






NILS-CHRISTIAN KEMPKE¹, STEPHEN J. MAHER²,
DANIEL REHFELDT³, AMBROS GLEIXNER⁴, THORSTEN
KOCH⁵, SVENJA USLU

Distributed Parallel Structure-Aware Presolving for Arrowhead Linear Programs

¹  0000-0003-4492-9818
²  0000-0003-3773-6882
³  0000-0002-2877-074X
⁴  0000-0003-0391-5903
⁵  0000-0002-1967-0077

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30 84185-0
Telefax: +49 30 84185-125

E-mail: bibliothek@zib.de
URL: <http://www.zib.de>

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

Distributed Parallel Structure-Aware Presolving for Arrowhead Linear Programs

Nils-Christian Kempke* Stephen J. Maher†
Daniel Rehfeldt‡ Ambros Gleixner§ Thorsten Koch¶
Svenja Uslu

Match 03, 2026


Abstract


We present a structure-aware parallel presolve framework specialized to arrowhead linear programs (AHLPs) and designed for high-performance computing (HPC) environments, integrated into the parallel interior point solver PIPS-IPM++. Large-scale LPs arising from automated model generation frequently contain redundancies and numerical pathologies that necessitate effective presolve, yet existing presolve techniques are primarily serial or structure-agnostic and can become time-consuming in parallel solution workflows.


Within PIPS-IPM++, AHLPs are stored in distributed memory, and our presolve builds on this to apply a highly parallel, distributed presolve across compute nodes while keeping communication overhead low and preserving the underlying arrowhead structure. We demonstrate the scalability and effectiveness of our approach on a diverse set of AHLPs and compare it against state-of-the-art presolve implementations, including PaPILO and the presolve implemented within Gurobi. Even on a single machine, our presolve significantly outperforms PaPILO by a factor of 18 and Gurobi’s presolve by a factor of 6 in terms of shifted geometric mean runtime, while reducing the problems by a similar amount to PaPILO. Using a distributed compute environment, we outperform Gurobi’s presolve by a factor of 13.


1 Introduction


Presolve is an indispensable component of modern linear programming (LP) solvers. Its primary purpose is not only to reduce solution time, but also to ensure numerical robustness and algorithmic stability by tightening formulations, eliminating redundancies, and correcting numerical pathologies before the actual solution process begins. In large-scale LPs, these effects are often decisive: poor conditioning, redundant constraints, or

*  0000-0003-4492-9818

†  0000-0003-3773-6882

‡  0000-0002-2877-074X

§  0000-0003-0391-5903

¶  0000-0002-1967-0077

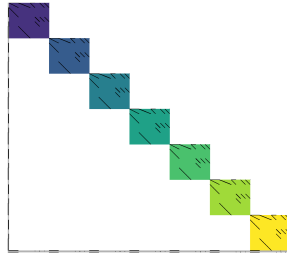


Figure 1: Constraint matrix with the arrowhead structure of a real-world ESOM.

rank deficiencies can severely degrade or even prevent the convergence of simplex, interior-point methods (IPMs), or first-order methods (FOMs). Importantly, such deficiencies are rarely the result of modeling errors. In practice, large-scale LPs are typically generated automatically using high-level modeling tools and scenario-based expansion, where redundancies and linear dependencies arise naturally as a by-product of modular and expressive model construction. Consequently, presolve has become a standard preprocessing step in all state-of-the-art LP solvers.

At the same time, the size and structural complexity of LPs arising in practice have grown dramatically. Many contemporary applications—particularly in energy systems, stochastic programming, and multi-stage planning—give rise to problems with millions of variables and constraints and highly structured constraint matrices. In such settings, solver scalability increasingly depends on the ability to exploit problem structure and parallel hardware effectively. While considerable progress has been made in developing structure-exploiting and distributed solution algorithms, presolve has largely remained a serial, structure-agnostic preprocessing step. As a result, presolve itself can become a computational bottleneck and, more critically, may interfere with or even destroy the very structure that specialized solvers rely on for scalability.

A recurring structural pattern in LPs is the *arrowhead* or *doubly bordered block-diagonal form*, as illustrated in Figure 1. Arrowhead LPs (AHLPs) contain *linking variables* (connecting diagonal blocks vertically) and *linking constraints* (connecting blocks horizontally). This structure generalizes both primal- and dual block-angular problems. AHLPs arise in a wide range of practical applications. Energy system models (ESMs), such as electricity market models with dispatch decisions Rehfeldt et al. (2022), renewable expansion planning Gils et al. (2017), Hörsch et al. (2018), and large-scale (stochastic) economic (re-)dispatch models, can be structured by grouping variables and constraints along spatial or temporal dimensions, resulting in an arrowhead LP. Multi-stage LPs, where decisions at each stage depend on preceding stages, and multi-stage stochastic LPs, covering applications such as asset–liability management, supply network design, revenue management, and portfolio optimization, also exhibit arrowhead structure Castro et al. (2023), Colombo et al. (2011), Steinbach (2001). Staircase LPs used in production scheduling, inventory management, transportation, and multistage system design show local linking structure connecting consecutive diagonal blocks Fourer (1982), Wittrock (1985). Band-diagonal matrices arising in distribution planning

can also be reformulated to expose arrowhead forms Gondzio and Grothey (2006).

The arrowhead structure is not limited to LPs but also arises in a variety of nonlinear optimization problems. When modeled as MIPs, ESMs naturally expose the same arrowhead structure as their LP counterparts Göke (2021), Wetzel et al. (2023), Wiese et al. (2018). Nonlinear portfolio optimization Gondzio and Grothey (2007), nonlinear dynamic optimization Word et al. (2014), model predictive control Rao et al. (1998), nonlinear parameter estimation Zavala et al. (2008), and multi-stage nonlinear programs Pacaud et al. (2024) also exhibit arrowhead system matrices.

Arrowhead structures can often be exploited algorithmically with specialized solution methods. Structure-exploiting algorithms promise to overcome scaling issues and push current computational limits by leveraging high-performance computing (HPC), where general LP solution techniques often scale poorly. Solution techniques for AHLPs include specialized simplex methods Fourer (1982), Friedlander et al. (1990), Dantzig-Wolfe decomposition Wittrock (1985), Benders decomposition Meersman et al. (2023), Zhang et al. (2024), Lagrangian decomposition Kim et al. (2019), and interior-point methods (IPMs) Castro (2000), Grigoriadis and Khachiyan (1996), Lubin et al. (2013). IPM solvers that explicitly exploit this structure include BlockIp Castro (2016), OOPS Gondzio and Grothey (2009), PIPS-IPM Petra et al. (2014), and MadNLP Pacaud et al. (2024). While these solvers routinely operate in distributed parallel environments, presolve is typically applied as a monolithic preprocessing step prior to the parallel solution phase, often in a simplified form to avoid destroying the arrowhead structure.

All solution techniques, whether structure-exploiting or not, run more robustly and efficiently on tight formulations free of redundancies and numerical pathologies. Presolve reformulates a given model by applying a sequence of reduction techniques, generating an equivalent but simpler and smaller *presolved* model. This presolved model is then passed to the solution algorithm, solved, and in the subsequent *postsolve* step, a solution to the original problem is obtained by reverting the presolve reductions. Presolve for LP has a long history Andersen and Andersen (1995), Gondzio (1997), Achterberg et al. (2020), Gemander et al. (2020), and today, all commercial LP solvers and most academic LP solvers apply presolve before starting the solution algorithm.

Combining presolve with structured LPs and specialized solvers is challenging: presolve must preserve enough of the underlying problem structure for the solver to work efficiently. While general-purpose presolve libraries exist, most notably the academic parallel presolve library PAPILO Gleixner et al. (2023), they cannot be applied directly to AHLPs without compromising exploitable structure. Additionally, these libraries fail to efficiently leverage the arrowhead structure. This creates a clear need for presolve techniques that are both parallel and structure-aware. In this paper, we present our implementation of specialized presolve routines within the distributed parallel IPM PIPS-IPM++. Our presolve efficiently distributes reductions across compute nodes, applies presolve routines in a distributed, parallel manner, and maintains low communication overhead while preserving the structure of AHLPs.

1.1 Contribution

We make the following contributions:

- A structure-aware distributed parallel presolve and postsolve framework for AHLPs.
- An experimental evaluation demonstrating its efficiency and scalability on a set of large-scale AHLPs.
- A comparison with state-of-the-art parallel methods, including the parallel presolve library PaPILO and Gurobi’s presolve implementation.

We demonstrate the scalability of our implementation and compare its performance against PaPILO and Gurobi’s presolve implementations in terms of runtime and problem reduction. Even on a single shared-memory machine, our presolve implementation outperforms both Gurobi and PaPILO by factors of 6 and 18, respectively, while achieving similar nonzero reductions as PaPILO and 7% fewer reductions than Gurobi. Leveraging a distributed compute environment, we outperform Gurobi by a factor of 13 in presolve time.

While our presolve routines are currently implemented within PIPS-IPM++, other solution algorithms can benefit from this framework as well. Our open-source implementation is under active development and is available at GitLab.

1.1.1 Previous proceedings publication

Early experiments on a first version of our presolve were published in the conference proceedings Gleixner et al. (2020). Since the initial submission, we significantly extend our implementation and, in the process, most of it has been rewritten and redesigned. This paper extends the contributions in Gleixner et al. (2020) by:

- Providing a fully implemented linear presolve framework.
- Presenting in-depth implementation and design details of our final framework.
- Highlighting data structure and synchronization mechanisms used by our implementation.
- Providing a substantially extended computational study demonstrating the efficiency and scalability of our implementation on a large set of AHLPs.

1.2 Notation

We use $I_n \in \mathbb{N}_0^{n \times n}$ for the identity matrix of size $n \in \mathbb{N}$, often dropping the subindex n when it can be easily inferred. Zero entries in large block matrices are sometimes omitted for readability; when shown explicitly, they are denoted by 0 and their size is always inferred from the context. For a matrix $A \in \mathbb{R}^{m \times n}$, $m, n \in \mathbb{N}$, we denote by $A_i. \in \mathbb{R}^{1 \times n}$, $i \in \{1, \dots, m\}$, the i -th row vector, and by $A_{.j} \in \mathbb{R}^{m \times 1}$, $j \in \{1, \dots, n\}$, the j -th column vector.

1.3 Message Passing Interface

Our parallel implementation relies on communication between independent processes in a distributed-memory environment. We use the Message Passing Interface (MPI) Clarke et al. (1994), the de facto standard for message-passing on distributed-memory systems. MPI provides a set of collective and point-to-point communication routines for coordinating parallel processes. We refer to the processes participating in MPI communication as *MPI processes* (or simply *processes*).

MPI supports collective operations such as `Reduce` and `Allreduce`, in which a user-defined reduction operator (e.g., summation) is applied to data contributed by all processes and the result is returned to a designated root process or to all processes, respectively. In this work, we employ the blocking variants of these collective operations, implying that a process entering a collective communication call suspends execution until all participating processes have entered the same call. MPI further allows the definition of custom reduction operators, which we exploit in Algorithm 2.

We typeset names of MPI communication routines (e.g., `Reduce`, `Allreduce`) using `monospace` font.

1.4 Outline

Section 2 gives an introduction to presolving before presenting our structure-aware presolving framework. Taking the parallel constraints presolver as an example, Section 3 describes in detail the implementation of our distributed presolving techniques—highlighting synchronization and communication issues. Section 4 evaluates our framework with a set of experiments showcasing its scalability and comparing its performance with other presolve implementations. Finally, Section 5 provides conclusions and an outlook for future research.

2 Structure-aware presolving techniques

We consider linear optimization problems of the form

$$\begin{aligned} \min_x \quad & c^\top x \\ \text{subject to} \quad & Ax = b \\ & d \leq Cx \leq f \\ & l \leq x \leq u, \end{aligned} \tag{1}$$

where $n, m_A, m_C \in \mathbb{N}$, $x \in \mathbb{R}^n$ are the decision variables, $c \in \mathbb{R}^n$ is the objective vector, $A \in \mathbb{R}^{m_A \times n}$ is the equality constraint matrix, $b \in \mathbb{R}^{m_A}$ is the equality right-hand side, $C \in \mathbb{R}^{m_C \times n}$ is the inequality constraint matrix, $d, f \in (\mathbb{R} \cup \{-\infty, \infty\})^{m_C}$ are the inequality lower and upper bounds, respectively, and $l, u \in (\mathbb{R} \cup \{-\infty, \infty\})^n$ are the variable lower and upper bounds, respectively. The dual problem of Equation (1) is given by

$$\begin{aligned} \max_{y, z^+, z^-, \gamma, \phi} \quad & b^\top y + d^\top z^+ - f^\top z^- + l^\top \gamma - u^\top \phi \\ \text{subject to} \quad & A^\top y + C^\top z^+ - C^\top z^- + \gamma - \phi = c \\ & z^+ \geq 0, z^- \geq 0, \gamma \geq 0, \phi \geq 0 \\ & y \text{ free.} \end{aligned} \tag{2}$$

Here, z_i^+ , z_i^- , γ_i , and ϕ_i are fixed at zero for $d_i = -\infty$, $f_i = \infty$, $l_i = -\infty$, and $u_i = \infty$, respectively. The variables $y \in \mathbb{R}^{m_A}$, z^+ , $z^- \in \mathbb{R}_{\geq 0}^{m_C}$ are the dual variables associated with the primal equality and inequality constraints, and $\gamma, \phi \in \mathbb{R}_{\geq 0}^n$ are the variable-bound duals.

A variety of presolving techniques exist Andersen and Andersen (1995), Gondzio (1997), Achterberg et al. (2020). In our framework, these techniques are implemented as *presolvers*. Each presolver applies one reduction or a combination of similar reductions usually iterating the problem’s constraints, variables, non-zeros, or a subset of these. Presolvers are applied iteratively in *rounds* until no further reductions are possible or predefined working limits are reached. Working limits are imposed to restrict the time spent in presolve. Within our solver, we impose a maximum number of presolving rounds, and a new round is only started if a sufficient number of reductions were applied in the previous round relative to the problem size. This prevents costly scans of the entire problem when only minimal additional reductions are expected.

2.1 Presolvers in PIPS-IPM++

The presolvers implemented in PIPS-IPM++ are shown in Table 1. The presolvers EmptyVar, ForcingConstr, RedundantConstr and TinyEntries are actually part of PIPS-IPM++’s ModelCleanup presolver, but for ease of presentation of the individual techniques, we disaggregated them in Table 1. The presolver RedundantExpr runs as part of the DualTightening presolver as it uses the same detection mechanism. The presolvers we implemented are not an exhaustive list of all available presolving techniques. Rather, we implemented all presolvers that seemed general and widely adopted. Additionally, we selected presolvers that were expected to perform reductions on the models of interest, which primarily come from energy modeling contexts. We omit a detailed explanation of each presolver’s reduction technique, except for the LinDependencies and Permutation presolvers. We instead provide references to the respective papers in the “Description” column of Table 1.

All of the presolvers are called in each round of the PIPS-IPM++ presolve routine. The exceptions are the Tiny Entries presolver, which is called once at the beginning of presolve, and the LinDependencies presolver, which are called once at the end of presolve. The presolvers Permutation and LinDependencies are specifically tailored to AHLPs and will be described in more detail in Section 2.4.2 and Section 2.4.1.

Presolver	Description
Aggregation	Aggregates two variables using doubleton constraints (Gondzio (1997) 2.2)
BoundTightening	One-constraint activity-based bound tightening (Achterberg et al. (2020) 3.2)
VarsFixation	Fixes variables with (nearly) equal bounds (Achterberg et al. (2020) 4.1)
DualTightening	Dual variable fixing and bound tightening (Achterberg et al. (2020) 4.4)
EmptyVar	Removes empty variables (Andersen and Andersen (1995) 3.1 (ii))
ForcingConstr	Fixes variables in forcing constraints (Andersen and Andersen (1995) 3.3 (x))
LinDependencies	Detects linear dependencies in the equality matrix (Section 2.4.1, Gondzio (1997) 2.1 and 3.1)
ParallelVars	Detects and merges parallel variables (Andersen and Andersen (1995) 3.4)
ParallelConstrs	Detects parallel or nearly parallel constraints (Achterberg et al. (2020) 5.2)
Permutation	Improves arrowhead structure by permuting variables and constraints (Section 2.4.2)
RedundantExpr	Removes redundant expressions (Achterberg and Wunderling (2013), 4.4 (i))
RedundantConstr	Removes redundant (w.r.t. activity) constraints (Achterberg et al. (2020) 3.1)
SingletonVar	Substitutes singleton variables (Andersen and Andersen (1995) 3.2 (vi–viii))
SingletonConstr	Transforms singleton constraints into variable bounds (Andersen and Andersen (1995) 3.1 (v))
TinyEntries	Removes small entries in the constraint matrix (Achterberg et al. (2020) 3.1)

Table 1: Presolving techniques in PIPS-IPM++.

2.2 Distributed Arrowhead presolving

Within PIPS-IPM++, the LP in Equation (1) is of *arrowhead* (primal-dual block-angular) form:

$$\begin{aligned}
& \min && c_0^T x_0 + c_1^T x_1 + \cdots + c_N^T x_N \\
& \text{subject to} && A_0 x_0 & & = b_0 \\
& && d_0 \leq C_0 x_0 & & \leq f_0 \\
& && A_1 x_0 + B_1 x_1 & & = b_1 \\
& && d_1 \leq C_1 x_0 + D_1 x_1 & & \leq f_1 \\
& && \vdots & & \vdots \\
& && A_N x_0 + & & + B_N x_N = b_N \\
& && d_N \leq C_N x_0 + & & + D_N x_N \leq f_N \\
& && F_0 x_0 + F_1 x_1 + \cdots + F_N x_N & & = b_{N+1} \\
& && d_{N+1} \leq G_0 x_0 + G_1 x_1 + \cdots + G_N x_N & & \leq f_{N+1} \\
& && & & l_i \leq x_i \leq u_i \quad \forall i = 0, \dots, N.
\end{aligned} \tag{3}$$

The system matrix is split into sub-matrices $A_i \in \mathbb{R}^{m_{i_A} \times n_0}$, $B_i \in \mathbb{R}^{m_{i_A} \times n_i}$, $F_i \in \mathbb{R}^{m_{N+1_A} \times n_i}$, for the equality constraints and $C_i \in \mathbb{R}^{m_{i_C} \times n_0}$, $D_i \in \mathbb{R}^{m_{i_C} \times n_i}$, $G_i \in \mathbb{R}^{m_{N+1_C} \times n_i}$ for the inequality constraints, where $i \in \{0, \dots, N\}$. The matrices F_i , G_i correspond to the linking constraints and the variables x_0 correspond to the linking variables. We call constraints associated with $[A_i \ B_i]$ or $[C_i \ D_i]$ *local constraints*, and variables associated with x_i , $i \neq 0$ *local variables*.

PIPS-IPM++ exploits this arrowhead structure to parallelize the linear algebra within its IPM and presolving. Within its IPM, PIPS-IPM++ uses MPI (Section 1.3) to implement a Schur complement decomposition Rehfeldt et al. (2022), Kempke et al. (2024) to factorize the system matrix for each IPM iteration. To achieve this, for a given set of MPI processes (or simply processes), each diagonal block i is assigned to exactly one process along with the block's data (A_i , B_i , C_i , D_i , F_i , G_i , b_i , d_i , f_i , c_i , l_i , u_i). Additionally, each process has access to the 0-block data. This data-dependent distribution naturally limits the amount of MPI processes that can be employed to solve a problem to N , the number of diagonal blocks. Building on this distribution of data, each process performs a series of computations in parallel interleaved with serial MPI communication with the other processes. Should there be fewer processes available than blocks, multiple blocks are sequentially assigned to a single process. In the following presentation we assume that the number of processes and the number of blocks coincide.

This leads to a simple classification of presolve reductions. *Local presolve reductions*, e.g., reductions considering only local constraints and variables, can be applied independently by each process and require no communication to be detected. Reductions on linking constraints or variables, *global presolve reductions*, require communication among processes. An invariant enforced during our presolve is that no presolver may destroy the arrowhead structure, even if it could improve size or stability, to preserve compatibility with subsequent presolve rounds and PIPS-IPM++'s IPM. We also do not allow the creation of new linking constraints or linking variables, since increased coupling between blocks reduces block separability, increases MPI communication during presolve and solution,

and thereby weakens the parallel scalability of the solver and presolver, which relies on large, weakly coupled diagonal blocks.

2.3 Parallel postsolving

Applying presolve to an LP produces the, often smaller, *reduced problem*, which can subsequently be solved using any LP solver. This yields a solution to the reduced problem. To obtain a solution to the original problem, a procedure called *postsolve* is applied. Depending on the solver and solution algorithm, one might obtain a primal solution satisfying Equation (1), a dual solution satisfying Equation (2), or a primal-dual optimal solution satisfying both. Our postsolve is designed for a primal-dual IPM Wright (1997) and thus aims to recover a fully primal-dual feasible solution (including strict primal-dual complementary slackness).

Given an optimal solution to the reduced problem, postsolve reverts all reductions applied during presolve in reverse order. This is typically achieved using a stack. During presolve, whenever a reduction is applied, all data required to recover an optimal primal-dual solution during postsolve is pushed onto the stack. During postsolve, the stack is traversed in reverse order.

As mentioned previously, in our distributed parallel framework reductions may be either local or global, the latter affecting linking variables or constraints and requiring MPI communication. We maintain N stacks, one per process. During presolve, each process pushes data from local and global reductions onto the stack. During postsolve, each process then traverses its own stack reverting local and global reductions. Local reductions can typically be reverted without communicating with other processes. Global reductions, such as removing a linking constraint or variable, typically involve MPI calls during their reversion. As in our framework MPI communication calls are blocking (Section 1.3), global reductions act as synchronization points during both presolve and postsolve. In this way, postsolve is fully parallel where possible, and synchronized when necessary.

2.4 Arrowhead-specific presolving techniques

We implemented two presolvers specialized to arrowhead LPs: *Permutation* and *LinDependencies*, described below.

2.4.1 Permutation Presolver

The permutation presolver ensures that constraints and variables are in positions that are most suitable according to the arrowhead structure Equation (3). For example, an equality linking constraint in $[F_0 \ F_1 \ \dots \ F_N]$ should be non-empty in at least two F_i , $i > 0$; otherwise, it should be placed in $[A_i \ B_i]$ or A_0 . Similarly, a linking variable should be non-empty in A_i or C_i for at least two $i \in \{1, \dots, N\}$; otherwise, it belongs in the respective local block.

Constraints and variables may be improperly positioned when reading the problem from file or become improperly positioned during presolve as other constraints/variables are removed. To maintain correct constraint and variable placement, the permutation presolver is called each presolve round. The application of a permutation also simplifies the implementation of other presolve reductions as these can rely on the fact that all

Figure 2: Constraint permutations executed by the permutation presolver.

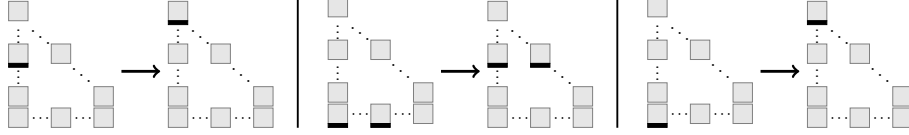
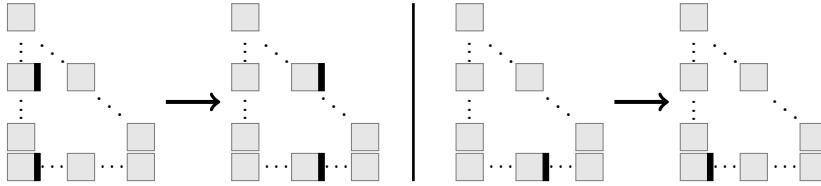


Figure 3: Variable permutations executed by the permutation presolver.



constraints and variables are correctly positioned. E.g., the `SingletonConstr` will look for singleton constraints only in the diagonal blocks, omitting all A_i , F_i , C_i , and G_i .

The constraint permutations that may be performed are displayed in Figure 2. Local constraints, if empty in B_i and D_i , are moved to A_0 and C_0 (Figure 2, left). Linking constraints can become either local constraints (Figure 2, middle), if nonzero only in F_0 , G_0 , and F_i and G_i for exactly one $i > 0$, or they are moved to the A_0 and C_0 (Figure 2, right), if they are nonzero only in F_0 and G_0 .

The possible variable permutations are shown in Figure 3. A linking variable can become a local variable (Figure 3, left), if it has nonzeros in A_0 , C_0 , and A_i , C_i for exactly one $i > 0$. Local variables, which are empty in their respective B_i and D_i (Figure 3, right) are permuted into F_0 and G_0 . This last reduction is particularly important for PIPS-IPM++, since the variables that appear only in the linking part of the problem, A_0 , F_0 , C_0 , and G_0 , receive special treatment when forming the Schur complement. This is because such variables require no additional MPI communication when forming the Schur complement and appear sparse in the Schur complement matrix. We put a longer description of this mechanism into Appendix A as this has not previously been documented in the literature.

2.4.2 Linear Dependencies Presolver

The `LinDependencies` presolver detects linearly dependent constraints in the equality matrix, ensuring full rank of the KKT system used in the IPM, which is critical for stability. Applying a full distributed Gaussian elimination for linking constraints however, would be prohibitively expensive. Instead, we detect dependencies only in the equality matrix excluding linking constraints. As such, it suffices to check that all B_i , $i = 1, \dots, N$, and A_0 are of full rank to establish that the sub-matrix

$$\begin{bmatrix} A_0 & & & & \\ A_1 & B_1 & & & \\ \vdots & & \ddots & & \\ A_N & & & B_N & \end{bmatrix}$$

is of full rank. These checks can be performed independently.

Each process i constructs its local matrix $[B_i \ I]$, with I a unit matrix of size $m_{i_A} \times m_{i_A}$, and applies a Gaussian elimination algorithm similar to MA50 Duff and Reid (1993). Pivoting is restricted to the first n_i matrix columns. After elimination, the matrix is partitioned as

$$\begin{bmatrix} \hat{B}_i & K_1 \\ 0 & K_2 \end{bmatrix}$$

where $\hat{B}_i \in \mathbb{R}^{\hat{m}_{i_A} \times n_i}$, $K_1 \in \mathbb{R}^{\hat{m}_{i_A} \times m_{i_A}}$, and $K_2 \in \mathbb{R}^{(m_{i_A} - \hat{m}_{i_A}) \times m_{i_A}}$. Each row in K_1 and K_2 represents the linear combination applied to the corresponding constraint of $[A_i \ B_i]$:

$$\begin{bmatrix} K_1 \\ K_2 \end{bmatrix} [A_i \ B_i] = P \begin{bmatrix} \hat{A}_{i1} & \hat{B}_i \\ \hat{A}_{i2} & 0 \end{bmatrix},$$

where $P \in \mathbb{N}_0^{m_{i_A} \times m_{i_A}}$ is a permutation matrix corresponding to the row/constraint permutation applied during the elimination. Constraints where \hat{A}_{i2} is empty are fully linearly dependent. Depending on the constraint right-hand sides they either prove infeasibility or are removed from the problem. Constraints where \hat{A}_{i2} is non-empty are moved to A_0 . Finally, all processes jointly apply Gaussian elimination to A_0 to detect further dependencies. The linear combinations used during elimination are stored on the stack for proper reversal during postsolve.

3 Algorithms and software design

The communication of data during an individual presolver is critical for the efficiency of the presolve procedure. Data is communicated both during the presolve and postsolve procedure to share local information to all processes. This is particularly important when performing pre- and postsolve procedures on linking constraints and variables. This section will first describe the communication of data in pre- and postsolve. A description of the parallel constraints presolver is then provided to illustrate the required communication among processes and the differences between local and global reductions.

3.1 Data communication during presolve

Communication during our presolve happens in two ways: during the application of a presolver, by reducing, e.g., communicating a hash as in Algorithm 3, or by periodically communicating problem characteristics as explained in the following. To apply our presolve efficiently without recomputing this data frequently, our implementation keeps track of the nonzeros of constraints and variables in the problem as well as the constraint activities, where the minimum (actmin) and maximum activity (actmax) of an (equality) constraint i in Equation (1) is given as

$$\begin{aligned} \text{actmin}_i &= \min_{l \leq x \leq u} A_i^T x = \sum_{a_{ij} > 0} l_i x_i + \sum_{a_{ij} < 0} u_i x_i, \\ \text{actmax}_i &= \max_{l \leq x \leq u} A_i^T x = \sum_{a_{ij} < 0} l_i x_i + \sum_{a_{ij} > 0} u_i x_i. \end{aligned}$$

Actmin and actmax are defined equivalently for inequality constraints. Should any of the variables used in either of the activity formulas be unbounded, the respective activity is set to $\pm\text{inf}$. Whenever a change in the problem occurs, these quantities get updated.

For linking variables and constraints, we cannot directly update the quantities, as changes might result from local reductions, known only to a single process. Rather, such changes are buffered and communicated periodically among all processes.

3.2 Data communication during postsolve

During postsolve, local reversions of presolve transformations must be consistent with globally shared quantities, such as dual variables and activities associated with linking constraints and variables. This requires MPI communication to ensure that all processes maintain a consistent view of these global quantities after local postsolve operations have been applied.

Analogous to presolve, postsolve communication occurs in two ways. First, communication is required during a postsolve operation whenever a quantity that depends on distributed data must be evaluated, for example when computing the activity of a linking constraint. Second, communication is required to synchronize buffered changes that originate from local postsolve operations and affect globally shared data.

The latter case arises, for instance, when a dual value is shifted between a constraint and a variable to revert a bound tightening identified during presolve. If the constraint is local but affects the bound of a linking variable, the corresponding dual of the linking variable is modified by a local reduction. Reverting this bound tightening on a single process leaves the dual values of the linking variable outdated on all other processes. To address this, we employ a buffering mechanism analogous to the one used for tracking global data during presolve: instead of immediately applying changes to globally required data (e.g., duals of linking constraints or variables), local modifications are buffered.

During presolve, *synchronization events* are recorded on the stack of each process whenever such globally relevant updates are deferred. In postsolve, after the corresponding local reversions have been applied, the stored synchronization event is executed and all buffered local changes are communicated to restore global consistency.

3.3 Parallel constraint detection

The parallel constraint detection in PIPS-IPM++ works similar to the one described in Achterberg et al. (2020). We apply a two level hashing algorithm to first identify constraints with the same support and later, within each hash bucket of constraints with the same support, identify constraints with the same coefficients. We do temporarily remove singleton variables from the constraints to be able to identify nearly parallel constraint as well. However, for the ease of presentation we omit the details on nearly parallel constraint in our description of the algorithm. For readers interested in the specific implementation details, we suggest looking at our GitLab repository.

As is usually the case in our framework, there are two different implementations for the reduction procedure, one for local constraints and one for global/linking constraints. In Algorithm 1 we give pseudocode

Algorithm 1 Parallel Constraint Reduction (Local)

Require: $[A_i \ B_i], [C_i \ D_i]$

- 1: Compute support hashes H_r , sort with (inverse) permutation π
- 2: $i \leftarrow 1$
- 3: **while** $i \leq n$ **do**
- 4: $j \leftarrow i$
- 5: **while** $j < n$ **and** $H_r[j] = H_r[j + 1]$ **do** $j \leftarrow j + 1$ \triangleright Detect bucket with equal support hash
- 6: **end while**
- 7: Compute coefficient hashes H_c for constraints $\pi(i), \dots, \pi(j)$
- 8: **for all** pairs $(\pi(l), \pi(k))$ with $i \leq l < k \leq j$ **do** \triangleright Process all pairs in bucket
- 9: **if** constraint $\pi(l), \pi(k)$ not removed **and** $H_c[\pi(l)] = H_c[\pi(k)]$ **then**
- 10: Reduce parallel constraint $\pi(l), \pi(k)$ \triangleright Eliminates one of the constraints
- 11: **end if**
- 12: **end for**
- 13: $i \leftarrow j + 1$
- 14: **end while**

for the algorithmic implementation of the detection of local parallel constraints. We first hash the constraints of a given matrix pair $[A_i \ B_i]$ and $[C_i \ D_i]$ with respect to its support and sort these hashes (Line 1). We then iterate all hashes and determine **buckets** of equal hashes (Line 5). For each bucket, its constraint's coefficients are hashed (Line 7), normalizing with the first coefficient of each constraint. We then quadratically compare all pairs of constraints in a bucket using their coefficient hashes (Line 8). In case we find parallel constraints, we remove one of them (Line 10) and continue our search in the bucket. Removing local constraints might potentially change the nonzeros in linking variables. As we use this information when removing parallel linking constraints (to detect singleton variables), we need to communicate the nonzero changes before beginning to detect parallel global constraints. Note, that we do not try to detect parallel constraints between different blocks, or parallel local and global constraints. Instead, we rely on the permutation presolver to first put these constraints into their correct positions (see Section 2.4.1).

Extending this detection for constraints in the A_0 and C_0 blocks is straightforward. However, linking constraints require communication among the processes. In Algorithm 2 we show pseudocode of our parallel linking constraint detection. The general structure of the algorithm is similar to Algorithm 1. First each process computes the *local* support hashes with respect to its local blocks F_i and G_i (Line 1). All processes then **Allreduce** their local hashes using a custom MPI operator (see Section 1.3) to form *global* support hashes. These global support hashes are sorted (Line 3) to aid in finding buckets of constraints with identical support hashes (Line 7). To determine whether constraints in buckets are actually globally parallel, each process computes for each bucket the local coefficient hashes of each constraint in the bucket (Line 9). Then each pair within the bucket is compared using the coefficient hash to determine whether the pair is parallel locally (Line 11). This information is stored in an array of truth values P and communicated using an **Allreduce** call

Algorithm 2 Parallel Constraint Detection (Global Linking)

Require: Matrices F_0, G_0 and all locally available F_i and G_i to detect parallel constraints

- 1: Compute local constraint support hashes H_r^{loc} using F_i and G_i
- 2: **Custom-Allreduce** H_r^{loc} to global hashes H_r ; append F_0 and G_0 to hashes
- 3: Sort hashes with (inverse) permutation π
- 4: Init array P ; $i \leftarrow 1$
- 5: **while** $i \leq N$ **do** ▷ Local parallel constraints detection
- 6: $j \leftarrow i$
- 7: **while** $j < N$ **and** $H_r[j] = H_r[j + 1]$ **do** $j \leftarrow j + 1$ ▷ Detect bucket with equal support hash
- 8: **end while**
- 9: Compute local coefficient hashes H_c^{loc} for constraints $\pi(i), \dots, \pi(j)$
- 10: **for all** pairs $(\pi(l), \pi(k))$ with $i \leq l < k \leq j$ **do** ▷ Process all pairs in bucket
- 11: $P[(\pi(l), \pi(k))] \leftarrow (H_c^{\text{loc}}[\pi(l)] = H_c^{\text{loc}}[\pi(k)])$ ▷ Store local result
- 12: **end for**
- 13: $i \leftarrow j + 1$
- 14: **end while**
- 15: **AND-Allreduce** P ; $i \leftarrow 1$
- 16: **while** $i \leq N$ **do** ▷ Apply reductions
- 17: $j \leftarrow i$
- 18: **while** $j < N$ **and** $H_r[j] = H_r[j + 1]$ **do** $j \leftarrow j + 1$
- 19: **end while**
- 20: **for all** pairs $(\pi(l), \pi(k))$ with $i \leq l < k \leq j$ **do** ▷ Process all pairs a second time
- 21: **if** constraints $\pi(l), \pi(k)$ not removed **and** $P[(\pi(l), \pi(k))]$ **then**
- 22: Reduce parallel constraints $\pi(l), \pi(k)$ ▷ Eliminates one of the constraints
- 23: **end if**
- 24: **end for**
- 25: $i \leftarrow j + 1$
- 26: **end while**

combined with the MPI logical AND operator (Line 15). In a second loop over each bucket and pair (buckets are not actually recomputed in Line 20 but buffered from the first loop), each process can finally apply parallel constraint reductions (Line 22).

In contrast to the purely local case, preprocessing parallel linking constraints requires explicit coordination across processes to ensure that reductions are applied consistently with respect to the distributed data. While local parallel constraint detection can be carried out independently within each block, the detection of parallel linking constraints requires structural information (support hashes), and local decisions (local parallelism stored in P) to be communicated via MPI. Managing this communication requires careful design and implementation to balance the trade-off between communication overhead and reduction strength, occasionally weakening presolve to limit overhead.

4 Computational Experiments

We conducted three experiments to evaluate our presolve implementation. In Section 4.1, we assess the scalability of our presolve on six different instances taken from our testset of AHLPS. In Section 4.2 we compare our implementation with respect to runtime and number of reductions with the presolve implementations of Gurobi and PaPILO. Finally, in Section 4.3 we evaluate the impact of our presolving on running the IPM implemented in PIPS-IPM++. For the evaluation of our experiments, we generally use the shifted geometric mean to lessen the impact of extreme outliers, using a shift of one second and a shift of one percent for the respective aggregated results.

Testset PIPS-IPM++, and thus our presolve implementation, can currently only be called via a GAMS¹ interface. This interface requires models to be annotated to assign variables and constraints to an arrowhead form (see Equation (3)). Our model library contains 81 instances of varying sizes, most of which come from the energy system context. We note that not all models in our testset expose a block structure that is favorable towards PIPS-IPM++’s Schur complement approach. As such, PIPS-IPM++ does not always outperform commercial solvers such as Gurobi on our testset. We have also used instances that PIPS-IPM++ cannot solve at all due to an overly large Schur complement. Still, we included these models to get a broad assessment of our presolve implementation. Whenever we run Gurobi and PaPILO on instances from our testset, we first converted the PIPS-IPM++ GAMS files to mps files. We display sizes and number of blocks of all instances in Appendix B, Table 5.

Hardware All experiments were conducted on the terrabyte supercomputer of the Leibniz Supercomputing Centre. Each node is equipped with 1024 GB RAM and two Intel Xeon Platinum 8380 CPUs at 2.3 GHz 40 cores each. All jobs always allocated full nodes, so no concurrent execution was permitted.

Software We used PaPILO 3.0.1, Gurobi Optimizer 13.0.0 and PIPS-IPM++, git hash 0587dbe3, for our experiments.

4.1 Scaling

First, we demonstrate the scaling of our presolve routines on six selected instances. We chose a subset of large instances from our testset and ran each of them with Gurobi, PaPILO and our presolve with different numbers of processes. The maximum number of processes used for PaPILO and Gurobi was 128, the maximum number that, allowing for hyperthreading, could still be run on a single compute node. PIPS-IPM++ was run with one core per process, and the maximum number of processes depends on the number of blocks in the AHLPS. In Table 2 we give size and block count of these instances, as well as nonzero reduction during presolve in percent for each presolve implementation. We give a more detailed analysis of the relative reduction performed on each instance in Section 4.2. For the instances used for our scaling experiments, Gurobi performs the most reductions, removing 3-13% more non-zeros than PaPILO and our

¹<https://www.gams.com>

presolve. PaPILO outperforms both PIPS-IPM++’s presolve and Gurobi on the ELMOD_4 instance. On Simple_1 and Simple_2, PIPS-IPM++ and PaPILO perform a similar amount of reductions. PaPILO was unable to presolve ELMOD_5, YSSP_exp_1, and YSSP_exp_2 within one hour.

Table 2: Instances and results for scaling experiment.

Instance	# blocks	# nonzeros	# constraints	# variables	% nonzeros reduced problem		
					PIPS-IPM++	Gurobi	PaPILO
Simple_1	1024	205 569 328	51 604 093	59 795 108	81.22	78.07	81.24
Simple_2	438	207 679 112	52 193 591	57 444 984	91.29	87.35	91.36
ELMOD_4	438	271 875 064	98 646 274	85 646 554	51.33	44.73	39.34
ELMOD_5	438	711 769 260	254 304 960	224 677 685	58.39	44.64	Timeout
YSSP_exp_1	96	316 863 066	96 851 394	110 650 876	54.07	43.46	Timeout
YSSP_exp_2	250	247 383 863	73 253 433	87 054 963	69.15	55.67	Timeout

In Figure 4 we depict the results of our scaling experiments for each instance. We plot for each instance the presolve time taken by PIPS-IPM++, Gurobi, and PaPILO against the number of MPI processes (for PIPS-IPM++)/number of threads (for Gurobi and PaPILO) available. We use a logarithmic scaling of the time axis and supply, as a reference, the optimal linear speed-up and the optimal speed-up for a program running with five percent sequential code according to Amdahls’s law. Neither Gurobi nor PaPILO scale well on the instances given achieving close to no speed-up for any combination of threads. On the other hand, PIPS-IPM++ performs worse than Gurobi and PaPILO when using only one thread, a fact that we attribute to the additional overhead required by the parallel presolve implementation as well as the AHL P treatment within our software. For the six given instances, the breakeven point of PIPS-IPM++’s presolve time and Gurobi and PaPILO’s lies mostly at two, for the two Simple models towards four processes. Our presolve’s scaling behavior lies somewhere below the optimal speedup for a program executing five percent sequential code. We note that the amount of sequential processing within PIPS-IPM++ strongly depends on the final structure of Equation (3). More linking variables and constraints correlate with a larger amount of sequential presolve where a fully block-diagonal matrix should scale linearly, as no communication within PIPS-IPM would be required. Lastly, the cost of communication does not grow linearly within PIPS-IPM++. While communication within a single CPU (1 to 32 processes) is cheapest, a first increase in cost per MPI operation occurs when going from one to two CPUs (32 to 64 processes) and an even steeper cost increase occurs going from one to multiple compute nodes (64 to 128 and more processes). Overall, PIPS-IPM++’s presolve implementation scales well with the number of available processes.

4.2 Comparison against other presolvers

To evaluate the efficiency and efficacy of our implementation we compare the amount of problem reductions (removed nonzeros, constraints, and variables) and execution time of our presolve against the presolve of Gurobi and PaPILO. We ran PIPS-IPM++ using one node with 64 processes and the maximum possible amount of processes using multiple nodes. While PaPILO and Gurobi are each run with 64 threads on one node. A timelimit of 1 hour is used for these experiments. Table 3 shows the number of instances where the full presolving procedure completes

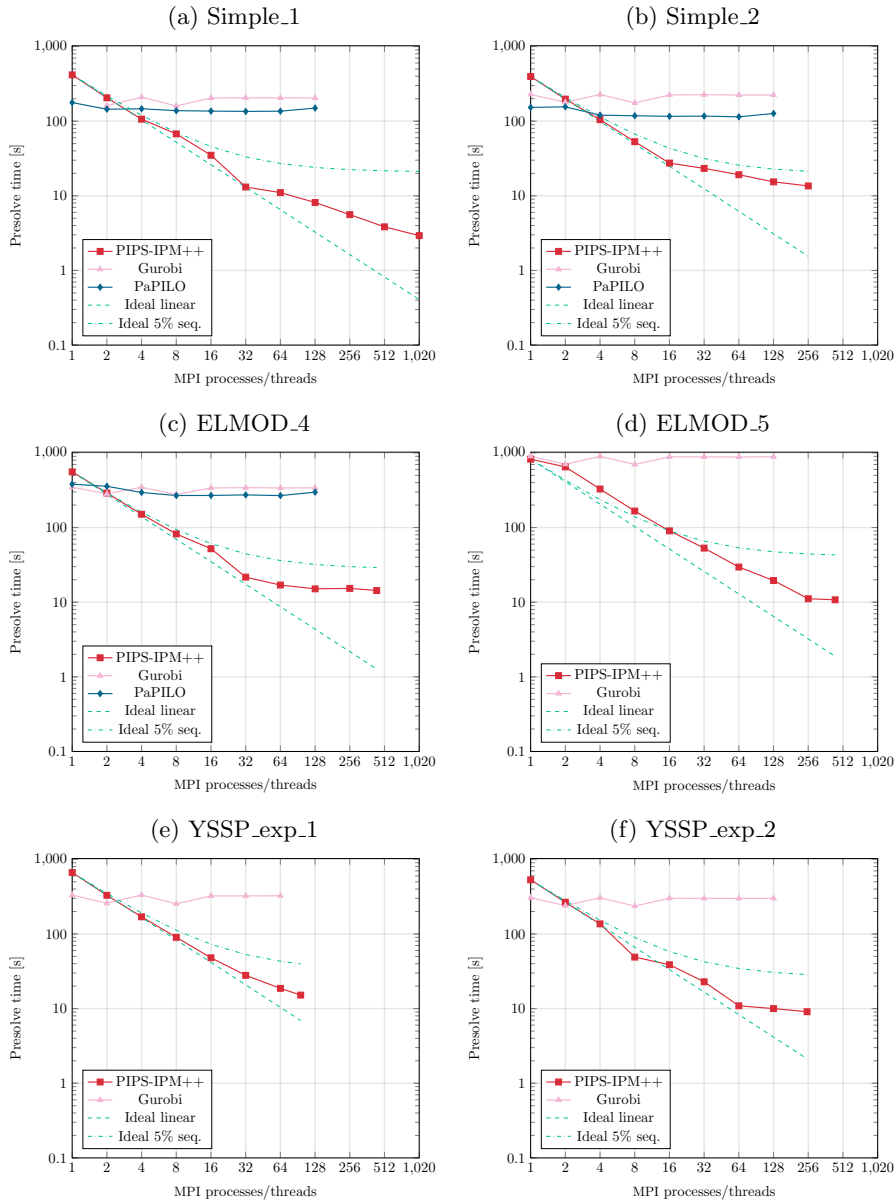


Figure 4: Presolve scaling behavior of PIPS-IPM++, PaPILO, and Gurobi.

within the timelimit, the time taken to complete the presolve procedure (sgm with shift of one second) and the reduced problem’s size relative to the original problem (sgm with shift of one percent).

Table 3: Presolve results of PIPS-IPM++, Gurobi, and PaPILO (sgm 1s/1%).

Presolver	# success	# timeout	time (s)	reduced problem size		
				nonzeros (%)	constraints (%)	variables (%)
PIPS-IPM++ (max proc.)	82	0	2.09	80.21	83.05	89.86
PIPS-IPM++ (64 proc.)	82	0	4.39	79.25	81.89	89.02
Gurobi	81	1	26.85	73.37	72.82	79.77
PaPILO	70	12	72.73	78.97	85.95	88.20

We first note that PIPS-IPM++’ presolve implementation is not deterministic. This is the result of different reduction orderings within MPI calls, depending on the dynamic allocation of machines and processes. While a deterministic order can be enforced, this usually leads to a decrease in performance, especially in distributed environments. This can be seen by the slight differences in problem reduction performed by our presolve using 64 and all possible processes. It is not generally true that a higher processor count leads to less reductions, though the two individual experiments indicate this is the case. While PaPILO struggles to presolve all instances successfully, Gurobi only runs into the timelimit on one instance; PIPS-IPM++ can presolve all instances successfully. Doing so, the presolve routines of PIPS-IPM++ using 64 processes are about a factor of six faster than Gurobi and a factor of 16 faster than PaPILO (in sgm). Using all possible processes this gap increases and PIPS-IPM++ is faster by about a factor of 13 when compared to Gurobi. Looking at the amount of nonzeros left in the problem after presolve—the most important metric for the performance of Simplex, IPM, and PDLP algorithms—we see that PIPS-IPM++ reduced the problems by a similar amount as PaPILO to roughly 80% of the problems original nonzero counts, PaPILO removing slightly more nonzeros than PIPS-IPM++. PIPS-IPM++ removes more constraints but less variables than PaPILO. This result can be explained by considering the variable aggregation presolver in PIPS-IPM++, which is one of the most effective presolve methods for non-trivial removal of variables. This presolver in PIPS-IPM++ is restricted by the requirement to preserve the arrowhead structure. As a result, PIPS-IPM++ is expected to remove less variables on average than the other presolve implementations. Gurobi dominates the results in terms of removed nonzeros, constraints, and variables, deleting five percent more nonzeros, nine percent more constraints and nearly eleven percent more variables than PIPS-IPM++. This advantage is somewhat expected, since Gurobi is a commercial code and has a long development history. However, it is encouraging that PaPILO and PIPS-IPM++ only fall short by about five to six percent in terms of nonzeros removed.

Overall, we see a strong runtime advantage of PIPS-IPM++ over both the academic and the commercial alternatives. Already on a single machine, our presolve implementation can significantly outperform even the state-of-the-art commercial solver Gurobi in terms of runtime, while presolving the problem to a similar extent as PaPILO.

4.3 Presolve impact in PIPS-IPM++

To investigate the impact of presolving on solving the selected models, we chose a subset of our models that either Gurobi or PIPS-IPM++ could solve within one hour. This selected resulted in a total of 67 models. We ran each of these models with PIPS-IPM++, setting a timelimit of 1 hour, with and without presolving enabled. Table 4 show the results of these experiments, comparing the number of successfully solved instances with and without presolve. We display the sgm (shift of one second) of all models solved and of all models. Models that ran into the time limit contributed 3600 seconds to “time all (s)”. The impact of presolving for

Table 4: PIPS-IPM++ performance with and without presolve.

Solver	presolve			No presolve		
	# solved	time solved (s)	time all (s)	# solved	time solved (s)	time all (s)
PIPS-IPM++	44	179.9	410.48	34	197.45	594.04

PIPS-IPM++ is significant. Within the timelimit, PIPS-IPM++ run with presolving solves 10 instances more, and the overall runtime decreases by over 30%. This highlights the importance of presolve within PIPS-IPM++. We also believe that other AHLP exploiting methods can benefit similarly from our contribution.

5 Conclusion

We presented and evaluated our distributed parallel, structure preserving presolve implemented in the parallel solver PIPS-IPM++. We demonstrated its scalability and general effectiveness on a large set of AHLPs showing that already on 64 threads our presolve outperforms Gurobi by a factor of six and PaPILO by a factor of 16 while performing similarly to PaPILO in terms of nonzero reductions applied to the problem. These speedups increase to factors of 13 and 36, respectively, when deploying our presolve in a distributed compute environment. We demonstrated that structure specific implementations of algorithms can help speed-up the solution process.

For our presolve implementation we mainly see two ways forward. First, make the implementation easily accessible to other structure exploiting methods that work on block structured LPs. Second, improve presolve further by adding more presolvers and tuning. Our experiments comparing against Gurobi indicate a gap in algorithmic presolve that we believe is not simply explained by block-structure preservation restrictions.

PIPS-IPM++ is being actively developed, with our focus currently lying on its accessibility, automated block structure detection and the improvement of its IPM implementation. Our code can be found on GitLab² and we encourage contributions.

²<https://gitlab.com/pips-ipmpp/pips-ipmpp>

B Instance sizes of testset

Table 5: Model sizes and scenarios

Name	Nonzeros	Columns	Rows	Scenarios
50hz	5,354,356	1,485,281	2,018,016	32
DLR_c1	2,891,033	814,785	770,996	120
DLR_c5	11,432,556	3,372,987	3,189,050	120
weather	128,229	35,380	35,720	4
yssp_1	69,048,687	25,633,269	21,944,574	250
yssp_2	27,927,785	9,478,844	8,532,569	1,460
yssp_3	80,669,654	25,373,033	28,333,877	96
yssp_4	86,328,614	25,373,033	29,034,677	96
yssp_5	89,659,570	32,272,423	28,584,269	288
yssp_6	98,244,854	34,462,665	31,738,111	88
remix_carbonbudget_1	3,040,418	885,037	990,163	24
remix_carbonbudget_2	3,040,418	885,037	990,163	24
remix_carbonbudget_3	3,793,885	885,039	990,164	24
remix_carbonbudget_4	3,793,885	885,039	990,164	24
remix_carbonbudget_5	3,909,783	884,963	990,089	24
remix_carbonbudget_6	3,909,783	884,963	990,089	24
remix_nagsys_1	6,190,830	1,394,571	1,431,256	120
remix_nagsys_2	6,092,056	1,394,188	1,431,100	120
remix_nagsys_3	66,684,676	14,845,262	16,178,512	120
remix_nagsys_4	71,467,522	15,808,845	16,704,135	120
remix_nagsys_5	71,467,522	15,808,845	16,704,135	60
remix_nagsys_6	70,941,922	15,546,045	16,441,335	120
remix_nagsys_7	70,941,922	15,546,045	16,441,335	60
remix_nagsys_8	22,486,172	5,013,276	5,458,440	120
dtu_1	2,583,153	794,461	847,809	32
dtu_2	75,437,128	26,586,252	20,652,148	104
eth_1	3,076,103	903,138	946,791	28
eth_2	1,942,667	499,122	552,239	28
eth_3	1,942,667	499,122	552,239	28
miso_1	89,659,570	32,272,423	28,584,269	144
museko_1	93,647,334	27,433,780	30,762,491	240
museko_2	93,647,334	27,433,780	30,762,491	240
museko_3	14,114,652	4,381,099	5,432,167	240
museko_4	5,406,200	1,524,618	1,682,288	240
psi	460,037,503	19,195,771	28,635,704	70
tud_set3	143,015,695	44,693,530	51,228,474	438
tud_set4_cwe14	271,621,021	85,585,234	98,532,392	438
tud_set4_cwe15	271,875,064	85,646,554	98,646,274	438
tud_set4_cwe16	272,602,144	85,883,074	98,909,074	438
tud_set5	711,769,260	224,677,685	254,304,960	438
simple1	207,679,112	57,444,984	52,193,591	438
simple2	103,223,688	28,593,472	25,965,179	438
disp_1	27,927,785	9,478,844	8,532,569	250
disp_2	185,938,670	74,465,252	60,665,309	1,460
disp_3	185,938,670	74,465,252	60,665,309	250
disp_4	68,856,337	25,440,919	21,944,574	250
disp_5	27,927,785	9,478,844	8,532,569	250

Table 5: Model sizes and scenarios (continued)

Name	Nonzeros	Columns	Rows	Scenarios
disp_6	183,106,786	71,633,368	60,665,309	250
exp_1	316,863,066	110,650,876	96,851,394	96
exp_2	85,255,376	28,787,314	25,098,178	1,460
exp_3	85,255,376	28,787,314	25,098,178	250
exp_4	32,185,332	10,267,367	9,320,973	1,460
exp_5	32,185,332	10,267,367	9,320,973	250
exp_6	247,383,863	87,054,963	73,253,433	1,460
exp_7	247,383,863	87,054,963	73,253,433	250
exp_lim_1	84,887,786	28,594,944	25,098,178	250
exp_lim_2	32,185,332	10,267,367	9,320,973	250
exp_lim_3	241,835,759	84,222,769	73,253,433	250
esom_1	34,325,147	13,613,490	11,142,720	32
esom_2	10,675,630	4,660,465	3,495,240	32
tb_100_10	39,106,660	10,854,122	9,830,197	1,024
tb_100_5	15,027,820	4,301,305	3,789,354	1,024
tb_200_10	78,212,296	21,707,144	19,659,349	1,024
tb_25_10	9,777,529	2,714,382	2,458,357	1,024
tb_25_20	19,932,047	5,530,132	5,018,077	1,024
tb_25_40	39,926,301	11,059,240	10,035,129	1,024
tb_400_10	156,423,568	43,413,188	39,317,653	1,024
tb_400_10	205,569,328	59,795,108	51,604,093	1,024
tb_50_10	19,554,034	5,427,664	4,915,669	1,024
tb_50_20	39,863,070	11,059,084	10,035,089	1,024
tb_51_20	20,298,692	5,631,788	5,110,273	512
tb_23_40	36,899,226	10,219,005	9,272,694	1,029
tb_3_60	20,935,670	5,785,000	5,259,041	2,920
artesis	7,532,720	2,383,344	2,102,960	12
simple_rt	1,047,769	298,227	263,174	60
simple_t	1,047,769	298,227	263,174	60
spde_1	5,407,085	1,489,282	1,314,105	120
spde_2	5,407,085	1,489,282	1,314,105	120
spde_3	5,407,085	1,489,282	1,314,105	120
spde_4	5,407,085	1,489,282	1,314,105	120
spde_weather	3,244,251	893,578	788,463	30

Acknowledgements

The work for this article has been conducted in the Research Campus MODAL funded by the German Federal Ministry of Research, Technology and Space (BMFT) (fund numbers 05M14ZAM, 05M20ZBM, 05M2025). Further, the described research activities were funded by the German Federal Ministry for Economic Affairs and Energy (BMWi) within the project PEREGRINE (grant number 03EI1082B). The authors gratefully acknowledge the computational and data resources provided by the Leibniz Supercomputing Centre (www.lrz.de).

References

- Achterberg T, Bixby RE, Gu Z, Rothberg E, Weninger D (2020) Presolve reductions in mixed integer programming. *INFORMS Journal on Computing* 32(2):473–506, ISSN 1526-5528, URL <http://dx.doi.org/10.1287/ijoc.2018.0857>.
- Achterberg T, Wunderling R (2013) *Mixed Integer Programming: Analyzing 12 Years of Progress*, 449–481 (Berlin, Heidelberg: Springer Berlin Heidelberg), ISBN 978-3-642-38189-8, URL http://dx.doi.org/10.1007/978-3-642-38189-8_18.
- Andersen ED, Andersen KD (1995) Presolving in linear programming. *Mathematical Programming* 71(2):221–245, URL <http://dx.doi.org/10.1007/BF01586000>.
- Castro J (2000) A Specialized Interior-Point Algorithm for Multicommodity Network Flows. *SIAM J. Optim.* 10(3):852–877, URL <http://dx.doi.org/10.1137/S1052623498341879>.
- Castro J (2016) Interior-point solver for convex separable block-angular problems. *Optim. Methods Softw.* 31(1):88–109, URL <http://dx.doi.org/10.1080/10556788.2015.1050014>.
- Castro J, Escudero LF, Monge JF (2023) On solving large-scale multistage stochastic optimization problems with a new specialized interior-point approach. *European J. Oper. Res* 310(1):268–285, ISSN 0377-2217, URL <http://dx.doi.org/10.1016/j.ejor.2023.03.042>.
- Clarke L, Glendinning I, Hempel R (1994) The MPI Message Passing Interface Standard. *Programming Environments for Massively Parallel Distributed Systems*, 213–218, ISBN 978-3-0348-8534-8, URL http://dx.doi.org/10.1007/978-3-0348-8534-8_21.
- Colombo M, Gondzio J, Grothey A (2011) A warm-start approach for large-scale stochastic linear programs. *Math. Program.* 127(2):371–397, URL <http://dx.doi.org/10.1007/s10107-009-0290-9>.
- Duff IS, Reid J (1993) MA48: A FORTRAN code for direct solution of sparse unsymmetric linear systems of equations. *Rutherford Appleton Laboratory Technical Reports* URL MA48: [AFORTRANcodefordirectsolutionofsparselinearsystemsofequations](http://dx.doi.org/10.1007/978-3-0348-8534-8_21).
- Fourer R (1982) Solving staircase linear programs by the simplex method, 1: Inversion. *Math. Program.* 23(1):274–313, URL <http://dx.doi.org/10.1007/BF01583795>.
- Friedlander A, Lyra C, Tavares H, Medina E (1990) Optimization with staircase structure: An application to generation scheduling. *Comput. Oper. Res.* 17(2):143–152, ISSN 0305-0548, URL [http://dx.doi.org/10.1016/0305-0548\(90\)90038-9](http://dx.doi.org/10.1016/0305-0548(90)90038-9).
- Gemander P, Chen WK, Weninger D, Gottwald L, Gleixner A, Martin A (2020) Two-row and two-column mixed-integer presolve using hashing-based pairing methods. *EURO Journal on Computational Optimization* 8(3):205–240, ISSN 2192-4406, URL <http://dx.doi.org/10.1007/s13675-020-00129-6>.
- Gils HC, Scholz Y, Pregger T, Luca de Tena D, Heide D (2017) Integrated modelling of variable renewable energy-based power supply in Europe. *Energy* 123:173–188, ISSN 0360-5442, URL <http://dx.doi.org/10.1016/j.energy.2017.01.115>.
- Gleixner A, Gottwald L, Hoen A (2023) PaPILO: A Parallel Presolving Library for Integer and Linear Optimization with Multiprecision

- Support. *INFORMS Journal on Computing* 35(6):1329–1341, ISSN 1526-5528, URL <http://dx.doi.org/10.1287/ijoc.2022.0171>.
- Gleixner A, Kempke NC, Koch T, Rehfeldt D, Uslu S (2020) First Experiments with Structure-Aware Presolving for a Parallel Interior-Point Method. *Oper. Res. Proc.*, 105–111 (Springer), URL http://dx.doi.org/10.1007/978-3-030-48439-2_13.
- Gondzio J (1997) Presolve analysis of linear programs prior to applying an interior point method. *INFORMS Journal on Computing* 9(1):73–91, URL <http://dx.doi.org/10.1287/ijoc.9.1.73>.
- Gondzio J, Grothey A (2006) Solving Distribution Planning Problems with the Interior Point Method. Technical report, The University of Edinburgh, URL <https://webhomes.maths.ed.ac.uk/~gondzio/reports/oopsDPP.pdf>.
- Gondzio J, Grothey A (2007) Solving non-linear portfolio optimization problems with the primal-dual interior point method. *European J. Oper. Res* 181(3):1019–1029, ISSN 0377-2217, URL <http://dx.doi.org/10.1016/j.ejor.2006.03.006>.
- Gondzio J, Grothey A (2009) Exploiting structure in parallel implementation of interior point methods for optimization. *Comput. Manag. Sci.* 135–160, URL <http://dx.doi.org/10.1007/s10287-008-0090-3>.
- Grigoriadis MD, Khachiyan LG (1996) An Interior Point Method for Bordered Block-Diagonal Linear Programs. *SIAM J. Optim.* 6(4):913–932, URL <http://dx.doi.org/10.1137/S1052623494263609>.
- Göke L (2021) A graph-based formulation for modeling macro-energy systems. *Applied Energy* 301:117377, ISSN 0306-2619, URL <http://dx.doi.org/10.1016/j.apenergy.2021.117377>.
- Hörsch J, Hofmann F, Schlachtberger D, Brown T (2018) PyPSA-Eur: An open optimisation model of the European transmission system. *Energy Strategy Reviews* 22:207–215, URL <http://dx.doi.org/10.1016/j.esr.2018.08.012>.
- Kempke NC, Rehfeldt D, Koch T (2024) A massively parallel interior-point-method for arrowhead linear programs. URL <http://dx.doi.org/10.48550/arXiv.2412.07731>.
- Kim K, Petra CG, Zavala VM (2019) An Asynchronous Bundle-Trust-Region Method for Dual Decomposition of Stochastic Mixed-Integer Programming. *SIAM J. Optim.* 29(1):318–342, URL <http://dx.doi.org/10.1137/17M1148189>.
- Lubin M, Martin K, Petra CG, Sandıkçı B (2013) On parallelizing dual decomposition in stochastic integer programming. *Oper. Res. Lett.* 41(3):252–258, ISSN 0167-6377, URL <http://dx.doi.org/10.1016/j.orl.2013.02.003>.
- Meersman T, Maenhout B, Van Herck K (2023) A nested Benders decomposition-based algorithm to solve the three-stage stochastic optimisation problem modeling population-based breast cancer screening. *European J. Oper. Res* 310(3):1273–1293, ISSN 0377-2217, URL <http://dx.doi.org/10.1016/j.ejor.2023.04.027>.
- Pacaud F, Schanen M, Shin S, Maldonado DA, Anitescu M (2024) Parallel interior-point solver for block-structured nonlinear programs on SIMD/GPU architectures. *Optim. Methods Softw.* 39(4):874–897, URL <http://dx.doi.org/10.1080/10556788.2024.2329646>.

- Petra CG, Schenk O, Lubin M, Gärtner K (2014) An Augmented Incomplete Factorization Approach for Computing the Schur Complement in Stochastic Optimization. *SIAM J. Sci. Comput.* 36(2):C139–C162, URL <http://dx.doi.org/10.1137/130908737>.
- Rao CV, Wright SJ, Rawlings JB (1998) Application of Interior-Point Methods to Model Predictive Control. *J. Optim. Theory Appl.* 99(3):723–757, ISSN 1573-2878, URL <http://dx.doi.org/10.1023/a:1021711402723>.
- Rehfeldt D, Hobbie H, Schönheit D, Gleixner A, Koch T, Möst D (2022) A massively parallel interior-point solver for LPs with generalized arrowhead structure, and applications to energy system models. *European J. Oper. Res.* 296(1):60–71, URL <http://dx.doi.org/10.1016/j.ejor.2021.06.063>.
- Steinbach MC (2001) *Hierarchical Sparsity in Multistage Stochastic Programs*, 385–410 (Boston, MA: Springer US), ISBN 978-1-4757-6594-6, URL http://dx.doi.org/10.1007/978-1-4757-6594-6_16.
- Wetzel M, Gils HC, Bertsch V (2023) Green energy carriers and energy sovereignty in a climate neutral European energy system. *Renewable Energy* 210:591–603, ISSN 0960-1481, URL <http://dx.doi.org/10.1016/j.renene.2023.04.015>.
- Wiese F, Bramstoft R, Koduvere H, Pizarro Alonso A, Balyk O, Kirkerud JG, Åsa Grytli Tveten, Bolkesjø TF, Münster M, Ravn H (2018) Balmorel open source energy system model. *Energy Strategy Reviews* 20:26–34, ISSN 2211-467X, URL <http://dx.doi.org/10.1016/j.esr.2018.01.003>.
- Wittrock RJ (1985) *Dual nested decomposition of staircase linear programs*, 65–86 (Springer Berlin Heidelberg), ISBN 9783642009198, URL <http://dx.doi.org/10.1007/bfb0121043>.
- Word DP, Kang J, Akesson J, Laird CD (2014) Efficient parallel solution of large-scale nonlinear dynamic optimization problems. *Comput. Optim. Appl.* 59(3):667–688, ISSN 1573-2894, URL <http://dx.doi.org/10.1007/s10589-014-9651-2>.
- Wright SJ (1997) *Primal-Dual Interior-Point Methods* (SIAM), URL <http://dx.doi.org/10.1137/1.9781611971453>.
- Zavala VM, Laird CD, Biegler LT (2008) Interior-point decomposition approaches for parallel solution of large-scale nonlinear parameter estimation problems. *Chem. Eng. Sci.* 63(19):4834–4845, ISSN 0009-2509, URL <http://dx.doi.org/10.1016/j.ces.2007.05.022>.
- Zhang H, Mazzi N, McKinnon K, Nava RG, Tomasgard A (2024) A stabilised Benders decomposition with adaptive oracles for large-scale stochastic programming with short-term and long-term uncertainty. *Comput. Oper. Res.* 167:106665, ISSN 0305-0548, URL <http://dx.doi.org/10.1016/j.cor.2024.106665>.