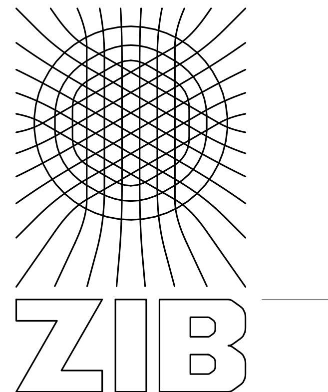


Zuse Institute Berlin

Takustr. 7
14195 Berlin
Germany

Ambros Gleixner, Nils-Christian Kempke, Thorsten Koch, Daniel Rehfeldt and Svenja Uslu

First Experiments with Structure-Aware Presolving for a Parallel Interior-Point Method

This report has been published in Operations Research Proceedings. Please cite as:

Gleixner et al., First Experiments with Structure-Aware Presolving for a Parallel Interior-Point Method. Operations Research Proceedings, 2019, DOI:10.1007/978-3-030-48439-2

ZIB Report 19-39 (July 2019)

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

First Experiments with Structure-Aware Presolving for a Parallel Interior-Point Method

Ambros Gleixner, Nils-Christian Kempke, Thorsten Koch, Daniel Rehfeldt and Svenja Uslu

Zuse Institute Berlin, Department of Mathematical Optimization,
{gleixner,kempke,koch,rehfeldt}@zib.de

July 26, 2019

Abstract

In linear optimization, matrix structure can often be exploited algorithmically. However, beneficial presolving reductions sometimes destroy the special structure of a given problem. In this article, we discuss structure-aware implementations of presolving as part of a parallel interior-point method to solve linear programs with block-diagonal structure, including both linking variables and linking constraints. While presolving reductions are often mathematically simple, their implementation in a high-performance computing environment is a complex endeavor. We report results on impact, performance, and scalability of the resulting presolving routines on real-world energy system models with up to 700 million nonzero entries in the constraint matrix.

1 Introduction

Linear programs (LPs) from energy system modeling and from other applications based on time-indexed decision variables often exhibit a distinct block-diagonal structure. Our extension [?] of the parallel interior-point solver PIPS-IPM [?] exploits this structure even when both linking variables and linking constraints are present simultaneously. It was designed to run on high-performance computing (HPC) platforms to make use of their massive parallel capabilities. In this article, we present a set of highly parallel presolving techniques that improve PIPS-IPM's performance while preserving the necessary structure of a given LP. We give insight into the implementation and the design of said routines and report results on their performance and scalability.

The mathematical structure of models handled by the current version of the solver are block-diagonal LPs as specified in Fig. 1. The $x_i \in \mathbb{R}^{n_i}$ are vectors of decision variables and $\ell_i, u_i \in (\mathbb{R} \cup \{\pm\infty\})^{n_i}$ are vectors of lower and upper bounds for $i = 0, 1, \dots, N$. The extended version of PIPS-IPM applies a parallel interior-point method to the problem exploiting the given structure for

$$
\begin{array}{rcccccl}
\min \quad \mathbf{c_0^T x_0} & + & c_1^T x_1 & + & \cdots & + & c_N^T x_N \\
\text{s.t.} \quad \mathbf{A_0 x_0} & & & & & & = \mathbf{b_0} \\
\mathbf{d_0} \le \mathbf{C_0 x_0} & & & & & & \le \mathbf{f_0} \\
A_1 \mathbf{x_0} & + & B_1 x_1 & & & & = b_1 \\
d_1 \le C_1 \mathbf{x_0} & + & D_1 x_1 & & & & \le f_1 \\
\vdots & & & & \ddots & & \vdots \\
A_N \mathbf{x_0} & + & & & & + & B_N x_N = b_N \\
d_N \le C_N \mathbf{x_0} & + & & & & + & D_N x_N \le f_N \\
\mathbf{F_0 x_0} & + & F_1 x_1 & + & \cdots & + & F_N x_N = \mathbf{b_{N+1}} \\
\mathbf{d_{N+1}} \le \mathbf{G_0 x_0} & + & G_1 x_1 & + & \cdots & + & G_N x_N \le \mathbf{f_{N+1}} \\
& & & & & & \ell_i \le x_i \le u_i \quad \forall i = 0, \ldots, N
\end{array}
$$

Figure 1: LP with block-diagonal structure linked by variables and constraints.

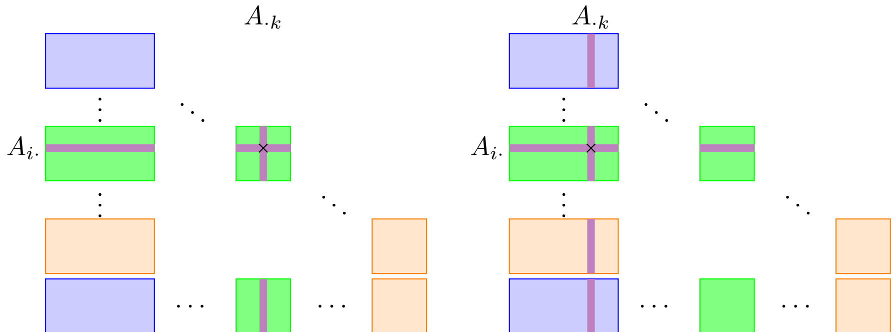
parallelizing expensive linear system solves. It distributes the problem among several different processes and establishes communication between them via the Message Passing Interface (MPI). Distributing the LP data among these MPI processes as evenly as possible is an elementary feature of the solver. Each process only knows part of the entire problem, making it possible to store and process huge LPs that would otherwise be too large to be stored in main memory on a single desktop machine.

The LP is distributed in the following way: For each index $i = 1, \ldots, N$ only one designated process stores the matrices A_i, B_i, C_i, D_i, F_i, G_i, the vectors c_i, b_i, d_i, f_i, and the variable bounds ℓ_i, u_i. We call such a unit of distribution a *block* of the problem. Furthermore, each process holds a copy of the block with $i = 0$, containing the matrices A_0, C_0, F_0, G_0 and the corresponding vectors for bounds. All in all, N MPI processes are used. Blocks may be grouped to reduce N. The presolving techniques presented in this paper are tailored to this special distribution and structure of the matrix.

2 Structure-Specific Parallel Presolving

Currently, we have extended PIPS-IPM by four different presolving methods. Each incorporates one or more of the techniques described in [**?**, **?**, **?**]: singleton row elimination, bound tightening, parallel and nearly parallel row detection, and a few methods summarized under the term model cleanup. The latter includes the detection of redundant rows as well as the elimination of negligibly small entries from the constraint matrix.

The presolving methods are executed consecutively in the order listed above. Model cleanup is additionally called at the beginning of the presolving. A presolving routine can apply certain reductions to the LP: deletion of a row or column, deletion of a system entry, modification of variable bounds and the left- and right-hand side, and modification of objective function coefficients. We distinguish between local and global reductions. While *local reductions* happen exclusively on the data of a single block, *global reductions* affect more



a) Singleton row leading to local changes. b) Singleton row leading to global changes.

Figure 2: LP data distributed on different processes and an entry a_{ik} with corresponding singleton row $A_{i\cdot}$ and column $A_{\cdot k}$.

than one block and involve communication between the processes. Since MPI communication can be expensive, we reduced the amount of data sent and the frequency of communication to a minimum and introduced local data structures to support the synchronization whenever possible.

In the following, singleton row elimination is used as an example to outline necessary data structures and methods. Although singleton row elimination is conceptually simple, its description still covers many difficulties arising during the implementation of preprocessing in an HPC environment. A singleton row refers to a row in the constraint matrix only containing one variable with nonzero coefficient. Both for a singleton equality and a singleton inequality row, the bounds of the respective variable can be tightened. This tightening makes the corresponding singleton row redundant and thus removable from the problem. In the case of an equality row, the corresponding variable is fixed and removed from the system.

Checking whether a non-linking row is singleton is straightforward since a single process holds all necessary information. The detection of singleton linking rows requires communication between the processes. Instead of asking all processes whether a given row is singleton, we introduced auxiliary data structures. Let $f = (f_0, f_1, \ldots, f_N)$ denote the coefficient vector of a linking row. Every process i knows the number of nonzeros in block i, i.e., $||f_i||_0$, and in block 0, i.e., $||f_0||_0$, at all times. At each synchronization point, every process also stores the current number of nonzeros overall blocks, $||f||_0$. Whenever local changes in the number nonzeros of a linking row occur, the corresponding process stores these changes in a buffer, instead of directly modifying $||f_i||_0$ and $||f||_0$. From that point on the global nonzero counters for all other processes are outdated and provide only an upper bound. Whenever a new presolving method that makes use of these counters is entered, the accumulated changes of all processes get broadcast. The local counters $||f_i||_0$ and $||f||_0$ are updated and stored changes are reset to zero.

After a singleton row is detected, there are two cases to consider, both visualized in Fig. 2. A process might want to delete a singleton row that has its singleton entry in a non-linking part of the LP (Fig. 2a). This can be done

Table 1: Runtimes and nonzero reductions for parallel presolving and sequential SoPlex presolving. The number of nonzeros in columns "nnzs" are given in thousands.

	input		PIPS-IPM			SoPlex	
instance	N	nnzs	t_1 [s]	t_N [s]	nnzs	t_S [s]	nnzs
oms_1	120	2891k	1.13	0.02	2362k	1.51	2391k
oms_2	120	11432k	5.10	0.19	9015k	11.09	9075k
oms_3	120	1696k	1.01	2.88	1639k	0.64	1654k
oms_4	120	131264k	57.25	3.45	126242k	206.31	127945k
oms_5	120	216478k	157.12	85.41	158630k	>24h	–
oms_6	120	277923k	187.73	88.39	231796k	>24h	–
elmod_1	438	272602k	125.62	0.48	208444k	>24h	–
elmod_2	876	716753k	365.47	1.05	553144k	>24h	–
yssp_1	250	27927k	13.01	0.44	22830k	92.63	23758k
yssp_2	250	68856k	33.80	7.28	55883k	1034.77	59334k
yssp_3	250	32185k	14.10	0.36	28874k	95.08	29802k
yssp_4	250	85255k	39.71	7.25	76504k	1930.16	80148k

immediately since none of the other processes is affected. By contrast, modifying the linking part of the problem is more difficult since all other processes have to be notified about the changes, e.g., when a process fixes a linking variable or when it wants to delete a singleton linking row (Fig. 2b). Again, communication is necessary and we implemented synchronization mechanisms for changes in variable bounds similar to the one implemented for changes in the nonzeros.

3 Computational Results

We conducted two types of experiments. First, we looked at the general performance and impact of our presolving routines compared with the ones offered by a different LP solver. For the second type of experiment, we investigated the scalability of our methods. The goal of the first experiment was to set the performance of our preprocessing into a more general context and show the efficiency of the structure-specific approach. To this end, we compared to the sequential, source-open solver SoPlex [?] and turned off all presolving routines that were not implemented in our preprocessing. With our scalability experiment, we wanted to further analyze the implementation and speed-up of our presolving. We thus ran several instances with different numbers of MPI processes.

The instances used for the computational results come from real-world energy system models found in the literature, see [2] (`elmod` instances) and [1] (`oms` and `yssp` instances). All tests with our parallel presolving were conducted on the JUWELS cluster at Jülich Supercomputing Centre (JSC). We used JUWELS' standard compute nodes running two Intel Xeon Skylake 8168 processors each with 24 cores 2.70 GHz and 96 GB memory. Since reading of the LP and presolving it with SoPlex was too time-consuming on JUWELS, we had to run the tests for SoPlex on a shared memory machine at Zuse institute Berlin with an Intel(R) Xeon(R) CPU E7-8880 v4, 2.2GHz, and 2 TB of RAM.

The results of the performance experiment are shown in Table 1. We compared the times spent in presolving by SoPlex t_S, our routines running with one MPI process t_1 and running with the maximal possible number of MPI processes t_N. The nnzs columns report the number of nonzeros when read in (input) and after preprocessing. The key observations are:

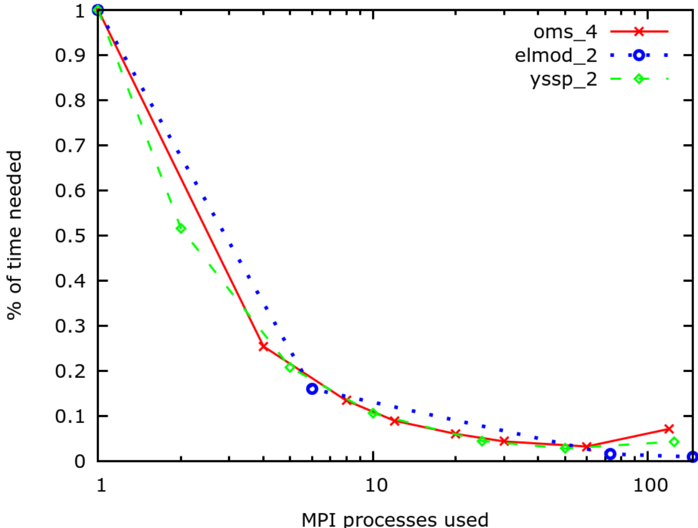


Figure 3: Total presolving time for three instances of each type, relative to time for sequential presolving with one MPI process.

- Except on the two smallest instances with less than 3 million nonzeros, already the sequential version of structure-specific presolving outperformed SoPlex significantly. The four largest instances with more than 200 million nonzeros could not be processed by SoPlex within 24 hours.
- The overall reduction performed by both was very similar with an average deviation of less than 2%. The nonzero reduction overall instances was about 16% on average.
- Parallelization reduced presolving times on all instances except the smallest instance `oms_3`. On `oms_2`, `elmod_{1,2}`, and `yssp_{2,4}` the speed-ups were of one order of magnitude or more. However, on instances `oms_{4,5,6}` and `yssp_{2,4}` the parallel speed-up was limited, a fact that is further analyzed in the second experiment.

The results of our second experiment can be seen in Figure 3. We plot times for parallel presolving, normalized by the time needed by one MPI process. Let $S_n = t_1/t_n$ denote the speed-up obtained with n MPI processes versus one MPI process. Whereas for `elmod_2` we observe an almost linear speed-up $S_{146} \approx 114$, on `yssp_2` and `oms_4` the best speed-ups $S_{50} \approx 36$ and $S_{60} \approx 31$, respectively, are sublinear. For larger numbers of MPI processes, runtimes even start increasing again.

The limited scalability on these instances is due to a comparatively large amount of linking constraints. As explained in Sec. 2, performing global reductions within linking parts of the problem increases the synchronization effort. As a result, this phenomenon usually leads to a "sweet spot" for the number of MPI processes used, after which performance starts to deteriorate again. This effect was also responsible for the low speed-up on `oms_{5,6}` in Table 1. A larger speed-up can be achieved when running with fewer processes.

To conclude, we implemented a set of highly parallel structure-preserving presolving methods that proved to be as effective as sequential variants found in an out-of-the-box LP solver and outperformed them in terms of speed on truly large-scale problems. Beyond the improvements of the presolving phase, we want to emphasize that the reductions helped to accelerate the subsequent interior-point code significantly. On the instance `elmod_1`, the interior-point time could be reduced by more than half, from about 780 to about 380 seconds.

Acknowledgements This work is funded by the Federal Ministry for Economic Affairs and Energy within the BEAM-ME project (ID: 03ET4023A-F) and by the Federal Ministry of Education and Research within the Research Campus MODAL (ID: 05M14ZAM). The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS Supercomputer JUWELS at Jülich Supercomputing Centre (JSC).

References

[1] K. Cao, J. Metzdorf, and S. Birbalta. Incorporating Power Transmission Bottlenecks into Aggregated Energy System Models. *Sustainability*, 10(6):1–32, 2018.

[2] F. Hinz. *Voltage Stability and Reactive Power Provision in a Decentralizing Energy System*. PhD thesis, TU Dresden, 2017.