Parallel Solution of Mixed Integer Linear Programs

Ted Ralphs[1]

Thorsten Koch[2], Stephen J. Maher[3], Yuji Shinano[2], Yan Xu[4]

[1]COR@L Lab, Lehigh University, Bethlehem, PA USA [2]Zuse Institute Berlin, Berlin, Germany [3]Lancaster University, Lancaster, UK [4]SAS Institute

Workshop on Optimization, Wuyishan, Fujian, China, 14 August 2018

COIN|OR
COMPUTATIONAL INFRASTRUCTURE FOR OPERATIONS RESEARCH

ISE Industrial and Systems Engineering

COR@L
COMPUTATIONAL OPTIMIZATION RESEARCH AT LEHIGH

Outline

1. Introduction
   - Tree Search Algorithms
   - Historical Perspective
2. Parallel Algorithms
   - Definitions and Background
   - State of the Art
   - Challenges
3. Assessing Effectiveness
   - Sequential Performance
   - Parallel Scalability
4. Computational Results
   - Sequential Performance
   - Parallel Performance
   - Parallel Scalability
5. Conclusions

This Talk

- This overview draws on material from several published and one unpublished paper, as well as one dissertation.
  - Xu [2007] (Dissertation on Parallel Tree Search)
  - Xu et al. [2009] (CHiPPS Framework)
  - Koch et al. [2012] (Forward-looking perspective
  - Ralphs et al. [2016] (⇐ Overview)
  - Maher et al. [2018] (Performance Assessment)
- Many details will be left out, but will be found in the above references.
- We focus on parallel MILP, but the principles apply much more broadly.

Setting

- We focus on the case of the *mixed integer linear optimization problem* (MILP), but many of the concepts are more general.

$$z_{IP} = \min_{x \in \mathcal{S}} c^\top x, \tag{MILP}$$

where, $c \in \mathbb{R}^n$, $\mathcal{S} = \{x \in \mathbb{Z}^r \times \mathbb{R}^{n-r} \mid Ax \leq b\}$ with $A \in \mathbb{Q}^{m \times n}$, $b \in \mathbb{Q}^m$.

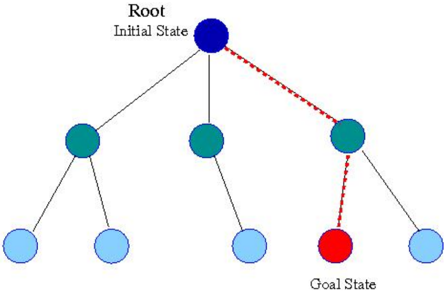
- For most of the talk, we consider the case $r = n$ and $\mathcal{P}$ bounded for simplicity.

Outline

1. Introduction
   - Tree Search Algorithms
   - Historical Perspective
2. Parallel Algorithms
   - Definitions and Background
   - State of the Art
   - Challenges
3. Assessing Effectiveness
   - Sequential Performance
   - Parallel Scalability
4. Computational Results
   - Sequential Performance
   - Parallel Performance
   - Parallel Scalability
5. Conclusions

Tree Search Algorithms

- *Tree search* algorithms systematically search the nodes of a dynamically constructed acyclic graph for certain *goal nodes*.



- Tree search algorithms are used in many areas such as
  - Constraint satisfaction,
  - Game search,
  - Constraint Programming, and
  - Mathematical programming.

Tree Search

- Tree search is not a single algorithm but an algorithmic framework.
- A generic tree search algorithm consists of the following elements:

Elements of Tree Search

- Processing method: Is this a goal node?
- Fathoming rule: Can node can be fathomed?
- Branching method: What are the successors of this node?
- Search strategy: What should we work on next?

- Beginning with a root node, the algorithm consists of choosing a candidate node, processing it, and either fathoming or branching.
- During the course of the search, various information (*knowledge*) is generated and can be used to guide the search.

Generic Algorithm

Algorithm 1: A Generic Tree Search Algorithm

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

Branch and Bound/Cut/Price

Algorithm 2: A Generic Branch-and-Cut Algorithm

```
Add root optimization problem $r$ to a priority queue $Q$. Set global upper bound $U \leftarrow \infty$
  and global lower bound $L \leftarrow -\infty$
while $L < U$ do
    Remove the highest priority subproblem $i$ from $Q$.
    Bound the subproblem $i$ to obtain (updated) final upper bound $U(i)$ and (updated)
      final lower bound $L(i)$.
    Set $U \leftarrow \min\{U(i), U\}$.
    if $L(i) < U$ then
        Branch to create child subproblems $i_1, \ldots, i_k$ of subproblem $i$ with
          - upper bounds $U(i_1), \ldots U(i_k)$ (initialized to $\infty$ by default); and
          - initial lower bounds $L(i_1), \ldots, L(i_k)$ (initialized to $L(i)$ by default).
        by partitioning the feasible region of subproblem $i$.
        Add $i_1, \ldots, i_k$ to $Q$.
        Set $L \leftarrow \min_{i \in Q} L(i)$.
```

Components

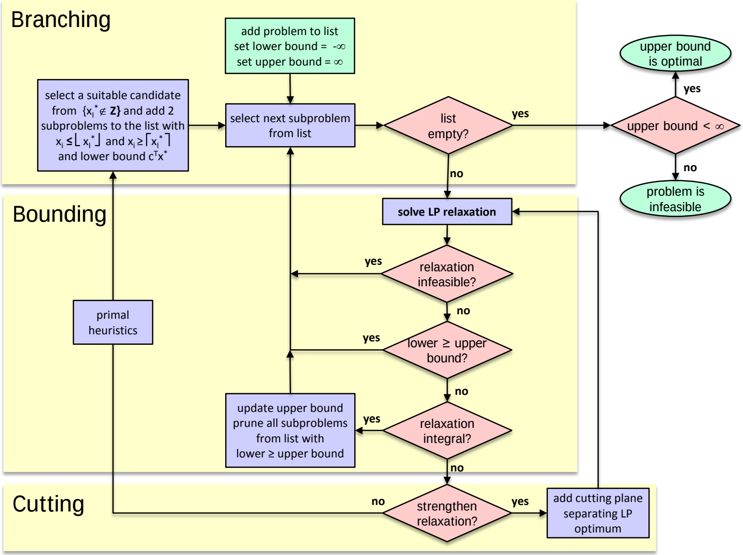
- *Bounding* is by solution of (iteratively strengthened) LP relaxations.
- *Branching* is done on *valid disjunctions*.

Definition

Let $\{X_i\}_{i=1}^k$ be a collection of subsets of $\mathbb{R}^n$. Then if $\bigcup_{1 \le i \le k} X_i \supseteq \mathcal{S}$, the disjunction associated with $\{X_i\}_{i=1}^k$ is said to be *valid* for an MILP with feasible set $\mathcal{S}$.

- *Search strategy* is aimed at carefully balancing
  - Improvement of upper and lower bound,
  - Efficiency of node processing (diving), and
  - Avoidance of *redundant work*.
- All of this is immensely more complex in the parallel case.

Current State-of-the-Art: Solver Workflow



Branching

add problem to list
set lower bound = $-\infty$
set upper bound = ∞

select a suitable candidate from $\{x_l^* \notin \mathbf{Z}\}$ and add 2 subproblems to the list with $x_i \leq \lfloor x_l^* \rfloor$ and $x_i \geq \lceil x_l^* \rceil$ and lower bound $c^T x^*$

select next subproblem from list

list empty?

yes

upper bound $< \infty$

yes

upper bound is optimal

no

problem is infeasible

no

Bounding

solve LP relaxation

relaxation infeasible?

yes

no

lower ≥ upper bound?

yes

no

relaxation integral?

yes

update upper bound
prune all subproblems from list with
lower ≥ upper bound

no

primal heuristics

Cutting

strengthen relaxation?

no

yes

add cutting plane separating LP optimum

Current State-of-the-Art: Algorithm Control

- A state-of-the-art solver is a **collection of algorithms and heuristics** for solving a variety of subsidiary optimization problems.
  - Whether to branch or continue iteratively improving the relaxation.
  - Which logical disjunction to branch on.
  - Which node to work on next.
  - What relaxation to use, how to strengthen it, and how to solve it.
  - What valid inequalities to generate.
  - What primal heuristics to try.
  - Etc.
- These are bound together by a sophisticated overall **control mechanism**.
- The individual components are mostly well-studied in the literature and relatively easy to assess in isolation.
- The behavior of the overall algorithm is poorly understood and difficult to study scientifically.

It’s All About Tradeoffs

- Algorithm control is about carefully managing various tradeoffs.
    - Time spent selecting disjunctions versus more enumeration.
    - Time spent cutting versus more enumeration.
    - Time spent branching versus time spent cutting.
    - Preprocessing and root node versus remainder of computation.
    - Emphasis on primal bound versus dual bound.
    - Primal heuristics versus cutting and branching.
- The way this is done is a big part of the “special sacue” of a solver and is not really documented.
- This gets much harder to do in the case of a parallel algorithm.

Auto-tuning and Algorithm Optimization

- In general, for a given instance, the solver tries to determine how to optimally balance multiple objectives.
    - Minimize solution time.
    - Accelerate improvement of upper bound.
    - Minimize gap at time limit.
    - ??
- This is a very complex multi-objective on-line optimization problem that is much more difficult to solve than the instance itself!

A Thousand Words

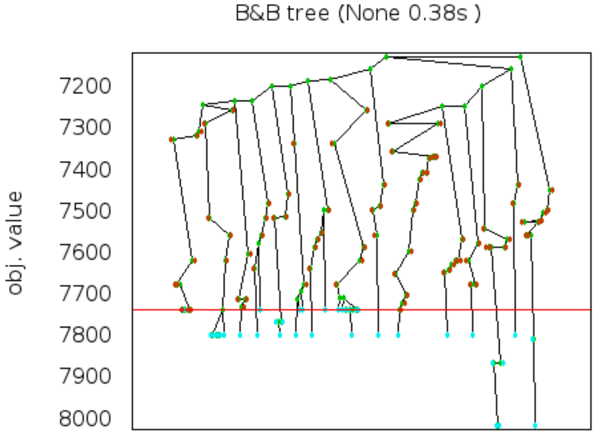


Figure: Tree after 400 nodes

A Thousand Words

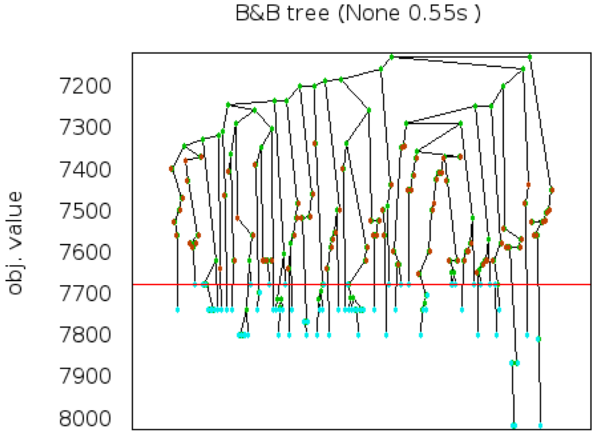


Figure: Tree after 1200 nodes

A Thousand Words

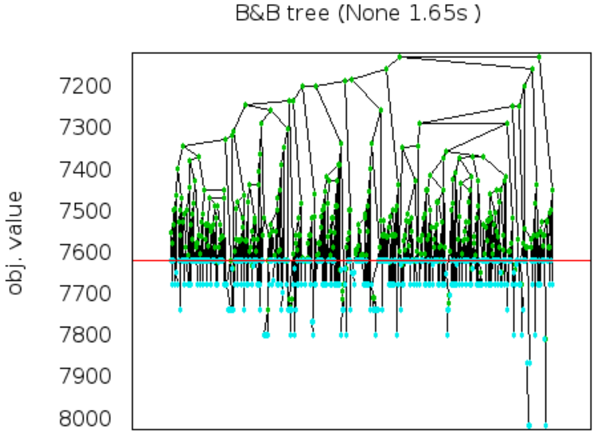


Figure: Final tree

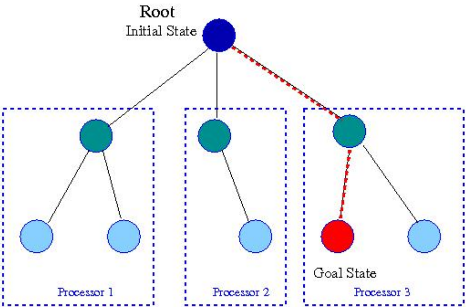
Parallelization of Tree Search

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.
- Branching 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.

Parallelizing Tree Search Algorithms

- In general, the search tree can be very large.
- The generic algorithm appears very easy to parallelize, however.

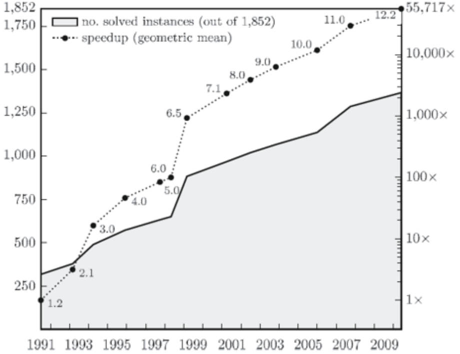


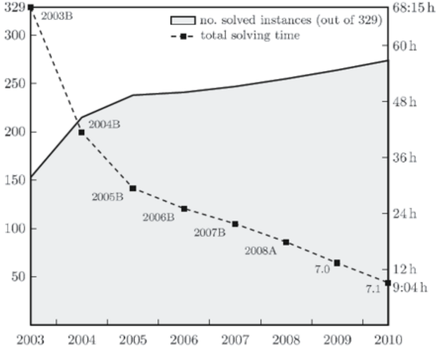
- The appearance is deceiving
  - The search graph is not known a priori and could be VERY unbalanced.
  - Naïve parallelization strategies are not generally effective.
  - It's difficult to determine how to divide the available work.

Outline

1. Introduction
    - Tree Search Algorithms
    - Historical Perspective
2. Parallel Algorithms
    - Definitions and Background
    - State of the Art
    - Challenges
3. Assessing Effectiveness
    - Sequential Performance
    - Parallel Scalability
4. Computational Results
    - Sequential Performance
    - Parallel Performance
    - Parallel Scalability
5. Conclusions

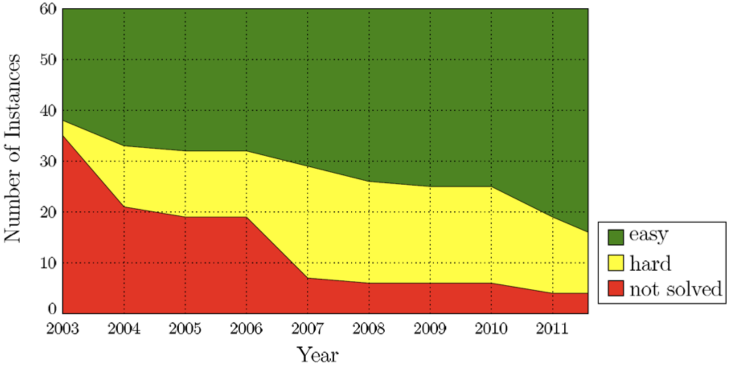
Evolution in Solver Performance





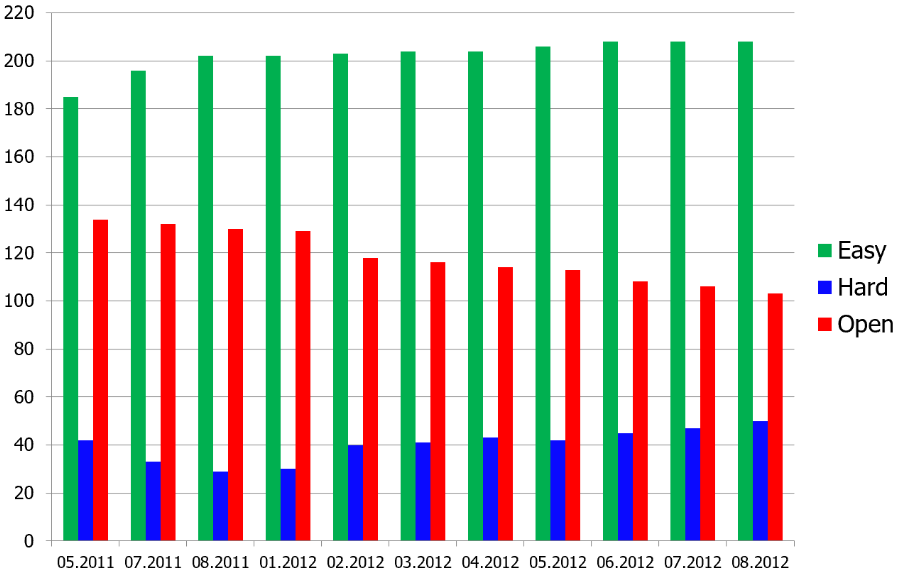
- Improvements in sequential performance have largely come from reductions in the amount of enumeration (smaller trees).
- Many specialized methods for addressing certain commonly occurring structures have been developed

Evolution of MIPLIB 2003

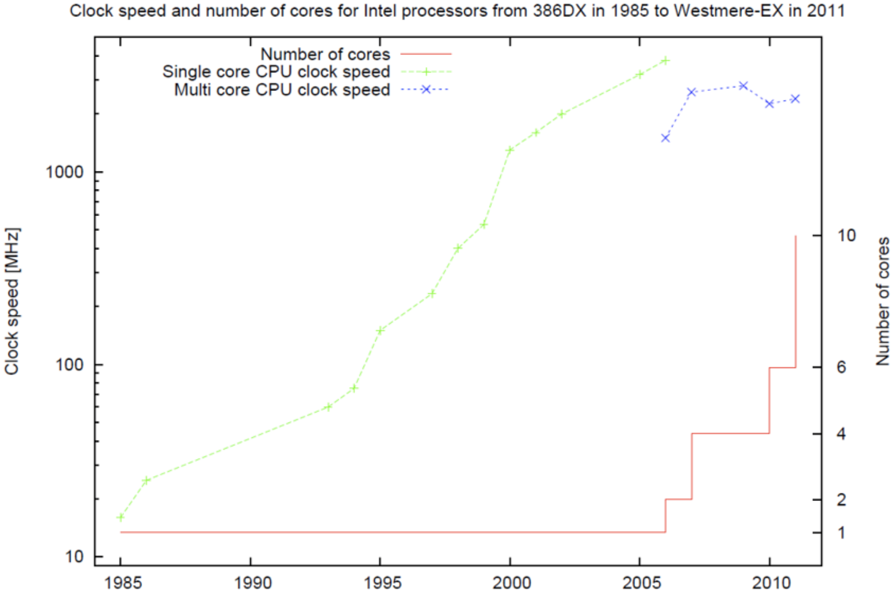


- *Easy* could be solved within an hour on a contemporary PC with a state-of-the-art solver.
- *Hard* are solvable but take a longer time or require specialized algorithms.
- *Open* problems are unsolved instances for which the optimal solution is not known.

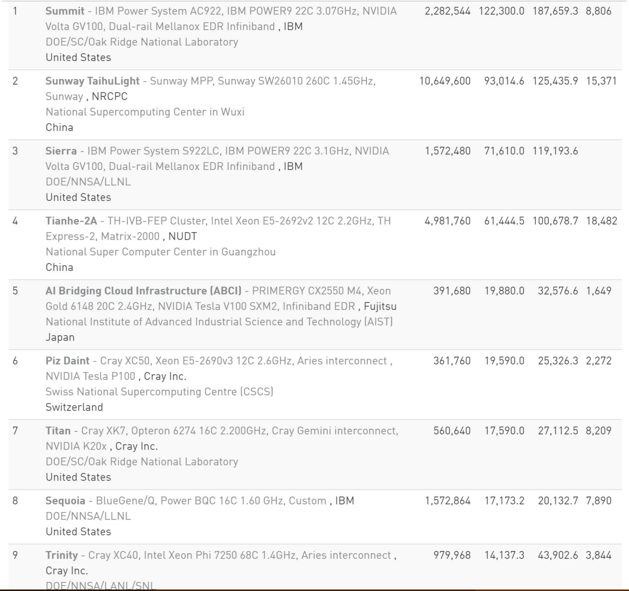
Evolution of MIPLIB 2010



Evolution of Parallel Architectures



Top 500



1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,282,544	122,300.0	187,659.3	8,806
2	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
3	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/NNSA/LLNL United States	1,572,480	71,610.0	119,193.6	
4	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
5	**AI Bridging Cloud Infrastructure (ABCI)** - PRIMERGY CX2550 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR , Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan	391,680	19,880.0	32,576.6	1,649
6	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	361,760	19,590.0	25,326.3	2,272
7	**Titan** - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x , Cray Inc. DOE/SC/Oak Ridge National Laboratory United States	560,640	17,590.0	27,112.5	8,209
8	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom , IBM DOE/NNSA/LLNL United States	1,572,864	17,173.2	20,132.7	7,890
9	**Trinity** - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray Inc. DOE/NNSA/LANL/SNL	979,968	14,137.3	43,902.6	3,844

Trends

- Total number of cores per parallel computer is increasing dramatically.
- Number of cores per CPU and per PE are also rising.
- The use of accelerators and other auxiliary processing is becoming more pervasive.
- The amount of memory per PE is rising, but amount of memory per core is generally falling.
- The memory/storage hierarchy is getting ever more complex.

Outline

1. Introduction
   - Tree Search Algorithms
   - Historical Perspective
2. Parallel Algorithms
   - Definitions and Background
   - State of the Art
   - Challenges
3. Assessing Effectiveness
   - Sequential Performance
   - Parallel Scalability
4. Computational Results
   - Sequential Performance
   - Parallel Performance
   - Parallel Scalability
5. Conclusions

Parallel Computers

- A *parallel computer* is a networked collection of processing elements, each comprised of
  - A collection of (multi-core) CPUs,
  - Memory and storage
  - Accelerators and co-processors
- Historically, most parallel computers could be considered to belong to one of two broad architectural classes:

- Shared memory
  - Each processor can access any memory location.
  - Processing units share information through memory IO.
  - *Software scales, hardware doesn't*.
- Distributed memory
  - Each processing unit has its own local memory and can only access its own memory directly.
  - Processing units share information via a network.
  - *Hardware scales, software doesn't*.

Algorithms and Parallel Systems

- 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.

What are the Goals?

Sequential Performance

Time (memory) required for a sequential algorithm to perform a fixed computation.

Parallel Scalability

- Classical: Time required for a parallel system to perform a fixed computation as a function of system resources (cores).
- Alternative 1: Amount of computation that can be done in fixed wallclock time as a function of system resources.
- Alternative 2: Amount of computation that can be done with fixed total resources as a function of wallclock time.

Overall Performance

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

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.

Performance versus Scalability

- As one may surmise, improving the sequential performance of a solver may be at odds with improving its scalability.
- Computations involving smaller trees are inherently more difficult to parallelize.
- This is one of many challenges facing us in parallelizing these algorithms.

Example: The Knapsack Problem

- We consider the binary knapsack problem:

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

- 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...

Outline

1. Introduction
   - Tree Search Algorithms
   - Historical Perspective
2. Parallel Algorithms
   - Definitions and Background
   - State of the Art
   - Challenges
3. Assessing Effectiveness
   - Sequential Performance
   - Parallel Scalability
4. Computational Results
   - Sequential Performance
   - Parallel Performance
   - Parallel Scalability
5. Conclusions

Current State of the Art

- Almost all parallel MILP solvers attempt to parallelize some *underlying sequential algorithm* (does this make sense?).
- Implementations differ in their approaches according to a number of properties.

Properties

- Tightness of the integration between the parallel framework and underlying sequential solver.
- Whether the parallel framework modifies the strategy taken by the underlying sequential solver.
- Granularity of the parallelization
- Approach to knowledge sharing and load balancing.
  - Initial static load balancing.
  - Dynamic load balancing in steady state.
- The degree to which they try to achieve determinism.

Granularity

Approaches differ according to the their level of*granularity*.

- Tree parallelism: Several trees are explored at once.
- Subtree parallelism: Several subtrees of the same tree may be searched simultaneously with little sharing of knowledge
- Node parallelism: A single tree can be searched in parallel by simply executing the sequential algorithm, but processing multiple nodes simultaneously
- Subnode parallelism: The processing of nodes can itself be parallelized.
    - Parallel solution of LP relaxation.
    - Parallel strong branching.
    - Parallel heuristics.
    - Decomposition methods.

Frameworks

- A number of generic frameworks have been developed which attempt to abstract out the approach to parallelization.
  - CHiPPS
  - UG
  - PEBBL
- A “framework” should be agnostic to the details of the underlying sequential algorithm.
- The degree to which one an existing sequential solver can be parallelized using a given framework depends on
  - the degree to which one can acccess the internals of the solver and
  - the degree to which the framework requires such access.

Shared versus Distributed Memory

- A shared memory parallel solver is relatively easy to develop, but difficult to make scalable.
  - Use of OpenMP compiler directives similar makes multi-threaded code easy to develop.
  - You must be careful with memory locks.
  - Overhead is more easily incurred than you would think.
- A distributed memory parallel solver is much harder to develop.
  - Requires more explicit communication with MPI or another message-passing protocol.
  - There are a wide array of frameworks that try to ease the complexity of implementation, but which one to use?
- Hybrid implementations are also obviously possible, but even more complex.

Outline

1. Introduction
   - Tree Search Algorithms
   - Historical Perspective
2. Parallel Algorithms
   - Definitions and Background
   - State of the Art
   - Challenges
3. Assessing Effectiveness
   - Sequential Performance
   - Parallel Scalability
4. Computational Results
   - Sequential Performance
   - Parallel Performance
   - Parallel Scalability
5. Conclusions

Where Can Parallel Computing Help?

- What are the reasons for poor sequential performance?
  - Genuine bad formulation
  - Bad dual bounds
  - LP is difficult/slow, especially reoptimizing
  - Bad numerical properties
  - Difficult to find primal solution
  - Large enumeration tree, e.g. due to symmetry
  - Just big
  - Nobody knows
- Important question: which of these can parallel computing help with?

Some Challenges We Face

- Inherent algorithmic difficulties
    - Tension between performance and scalability.
    - Unpredictable/Unbalanced trees.
    - Performance variability and non-determinism.
    - Ramp-up/Ramp-down.
    - Automatic tuning is crucial, but extremely difficult.
    - Many instances simply aren't good candidates.
- Difficulties in research and development
    - Instrumentation and debugging.
    - Non-determinism.
- Difficulties in assessment and analysis of results
    - Difficult to find a good test set.
    - Difficult to compare approaches/solvers.
    - Difficult to separate effects of hardware, software, and algorithm components.
- Difficulties in deployment
    - Difficult to develop portable approaches.
    - Hardware changes quickly.

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!

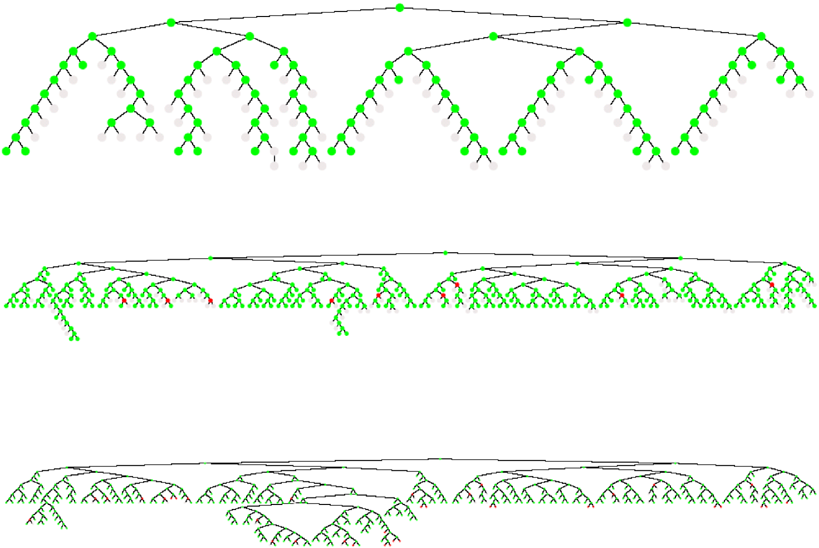
Barriers to Scalability: Sophisticated Architectures

- 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.

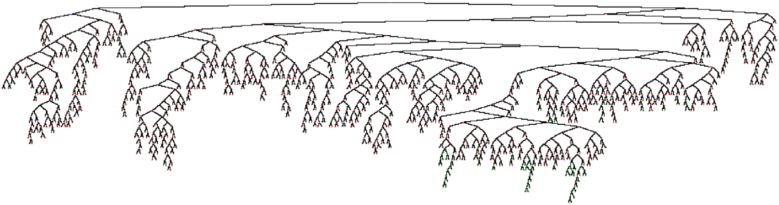
> - Cache memory 1-16x
> - Main memory (local to core) 10-100x
> - Main memory (attached to other cores) 100-700x
> - Co-located distributed memory
> - Remotely located distributed memory >1000x
> - Local disk >3,000,000x

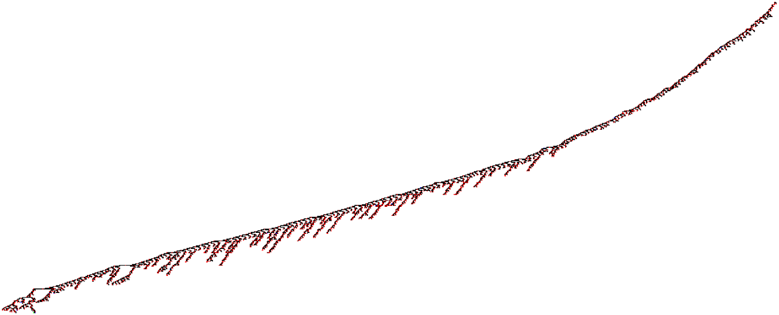
- Such complexity makes it *harder to achieve good parallel performance rather than easier*.
- Tools can help, but to a very limited extent.

Challenges from Tree Shape: Nice Trees

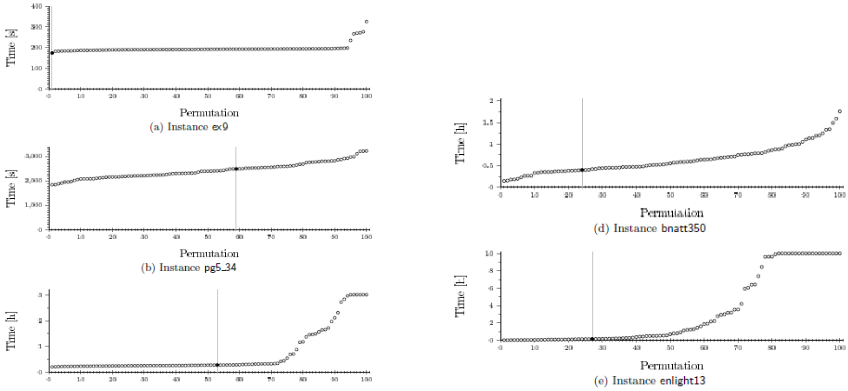


Challenges from Tree Shape: Ugly Trees





Challenges from Performance Variability



(a) Instance ex9

(b) Instance pg5_34

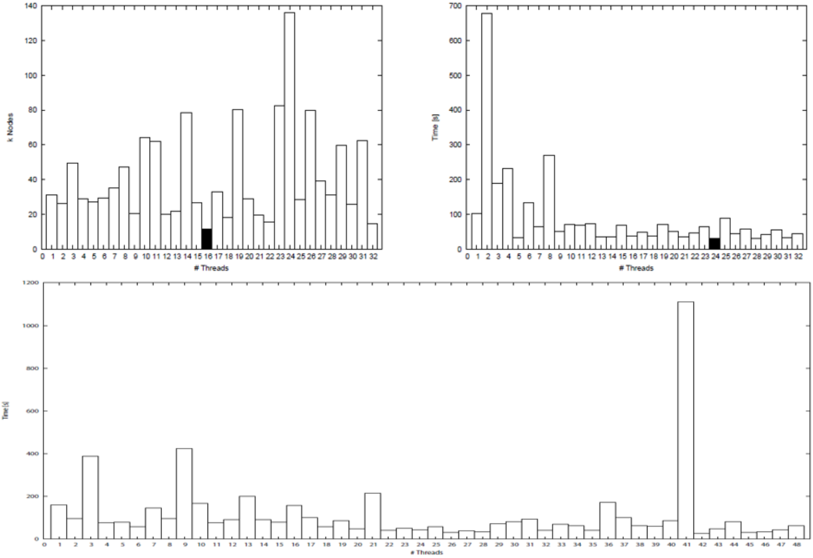
(c) Instance neos13

(d) Instance bnatt350

(e) Instance enlight13

Fig. 3: Solution times for 100 permutations

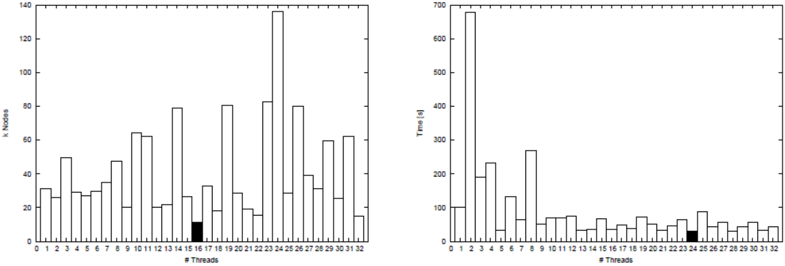
Challenges from Performance Variability



Numbers

courtesy of K. Fujisawa

Challenges from Performance Variability



(a) Total number of nodes explored (CPLEX)

(b) Wall clock solution time (GUROBI)

Fig. 4: Example of performance variability depending on the number of threads. Instance roll3000 on a 32 core computer. Filled bar indicates minimum

What Can Parallel Computing Realistically Do?

- The number of nodes in a given complete tree *doubles with each level*.
- With luck, doubling the number of processors allows exploring **one further level in the tree**.
- This is not typically enough to solve an unsolved problem or make a hard problem easy.
- We can really only hope to solve problems we can already solve *faster*.

Assessing Effectiveness

- 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?

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?

The Cold Hard Reality

Despite immense effort, efforts at parallelization have not been as successful as one would hope (to date).

Why is this?

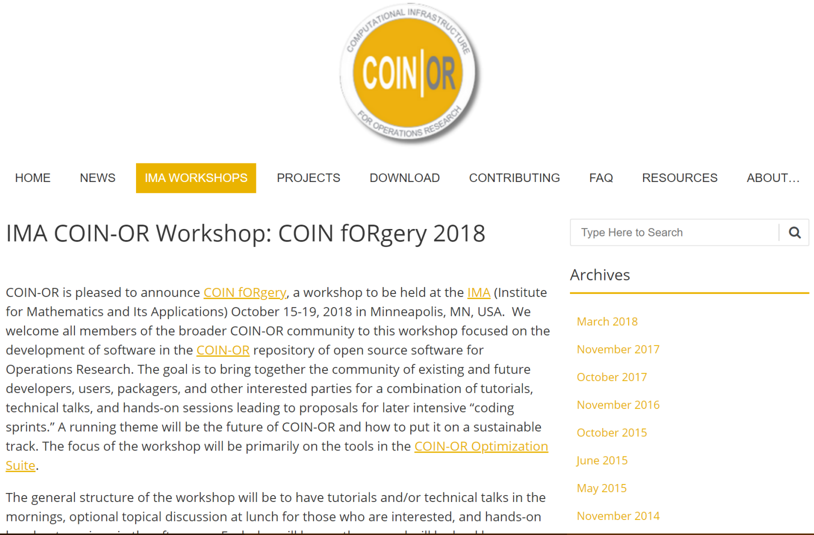
- It takes immense effort to do a single implementation.
- One must fix certain design details ahead of time using one's best understanding.
- Once the implementation is completed, one faces the challenge of assessing its performance and understanding how to improve it.
- It is difficult, if not impossible, to compare different approaches.
- All in all, making progress is very difficult.

Questions for Reflection

- Research Direction
  - Should we even bother to think about how to improve sequential algorithms without considering the implications for parallelization?
  - Should all algorithmic research be pursued taking into account that the algorithm needs to be parallelizable?
  - Is parallelizing the best sequential algorithms the right approach?
  - Should we start from scratch to develop parallel algorithms that achieve a better balance of performance and scalability?
  - Can we exploit GPUs?
- Practical/Software Issues
  - How do we support the maintenance of free and open source building blocks that enable experimentation?
  - How do we train our students in the fundamentals of computation?
  - How do we support the publication of both quality computational studies and quality software?

Commercial Break

https://www.coin-or.org/ima/oct2018/



IMA COIN-OR Workshop: COIN fORgery 2018

COIN-OR is pleased to announce COIN fORgery, a workshop to be held at the IMA (Institute for Mathematics and Its Applications) October 15-19, 2018 in Minneapolis, MN, USA. We welcome all members of the broader COIN-OR community to this workshop focused on the development of software in the COIN-OR repository of open source software for Operations Research. The goal is to bring together the community of existing and future developers, users, packagers, and other interested parties for a combination of tutorials, technical talks, and hands-on sessions leading to proposals for later intensive "coding sprints." A running theme will be the future of COIN-OR and how to put it on a sustainable track. The focus of the workshop will be primarily on the tools in the COIN-OR Optimization Suite.

The general structure of the workshop will be to have tutorials and/or technical talks in the mornings, optional topical discussion at lunch for those who are interested, and hands-on

Archives

- March 2018
- November 2017
- October 2017
- November 2016
- October 2015
- June 2015
- May 2015
- November 2014

Outline

1. Introduction
   - Tree Search Algorithms
   - Historical Perspective
2. Parallel Algorithms
   - Definitions and Background
   - State of the Art
   - Challenges
3. Assessing Effectiveness
   - Sequential Performance
   - Parallel Scalability
4. Computational Results
   - Sequential Performance
   - Parallel Performance
   - Parallel Scalability
5. Conclusions

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]

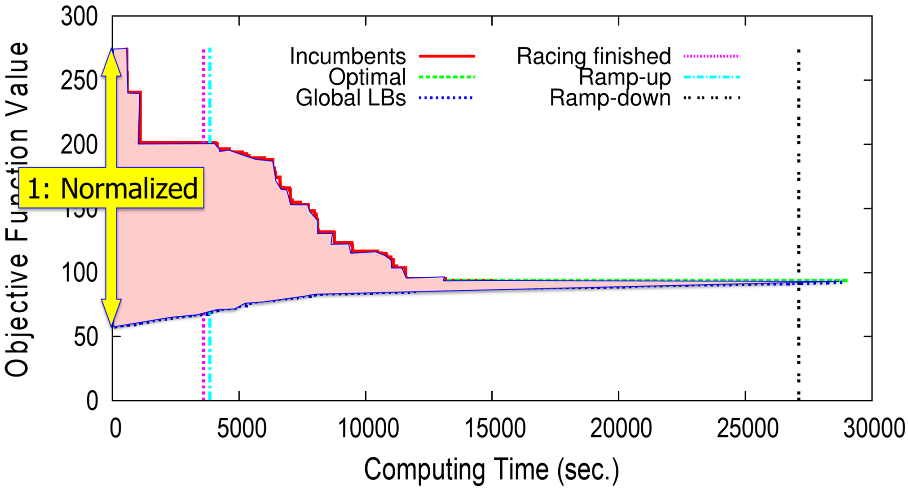


Figure: Example of a PDI plot

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* ⇐ Important!
- Possible measures for MILP
  - Gap
  - PDI

Outline

1. Introduction
   - Tree Search Algorithms
   - Historical Perspective
2. Parallel Algorithms
   - Definitions and Background
   - State of the Art
   - Challenges
3. Assessing Effectiveness
   - Sequential Performance
   - Parallel Scalability
4. Computational Results
   - Sequential Performance
   - Parallel Performance
   - Parallel Scalability
5. Conclusions

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.

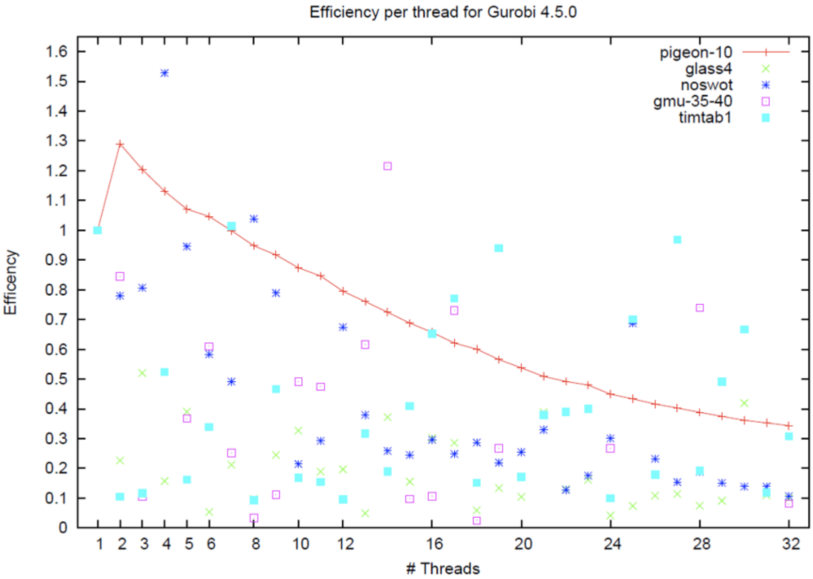
Alternatives to Classical Analysis

- 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

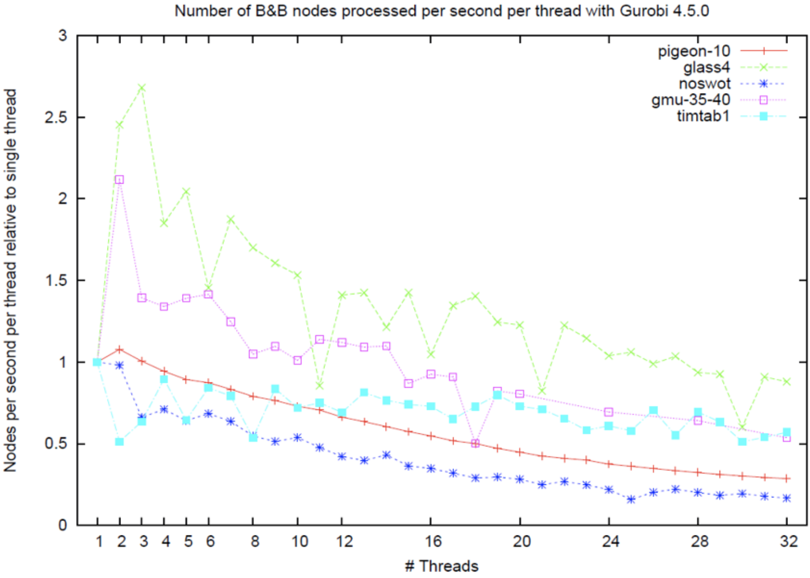
Direct Measures of Overhead

- 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

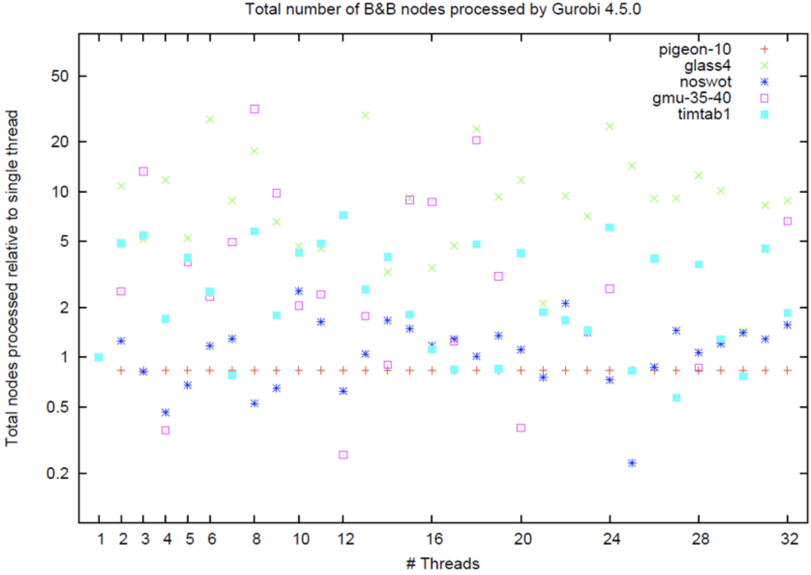
Efficiency Per Thread (Gurobi)



Node Throughput Versus Number of Threads

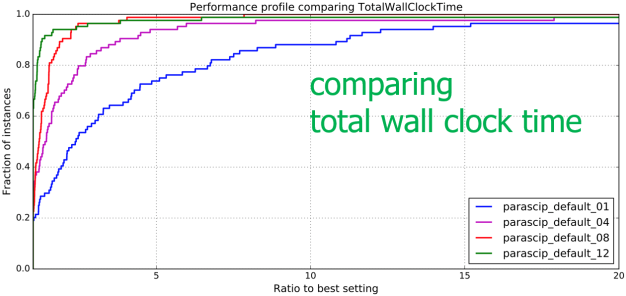


Node Efficiency Versus Number of Threads



Performance Profiles for Scalability Analysis

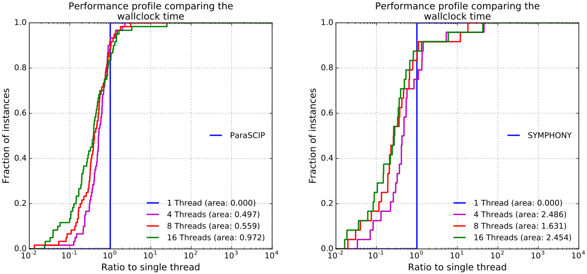
- Performance profiles are typically used to compare different algorithms
- They can, however, be used to compare the same algorithm under different conditions.
- For scalability, we compare with differing numbers of threads.
- A down side is that performance profiles compare to virtual best, whereas scalability compares to single-thread.



comparing
total wall clock time

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.

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

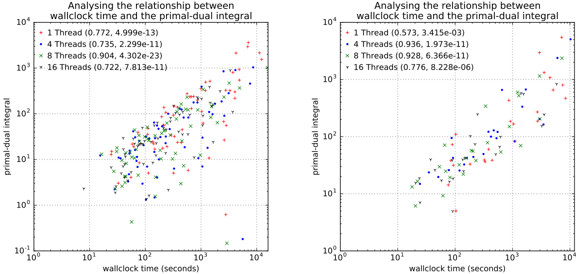
- A *measure of progress* is an estimate of what fraction of a computation has been completed.
- It may be very difficult to predict how much time remains in a computation.
- However, for computations that have already been performed once, it may be possible.
- Measures of progress can be used to assess 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 now “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.

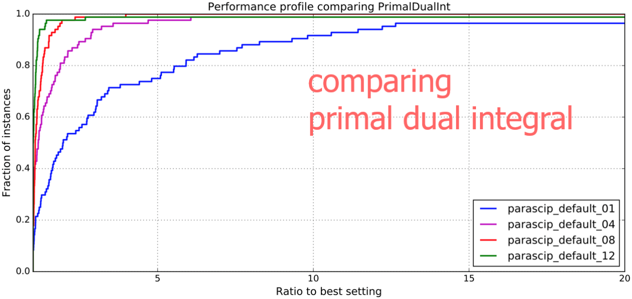


(a) ParaSCIP

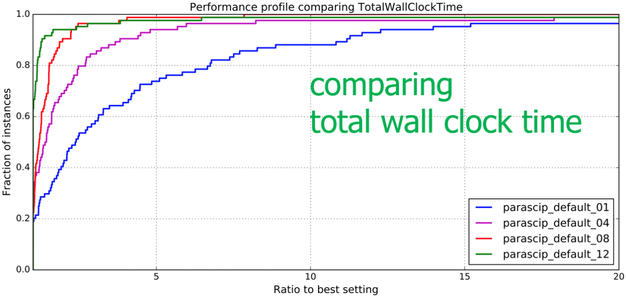
(b) SYMPHONY

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

Performance Profiles of Extended PDI and Wallclock



comparing
primal dual integral

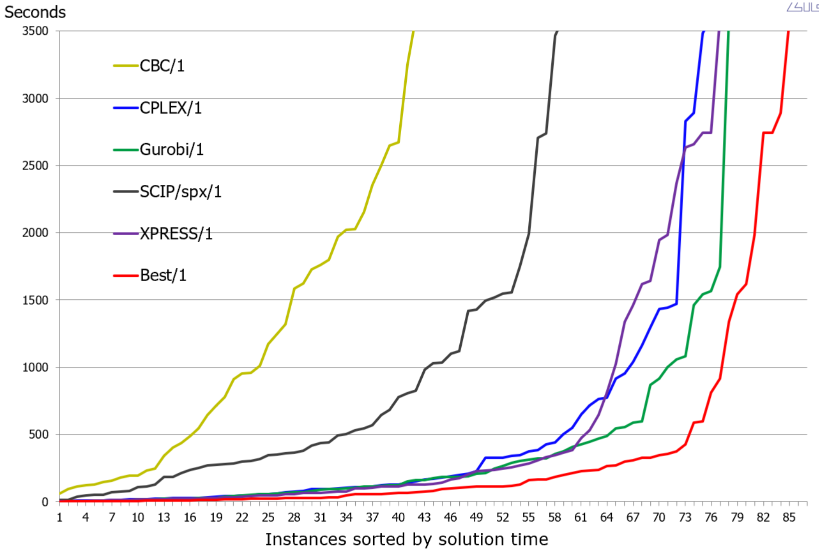


comparing
total wall clock time

Outline

1. Introduction
   - Tree Search Algorithms
   - Historical Perspective
2. Parallel Algorithms
   - Definitions and Background
   - State of the Art
   - Challenges
3. Assessing Effectiveness
   - Sequential Performance
   - Parallel Scalability
4. Computational Results
   - Sequential Performance
   - Parallel Performance
   - Parallel Scalability
5. Conclusions

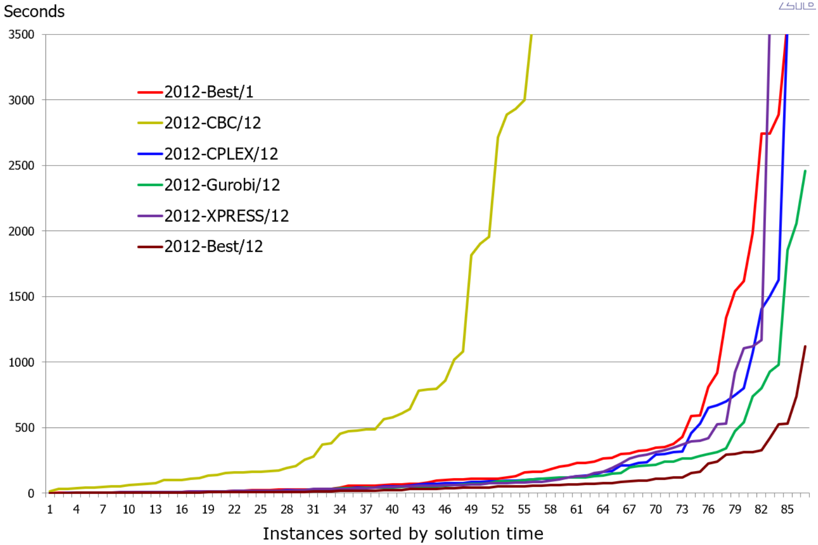
Sequential Performance of Solvers (Single Thread)



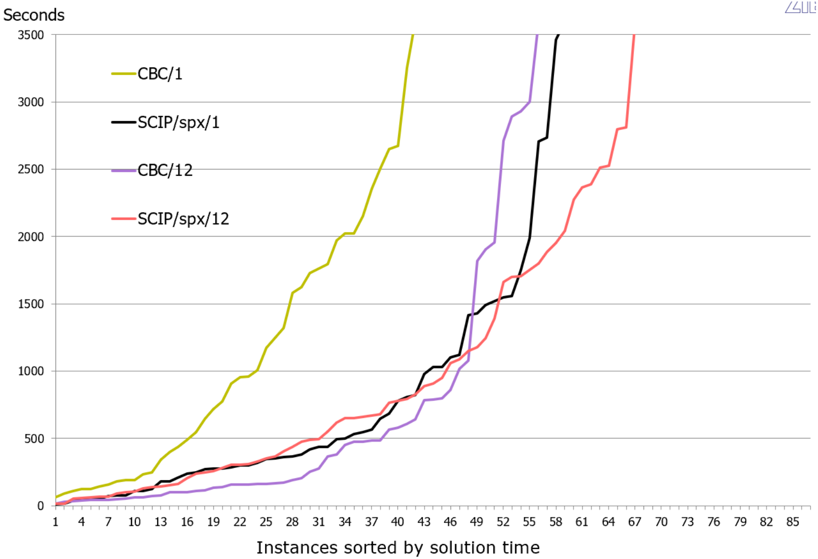
Outline

1. Introduction
   - Tree Search Algorithms
   - Historical Perspective
2. Parallel Algorithms
   - Definitions and Background
   - State of the Art
   - Challenges
3. Assessing Effectiveness
   - Sequential Performance
   - Parallel Scalability
4. Computational Results
   - Sequential Performance
   - Parallel Performance
   - Parallel Scalability
5. Conclusions

Parallel Performance of Solvers (Shared Memory, 12 Threads)



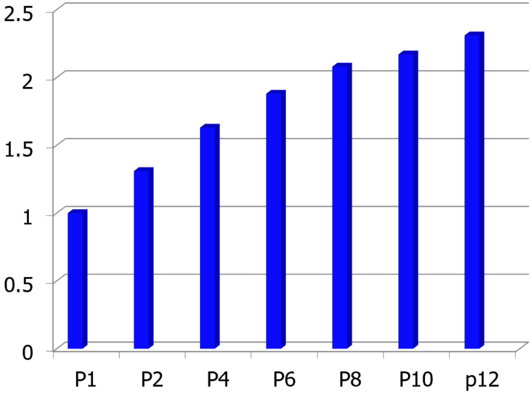
Parallel Performance of Solvers (Shared Memory, 12 Threads)



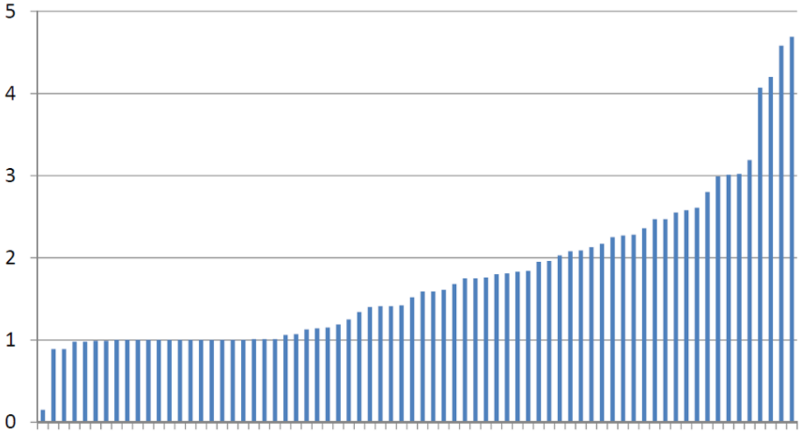
Parallel Performance of Early Gurobi Version

#Threads	Speedup
1	1.0
2	1.31
4	1.63
6	1.88
8	2.08
10	2.17
12	2.31

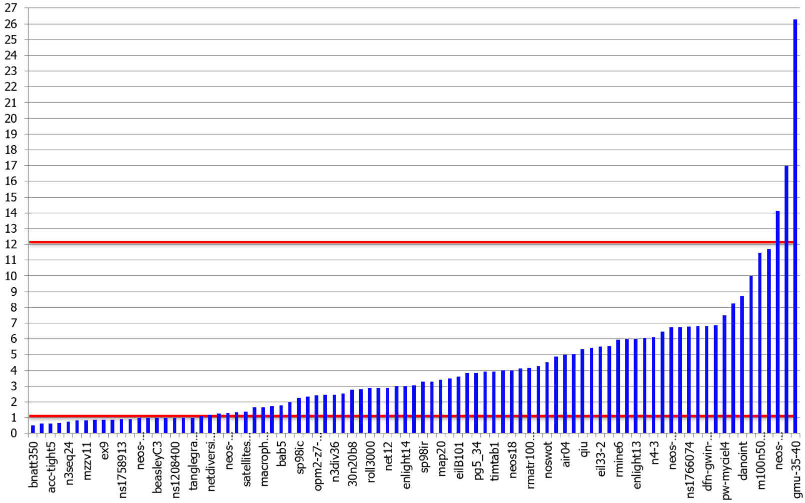
Slide courtesy of Ed Rothberg



Speeedups on 4 Cores by Model (Gurobi)



Speeedups Best 1/12 by Model (Gurobi)



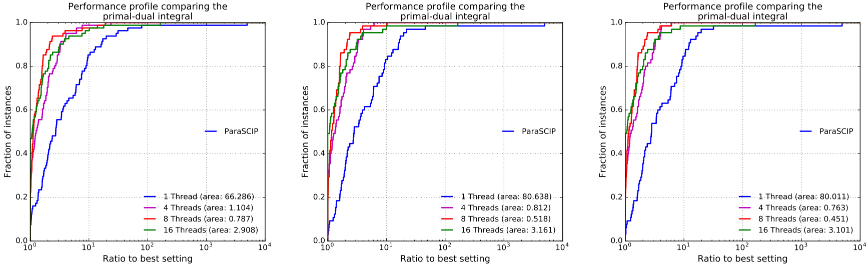
Outline

1. Introduction
   - Tree Search Algorithms
   - Historical Perspective
2. Parallel Algorithms
   - Definitions and Background
   - State of the Art
   - Challenges
3. Assessing Effectiveness
   - Sequential Performance
   - Parallel Scalability
4. Computational Results
   - Sequential Performance
   - Parallel Performance
   - Parallel Scalability
5. Conclusions

Experiments Assessing Parallel Scalability

- 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



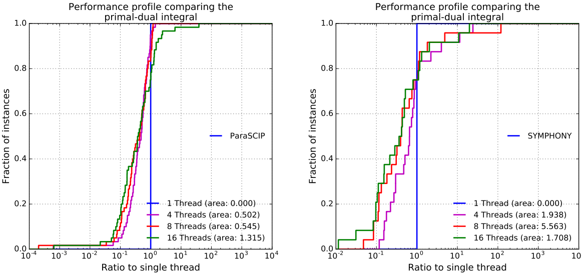
(a) 18000 seconds limit

(b) 9000 seconds limit

(c) 4500 seconds limit

Figure: Performance profile of PDI for ParaSCIP on MIPLIB2010.

Scalability Profile Using Extended PDI

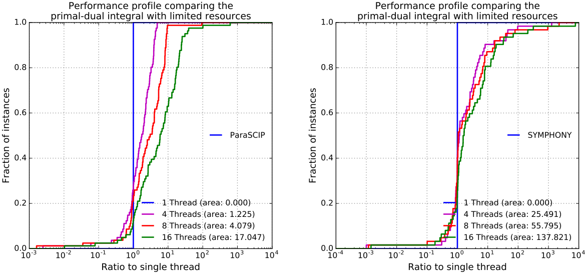


(a) ParaSCIP

(b) SYMPHONY

Figure: Scalability profile of the extended PDI

Scalability Profile with Fixed Compute Time

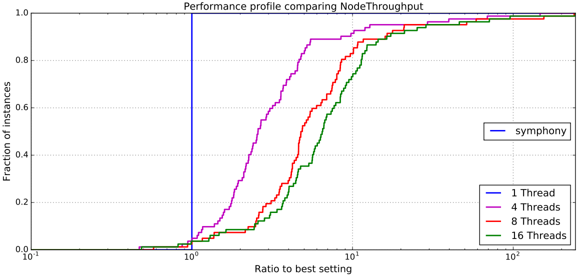


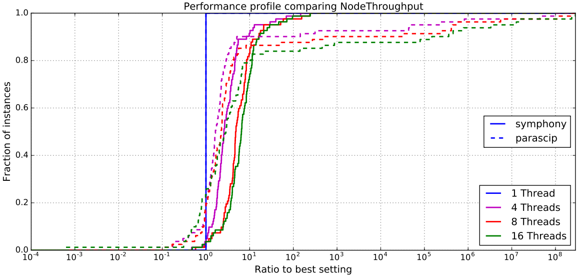
(a) ParaSCIP

(b) SYMPHONY

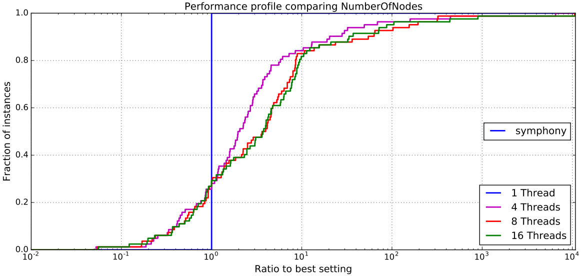
Figure: The scalability profile of PDI with fixed compute time.

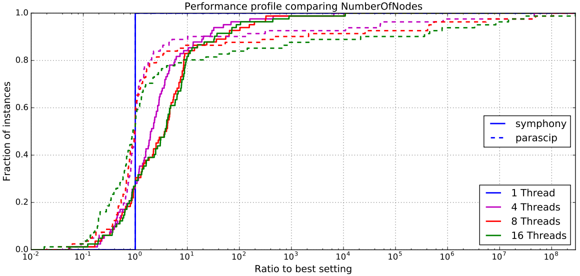
Node Throughput Scalability Profile



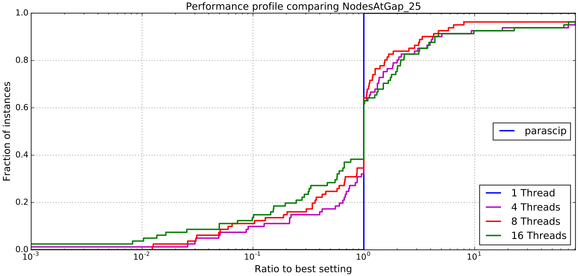


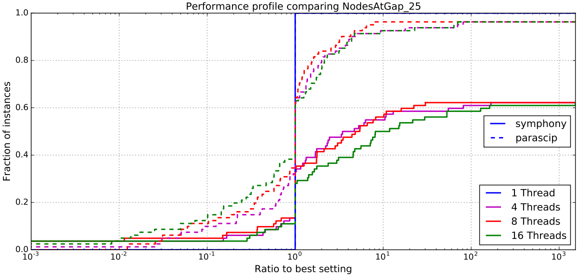
Number of Nodes Scalability Profile





Number of Nodes at Gap Scalability Profile





Conclusions

- We presented an overview of the current state-of-the-art and challenges facing developers of solvers for MILP.
- Parallelization of algorithms for solution of MILPs is a very difficult challenge that is far from solved.
- It is not clear if we are going down the right road or whether we should start from scratch with some fresh thinking.
- Ideas welcome!

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.

S.J. Maher, T.K. Ralphs, and Y. Shinano. Assessing effectiveness of branch-and-bound algorithms. 2018.

T.K. Ralphs and M. 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.

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-R3, Lehigh University, 2016. 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. *Scalable Algorithms for Parallel Tree Search*. Phd, Lehigh University, 2007. URL http://coral.ie.lehigh.edu/{~}ted/files/papers/YanXuDissertation07.pdf.

References III

Y. Xu, T.Kk Ralphs, L. Ladányi, and M.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.

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.

Y. Xu, T.K. Ralphs, L. Ladányi, and M.J. Saltzman. Computational experience with a software framework for parallel integer programming. *The INFORMS Journal on Computing*, 21:383–397, 2009. doi: 10.1287/ijoc.1090.0347. URL http://coral.ie.lehigh.edu/~ted/files/papers/CHiPPS-Rev.pdf.