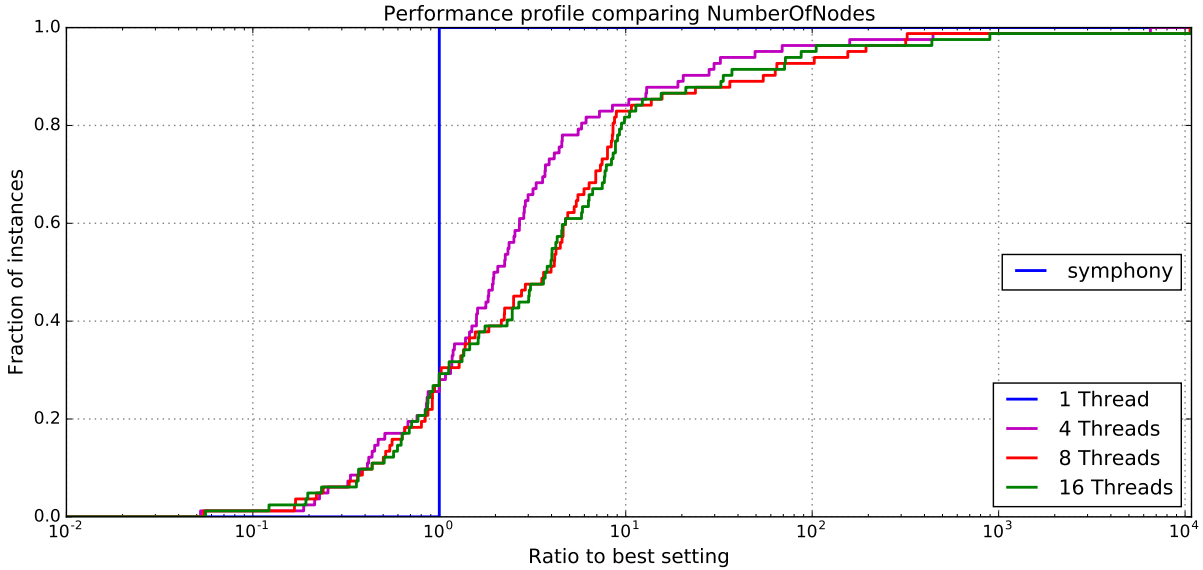


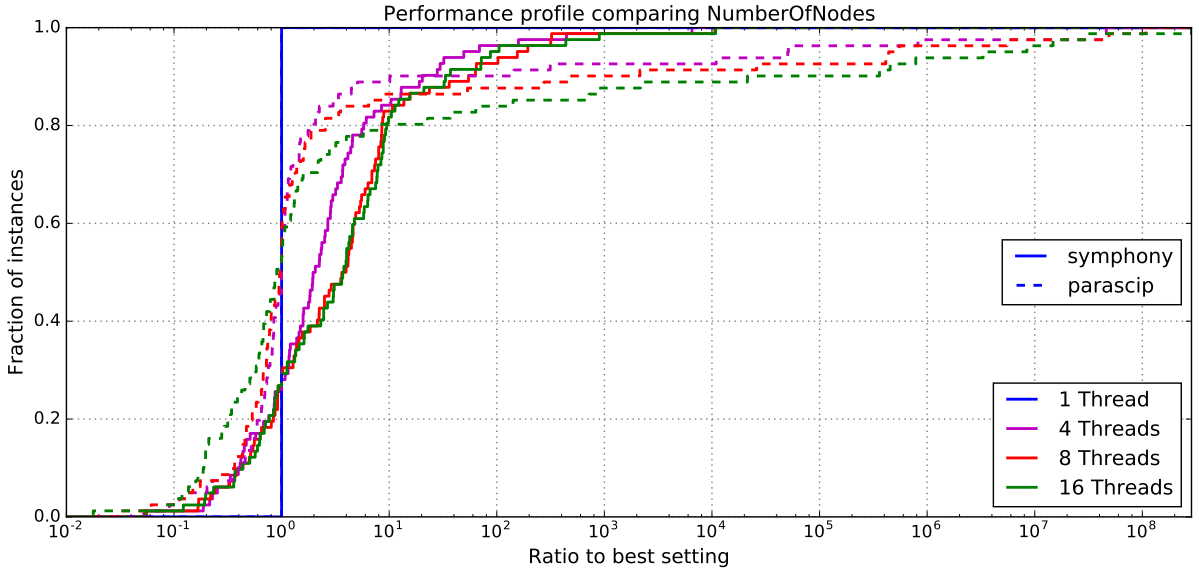
Node Throughput Profiles



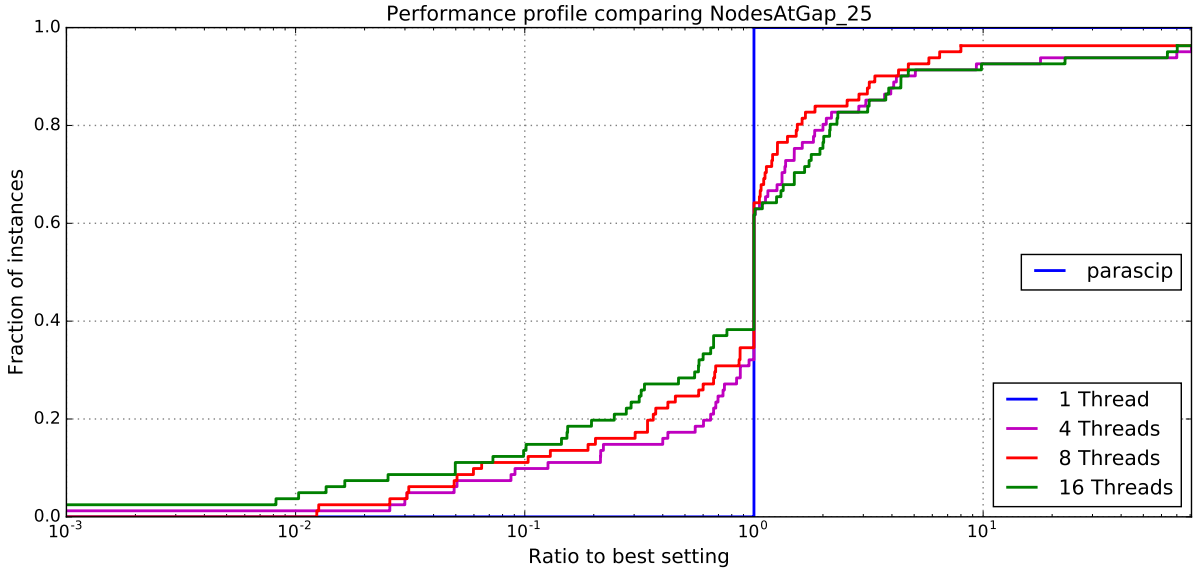


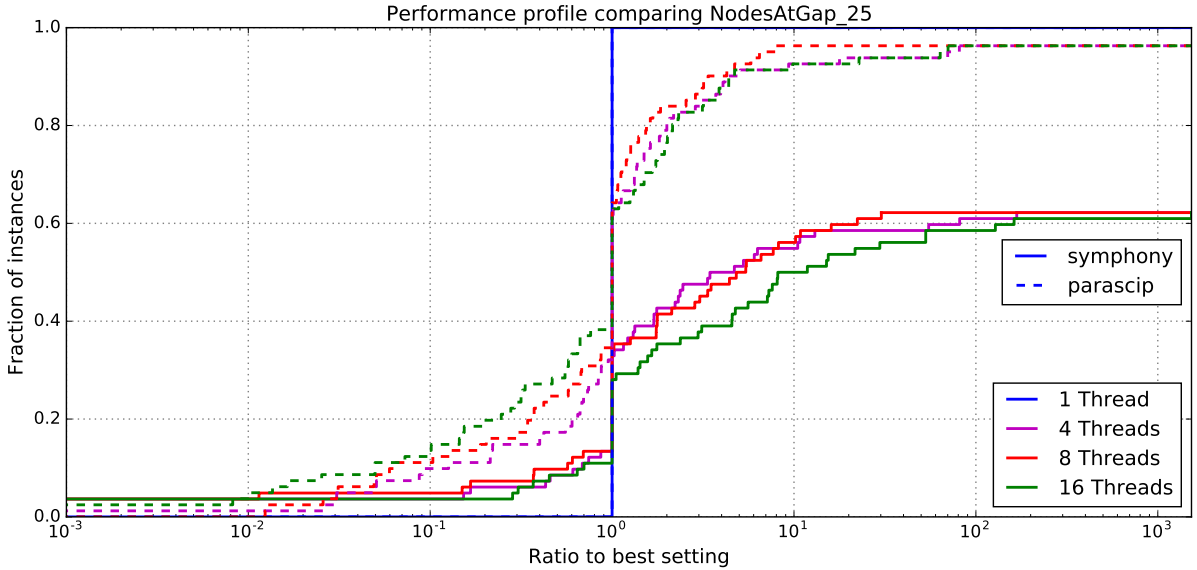
Number of Nodes





Number of Nodes at Gap





Outline

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

Timo Berthold. Measuring the impact of primal heuristics. ZIB-Report 13-17, Zuse Institute Berlin, Takustr. 7, 14195 Berlin, 2013.

Elizabeth Dolan and Jorge Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91:201–213, 2002.

T Koch, TK Ralphs, and Y Shinano. Could we use a million cores to solve an integer program? *Mathematical Methods of Operations Research*, 76:67–93, 2012. doi: 10.1007/s00186-012-0390-9. URL `http://coral.ie.lehigh.edu/~ted/files/papers/Million11.pdf`.

References II

T K Ralphs, Y Shinano, T Berthold, and T Koch. Parallel solvers for mixed integer linear programing. Technical report, COR@L Laboratory Report 16T-014-R1, Lehigh University, 2016. URL http://coral.ie.lehigh.edu/~ted/files/papers/ParallelMILPSurvey16.pdf.

Ted K Ralphs and Menal Güzelsoy. The symphony callable library for mixed-integer linear programming. In *Proceedings of the Ninth INFORMS Computing Society Conference*, pages 61–76, 2005. doi: 10.1007/0-387-23529-9_5. URL http://coral.ie.lehigh.edu/~ted/files/papers/SYMPHONY04.pdf.

Yuji Shinano, Stefan Heinz, Stefan Vigerske, and Michael Winkler. FiberSCIP – a shared memory parallelization of SCIP. ZIB-Report 13-55, Zuse Institute Berlin, 2013.

References III

Yan Xu, Ted K Ralphs, Laszlo Ladányi, and Matthew J Saltzman. Alps: A framework for implementing parallel search algorithms. In *The Proceedings of the Ninth INFORMS Computing Society Conference*, pages 319–334, 2005. doi: 10.1007/0-387-23529-9_21. URL http://coral.ie.lehigh.edu/~ted/files/papers/ALPS04.pdf.

Yan Xu, Ted K Ralphs, Laszlo Ladányi, and Matthew J Saltzman. Computational experience with a framework for parallel integer programming. Technical report, COR@L Laboratory Report , Lehigh University, 2007. URL http://coral.ie.lehigh.edu/~ted/files/papers/CHiPPS.pdf.