A Distributed Interior-Point KKT Solver for Multistage Stochastic Optimization

Jens Hübner[1], Martin Schmidt[2], Marc C. Steinbach[1]

Abstract. Multistage stochastic optimization leads to NLPs over scenario trees that become extremely large when many time stages or fine discretizations of the probability space are required. Interior-point methods are well suited for these problems if the arising huge, structured KKT systems can be solved efficiently, for instance, with a large scenario tree but a moderate number of variables per node. For this setting we develop a distributed implementation based on data parallelism in a depth-first distribution of the scenario tree over the processes. Our theoretical analysis predicts very low memory and communication overheads. Detailed computational experiments confirm this prediction and demonstrate the overall performance of the algorithm. We solve multistage stochastic quadratic programs with up to 400×10^6 variables and 8.59×10^9 KKT matrix entries or 136×10^6 variables and 12.6×10^9 entries on a compute cluster with 384 GiB of RAM.

1. Introduction

The problems considered here arise from dynamic optimization models over a probability space where the objective takes the form of an expectation and the constraints have to be satisfied with probability one. In specific, the objective and constraints are generated by an exogenous random process that is assumed to be known. Discretization yields a separable nonlinear optimization problem (NLP) over a scenario tree whose nodes carry objective terms, constraints terms, and probabilities. The scenario tree thus models a discrete time filtration on a finite probability space. Its root represents time zero ("here-and-now") while the remaining nodes represent all possible future events. This way the current decision takes into account the possible future, and the NLP solution is a policy rather than a single optimal decision: it specifies precomputed optimal decisions for all future events.

Standard solution algorithms for multistage stochastic problems include various decomposition methods and interior-point methods; see [2] and [12] for an overview of algorithmic approaches and of stochastic optimization in general. Primal decomposition methods optimize individual nodes and treat the vertical coupling between stages iteratively, whereas dual decomposition methods optimize individual scenarios and iterate on the non-anticipativity condition representing the horizontal coupling. Interior-point methods, in contrast, iterate on the (nonlinear) KKT conditions of the entire problem wherein the global coupling across the tree is treated explicitly. All these algorithms offer a significant degree of inherent parallelism.

The focus of this paper is the efficient distributed implementation of tree-sparse linear algebra operations in an interior-point method, primarily the KKT solver and the matrix-vector product. The distributed implementation is based on data parallelism where nodes of the scenario tree with their associated NLP data are

Date: February 9, 2016.

2000 *Mathematics Subject Classification.* 65K10, 90-08, 90C06, 90C35, 90C90.

Key words and phrases. Multistage stochastic programming, Parallel computing, Distributed computing, KKT systems, Interior-point methods.

distributed in a depth-first fashion. This is a natural ordering that has first been proposed by Blomvall [3]; here we provide a theoretical analysis of the communication overhead and confirm the analysis by computational experiments.

For simplicity we restrict ourselves to convex NLPs and focus on issues of the distributed solution by interior-point methods. We also assume that the required first-order and second-order derivatives are readily available. Topics like structured convexification and regularization techniques or quasi-Newton updates for nonconvex multistage stochastic NLPs are treated in the PhD thesis [11].

Other structure-exploiting parallel interior-point approaches for stochastic programs include the primal solver for convex NLPs by Blomvall [3] and Lindberg and Blomvall [4] and the object-oriented solver OOPS for structured convex QPs by Gondzio and Grothey [6, 7, 8]. A dense solver for related saddle-point systems is presented by Lubin, Petra, and Anitescu [14]. All the structured approaches feature related KKT algorithms with linear complexity. However, there are significant modeling differences regarding the formulation of system dynamics and the level of detail of other constraints. See [10] for an overview of these approaches, and [20] for an earlier overview of structured (sequential) interior-point methods. Blomvall and Lindberg [4] parallelize the KKT algorithm in the same static fashion as we do while Gondzio and Grothey [6] apply a dynamic programming model based on matrix subblock prototypes. While Blomvall and Lindberg consider only convex NLPs, Gondzio and Grothey also solve nonconvex multistage stochastic problems using OOPS within a textbook SQP framework [9]. Although we restrict ourselves to the convex case in this paper, our primal-dual method handles nonconvex multistage stochastic NLPs directly [11, 18].

The remainder of the paper is organized as follows. To keep the exposition self-contained, the mathematical background of our interior-point code is briefly sketched in Sect. 2, and the sequential tree-sparse KKT algorithm with an application example is presented in Sect. 3. Our general framework for distributed tree algorithms is developed and thoroughly analyzed in Sect. 4 and then specialized to the tree-sparse problems under consideration in Sect. 5. Detailed computational experiments in Sect. 6 demonstrate the performance of our algorithms and confirm the theoretical results. We conclude the paper with a summary and discussion in Sect. 7.

2. Interior-Point Methods for Convex NLPs

We consider standard interior-point methods for convex NLPs of the general form

$$\min_{y\in\mathbb{R}^n} f(y) \quad \text{s.t.} \quad c_i(y) = 0,\ i \in \mathcal{E}, \quad c_i(y) \geq 0,\ i \in \mathcal{I}, \tag{1}$$

where the objective f and the constraints c_i are assumed to be $\mathcal{C}^2$ functions with f convex, c_i affine for $i \in \mathcal{E}$, and c_i concave for $i \in \mathcal{I}$. For more details of the interior-point approach sketched in the following see, e.g., [16, 23, 24].

To apply an interior-point method to (1), we introduce slack variables $s_i > 0$ for the inequality constraints and solve a series of barrier problems

$$\min_{y,s} f(y) - \beta \sum_{i\in\mathcal{I}} \ln s_i \quad \text{s.t.} \quad c_{\mathcal{E}}(y) = 0, \quad c_{\mathcal{I}}(y) - s = 0,$$

where $c_{\mathcal{E}} := (c_i)_{i\in\mathcal{E}}$, $c_{\mathcal{I}} := (c_i)_{i\in\mathcal{I}}$, and $\beta > 0$ is the barrier parameter that has to be driven to zero in the limit. As it is well known, this approach is equivalent to applying a homotopy method to the perturbed primal-dual KKT system of (1),

$$\begin{aligned} \nabla f(y) - C_{\mathcal{E}}(y)^T z - C_{\mathcal{I}}(y)^T v = 0, \quad SVe - \beta e = 0, \\ c_{\mathcal{E}}(y) = 0, \quad c_{\mathcal{I}}(y) - s = 0. \end{aligned} \tag{2}$$

Here $C_{\mathcal{E}}(y) \in \mathbb{R}^{|\mathcal{E}|\times n}$ and $C_{\mathcal{I}}(y) \in \mathbb{R}^{|\mathcal{I}|\times n}$ are the respective Jacobians of the equality and inequality constraints, S and V denote diagonal matrices made up of the entries of s and v, and e is the vector of all ones with appropriate dimension. Note that the KKT conditions of the original problem (1) are given by (2) with $\beta = 0$ and $s, v \geq 0$. In the context of barrier methods, the latter conditions are strengthened to $s, v > 0$ to ensure that the barrier problem is well-defined.

Now, primal-dual interior-point methods proceed as follows. Given a primal-dual iterate (y^k, s^k, z^k, v^k) with $s^k, v^k > 0$, the KKT system

$$\begin{bmatrix} H^k & 0 & -(C_{\mathcal{E}}^k)^T & -(C_{\mathcal{I}}^k)^T \\ 0 & V^k & 0 & S^k \\ C_{\mathcal{E}}^k & 0 & 0 & 0 \\ C_{\mathcal{I}}^k & -I & 0 & 0 \end{bmatrix} \begin{pmatrix} \Delta y^k \\ \Delta s^k \\ \Delta z^k \\ \Delta v^k \end{pmatrix} = - \begin{pmatrix} \nabla_y \mathcal{L}^k \\ S^k V^k e - \beta^k e \\ c_{\mathcal{E}}^k \\ c_{\mathcal{I}}^k - s^k \end{pmatrix}$$

determines a primal-dual search direction $(\Delta y^k, \Delta s^k, \Delta z^k, \Delta v^k)$. Here $C_{\mathcal{E}}^k = C_{\mathcal{E}}(y^k)$, $C_{\mathcal{I}}^k = C_{\mathcal{I}}(y^k)$, $\nabla_y \mathcal{L}^k$ is the gradient of the Lagrangian of (1) in iteration k,

$$\nabla_y \mathcal{L}^k = \nabla f(y^k) - (C_{\mathcal{E}}^k)^T z^k - (C_{\mathcal{I}}^k)^T v^k,$$

and H^k denotes the Hessian of the Lagrangian. The slack variables Δs^k are easily eliminated using the second block row of this system, which yields the reduced and symmetrized 3×3 KKT system

$$\begin{bmatrix} H^k & (C_{\mathcal{E}}^k)^T & (C_{\mathcal{I}}^k)^T \\ C_{\mathcal{E}}^k & 0 & 0 \\ C_{\mathcal{I}}^k & 0 & -(V^k)^{-1} S^k \end{bmatrix} \begin{pmatrix} \Delta y^k \\ -\Delta z^k \\ -\Delta v^k \end{pmatrix} = - \begin{pmatrix} \nabla_y \mathcal{L}^k \\ c_{\mathcal{E}}^k \\ c_{\mathcal{I}}^k - \beta^k (V^k)^{-1} e \end{pmatrix}. \tag{3}$$

To preserve positivity of the iterates s^k and v^k, upper limits on the primal and dual step lengths are imposed using the fraction-to-the-boundary rule,

$$\bar{\alpha}_p^k := \max_{\alpha \in (0,1]} \{\alpha \Delta s^k \geq -\tau s^k\}, \quad \bar{\alpha}_d^k := \max_{\alpha \in (0,1]} \{\alpha \Delta v^k \geq -\tau v^k\}, \tag{4}$$

where $\tau \in (0, 1)$ is the fraction-to-the-boundary parameter. In the convex case considered here, these values are actually used as step lengths α_p^k and α_d^k, whereas additional line-search or trust region techniques are required to determine α_p^k, α_d^k in the nonconvex case. The new iterate is then given by

$$(y^{k+1}, s^{k+1}) = (y^k, s^k) + \alpha_p^k (\Delta y^k, \Delta s^k), \tag{5a}$$

$$(z^{k+1}, v^{k+1}) = (z^k, v^k) + \alpha_d^k (\Delta z^k, \Delta v^k). \tag{5b}$$

The barrier parameter β^k is either updated or kept for the next iteration. Here the overall update mechanism has to ensure that the sequence (β^k) converges to zero.

The interior-point method terminates when a KKT point of (1) is approximately reached, i.e., when an iterate satisfies

$$e^k := \max\{\|\nabla_y \mathcal{L}^k\|, \|c_{\mathcal{E}}^k\|, \|c_{\mathcal{I}}^k - s^k\|, \|S^k V^k e\|\} \leq \varepsilon \tag{6}$$

with a user-specified tolerance $\varepsilon > 0$ and a suitable norm.

3. Tree-Sparse NLPs and Algorithms

One advantage of interior-point methods for multistage stochastic problems over primal or dual decomposition approaches is that they are particularly well suited for nonlinear and nonconvex problems. Moreover, they are applicable to nonstandard problems with global constraints whose recourse structure is implicit. These advantages come at the price of a huge KKT system (3) in every iteration of the interior-point method, hence efficient and robust KKT solvers are essential. To this end, the third author has introduced an integrated modeling and solution framework

that we refer to as "tree-sparse" [20]. It consists of natural model formulations that have favorable regularity properties and block-level sparsity admitting $O(|V|)$ KKT solution algorithms where $|V|$ is the number of tree nodes. There are three standard forms that cover, respectively, the general case with no further structure and two possible control formulations with different stochastic interpretations.

Here we present one of the control forms for convex NLPs and we fix the notation for the QP case whose matrix blocks and vectors appear in the KKT system. The associated KKT solution algorithm is then presented in tabular form. Local equality constraints are omitted to avoid unnecessary technical complexity. Full technical details for these constraints can be found in [20] and the references therein. The formulation considered here is required for the application example in Sect. 3.3 and is used in the computational experiments.

For the following, let $V = \{0, \dots, n\}$ denote the node set of a tree $T = (V, E)$ rooted in 0. Given a node $j \in V$, we denote the set of successors by $S(j)$, the predecessor by $i = \pi(j)$ (if $j \neq 0$), and the path to the root by $\Pi(j) = (j, \pi(j), \dots, 0)$. The level of j is the path length, $t(j) = |\Pi(j)| - 1$. Finally L_t stands for the set of level t nodes and L for the set of leaves.

3.1. **Tree-Sparse NLP.** In this paper, we only need the *incoming control from*,

$$\min_{u,x} \quad \sum_{j \in V} (\phi_{ij}(x_i, u_j) + \phi_j(x_j)) \tag{7a}$$

$$\text{s.t.} \quad x_j = G_j x_i + E_j u_j + h_j, \qquad j \in V, \tag{7b}$$

$$r_{ij}(x_i, u_j) \geq 0,\ r_j(x_j) \geq 0, \qquad j \in V, \tag{7c}$$

$$u_j \in [u_j^-, u_j^+],\ x_j \in [x_j^-, x_j^+], \quad j \in V, \tag{7d}$$

$$\sum_{j \in V} (F_j x_j + D_j u_j) + e_V = 0. \tag{7e}$$

Here x_i is empty for $j = 0$ ($x_i \in \mathbb{R}^0$ formally), and the bounds on u_j, x_j are optional. The local objectives ϕ_{ij}, ϕ_j are convex and the local range constraints r_{ij}, r_j are concave. In the QP case, these functions take the respective forms

$$\phi_{ij}(x_i, u_j) + \phi_j(x_j) = x_i^T J_j u_j + \frac{1}{2} u_j^T K_j u_j + \frac{1}{2} x_j^T H_j x_j + d_j^T u_j + f_j^T x_j, \tag{8a}$$

$$F_{ij}^r x_i + D_j^r u_j \in [r_{ij}^-, r_{ij}^+], \qquad F_j^r x_j \in [r_j^-, r_j^+]. \tag{8b}$$

The nonconvex NLP case as well as the outgoing control form are discussed in [11].

3.2. **Tree-Sparse KKT Algorithm.** In node-wise representation, the tree-sparse KKT system (3) corresponding to the incoming control problem (7) reads

$$\begin{aligned}
J_j x_i + (K_j + \Phi_j^u) u_j - E_j^T \lambda_j - (D_j^r)^T v_j^u - D_j^T \mu + d_j &= 0, \quad j \in V, \\
(H_j + \Phi_j^x) x_j + \sum_{k \in S(j)} J_j^T u_k + \lambda_j - \sum_{k \in S(j)} G_j^T \lambda_k & \\
- \sum_{k \in S(j)} (F_{ij}^r)^T v_k^u - (F_j^r)^T v_j^x - F_j^T \mu + f_j &= 0, \quad j \in V, \\
G_j x_i + E_j u_j - x_j + h_j &= 0, \quad j \in V, \\
F_{ij}^r x_i + D_j^r u_j + (\Psi_{ij})^{-1} v_j^u + r_{ij} &= 0, \quad j \in V, \\
F_j^r x_j + (\Psi_j)^{-1} v_j^x + r_j &= 0, \quad j \in V, \\
\sum_{j \in V} D_j u_j + \sum_{j \in V} F_j x_j + e_V &= 0.
\end{aligned}$$

TABLE 1. KKT solution for the incoming control case

	Factorization ↓	Inward subst. ↓	Outward subst. ↑
1:	$K_j \mathrel{+}= \Phi_j^u$		$-v_{ij} \leftarrow \Psi_{ij}(-v_{ij})$
2:	$H_j \mathrel{+}= \Phi_j^x$		$-v_j \leftarrow \Psi_j(-v_j)$
3:	$K_j \mathrel{+}= D_j^{r\,T}\Psi_{ij}D_j^r$	$d_j \mathrel{+}= D_j^{r\,T}\Psi_{ij}r_{ij}$	$-v_{ij} \mathrel{+}= D_j^r u_j$
4:	$J_j \mathrel{+}= D_j^{r\,T}\Psi_{ij}F_{ij}^r$		
5:	$H_i \mathrel{+}= F_{ij}^{r\,T}\Psi_{ij}F_{ij}^r$	$f_i \mathrel{+}= F_{ij}^{r\,T}\Psi_{ij}r_{ij}$	$-v_{ij} \mathrel{+}= F_{ij}^r x_i$
6:	$H_j \mathrel{+}= F_j^{r\,T}\Psi_j F_j^r$	$f_j \mathrel{+}= F_j^{r\,T}\Psi_j r_j$	$-v_j \mathrel{+}= F_j^r x_j$
7:	$K_j \mathrel{+}= E_j^T H_j E_j$	$d_j \mathrel{+}= E_j^T(H_j h_j + f_j)$	$-\lambda_j \mathrel{+}= H_j x_j$
8:	$K_j \leftarrow L_j L_j^T$		
9:	$J_j \mathrel{+}= E_j^T H_j G_j$		$x_j \mathrel{+}= E_j u_j$
10:	$H_i \mathrel{+}= G_j^T H_j G_j$	$f_i \mathrel{+}= G_j^T(H_j h_j + f_j)$	$x_j \mathrel{+}= G_j x_i$
11:	$F_i \mathrel{+}= F_j G_j$	$e_V \mathrel{+}= F_j h_j$	$-\lambda_j \mathrel{+}= F_j^T(-\mu)$
12:	$D_j \mathrel{+}= F_j E_j$		
13:	$D_j \leftarrow D_j L_j^{-T}$	$h_j \leftrightarrow f_j$	
14:	$J_j \leftarrow L_j^{-1} J_j$	$d_j \leftarrow L_j^{-1} d_j$	$u_j \leftarrow -L_j^{-T} u_j$
15:	$H_i \mathrel{-}= J_j^T J_j$	$f_i \mathrel{-}= J_j^T d_j$	$u_j \mathrel{+}= J_j x_i$
16:	$F_i \mathrel{-}= D_j J_j$		
17:	$X_V \mathrel{+}= D_j D_j^T$	$e_V \mathrel{-}= D_j d_j$	$u_j \mathrel{+}= D_j^T(-\mu)$
18:	$X_V \leftarrow LL^T$	$e_V \leftarrow L^{-1} e_V$	$-\mu \leftarrow L^{-T}(-\mu)$

Here $\Psi_{ij}, \Psi_j > 0$ and $\Phi_j^u, \Phi_j^x \geq 0$ denote the respective diagonal barrier matrices corresponding to the inequalities (7c) and the bounds (7d). The Lagrange multipliers λ_j, v_{ij}, v_j, and μ correspond to the constraints (7b), (7c), and (7e), respectively.

Block-level node operations of the solution algorithm are listed in Table 1. The fill-in block of the global equalities (7e) is initialized to $X_V := 0$. Operations of the factorization and inward substitution phases are executed in the order 1 to 17 (↓) and node $j \in V$ is processed after completing all its successors. The outward substitution phase is executed in reverse order, from 17 to 1 (↑), and node j is processed after its predecessor i. The execution order of sibling nodes $j \in S(i)$ is arbitrary. Operation 18 is only applied at the root node.

The distributed implementation of the algorithm works exactly the same except that we need to transmit certain data during the process (see Sect. 5.1). A more detailed discussion of the tree-sparse KKT algorithm is presented in [11] and the references therein.

3.3. **Application Example.** As a real-life example we consider a multi-period portfolio management problem; cf. [5]. Although the objective has Markowitz type, it can be shown that for moderate target returns this model approximately minimizes the lower semi-variance [19].

For $t = 0, \ldots, T$, denote by $x_t = (z_t, z_t^c) \in \mathbb{R}^{n+1}$ the values of investments in n risky assets and in cash with random returns $r_{t+1} \in \mathbb{R}^n$ and interest rates r_{t+1}^c, by $v_t^{\pm} \in \mathbb{R}^n_{\geq 0}$ respective amounts purchased and sold at time t with transaction cost coefficients $c_t^{\pm} \in \mathbb{R}^n_{\geq 0}$, and by ϕ_t an external cash flow. With $e := (1, \ldots, 1)^T \in \mathbb{R}^n$,

we then have

$$z_t = \mathrm{Diag}[r_t] z_{t-1} + v_t^+ - v_t^-,$$
$$z_t^c = r_t^c z_{t-1}^c - (e + c_t^+)^T v_t^+ + (e - c_t^-)^T v_t^- + \phi_t.$$

The investor prescribes a desired expected return ρ at time $T+1$ and minimizes the associated variance, the "risk" R:

$$\rho = \mathbf{E}[r_{t+1}^T z_t + r_{t+1}^c z_t^c], \quad R = \mathbf{E}[(r_{t+1}^T z_t + r_{t+1}^c z_t^c - \rho)^2].$$

In addition we consider application-specific inequality constraints, $B_t x_t \in [b_t^-, b_t^+]$. Given a scenario tree over $\{0, \dots, T\}$ with node probabilities p_j, we denote realizations of state and control variables at $j \in L_t$ by $x_j = (z_j, z_j^c)$ and $u_j = (v_j^+, v_j^-)$ and let

$$G_j = \mathrm{Diag}[r_j, r_t^c], \quad E_t = \begin{bmatrix} I & -I \\ (e + c_t^+)^T & (e - c_t^-)^T \end{bmatrix}, \quad h_t = \begin{pmatrix} 0 \\ \phi_t \end{pmatrix}.$$

With suitably defined conditional expectations $\bar{\Sigma}_j, \bar{r}_j$ for $j \in L$ [19], the resulting multistage mean-variance optimization problem then has incoming control form:

$$\min_{u \geq 0, x} \sum_{j \in L} p_j x_j^T \bar{\Sigma}_j x_j \quad \text{s.t.} \quad \sum_{j \in L} p_j \bar{r}_j^T x_j = \rho,$$
$$G_j x_i + E_t u_j + h_t = x_j, \quad B_t x_j \in [b_t^-, b_t^+], \quad j \in V.$$

A subindex t designates deterministic quantities here: all "realizations" for $j \in L_t$ are identical. In fact, $E_t, B_t, b_t^\pm$ are usually not even time-dependent.

4. Distributed DFS-Based Tree Algorithms

The distributed tree algorithms considered in the following are based on a static partitioning of the underlying tree: the data of the tree-sparse NLP (7) and the operations of the tree-sparse KKT solution algorithm are associated with the nodes of the tree, which we distribute among the participating processes. This way, the problem data and the workload are distributed as well. We assign the nodes in depth-first order to the processes (like Blomvall [3]) and present suitable DFS-based traversals of nodes for the resulting distributed tree parts. These node traversals avoid unnecessary idle times and allow a systematic reduction of the required communication in the distributed execution of the tree-sparse KKT solver.

In what follows, we view the factorization and the substitutions of the tree-sparse KKT algorithm as special cases of *DFS-based tree algorithms* and present general results for the abstract algorithms. In Sect. 5, these general results are applied to the specific algorithms of the tree-sparse KKT solution.

The section is organized as follows. Subsection 4.1 provides basic graph terminology and graph notation. The abstract DFS-based tree algorithms are introduced in Sect. 4.2, and Sect. 4.3 describes the node distribution in depth-first order. In Sect. 4.4, we introduce terminology and notation for distributed trees and establish the concept of a *depth-first distributed tree*. The theoretical results for depth-first distributed trees, which we present in Sect. 4.5, form the basis of our approach for *distributed DFS-based tree algorithms* in Sect. 4.6.

4.1. Graph and Tree Terminology.

Let $G = (V, E)$ be a graph with node set $V = \{0, \dots, n\}$ and edge set E. In a directed graph, the edge $(i, j) \in E$ is an *outedge* of i and an *inedge* of j. An acyclic undirected graph is a *forest*, and a connected forest is a *tree*. All trees considered in this paper are rooted out-trees with root $0 \in V$, and thus have an induced edge direction that points away from the root. Hence, the edge set E contains the inedge (i, j) of $j \in V$ incident on its predecessor i, and the outedges of j incident on the successors $S(j)$.

A *node iteration* over a tree T defines a finite sequence $\{v_l\}_{l=1}^K$ of nodes within which the nodes are *visited*. We call a node iteration *complete* if all nodes of T are visited at least once. *Traversals* are methods for traversing the nodes and edges of a tree. They induce complete node iterations for which any two consecutive nodes are adjacent to each other. By convention traversals of rooted out-trees always start at the root node. During such a tree traversal a node is said to be *discovered* when it is visited through its inedge. The node is *finished* when all successors have been visited and the node has been left through the inedge (except for the root).

4.2. **DFS-Based Tree Algorithms.** The tree-sparse KKT algorithm in Table 1 consists of the factorization phase and two substitution phases. Each phase is an algorithm described by a set of node operations $\mathrm{OP}(j)$ with certain restrictions on the computation order. The operations $\mathrm{OP}(j)$ always involve steps 1 to 17 while step 18 belongs only to $\mathrm{OP}(0)$. The factorization and the inward substitution require node j to be processed after its successors, implying that the operations $\mathrm{OP}(j)$ are performed after completion of all $\mathrm{OP}(k)$ for $k \in S(j)$. This is achieved by using a depth-first traversal over the tree and applying the operations $\mathrm{OP}(j)$ upon finishing node j. During the outward substitution the predecessor's operations $\mathrm{OP}(i)$ need to be completed before executing $\mathrm{OP}(j)$. Using again a depth-first traversal, this is achieved by applying the operations $\mathrm{OP}(j)$ upon discovery of node j.

We classify the operations of $\mathrm{OP}(j)$ into *local* and *nonlocal* ones depending on the data that are involved. Data items (such as vectors and matrices) with index j or ij are associated with node j, and operations that involve only those *local data* are also called *local*. This applies to steps 1–4, 6–9, and 12–14 in Table 1. Data items without index or with index V are called *global*. They belong to the entire tree and are formally associated with the root. Operations of $\mathrm{OP}(j)$ that involve global data or data of the predecessor i are called *nonlocal*. This applies to steps 5, 10, 11, and 15–17 in Table 1. Those nonlocal operations that need read access to nonlocal data but only modify local data will be referred to as *nonlocal read* operations. On the other hand, *nonlocal write* operations use local data to modify nonlocal data. All nonlocal operations of the factorization and the inward substitution are nonlocal write operations, whereas the outward substitution has only nonlocal read operations.

In summary, a DFS-based algorithm consists of a set $\mathrm{OP}(j)$ of local and nonlocal operations for each node that are executed during a depth-first traversal over the tree. In an *outward algorithm* the operations $\mathrm{OP}(j)$ are applied upon discovery of node j, i.e., in the direction of the tree edges. In an *inward algorithm* the operations $\mathrm{OP}(j)$ are applied upon finishing the node, i.e., in the reverse direction.

4.3. **Node-Based Distribution.** Our distributed approach for tree-sparse problems is based on a partitioning of the tree. The nodes of the tree are distributed statically among the participating processes, i.e., each node j is uniquely assigned to one process during the entire algorithm. Each process holds the data associated with its assigned nodes and is responsible for its node operations $\mathrm{OP}(j)$. This way, the node distribution induces a distribution of the entire problem data as well as a distribution of the workload.

After partitioning the tree, the processes work in parallel on their assigned nodes. For nonlocal operations that involve two nodes i, j on different processes, interprocess communication is required to access the nonlocal data. The communication can cause significant overhead, and idle times occur when a process has to wait for another process to complete an adjacent node. The tree partitioning should therefore be chosen in a way that yields low communication overhead and idle times.

Blomvall proposes to distribute the nodes in a natural order by applying a depth-first strategy [3]. In combination with an elaborate computation order this can achieve a good compromise between both goals: reducing the idle times while keeping the communication low. In the following sections, we follow Blomvall's idea and formalize the concept of a depth-first distributed tree.

4.4. **Distributed Trees.** To split a tree $T = (V, E)$ and distribute it among q processes, let the node set be partitioned into q parts, $V = \bigcup_{p=1}^{q} V_p$. The subgraph induced by the node subset V_p is a forest that we denote by $\mathcal{F}_p = (V_p, E_p)$. In what follows, we use $\mathcal{T}$ to denote trees within such a forest in order to distinguish them from the original tree T. For a node $v \in V_p$, the path $\Pi(v)$ remains the path to the root of T and $t(v)$ still refers to the level of v in T.

Every subtree $\mathcal{T}_{p,r}$ in $\mathcal{F}_p$ is rooted in the unique node r with lowest level in $\mathcal{T}_{p,r}$. The *set of roots* in the forest $\mathcal{F}_p$ is denoted by $\mathcal{R}_p$, and the *set of all roots* is the union $\mathcal{R} = \bigcup_{p=1}^{q} \mathcal{R}_p$. For $q > 1$, the union of all forest edges is a strict subset of E. *Loose edges* are those edges in E that do not belong to any set E_p. They form the set $\mathcal{E} := E \setminus \bigcup_{p=1}^{q} E_p$. For a forest $\mathcal{F}_p$, the subset $\mathcal{E}_p \subseteq \mathcal{E}$ consists of those loose edges that are incident on a node in V_p. Every loose edge $(s, r) \in \mathcal{E}$ is an inedge of a root $r \in \mathcal{R} \setminus \{0\}$. A node s is called a *sender* if at least one of its outedges $(s, \cdot)$ is incident on a root $r \in \mathcal{R}$, i.e., $(s, r) \in \mathcal{E}$. Note that a sender can also be a root. The set $\mathcal{S}_p$ is the *set of senders* in the forest $\mathcal{F}_p$. The *set of all senders* is the union $\mathcal{S} = \bigcup_{p=1}^{q} \mathcal{S}_p$. Now, a *distributed tree* is the collection $\mathcal{D} = (T, \mathcal{P}_1, \ldots, \mathcal{P}_q, \mathcal{R}, \mathcal{S})$ that results from the node partitioning $V = \bigcup_{p=1}^{q} V_p$. Each *part* $\mathcal{P}_p = (\mathcal{F}_p, \mathcal{E}_p, \mathcal{R}_p, \mathcal{S}_p)$ thus consists of a forest and the associated loose edges, roots, and senders.

A tree T is *(DFS-)ordered* if its node set V is numbered based on a depth-first traversal. Denote by $T(v) = (V(v), E(v))$ the subtree of T rooted in v. Then T is *post-ordered* if $V(v) = \{v, v+1, \ldots, v+n_v-1\}$ for all $v \in V$, where $n_v = |V(v)|$. Post-orders are obtained by assigning consecutive numbers to each node upon discovery during the depth-first traversal. An *ascending node partitioning* of V with respect to its node numbering has the form

$$V = \bigcup_{p=1}^{q} V_p \quad \text{with} \quad V_p = \left\{ \sum_{k=1}^{p-1} |V_k|, \ldots, \sum_{k=1}^{p} |V_k| - 1 \right\}.$$

A *depth-first distributed tree* is a distributed tree that results from an ascending node partitioning of a post-ordered tree T. Figure 1 shows a depth-first distributed tree that consists of four equally sized parts.

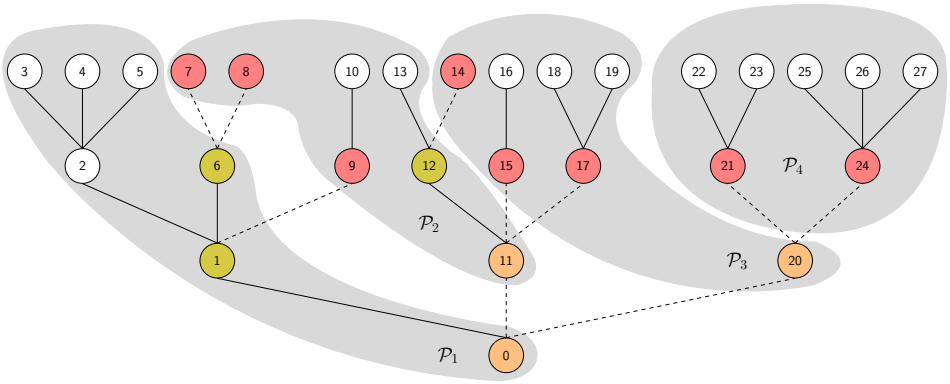
4.5. **Properties of Depth-First Distributed Trees.** For later reference, we first state an immediate consequence of our notation that holds for any tree.

Proposition 1. *For any two nodes $v_1 \neq v_2$ in a tree T it holds:*
(a) $v_1 \in \Pi(v_2)$ if and only if $v_2 \in V(v_1)$.
(b) $t(v_1) = t(v_2)$ implies $v_1 \notin V(v_2)$ and $v_1 \notin \Pi(v_2)$.

In the following we present some general properties of post-ordered trees and of depth-first distributed trees. The next proposition is an immediate consequence of the definition of post-ordered trees.

Proposition 2. *For each node v in a post-ordered tree T it holds:*
(a) $v < j$ for all nodes j in the node set $V(v) \setminus \{v\}$.
(b) $j < v$ for all nodes j on the path $\Pi(v) \setminus \{v\}$.

Proof. The first claim follows from $j \in V(v) = \{v, v+1, \ldots, v+n_v-1\}$ and the definition of the post-order. The second claim follows from (a) by applying Prop. 1(a), i.e., $v \in V(j)$. □



p	$\mathcal{R}_p$	$\mathcal{S}_p$	$\mathcal{E}_p$
1	0	0,1,6	$(0,11),(0,20),(1,9),(6,7),(6,8)$
2	7,8,9,11	11,12	$(6,7),(6,8),(1,9),(0,11),(11,15),(11,17),(12,14)$
3	14,15,17,20	20	$(0,20),(11,15),(11,17),(12,14),(20,21),(20,24)$
4	21,24	—	$(20,21),(20,24)$

FIGURE 1. A depth-first distributed tree with its structural data

Proposition 3. *Let $v_1 < v_2$ be two nodes in a post-ordered tree T with $v_2 \notin V(v_1)$. Then $j < v_2$ holds for all nodes $j \in V(v_1)$.*

Proof. Since T is post-ordered, we have $V(v_1) = \{v_1, \dots, v_1 + n_{v_1} - 1\}$. With $v_2 > v_1$ and $v_2 \notin V(v_1)$ it follows that $v_2 \geq v_1 + n_{v_1} > j$ for all $j \in V(v_1)$. □

Proposition 4. *Let $v_1 < v_2$ be two nodes in a post-ordered tree T with predecessors $\pi_1 := \pi(v_1)$, $\pi_2 := \pi(v_2)$. If $\pi_1 \neq \pi_2$ holds, then exactly one of the following statements is true:*

(a) $\pi_2 \in \Pi(\pi_1)$ or
(b) the predecessors have the order $\pi_1 < \pi_2$.

Proof. Suppose that (a) holds. From Prop. 2(b) we have $\pi_2 < \pi_1$, hence (b) cannot be true. Thus it suffices to verify that $\pi_2 \notin \Pi(\pi_1)$ implies (b). This can be shown by contradiction. Assume that $\pi_2 \notin \Pi(\pi_1)$ and $\pi_2 < \pi_1$. From Prop. 1(a) we have $\pi_1 \notin V(\pi_2)$ and from Prop. 3 we conclude that $j < \pi_1 < v_1$ for all nodes $j \in V(\pi_2)$. With $v_2 \in V(\pi_2)$ this implies $v_2 < v_1$, which contradicts the assumption $v_1 < v_2$. Hence, $\pi_2 \notin \Pi(\pi_1)$ implies $\pi_1 < \pi_2$. □

We now turn to the properties of depth-first distributed trees. In what follows, we use the function $P(v) = p$ to map the node $v \in V$ on its distributed tree part $\mathcal{P}_p$. The first proposition is an immediate consequence of the definitions of roots and senders.

Proposition 5. *For a depth-first distributed tree $\mathcal{D}$ it holds:*

(a) $P(v_1) \leq P(v_2)$ for any two nodes $v_1 < v_2$ in V.
(b) $P(\pi(r)) < P(r)$ for any root $r \in \mathcal{R} \setminus \{0\}$.
(c) There is a node $j \in S(s)$ with $P(s) < P(j)$ for any sender $s \in \mathcal{S}$.

Proof. Claim (a) is true by definition. For a root $r \in \mathcal{R} \setminus \{0\}$ we have $\pi(r) < r$ due to the post-order, hence $P(\pi(r)) \leq P(r)$ follows from (a). Second, it is $P(\pi(r)) \neq P(r)$ since otherwise r would be a non-root node in $\mathcal{T}_{P(r),\pi(r)}$. Hence (b) is true. For any sender $s \in \mathcal{S}$ there is by definition at least one root $r \in \mathcal{R} \setminus \{0\}$ such that $\pi(r) = s$, thus $r \in S(s)$. Therefore, (c) follows from (b). □

Proposition 6. *Let $\mathcal{D}$ be a depth-first distributed tree and $s \in \mathcal{S}_p$. For any node $v > s$ with $v \notin V(s)$ it holds $P(v) > P(s)$.*

Proof. According to Prop. 5(c) there is a node $j_1 \in S(s) \subseteq V(s)$ with $P(j_1) > P(s)$. With $v \notin V(s)$, Prop. 3 provides $v > j$ for all $j \in V(s)$. By Prop. 5(a) we have $P(v) \geq P(j_1) > P(s)$. □

Proposition 7. *Let $\mathcal{D}$ be a depth-first distributed tree. Let the node v be in part $\mathcal{P}_p$ and $s \in \mathcal{S}_p$ with $s \neq v$ and $t(v) = t(s)$. Then we have $v < s$ and $v \notin \mathcal{S}_p$.*

Proof. The level assumption $t(v) = t(s)$ yields $v \notin V(s)$ by Prop. 1(b). By Prop. 6, the order $v > s$ would imply $P(v) > P(s)$, which contradicts the assumption $P(v) = P(s)$. Hence, $v < s$ must hold. The same argument leads to the contradiction $P(v) < P(s)$ if $v \in \mathcal{S}_p$. □

The first theorem now states that all senders of the same part of a distributed tree lie on the same path to the root of the tree.

Theorem 1 (Sender-path)**.** *Let $\mathcal{D}$ be a depth-first distributed tree and $s_1, s_2 \in \mathcal{S}_p$ with $s_1 < s_2$. Then $s_1 \in \Pi(s_2)$, i.e., s_1 lies on the path from s_2 to the root.*

Proof. Let $s_1, s_2 \in \mathcal{S}_p$ with $s_1 < s_2$. Assume that $s_1 \notin \Pi(s_2)$, which is equivalent to $s_2 \notin V(s_1)$ by Prop. 1(a). Proposition 6 states $P(s_2) > P(s_1)$, contradicting $P(s_2) = P(s_1)$. Hence, $s_1 \in \Pi(s_2)$ holds. □

This theorem leads us to the following definition. For a depth-first distributed tree $\mathcal{D}$ the *sender-path* of part $\mathcal{P}_p$ is given by

$$\Pi_p^{\mathcal{S}} := \{v \in V_p \colon \text{it exists } s \in \mathcal{S}_p \text{ with } v \in \Pi(s)\}\,.$$

The root of the sender-path will be called *sender-root* $r_p^{\mathcal{S}}$ in the following. It satisfies

$$t(r_p^{\mathcal{S}}) = \min\left\{t(v) \colon v \in \Pi_p^{\mathcal{S}}\right\}.$$

Obviously, $r_p^{\mathcal{S}} \in \mathcal{R}$ since otherwise the predecessor $\pi(r_p^{\mathcal{S}})$ would be on the same part and therefore be the root of the sender-path. The *sender-subtree* $\mathcal{T}_p^{\mathcal{S}}$ is the subtree of the forest $\mathcal{F}_p$ that is rooted at the sender-root.

The next theorem provides that two roots on the same tree level of a process share a common predecessor.

Theorem 2 (Root's predecessors)**.** *Let $\mathcal{D}$ be a depth-first distributed tree. Then, any two roots $r_1, r_2 \in \mathcal{R}_p$ with $t(r_1) = t(r_2)$ have the same predecessor, $\pi(r_1) = \pi(r_2)$.*

Proof. Nothing is to show for $r_1 = r_2$. Thus, let $r_1 < r_2$ without loss of generality, let $\pi_1 := \pi(r_1)$, $\pi_2 := \pi(r_2)$, and assume that $\pi_1 \neq \pi_2$. The relation $t(r_1) = t(r_2)$ implies $t(\pi_1) = t(\pi_2)$, and from Prop. 1(b) we conclude that π_2 belongs neither to the path $\Pi(\pi_1)$ nor to the subtree $T(\pi_1)$. Applying Prop. 4 gives $\pi_1 < \pi_2$, and by Prop. 3 we have $j < \pi_2$ for all $j \in V(\pi_1)$. Since $r_1 \in V(\pi_1)$, Prop. 5 leads to the order $P(\pi_1) < P(r_1) \leq P(\pi_2) < P(r_2)$, contradicting the assumption that $P(r_1) = P(r_2)$. Hence, $\pi_1 = \pi_2$. □

The remaining results state relations between root levels and depth-first orders on the distributed tree parts. They also highlight the special case of the sender-root.

Theorem 3 (Descending root levels)**.** *Let $\mathcal{D}$ be a depth-first distributed tree. Then for any two roots $r_1, r_2 \in \mathcal{R}_p$ with $r_1 \leq r_2$ it is $t(r_1) \geq t(r_2)$.*

Proof. The claim is obvious for $r_1 = r_2$. Therefore, consider the case $r_1 < r_2$ and assume $t(r_1) < t(r_2)$. With $\pi_1 := \pi(r_1)$ and $\pi_2 := \pi(r_2)$ we have $t(\pi_1) < t(\pi_2)$, and thus $\pi_2 \notin \Pi(\pi_1)$. As in the proof of Thm. 2, Prop. 4 yields $\pi_1 < \pi_2$, and Prop. 3 together with Prop. 5 leads to the order $P(\pi_1) < P(r_1) \leq P(\pi_2) < P(r_2)$, which contradicts $P(r_1) = P(r_2)$. Hence, $t(r_1) \geq t(r_2)$. □

Theorem 4 (Sender-root level)**.** *Let $\mathcal{D}$ be a depth-first distributed tree. Then, if the sender-root $r_p^{\mathcal{S}}$ exists, it has the lowest tree level on $\mathcal{P}_p$,*

$$t(r_p^{\mathcal{S}}) = \min\{t(v)\colon v \in V_p\}.$$

Proof. It suffices to show that $r_p^{\mathcal{S}}$ has the minimum tree level among the roots $\mathcal{R}_p$, since for $v \notin \mathcal{R}_p$ we have $\pi(v) \in V_p$ and $t(\pi(v)) < t(v)$. For $|\mathcal{R}_p| = 1$ there is nothing to show. For $|\mathcal{R}_p| > 1$ consider a root $r \in \mathcal{R}_p$ with $r \neq r_p^{\mathcal{S}}$ and assume that $t(r) < t(r_p^{\mathcal{S}})$. From Thm. 3 we conclude that $r > r_p^{\mathcal{S}}$. As a root, r does not belong to the subtree $\mathcal{T}_p^{\mathcal{S}}$ of $T(r_p^{\mathcal{S}})$. With $V(s) \subseteq V(r_p^{\mathcal{S}})$ for any sender $s \in \mathcal{S}_p$, Prop. 6 thus yields the contradiction $P(r) > P(r_p^{\mathcal{S}})$. Hence, $t(r) \geq t(r_p^{\mathcal{S}})$. □

Corollary 1. *Let $\mathcal{D}$ be a depth-first distributed tree, and suppose that the sender-root and a further root on the same level exist, i.e., $r_p^{\mathcal{S}}, r \in \mathcal{R}_p$ with $r \neq r_p^{\mathcal{S}}$ and $t(r) = t(r_p^{\mathcal{S}})$. Then we have $r < r_p^{\mathcal{S}}$.*

Proof. The argument for this proof is similar the proof of Thm. 4. Since $r \in \mathcal{R}_p$ is a root, it cannot belong to the subtree $T(r_p^{\mathcal{S}})$. With Prop. 3 and Prop. 5, the assumption $r > r_p^{\mathcal{S}}$ yields the contradiction $P(r) > P(r_p^{\mathcal{S}})$. Hence, $r < r_p^{\mathcal{S}}$. □

4.6. **Distributed DFS-Based Tree Algorithms.** A distributed DFS-based tree algorithm is a distributed extension of the sequential DFS-based tree algorithm described in Sect. 4.2. Each process p applies the underlying inward or outward algorithm to every subtree $\mathcal{T}_{p,r}$ in its forest $\mathcal{F}_p$. That is, for every subtree a depth-first traversal is performed and the operations $\mathrm{OP}(j)$ are executed upon discovering or finishing node j, respectively. This way, the overall requirements on the computation order are guaranteed. Clearly, communication routines are invoked whenever data need to be transmitted via loose edges.

There are two types of flexibility in the computation order of a process. First, the subtrees $\mathcal{T}_{p,r}$ may be visited in arbitrary order. Second, every subtree admits several equivalent depth-first traversals with different post-orders. We use both types of flexibility to reduce idle times. In inward algorithms, we iterate over the subtrees $\mathcal{T}_{p,r}$ in ascending order with respect to the root number r. The nodes of every subtree are then processed in the induced order. The rationale is to process the senders at the latest possible time since they require data from other processes. Therefore, the sender-subtree $\mathcal{T}_p^{\mathcal{S}}$ will be processed last. Proposition 7 provides that nodes with a lower number have a higher priority, hence we apply the induced computation order to the sender-subtree. For the remaining subtrees the applied computation order does not affect the idle times, so that there is no harm in using the induced order as well. Applying the node operations in opposite direction to the tree T (cf. Sect. 4.2) implies that for two senders $s_1, s_2 \in \mathcal{S}_p$ with $t(s_1) < t(s_2)$ the node s_2 will be processed first, and therefore the nonlocal data from the adjacent roots $r \in S(s_2)$ are required earlier. From Thm. 3, Thm. 4, and Prop. 1 we conclude that the subtrees $\mathcal{T}_{p,r}$ with lower root numbers have higher priority.

In outward algorithms the computation order criteria are exactly the opposite. That is, the senders should be processed at the earliest possible time and subtrees with a lower tree level should be prioritized since the nonlocal data that they require are sooner available. This leads to the following computation order for outward algorithms. We iterate over the subtrees in descending order with respect to the root number. The nodes of every subtree are then processed in the reverse order.

Finally, all nonlocal read operations in our DFS-based algorithms share the property that any two nodes j_1, j_2 with a common predecessor i need the same nonlocal data for their operation. For instance, Step 10 of the outward substitution

is a nonlocal read operation that takes the form

$$x_{j_1} \mathrel{+}= G_{j_1} x_i,\ x_{j_2} \mathrel{+}= G_{j_2} x_i \quad \text{for } j_1, j_2 \in V \quad \text{with} \quad \pi(j_1) = \pi(j_2) = i. \tag{9}$$

On the other hand, nonlocal write operations always modify nonlocal data in an additive way. For instance, step 10 of the factorization takes the form

$$H_i \mathrel{+}= G_{j_1}^T H_{j_1} G_{j_1} + G_{j_2}^T H_{j_2} G_{j_2} \quad \text{for } j_1, j_2 \in V \quad \text{with} \quad \pi(j_1) = \pi(j_2) = i. \tag{10}$$

These common properties can be exploited to reduce the number of communication calls during a distributed DFS-based algorithm. Consider two roots $r_1, r_2 \in \mathcal{R}_{p_1}$ of a distributed tree $\mathcal{D}$ with $\pi(r_1) = \pi(r_2) = s \in \mathcal{S}_{p_2}$. For the read operation (9), the data x_s need to be transmitted from process p_2 to p_1. Since the read operations in $\mathrm{OP}(r_1)$ and $\mathrm{OP}(r_2)$ need the same data, it suffices to send x_s to p_1 only once. For the write operation (10), we accumulate $\bar{H}_s = G_{r_1}^T H_{r_1} G_{r_1} + G_{r_2}^T H_{r_2} G_{r_2}$ on process p_1 and then send the result $\bar{H}_s$ to process p_2. This way the number of communication calls has been reduced to one. Informally speaking, the loose edges $(s, r_1), (s, r_2) \in \mathcal{E}$ have been merged into one. This reduction extends to any number of roots with a common predecessor.

Actually, the post-distribution communication reduction just described benefits from distributing the original tree in depth-first order. Theorem 2 provides that two roots $r_1, r_2 \in \mathcal{R}_p$ have the same predecessor *whenever* they are on the same tree level. Together with the root level order provided by Thm. 3, this guarantees that the communication between two processes during a distributed DFS-based tree algorithm requires at most one communication call per tree level.

5. A Distributed Solver for Tree-Sparse Problems

As outlined in Sect. 2, we use a primal-dual interior-point method to solve the tree-sparse NLPs (or QPs) presented in Sect. 3.1. As usual, the memory requirements and computational cost of the interior-point method are dominated by storing and solving the KKT system (3). Our parallelization of the solution algorithm is based on a static single-program-multiple-data (SPMD) programming model. In SPMD models, the same parallel program is executed by all participating processes where each process works asynchronously on a different part of the data (see, e.g., [17]).

In our SPMD approach, a static partitioning of the problem data is induced by the distribution of the underlying tree T over the processes (see Sect. 4.3). Memory overhead is only caused by

- local buffers for the communication routines;
- local copies of global vector data (cf. Sect. 4.2);
- local workspace for certain interior-point operations.

Most of the interior-point operations decompose into sets of node operations $\mathrm{OP}(j)$ for $j \in T$ (cf. Sect. 4.2). Vector operations such as vector addition and norms, for example, inherit a natural decomposition induced by the data distribution. The node operations $\mathrm{OP}(j)$ are assigned to the processes in the same way as the problem data. Computational overhead is caused by communication routines and by nondecomposable interior-point operations, which are simply duplicated on each process. Altogether, the resulting interior-point method for tree-sparse NLPs is *entirely* distributed.

In the following, we describe the distributed execution of all interior-point operations. Section 5.1 first presents the distributed versions of the tree-sparse KKT algorithms. The remaining interior-point operations are then discussed in Sect. 5.2.

TABLE 2. Distributed KKT solution for the incoming control case

	Factorization (↓)	Inward subst. (↓)	Outward subst. (↑)
i:	recv. $\leftarrow H_j^p, F_j^p, X_V^p$	recv. $\leftarrow f_j^p, e_V^p$	send $\rightarrow x_j, \mu$
1:	K_j += Φ_j^u		$-v_{ij} \leftarrow \Psi_{ij}(-v_{ij})$
	⋮	⋮	⋮
17:	X_V += $D_j D_j^T$	e_V -= $D_j d_j$	u_j += $D_j^T(-\mu)$
ii:	send $\leftarrow H_i, F_i, X_V$	send $\leftarrow f_i, e_V$	recv. $\rightarrow x_i, -\mu$
18:	$X_V \leftarrow LL^T$	$e_V \leftarrow L^{-1}e_V$	$-\mu \leftarrow L^{-T}(-\mu)$

5.1. **Distributed Tree-Sparse KKT Solver.** The tree-sparse KKT solver consists of three phases: two inward algorithms (factorization and inward substitution) and one outward algorithm (outward substitution). Based on the theoretical results for DFS-based tree algorithms of Sect. 4, we now describe the distributed versions of the tree-sparse KKT algorithms.

The distributed factorization proceeds as follows. Every process iterates over its subtrees in descending root level order and applies the tree-sparse factorization of Table 1 to each subtree. For senders $j \in \mathcal{S}$, nonlocal data are received from successor nodes and accumulated with the local node data *before* the node operations are executed. As discussed in Sect. 4.6, there is at most one incoming communication from each process p, which we designate by superscript p. The accumulation operations read H_j += $\sum_p H_j^p$, F_j += $\sum_p H_j^p$, and X_V += X_V^p. Steps 1 to 17 are then executed exactly as in the sequential case. If the current node is a root, $j \in \mathcal{R} \setminus \{0\}$, the nonlocal data H_i, F_i, and X_V are sent to the predecessor. Step 18 is only executed at the global root $j = 0$.

The distributed inward substitution works the same way as the factorization. The distributed outward substitution proceeds in the opposite direction. At the global root $j = 0$, step 18 yields the global Lagrange multiplier μ. At other roots $j \in \mathcal{R} \setminus \{0\}$, the nonlocal data μ, x_i are received from the predecessor. Steps 17 to 1 are then executed as in the sequential case. If the current node is a sender $j \in \mathcal{S}$, x_j and μ are sent to the successors. With post-distribution communication reduction (cf. Sect. 4.6) this requires at most one communication for each process. The complete distributed tree-sparse KKT solver for the incoming control case is presented in Table 2.

5.2. **Distributing the Interior-Point Method.** In addition to solving the KKT system (3), the interior-point method involves matrix-vector products with the KKT matrix (or submatrices thereof), the calculation of primal and dual step sizes (4) and iterates (5), evaluations of norms (6), and an update of the barrier parameter β^k. These operations consist of elementary operations that can be classified into

- problem-specific matrix-vector algorithms (product with KKT matrix and KKT system solution);
- vector-valued vector operations (such as addition and scalar multiplication);
- collective operations (such as norms and scalar products);
- scalar operations.

The tree-sparse KKT matrix-vector product is implemented as a DFS-based inward algorithm whose distributed version proceeds the same way as the factorization and the inward substitution described in Sect. 5.1.

The primal-dual update (5) requires only vector-valued vector operations. Since the vector data are distributed along with the nodes of the underlying tree (Sect. 4.3),

every process operates on its own part of the data and the parallel execution does not require any communication between the processes.

The update of β^k is a scalar operation (which may be based on results of more complex operations, of course). In our SPMD programming model, such scalar operations are sequential operations, which are simply duplicated on each process.

Another relevant sequential operation is the synchronization of collective operations. Scalar-valued vector operations such as the step length calculation (4) or the vector norms in (6), for instance, create a single scalar value from one or more vectors. In the distributed case, each process creates a scalar value for its part of the vector, and all these values are finally gathered into a single value. This involves a so-called *reduce-and-scatter* communication routine. The process values are brought together on a dedicated *root process* (in our case the process that owns the root node 0), and the merged result is then sent to all non-root processes.

5.3. **Numerical Issues.** In the context of multistage stochastic optimization, the collective operations often evaluate expected values over the scenario tree. Thus we obtain the sum of a potentially huge number of values, each weighted with the associated node probability p_j of magnitude $\sim 1/|L_t|$. The resulting terms will usually have decreasing magnitude with increasing tree level t, so that we need to address the effects of rounding errors. Clearly, the sum should be formed level-wise in this situation. Therefore we perform a DFS-based inward traversal over each subtree, and for each node j we calculate the weighted sum over the successors $k \in S(j)$ as an intermediate result before adding it to the local term.

6. Computational Results

In the following, we present computational results with our distributed solution algorithm by solving a series of huge-scale portfolio optimization problems. After stating the computational setup and introducing the portfolio test sets (Sect. 6.1), we first discuss some general observations (Sect. 6.2). Then we analyze the parallel performance of our algorithms in detail, beginning with the distributed KKT solver (Sect. 6.3) and considering the entire IPM algorithm afterwards (Sect. 6.4). Finally, we discuss the effects of the post-distribution communication reduction on the trees occurring in the portfolio test sets (Sect. 6.5).

6.1. **Computational Setup.** All computations are performed on the compute cluster of the Institute of Applied Mathematics at Leibniz Universität Hannover. The cluster consists of 8 compute nodes with an InfiniBand interconnection and 48 GiB RAM each. Four of the compute nodes have 8 Intel(R) Xeon(R) X5570 cores each running at 2.93 GHz, the other four nodes have 12 X5675 cores each running at 3.07 GHz. In total, the cluster comes with 80 cores and 384 GiB RAM.

In our computations, we run at most one process per core, and we always use identical numbers of cores per compute node to obtain a uniform distribution of the memory requirements, leading to a maximum of 64 used cores.

The software is written in C++11 using the standard C++ library and Boost [21]. Linear algebra operations are based on standard BLAS [13] and LAPACK [1] routines. The parallel implementation uses the Message Passing Interface (MPI) employing Open MPI [22] through the C++ interface of Boost.MPI. The optimization problems are solved by the generic interior-point code Clean::IPM [18] in combination with the tree-sparse KKT solver.

The performance of parallel algorithms is measured by means of the *speedup* $S_{n_p} := t_{\text{seq}}/t_{n_p}$ and the *efficiency* $E_{n_p} := S_{n_p}/n_p$. Here, t_{seq} is the runtime of the sequential algorithm and t_{n_p} is the runtime of the parallel algorithm on n_p processes. Up to inaccuracies in measured computing time, the speedup is bounded by the

number of processes. Hence, it is perfect if $S_{n_p} = n_p$ with perfect efficiency $E_{n_p} = 1$. For more details on performance analysis of parallel programs the reader is referred to standard textbooks such as [17].

We monitor wall-clock times $t^{\text{wc}}_{n_p}$ and *accumulated* CPU times $t^{\text{cpu}}_{n_p}$ of the respective algorithms. Since we do not have a dedicated compute cluster, the measured times will usually be too large because of other users' processes. To reduce that systematic bias, each measurement is executed repeatedly and we take the best result. In our results, the speedup is given with respect to wall-clock time, $S_{n_p} = t^{\text{wc}}_{\text{ref}}/t^{\text{wc}}_{n_p}$. As reference we take the best wall-clock time normalized to one process, $t^{\text{wc}}_{\text{ref}} = \min_{n_p}\{n_p t^{\text{wc}}_{n_p}\}$. The efficiency is computed with respect to CPU time, $E_{n_p} = t^{\text{cpu}}_{\text{ref}}/t^{\text{cpu}}_{n_p}$, where the reference time is $t^{\text{cpu}}_{\text{ref}} = \min_{n_p}\{t^{\text{cpu}}_{n_p}\}$.

For generating test problems, we consider artificial portfolios with n_a risky assets classified into three risk categories: low, medium, and high. We model the development of stock prices by an n_a-dimensional geometric Brownian motion and apply a time discretization yielding multivariate lognormal return distributions in every time step t, with first and second moments μ_t and Σ_t, respectively. We perturb these moments randomly in every nonleaf node j of the scenario tree, $\mu_j = \mu_{t(j)} + \delta\mu_j$ and $\Sigma_j = \Sigma_{t(j)} + \delta\Sigma_j$, and approximate each perturbed distribution by a uniform discrete distribution. A nondegenerate discrete approximation requires $n_a + 1$ (or more) realizations $\{r_k\}_{k\in S(j)}$ that should reproduce the two moments,

$$\mu_j = \sum_{k\in S(j)} p_k r_k, \quad \Sigma_j = \sum_{k\in S(j)} p_k (r_k - \mu_j)(r_k - \mu_j)^T, \quad p_k = \frac{1}{|S_j|}. \tag{11}$$

For the large number of scenarios to be generated here, we avoid solving the feasibility problem (11) in every node. Instead we use a simple heuristic that matches only the first moments μ_j exactly. Comparisons in [11] confirm the expectation that this has no measurable influence on the computational performance.

The size of a portfolio optimization problem is determined by the depth d of the scenario tree and the number of assets n_a. In addition, the shape of the trees depends on n_a since every node has $n_a + 1$ successors. We consider the two portfolio test collections PT1 and PT2 shown in Table 3, with rows sorted by decreasing tree depth and increasing number of assets, and hence by increasing workload per node. The instances of collection PT1 feature the largest combinations of d and n_a that can be run in the memory of a single compute node, whereas all instances in collection PT2 require the memory of all eight compute nodes. In total, we run three test series: collection PT1 on one compute node (shared memory system), and both collections on all eight compute nodes (distributed memory system).

The tests in collection PT2 require two cores or more per compute node: on a single core, the lengths of the data arrays for the KKT matrix and its factorization often exceed the integer range that is available in BLAS and LAPACK ($2^{31}-1 \approx 2.15 \times 10^9$). Table 3 lists the number of nonzero entries of the KKT matrix and its factors for both collections. Note further that our implementations of the tree-sparse KKT matrix and its factorization do not exploit the application-specific sparsity of blocks associated with a node j. Specifically, the KKT matrix stores in dense format every block except for the identity I in (7b). This way, the performance results reported here depend only on tree sizes and block dimensions. Corresponding results for an application-specific implementation that does exploit the sub-block sparsity are given in [11]. Moreover, the matrix factor stores only those blocks that differ from the original matrix, i.e., the Hessian blocks K_j, H_j, J_j in (8a) and global constraints blocks F_j, D_j in (7e). The dynamics blocks G_j, E_j in (7b) and range constraints blocks F^r_{ij}, D^r_j, F^r_j in (8b) remain unmodified. As a consequence, the factor requires less memory than the KKT matrix.

TABLE 3. Portfolio test collections PT1 and PT2: tree depth (d), number of assets (n_a), and resulting numbers (all in millions) of tree nodes, scenarios, variables, equality and inequality constraints, and stored nonzero entries of the KKT matrix and its factor

No.	d	n_a	Nodes	Scen.	Var.	Eq.	Ineq.	Matrix	Factor
					PT1				
1	11	3	5.59	4.19	50.33	16.78	16.78	503.32	301.99
2	9	4	2.44	1.95	29.30	9.77	12.21	385.74	219.73
3	8	5	2.02	1.68	30.23	10.01	12.09	483.73	272.10
4	7	7	2.40	2.10	50.33	16.78	16.78	1073.74	603.98
5	6	9	1.11	1.00	30.00	10.00	10.00	810.00	450.00
6	5	14	0.81	0.76	34.17	11.39	10.58	1395.35	768.87
7	4	23	0.35	0.33	23.89	7.96	6.58	1560.67	859.97
					PT2				
1	12	3	22.37	16.78	201.33	67.11	67.11	2013.27	1207.96
2	10	4	12.21	9.77	146.48	48.83	61.04	1928.71	1098.63
3	8	7	19.17	16.78	402.65	134.22	134.22	8589.93	4831.84
4	7	9	11.11	10.00	300.00	100.00	100.00	8100.00	4500.00
5	6	12	5.23	4.83	188.23	62.75	57.52	6619.97	3670.79
6	5	19	3.37	3.20	192.00	64.00	53.89	10432.00	5760.00
7	4	33	1.38	1.34	136.31	45.43	34.42	12585.61	6951.62

TABLE 4. Wall-clock reference times ($t_{\text{ref}}^{\text{wc}}$) in seconds for one IPM iteration consisting of tree-sparse (sub-)algorithms factorization, inward substitution (In), outward substitution (Out), matrix-vector product (MVP) and remaining IPM operations (Rest)

	PT1					PT2				
No.	Fact.	In	Out	MVP	Rest	Fact.	In	Out	MVP	Rest
1	12.95	3.60	3.39	1.37	29.91	56.88	13.23	15.71	5.78	126.97
2	8.99	1.87	2.43	0.75	17.59	45.42	9.58	10.73	3.83	91.23
3	10.67	1.98	1.99	0.72	18.33	215.82	29.70	27.60	9.15	246.95
4	24.32	3.63	3.32	1.12	29.97	196.74	21.94	21.21	6.37	180.43
5	19.32	2.14	2.10	0.63	17.75	195.80	14.17	15.35	3.80	114.91
6	42.90	2.59	2.76	0.68	20.56	414.33	16.71	17.58	4.15	123.49
7	67.38	2.27	2.39	0.55	15.53	756.12	17.42	17.89	4.60	103.72

6.2. **General Observations.** The instances in collections PT1 and PT2 are solved with our code Clean::IPM using an error tolerance of 10^{-6}. For the barrier parameter updates we use Mehrotra's predictor-corrector method [15]. We measure the runtimes of the entire interior-point iteration (excluding the initialization stage) and of the four tree-sparse subalgorithms: factorization, inward and outward substitutions, and matrix-vector product. This leaves certain vector operations and scalar operations as remaining IPM operations. As wall-clock reference times $t_{\text{ref}}^{\text{wc}}$ for the speedup we list in Table 4 the average per-iteration runtimes (taken from the fastest test run) for each of the tree-sparse subalgorithms and for the remaining operations. Note finally that runtimes for the matrix vector product are actually average values of three variants: the products with the first block column of the KKT matrix in (3)

TABLE 5. Wall-clock runtimes ($t^{\text{wc}}_{n_p}$) in seconds with numbers of IPM iterations for different numbers of cores and compute nodes

		PT1 (shared memory)				PT1 (distributed)			PT2 (distributed)		
No.	Iter.	1	2	8	12	8×1	8×2	8×8	Iter.	8×2	8×8
1	13	912	447	150	120	117	66	23	13	241	86
2	13	544	270	92	79	71	38	14	14	188	72
3	11	495	248	85	70	64	33	12	10	426	177
4	9	809	375	126	102	98	53	23	9	319	119
5	6	351	172	59	49	45	23	10	7	188	73
6	8	766	358	117	96	93	47	17	6	361	119
7	6	716	341	106	80	90	45	14	8	552	178

consisting of $H^k, C^k_{\mathcal{E}}, C^k_{\mathcal{I}}$, and with each of the two transposed constraints matrices $(C^k_{\mathcal{E}})^T, (C^k_{\mathcal{I}})^T$.

The tests in PT1 are run on a single compute node using 1 to 12 cores with shared memory, and on the distributed memory system consisting of all eight compute nodes using 1 to 8 cores per node. The tests in PT2 are run on all eight compute nodes using 2 to 8 cores per node. The total runtimes $t^{\text{wc}}_{n_p}$ with respective numbers of IPM iterations are given in Table 5 for nine selected configurations.

The data in Table 5 show that every test instance is solved in less than three minutes when all available resources are used (12 cores on the shared memory system or 8×8 cores on the distributed memory system). Moreover, the results demonstrate a significant performance boost due to the parallelization. For collection PT1, the 12-core runtimes t^{wc}_{12} are 83 % to 89 % smaller than the corresponding sequential runtimes t^{wc}_1, and the 64-core runtimes t^{wc}_{64} are up to 98 % smaller. For collection PT2, the reductions from t^{wc}_{16} to t^{wc}_{64} lie in the range 59 % to 67 %.

More detailed discussions of the parallel performance of the tree-sparse algorithms and the IPM will be given in the subsequent sections. Here we wish to highlight two striking observations concerning the results of PT1. First, all instances feature a superlinear speedup when moving from one to two cores: $2t^{\text{wc}}_2 < t^{\text{wc}}_1$. Superlinear speedup is occasionally reported in the literature, specifically also in the related work of Gondzio and Grothey [9]. In our case the effect is probably caused by larger access times for larger data arrays: in PT1 their lengths are inversely proportional to the number of processes. Second, the runtimes t^{wc}_8 of all instances are significantly larger on the shared memory system than on the distributed memory system (8×1). Usually one would expect the opposite observation since communication in a distributed system is slower than in a shared memory system. In addition, the shared memory tests are run on one of the 12-core compute nodes, which are slightly faster than the 8-core nodes. Hence, this effect demonstrates that our parallel algorithm has relatively small communication overhead whereas its computational performance apparently suffers when several processes have to share the hardware of a single compute node.

6.3. **Parallel Performance of the KKT Solver.** Here we discuss the tree-sparse KKT solver consisting of the factorization and inward/outward substitutions. In all three test series (PT1 shared and distributed, PT2), the parallel performance is qualitatively similar.

Let us first consider the values of speedup S_{n_p} and efficiency E_{n_p} for test collection PT2 given in Figure 2. As one would expect, the factorization offers the best parallel performance, with almost linear speedup and all efficiency values lying above 94 %. With one Cholesky factorization and several level 3 BLAS operations (matrix-matrix

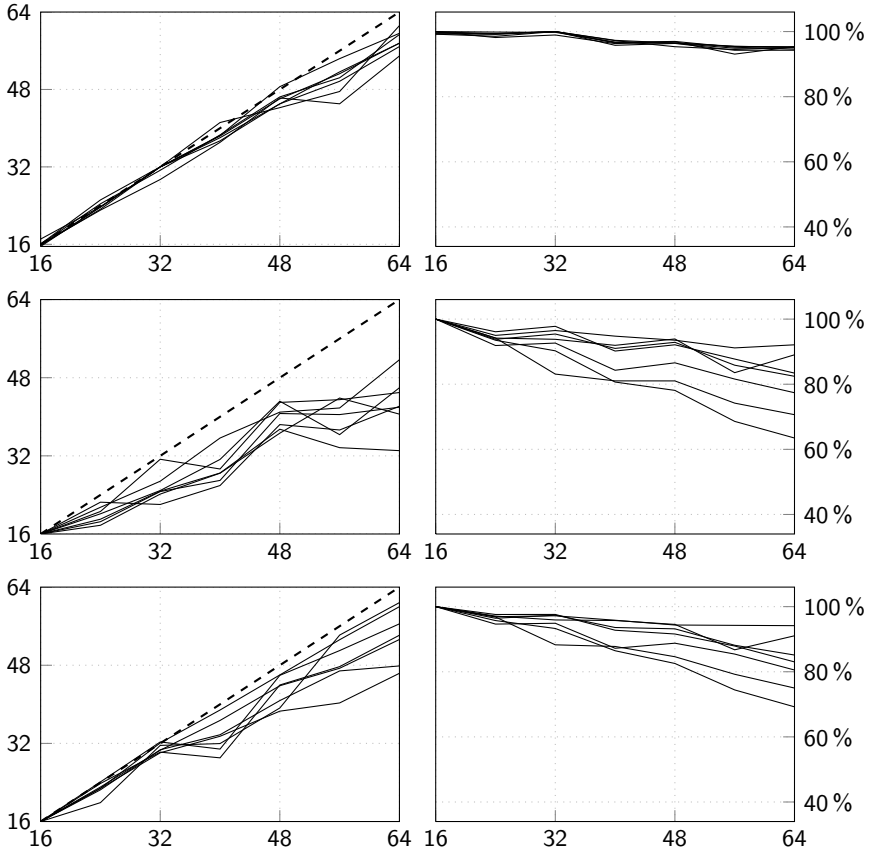


FIGURE 2. PT2: speedup (left) and efficiency (right) of factorization, inward substitution, outward substitution (top to bottom)

products) performed on every node in the tree, the factorization features a heavy workload and hence relatively small communication and idle times. The tree-sparse substitutions perform less favorable than the factorization but still good: we observe efficiency values above 70 %, with just one exception. Here the workload consists only of level 1 and level 2 BLAS operations and is significantly smaller than in the factorization, so that the communication and idle times are no longer negligible. Although the workload per node increases monotonously from instance 1 to instance 7, the substitutions perform best on the first three instances in the collection. This surprising result can be explained by observations in Sect. 6.5: the communication overhead is almost uniformly balanced over the processes in the first instances and rather unbalanced in the last ones.

The performance values for test collection PT1 are illustrated in Fig. 3, comparing the efficiency on the shared memory system (left) with the distributed system (right). Regarding the relative performance of the three tree-sparse algorithms, the same observations as for collection PT2 apply, see Fig. 2. Moreover, as before one observes that the instances of PT1 perform better on a distributed system (8×1) than on a shared memory system with the same number of cores.

The performance results of PT1 and PT2 demonstrate that our parallel approach scales well in two ways: First, if we fix the problem size and increase the number of processes (left vs. right sides in Fig. 3), the efficiency values remain relatively high for both shared and distributed memory systems. Furthermore, the efficiency curves

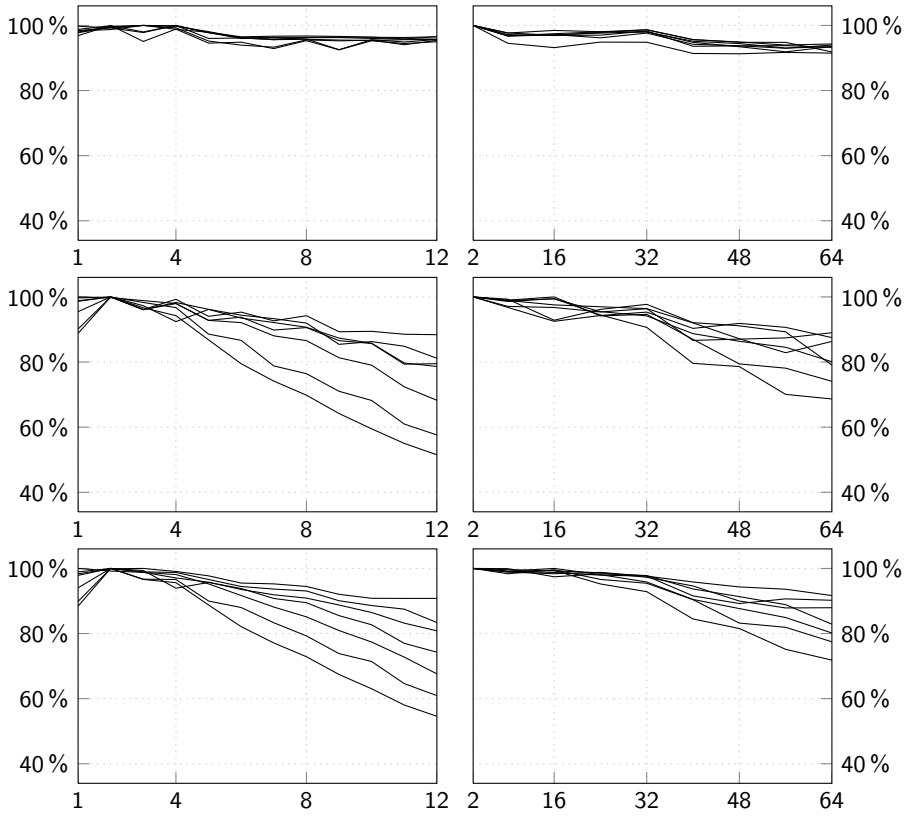


FIGURE 3. PT1: efficiency of factorization, inward substitution, outward substitution (top to bottom) on shared memory system (left) and distributed memory system (right)

show that the performance decrease does not depend on the absolute number of processes but rather on the ratio of used and available processes, i.e., the saturation of hardware resources. Second, if we fix the number of processes and increase the problem sizes (Fig. 3 right-hand side vs. Fig. 2), the performance on the smaller problems in PT1 is as good as the performance on the huge problems in PT2.

Let us finally point out that our parallel approach and its implementation are highly memory efficient, with virtually negligible memory overhead. All tree-sparse algorithms share a common workspace W for intermediate results, a second common workspace L for the level-wise summation (cf. Sect. 5.3), and a common communication buffer B for data transmission. The sizes of W, L, B are determined by the algorithm with the highest memory demand, i.e., the factorization. With $|\cdot|$ denoting the memory requirements of a workspace, buffer, or node sub-block, the required sizes are given by

$$\begin{aligned} |W| &= 4|H_{\max}| + 2|K_{\max}| + |J_{\max}| + 2|F^r_{\max}| + |D^r_{\max}| \\ &\quad + 4|x_{\max}| + 4|u_{\max}| + 2|v_{\max}|, \\ |B| &= |H_{\max}| + |F_{\max}| + 2|X_V|, \\ |L| &= d|X_V|, \end{aligned}$$

where the subindex "max" refers to the largest sub-block or vector of all tree nodes. The resulting memory overhead for parallel execution with n_p processes is

$$(n_p - 1)(|W| + |L|) + n_p|B|.$$

Thus, our memory overhead per process grows only linearly with the tree depth d, whereas the overall memory requirements grow linearly with the total number of tree nodes, $|V|$. Moreover, the absolute sizes of W, L, B are extremely small: the largest overhead in all our tests occurs for instance 7 in PT2, with $|W| = 132\,\text{KiB}$, $|L| = 0\,\text{KiB}$, and $|B| = 16\,\text{KiB}$, yielding a total overhead of 9.25 MiB for 64 processes, or 1.16 MiB per 48 GiB compute node.

The memory efficiency just described is beneficial in two ways. First, the memory requirements of a given problem do not increase significantly with the number of processes, which is in stark contrast to other approaches. The authors of [14], for example, reserve a fixed buffer of 250 MB per process. Second, due to our static data distribution, the memory requirements for each process are fixed and known a priori. Therefore, in our case there is no need to measure a peak memory per process as it is done in dynamic parallel approaches such as [6] and [14]. For instance, the peak memory reported in [6] grows from 6816 MB for 16 processes to 23 552 MB for 512 processes (a factor of 3.46), and the peak memory reported in [14] grows from 116 352 MB for 64 processes to 1 071 104 MB for 2048 processes (a factor of 9.21).

6.4. **Parallel Performance of the IPM.** In the following, we discuss the performance of the entire IPM and the relative performance of its principal components: the KKT solver, matrix-vector product, and remaining IPM operations. As before, the results for the three test series are qualitatively similar. We focus here on the performance of PT2, for which Fig. 4 illustrates the values of speedup (left) and efficiency (right) of a single tree-sparse KKT solution, a single matrix-vector product, and the remaining operations of a single IPM iteration.

The KKT solver (Fig. 4, top) clearly performs best, with efficiency values lying above 73 %. As already discussed, the best performance values are achieved for instances with heavy workload per node, specifically for instance 7.

The performance of the tree-sparse matrix-vector product (Fig. 4, middle) is significantly lower than the performance of the KKT solver. Here some of the efficiency values go down below 60 %. As one would expect, the matrix-vector product performs similar to the tree-sparse inward and outward substitutions (though slightly less favorable): its node operations also consist of level 1 and level 2 BLAS operations, but its workload per tree node is lower than that of the substitutions. (Recall that we average over three products, all of which involve only parts of the KKT matrix.) Consequently, the best MVP performance is achieved on the same test instances as for the substitutions, i.e., on instances 1 to 3.

Finally, the remaining IPM operations as a whole show the lowest performance (Fig. 4, bottom). The efficiency values decrease quickly below 80 %, with minimum values ranging from 54 % to 60 %. In addition to many fully parallel vector-vector operations, the IPM iteration also contains several sequential parts, specifically scalar operations and synchronizations. For example, 75 reduce-and-scatter communication calls are invoked during one IPM iteration in our setup, each of which requires synchronization. As a consequence, the parallel performance of the remaining IPM operations suffers noticeably from the poor ratio T_p/T_s of parallel vs. sequential algorithm parts, which yields an upper bound of the speedup S according to Amdahl's law, $S \leq 1 + T_p/T_s$. However, the parallel performance of the remaining IPM operations does not depend on the tree structure. Hence it is almost identical for all instances in the PT collections.

Figure 5 illustrates the performance of the entire interior-point solver Clean::IPM on the test collections PT1 and PT2. Here we take into account that certain IPM subalgorithms are executed several times: an IPM iteration requires just one factorization but two inward substitutions, two outward substitutions, and six

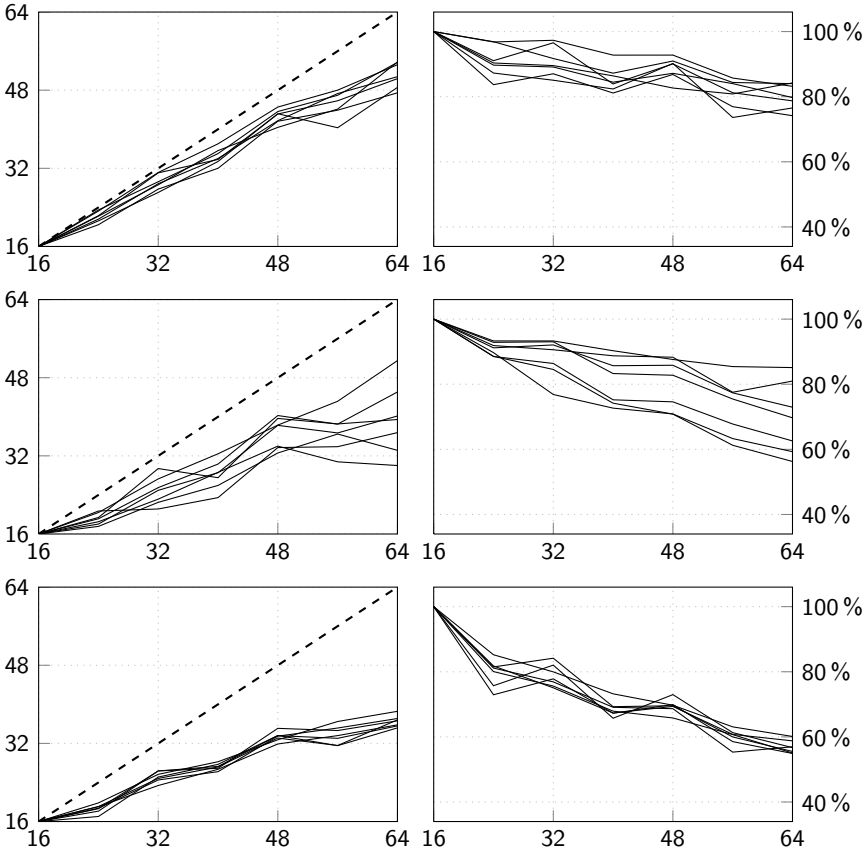


FIGURE 4. PT2: speedup (left) and efficiency (right) of KKT solver, matrix-vector product, remaining IPM operations (top to bottom)

matrix-vector products. The resulting runtime percentages of each subalgorithm and of the remaining IPM operations are given in Table 6.

As we can see, the IPM performs rather well when at most half of the available cores are in use (up to 6 on one compute node and up to 32 on 8 compute nodes), producing efficiency values above 80 %. When all available cores are in use, the performance is still good with most of the efficiency values over 60 %. The best performance results are again obtained on instances with a heavy workload per node, specifically instance 7 in both test collections. Table 6 shows that with increasing node dimensions (n_j^u, n_j^x, and l_j^r) the runtime percentage of the factorization increases while the respective percentages of inward and outward substitutions, matrix-vector-products and remaining IPM operations decrease.

The performance results of the entire distributed IPM show that the scalability is still quite good, although of course to a minor extent than for the KKT solver discussed in Sect. 6.3. Moreover, the distributed IPM remains highly memory efficient since the memory overhead is dominated by the tree-sparse algorithms. In addition to the workspaces and communication buffers for the tree-sparse algorithms, only scalars in the IPM are duplicated for each process.

6.5. **Post-Distribution Communication Reduction.** In this section we discuss the effects of the post-distribution communication reduction (PDCR), cf. Sect. 4.6. Considering the trees of the test problems in Table 3, we investigate the number of communication calls that can be saved during one run of a distributed DFS-based

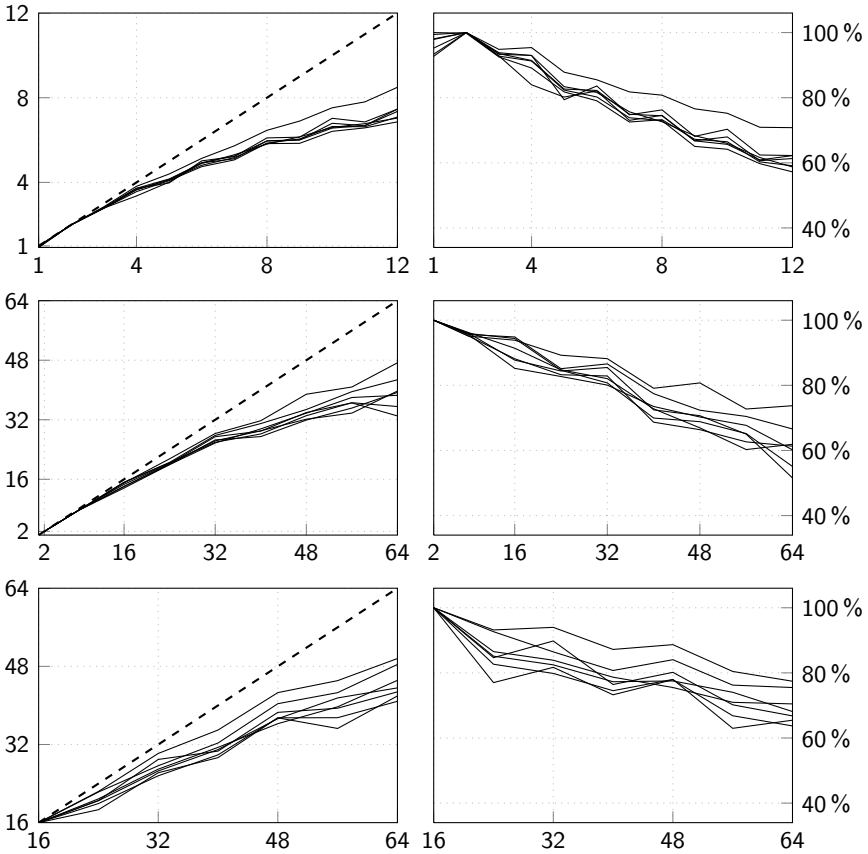


FIGURE 5. IPM performance results for PT1 on one compute node, PT1 on 8 compute nodes, PT2 on 8 compute nodes (top to bottom)

tree algorithm by merging data for the same destination or by sending the same data to a process only once. After partitioning a tree, we determine the numbers of senders $|\mathcal{S}|$ and roots $|\mathcal{R}|$ of the distributed tree and compare them to the number of communication calls n_c after applying the PDCR. By construction, there is at least one communication call per sender and, without PDCR, every root $r \in \mathcal{R} \setminus \{0\}$ causes exactly one communication call. The actual number of communication calls thus satisfies $|\mathcal{S}| \leq n_c \leq |\mathcal{R}| - 1$. We also count the number of communication calls of each individual process p, and the respective maximum numbers of inward and outward communication calls over all processes, $n_{\text{in}}^{\max}$ and $n_{\text{out}}^{\max}$.

Figure 6 shows the results for three selected trees that represent typical cases when partitioning the trees in PT1 and PT2 over an increasing number of processes. Results for the remaining trees can be found in [11]. Figure 6 demonstrates that the PDCR works very well: the number of communication calls is reduced substantially, yielding actual numbers n_c close to the lower bound $|\mathcal{S}|$. Furthermore, the PDCR approach is scalable and works even better with an increasing number of processes.

The results for instance PT2.5 exemplify an optimal exploitation of the PDCR. There is a large difference between the numbers of roots $|\mathcal{R}|$ and senders $|\mathcal{S}|$, and the number of communication calls n_c is close to its lower bound $|\mathcal{S}|$. Moreover, the maximum numbers of inward and outward communications, $n_{\text{in}}^{\max}$ and $n_{\text{out}}^{\max}$, are only small fractions of the overall number n_c, which implies that the communication calls are well-distributed among the participating processes.

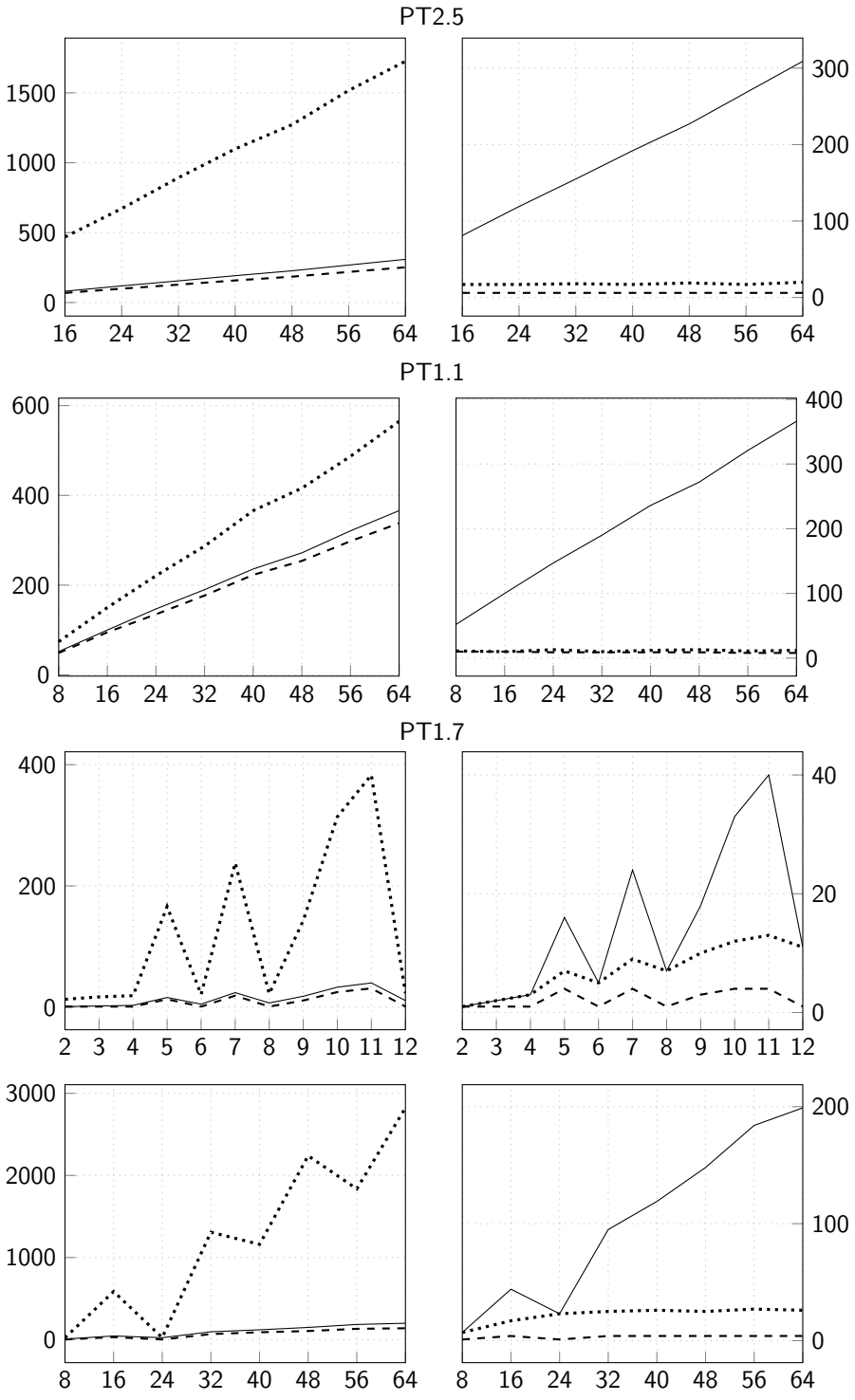


FIGURE 6. Effects of PDCR on selected trees. Left: number of communications with PDCR (solid line), without PDCR ($|\mathcal{R}| - 1$, dotted line) and lower bound ($|\mathcal{S}|$, dashed line). Right: maximum numbers of inward/outward communications ($n_{\text{in}}^{\max}$ dashed, $n_{\text{out}}^{\max}$ dotted) and number of communications with PDCR (solid line)

TABLE 6. PT1 and PT2: IPM statistics with wall-clock reference time $t_{\text{ref}}^{\text{wc}}$ (in seconds), per-node numbers of state variables (n_j^x), control variables (n_j^u), inequalities (l_j^{rx}), and runtime percentages of factorization (1F), inward substitutions (2I), outward substitutions (2O), matrix-vector products (6M), remaining operations (Rest)

No.	$t_{\text{ref}}^{\text{wc}}$ (s)	n_j^x	n_j^u	l_j^{rx}	1F	2I	2O	6M	Rest
				PT1					
1	64.16	3	6	3	20.18	9.54	10.89	12.77	46.62
2	38.83	4	8	5	22.91	9.65	10.52	11.62	45.29
3	41.28	5	10	6	25.88	9.62	9.65	10.41	44.45
4	74.92	7	14	7	32.46	9.69	8.87	8.98	40.01
5	49.42	9	18	9	39.15	8.70	8.52	7.65	35.98
6	78.24	14	28	13	54.83	9.96	7.05	5.21	26.28
7	95.66	23	46	19	70.54	4.75	5.01	3.44	16.26
				PT2					
1	275.11	3	6	3	20.68	9.62	10.96	12.60	46.15
2	200.26	4	8	5	22.68	9.57	10.72	11.47	45.56
3	632.29	7	14	7	34.13	9.39	8.73	8.69	39.06
4	501.69	9	18	9	39.22	8.75	8,46	7.62	35.97
5	391.84	12	24	11	49.79	7.23	7.83	5.81	29.33
6	631.29	19	38	16	65.63	5.29	5.57	3.94	19.56
7	954.84	33	66	25	79.19	3.65	3.75	2.55	10.86

The second set of diagrams in Fig. 6 shows the results when distributing tree PT1.1 over eight compute nodes. Here the PDCR is less effective since the numbers $|\mathcal{R}|$ and $|\mathcal{S}|$ are relatively close to each other. This is caused by the small branching number ($b = 3$) in comparison to the large depth of the tree ($d = 11$). Each part $\mathcal{P}_p$ of the distributed tree $\mathcal{D}$ (except for $\mathcal{P}_0$) consists of many small subtrees $\mathcal{T}_{p,r} \in \mathcal{F}_p$. However, the communication calls are again well-distributed among the participating processes.

Tree PT1.7 demonstrates the distribution of a tree that is relatively flat ($d = 4$) and widespread ($b = 24$). The PDCR is effective but a comparatively large number of outward communications are concentrated on the root process: many successors of the global root 0 belong to different processes, so that it is involved in many of the outward communications. The results for PT1.7 also show that the number of communication calls is small if the number of participating processes n_p coincides with the branching number b. In this case the tree is only split at the global root 0, and the total number of roots is $|\mathcal{R}| = n_p$. A similar effect also occurs several times in the collections PT1 and PT2 when the number of processes n_p is a multiple of the branching number b.

Finally, Fig. 7 gives a graphical overview of relative reductions with respect to the difference $|\mathcal{R}| - 1 - |\mathcal{S}|$ (the maximum possible reduction) for all three test series. Here we count seven instances with no reduction, one with about 10 %, 13 in the range 40 % to 80 %, and all the remaining 161 instances well above 80 %.

7. Conclusion

For multistage stochastic NLPs with large scenario trees but moderate numbers of variables and matrix entries per node we have presented a single-program-multiple-data parallelization of the KKT solver and remaining operations in an interior-point

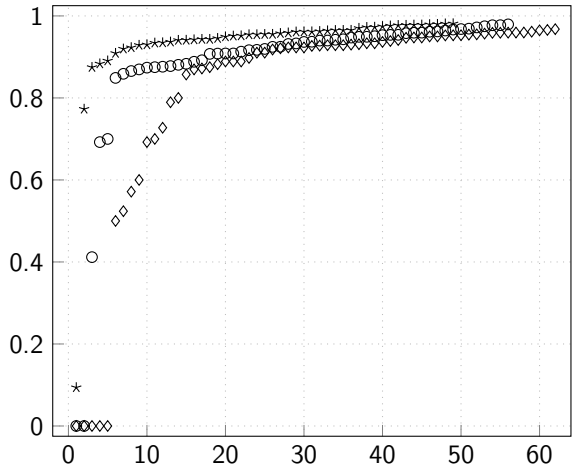


FIGURE 7. Relative reduction due to PDCR: PT1 on one compute node ($\diamond$), PT1 on eight compute nodes ($\circ$), and PT2 ($\star$)

method based on a static partitioning of the tree with associated NLP data. The distributed algorithm is highly scalable and well suited for distributed and shared-memory computing environments. In addition, as predicted by the theoretical analysis, it features very low communication and memory overheads that are known a priori. The low memory overhead resulting from the static tree distribution is probably the most distinctive feature of our approach. It is the major benefit in comparison to dynamic parallel approaches, and it fits well together with the low memory requirements of the underlying sequential algorithm. The author of [10] has already asked for a parallelization of our sequential algorithm, so that the investigation just presented fills a known gap in the literature.

One might also consider iterative solution methods for the huge KKT systems in multistage stochastic NLPs. In the test problems discussed above, however, a single direct KKT system solution costs only 15 to 170 averaged matrix-vector-products (MVPs) on one core and 2 to 17 MVPs on 64 cores. A complete Mehrotra IPM iteration requires two KKT solves with the same matrix, yielding cost ratios of the direct solver over two MVPs from 10 to 90 on one core and from 2 to 10 on 64 cores. It appears unrealistic to obtain sufficiently accurate KKT solutions iteratively with a comparable effort. Of course, this can change with different problem characteristics or when an exceptionally cheap preconditioner becomes available. For the time being, we believe that our parallel approach is among the most efficient ones for the problem class under consideration.

REFERENCES

[1] E. ANDERSON, Z. BAI, C. BISCHOF, J. DEMMEL, J. DONGARRA, J. du CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, S. OSTROUCHOV, and D. SORENSEN. *LAPACK Users' Guide*. `http://www.netlib.org/lapack`. SIAM. Philadelphia, PA, 1992.

[2] J. R. BIRGE and F. LOUVEAUX. *Introduction to stochastic programming*. Springer Science & Business Media, 2011. DOI: `10.1007/978-1-4614-0237-4`.

[3] J. BLOMVALL. "A multistage stochastic programming algorithm suitable for parallel computing." In: *Parallel Computing* 29.4 (2003), pp. 431–445. DOI: `10.1016/S0167-8191(03)00015-2`.

[4] J. Blomvall and P. O. Lindberg. "A Riccati-based primal interior point solver for multistage stochastic programming." In: *European Journal of Operational Research* 143.2 (2002), pp. 452–461. doi: 10.1016/S0377-2217(02)00301-6.

[5] K. Frauendorfer and H. Siede. "Portfolio Selection Using Multi-Stage Stochastic Programming." In: *Central European J. Oper. Res.* 7.4 (2000), pp. 277–290.

[6] J. Gondzio and A. Grothey. "Direct Solution of Linear Systems of Size 10^9 Arising in Optimization with Interior Point Methods." In: *Parallel Processing and Applied Mathematics*. Ed. by R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Waśniewski. Vol. 3911. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 513–525. doi: 10.1007/11752578_62.

[7] J. Gondzio and A. Grothey. "Exploiting structure in parallel implementation of interior point methods for optimization." In: *Computational Management Science* 6.2 (2009), pp. 135–160. doi: 10.1007/s10287-008-0090-3.

[8] J. Gondzio and A. Grothey. "Parallel interior-point solver for structured quadratic programs: Application to financial planning problems." In: *Annals of Operations Research* 152.1 (2007), pp. 319–339. doi: 10.1007/s10479-006-0139-z.

[9] J. Gondzio and A. Grothey. "Solving non-linear portfolio optimization problems with the primal-dual interior point method." In: *European Journal Operational Research* 181.3 (2007), pp. 1019–1029. doi: 10.1016/j.ejor.2006.03.006.

[10] A. Grothey. "Financial applications: Parallel portfolio optimization." In: *Parallel Computing*. Springer, 2009, pp. 435–469. doi: 10.1007/978-1-84882-409-6_14.

[11] J. Hübner. "Distributed Algorithms for Nonlinear Tree-Sparse Problems." PhD thesis. Gottfried Wilhelm Leibniz Universität Hannover, 2016.

[12] P. Kall and S. W. Wallace. *Stochastic Programming*. Wiley-Interscience Series in Systems and Optimization. New York: Wiley, 1994.

[13] C. Lawson, R. Hanson, D. Kincaid, and F. Krough. "Basic Linear Algebra Subprograms for FORTRAN usage." In: *ACM Trans. Math. Software* 5 (1979), pp. 308–325. doi: 10.1145/355841.355847.

[14] M. Lubin, C. G. Petra, and M. Anitescu. "The parallel solution of dense saddle-point linear systems arising in stochastic programming." In: *Optimization Methods and Software* 27.4-5 (2012), pp. 845–864. doi: 10.1080/10556788.2011.602976.

[15] S. Mehrotra. "On the Implementation of a Primal-Dual Interior Point Method." In: *SIAM J. Optim.* 2.4 (1992), pp. 575–601. doi: 10.1137/0802028.

[16] J. Nocedal and S. J. Wright. *Numerical Optimization*. 2nd. Berlin: Springer, 2006. doi: 10.1007/978-0-387-40065-5.

[17] T. Rauber and G. Rünger. *Parallel programming: For multicore and cluster systems*. Springer Science & Business, 2013. doi: 10.1007/978-3-642-37801-0.

[18] M. Schmidt. "A Generic Interior-Point Framework for Nonsmooth and Complementarity Constrained Nonlinear Optimization." PhD thesis. Gottfried Wilhelm Leibniz Universität Hannover, 2013.

[19] M. C. Steinbach. "Markowitz Revisited: Mean-Variance Models in Financial Portfolio Analysis." In: *SIAM Rev.* 43.1 (2001), pp. 31–85. doi: 10.1137/S0036144500376650.

[20] M. C. Steinbach. "Tree-Sparse Convex Programs." In: *Math. Methods Oper. Res.* 56.3 (2002), pp. 347–376. doi: 10.1007/s001860200227.

[21] *Boost C++ Libraries*. http://www.boost.org/. 1998–2016.

[22] *Open MPI: Open Source High Performance Computing*. http://www.open-mpi.org/. 2004–2016.

[23] R. J. Vanderbei and D. F. Shanno. "An Interior-Point Algorithm For Nonconvex Nonlinear Programming." In: *Comput. Optim. Appl.* 13 (1997), pp. 231–252. doi: 10.1023/A:1008677427361.

[24] A. Wächter and L. T. Biegler. "On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming." In: *Math. Program.* 106.1 (2006), pp. 25–57. doi: 10.1007/s10107-004-0559-y.

[1]Leibniz Universität Hannover, Institute for Applied Mathematics, Welfengarten 1, 30167 Hannover, Germany; [2](a) Department Mathematik, Friedrich-Alexander-Universität Erlangen-Nürnberg, Cauerstr. 11, 91058 Erlangen, Germany; (b) Energie Campus Nürnberg, Fürther Str. 250, 90429 Nürnberg, Germany

E-mail address: [1]{huebner,mcs}@ifam.uni-hannover.de, [2]mar.schmidt@fau.de