New Options for Solving Giant LPs

Ed Rothberg, Ph.D. Chief Scientist and Chairman,
Gurobi Optimization

Expanding Horizons for LP

First-Order Methods

- Recent excitement about first-order methods for LP
  - “*Practical Large-Scale Linear Programming using Primal-Dual Hybrid Gradient*”, Applegate, Diaz, Hinder, Lu, Lubin, O’Donoghue, and Schudy, NeurIPS 2021
  - Building on: “*An efficient primal-dual hybrid gradient algorithm for total variation image restoration*”, Zhu and Chan, 2008
- Interest driven largely by prevalence and power of GPUs
  - “*cuPDLP.jl: A GPU Implementation of Restarted Primal-Dual Hybrid Gradient for Linear Programming in Julia*”, Lu and Yang, 2023
  - “*cuPDLP-C: A Strengthened Implementation of cuPDLP for Linear Programming by C language*”, Lu, Yang, Hu, Huangfu, Liu, Liu, Ye, Zhang, and Ge, 2023
- Important to understand where these methods fit in the LP landscape

Characteristics of PDHG

- Positives:
    - Excellent parallelization on GPUs
        - A sequence of…
            - Sparse matrix-vector multiplies
            - Dense vector operations
        - Perfect for GPU
            - HBM (High Bandwidth Memory)
                - 8-40X+ faster than CPU memory

- Negatives:
    - Low-accuracy solutions
        - PDHG typical: 1e-4 relative tol
        - Gurobi default: 1e-6 absolute tol
        - Could limit crossover
    - Very sensitive to parameters
    - Often fail to converge

Parallelism in LP

Parallel Barrier

ELSEVIER

Operations Research Letters
Volume 18, Issue 4, February 1996, Pages 157-165

Gigaflops in linear programming

Irvin J. Lustig [a], Edward Rothberg [b]

Show more

- Barrier solver, modest parallelism
  - Typically 16 or fewer cores
- Barrier runtime historically dominated by sparse Cholesky factorization
  - Parallelizes quite nicely
  - Barrier now significantly faster than simplex on a broad test set
    - Mainly due to parallelism
- How far can it be scaled?

Important Steps in Barrier Algorithm

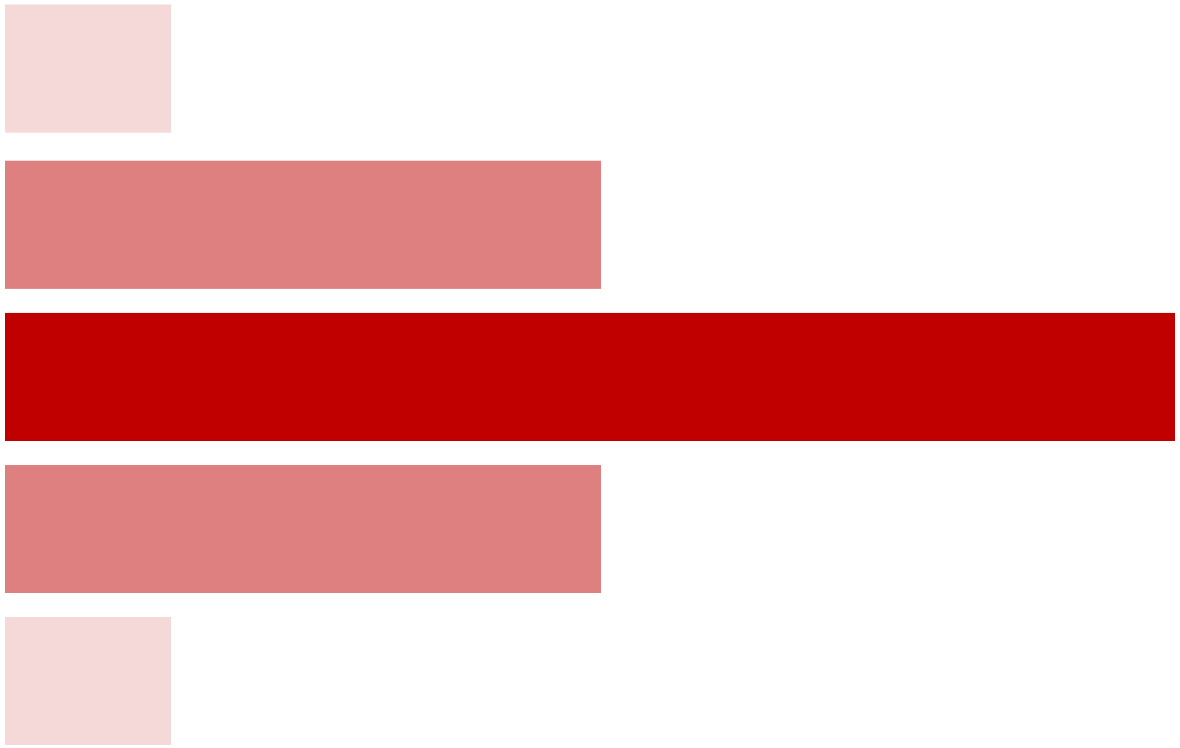
- Presolve
- Sparse matrix reordering
  - Minimum degree, nested dissection
- Factorization
  - Sparse Cholesky
- Step computation
  - Triangular solves using factor matrix
  - Often many steps from one factor ("Multiple central corrections")

Typically followed by…

- Crossover
  - From an interior solution to a basic solution
  - Essentially a less complicated form of simplex

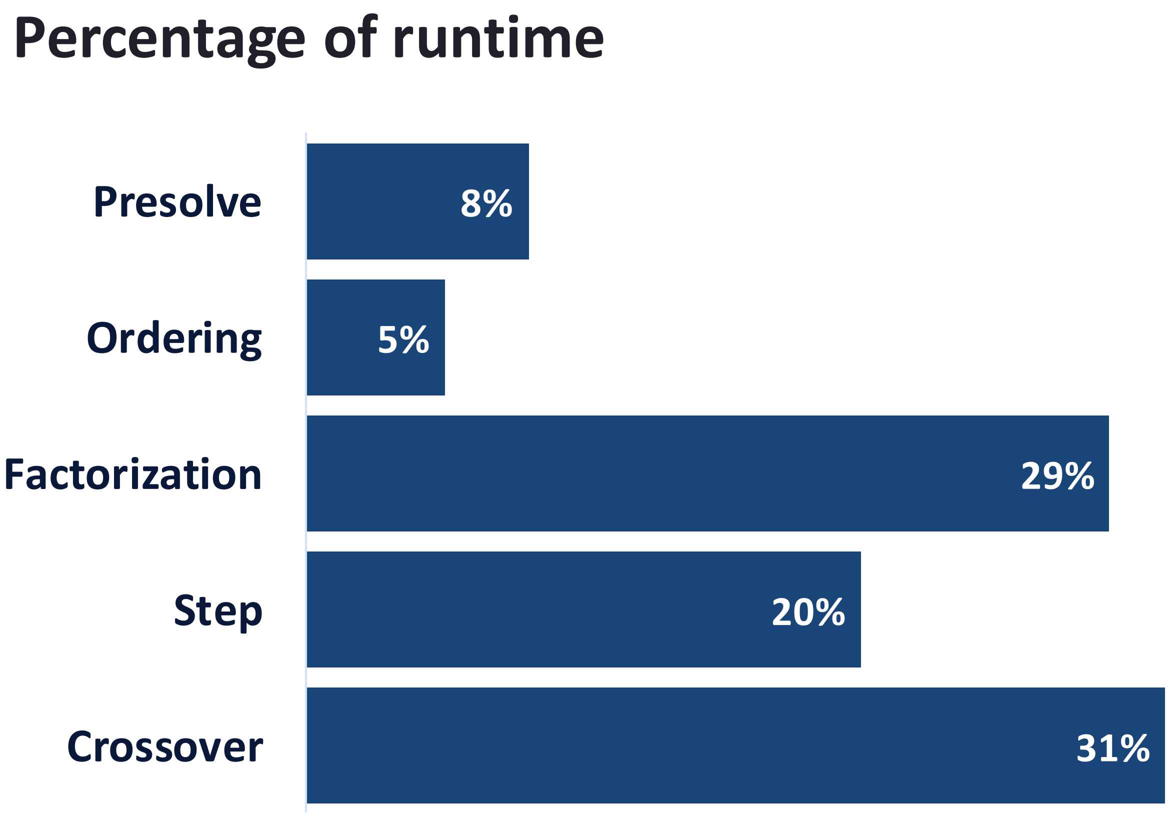
Parallelization Opportunities

- Presolve
- Sparse matrix reordering
- Factorization
- Step computation
- Crossover



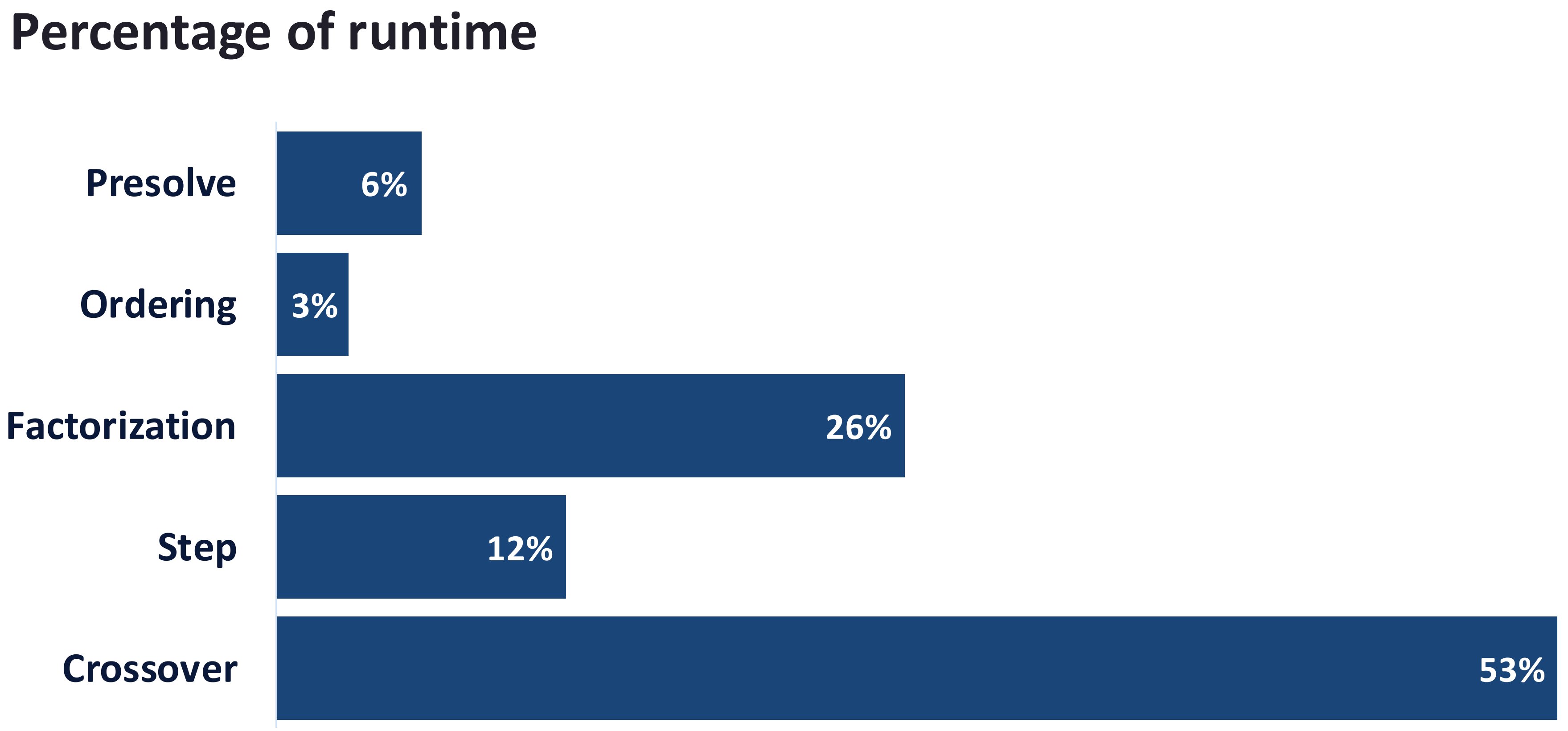
Barrier Runtime Breakdown

- Runtime breakdown
  - AMD 7313P, 16-core CPU, runtime>100s (199 models)



Barrier Runtime Breakdown

- Runtime breakdown
  - AMD 7313P, 16-core CPU, runtime>1000s (46 models)



Gauging Scope for Improvement

Costs

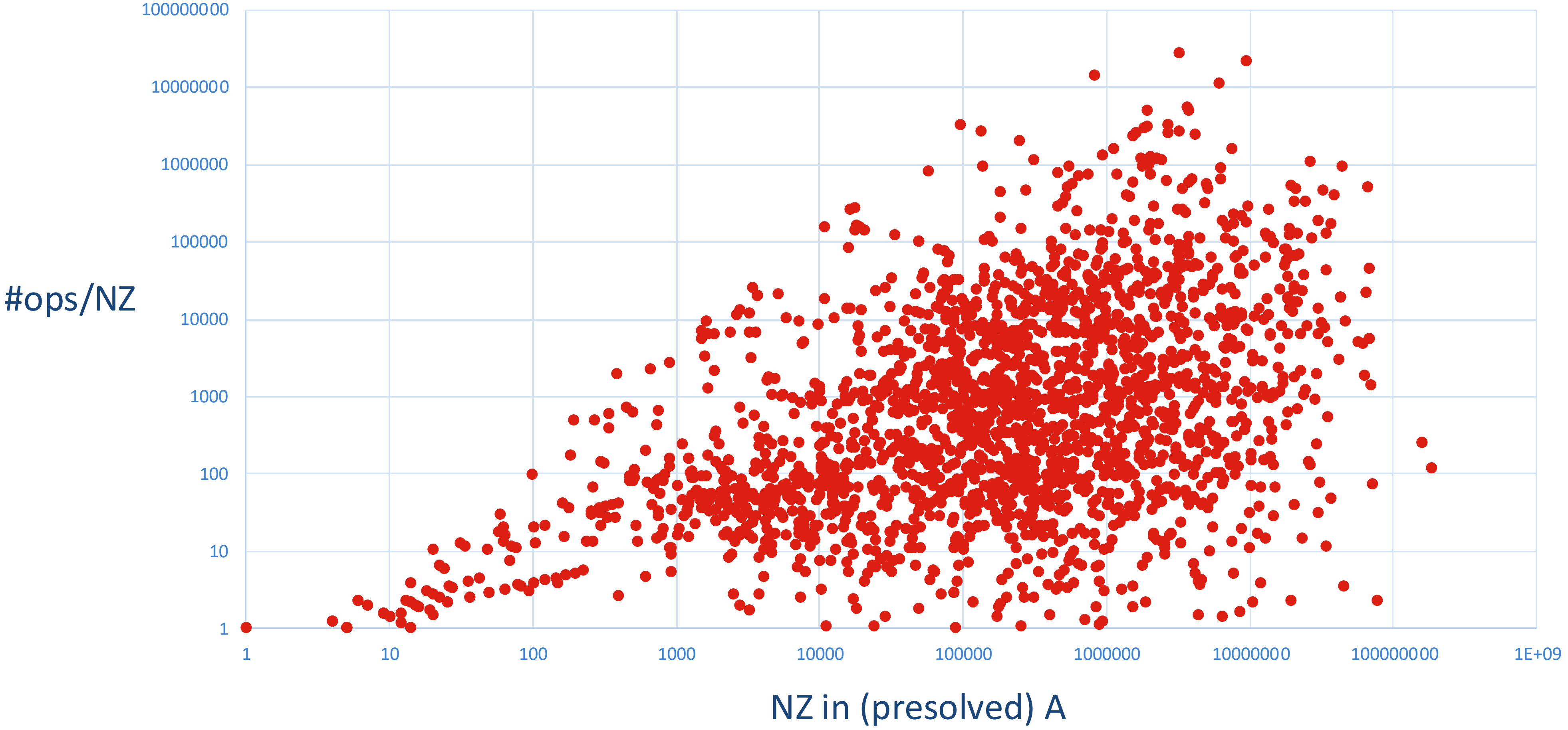
- Interior-point:
  - Dominant cost: sparse factorization
  - Metric: FP operations
  - Number of iterations: ~100

- PDHG:
  - Dominant cost: matrix-vector multiply
  - Metric: NZ in A
  - Number of iterations: 10K+

- Scope for improvement: ratio of

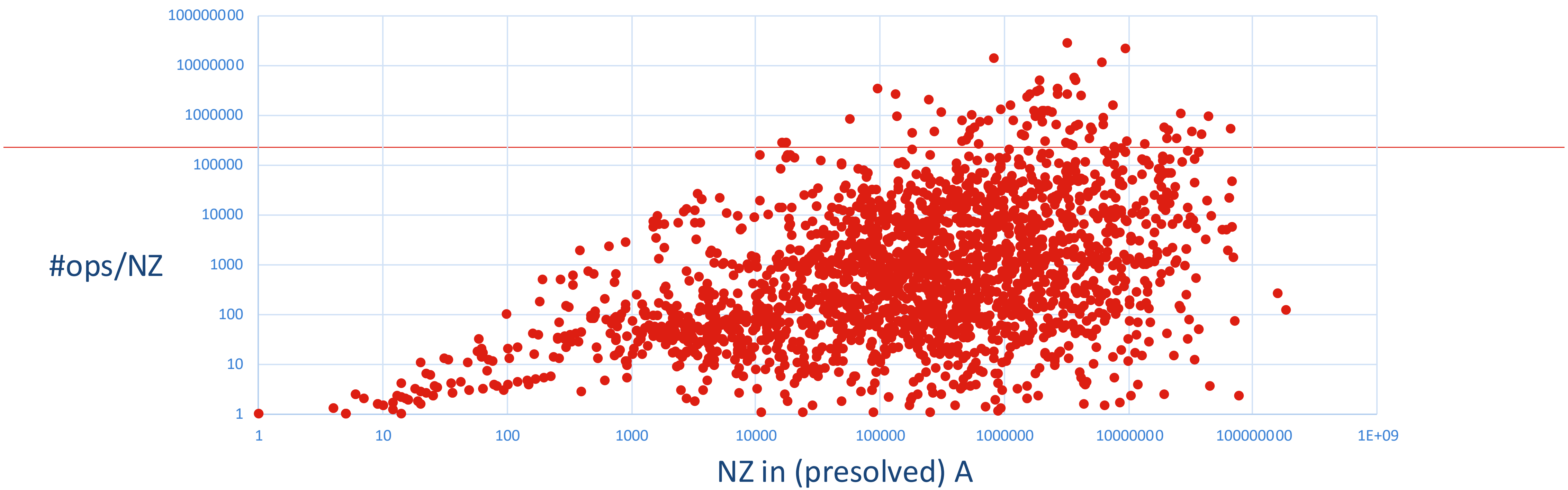
#Factorization ops / #NZ in A

#Factorization ops / #NZ in A



- For 2576 models in LP test set

#Factorization ops / #NZ in A



- Back of the envelope – modern CPU…
  - Grace CPU: 3.5 Gflops vs (384 GB/s / 24 bytes per NZ / iteration): 220 flops = 1 NZ
  - ~100 barrier iterations vs at least 10000 PDHG iterations to solve: 100X more iterations
  - Breakeven is probably ~52K ops/NZ
    - Look for >220K ops/NZ to have significant scope for improvement

Scope for Improvement: Computing

PDHG Paper…

- PDHG on modern GPU versus barrier on…

CPU-based methods, respectively. To enable comprehensive understanding for cuPDLP-C under advanced hardware, we run the CPU solvers on AMD Ryzen 9 5900X, whereas the GPU ones are tested on NVIDIA H100 80GB HBM3. Except for techniques like

The AMD Ryzen 9 5900X was released on **November 5, 2020**. It was a desktop processor with a launch price of $549.

PDHG Blog Post…

- For the CPU LP solver, I used a recommended CPU setup: AMD EPYC 7313P servers with 16 cores and 256 GB of DDR4 memory.
- For the cuOpt LP solver, I used an NVIDIA H100 SXM Tensor Core GPU to benefit from the high bandwidth and ran without presolve.

The AMD EPYC 7313P was released on March 15, 2021:

	AMD EPYC 7313P
Release date	March 15, 2021
Market	Server/workstation
Launch price	$1,083

GPU Advantages (vs Modern Desktop)

- Floating-point performance (fp64 peak):
  - Nvidia GH200 GPU: 34 Tflops
  - AMD Zen 5 (16 cores): 2.6 Tflops
- Memory bandwidth (peak):
  - Nvidia GH200 GPU: 4 Tbytes/s (HBM 3)
  - AMD Ryzen 5: 90 GB/s (DDR5)

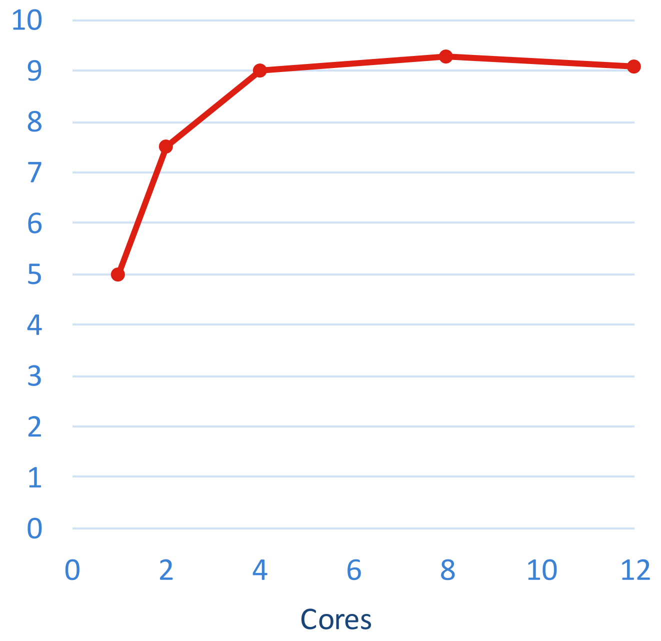
GPU Advantages (vs Modern High-End CPU)

- Floating-point performance (fp64 peak):
  - NVidia GH200 GPU: 34 Tflops
  - AMD Zen 5 (16 cores): 2.6 Tflops
  - NVidia GH200 CPU (72 cores): 3.5 Tflops
  - AMD EPYC 5 (128 cores): 23 Tflops
- Memory bandwidth (peak):
  - NVidia GH200 GPU: 4 Tbytes/s (HBM 3)
  - AMD Ryzen 5: 90 GB/s (DDR5)
  - NVidia GH200 CPU: 384 Gbytes/s (LPDDR5X)
  - AMD EPYC 5: 576 Gbytes/s
  - Apple M4 Max: 546 Gbytes/s

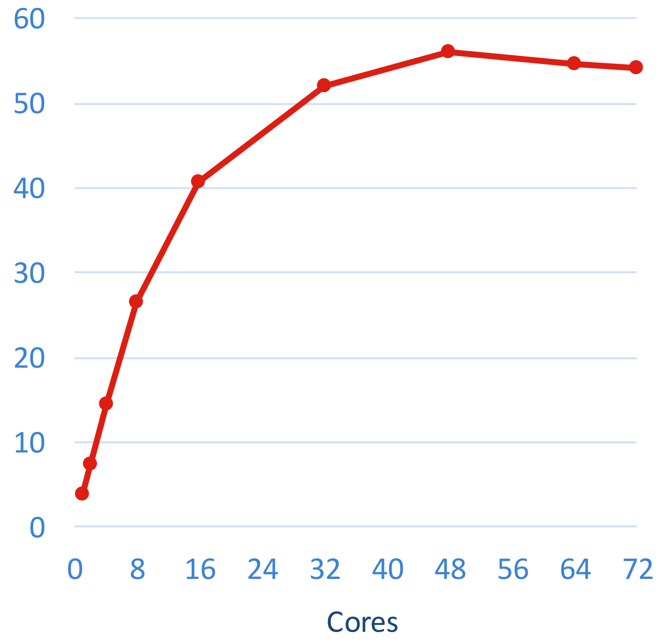
Memory-Bound vs CPU-Bound Applications

- Long-standing historical distinction:
  - *Memory-bound* applications:
    - Performance limited by memory system speed
    - Benchmark: STREAM, HPCG
    - Closest LP algorithm: PDHG
  - *Compute-bound* applications:
    - Performance limited by processor speed
    - Benchmark: LINPACK, HPL
    - Closest LP algorithm: Barrier

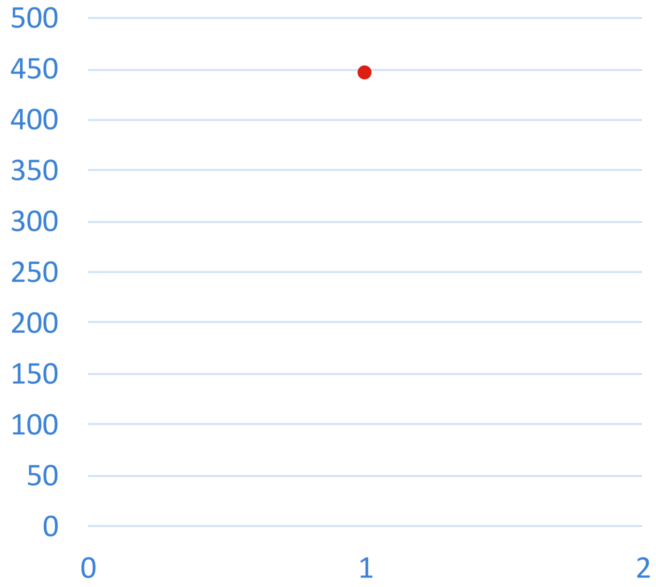
Scaling – PDHG (Iterations per Second, model *zib01*)



AMD Ryzen 9 9900X, 90 GB/s



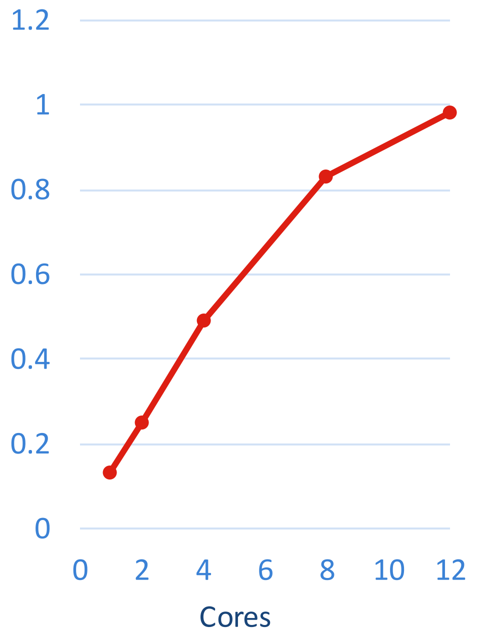
Nvidia Grace CPU, 384 GB/s



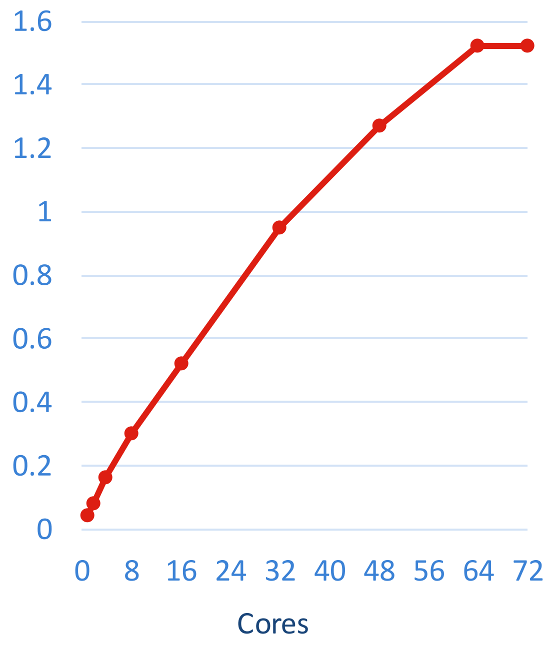
Nvidia Hopper GPU, 4 TB/s

- Improvement from Grace CPU to Hopper GPU: ~8X

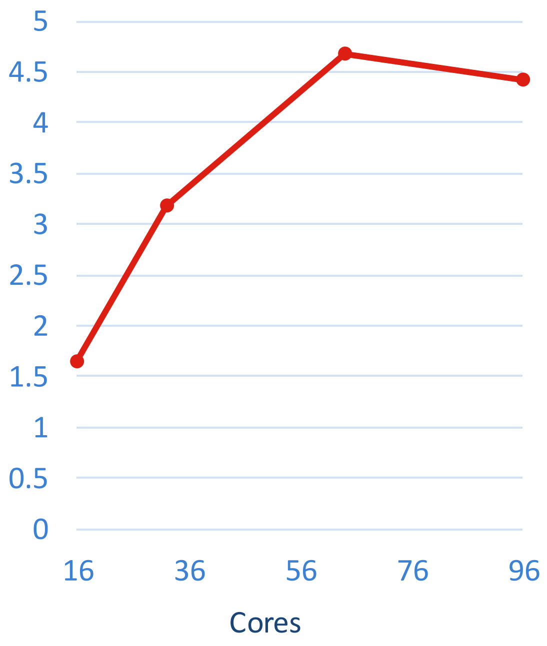
Scaling – Matrix-Matrix Multiply (fp64 Tflops)



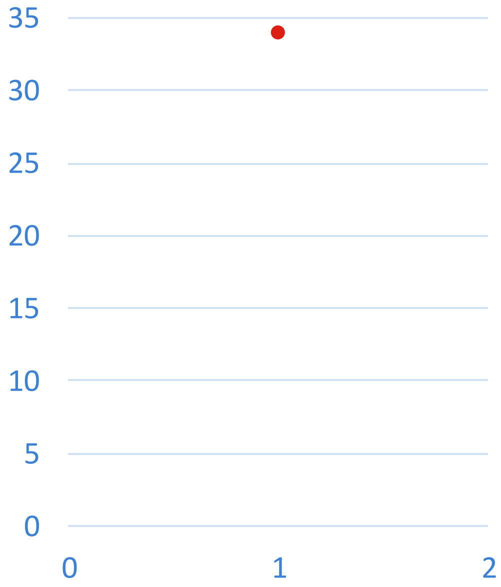
Ryzen 9 9900X, 90 GB/s



Nvidia Grace CPU, 384 GB/s



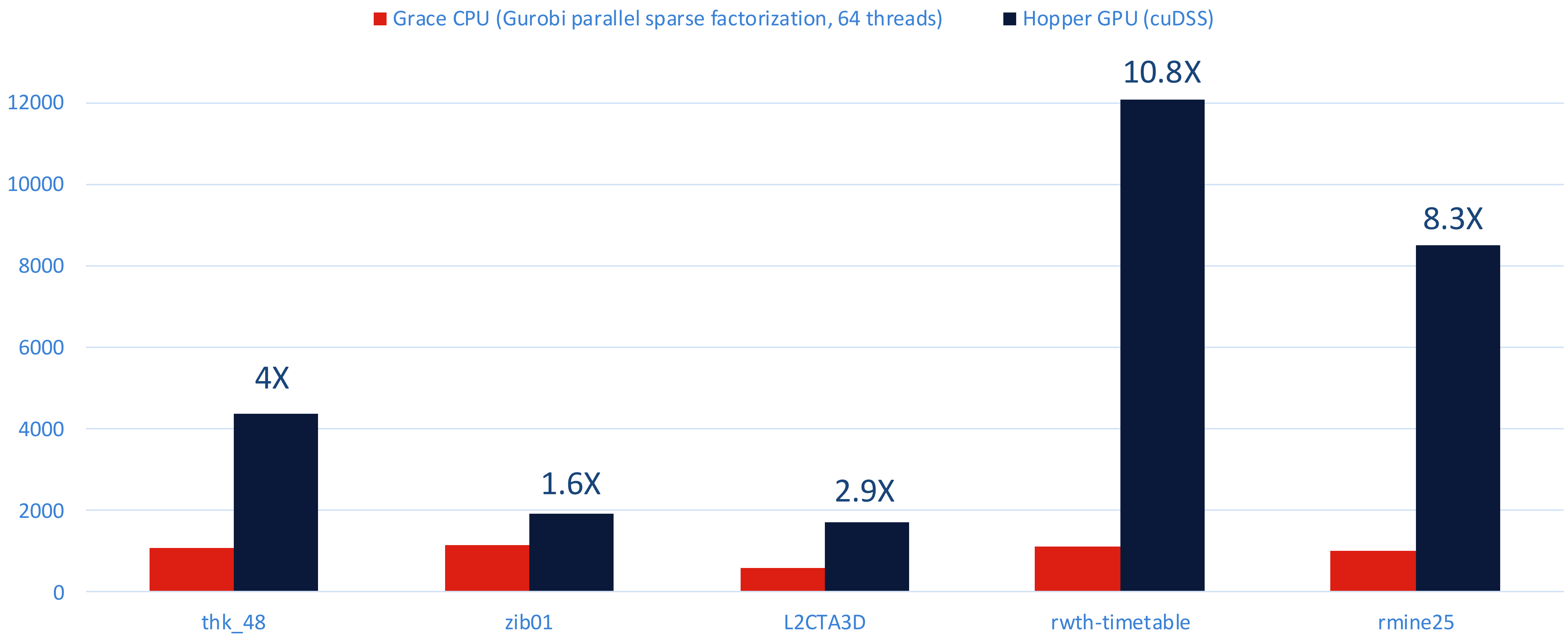
EPYC 9655 (96 cores), 576 GB/s



Nvidia Hopper GPU, 4 TB/s

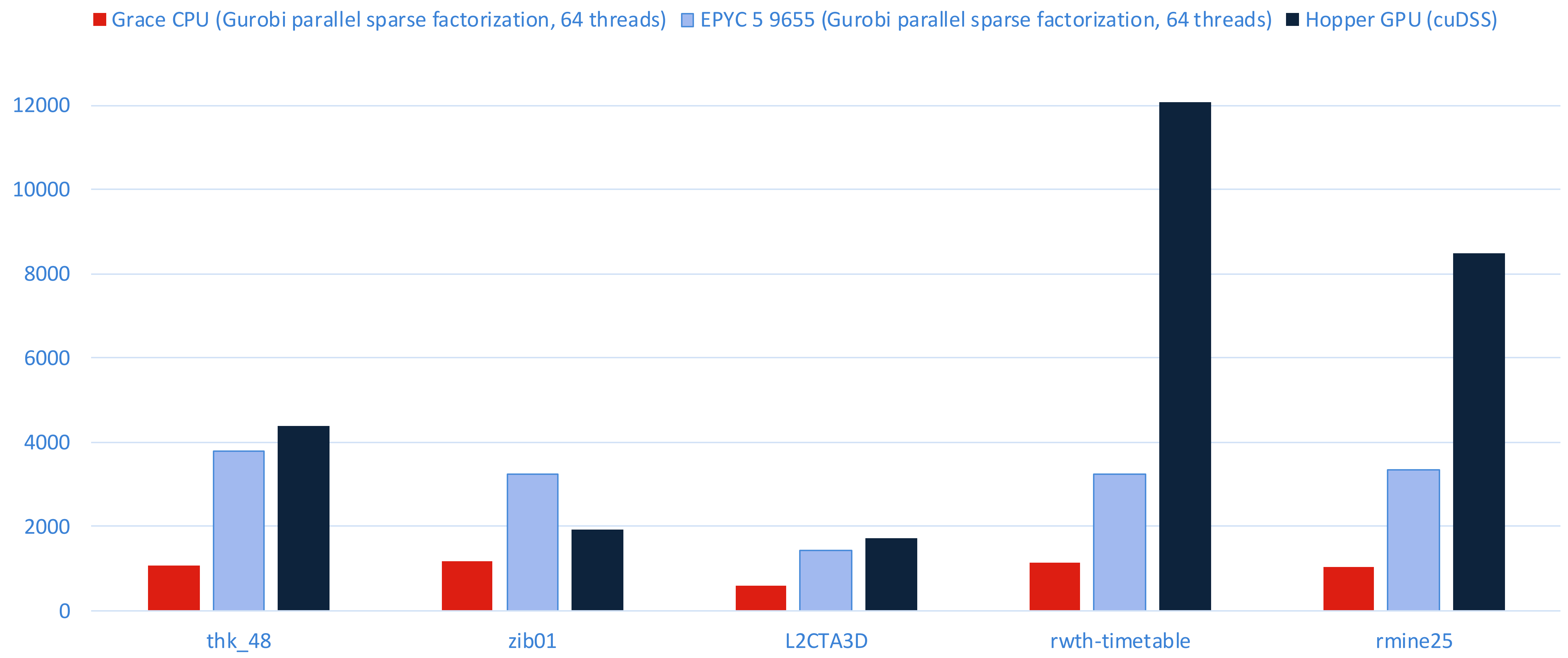
- Improvement from Grace CPU to Hopper GPU: ~23X
- Improvement from EPYC 9655 to Hopper GPU: ~7X

Sparse Factorization Performance (Gflops)



- Geometric mean improvement – Grace CPU to Hopper GPU: ~4.5X

Sparse Factorization Performance (Gflops)



- Geometric mean improvement - EPYC 9655 to Hopper GPU: ~1.5X

Options for Exploiting Parallelism

PDHG (parallel CPU and GPU implementations)

- Implementation mirrors NeurIPS 2021 paper
  - …and associated open-source code
- Performance limited by speed of sparse matrix-vector multiplies
  - Easily saturates most memory systems
- Integrated with Gurobi 12.0, but not yet released

[Submitted on 9 Jun 2021 (v1), last revised 7 Jan 2022 (this version, v2)]

Practical Large-Scale Linear Programr

David Applegate, Mateo Díaz, Oliver Hinder, Haihao Lu, Mi

We present PDLP, a practical first-order method for linear prograr
applications. In addition, it can scale to very large problems becau
hybrid gradient (PDHG) method, popularized by Chambolle and P
new techniques with older tricks from the literature; the enhancer
PDLP improves the state of the art for first-order methods appliec
from MIPLIB 2017. With a target of 10^{-8} relative accuracy and 1 h
reduction in the number of instances unsolved (from 227 to 49).
where our open-source prototype of PDLP, written in Julia, outper

Barrier – Parallel Cholesky Factorization

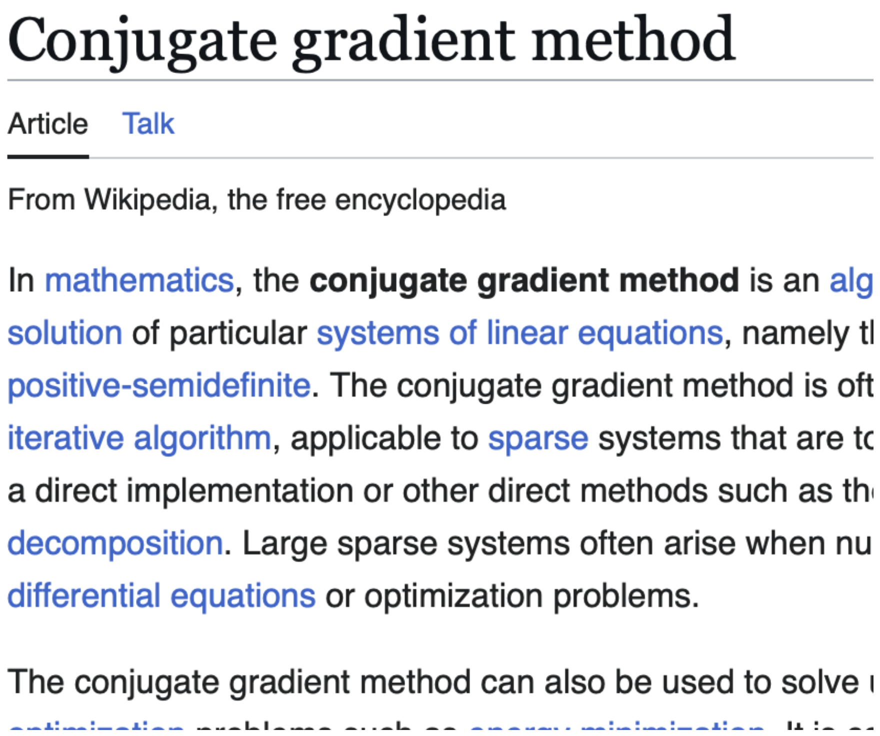
- Standard Gurobi barrier code
- Factorization scales extremely well to 32 cores
  - Most other steps scale fairly well
  - A few notable exceptions
  - How well does it scale on 64+ cores?
- We did some tuning in Gurobi 12.0 barrier to get more out of modern, high-core-count systems

Other Parallel Barrier Options: GPU Factorization

- Use GPU for factorization instead of PDHG
    - Not as massively parallel, but still quite parallel
- Nvidia cuDSS (Direct Sparse Solver) library
    - Very fast for factorization and solves
- Integrated with Gurobi 12.0, but not yet released

Other Parallel Barrier Options: Iterative Solver

- Lots of interest in the past
- Parallelism?
  - Basically the same as PDHG
    - Sparse matrix-vector multiplies
    - Dense vector operations
- Two main problems:
  - Low accuracy solutions
  - Fails to converge on lots of models
  - Sound familiar?
- Gurobi 12.0 barrier includes a parallel iterative solver
  - Triggered automatically
- Nice property
  - Iterates are 'well centered'
  - Can transition smoothly to factorization



Conjugate gradient method

Article Talk

From Wikipedia, the free encyclopedia

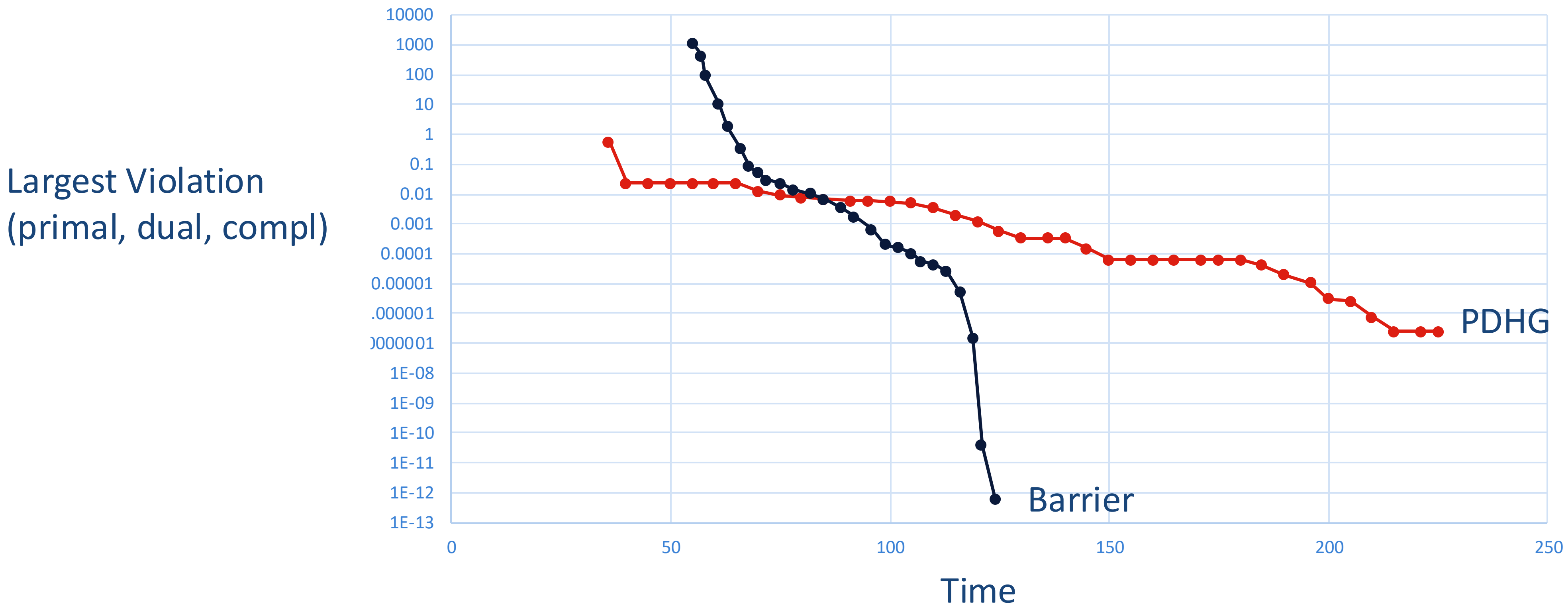
In mathematics, the **conjugate gradient method** is an alg
solution of particular systems of linear equations, namely tl
positive-semidefinite. The conjugate gradient method is oft
iterative algorithm, applicable to sparse systems that are to
a direct implementation or other direct methods such as th
decomposition. Large sparse systems often arise when nu
differential equations or optimization problems.

The conjugate gradient method can also be used to solve

Tolerances and Convergence

Linear vs (Locally) Quadratic Convergence

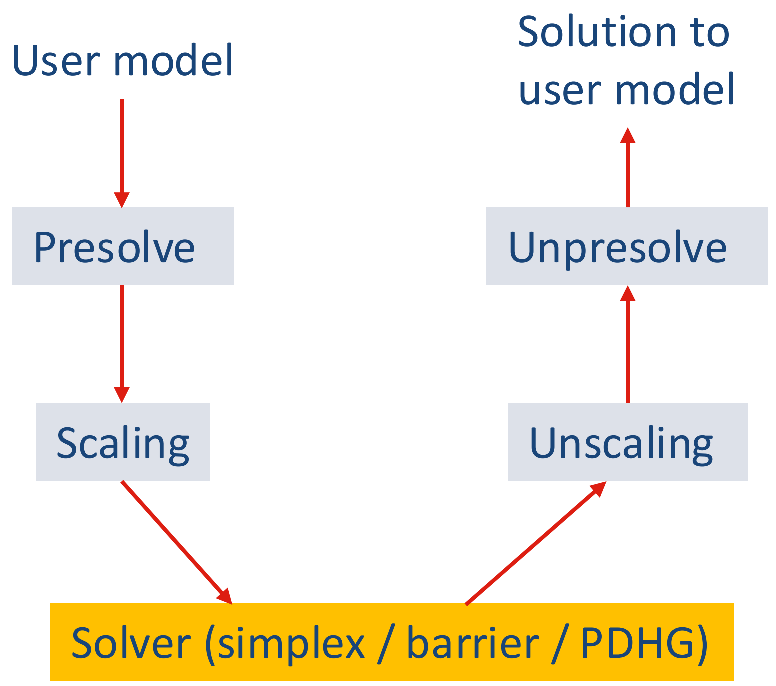
Model *thk_63*



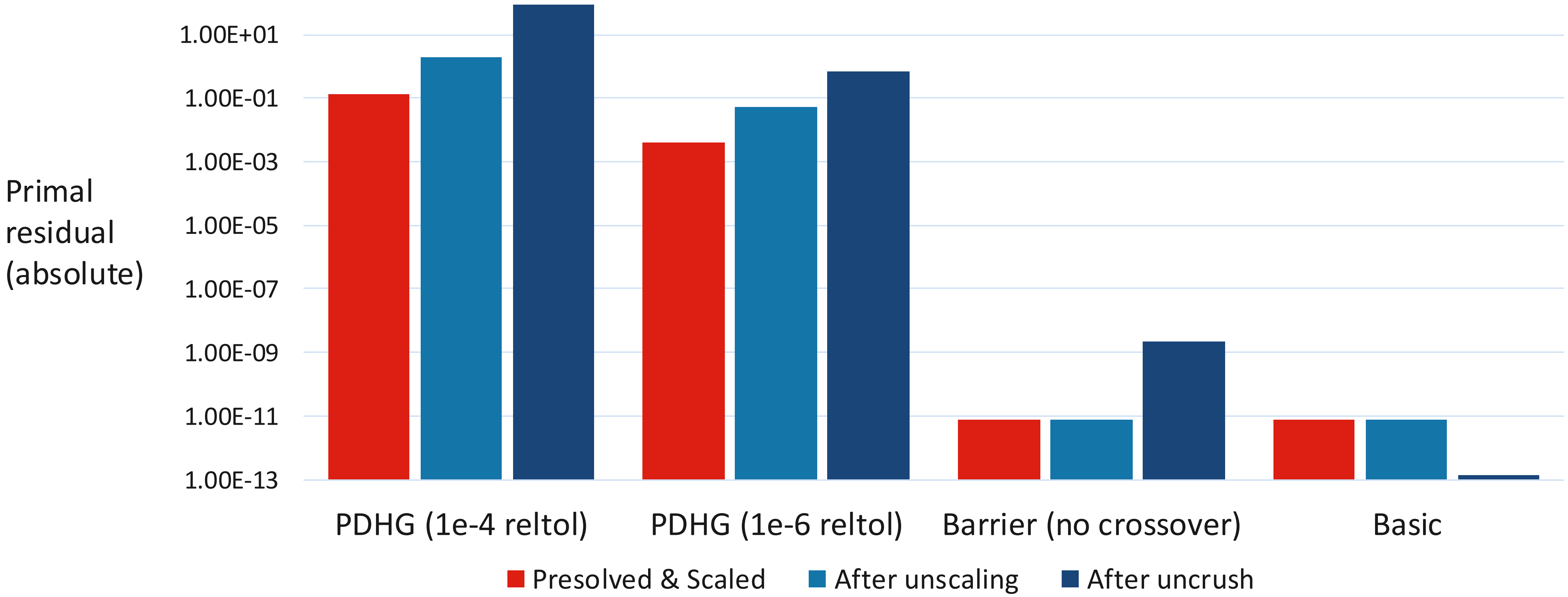
- Practical implication:
  - Termination decision required for first-order method
  - No real decision needed for interior-point method

Accumulated Error

- A few sources of additional error:
  - *Scaling*
    - Essential for convergence
  - *Presolve*
    - Important for reducing model size

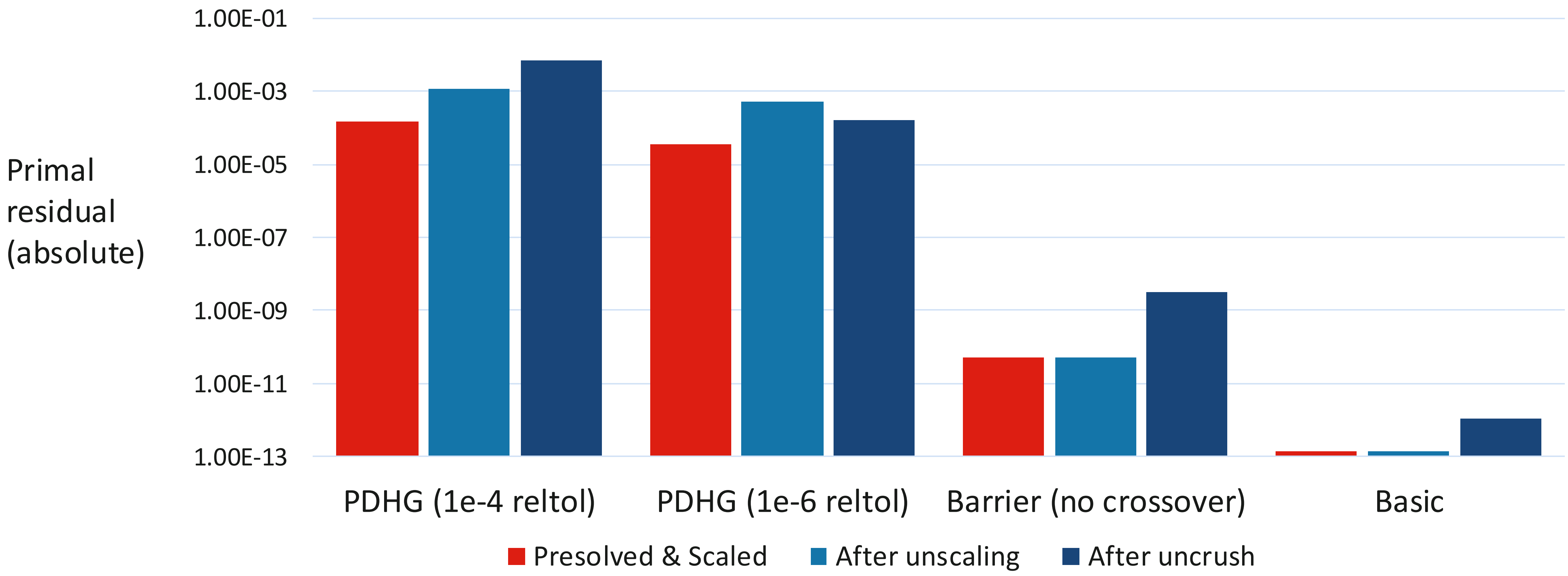


Presolve and Scaling (model *pds-100*)



- Residuals grow substantially

Presolve and Scaling (model *rwth-timetable*)



- Residuals grow substantially

Tolerances

- Termination tolerances
  - Barrier, simplex (absolute): $||b-Ax||_\infty$
  - PDHG (relative): $||b\text{-}Ax||_2 \,/\, ||b||_2$
- What termination criteria are acceptable?
  - Not a lot of experience with this tradeoff
- Intuitive interpretation of relative tolerance…
  - Solution is 'close' to optimal
  - A few digits of accuracy
- Consider:
  - A model with:
    - 1M rows, all '$x_i + x_j <= 1$'
    - One 'capacity' constraint: $y_1 + y_2 <= 10^7$
  - A 1e-4 relative tolerance allows:
    - Solution with '$x_i + x_j = 2$' for every constraint

Example – pilots

- Termination: 1e-4 relative residual

- Constraint:

  c855: 1.406874 x1087 + 1.406874 x1088 + 1.48092 x1140 + 1.48092 x1141+ 1.406874 x1086 + 1.558863 x1195 + 1.558863 x1196 + 1.48092 x1139 + 1.640908 x1256 + 1.640908 x1257 + 1.558863 x1194 + 1.727272 x1321 + 1.727272 x1322 + 1.640908 x1255 + 1.818181 x1386 + 1.818181 x1387 + 1.727272 x1320 - x1431 + 1.818181 x1385 - 2.272727 x1429 <= -1.799672

- Constraint violation: 0.4

- 1e-4 relative residual leads to 22% constraint violation

Example – pds-100 (network flow model)

- Termination: 1e-4 relative residual
- Flow conservation constraint:

  R13193: - C060622 + C060633 + C060666 + C060699 + C060732 + C060765 + C060798 + C060831 + C060864 + C060897 + C060930 + C060963 + C060996 + C061029 = 0

- Flow through node: C060622 = 372
- Constraint violation: 13.5
- 1e-4 relative residual leads to 4% phantom flow through node

Crossover As Equalizer?

- We suspect that most applications require small absolute residuals
- How to reconcile that with PDHG?
- Crossover
  - Routine part of barrier solution process
  - Barrier often produces solutions with non-trivial errors
  - Modern crossover codes can handle this (to some extent)

Performance Results

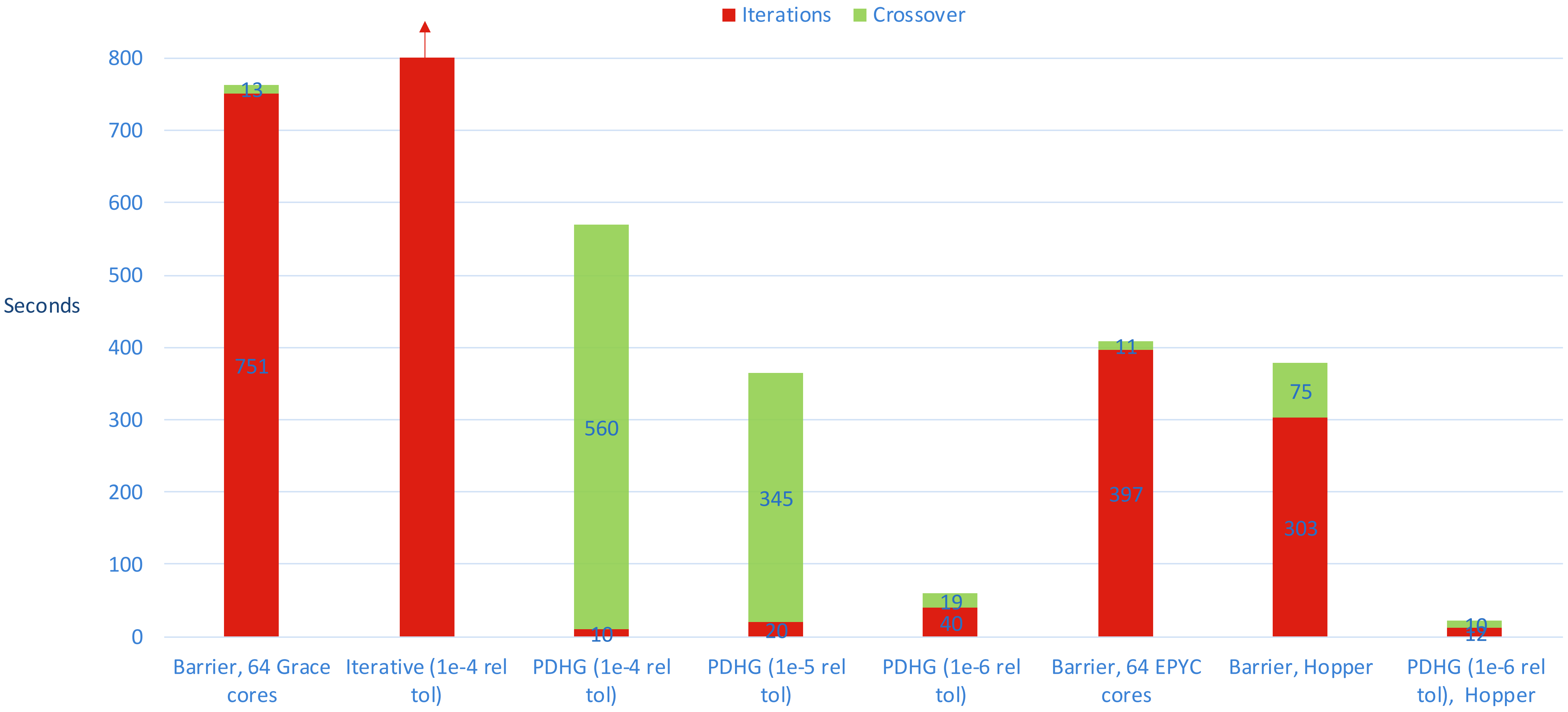
Performance Tests – Algorithms and Machines

- Consider 8 options:
  - Barrier on 72-core Grace CPU
    - Cholesky factorization
    - Iterative solver
  - PDHG on 72-core Grace CPU
    - Relative tolerances of 1e-4, 1e-5, 1e-6
  - Barrier on absolute latest 96-core CPU
    - AMD EPYC 9655
  - Barrier and PDHG (1e-6) on Hopper GPU
    - Cholesky factorization and solves run on GPU
- Crossover always runs on the CPU
- All are measured results from our own implementations
  - Some not yet in a released product
- First five all run on the same hardware

Performance Tests - Models

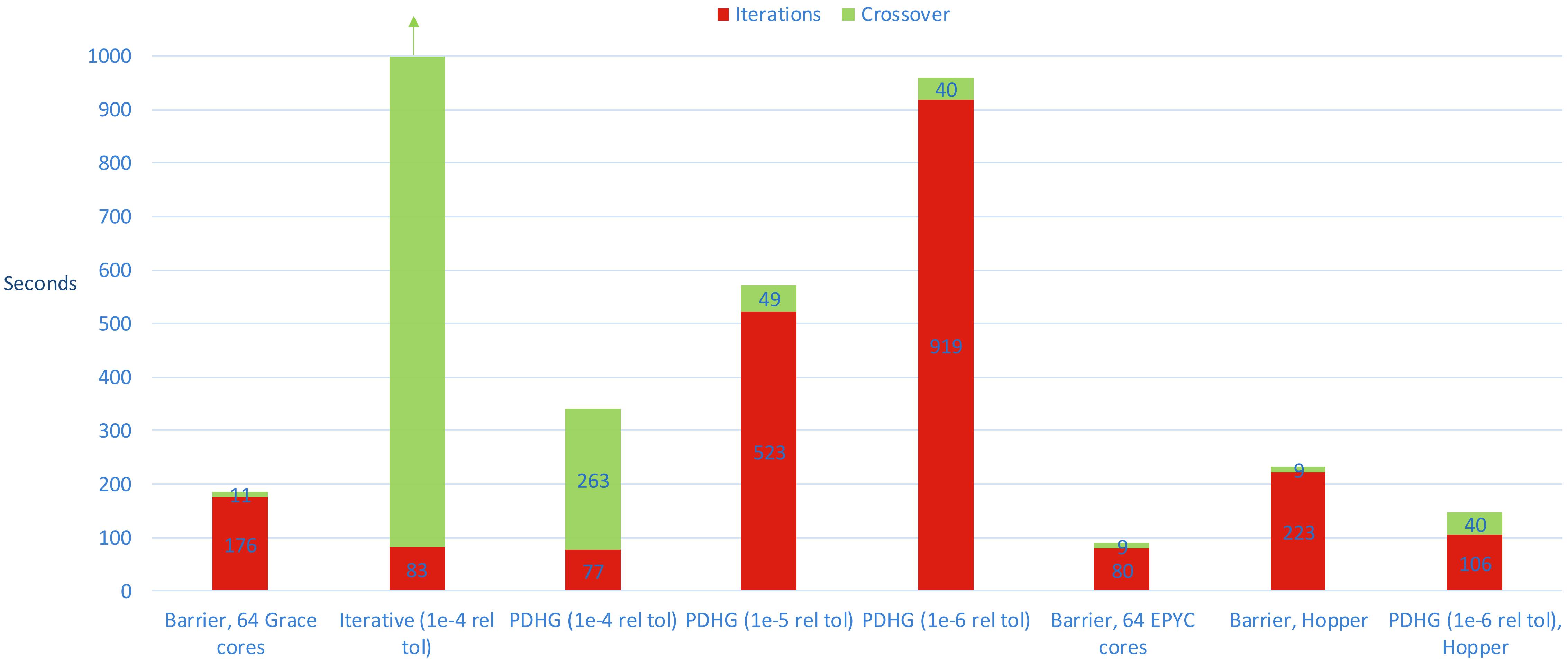
- Choose a set of models where existing methods struggle
  - Goal is to understand how to push the frontier
- Nearly all have been mentioned in papers about PDHG

Performance – model rwth-timetable



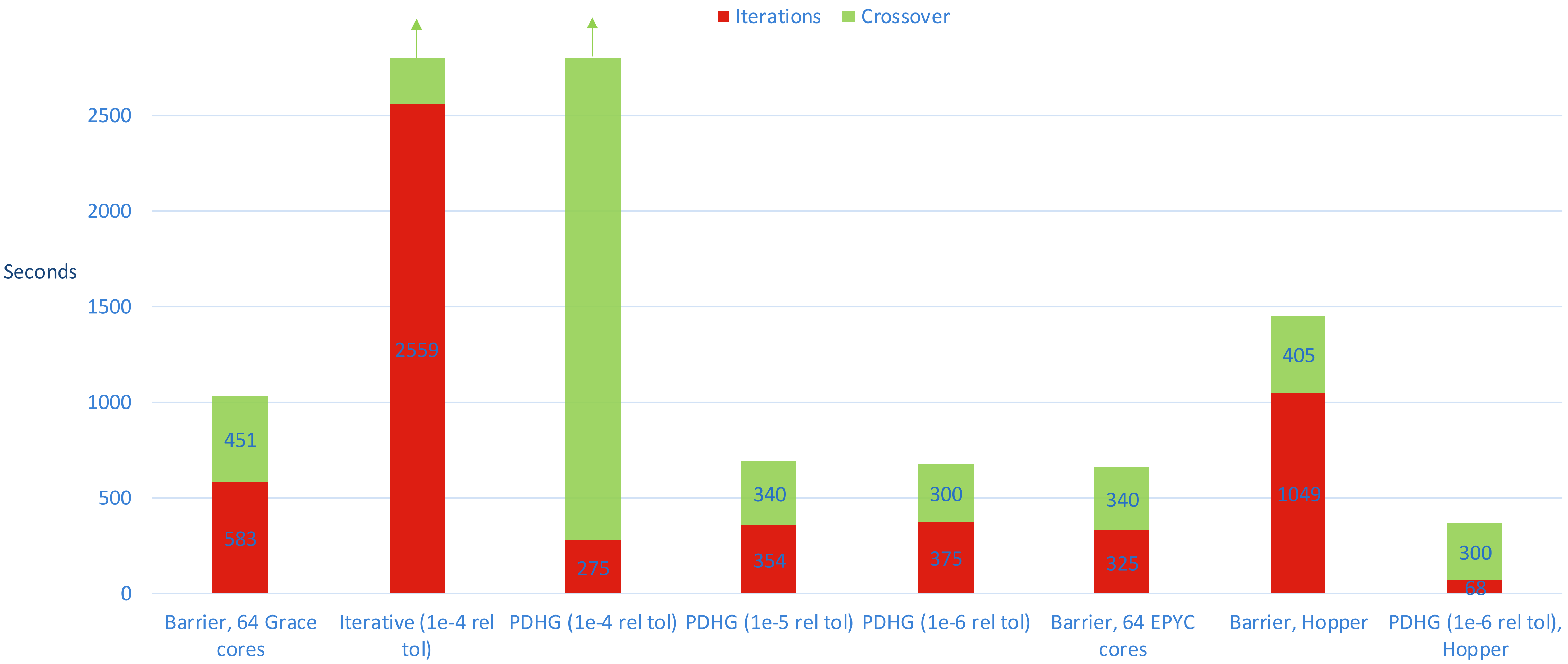
- Barrier benefits from faster CPU/GPU
- Crossover time depends on start point quality

Performance – model grid10



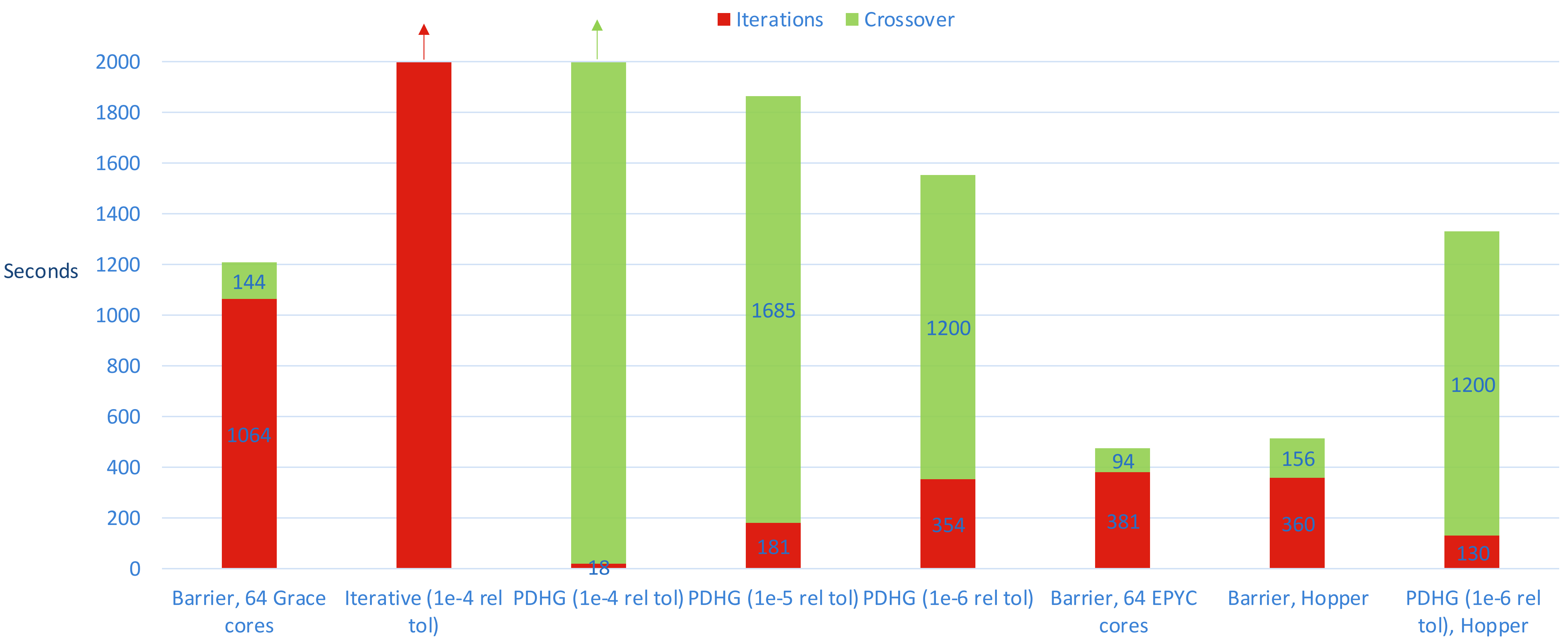
- More accurate start doesn't always pay off
- GPU barrier isn't always faster

Performance – model zib01



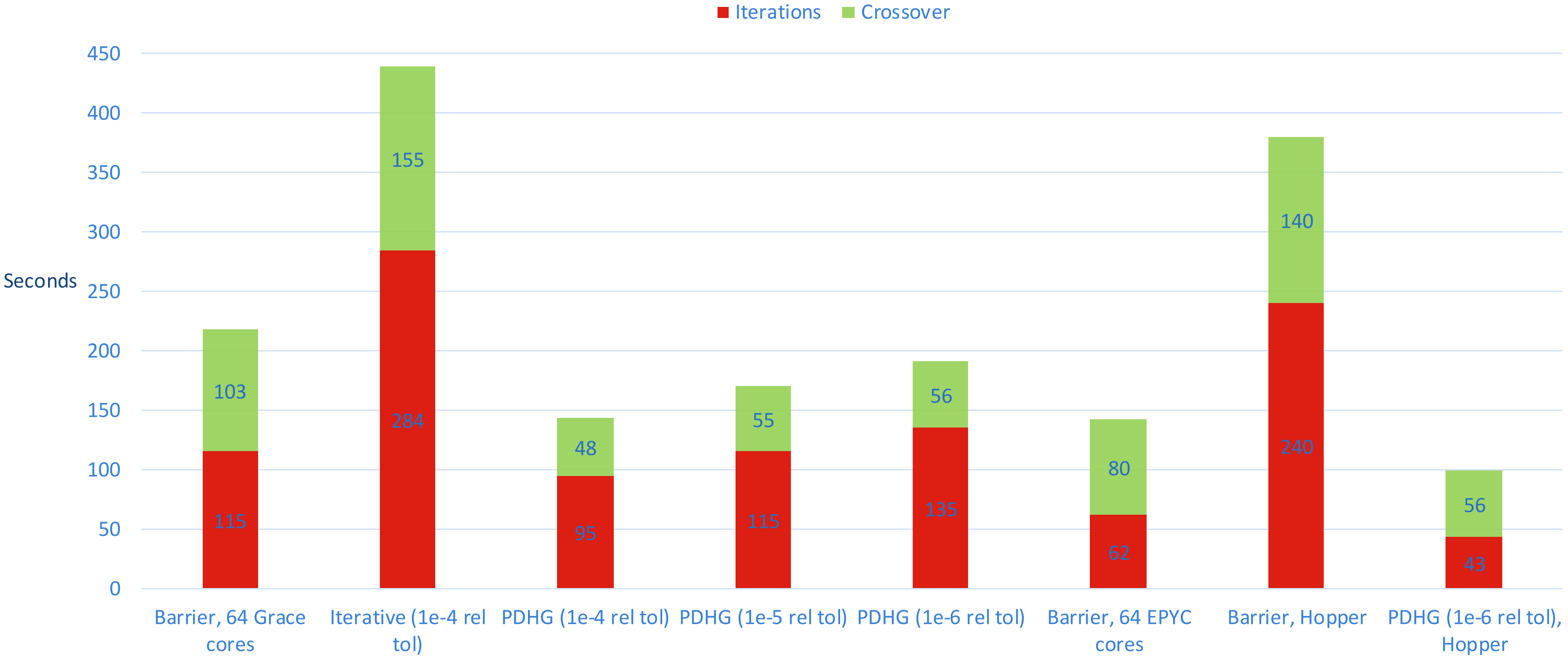
- Crossover from PDHG can be fickle

Performance – model rmine25



- Crossover sometimes requires an extremely accurate start point

Performance – model thk_63



- Sometimes there are lots of good options

Conclusions

Convergence Criteria

- PDHG first LP algorithm to put convergence tolerances front and center
  - Never an issue for simplex or barrier
- What is acceptable accuracy?
  - Before scaling and presolve (typically) degrade it

Crossover

- Effective general-purpose method probably requires crossover to a basic solution
  - Issues:
    - Highly sequential
    - Can be slow when starting point has substantial violations
- Pushes tolerance question even further into the fore
- New algorithms?

A New Horse in the Race

- PDHG…
  - Competitive in a few years
  - That's exciting
    - Even if only on a small set of models
  - Technology is still evolving

- Difficult LPs always benefited from having multiple algorithms available
  - Primal simplex/dual simplex/barrier/concurrent

- Could make new classes of models solvable

GPU

- GPU is a 10X opportunity
  - For both PDHG and barrier
- Doesn’t always wind up on the bottom line
  - Still early

Conclusions

- Several new options for solving giant LP models
- As is typical, each has strengths and weaknesses
  - Probably opportunities for them to cooperate
- Accuracy
  - PDHG: strong speed/accuracy tradeoff
  - Significant opportunities if that tradeoff can be managed

