

Assessing Performance of Parallel MILP solvers

Ted Ralphs¹

Stephen J. Maher², Yuji Shinano³

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

UG Workshop, ZIB, Berlin, 14 January 2018



ISE

Industrial and
Systems Engineering

COR@L

COMPUTATIONAL OPTIMIZATION
RESEARCH AT LEHIGH



- 1 Introduction
- 2 Measures of Performance
 - Efficiency
 - Scalability
- 3 Performance Analysis
 - Challenges
 - Alternatives to Classical Analysis
 - Sample Computational Results
- 4 Conclusions

- 1 Introduction
- 2 Measures of Performance
 - Efficiency
 - Scalability
- 3 Performance Analysis
 - Challenges
 - Alternatives to Classical Analysis
 - Sample Computational Results
- 4 Conclusions

Assessing “Performance”

- Fundamental questions we would like to answer.
 - **How to assess algorithm “performance”?**
 - **How to compare solvers with disparate capabilities?**
 - **How to isolate determinants of “performance”?**
- 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 performance?
- Can we answer these questions by observation without (much) instrumentation?

What Is “Performance”?

- We carefully distinguish between two aspects of what we generally (and loosely) refer to as the “**performance**” 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 allocated to a computation (e.g., processors, time, memory).

Efficiency

- The effectiveness with which an implementation utilizes the resources on a system to perform a *benchmark computation*.
- Measures requirement for one resource if others are held constant.

Scalability

- Measures how efficiency changes as resource allocation changes.
- Measures the *tradeoff* between resources.

Roadmap for This Talk

- Focus of this talk will be primarily on assessing scalability, but measurement of efficiency is inherently involved.
- Due to time constraints, I'll skip many definitions.
- There is a full paper that goes into much more detail.
- It is expected that there will be lots of discussion and controversy during this talk, so please jump in!

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

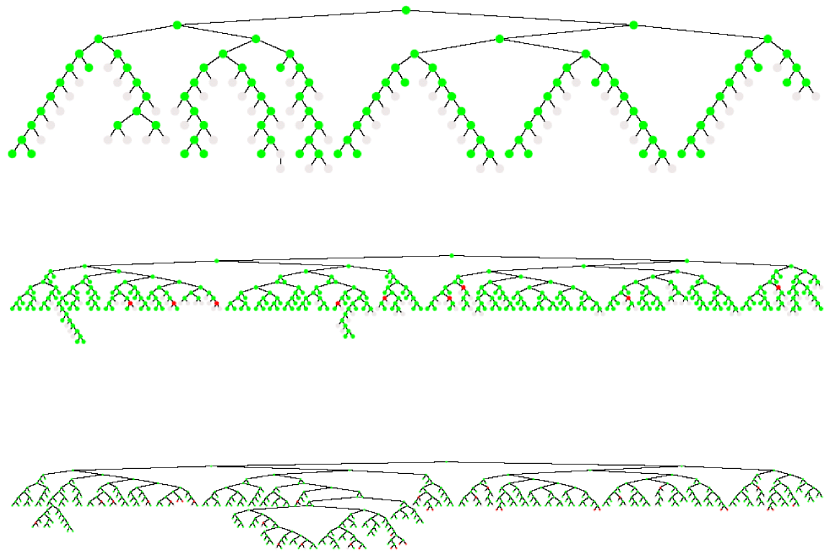
```
1 Add root node  $r$  to a priority queue  $Q$ .
2 while  $Q$  is not empty do
3   | Choose a node  $i$  from  $Q$ .
4   | Process the node  $i$ .
5   | Apply pruning rules (can  $i$  or a successor be a goal node?)
6   | if Node  $i$  can be pruned then
7   |   | Prune (discard) node  $i$  (save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

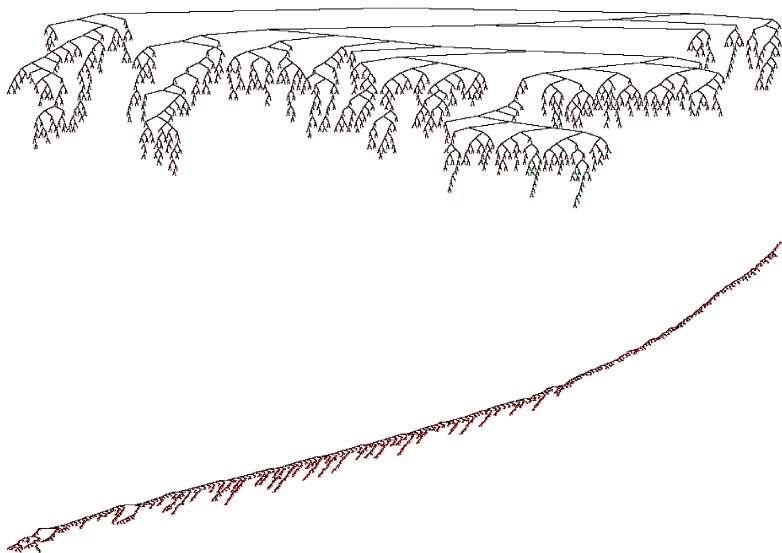
Tree search is easy to parallelize in principle...

- Most straightforwardly, we can parallelize the while loop.
- Naively, this means processing multiple nodes in parallel.
- 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



Ugly Trees



Barriers to Scalability: Sophistication of Solvers

- A vast amount of effort has gone into improving the performance of sequential solvers.
- It's been estimated that overall solver performance has improved by a factor of approximately 2 trillion in past decades.
- Major advances in solver technology have mostly made good parallel scalability *more difficult* to achieve.
 - **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!

Example: The Knapsack Problem

- We consider the binary knapsack problem:

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

- 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...

State-of-the-Art Workflow

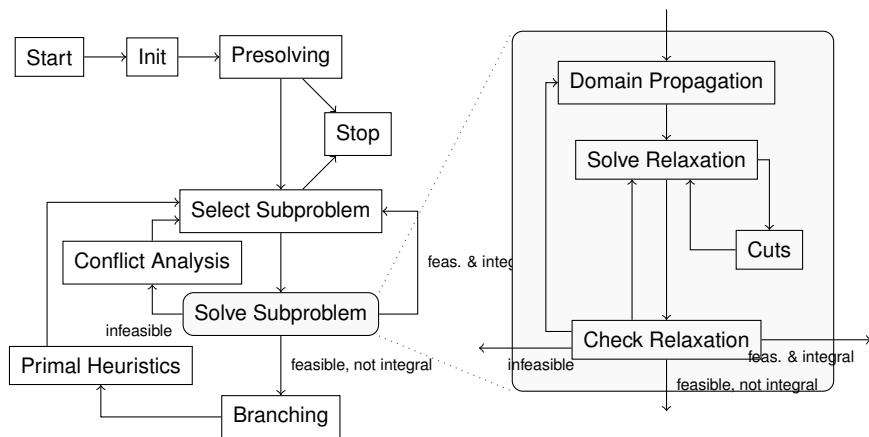


Figure: Flowchart of the main solving loop of a typical MILP solver

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.

Definitions [Ralphs et al., 2018]

- A *sequential algorithm* is a procedure for solving a given (optimization) problem on a single processor (computing core).
- A *parallel control mechanism* is a scheme for performing an equivalent set of computations but using multiple processors.
- A parallel mechanism's performance is inherently affected by that of the *underlying sequential algorithm*.
- A (*parallel*) *computing system* is a combination of the
 - hardware,
 - software,
 - OS,
 - toolchain, and
 - communication infrastructure.
- Performance can only be measured wrt a given computing system.
- It may be difficult to separate the effects of different components.

Outline

1 Introduction

2 Measures of Performance

- Efficiency
- Scalability

3 Performance Analysis

- Challenges
- Alternatives to Classical Analysis
- Sample Computational Results

4 Conclusions

- 1 Introduction
- 2 Measures of Performance
 - Efficiency
 - Scalability
- 3 Performance Analysis
 - Challenges
 - Alternatives to Classical Analysis
 - Sample Computational Results
- 4 Conclusions

Some Definitions

Definition

A *measure of efficiency* for a given benchmark computation is the amount of one chosen resource that is required to perform that computation, once the amounts of all other resources are fixed.

Definition

A *measure of work* is a direct measure of the amount of work done in performing a benchmark computation, generally expressed in terms of the number of atomic units of computational effort executed in order to perform the computation.

Definition

A *measure of progress* is an estimate of or proxy for the fraction of a benchmark computation completed by an implementation, given a fixed bundle of resources.

Measures of Efficiency

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]

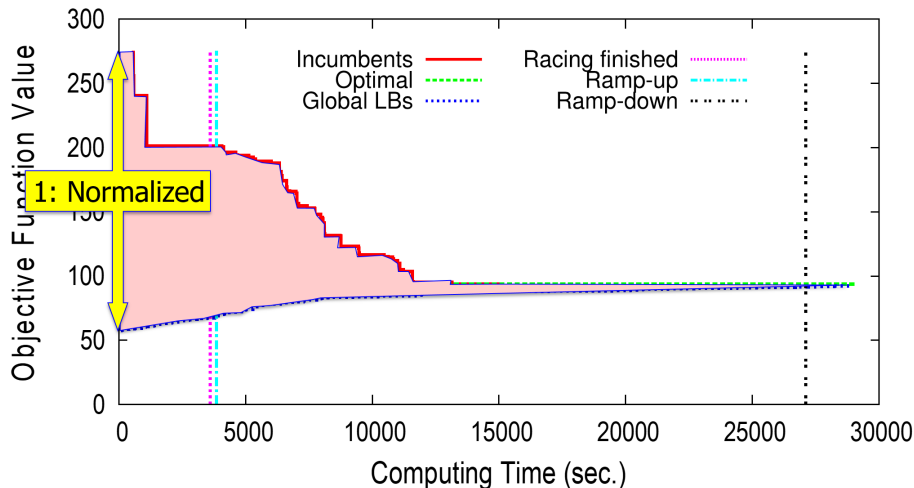
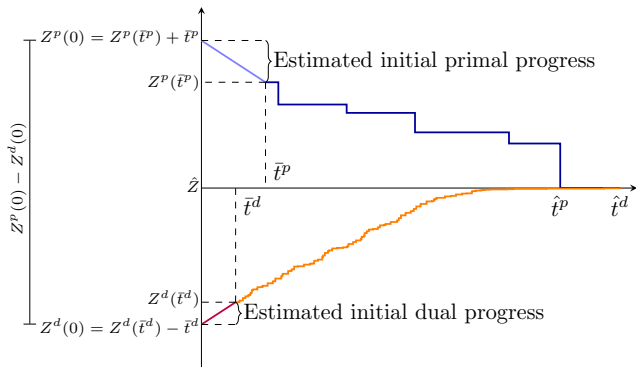


Figure: Example of a PDI plot

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.

The Extended Primal Dual Integral



Outline

- 1 Introduction
- 2 Measures of Performance
 - Efficiency
 - Scalability
- 3 Performance Analysis
 - Challenges
 - Alternatives to Classical Analysis
 - Sample Computational Results
- 4 Conclusions

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* **← 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.

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 Performance
 - Efficiency
 - Scalability
- 3 Performance Analysis**
 - Challenges**
 - Alternatives to Classical Analysis**
 - Sample Computational Results**
- 4 Conclusions

Outline

- 1 Introduction
- 2 Measures of Performance
 - Efficiency
 - Scalability
- 3 Performance Analysis
 - Challenges
 - Alternatives to Classical Analysis
 - Sample Computational Results
- 4 Conclusions

Challenges in Analyzing Efficiency

- Branch-and-Bound algorithms generally consist of collections of semi-independent algorithmic components.
 - Each component controls a different aspect of the overall algorithm and can be implemented in a range of different ways.
 - The components are bound together by a control strategy, which, in turn, has its own implementation.
 - It is difficult to isolate the effects of individual components, though thorough analysis requires this.
 - Comparing implementation of the same component within different solvers is difficult due to confounding effects of other components.
- Efficiency will generally vary widely across instances.
 - Is it better to focus on a narrow class of instances or strive for a “representative” test set?
 - Test sets inevitably favor certain types of algorithms over others.
- Performance variability can be HUGE.
- What is the right metric for comparison?
- How should we summarize results and draw 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!

Outline

- 1 Introduction
- 2 Measures of Performance
 - Efficiency
 - Scalability
- 3 Performance Analysis
 - Challenges
 - **Alternatives to Classical Analysis**
 - Sample Computational Results
- 4 Conclusions

Alternatives to Classical Scalability Analysis

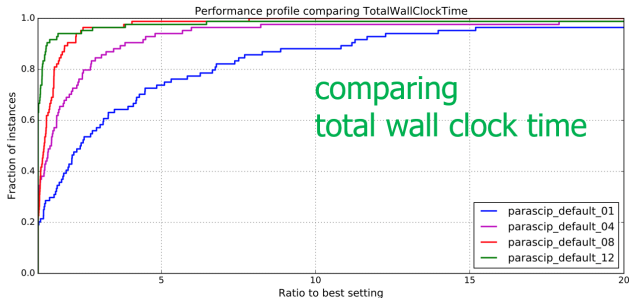
- Direct measures of overhead.
 - Node throughput
 - Ramp-up/Ramp-down time
 - Idle time/Lock time/Wait time
 - Number of nodes
- Performance, scalability, and cumulative profiles.
- Analysis based on measures of progress.
 - Gap
 - Extended PDI

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

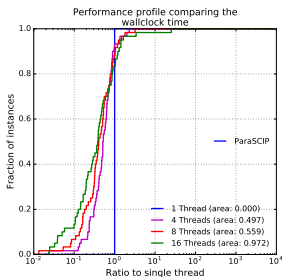
Performance Profiles for Scalability Analysis

- 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.

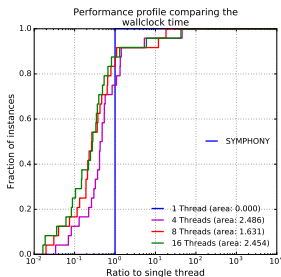


Scalability Profiles

- 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.



(a) ParaSCIP

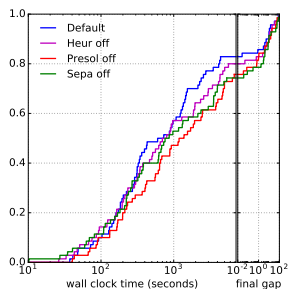


(b) SYMPHONY

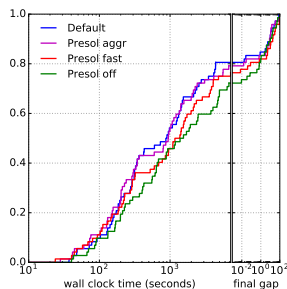
Figure: Scalability profile of wallclock running time.

Cumulative Profiles

- Cumulative profiles compare progress on solving an entire test set on a “virtual” parallel computing system.
- We simulate all instances running simultaneously and plot the number that have solved at each point in time.
- Instances that didn't solve in the time limit are represented by their final gap.



(a) Emphasis - Off



(b) Presolve emphasis

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 performance of algorithms *even if the computation doesn't complete* ← Important!
- Possible measures for MILP
 - Gap
 - *Extended PDI*

Gap versus Extended PDI

- Gap

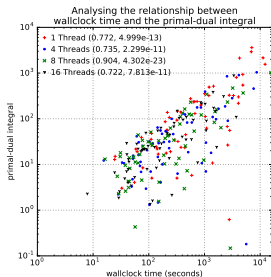
- Final value is always zero
- Progress can be “irregular”.
- Current value doesn't really indicate how “close” the computation is to finishing.

- Extended PDI

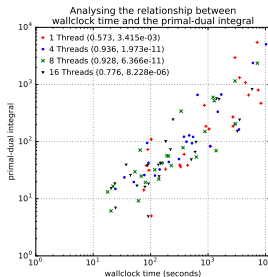
- Final value can be anything from 0 to the time required for computation (normalized version).
- Can be normalized to $[0, 1]$, but the final value is still variable.
- Progress can be “irregular”.
- Still, it seems to be a reasonable proxy for wallclock running time.

Extended PDI versus Wallclock

- 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.



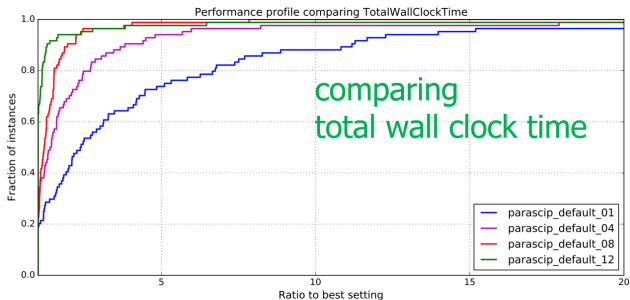
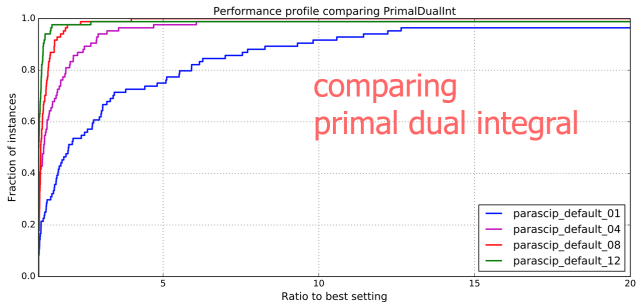
(a) ParaSCIP



(b) SYMPHONY

Figure: The relationship between the wall clock time and the extended PDI.

Performance Profiles of Extended PDI and Wallclock



- 1 Introduction
- 2 Measures of Performance
 - Efficiency
 - 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]

Ideal Scaling: Knapsack Problems

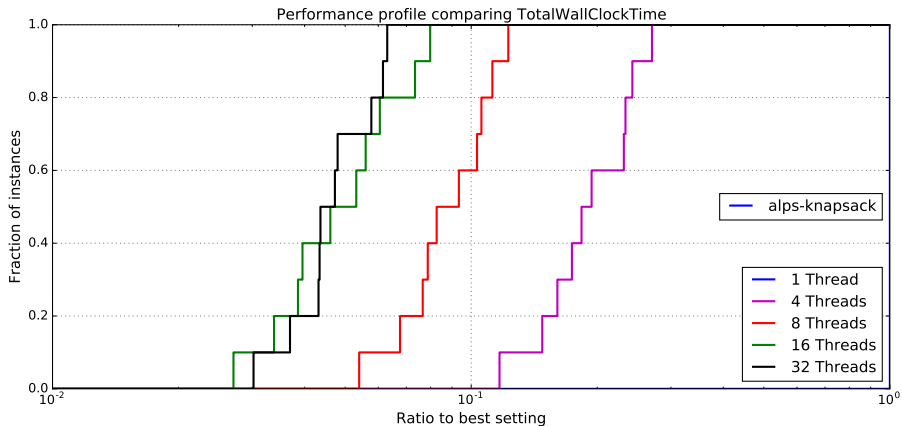
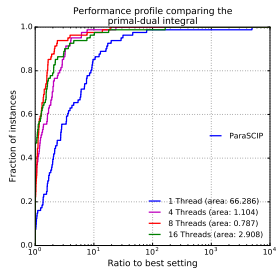
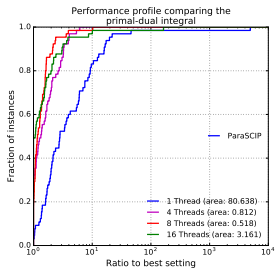


Figure: Scalability profile of Wallclock time for Knapsack Problems on ALPS.

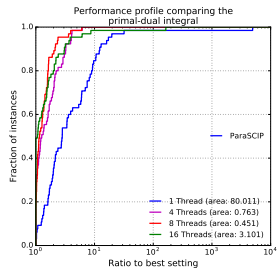
Performance Profile Using Extended PDI



(a) 18000 seconds limit



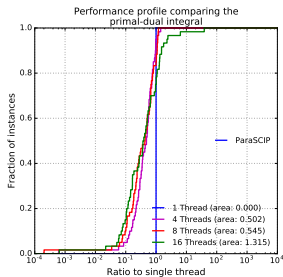
(b) 9000 seconds limit



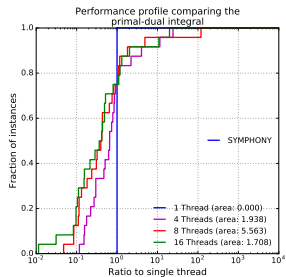
(c) 4500 seconds limit

Figure: Performance profile of PDI for ParaSCIP on MIPLIB2010.

Scalability Profile Using Extended PDI



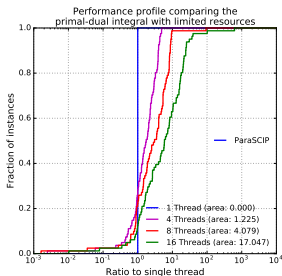
(a) ParaSCIP



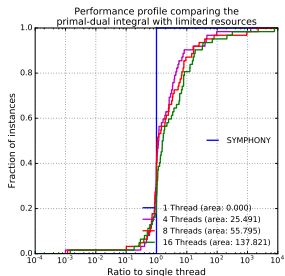
(b) SYMPHONY

Figure: Scalability profile of the extended PDI

Scalability Profile with Fixed Compute Time



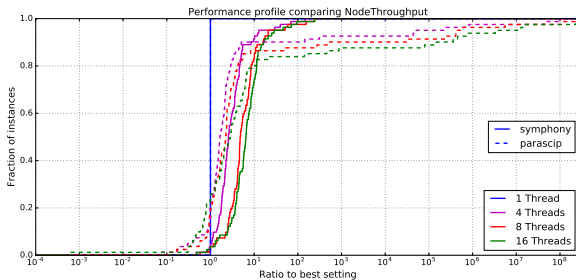
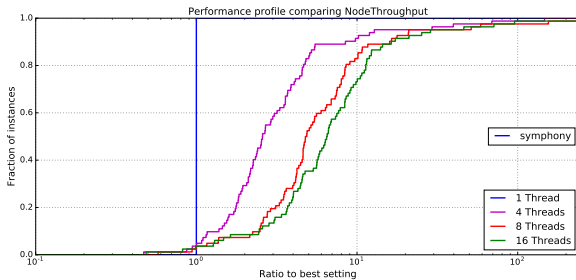
(a) ParaSCIP



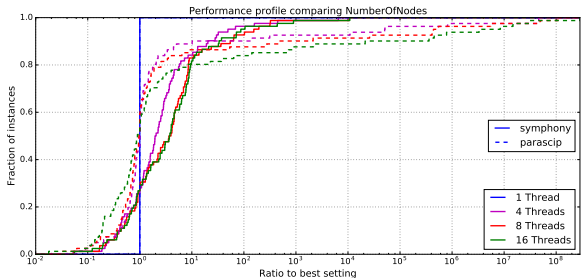
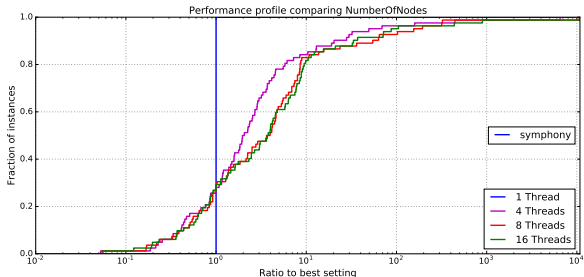
(b) SYMPHONY

Figure: The scalability profile of PDI with fixed compute time.

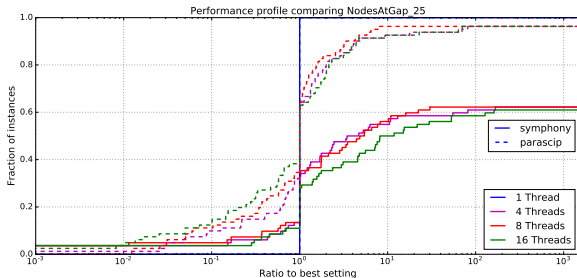
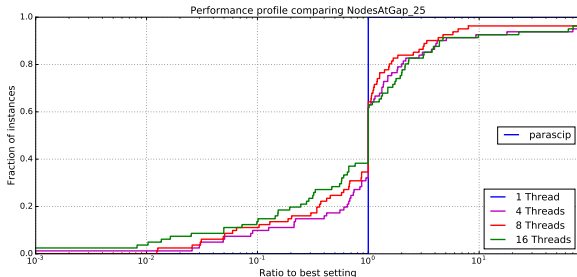
Node Throughput Scalability Profile



Number of Nodes Scalability Profile



Number of Nodes at Gap Scalability Profile



Outline

- 1 Introduction
- 2 Measures of Performance
 - Efficiency
 - 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. [2018]

References I

- T. Berthold. Measuring the impact of primal heuristics. ZIB-Report 13-17, Zuse Institute Berlin, Takustr. 7, 14195 Berlin, 2013.
- T. Koch, T.K. 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>.
- T.K. Ralphs and M. Güzelsoy. The SYMPHONY Callable Library for Mixed Integer 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>.

References II

- T.K. Ralphs, Y. Shinano, T. Berthold, and T. Koch. Parallel Solvers for Mixed Integer Linear Programming. In Y Hamadi and L Sais, editors, *Handbook of Parallel Constraint Reasoning*, chapter 8, pages 283–336. Springer Berlin / Heidelberg, 2018. URL <http://coral.ie.lehigh.edu/~ted/files/papers/ParallelMILPSurvey16.pdf>.
- Y. Shinano, S. Heinz, S. Vigerske, and M. Winkler. FiberSCIP – a shared memory parallelization of SCIP. ZIB-Report 13-55, Zuse Institute Berlin, 2013.
- Y. Xu, T.K. Ralphs, L. Ladányi, and M.J. Saltzman. ALPS: A Framework for Implementing Parallel Tree Search Algorithms. In *The Proceedings of the Ninth INFORMS Computing Society Conference*, volume 29, pages 319–334, 2005. doi: 10.1007/0-387-23529-9_21. URL <http://coral.ie.lehigh.edu/~ted/files/papers/ALPS04.pdf>.

References III

Y. Xu, T.K. Ralphs, L. Ladányi, and M.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>.