PaNGEA: Parallel Node Generation and Exploration Algorithm

Jean Pauphilet
London Business School, jpauphilet@london.edu

Yupeng Wu
London Business School, yupengw@london.edu

Abstract. Primal heuristics for finding high-quality feasible solutions are an important component in mixed-integer optimization (MIO) solvers. Recent advances in GPU-accelerated optimization algorithms show the potential of GPU acceleration for continuous optimization. In this paper, we introduce the Parallel Node Generation and Exploration Algorithm (PaNGEA), a GPU-friendly MIO primal heuristic. PaNGEA explores restricted subproblems by combining linear-relaxation solves with a local-search procedure designed for efficient batched execution on GPUs. In addition, instead of relying on a single heuristic for iterative variable fixing, we leverage GPU batching capabilities to generate and explore multiple subproblems in parallel. PaNGEA leverages GPU capabilities in two ways. First, on 283 MIPcc26 and MIPLIB instances, implementing a single-node primal heuristic on GPU reduces the average gap integral by 8–19% relative to its CPU counterpart. Second, generating and exploring multiple nodes in parallel further reduces the average gap integral by 12–19%.

Key words: mixed integer optimization, primal heuristic, GPU acceleration

1. Introduction

Mixed-Integer Optimization (MIO) is a powerful paradigm for modeling and solving real-world decision problems. While NP-hard in the worst case (Wolsey 2008), many large-scale MIO problems can now be solved efficiently, with speed-ups attributable to both better algorithms and hardware (Bixby 2012).

On the algorithm side, efficient solvers combine the implementation of certifiably optimal methods like branch-and-bound with efficient primal heuristics. By finding high-quality solutions early, primal heuristics can help prune the branch-and-bound tree quickly. Conversely, they benefit from being applied on restricted subspaces such as intermediate nodes of the branch-and-bound process (Fischetti and Lodi 2003, Danna et al. 2005).

On the hardware side, current MIO solvers have mostly benefited from the improvement in CPU chips. Over the past 10 years, GPUs have received a lot of attention from the deep learning community (Krizhevsky et al. 2012). Their ability to solve multiple linear systems in parallel creates new opportunities for optimization algorithms. For example, Applegate et al. (2023), Lu and Yang

(2025) propose efficient GPU implementations of first-order methods for linear (continuous) optimization problems. However, exploiting GPU parallelization for MIO primal heuristics remains an open challenge.

In this paper, we develop such a GPU-accelerated primal heuristic. Our algorithm, Parallel Node Generation and Exploration Algorithm (PaNGEA), implements an efficient local-search procedure on restricted subspaces (or nodes). Instead of relying on a single heuristic rule to construct these nodes, we leverage GPU capabilities to generate and evaluate multiple candidate nodes in a batched way.

1.1. Problem Setting

We consider a general mixed-integer linear optimization problem

$$\min_{\boldsymbol{x}\in\mathbb{R}^n} \ \boldsymbol{c}^\top \boldsymbol{x} \quad \text{s.t.} \quad \boldsymbol{A}\boldsymbol{x} \le \boldsymbol{b}, \quad \boldsymbol{\ell}^0 \le \boldsymbol{x} \le \boldsymbol{u}^0, \quad x_j \in \mathbb{Z},\ \forall j \in \mathcal{I}, \tag{$\mathcal{P}$}$$

where $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, $\boldsymbol{b} \in \mathbb{R}^m$, $\boldsymbol{\ell}^0, \boldsymbol{u}^0 \in \mathbb{R}^n$, and $\mathcal{I} \subseteq [n] := \{1, \dots, n\}$ is the index set of integer variables. It will be convenient to consider a restricted version of Problem ($\mathcal{P}$), also called a subproblem or node.

DEFINITION 1 (NODE). For any pair of vectors $\boldsymbol{\ell}, \boldsymbol{u} \in \mathbb{R}^n$ satisfying $\boldsymbol{\ell}^0 \le \boldsymbol{\ell} \le \boldsymbol{u} \le \boldsymbol{u}^0$, we call a *node*, and denote $\mathcal{P}(\boldsymbol{\ell}, \boldsymbol{u})$, the optimization problem ($\mathcal{P}$) with the additional constraints $\boldsymbol{\ell} \le \boldsymbol{x} \le \boldsymbol{u}$. For concision, we denote by $[\boldsymbol{\ell}, \boldsymbol{u}]$ the set of vectors $\boldsymbol{x}$ such that $\boldsymbol{\ell} \le \boldsymbol{x} \le \boldsymbol{u}$. By definition, we have $\mathcal{P} = \mathcal{P}(\boldsymbol{\ell}^0, \boldsymbol{u}^0)$, which is commonly referred to as the root node. Further, if $\ell_j = u_j$ for some $j \in [n]$, then $\mathcal{P}(\boldsymbol{\ell}, \boldsymbol{u})$ fixes coordinate j of $\boldsymbol{x}$ to the common value $\ell_j = u_j$. We call the (linear) *relaxation* of $\mathcal{P}(\boldsymbol{\ell}, \boldsymbol{u})$ the optimization problem obtained by dropping the integrality constraints.

1.2. Related Work

The current algorithmic framework of MIO solvers involves multiple components, including continuous relaxations (lower bounds), branching, node selection, and feasible solution heuristics (upper bounds).

With the GPU acceleration of continuous optimization algorithms, such as the primal-dual linear programming (PDLP) method of Lu and Yang (2025), a natural direction is to use GPUs for solving the continuous relaxations of MIO problems faster. For instance, promising results have been obtained by using GPU-accelerated versions of PDLP (Blin et al. 2026, De Rosa et al. 2026), the alternating direction method of multipliers (Meng et al. 2026), or proximal gradient algorithms (Liu and Lodi 2026, Liu et al. 2026).

Several works investigate the deeper integration of GPU acceleration within a branch-and-bound scheme. Blin et al. (2026), Meng et al. (2026), Liu et al. (2026) consider solving similar relaxations (corresponding to different active nodes in the branch-and-bound tree) in a batched way. Liu and Lodi (2026) transfer additional node processing steps to the GPU, including rounding heuristics and branching selection. Beyond MIO, Zhang et al. (2025) leverage GPU parallelization in a spatial branch-and-bound algorithm for global (non-convex) optimization.

Another central component is finding high-quality feasible solutions, which typically relies on primal heuristics. A major idea in primal heuristics is to search for a high-quality solution within a restricted subspace or neighborhood. Classic examples include local branching (Fischetti and Lodi 2003) or relaxation-induced neighborhood search (Danna et al. 2005). However, these techniques rely on solving a small MIO problem. Several alternatives have been proposed to circumvent the need for an MIO solver: diving (Guzelsoy et al. 2013) and fix-and-propagate (Salvagnin et al. 2025) conduct a depth-first search on the branch-and-bound tree to fix variables iteratively; feasibility pump (Fischetti et al. 2005) iterates between solving relaxations and projection; move-based approaches, such as feasibility jump (Luteberget and Sartor 2023) and Local-MIP (Lin et al. 2025), make one-coordinate changes on the decision variables to greedily construct better solutions. These heuristics are potentially easier to accelerate on GPUs because they rely on solving linear relaxations or on performing a large number of simple algebraic operations. Successful examples include the GPU acceleration of fix-and-propagate (Kempke and Koch 2025), local search (Çördük et al. 2025, Tjusila et al. 2026), rounding, and bound propagation (Çördük et al. 2025).

It is worth mentioning that GPU acceleration of several primal heuristics has been discussed for the cases of specific problem classes such as vehicle routing (Schulz 2013, Abdelatti and Sodhi 2020), scheduling (Czapiński and Barnes 2011), or pure-binary optimization problems (Abbas and Swoboda 2022, Wei and Liang 2025). We refer to Neira et al. (2024, section 2) for a broader survey.

Our work contributes to the literature on general-purpose GPU-accelerated primal heuristics for mixed-integer linear optimization. As in Çördük et al. (2025) and Tjusila et al. (2026), which are the closest works to ours, our algorithm relies on an efficient GPU implementation of the Local-MIP procedure (Lin et al. 2025). Our algorithm has two distinctive features. First, instead of maintaining one search region, we leverage the batching capabilities of GPUs to conduct several local searches in parallel on different nodes. Second, we develop a two-stage implementation of Local-MIP that is more efficient at finding high-quality solutions on a restricted subproblem (in particular, when some of the coordinates are fixed). So far, we develop PaNGEA as a standalone

primal heuristic, which does not rely on any MIO solver. Future work, however, could investigate the integration of our parallel node generation and exploration within a non-sequential branch-and-bound algorithm such as Liu and Lodi (2026).

1.3. Contributions

Our contributions can be summarized as follows.

First, we develop an efficient GPU-friendly heuristic for exploring a given subproblem (or node), based on solving its linear relaxation and then applying a Local-MIP procedure. Instead of applying Local-MIP to the restricted subproblem directly, we propose a two-stage implementation, i.e., we impose the additional node bounds only after obtaining a globally feasible solution. We find that our two-stage implementation is significantly more effective at finding feasible and high-quality solutions than the naive implementation of Local-MIP (reduces the average optimality gap by 30%), and that warm-starting with the linear relaxation reduces the optimality gap by a factor of five.

Second, we use GPU batching not only to accelerate node exploration, but to diversify how nodes are generated: instead of relying on a single node-generation heuristic like traditional solvers, PaNGEA evaluates multiple strategies in parallel and retains the most promising nodes at each iteration. GPU batching not only provides acceleration of traditional methods (our single-node GPU implementation of Local-MIP reduces the gap integral by 8% and 18% on the MIPcc26 and MIPLIB instances, respectively) but also enables parallel exploration of multiple node-generation rules (which reduces the gap integral further by 19% and 12%, respectively).

The rest of the paper is organized as follows: We present our overall algorithm and its different components in Section 2, and evaluate it numerically in Section 3.

2. Algorithm

Our algorithm relies on a new efficient strategy for conducting local search over a subproblem (Section 2.1). In particular, this strategy can be effectively conducted on GPUs in a batched way. We leverage this design in our main algorithm (Section 2.2), which iteratively generates and explores different subproblems in parallel (Section 2.3) and retains the most promising ones at each iteration (Section 2.4). We discuss the details of our GPU implementation and batching in Section 2.5.

2.1. Core Subroutine

At the core of our algorithm is a two-stage implementation of the local-search procedure of Lin et al. (2025), tailored to improve its performance when restricted to subproblems. Our improvements rely on two ingredients. First, we initialize the search (including the one at the root node) at

the solution of the continuous (linear) node relaxation, instead of an all-zero vector as in Lin et al. (2025). Second, we delay imposing the node bounds $\boldsymbol{\ell} \leq \boldsymbol{x} \leq \boldsymbol{u}$ until after a feasible solution has been found.

For problem $\mathcal{P}(\boldsymbol{\ell}^0, \boldsymbol{u}^0)$, Local-MIP (Lin et al. 2025) starts from an initial point $\boldsymbol{x}^0$ (an all-zero vector in their implementation), rounds the integer coordinates of $\boldsymbol{x}^0$ to the nearest integer, projects it onto the box $[\boldsymbol{\ell}^0, \boldsymbol{u}^0]$, and iteratively applies one-coordinate changes of the form $x_j^{t+1} = x_j^t + \delta_j^t$. Lin et al. (2025) describe several rules to construct admissible updates δ_j^t as well as a scoring system used to pick which coordinate $j \in [n]$ to update. Typically, the scoring system favors moves that reduce constraint violation (when the incumbent is infeasible) or improve the objective value. Variable bounds are treated differently than other constraints: the step sizes δ_j^t are restricted to $[\ell_j^0 - x_j^t, u_j^0 - x_j^t]$. Because the initial point after rounding and projection belongs to $[\boldsymbol{\ell}^0, \boldsymbol{u}^0]$, we always have $\boldsymbol{x}^t \in [\boldsymbol{\ell}^0, \boldsymbol{u}^0]$.

For a given node $\mathcal{P}(\boldsymbol{\ell}, \boldsymbol{u})$, the naive way of doing local search is to apply the procedure described above with the tighter bounds $(\boldsymbol{\ell}, \boldsymbol{u})$. Because local search makes myopic one-coordinate changes, it sometimes needs to break feasibility and violate constraints in order to find feasible or better solutions in the end. If so, restricting the variable bounds (and, in many cases, fixing some of the coordinates), hence limiting the range of admissible step sizes, may negatively impact performance. Instead, we make two algorithmic changes to *incentivize* solutions within $[\boldsymbol{\ell}, \boldsymbol{u}]$.

First, we run a local-search procedure on the entire space, i.e., for $\mathcal{P}(\boldsymbol{\ell}^0, \boldsymbol{u}^0)$, but initialized with the solution of the linear relaxation of $\mathcal{P}(\boldsymbol{\ell}, \boldsymbol{u})$. In this case, $\boldsymbol{x}^0 \in [\boldsymbol{\ell}, \boldsymbol{u}] \subseteq [\boldsymbol{\ell}^0, \boldsymbol{u}^0]$. At each iteration, the admissible step sizes ensure that iterates remain in $[\boldsymbol{\ell}^0, \boldsymbol{u}^0]$ but they can leave the subregion $[\boldsymbol{\ell}, \boldsymbol{u}]$. Second, once a feasible solution to $(\mathcal{P})$ is found, we impose the node variable bounds $(\boldsymbol{\ell}, \boldsymbol{u})$ by restricting the step sizes to $\delta_j^t \in [\ell_j - x_j^t, u_j - x_j^t]$. This implementation does not immediately project the next iterates onto $[\boldsymbol{\ell}, \boldsymbol{u}]$ but rather incentivizes convergence towards a solution that belongs to the node.

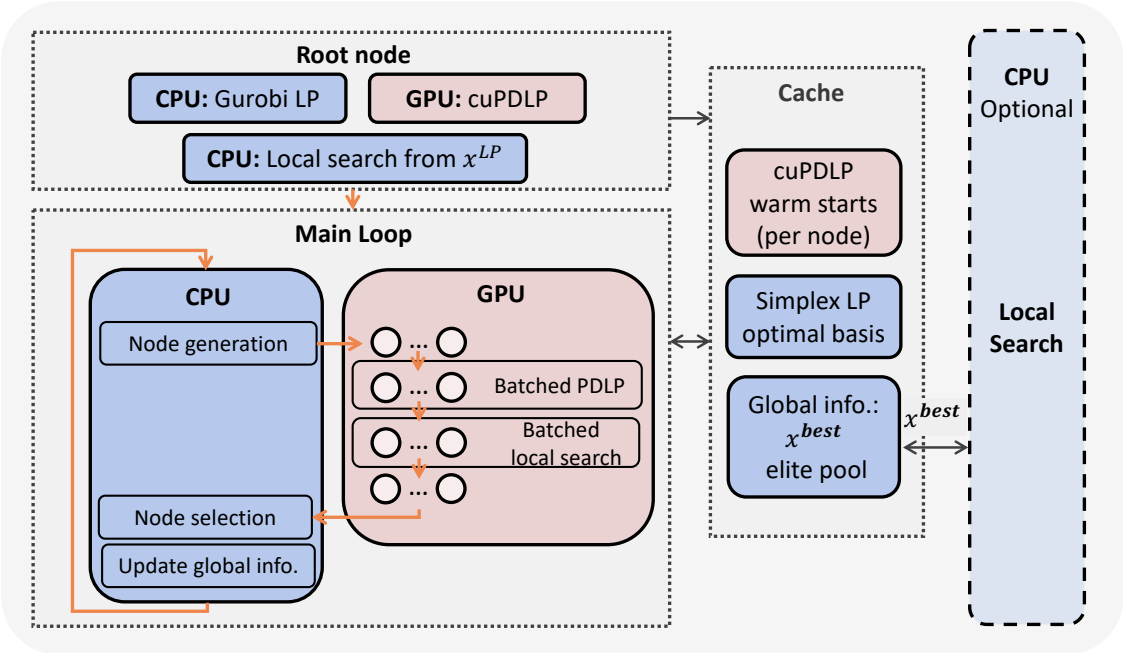
2.2. Overview of PaNGEA

We briefly describe our algorithm in this section, which is visually summarized in Figure 1. We provide additional implementation details in Appendix A.

First, we apply our core subroutine (linear relaxation followed by local search) from Section 2.1 at the root node. We duplicate the root node into W nodes, $\mathcal{N}_1, \ldots, \mathcal{N}_W$.

The main part of our algorithm (the main loop) then follows. From each node $\mathcal{N}_w, w \in [W]$ we construct R new nodes, $\mathcal{N}_{w,1}, \ldots, \mathcal{N}_{w,R}$ by applying different node-generation strategies (see

Figure 1 **Visual representation of the architecture of PaNGEA.**



Section 2.3). We leverage the batching capabilities of GPUs to apply the core subroutine to all $W \times R$ nodes $\{\mathcal{N}_{w,r}\}$ in parallel. We comment on the GPU implementation of the subroutine in Section 2.5. We then retain only the best W nodes and proceed to the next iteration (Section 2.4).

We keep in memory (on the CPU) the best solution found $\boldsymbol{x}^{\texttt{best}}$, its objective value `bestUB`, and a pool of high-quality solutions $\mathcal{E}$ ($|\mathcal{E}| = 20$ in our implementation). In addition, we use a separate CPU thread for local search. Every time the best incumbent $\boldsymbol{x}^{\texttt{best}}$ is updated by the main loop, we restart the local search from $\boldsymbol{x}^{\texttt{best}}$ on this CPU thread. The separate CPU thread serves as a more focused local search while broader exploration of the feasible space is performed on the GPU. Such a hybrid CPU–GPU implementation is common in recent developments of (discrete) optimization algorithms, to leverage the strengths of each hardware platform and provide more robust performance (see, e.g., Tjusila et al. 2026).

2.3. Node-Generation Strategies

We now describe six strategies for generating alternative nodes to explore. Unlike traditional CPU-based approaches, our algorithm does not rely on picking one strategy but instead applies all of them in parallel at each iteration.

The first two strategies modify the bounds of a parent node $\mathcal{P}(\boldsymbol{\ell}, \boldsymbol{u})$, for which we have access to primal and dual solutions of the linear relaxation, $\boldsymbol{x}^{\text{LP}}$ and $\boldsymbol{y}^{\text{LP}}$, as well as a solution from the local-search procedure, $\boldsymbol{x}^{\text{LS}}$.

Reduced costs. From the linear relaxation, we can compute the reduced costs $\boldsymbol{r} = \boldsymbol{c} - \boldsymbol{A}^\top \boldsymbol{y}^{\texttt{LP}}$. Our local search starts from $\boldsymbol{x}^{\texttt{LP}}$ and returns $\boldsymbol{x}^{\texttt{LS}}$. We can thus assign a cost $s_j = r_j\big(x_j^{\texttt{LS}} - x_j^{\texttt{LP}}\big)$ to each coordinate change. Let $\mathcal{J} \subseteq \mathcal{I}$ denote the indices with the lowest cost s_j (typically, we fix $|\mathcal{J}|/|\mathcal{I}| = 4\%$ in our experiments). These correspond to the local-search moves with the lowest compromise in objective value. For each $j \in \mathcal{J}$, if $\ell_j < u_j$ (unfixed coordinate), we set $\ell_j = u_j = x_j^{\texttt{LS}}$ (we fix it), otherwise we set $\ell_j = \ell_j^0$ and $u_j = u_j^0$. This fixing/unfixing strategy is similar in spirit to restrict-and-relax search (Guzelsoy et al. 2013).

Diving. Starting from the current bounds $(\boldsymbol{\ell}, \boldsymbol{u})$, we randomly select $j \in \mathcal{I}$ such that $\ell_j < u_j$. If $x_j^{\texttt{LP}} \in \mathbb{Z}$, we set $\ell_j = u_j = x_j^{\texttt{LP}}$. Otherwise, we solve two linear relaxations, with the additional constraint $x_j = \lfloor x_j^{\texttt{LP}} \rfloor$ or $x_j = \lceil x_j^{\texttt{LP}} \rceil$, respectively, and fix $\ell_j = u_j$ to the value with the tighter (i.e., larger) relaxation bound. We repeat this process until a target number of variables has been fixed.

The next two strategies rely on the solution pool maintained on the CPU, $\mathcal{E} = \{\boldsymbol{x}^{(e)}, e = 1, \dots, E\}$, and are described for the subset of indices $\mathcal{I}_b \subseteq \mathcal{I}$ that corresponds to binary variables only.

Consensus restart. Starting from $(\boldsymbol{\ell}, \boldsymbol{u}) = (\boldsymbol{\ell}^0, \boldsymbol{u}^0)$ we fix $\ell_j = u_j$ to 0 (resp. 1) if at least 80% of the solutions in the pool $\mathcal{E}$ satisfy $x_j^{(e)} = 0$ (resp. 1).

Elite. Starting from the binary coordinates that are currently fixed, the Elite strategy first unfixes and then fixes some of these coordinates. These decisions are made by following the most common choice in the elite pool. We refer to Appendix A for a more formal description of the criteria used.

We consider one strategy aimed at exploring the neighborhood of $\boldsymbol{x}^{\texttt{best}}$.

Best. Starting from $(\boldsymbol{\ell}, \boldsymbol{u}) = (\boldsymbol{\ell}^0, \boldsymbol{u}^0)$, for each $j \in \mathcal{I}$, we set $\ell_j = u_j = x_j^{\texttt{best}}$ with probability 90%, independently.

Finally, we consider one strategy with bound propagation to address instances where finding feasible solutions is challenging.

Deep diving with bound propagation. Starting from $(\boldsymbol{\ell}, \boldsymbol{u}) = (\boldsymbol{\ell}^0, \boldsymbol{u}^0)$, we sample $j \in \mathcal{I}_b$, set $\ell_j = u_j = 1$, and apply bound propagation. We repeat this process until all binary variables are fixed or a conflict in bound propagation is raised.

2.4. Node Selection

After exploring all nodes $\{\mathcal{N}_{w,r}\}$ in one iteration of the main loop, we retain the 'best' W nodes for the next iteration. Our selection is based on the following criteria:

- We discard suboptimal nodes, i.e., such that $\boldsymbol{c}^\top \boldsymbol{x}^{\texttt{LP}} > \texttt{bestUB}$ when $\boldsymbol{x}^{\texttt{LP}}$ is optimal.

- We sort nodes by increasing objective value, $\boldsymbol{c}^\top \boldsymbol{x}^{\mathrm{LS}}$, and select the best W nodes.

If a node relaxation is infeasible ('$\boldsymbol{c}^\top \boldsymbol{x}^{\mathrm{LP}} = +\infty$'), the first criterion discards the node unless no feasible incumbent has been found yet (i.e., $\texttt{bestUB} = +\infty$). Furthermore, if this criterion leaves fewer than W nodes to choose from, we randomly decide to keep some of the discarded nodes and unfix 10% of the coordinates they currently fix.

For the second criterion, the top-W nodes may include nodes for which the local search failed to find any feasible solution ('$\boldsymbol{c}^\top \boldsymbol{x}^{\mathrm{LS}} = +\infty$'). In these cases, we order nodes by increasing number of violated constraints, $\sum_{j\in[m]} \mathbf{1}\left(\boldsymbol{a}_j^\top \boldsymbol{x}^{\mathrm{LS}} - b_j > 0\right)$.

2.5. GPU Implementation and Batching

A central design of our algorithm is that each step of our core subroutine (node relaxation and local search) can be implemented on GPUs and that nodes only differ in their bounds. Therefore, we process multiple nodes in a batched fashion and benefit from GPU acceleration. The shared static data is stored on the GPU once at the beginning of the algorithm, while node-specific data is stacked across the batch dimension and updated in parallel.

For solving the node relaxation, we use cuPDLP (Lu and Yang 2025), a GPU implementation of a primal-dual hybrid gradient method for linear programming (PDLP; see Applegate et al. 2023). PDLP solves the corresponding primal-dual saddle-point problem with first-order updates. The dominant computations are sparse matrix-vector products $\boldsymbol{A}\boldsymbol{x}$ and coordinate-wise projections, both of which are GPU-friendly. To process $B = W \times R$ nodes in parallel, we store the iterates $\boldsymbol{x}$ and node bounds $(\boldsymbol{\ell}, \boldsymbol{u})$ as three matrices in $\mathbb{R}^{n\times B}$. With this layout, each PDLP update is carried out for all nodes in parallel, using sparse matrix-matrix products and coordinate-wise projections. The batched formulation preserves the regular structure of PDLP while spreading the cost of GPU kernel launches across multiple nodes and further improving kernel throughput.

In the local-search algorithm, the dominant computations occur in generating and evaluating the candidate moves, δ_j^t, which are all independent and thus parallelizable. However, the GPU implementation is not straightforward, as there are multiple possible mappings between threads and computations, which may affect load balance. For example, a move (j, δ_j) may be computationally heavy to evaluate if the jth column of $\boldsymbol{A}$ is dense (i.e., x_j is involved in many constraints). In our implementation, we map a warp to 32 nonzeros in $A_{\cdot j}$ for any j. We do not see a significant speedup from using CUDA graphs; see Çördük et al. (2025) for a discussion on a CUDA-graph implementation of Local-MIP. As for solving the relaxations, all nodes in a batch share the same problem data, while node-specific quantities such as variable bounds and the search state are stored

column-wise across the batch. The searches proceed independently for every node but use common GPU kernels to generate or evaluate candidate moves.

3. Numerical Results

In this section, we first validate the effectiveness of our node exploration subroutine (Section 3.1). We demonstrate the benefit of using multiple ($R > 1$) node-generation strategies concurrently in Section 3.2 and the benefit of exploring multiple nodes in parallel ($W > 1$) in Section 3.3. We then benchmark our method against alternatives in Section 3.4.

Throughout this section, we use the MIPcc26[1] testbed and the MIPLIB2017 (Gleixner et al. 2021) benchmarking set. We measure performance in terms of final (primal) gap at termination (which measures the quality of the best solution found) and (primal) gap integral (which captures the time/quality trade-off) as defined in Berthold (2013). The 'gap' is relative to a best-known objective, which in our case is the best objective found across all algorithms in the comparison as well as the 'best solution' provided with the MIPcc26 and MIPLIB instances. We use an NVIDIA H100 GPU for final benchmarking (Section 3.4) and lower-end GPUs in other experiments. The time limit is 5 minutes.

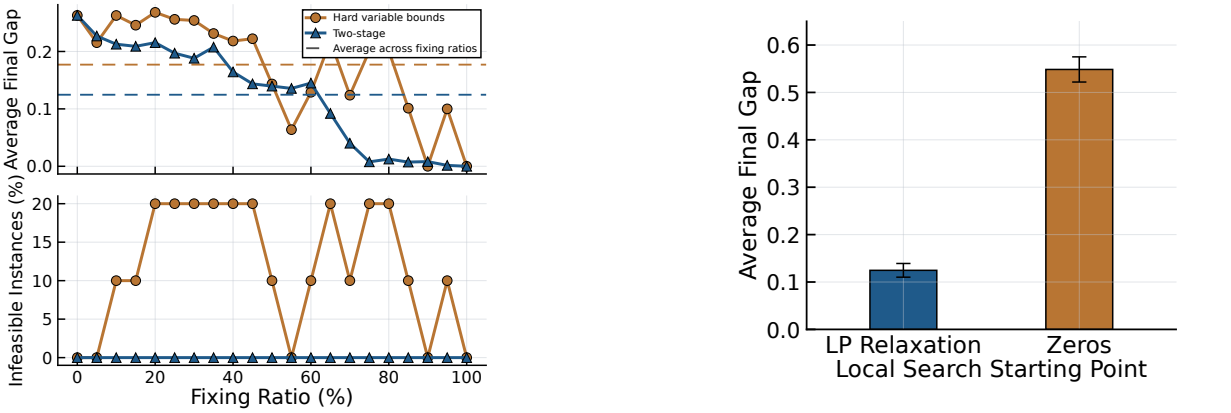
3.1. Local-MIP with Node Pressure

We first demonstrate the effectiveness of our local-search subroutine (Section 2.1) on a subset of the MIPcc26 instances, namely instances $\{1, 12, 14, 15, 24, 31, 32, 42, 43, 50\}$. On each instance, we find a (near)-optimal solution using Gurobi and construct 21 subproblems by fixing some of the integral coordinates to their values in that solution.

Figure 2(a) compares our two-stage procedure with the direct application of local search on the subproblem $\mathcal{P}(\boldsymbol{\ell}, \boldsymbol{u})$ ('hard variable bounds'). Both methods are initialized with the solution of the node relaxation. In terms of average gap (top panel), both methods improve as we fix more coordinates (as expected). For a fixed fraction of fixed coordinates, however, our two-stage algorithm achieves significantly smaller gaps, with a reduction of five percentage points on average across different fractions of fixed coordinates. This improvement is partly due to the fact that the naive implementation of local search often fails even to find a feasible solution to ($\mathcal{P}$) (bottom panel), even when a large fraction (70–90%) of the coordinates are fixed, unlike our two-stage implementation.

We evaluate the effectiveness of our initialization strategy in Figure 2(b). Starting from the node-relaxation solution in our two-stage procedure leads to a substantially smaller average gap (12%)

Figure 2 **Performance comparison between our two-stage local search and a naive implementation of Lin et al. (2025) on $\mathcal{P}(\boldsymbol{\ell}, \boldsymbol{u})$.**



(a) (b)

Note. Panel (a) reports the average gap (top) and fraction of subproblems where the search fails to find a globally feasible solution (bottom), for both methods, averaged over instances with the same ratio of fixed integer coordinates. Panel (b) reports the average gap achieved by our two-stage local search when initialized with the node relaxation solution versus a vector of all zeros.

than starting from zeros (55%). While Lin et al. (2025) describe their method as being 'LP-free', our results confirm that they can benefit from being initialized with the linear relaxation.

3.2. Diversifying Node-Generation Strategies

We evaluate the benefit of considering multiple node-generation strategies in parallel using the MIPcc26 instances. We fix $W = 1$ and compare two variants: $R = 1$ with the reduced-cost strategy and $R = 6$ with up to all six node-generation strategies.

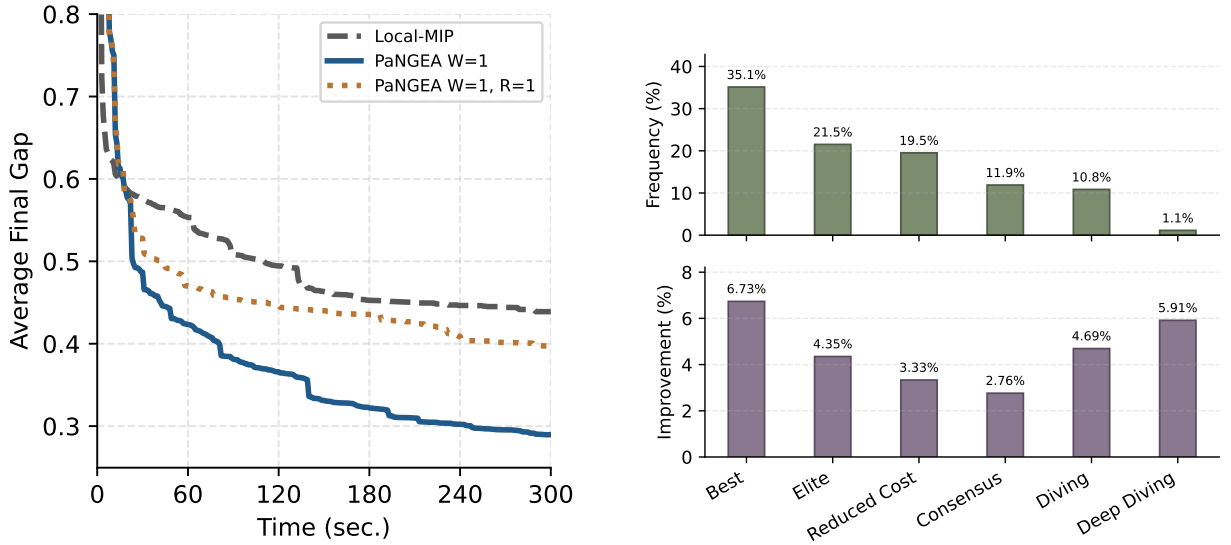
Figure 3(a) shows that parallel node exploration improves the global gap vs. time convergence profile. Our algorithm with $R = 6$ achieves a final gap that is 11 percentage points lower than the same algorithm with $R = 1$.

To further understand these gains, whenever the best incumbent solution $\boldsymbol{x}^{\texttt{best}}$ is updated, we record which node-generation strategy generated this incumbent (which we refer to as a 'success') and by how much `bestUB` is improved. Figure 3(b) reports the frequency and objective improvement associated with each rule. We observe that some strategies, such as Deep Diving, are rarely beneficial (1.1% of the time) but provide large improvements (5.91%) when they do. Others, such as Best or Elite, frequently generate new best incumbents (35.1% and 21.5% of the time, respectively) and yield sizeable improvements (6.73% and 4.35%, respectively).

3.3. Parallel Exploration of Multiple Nodes

The number of nodes explored in parallel, $B = W \times R$, is a key parameter in PaNGEA. Increasing W (hence, B) increases the amount of parallel work and typically improves GPU utilization. How-

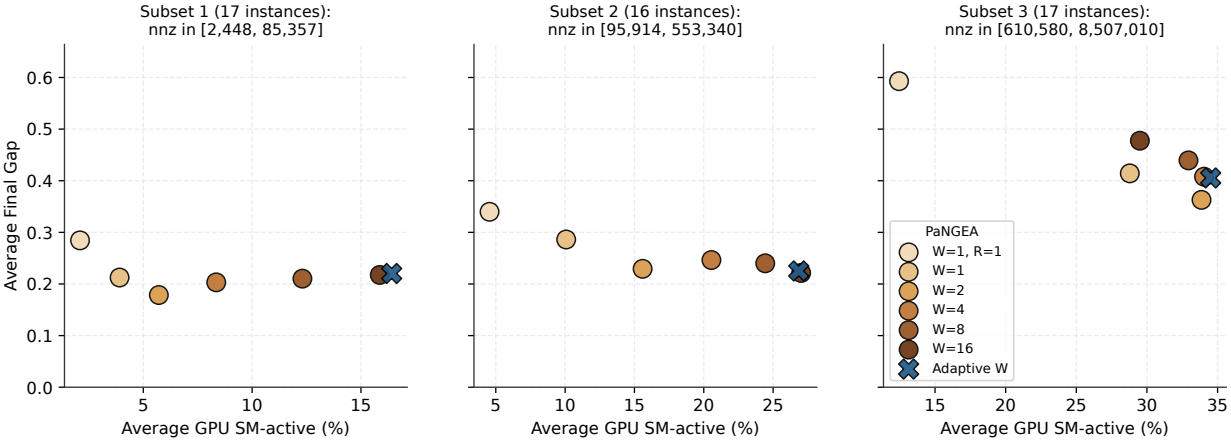
Figure 3 Computational impact of generating and exploring multiple nodes in parallel.



(a) Gap vs. elapsed time. (b) Profile of node-generation strategies.

Note. Panel (a) reports the gap vs. time profile for Local-MIP (Lin et al. 2025), PaNGEA with $W = 1, R = 1$ (reduced cost strategy), and PaNGEA with $W = 1$ and all strategies. Results are averaged over all MIPcc26 instances. Panel (b) reports the profile of node-generation strategies for PaNGEA: the frequency at which a strategy finds the new best solution (top) and the relative improvement in `bestUB` (bottom).

Figure 4 Final gap vs. GPU SM-active rate with different batch sizes in three instance-size subsets.



Note. Dynamically adapting the number of nodes retained at each iteration, W, effectively increases GPU utilization and heuristic performance. Results are averaged within three subsets of MIPcc26 instances grouped by the number of nonzeros in $\boldsymbol{A}$.

ever, each main loop iteration terminates once all nodes $\{\mathcal{N}_{w,r}\}$ have completed their work. Hence, running more nodes in parallel raises the risk of one node slowing down the others. Therefore, the best choice of W inherently depends on the specific instance and device.

Table 1 MIPcc26 benchmarking results in terms of gap integral, final gap, and number of feasible instances.

Method	Gap Integral	Final Gap	# Feasible
PaNGEA	**112.94**	**0.27**	**44**
PaNGEA without CPU thread	160.47	0.49	39
PaNGEA W=1,R=1	138.87	0.40	41
Local-MIP	151.25	0.45	42
Gurobi heuristics	**85.42**	**0.20**	**45**
Gurobi heuristics without branching	111.86	0.30	45

Figure 4 illustrates this trade-off by reporting the average final gap and GPU utilization (measured via the GPU SM-active rate; see Appendix B for a detailed discussion) for different values of W. We stratify results across three subsets of instances, based on the number of nonzero entries in the constraint matrix $\boldsymbol{A}$. We also report the performance of an adaptive strategy where W is set according to the instance size (in bytes) and the available GPU memory. Precisely, we set W to the closest positive integer to $\min(5 \cdot 10^{-3} \cdot \frac{\text{GPU memory}}{R \times \text{size of } \mathcal{P}}, 16)$.

Across the three subsets of instances, we observe that higher GPU SM-active rate is generally associated with better final gap; that the best fixed value of W varies by subset; that no single value of W dominates across all instances; and that our adaptive choice of W is effective, achieving competitive final gaps while maintaining high GPU SM-active rates. In Appendix B, we report the GPU SM-active rates for the relaxation and the local-search phases of our core subroutine separately.

3.4. Benchmarking

We report the aggregated benchmarking results on the 50 MIPcc26 instances and 233 MIPLIB instances[2] in Tables 1 and 2, respectively. All instances are processed through Gurobi presolve. We compare our algorithm PaNGEA (with W adaptive and $R = 6$) with a variant of PaNGEA without the CPU local-search thread, a non-parallel version of PaNGEA with $W = 1$ and $R = 1$, a CPU implementation of Local-MIP[3] (Lin et al. 2025), Gurobi heuristics, and Gurobi heuristics without branching[4]. We report instance-level feasible objective value results in Tables D.1–D.6.

Table 2 MIPLIB benchmarking results in terms of gap integral, final gap, and number of feasible instances.

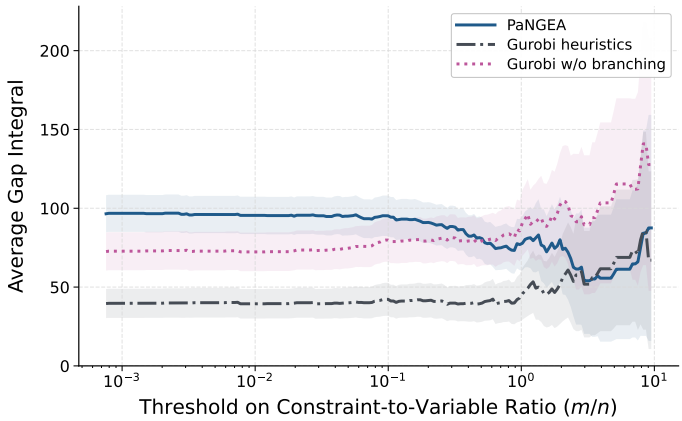
Method	Gap Integral	Final Gap	# Feasible
PaNGEA	**91.75**	**0.24**	**211**
PaNGEA without CPU thread	112.90	0.33	203
PaNGEA W=1,R=1	104.22	0.29	205
Local-MIP	126.83	0.39	203
Gurobi heuristics	**29.20**	**0.05**	**226**
Gurobi heuristics without branching	63.14	0.17	214

PaNGEA achieves the best performance among the non-commercial methods on both benchmark sets and on all three performance criteria (gap integral, final gap, number of feasible instances). It achieves average gap integrals of 112.94 on MIPcc26 and 91.75 on MIPLIB, and average final gaps of 0.27 on MIPcc26 and 0.24 on MIPLIB. Compared to the non-commercial baselines, PaNGEA achieves 12–30% lower gap integral and 17–45% lower final gap. In other words, it finds better feasible solutions faster on average. By looking at the distribution of final gap and gap integral over instances (see Figures C.1-C.2 in Appendix C), we observe that PaNGEA not only improves the averages but shifts the distribution downwards, with noticeable benefit on the tail (i.e., the hardest instances). For example, it reduces the 75th percentile of the gap integral on MIPcc26 (resp. MIPLIB) from 268 (resp. 247) for Local-MIP down to 140 (resp. 146).

Comparing the performance of PaNGEA with that of PaNGEA without the CPU thread, we observe that the background CPU Local-MIP thread is an important component of our architecture, corroborating observations made for similar hybrid implementations in related contexts (e.g., Tjusila et al. 2026, Liu and Lodi 2026). The gain is due to a different behavior of Local-MIP on CPUs vs. GPUs. On the GPU, batching increases the within-iteration workload at the expense of fewer iterations. Hence, running Local-MIP on the CPU thread can be seen as an intensification mechanism that explores the neighborhood of the best solution found so far more deeply.

We now compare the performance of PaNGEA with commercial methods. We note that Gurobi's performance is much stronger on the MIPLIB instances, most likely because these instances have been widely used to develop and engineer existing solvers. On MIPcc26, PaNGEA achieves a smaller final gap than Gurobi without branching (0.27 vs 0.30), while their gap integrals are similar (112.94 vs 111.86). On MIPLIB, Gurobi without branching achieves a smaller gap integral and final gap (63.14 and 0.17 vs 91.75 and 0.24). Gurobi heuristics achieves a smaller final gap over both test sets.

Figure 5 Average gap integral over benchmark instances stratified by constraint-to-variable ratio.



Note. The x-axis is a threshold t on the constraint-to-variable ratio and is shown on a log scale. Each line represents the average gap integral over all instances with a constraint-to-variable ratio m/n greater than t, as t increases. Shaded regions are 95% confidence intervals.

We find that a key driver in relative performance is the constraint-to-variable ratio, m/n. Figure 5 shows the average gap integral over instances such that $m/n >$ is greater than a threshold (x-axis). We observe that Gurobi's performance tends to deteriorate (gap integral increases) as the constraint-to-variable ratio t increases. In contrast, PaNGEA exhibits the opposite behavior. For example, PaNGEA achieves a lower average gap integral than Gurobi without branching on the instances with $m/n > 0.5$, and than Gurobi heuristics on the instances with $m/n > 3$. We observe a similar pattern for final gap as well (Figure C.3). This suggests that PaNGEA could serve as a useful complement to, not substitute of, existing methods, especially for instances with a high constraint-to-variable ratio. We hypothesize that the poor performance of PaNGEA when m/n is small could be due to the local-search procedure itself, which only changes one coordinate at each iteration (hence, is sensitive to high n values), while updating the left-hand side of the m constraints is efficiently handled on GPUs.

4. Conclusion

We introduce PaNGEA, a primal heuristic for MIO that uses GPUs not only for acceleration but also to explore multiple node-generation strategies in parallel. The key idea is to replace heuristic selection by batched heuristic exploration: rather than betting on one rule, PaNGEA evaluates several candidate node-generation strategies simultaneously and continues from the most promising nodes. Empirically, this design leads to the strongest performance among the non-commercial methods in the MIPcc26 and MIPLIB benchmarks, with clear gains over non-parallel variants.

More broadly, our results suggest that GPU parallelism can enable new heuristic architectures for MIO, not just faster implementations of existing ones.

Notes

[1]https://www.mixedinteger.org/2026/competition/

[2]We exclude 7 infeasible instances.

[3] For Local-MIP, we follow https://github.com/shaowei-cai-group/Local-MIP. We do not compare with Çördük et al. (2025) and Tjusila et al. (2026) because the corresponding benchmarking code is not available at the time of our experiment.

[4]We configure the "heuristics" mode of Gurobi by setting $\texttt{MIPFocus} = 1, \texttt{Heuristics} = 1$ and the "without branching" mode by setting $\texttt{Cuts} = 3, \texttt{NodeLimit} = 1, \texttt{MIPFocus} = 1, \texttt{Heuristics} = 1$.

References

Abbas A, Swoboda P (2022) FastDOG: Fast discrete optimization on GPU. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 439–449.

Abdelatti MF, Sodhi MS (2020) An improved GPU-accelerated heuristic technique applied to the capacitated vehicle routing problem. *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, 663–671.

Applegate D, Hinder O, Lu H, Lubin M (2023) Faster first-order primal-dual methods for linear programming using restarts and sharpness. *Mathematical Programming* 201(1):133–184.

Berthold T (2013) Measuring the impact of primal heuristics. *Operations Research Letters* 41(6):611–614.

Bixby RE (2012) A brief history of linear and mixed-integer programming computation. *Optimization Stories*, 107–121 (European Mathematical Society-EMS-Publishing House GmbH).

Blin N, Gualandi S, Maes C, Lodi A, Stellato B (2026) Batched first-order methods for parallel LP solving in MIP. *arXiv preprint arXiv:2601.21990* .

Çördük A, Sielski P, Boucher A, Aatish K (2025) GPU-accelerated primal heuristics for mixed integer programming. *arXiv preprint arXiv:2510.20499* .

Czapiński M, Barnes S (2011) Tabu search with two approaches to parallel flowshop evaluation on CUDA platform. *Journal of Parallel and Distributed Computing* 71(6):802–811.

Danna E, Rothberg E, Pape CL (2005) Exploring relaxation-induced neighborhoods to improve MIP solutions. *Mathematical Programming* 102(1):71–90.

De Rosa A, Khajavirad A, Wang Y (2026) On the power of linear programming for k-means clustering. *INFORMS Journal on Optimization* Forthcoming.

Fischetti M, Glover F, Lodi A (2005) The feasibility pump. *Mathematical Programming* 104(1):91–104.

Fischetti M, Lodi A (2003) Local branching. *Mathematical Programming* 98(1):23–47.

Gleixner A, Hendel G, Gamrath G, Achterberg T, Bastubbe M, Berthold T, Christophel P, Jarck K, Koch T, Linderoth J, et al. (2021) MIPLIB 2017: Data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation* 13(3):443–490.

Guzelsoy M, Nemhauser G, Savelsbergh M (2013) Restrict-and-relax search for 0–1 mixed-integer programs. *EURO Journal on Computational Optimization* 1(1–2):201–218.

Kempke NC, Koch T (2025) Low-precision first-order method-based fix-and-propagate heuristics for large-scale mixed-integer linear optimization. *arXiv preprint arXiv:2503.10344* .

Krizhevsky A, Sutskever I, Hinton GE (2012) ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems* 25.

Lin P, Cai S, Zou M, Lin J (2025) Local-MIP: Efficient local search for mixed integer programming. *Artificial Intelligence* 104405.

Liu J, Lodi A (2026) From sequential nodes to GPU batches: Parallel branch and bound for optimal k-sparse GLMs. *arXiv preprint arXiv:2605.22188* .

Liu J, Lodi A, Shafiee S (2026) GPU-friendly and linearly convergent first-order methods for certifying optimal k-sparse GLMs. *arXiv preprint arXiv:2603.01306* .

Lu H, Yang J (2025) cuPDLP.jl: A GPU implementation of restarted primal-dual hybrid gradient for linear programming in Julia. *Operations Research* 73(6):3440–3452.

Luteberget B, Sartor G (2023) Feasibility jump: An LP-free lagrangian MIP heuristic. *Mathematical Programming Computation* 15(2):365–388.

Meng X, Lucas R, Mazumder R (2026) A GPU-accelerated nonlinear branch-and-bound framework for sparse linear models. *arXiv preprint arXiv:2602.04551* .

Neira DEB, Laird CD, Lueg LR, Harwood SM, Trenev D, Venturelli D (2024) Utilizing modern computer architectures to solve mathematical optimization problems: A survey. *Computers & Chemical Engineering* 184:108627.

Salvagnin D, Roberti R, Fischetti M (2025) A fix-propagate-repair heuristic for mixed integer programming. *Mathematical Programming Computation* 17(1):111–139.

Schulz C (2013) Efficient local search on the GPU: Investigations on the vehicle routing problem. *Journal of Parallel and Distributed Computing* 73(1):14–31.

Tjusila GK, Hoen A, Kempke NC, Mexi G, Berthold T, Gleixner A, Koch T, Pokutta S (2026) CHAP: A hybrid GPU-CPU heuristic for MIP. *arXiv preprint arXiv:2605.05086* .

Wei N, Liang J (2025) GFORS: GPU-accelerated first-order method with randomized sampling for binary integer programs. *arXiv preprint arXiv:2510.27117* .

Wolsey LA (2008) *Mixed Integer Programming*, 1–10 (John Wiley & Sons, Ltd), ISBN 9780470050118.

Zhang H, Kerkenhoff T, Kichler N, Dahmen M, Mitsos A, Naumann U, Bongartz D (2025) Accelerating deterministic global optimization via GPU-parallel interval arithmetic. *arXiv preprint arXiv:2507.20769* .

Appendix A: Detailed Algorithm Description

We now provide details on the implementation of each component in the PaNGEA algorithm (see Figure 1).

A.1. Initialization

The algorithm starts by analyzing the root node using our core subroutine.

First, we solve the root node relaxation using two solvers concurrently on different threads: a primal-dual first-order method (PDLP) and the commercial solver Gurobi. Solving the root node relaxation is important since its solution serves as a warm start for other relaxation solves. We use both solvers concurrently to achieve robust performance, since the performance of a specific method is highly instance-dependent.

Then, we run local search from the relaxed solution. If the local search fails to find a feasible solution, we consider the $R = 6$ node-generation strategies from Section 2.3, including the *deep diving with bound propagation* strategy. Otherwise, we only consider the other $R = 5$ strategies.

A.2. Implementation of the core subroutine

Stopping criteria. Our main objective is to encourage a diverse exploration of the feasible space, so we are comfortable stopping each step of the process (node relaxation and local search) early. For the node relaxation, in addition to the PDLP convergence criteria, we impose a 5-second time limit. For the local search, we stop after 5 seconds, $2,000,000$ iterations, or if there is no improvement on the best solution found for more than 2 seconds.

Fallback. There are cases in which solving the node relaxation on GPUs with PDLP is less efficient than using a commercial solver like Gurobi on CPUs. To ensure our algorithm remains competitive, we implement the following strategy: when a node has at least 95% of integral coordinates fixed and less than $10,000$ effective variables, or when the root node relaxation was solved (on CPU) in less than $5/B$ seconds, we use Gurobi to solve the node relaxation rather than batching it with the other node relaxations on the GPU.

Note that we use the output from the node relaxation $\boldsymbol{x}^{\mathrm{LP}}$, $\boldsymbol{y}^{\mathrm{LP}}$ as initialization for the local-search procedure. When using PDLP, we take the last iterate, even when the algorithm did not converge (because of the time limit or infeasibility). When using Gurobi, however, it may not provide a candidate $(\boldsymbol{x}^{\mathrm{LP}}, \boldsymbol{y}^{\mathrm{LP}})$ if the algorithm times out or if the relaxation is infeasible. To circumvent the issue, whenever we decide to solve at least one node relaxation with Gurobi instead of PDLP, we first run these Gurobi solves. If a node relaxation is either infeasible or times out with Gurobi, we then solve it again with PDLP on the GPU (batched with the other node relaxations), to obtain an initial $(\boldsymbol{x}^{\mathrm{LP}}, \boldsymbol{y}^{\mathrm{LP}})$ for the local search.

Partial solve. After each local-search procedure, we perform partial optimization on the best node (with the best feasible objective), i.e., we optimize the objective with respect to the continuous variables while all integer variables are fixed at their current value. This (highly) restricted linear optimization problem can be solved very efficiently (we use Gurobi on the CPU). To avoid delaying the local-search iteration too much, we impose a 0.5-second time limit on the solver.

A.3. Caching

We store the instance-specific data persistently on both the GPU and CPU, namely, $\boldsymbol{\ell}^0$, $\boldsymbol{u}^0$, $\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{c}$, $\mathcal{I}$, and $\mathcal{I}_b$.

To efficiently warm-start the PDLP algorithm at each iteration of the main loop, we cache the terminal state of the PDLP algorithm for each node and use it to warm-start the algorithm for child nodes. We also have a CPU-side cache for the linear optimization solver (optimal basis). Recall that the CPU linear optimization solver is needed (i) to quickly solve node relaxations in the diving strategy; (ii) when doing partial solves in the local-search procedure; or (iii) when it is expected to be more efficient than PDLP on GPUs for node relaxations.

Finally, we keep a pool of 'elite' solutions, $\mathcal{E}$. We limit the size of the elite pool to $|\mathcal{E}| \leq 20$. For memory efficiency, only the binary coordinates $\boldsymbol{x}_{\mathcal{I}_b}$ are stored. Note that the elite pool does not necessarily contain feasible solutions. Before the first feasible solution is found, the elite pool consists of 'promising' infeasible solutions (typically evaluated based on constraint violation). Once a feasible solution is found, all infeasible solutions are removed from the pool.

The elite pool is populated with objective value and diversity in mind. When a new feasible solution is found, we add it to the pool if $|\mathcal{E}| < 20$. If $|\mathcal{E}| = 20$, we compute the Hamming distance between the new solution and all the solutions in the pool. If no elite solution is at a distance less than 5, the new solution is compared to the worst elite solution and replaces it provided it achieves a better (i.e., lower) objective value. If there exists an elite solution at a distance less than 5, the new solution is compared to the closest elite solution and replaces it provided it achieves a better (i.e., lower) objective value.

A.4. Details about node-generation strategies

Elite. The *Elite* node-generation strategy first unfixes and then fixes some of the binary coordinates in $\mathcal{I}_b$. Let $\mathcal{F}_0$ and $\mathcal{F}_1$ denote the set of binary coordinates fixed to 0 and 1 in the current node, respectively.

For any $j \in \mathcal{I}_b$, we compute the average value of x_j in the pool: $p_{\text{base}}(j) = \frac{1}{E}\sum_{e=1}^{E} x_j^{(e)}$. For every $k \in \mathcal{F}_1$ (i.e., we are currently fixing $x_k = 1$), $p_{\text{base}}(k)$ indicates the fraction of elite solutions that also have $x_k = 1$. Similarly for $\mathcal{F}_0$ and $1 - p_{\text{base}}(k)$. Accordingly, we sort $k \in \mathcal{F}_0 \cup \mathcal{F}_1$ according to the so-called alignment

$$a_{\texttt{unfix}}(k) = \begin{cases} p_{\text{base}}(k) & \text{if } k \in \mathcal{F}_1, \\ 1 - p_{\text{base}}(k) & \text{if } k \in \mathcal{F}_0. \end{cases},$$

and unfix coordinates with the lowest alignment. We update the sets $\mathcal{F}_0$ and $\mathcal{F}_1$ accordingly.

Then, for each unfixed binary coordinate $j \in \mathcal{I}_b \setminus (\mathcal{F}_0 \cup \mathcal{F}_1)$, we compute a score $a_{\texttt{fix}}(j)$ to capture how popular $x_j = 1$ or $x_j = 0$ is among the elite pool. Formally, we set $a_{\texttt{fix}}(j) = \lambda\, p_{\text{base}}(j) + (1-\lambda)\, p_{\text{corr}}(j)$ with $\lambda = 1/2$ and

$$p_{\text{corr}}(j) = \frac{1}{|\mathcal{F}_1|} \sum_{k \in \mathcal{F}_1} \left(\frac{1}{|\mathcal{E} \cap \{\boldsymbol{x} : x_k = 1\}|} \sum_{e \in \mathcal{E} : x_k^{(e)} = 1} x_j^{(e)} \right).$$

The inner term provides the average value of x_j over the elite solutions that have their binary coordinate k fixed to 1, and $p_{\text{corr}}(j)$ averages this probability over $k \in \mathcal{F}_1$. We set the averages to $1/2$ (neutral) when the denominator is zero. The intuition is that $p_{\text{corr}}(j)$ constitutes an (imperfect) adjustment to $p_{\text{base}}(j)$ based on the variables that are currently fixed to 1. We only consider variables that are fixed to 1 because they are often more informative than those fixed to 0. We then decide to fix the variables with the largest value of $|a_{\texttt{fix}}(j) - 1/2|$.

As a safeguard against potentially time-consuming computation for $p_{\text{corr}}(j)$, we fall back to $a_{\texttt{fix}}(j) = p_{\text{base}}(j)$ if $|\mathcal{F}_1| \cdot |\mathcal{I}_b \setminus (\mathcal{F}_0 \cup \mathcal{F}_1)| > 5 \cdot 10^6$.

Time limit. For *Diving*, *Elite*, and *Deep diving with bound propagation*, we allow for at most 3 seconds per strategy per iteration and fall back to *reduced costs* when the time budget is exceeded.

Hyperparameters. The 'Reduced costs' strategy changes the fixing status of $\mu = |\mathcal{J}|/|\mathcal{I}| = 4\%$ of the integer variables. 'Diving' fixes $\mu = 5\%$ of the remaining (i.e., unfixed) integer variables. 'Consensus restart' defines consensus as shared across 80% of the elite solutions. 'Elite' unfixes and then fixes $\mu = 4\%$ of the binary variables $\mathcal{I}_b$. 'Best' fixes 90% of the coordinates to those of the best incumbent. 'Deep diving' does not have a hyperparameter.

For the methods with a hyperparameter denoted by μ (namely, reduced costs, diving, elite), we consider an actual rate equal to μ plus Gaussian noise $\mathcal{N}(0, (\mu/3)^2)$ (with proper clamping), to avoid duplicating the same nodes.

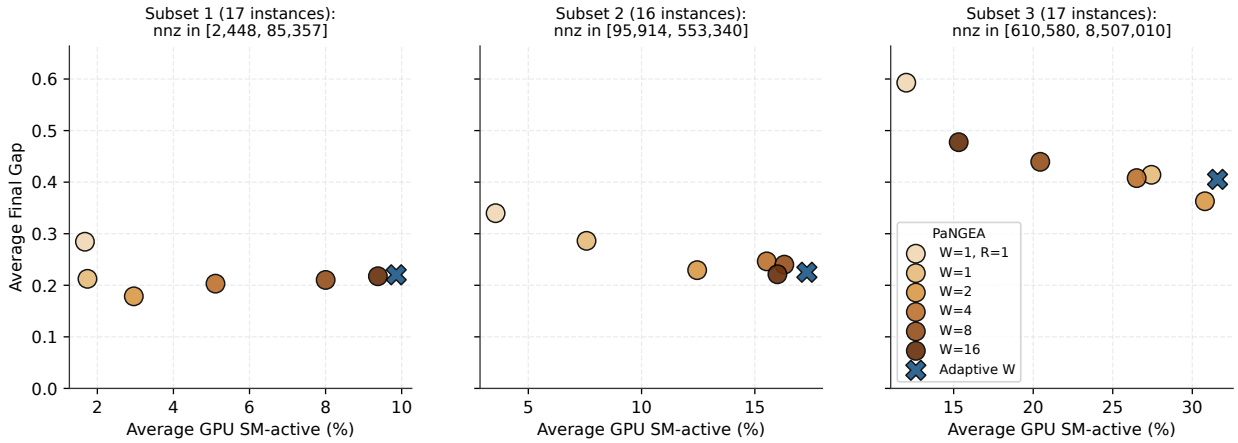
Appendix B: GPU Utilization Diagnostics

We report the GPU SM-active rate rather than the `nvidia-smi` GPU-Util metric in Section 3.3. NVIDIA's DCGM documentation defines SM activity as the fraction of time at least one warp is active on a multiprocessor, averaged over all multiprocessors.[1] In contrast, NVIDIA's `nvidia-smi` documentation defines GPU-Util as the percent of time over the sampling period during which one or more kernels execute on the GPU.[2] Thus, GPU-Util is a busy-time measure and may be misleading as a proxy for how much of the GPU is occupied, while GPU SM-active rate reflects how broadly work is distributed across the GPU's streaming multiprocessors.

Figures B.1 and B.2 separate the relaxation-solving and local-search phases, respectively. The relaxation-solving phase can terminate early for some nodes, for example, when a relaxation is solved to optimality or proved infeasible. As a result, increasing W does not necessarily increase the effective amount of GPU work throughout the relaxation-solving phase, and the GPU SM-active rate curve need not be monotone in W. This is what we observe on the largest instances (in terms of nonzero entries of $\boldsymbol{A}$, right panel). By contrast, local search has no analogous solved-node certificate; once the local-search phase is launched, each node contributes search work until the time (iteration) limit. This matches our observation that local-search GPU SM-active rate increases monotonically with the batch size for all subsets of instances.

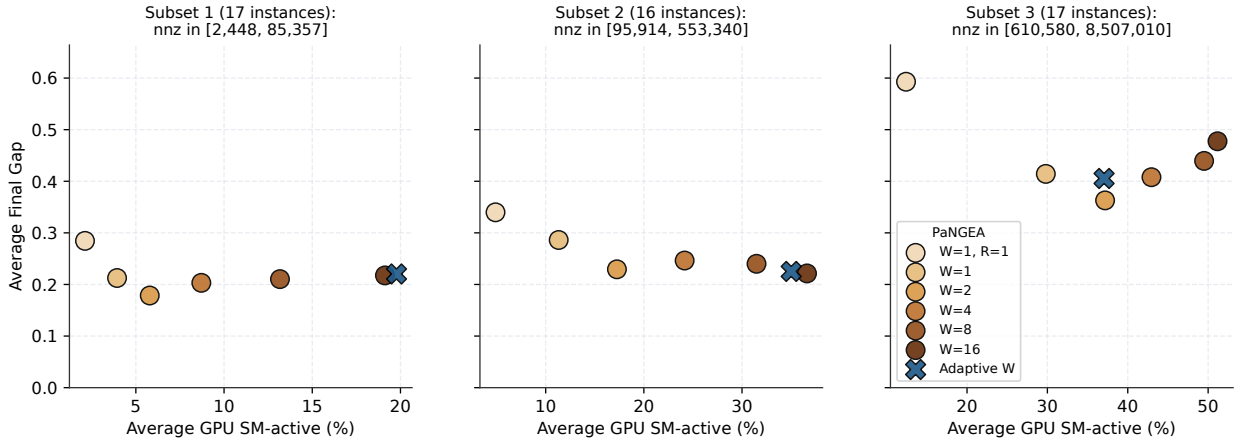
The gap results should therefore be read as a performance trade-off rather than as a target to maximize GPU SM-active rate alone. Larger W exposes more parallel node exploration and usually increases GPU activity, especially during local search, but it can also allocate work to less promising nodes. Consequently, the final gap is not monotone in GPU SM-active rate. We observe that our adaptive rule generally achieves higher GPU SM-active rates and better final gaps at the same time.

Figure B.1 GPU SM-active rate and final gap during solving node relaxations.



Note. Average final gap and GPU SM-active rate for PaNGEA with different values of W, computed from the node-relaxation phase and averaged within three subsets of MIPcc26 instances grouped by the number of nonzeros in $\boldsymbol{A}$.

Figure B.2 GPU SM-active rate and final gap during local search.



Note. Average final gap and GPU SM-active rate for PaNGEA with different values of W, computed from the local-search phase and averaged within three subsets of MIPcc26 instances grouped by the number of nonzeros in $\boldsymbol{A}$.

Appendix C: Supplementary Benchmarking Analysis Results

This section reports supplementary benchmarking diagnostics in Section 3.4 with results in terms of the final gap.

Figures C.1 and C.2 report boxplots of the gap integral and final gap distribution, respectively, across the MIPcc26 and MIPLIB instances.

Figure C.3 displays the average final gap over all instances (in MIPcc26 and MIPLIB combined) such that $m/n > t$ for increasing values of t.

Figure C.1 Gap integral distribution for benchmark methods.

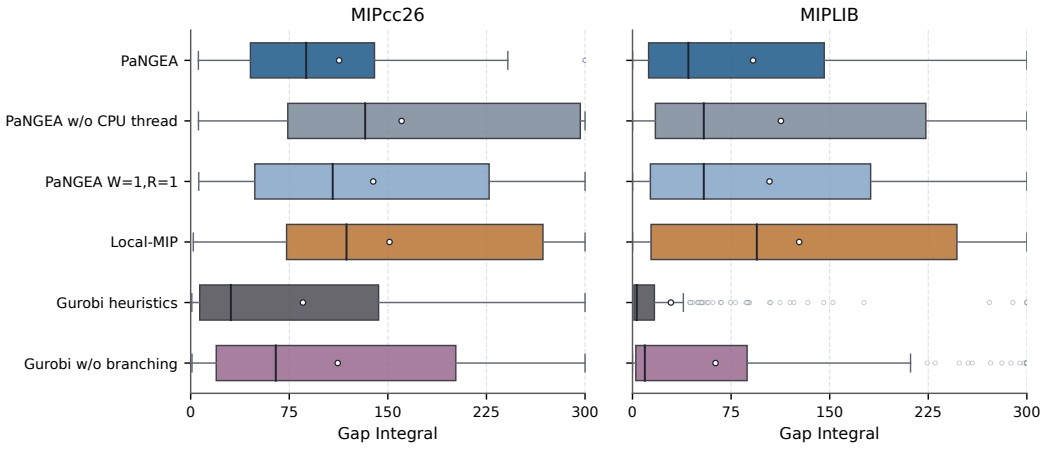


Figure C.2 Final-gap distributions for benchmark methods.

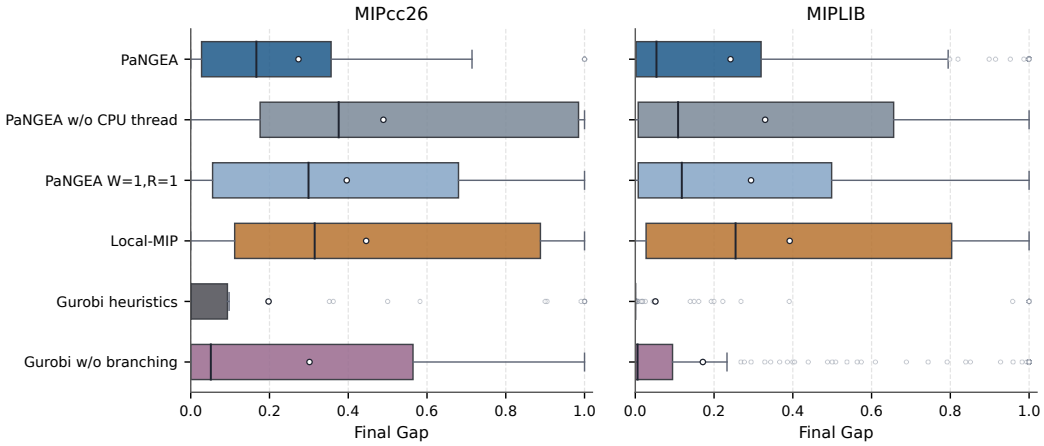
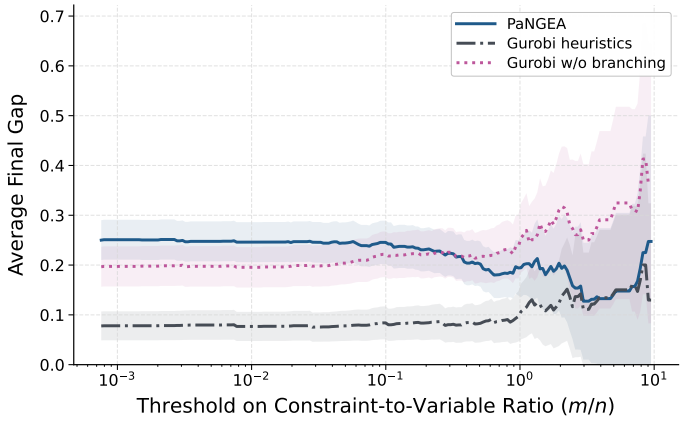


Figure C.3 Average final gap over instances stratified by constraint-to-variable ratio.



Note. The x-axis is a threshold t on the constraint-to-variable ratio and is shown on a log scale. Each line represents the average final gap over all instances with a constraint-to-variable ratio m/n greater than t, as t increases. Shaded regions are 95% confidence intervals.

Table C.1 counts the number of instances where PaNGEA wins/ties/loses against Gurobi's baseline, in terms of gap integral, over all instances and over instances with $m/n > 1$. The win rate of PaNGEA vs. Gurobi heuristics (resp., Gurobi without branching) increases by around 90% (resp., 50%) after narrowing down to instances with $m/n > 1$. Table C.2 reports the same analysis, using final gap instead as a metric.

Table C.1 PaNGEA win/tie/loss counts against benchmark comparators in terms of gap integral.

PaNGEA vs.	Win / Tie / Loss	
	All instances	$m/n > 1$
Gurobi heuristics	11 (3.9%) / 62 / 210	8 (7.3%) / 32 / 69
Gurobi without branching	73 (25.8%) / 59 / 151	42 (38.5%) / 26 / 41

Table C.2 PaNGEA win/tie/loss counts against benchmark comparators in terms of final gap.

PaNGEA vs.	Win / Tie / Loss	
	All instances	$m/n > 1$
Gurobi heuristics	30 (10.6%) / 13 / 240	20 (18.3%) / 5 / 84
Gurobi without branching	68 (24.0%) / 18 / 197	43 (39.4%) / 6 / 60

Appendix D: Instance-wise Results on the Test Set

Tables D.1–D.6 report the instance-wise feasible objective values of the algorithms benchmarked in Section 3.4.

Table D.1 Computational results on the MIPcc26 test set in terms of feasible objective value.

Instance	Final Feasible Objective						
	PaNGEA	PaNGEA w/o CPU thread	PaNGEA W=1,R=1	Local-MIP	Gurobi heuristics	Gurobi w/o branching	Provided baseline
01	−2.14e5	−2.14e5	−2.08e5	−1.12e5	**−2.19e5**	−2.19e5	−2.19e5
02	Inf	Inf	Inf	Inf	**−3.58e5**	−3.57e5	−3.58e5
03	284.00	837.16	323.60	335.40	**248.50**	350.10	**248.50**
04	1.07e3	1.22e3	Inf	Inf	882.00	**868.30**	1.09e3
05	1.07e3	1.14e3	Inf	Inf	922.80	**833.00**	1.04e3
06	5.35e4	5.84e4	5.40e4	5.38e4	**5.28e4**	5.63e4	5.51e4
07	8.17e4	8.19e4	7.95e4	8.77e4	7.92e4	**7.86e4**	8.10e4
08	4.32e4	4.29e4	4.50e4	4.39e4	**3.94e4**	4.07e4	4.01e4
09	15.00	16.00	17.00	23.00	**10.00**	215.00	**10.00**
10	12.00	11.00	14.00	13.00	**10.00**	88.00	**10.00**
11	676.00	789.00	687.00	716.00	**492.00**	603.00	729.00
12	211.00	256.00	207.00	214.00	**193.00**	198.00	**193.00**
13	Inf	Inf	Inf	Inf	**5.00e3**	5.02e3	5.05e3
14	8.63e5	9.04e5	1.56e6	1.56e6	**4.92e5**	4.94e5	4.95e5
15	1.02e6	1.25e6	1.73e6	1.69e6	6.78e5	**6.76e5**	6.78e5
16	46.00	Inf	74.00	**45.00**	Inf	Inf	Inf
17	Inf	Inf	Inf	Inf	Inf	Inf	Inf
18	17.00	19.00	**16.00**	17.00	**16.00**	Inf	**16.00**
19	6.00	7.00	6.00	6.00	**5.00**	6.00	**5.00**
20	11.00	11.00	11.00	11.00	**7.00**	**7.00**	**7.00**
21	12.00	14.00	12.00	14.00	**10.00**	**10.00**	**10.00**
22	228.63	221.63	218.44	250.95	**171.50**	205.88	**171.50**
23	263.90	263.90	339.30	403.36	**100.84**	149.56	**100.84**
24	436.51	465.02	449.30	462.09	**431.24**	436.51	**431.24**
25	**−32.00**	**−32.00**	**−32.00**	**−32.00**	**−32.00**	−28.00	**−32.00**
26	**−88.00**	**−88.00**	−87.00	−84.00	−83.00	−84.00	−86.00
27	**−142.00**	−100.00	**−142.00**	−100.00	−92.00	−92.00	−92.00
28	2.68e5	3.88e5	2.41e5	2.45e5	**2.12e5**	4.30e5	3.74e5
29	8.61e5	Inf	1.43e6	**8.05e5**	Inf	1.47e6	Inf
30	**3.59e6**	Inf	Inf	4.26e6	Inf	1.51e7	Inf
31	−2.47e3	−1.66e3	−2.46e3	−2.45e3	−2.51e3	**−2.51e3**	−2.51e3
32	−3.41e3	−1.09e3	−3.39e3	−3.36e3	−3.49e3	**−3.49e3**	−3.49e3
33	**−2.04e3**	−1.62e3	−872.82	−1.57e3	−193.65	−193.65	−331.70
34	108.00	120.00	110.00	118.00	**100.00**	106.00	**100.00**
35	158.00	176.00	154.00	180.00	**148.00**	150.00	**148.00**
36	382.00	530.00	380.00	490.00	**328.00**	**328.00**	**328.00**
37	7.00	26.00	6.00	4.00	4.00	Inf	**2.00**
38	147.00	3.48e3	5.30e4	5.51e4	**83.00**	272.00	89.00
39	Inf	Inf	Inf	Inf	478.00	**464.00**	**464.00**
40	Inf	Inf	Inf	Inf	570.00	**524.00**	Inf
41	Inf	Inf	Inf	Inf	Inf	Inf	Inf
42	4.12e7	5.15e7	2.72e7	3.78e8	**2.49e7**	2.49e7	2.49e7
43	2.85e7	2.59e7	3.53e7	4.12e8	**2.43e7**	2.43e7	2.43e7
44	1.36e8	Inf	1.05e8	2.12e9	5.70e7	**5.70e7**	5.70e7
45	**4.00**	4.55e3	23.00	316.00	454.00	8.89e3	**4.00**
46	**1.00**	81.00	**1.00**	**1.00**	10.00	180.00	56.00
47	73.00	Inf	**67.00**	97.00	105.00	128.00	116.00
48	1.90e8	1.72e8	2.32e8	3.00e8	**1.52e8**	1.52e8	1.52e8
49	**1.67e8**	2.21e8	2.21e8	2.78e8	3.02e8	3.03e8	2.69e8
50	1.92e8	1.75e8	2.19e8	2.82e8	1.30e8	**1.30e8**	1.30e8

Table D.2 Computational results on the MIPLIB test set in terms of feasible objective value.

Instance	Final Feasible Objective						
	PaNGEA	PaNGEA w/o CPU thread	PaNGEA W=1,R=1	Local-MIP	Gurobi heuristics	Gurobi w/o branching	Provided baseline
30n20b8	402.00	553.00	553.00	553.00	302.00	302.00	302.00
50v-10	3.41e3	3.50e3	3.62e3	4.70e3	3.31e3	3.42e3	3.31e3
CMS750_4	252.00	252.00	252.00	252.00	252.00	252.00	252.00
academictimetablesmall	681.00	678.00	641.00	1.08e3	28.00	3.30e3	0.00
air05	2.65e4	2.75e4	2.91e4	3.44e4	2.64e4	2.66e4	2.64e4
app1-1	−3.00	−2.00	−2.00	Inf	−3.00	−3.00	−3.00
app1-2	Inf	−20.00	Inf	Inf	−30.00	−30.00	−41.00
assign1-5-8	212.00	212.00	212.00	212.00	212.00	212.00	212.00
atlanta-ip	110.01	116.01	126.01	158.01	90.01	93.01	90.01
b1c1s1	3.95e4	4.70e4	3.92e4	4.55e4	2.45e4	2.61e4	2.45e4
bab2	Inf	Inf	Inf	Inf	−3.58e5	−3.54e5	−3.58e5
bab6	−1.93e5	Inf	Inf	Inf	−2.84e5	−2.83e5	−2.84e5
beasleyC3	781.00	775.00	842.00	850.00	754.00	754.00	754.00
binkar10_1	6.87e3	6.80e3	8.11e3	9.04e3	6.74e3	6.75e3	6.74e3
blp-ar98	6.91e3	6.79e3	7.11e3	8.85e3	6.21e3	6.21e3	6.21e3
blp-ic98	4.98e3	4.64e3	5.03e3	6.05e3	4.49e3	4.49e3	4.49e3
bnatt400	Inf	Inf	Inf	Inf	1.00	Inf	1.00
bppc4-08	53.00	54.00	53.00	53.00	53.00	56.00	53.00
brazil3	237.00	337.00	227.00	237.00	24.00	331.00	24.00
buildingenergy	4.54e4	5.12e4	5.19e4	8.42e6	3.33e4	3.33e4	3.33e4
cbs-cta	0.00	43.16	0.00	0.00	0.00	0.00	0.00
chromaticindex1024-7	4.00	4.00	4.00	4.00	4.00	4.00	4.00
chromaticindex512-7	4.00	4.00	4.00	4.00	4.00	4.00	4.00
cmflsp50-24-8-8	Inf	Inf	Inf	Inf	5.69e7	Inf	5.58e7
co-100	4.95e6	6.11e6	6.08e6	6.17e6	2.64e6	2.64e6	2.64e6
cod105	−12.00	−12.00	−12.00	−12.00	−12.00	−12.00	−12.00
comp07-2idx	6.00	15.00	6.00	6.00	6.00	10.00	6.00
comp21-2idx	113.00	117.00	115.00	113.00	86.00	160.00	74.00
cost266-UUE	2.58e7	2.60e7	2.92e7	3.17e7	2.51e7	2.65e7	2.51e7
cryptanalysiskb128n5obj16	Inf	Inf	Inf	Inf	Inf	Inf	0.00
csched007	Inf	Inf	Inf	Inf	351.00	572.00	351.00
csched008	179.00	183.00	179.00	179.00	173.00	186.00	173.00
cvs16r128-89	−95.00	−91.00	−96.00	−94.00	−97.00	−95.00	−97.00
dano3_3	623.21	897.25	634.54	622.97	576.34	576.34	576.34
dano3_5	660.41	Inf	647.15	655.56	576.92	576.92	576.92
decomp2	−160.00	−160.00	−160.00	−160.00	−160.00	−160.00	−160.00
drayage-100-23	1.03e5	1.03e5	1.03e5	1.83e5	1.03e5	1.03e5	1.03e5
drayage-25-23	1.01e5	1.01e5	1.07e5	1.71e5	1.01e5	1.01e5	1.01e5
dws008-01	Inf	Inf	Inf	Inf	3.74e4	7.62e4	3.74e4
eil33-2	934.01	988.14	974.66	989.39	934.01	1.07e3	934.01
eilA101-2	1.43e3	2.41e3	1.46e3	1.38e3	923.01	923.01	880.92
enlight_hard	37.00	37.00	37.00	37.00	37.00	37.00	37.00
ex10	100.00	100.00	100.00	100.00	100.00	100.00	100.00
ex9	81.00	81.00	81.00	81.00	81.00	81.00	81.00
exp-1-500-5-5	7.74e4	7.40e4	1.17e5	1.41e5	6.59e4	6.72e4	6.59e4
fast0507	175.00	175.00	183.00	191.00	174.00	175.00	174.00
fastxgemm-n2r6s0t2	230.00	230.00	230.00	230.00	230.00	527.00	230.00
fhnw-binpack4-48	0.00	0.00	0.00	0.00	0.00	Inf	0.00
fiball	138.00	138.00	138.00	138.00	138.00	138.00	138.00
gen-ip002	−4.75e3	−4.75e3	−4.72e3	−4.48e3	−4.78e3	−4.75e3	−4.78e3

Table D.3 Computational results on the MIPLIB test set in terms of feasible objective value (continued).

Instance	Final Feasible Objective						
	PaNGEA	PaNGEA w/o CPU thread	PaNGEA W=1,R=1	Local-MIP	Gurobi heuristics	Gurobi w/o branching	Provided baseline
gen-ip054	6.90e3	6.87e3	6.89e3	6.92e3	6.84e3	6.90e3	6.84e3
germanrr	5.23e7	5.36e7	5.24e7	5.32e7	4.72e7	4.71e7	4.71e7
gfd-schedulen180f7d50m30k18	Inf	Inf	Inf	Inf	Inf	Inf	1.00
glass-sc	23.00	23.00	23.00	23.00	23.00	23.00	23.00
glass4	1.45e9	1.58e9	1.60e9	1.70e9	1.20e9	1.70e9	1.20e9
gmu-35-40	−2.40e6	−2.40e6	−2.39e6	−2.34e6	−2.41e6	−2.41e6	−2.41e6
gmu-35-50	−2.60e6	−2.60e6	−2.56e6	−2.56e6	−2.61e6	−2.61e6	−2.61e6
graph20-20-1rand	−9.00	−9.00	−9.00	−9.00	−9.00	−9.00	−9.00
graphdraw-domain	1.98e4	1.97e4	1.99e4	2.10e4	1.97e4	2.55e4	1.97e4
h80x6320d	8.24e3	8.24e3	8.24e3	1.61e4	6.38e3	6.38e3	6.38e3
highschool1-aigio	1.05e3	1.38e3	1.33e3	1.81e3	28.00	Inf	0.00
hypothyroid-k1	−2.85e3	−2.85e3	−2.85e3	−2.85e3	−2.85e3	−2.85e3	−2.85e3
ic97_potential	3.98e3	4.00e3	4.04e3	4.05e3	3.94e3	4.05e3	3.94e3
icir97_tension	6.64e3	Inf	Inf	6.99e3	6.38e3	Inf	6.38e3
irish-electricity	6.04e6	Inf	5.70e6	6.18e6	3.72e6	3.72e6	3.72e6
irp	1.22e4	1.22e4	1.22e4	1.22e4	1.22e4	1.22e4	1.22e4
istanbul-no-cutoff	257.94	Inf	Inf	Inf	204.08	204.08	204.08
k1mushroom	−3.29e3	−756.00	−3.29e3	−3.29e3	−3.29e3	−3.29e3	−3.29e3
lectsched-5-obj	24.00	26.00	24.00	24.00	24.00	27.00	24.00
leo1	4.52e8	4.15e8	4.69e8	5.01e8	4.05e8	4.08e8	4.04e8
leo2	4.49e8	4.39e8	5.42e8	5.50e8	4.06e8	4.06e8	4.04e8
lotsize	2.66e6	2.54e6	3.32e6	3.77e6	1.48e6	1.53e6	1.48e6
mad	0.31	0.34	0.33	0.52	0.04	0.66	0.03
map10	−475.00	−430.00	−481.00	−414.00	−495.00	−478.00	−495.00
map16715-04	−94.00	−91.00	−70.00	−55.10	−111.00	−108.00	−111.00
markshare2	75.00	105.00	114.00	107.00	24.00	162.00	1.00
markshare_4_0	3.00	6.00	3.00	3.00	1.00	56.00	1.00
mas74	1.23e4	1.30e4	1.33e4	1.98e4	1.19e4	1.25e4	1.18e4
mas76	4.10e4	4.09e4	4.16e4	4.67e4	4.00e4	4.00e4	4.00e4
mc11	1.19e4	1.20e4	1.35e4	2.94e4	1.17e4	1.17e4	1.17e4
mcsched	2.14e5	2.15e5	2.14e5	2.16e5	2.12e5	2.14e5	2.12e5
mik-250-20-75-4	−5.19e4	−5.19e4	−4.47e4	−4.24e4	−5.23e4	−5.23e4	−5.23e4
milo-v12-6-r2-40-1	1.24e6	1.24e6	Inf	Inf	3.26e5	7.66e5	3.26e5
momentum1	Inf	Inf	Inf	Inf	1.09e5	1.09e5	1.09e5
mushroom-best	0.06	0.06	0.06	0.06	0.06	0.06	0.06
mzzv11	−1.90e4	−1.98e4	−2.00e4	−1.85e4	−2.17e4	−2.17e4	−2.17e4
mzzv42z	−2.01e4	−1.95e4	−2.01e4	−1.97e4	−2.05e4	−2.05e4	−2.05e4
n2seq36q	5.88e4	8.48e4	5.38e4	9.50e4	5.22e4	5.22e4	5.22e4
n3div36	1.31e5	1.31e5	1.49e5	1.47e5	1.31e5	1.31e5	1.31e5
n5-3	9.52e3	1.03e4	1.09e4	1.32e4	8.10e3	8.14e3	8.10e3
neos-1122047	161.00	161.00	161.00	161.00	161.00	161.00	161.00
neos-1171448	−309.00	−309.00	−309.00	−299.00	−309.00	−309.00	−309.00
neos-1171737	−193.00	−193.00	−192.00	−190.21	−195.00	−195.00	−195.00
neos-1354092	Inf	Inf	Inf	Inf	57.00	46.00	46.00
neos-1445765	−1.74e4	−1.74e4	−1.54e4	−1.33e4	−1.78e4	−1.78e4	−1.78e4
neos-1456979	263.00	315.00	351.00	357.00	176.00	451.00	176.00
neos-1582420	102.00	103.00	129.00	153.00	91.00	95.00	91.00
neos-2657525-crna	5.68	7.23	1.81	5.77	1.81	12.15	1.81
neos-2746589-doon	Inf	Inf	Inf	Inf	2.01e3	3.58e3	2.01e3
neos-2978193-inde	−0.67	1.91	4.54	13.54	−2.39	−2.39	−2.39

Table D.4 Computational results on the MIPLIB test set in terms of feasible objective value (continued).

Instance	Final Feasible Objective						
	PaNGEA	PaNGEA w/o CPU thread	PaNGEA W=1,R=1	Local-MIP	Gurobi heuristics	Gurobi w/o branching	Provided baseline
neos-2987310-joes	−5.75e8	−3.64e8	−5.96e8	−5.39e8	−6.08e8	−6.08e8	−6.08e8
neos-3004026-krka	0.00	0.00	0.00	0.00	0.00	0.00	0.00
neos-3024952-loue	6.06e4	5.62e4	1.12e5	1.50e5	2.68e4	Inf	2.68e4
neos-3046615-murg	1.63e3	1.62e3	1.69e3	1.75e3	1.60e3	1.64e3	1.60e3
neos-3083819-nubu	6.32e6	6.32e6	6.37e6	6.49e6	6.31e6	6.31e6	6.31e6
neos-3216931-puriri	1.51e5	Inf	1.31e5	1.56e5	7.13e4	Inf	7.13e4
neos-3381206-awhea	473.00	Inf	471.00	Inf	453.00	453.00	453.00
neos-3402294-bobin	0.07	0.07	0.07	0.07	0.07	0.07	0.07
neos-3555904-turama	−34.70	−34.70	−34.70	−34.70	−34.70	−34.70	−34.70
neos-3627168-kasai	9.89e5	9.89e5	9.95e5	1.00e6	9.89e5	9.89e5	9.89e5
neos-3656078-kumeu	−1.30e4	−1.24e4	−1.29e4	−1.29e4	−1.32e4	Inf	−1.32e4
neos-3754480-nidda	1.30e4	1.30e4	1.51e4	1.66e4	1.29e4	1.45e4	1.29e4
neos-4300652-rahue	3.45	6.23	3.55	4.74	2.14	2.14	2.14
neos-4338804-snowy	1.48e3	1.48e3	1.48e3	1.49e3	1.48e3	1.59e3	1.47e3
neos-4387871-tavua	34.38	36.38	47.87	86.67	33.38	34.56	33.38
neos-4413714-turia	56.04	Inf	Inf	Inf	45.37	45.37	45.37
neos-4532248-waihi	61.60	65.50	74.70	Inf	61.60	69.80	61.60
neos-4647030-tutaki	3.64e4	4.33e4	4.35e4	6.39e4	2.73e4	2.73e4	2.73e4
neos-4722843-widden	2.54e4	2.54e4	Inf	Inf	2.50e4	2.56e4	2.50e4
neos-4738912-atrato	2.99e8	2.99e8	3.22e8	4.17e8	2.84e8	2.85e8	2.84e8
neos-4763324-toguru	1.71e3	1.80e3	1.83e3	2.58e3	1.61e3	1.61e3	1.61e3
neos-4954672-berkel	2.76e6	2.98e6	3.55e6	4.01e6	2.61e6	2.72e6	2.61e6
neos-5049753-cuanza	672.00	Inf	1.71e3	2.08e3	562.00	573.00	562.00
neos-5052403-cygnet	188.00	191.00	183.00	191.00	182.00	182.00	182.00
neos-5093327-huahum	Inf	Inf	Inf	Inf	6.26e3	3.89e4	6.26e3
neos-5104907-jarama	Inf	Inf	Inf	2.68e3	Inf	946.00	935.00
neos-5107597-kakapo	4.45e3	1.87e4	5.83e3	2.19e4	3.64e3	1.42e4	3.64e3
neos-5114902-kasavu	Inf	Inf	2.58e3	1.97e3	657.00	660.00	655.00
neos-5188808-nattai	0.16	2.34	0.12	0.14	0.11	0.22	0.11
neos-5195221-niemur	0.01	0.01	0.01	0.01	0.00	0.01	0.00
neos-631710	206.00	212.00	205.00	206.00	203.00	203.00	203.00
neos-662469	2.35e5	2.35e5	4.85e5	1.35e6	1.84e5	1.84e5	1.84e5
neos-787933	30.00	30.00	30.00	30.00	30.00	30.00	30.00
neos-827175	112.00	112.00	112.00	112.00	112.00	112.00	112.00
neos-848589	1.31e6	2.51e7	1.64e7	3.11e7	2.35e3	2.53e3	2.35e3
neos-860300	3.48e3	3.46e3	3.69e3	Inf	3.20e3	3.54e3	3.20e3
neos-873061	425.78	5.43e3	138.67	1.61e4	113.66	113.66	113.66
neos-911970	56.05	55.85	59.31	64.21	54.76	55.26	54.76
neos-933966	342.00	1.34e3	321.00	1.38e5	318.00	318.00	318.00
neos-950242	4.00	4.00	4.00	4.00	4.00	4.00	4.00
neos-957323	−237.75	−237.75	−237.75	−229.72	−237.76	−237.76	−237.76
neos-960392	−236.00	−36.00	−238.00	0.00	−238.00	−238.00	−238.00
neos17	0.35	0.35	0.36	0.41	0.15	0.15	0.15
neos5	15.00	15.00	15.00	15.08	15.00	15.00	15.00
neos8	−3.72e3	−3.72e3	−3.52e3	−3.52e3	−3.72e3	−3.72e3	−3.72e3
net12	214.00	214.00	214.00	214.00	214.00	296.00	214.00
netdiversion	398.00	440.00	514.00	525.00	242.00	382.00	242.00
nexp-150-20-8-5	253.00	250.00	258.00	262.00	231.00	243.00	231.00
ns1116954	0.00	0.00	0.00	0.00	0.00	Inf	0.00
ns1208400	2.00	2.00	2.00	2.00	2.00	2.00	2.00

Table D.5 Computational results on the MIPLIB test set in terms of feasible objective value (continued).

Instance	Final Feasible Objective						
	PaNGEA	PaNGEA w/o CPU thread	PaNGEA W=1,R=1	Local-MIP	Gurobi heuristics	Gurobi w/o branching	Provided baseline
ns1644855	−791.33	−933.00	−302.79	−46.71	−1.52e3	−1.52e3	−1.52e3
ns1760995	−539.11	−536.14	−549.21	−547.28	−549.21	−360.49	−549.21
ns1830653	2.06e4	2.26e4	2.06e4	2.56e4	2.06e4	3.46e4	2.06e4
ns1952667	Inf	Inf	Inf	Inf	0.00	Inf	0.00
nu25-pr12	5.39e4	5.39e4	5.39e4	5.39e4	5.39e4	5.39e4	5.39e4
nursesched-medium-hint03	2.41e3	2.19e3	4.01e3	8.35e3	118.00	553.00	115.00
nursesched-sprint02	74.00	88.00	65.00	79.00	58.00	58.00	58.00
nw04	1.73e4	1.70e4	1.81e4	2.11e4	1.69e4	1.69e4	1.69e4
opm2-z10-s4	−2.81e4	−2.87e4	−2.91e4	−2.75e4	−3.27e4	−3.24e4	−3.33e4
p200x1188c	1.51e4	1.51e4	1.51e4	1.51e4	1.51e4	1.51e4	1.51e4
peg-solitaire-a3	Inf	Inf	Inf	Inf	1.00	Inf	1.00
pg	−7.96e3	−7.87e3	−6.40e3	−5.52e3	−8.67e3	−8.65e3	−8.67e3
pg5_34	−1.42e4	−1.42e4	−1.37e4	−1.16e4	−1.43e4	−1.43e4	−1.43e4
physicianshed3-3	2.97e6	3.09e6	3.02e6	3.12e6	2.65e6	3.42e6	2.62e6
physicianshed6-2	4.93e4	4.93e4	4.93e4	4.93e4	4.93e4	4.93e4	4.93e4
piperout-08	1.25e5	1.42e5	1.25e5	1.25e5	1.25e5	1.25e5	1.25e5
piperout-27	8.12e3	8.12e3	8.12e3	8.12e3	8.12e3	8.12e3	8.12e3
pk1	23.00	18.00	22.00	28.00	11.00	22.00	11.00
proteindesign121hz512p9	1.50e3	1.51e3	1.49e3	1.51e3	1.48e3	1.49e3	1.47e3
proteindesign122trx11p8	1.76e3	1.76e3	1.75e3	1.76e3	1.75e3	1.76e3	1.75e3
qap10	340.00	418.00	368.00	376.00	340.00	340.00	340.00
radiationm18-12-05	1.76e4	1.76e4	1.76e4	1.76e4	1.76e4	1.76e4	1.76e4
radiationm40-10-02	1.55e5	1.55e5	1.55e5	1.55e5	1.55e5	1.55e5	1.55e5
rail01	Inf	Inf	Inf	Inf	Inf	Inf	−70.57
rail02	Inf	Inf	Inf	Inf	Inf	Inf	−200.45
rail507	176.00	175.00	179.00	190.00	174.00	174.00	174.00
ran14x18-disj-8	4.06e3	4.02e3	4.19e3	4.42e3	3.71e3	3.77e3	3.71e3
rd-rplusc-21	Inf	Inf	Inf	Inf	1.65e5	Inf	1.65e5
reblock115	−3.57e7	−3.56e7	−3.56e7	−3.54e7	−3.68e7	−3.68e7	−3.68e7
rmatr100-p10	433.00	480.00	459.00	425.00	423.00	429.00	423.00
rmatr200-p5	5.46e3	5.67e3	5.39e3	5.31e3	4.52e3	4.64e3	4.52e3
rocI-4-11	−5.05e6	−4.04e6	−5.05e6	−5.05e6	−5.05e6	−4.04e6	−6.02e6
rocII-5-11	−5.66	−1.66	−5.66	−3.50	−5.68	Inf	−6.68
rococoB10-011000	2.56e4	2.98e4	2.40e4	2.64e4	1.94e4	2.14e4	1.94e4
rococoC10-001000	1.40e4	1.30e4	1.41e4	1.58e4	1.15e4	1.16e4	1.15e4
roi2alpha3n4	−50.40	−32.30	−9.68	−57.53	−63.21	−60.85	−63.21
roi5alpha10n8	−9.45	−12.66	−1.74	−0.24	−51.54	−48.68	−52.32
roll3000	1.42e4	1.45e4	1.49e4	1.51e4	1.29e4	1.32e4	1.29e4
s100	−0.12	−0.12	−0.14	−0.13	−0.17	−0.17	−0.17
s250r10	−0.12	−0.09	−0.12	−0.11	−0.17	−0.17	−0.17
satellites2-40	22.00	27.00	−17.00	22.00	−19.00	16.00	−19.00
satellites2-60-fs	−16.00	5.00	9.00	−19.00	−19.00	−19.00	−19.00
savsched1	1.60e4	1.05e5	1.06e4	1.64e4	3.22e3	3.22e3	3.22e3
sct2	−228.89	−228.05	−224.11	−222.58	−230.99	−230.99	−230.99
seymour	423.00	424.00	423.00	424.00	423.00	426.00	423.00
seymour1	412.87	413.61	414.67	415.10	410.76	410.84	410.76
sing326	1.06e7	1.32e7	2.51e7	1.52e8	7.75e6	7.75e6	7.75e6
sing44	8.77e6	9.69e6	1.19e7	1.35e8	8.13e6	8.13e6	8.13e6
snp-02-004-104	1.19e9	1.31e9	6.57e8	8.11e9	5.87e8	5.87e8	5.87e8
sorrell3	−16.00	−16.00	−16.00	−15.00	−16.00	−15.00	−16.00

Table D.6 Computational results on the MIPLIB test set in terms of feasible objective value (continued).

Instance	Final Feasible Objective						
	PaNGEA	PaNGEA w/o CPU thread	PaNGEA W=1,R=1	Local-MIP	Gurobi heuristics	Gurobi w/o branching	Provided baseline
sp150x300d	69.00	69.00	69.00	69.00	69.00	69.00	69.00
sp97ar	6.90e8	7.06e8	7.32e8	1.08e9	6.61e8	6.63e8	6.61e8
sp98ar	5.59e8	5.45e8	5.68e8	8.29e8	5.30e8	5.30e8	5.30e8
splice1k1	−394.00	−394.00	−394.00	−394.00	−394.00	−338.00	−394.00
square41	64.00	56.00	123.00	121.00	15.00	15.00	15.00
square47	76.00	210.00	Inf	Inf	20.00	20.00	16.00
supportcase10	10.00	16.00	8.00	10.00	9.00	9.00	7.00
supportcase12	−7.19e3	−2.86e3	−2.96e3	−154.14	−7.56e3	−7.55e3	−7.56e3
supportcase18	50.00	50.00	50.00	50.00	48.00	49.00	48.00
supportcase19	Inf	Inf	Inf	Inf	Inf	Inf	1.27e7
supportcase22	Inf	Inf	Inf	Inf	Inf	Inf	110.00
supportcase26	1.75e3	1.75e3	1.75e3	1.75e3	1.75e3	1.93e3	1.75e3
supportcase33	−270.00	−200.00	−270.00	−250.00	−345.00	−315.00	−345.00
supportcase40	2.55e4	2.56e4	2.46e4	3.51e4	2.43e4	2.48e4	2.43e4
supportcase42	8.00	8.00	8.00	40.00	7.76	8.00	7.76
supportcase6	6.24e4	6.24e4	6.21e4	7.25e4	5.19e4	5.19e4	5.19e4
supportcase7	−1.01e3	−983.97	−595.18	−628.74	−1.13e3	−1.13e3	−1.13e3
swath1	386.71	386.71	457.32	876.91	379.07	382.89	379.07
swath3	472.41	450.27	495.24	898.52	397.76	401.69	397.76
tbfp-network	81.17	80.29	66.48	76.31	24.16	25.29	24.16
thor50dday	1.32e5	1.44e5	8.51e4	2.05e5	4.04e4	4.04e4	4.04e4
timtab1	8.60e5	8.97e5	9.24e5	1.00e6	7.65e5	8.78e5	7.65e5
tr12-30	1.50e5	1.52e5	1.56e5	1.62e5	1.31e5	1.31e5	1.31e5
traininstance2	7.48e4	7.66e4	7.43e4	7.50e4	7.18e4	7.86e4	7.18e4
traininstance6	2.84e4	2.87e4	2.84e4	2.83e4	2.83e4	3.29e4	2.83e4
trento1	2.52e7	5.55e7	3.53e7	1.26e8	5.19e6	5.65e6	5.19e6
triptim1	Inf	Inf	Inf	Inf	22.87	22.87	22.87
uccase12	1.35e4	1.15e4	1.55e4	9.50e5	1.15e4	1.15e4	1.15e4
uccase9	1.25e4	2.76e4	2.47e4	2.57e6	1.10e4	1.11e4	1.10e4
uct-subprob	334.00	334.00	347.00	353.00	314.00	341.00	314.00
unitcal_7	2.03e7	2.04e7	2.16e7	2.43e7	1.96e7	1.96e7	1.96e7
var-smallemery-m6j6	−142.03	−146.19	−136.31	0.00	−149.38	−147.94	−149.38
wachplan	−8.00	−8.00	−8.00	−8.00	−8.00	−8.00	−8.00

Notes

[1]NVIDIA DCGM's SM Activity metric, https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/feature-overview.html.

[2]NVIDIA System Management Interface documentation, "Utilization," https://docs.nvidia.com/deploy/nvidia-smi/index.html.