Exploiting Structure in Parallel Implementation of Interior Point Methods for Optimization

Jacek Gondzio Andreas Grothey

Technical Report, MS-04-004

December 18, 2004

Abstract

OOPS is an object oriented parallel solver using the primal dual interior point methods. Its main component is an object-oriented linear algebra library designed to exploit nested block structure that is often present is truly large-scale optimization problems. This is achieved by treating the building blocks of the structured matrices as objects, that can use their inherent linear algebra implementations to efficiently exploit their structure both in a serial and parallel environment. Virtually any nested block-structure can be exploited by representing the matrices defining the problem as a tree build from these objects.

We give details of supported structures and their implementations. Further we give details of how parallelisation is managed in the object-oriented framework.

1 Introduction

The aim of this paper is to give a detailed description of the object-oriented linear algebra module used inside our interior point code OOPS: Object-Oriented Parallel Solver. OOPS has been the subject of several reports [17, 14]. However, while these papers mention the underlying object-oriented design, their main concern is with practical applications without giving much detail about the actual implementation. The purpose of this paper is to fi ll this gap.

The prime motivation behind the development of OOPS is our interest in truly large scale optimization: problems with upwards of one million variables and constraints. In out observation these large scale optimization problems are not merely sparse, but also (block-)structured. Structure is not merely a byproduct of sparsity, but an essential feature of such problems: truly large scale problems are by necessity generated by some repeated process, be it discretizations (in time or space for control problems, of uncertainty for stochastic programming problems [14, 16]), or other kinds, such as the numerous repetitions of matrix blocks in reliability optimization for network problems [17, 15]. It is a fair assumption that the knowledge of the process that generated the problem structure can be passed on to the solver, to be used to its advantage. Furthermore structure is usually nested: Matrices are made up of sub-matrices, which themselves can be further divided.

The linear algebra operations to exploit all of these block-structures are well known and could be exploited at every level in the problem. However this is hardly ever done to its full capacity - except in special situations, like stochastic programming - due to the prohibitive coding effort that would be needed.

School of Mathematics, University of Edinburgh, Scotland, J.Gondzio@ed.ac.uk, A.Grothey@ed.ac.uk

OOPS provides an implementation of sparse, structured linear algebra operations that can exploit such nested structure in an efficient way. Since linear algebra operations that exploit block-structure lend themselves to parallelisation, emphasis has been placed on designing the package in such a form that all operations will be efficiently performed in parallel, should more than one processor be available for its computation.

The design of OOPS follows object-oriented principles, treating the blocks (and sub-blocks) of matrices as objects. We introduce a `Matrix` interface that defines all linear algebra methods needed for an interior point method. Several specialised classes provide concrete implementation of the `Matrix` interface, each exploiting a different possible structure. The matrix blocks are represented by objects of these classes, therefore every block of the matrix carries its own implementation of linear algebra routines, specialised for the structure present in this block.

A different interpretation of the object-oriented approach can be gained by introducing the concept of *elimination trees*: Elimination tree is a well known concept in the context of parallelising linear algebra operations for symmetric matrices [10, 12]. It carries information about dependencies between rows for elimination operations of a matrix and hence guides the distribution of parts of a matrix among processors. The elimination tree depends on the row and column ordering of the matrix. A balanced elimination tree makes for a more efficient exploitation of parallelism, however finding a corresponding row and column re-ordering is a non-trivial task.

Elimination trees can be generalised to block-elimination trees, where each node in the tree corresponds to a block of the matrix rows rather than a single row. While finding an efficient elimination tree for blocks is just as difficult, knowledge of the process that generated the block-structure can be easily exploited to this purpose. In fact every generating process will imply a characteristic block-elimination tree. As outlined before nodes in the block-elimination tree are treated as `Matrix`-objects, each of which carries information about how to best exploit the particular structure (elimination order) at this node.

The linear algebra kernel is used inside a primal dual interior-point solver targeted at convex optimization problems. Interior point methods (IPMs) are well suited to large scale optimization since they feature a consistently small number of iterations needed to reach the optimal solution of the problem as well as requiring fairly simple linear algebra. Indeed, modern IPMs rarely need more than 20-30 iterations to solve a quadratic program, and this number does not increase significantly even for problems with millions of variables. The linear algebra requirements boil down to factorizations and solves with the augmented system matrix of the problem. These can however be costly operations performed on huge matrices, so a highly optimized linear algebra is paramount to the design of an efficient IPM solver.

As far as we are aware our approach to an object-oriented linear algebra library is unique. There are various object-oriented implementations of IPMs and more general optimization algorithms reported in the literature: OOQP[13], TAO[4], OPT++[22] to name but a few (also see [13] for a summary of various ongoing efforts). However all of these use object-oriented concepts on the level of the interior point method: They aim to separate the logic of the interior point method from the used data-types and linear algebra implementation. The linear algebra used in these codes is still a traditional problem dependent implementation.

Throughout this paper we will use `Java` vocabulary to explain object-oriented terminology such as classes, interfaces and methods. We also use syntax such as `object.method` to refer to a method associated with a certain object.

The paper is organized as follows. In the following Section 2 we briefly review the linear algebra needed in Interior Point Methods and show that these are essentially the same for LP, QP and NLP problems. Section 3 clarifies the concept of nested block-structured matrices by extending the concept of elimination trees for factorizations to block-operations. Section 4 is concerned with the implementational details of the object-oriented design of the linear algebra routines, while section 5 gives details of the implementations of supported matrix structures. Section 6 is concerned with parallelisation aspects of OOPS. Finally, Section 7 repeats some computational results to illustrate the efficiency of OOPS.

2 Linear Algebra in Interior Point Methods

Interior point methods provide a unified framework for optimization algorithms for linear, quadratic and nonlinear programming. The reader interested in interior point methods may consult [28] for an excellent explanation of their theoretical background and [2] for a discussion of implementation issues. We show in this section that all these algorithms require similar linear algebra operations. Consequently, subject to minor modifications, the same linear algebra kernel may be used to implement interior point methods for all three classes of optimization problems.

2.1 Linear Programming

Consider the linear programming problem

$$\begin{array}{rr} \min & c^T x \\ \text{s.t.} & Ax = b, \\ & x \geq 0, \end{array}$$

where $A \in \mathcal{R}^{m\times n}$ is the full rank matrix of linear constraints and vectors x, c and b have appropriate dimensions. The usual transformation in interior point methods consists in replacing inequality constraints with the logarithmic barriers to get

$$\begin{array}{rl} \min & c^T x - \mu \sum_{j=1}^{n} \ln x_j \\ \text{s.t.} & Ax \ = b, \end{array}$$

where $\mu \geq 0$ is a barrier parameter. The Lagrangian associated with this problem has the form:

$$L(x,y,\mu) = c^T x - y^T (Ax - b) - \mu \sum_{j=1}^{n} \ln x_j$$

and the conditions for a stationary point write

$$\begin{array}{rcrcl} \nabla_x L(x,y,\mu) & = & c - A^T y - \mu X^{-1} e & = & 0 \\ \nabla_y L(x,y,\mu) & = & Ax - b & = & 0, \end{array}$$

where $X^{-1} = \text{diag}\{x_1^{-1}, x_2^{-1}, \ldots, x_n^{-1}\}$. Having denoted

$$s = \mu X^{-1} e, \qquad \text{i.e.} \qquad XSe = \mu e,$$

where $S = \text{diag}\{s_1, s_2, \dots, s_n\}$ and $e = (1, 1, \dots, 1)^T$, the first order optimality conditions (for the barrier problem) are:

$$\begin{array}{rcl} Ax & = & b, \\ A^T y + s & = & c, \\ XSe & = & \mu e \\ (x,s) & \geq & 0. \end{array} \tag{1}$$

Interior point algorithm for linear programming [28] applies Newton method to solve this system of nonlinear equations and gradually reduces the barrier parameter μ to guarantee the convergence to the optimal solution of the original problem. The Newton direction is obtained by solving the system of linear equations:

$$\begin{bmatrix} A & 0 & 0 \\ 0 & A^T & I \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \end{bmatrix} = \begin{bmatrix} \xi_p \\ \xi_d \\ \xi_\mu \end{bmatrix}, \tag{2}$$

where

$$\xi_p = b - Ax, \qquad \xi_d = c - A^T y - s, \qquad \xi_\mu = \mu e - XSe.$$

By elimination of

$$\Delta s = X^{-1}(\xi_\mu - S\Delta x) = -X^{-1}S\Delta x + X^{-1}\xi_\mu,$$

from the second equation we get the symmetric indefinite augmented system of linear equations

$$\begin{bmatrix} -\Theta_P^{-1} & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} \xi_d - X^{-1}\xi_\mu \\ \xi_p \end{bmatrix}. \tag{3}$$

where $\Theta_P = XS^{-1}$ is a diagonal scaling matrix. By eliminating Δx from the first equation we can reduce (3) further to the form of normal equations

$$(A\Theta_P A^T)\Delta y \; = \; \mathbf{b}_{LP}.$$

2.2 Quadratic Programming

Consider the convex quadratic programming problem

$$\begin{array}{ll} \min & c^T x + \frac{1}{2} x^T Q x \\ \text{s.t.} & Ax = b, \\ & x \geq 0, \end{array}$$

where $Q \in \mathcal{R}^{n \times n}$ is positive semidefinite matrix, $A \in \mathcal{R}^{m \times n}$ is the full rank matrix of linear constraints and vectors x, c and b have appropriate dimensions. The inequality constraints are again replaced with the logarithmic barriers

$$\begin{array}{ll} \min & c^T x + \dfrac{1}{2} x^T Q x - \mu \sum\limits_{j=1}^{n} \ln x_j \\ \text{s.t.} & Ax = b, \end{array}$$

where $\mu \geq 0$ is a barrier parameter and the associated Lagrangian has the form:

$$L(x,y,\mu) = c^T x + \frac{1}{2} x^T Qx - y^T (Ax-b) - \mu \sum_{j=1}^{n} \ln x_j.$$

The conditions for a stationary point write

$$\begin{array}{rcrcl} \nabla_x L(x,y,\mu) & = & c - A^T y - \mu X^{-1} e + Qx & = & 0 \\ \nabla_y L(x,y,\mu) & = & Ax - b & = & 0. \end{array}$$

With the usual notation for diagonal matrices: $X^{-1} = \mathrm{diag}\{x_1^{-1}, x_2^{-1}, \ldots, x_n^{-1}\}$ and $S = \mathrm{diag}\{s_1, s_2, \ldots, s_n\}$, the first order optimality conditions (for the barrier problem) are:

$$\begin{array}{rcl} Ax & = & b, \\ A^T y + s - Qx & = & c, \\ XSe & = & \mu e \\ (x,s) & \geq & 0. \end{array} \tag{4}$$

Interior point algorithm for quadratic programming [28] applies Newton method to solve this system of nonlinear equations and gradually reduces the barrier parameter μ to guarantee the convergence to the optimal solution of the original problem. The Newton direction is obtained by solving the system of linear equations:

$$\begin{bmatrix} A & 0 & 0 \\ -Q & A^T & I \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \end{bmatrix} = \begin{bmatrix} \xi_p \\ \xi_d \\ \xi_\mu \end{bmatrix}, \tag{5}$$

where

$$\xi_p = b - Ax, \qquad \xi_d = c - A^T y - s + Qx, \qquad \xi_\mu = \mu e - XSe.$$

By elimination of

$$\Delta s = X^{-1}(\xi_\mu - S\Delta x) = -X^{-1} S \Delta x + X^{-1} \xi_\mu,$$

from the second equation we get the symmetric indefinite augmented system of linear equations

$$\begin{bmatrix} -Q - \Theta_P^{-1} & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} \xi_d - X^{-1} \xi_\mu \\ \xi_p \end{bmatrix}. \tag{6}$$

where $\Theta_P = XS^{-1}$ is a diagonal scaling matrix. By eliminating Δx from the first equation we could reduce (6) further to the form of normal equations

$$(A(Q + \Theta_P^{-1})^{-1} A^T) \Delta y \ = \ \mathbf{b}_{QP}.$$

2.3 Nonlinear Programming

Consider the convex nonlinear optimization problem

$$\begin{array}{lll} \min & f(x) & \\ \text{s.t.} & g(x) & \leq 0, \end{array}$$

where $x \in \mathcal{R}^n$, and $f : \mathcal{R}^n \mapsto \mathcal{R}$ and $g : \mathcal{R}^n \mapsto \mathcal{R}^m$ are convex, twice differentiable. Having replaced inequality constraints with an equality $g(x)+z=0$, where $z \in \mathcal{R}^m$ is a nonnegative slack variable, we can formulate the associated barrier problem

$$\begin{array}{ll} \min & f(x) - \mu \sum_{i=1}^{m} \ln z_i \\ \text{s.t.} & g(x)+z=0. \end{array}$$

and write the Lagrangian for it

$$L(x,y,z,\mu) = f(x) + y^T(g(x)+z) - \mu \sum_{i=1}^{m} \ln z_i.$$

The conditions for a stationary point write

$$\begin{array}{rclcl} \nabla_x L(x,y,z,\mu) & = & \nabla f(x) + \nabla g(x)^T y & = & 0 \\ \nabla_y L(x,y,z,\mu) & = & g(x)+z & = & 0 \\ \nabla_z L(x,y,z,\mu) & = & y - \mu Z^{-1}e & = & 0, \end{array}$$

where $Z^{-1} = \text{diag}\{z_1^{-1}, z_2^{-1}, \cdots, z_m^{-1}\}$. The first order optimality conditions (for the barrier problem) have thus the following form

$$\begin{array}{rcl} \nabla f(x) + \nabla g(x)^T y & = & 0, \\ g(x)+z & = & 0, \\ YZe & = & \mu e \\ (y,z) & \geq & 0, \end{array} \tag{7}$$

where $Y = \text{diag}\{y_1, y_2, \cdots, y_m\}$. Interior point algorithm for nonlinear programming [28] applies Newton method to solve this system of equations and gradually reduces the barrier parameter μ to guarantee the convergence to the optimal solution of the original problem. The Newton direction is obtained by solving the system of linear equations:

$$\begin{bmatrix} Q(x,y) & A(x)^T & 0 \\ A(x) & 0 & I \\ 0 & Z & Y \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} -\nabla f(x) - A(x)^T y \\ -g(x) - z \\ \mu e - YZe, \end{bmatrix}, \tag{8}$$

where

$$\begin{array}{rclcl} A(x) & = & \nabla g(x) & \in & \mathcal{R}^{m \times n} \\ Q(x,y) & = & \nabla^2 f(x) + \sum_{i=1}^{m} y_i \nabla^2 g_i(x) & \in & \mathcal{R}^{n \times n}. \end{array}$$

Using the third equation we eliminate

$$\Delta z = \mu Y^{-1}e - Ze - ZY^{-1}\Delta y,$$

from the second equation and get

$$\begin{bmatrix} -Q(x,y) & A(x)^T \\ A(x) & \Theta_D \end{bmatrix} \begin{bmatrix} \Delta x \\ -\Delta y \end{bmatrix} = \begin{bmatrix} \nabla f(x) + A(x)^T y \\ -g(x) - \mu Y^{-1}e \end{bmatrix}, \tag{9}$$

where $\Theta_D = ZY^{-1}$ is a diagonal scaling matrix. The matrix involved in this set of linear equations is symmetric and indefinite. For convex optimization problem (when f and g are convex), the matrix Q is positive semidefinite and if f is strictly convex, Q is positive definite. Similarly to the case of quadratic programming by eliminating Δx from the first equation we could reduce this system further to the form of normal equations

$$\left(A(x)Q(x,y)^{-1}A(x)^T + ZY^{-1}\right)\Delta y = \mathbf{b}_{NLP}.$$

2.4 Indefinite Systems in Interior Point Methods

The three systems (3), (6) and (9) have many similarities. In (3) and (6) only the diagonal scaling matrix Θ_P changes from iteration to iteration; in the case of nonlinear programming the matrix $\Theta_D = ZY^{-1}$ and the matrices $Q(x,y)$ and $A(x)$ in (9) change in every iteration. Without the loss of generality to simplify notation in the following sections we will assume that A and Q are constant matrices as if we were concerned with the quadratic optimization problems.

Every iteration of the interior point method for linear, quadratic or nonlinear programming requires the solution of a possibly *large* and almost always *sparse* linear system

$$\begin{bmatrix} -Q-\Theta_P^{-1} & A^T \\ A & \Theta_D \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}. \tag{10}$$

In this system, $\Theta_P \in \mathcal{R}^{n\times n}$ and $\Theta_D \in \mathcal{R}^{m\times m}$ are diagonal scaling matrices with strictly positive elements. Depending on the problem type one or both matrices Θ_P and Θ_D may be present in this system. For linear and quadratic programs with equality constraints $\Theta_D = 0$. For nonlinear programs with inequality constraints (and variables without sign restriction) $\Theta_P^{-1} = 0$. For ease of the presentation we assume that we deal with convex programs hence the Hessian $Q \in \mathcal{R}^{n\times n}$ is a symmetric positive defi nite matrix. $A \in \mathcal{R}^{m\times n}$ is the matrix of linear constraints (or the linearization of nonlinear constraints); we assume it has a full rank.

The matrix in (10) is indefi nite and can be transformed to a quasidefi nite one. Quasidefi nite matrix [27] has the form $\begin{bmatrix} -E & A^T \\ A & F \end{bmatrix}$, where E and F are symmetric positive defi nite matrices and A has full rank. As shown in [27], quasidefi nite matrices are strongly factorizable, i.e., a Cholesky-like factorization LDL^T with a diagonal D exists for any symmetric row and column permutation of the quasidefi nite matrix. The diagonal matrix D has n negative and m positive pivots. This feature implies a major advantage: similarly to a case of factorization of positive defi nite matrix, it is possible to sperate the sparsity analysis phase from the numerical factorization phase. (The reader should notice that when a general indefi nite matrix is decomposed into the form LDL^T where L is a unit lower triangular matrix then the easily invertible matrix D has to allow 2×2 pivots [3, 10], which requires merging analyse and factorize phases.)

The elements of diagonal matrices Θ_P and Θ_D in the system (10) display a drastic difference of magnitude: some of them tend to zero while others go to infi nity. In consequence Θ_P and Θ_D as well as the whole system (10) are very ill-conditioned. Careless pivoting on *any* diagonal element could lead to a disastrous loss of accuracy. To prevent this from hapenning we have implemented the regularization approach of [1]. Namely, whenever necessary we strengthen the quasidefi nite property of the matrix involved in (10) by adding terms to unacceptably small pivot candidates in blocks E and F. Consequently, we deal with the matrix

$$H_R = \begin{bmatrix} -Q-\Theta_P^{-1} & A^T \\ A & \Theta_D \end{bmatrix} + \begin{bmatrix} -R_P & 0 \\ 0 & R_D \end{bmatrix},$$

where diagonal positive defi nite matrices $R_P \in \mathcal{R}^{n\times n}$ and $R_D \in \mathcal{R}^{m\times m}$ can be interpreted as adding proximal terms (*regularizations*) to the primal and dual objective functions, respectively. In the method of [1] the entries of the regularizing matrices are chosen dynamically: the negligibly small terms are used for all acceptable pivots and the stronger regularization terms are used whenever a dangerously small pivot

$$\begin{bmatrix} x & & & & x & & x & \\ & x & & x & & x & & \\ & & x & & & & x & x \\ & x & & x & & x & & \\ x & & & & x & x & x & \\ & x & & x & x & x & & \\ x & & x & & x & & x & \\ & & x & & & & & x \end{bmatrix} \quad \begin{bmatrix} x & & & & & & & \\ & x & & & & & & \\ & & x & & & & & \\ & x & & x & & & & \\ x & & & & x & & & \\ & x & & & x & x & & \\ x & & x & & x & f & x & \\ & & x & x & & & f & x \end{bmatrix}$$

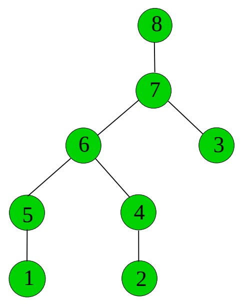

Figure 1: Matrix Φ, its Cholesky factor L and the associated elimination tree $\mathcal{T}$.

$$\begin{bmatrix} x & & x & & & & & x \\ & x & x & & & & & x \\ x & x & x & & & & & x \\ & & & x & & & x & x \\ & & & & x & & x & x \\ & & & & & x & x & x \\ & & & x & x & x & x & x \\ x & x & x & x & x & x & x & x \end{bmatrix} \quad \begin{bmatrix} x & & & & & & & \\ & x & & & & & & \\ x & x & x & & & & & \\ & & & x & & & & \\ & & & & x & & & \\ & & & & & x & & \\ & & & x & x & x & x & \\ x & x & x & x & x & x & x & x \end{bmatrix}$$

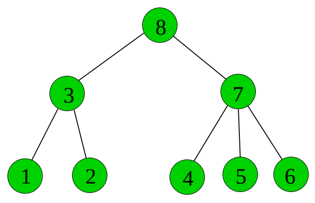

Figure 2: Matrix Φ, its Cholesky factor L and the associated elimination tree $\mathcal{T}$.

candidate appears. The use of dynamic regularization introduces little perturbation to the original system because the regularization concentrates uniquely on potentially unstable pivots. The use of primal and dual regularizations makes the factorization of quasidefi nite matrix numerically stable and therefore viable for application in the context of interior point methods.

3 Exploiting Nested Block-Structure

3.1 Elimination Tree

Consider a sparse triangular matrix $L \in \mathcal{R}^{\ell\times\ell}$. Following [10, 12] we associate with this matrix an elimination tree $\mathcal{T}$, a graph with ℓ nodes $\{1,2,\dots,\ell\}$ and $\ell-1$ arcs connecting a given node j with its ancestor node:

$$a = \min\{i > j \mid l_{ij} \neq 0\}.$$

If L is irreducible then $\mathcal{T}$ is indeed a tree; for a reducible matrix (decomposable to block-diagonal form) $\mathcal{T}$ is a forrest of trees associated with each irreducible diagonal block. An example in Fig 1 displays the sparsity patterns of a symmetric 8×8 matrix Φ, its Cholesky factor L and the associated elimination tree $\mathcal{T}$. The nonzero elements in the matrix are denoted with x and the fi ll-in elements in the Cholesky factor with f.

Tree defi nes a precedence of elimination operations: if a is an ancestor of j then column j has to be processed before column a. By analysing the elimination tree one may deduce the best way to exploit parallelism in the computation of Cholesky factor. For the matrix presented in Fig 1 the decomposition can be performed independently for three buckets of columns: $\{3\}$, $\{1,5\}$ and $\{2,4\}$ corresponding to independent branches of the tree. Then the last two contribute to the column 6 and this column together

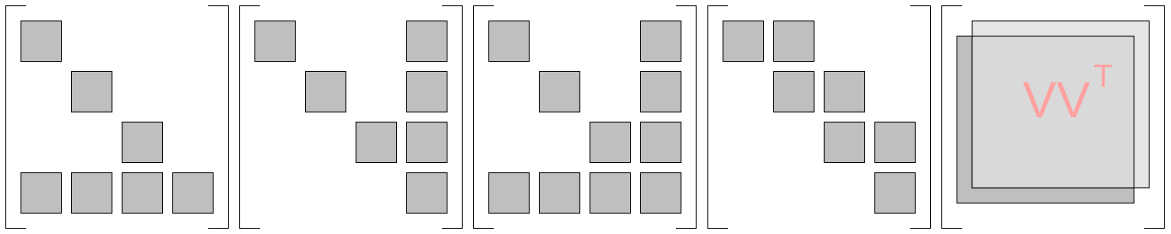

Figure 3: Different exploitable structures: primal- and dual block-angular, bordered block-diagonal, block-banded and rank-corrector.

with the first bucket contribute to column 7, and eventually to column 8.

The elimination tree changes when the matrix is re-ordered using symmetric row and column permutations. Obviously a balanced elimination tree where all branches have a similar length is better suited to parallelism, than one where most nodes are in one long branch. However finding a re-ordering of the matrix that leads to a balanced elimination tree is a non-trivial task.

In many situations however information about how to create a balanced elimination tree is readily available. As a motivating example we display the nested bordered diagonal matrix in Figure 2 with its corresponding elimination tree. No fill-in is created by factorizing this matrix and furthermore its eliminations tree is balanced. Nodes $\{1,2,3\}$ can be eliminated indpendent of $\{4,5,6,7\}$ and then each of the leaf nodes $\{1,2,4,5,6\}$ is independent of the others. While recognising such a structure in an anonymous sparse matrix might require a considerable effort, many real life problems possess a block structure of this pattern which is known at modelling time and could hence be passed to the solver to exploit. OOPS is an interior point solver aimed at exploiting known block elimination trees.

3.2 Nested Block-Structured Matrices

By a block-structured matrix we understand a matrix that is composed of sub-matrices. This could be a matrix whose sub-blocks form a particular sparse pattern, such as a bordered block-diagonal or block-banded matrix (see Figure 3). Alternatively this could be a structured sum of two matrices, such as the rank-corrector matrix

$$\tilde{A} = A + VV^T$$

where $V \in \mathbb{R}^{n\times k}$ has a small number of columns, so that VV^T is a low-rank correction to A.

By a nested block-structured matrix we understand a matrix where each sub-matrix is a block-structured matrix itself. The particular structure of the sub-matrix might well be different from the structure of the parent matrix. There is no limit on the depth to which this nesting can be extended.

Nested block-structured matrices occur frequently in applications. Multistage stochastic programming, where every modelled stage corresponds to one level of nesting in the resulting system matrix is just one example. Other examples are various network problems (joint optimal synthesis of base and spare network capacity, multicommodity network flow problems, etc) solved in telecommunications applications [17, 15]. Some formulations of Support Vector Machines [8, 11] have system matrices of rank-corrector structure, as have some convex reformulations of Markowitz-type financial planning problems [14, 16]. Rank-corrector structure also occurs when the Hessian matrix of a nonlinear programming problem is not known explicitly but estimated by a quasi-Newton scheme.

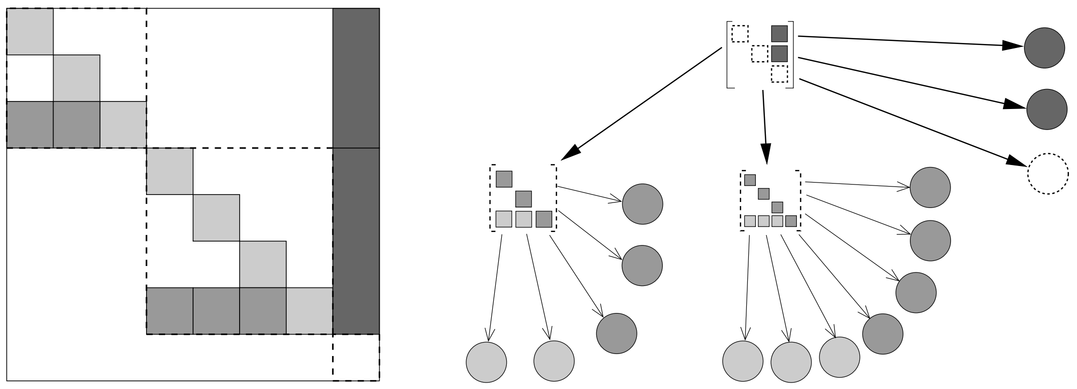

Figure 4: Nested Block-Structured Constraint Matrix with its Tree Representation.

We assume the nested structure is known to the solver.

The nested block-structure of a matrix can be thought of as a tree. Its root is the whole matrix and every block of a particular sub-matrix is a child node of the node representing this sub-matrix. Leaf nodes correspond to the elementary sub-matrices that can no longer be divided into blocks. With every node of the tree we associate information about the type of structure this node represents. Figure 4 shows an example of a nested block-structured matrix together with the tree that represents it. The partitioning of the constraint matrix *A* into blocks induces the partitioning of associated vectors into subvectors. The tree representation of the matrix therefore implies a tree representation of vectors in the primal and dual spaces. We will discuss in detail the relations between `Matrix` its `Tree` and the associated `VectorTree` in Section 4.4.

3.3 Node-oriented linear algebra

Effi cient linear algebra routines to exploit a certain known block-structure of a problem are well known and a multitude of different implementations exist [6, 7, 18, 19, 20, 21, 25, 26]. The reader interested in other paralell developments for optimization should consult [9, 24, 23] and the references therein. Every different structure however needs its own linear algebra implementation. In principle nested structures could be exploited in the same way, however the coding effort involved is tremendously magnifi ed, as is the multitude of different combined structures that would need to be covered. Indeed we do not know of any such effort.

The design of OOPS is based on the fact that any method supported by our linear algebra library can be performed by working through the tree: At every node evaluating the required linear algebra operation for the matrix corresponding to this node can be broken down into a sequence of operations performed on its sub-blocks (i.e. child nodes in the tree). The exact sequence of these operations does of course depend on the type of structure present at this node. The crucial observation is that at this particular node the type of its child-node is of no importance, as long as they can perform the operations they are asked to do. *How* the operations are performed on the children nodes is of no concern to the parent.

This is the basis of the object-oriented design of OOPS: We introduce a `Matrix` interface, a collection of linear algebra routines (methods) that need to be implemented for all supported structures. Every node of the tree is then represented by an object of `Matrix`-type. The actual implementation for each method

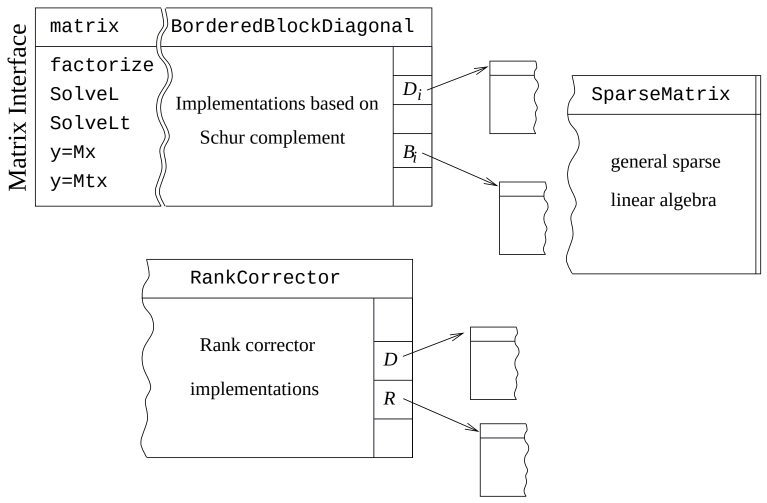

Figure 5: The matrix interface and several implementations of it: Building blocks for the tree-structure.

will differ from node to node depending on the type of structure present. When an implementation of a particular method needs access to its subnodes, it does so by calling its subnodes `Matrix` methods, which will then invoke an effi cient way of performing the required operation on the child.

Clearly only one implementation of each method is needed for each type of structure that we want to exploit: For every such structure we have one implementation of the `Matrix` interface. A nested block-structured matrix is represented in OOPS as its tree (as in Figure 4), where each node is an object of one of the classes that implement the `Matrix` interface (see Figure 5).

3.4 Structured Augmented System Matrices

Since our library is designed for use in IPMs for quadratic or nonlinear programming our main interest is in exploiting structure in the augmented system matrix $\Phi = \begin{bmatrix} -Q-\Theta_P^{-1} & A^T \\ A & \Theta_D \end{bmatrix}$. The question of whether an exploitable nested block-structure of the matrices A and Q can be combined into an exploitable structure of Φ seems non-trivial at fi rst. However this can always be done in a generic way, providing that a few simple restrictions are satisfi ed

- The primal `VectorTree`s of both A and Q have to be identical.
- For every block in A the number of child nodes in its primal and dual `VectorTree`s differ by at most one.

Although these restrictions seem fairly strong, they in fact only restrict the way in which the sparsity pattern of A and Q should be represented in a nested block-structured way; they do not restrict the sparsity patterns of A and Q themselves.

To see this, note that the fi rst restriction can always be satisfi ed by using the same primal `VectorTree` for Q that was used for A: i.e. the division of the rows and columns of Q into blocks and sub-blocks is given by the division used for the columns of A. It is conceivable that this process might lead to an undesirable

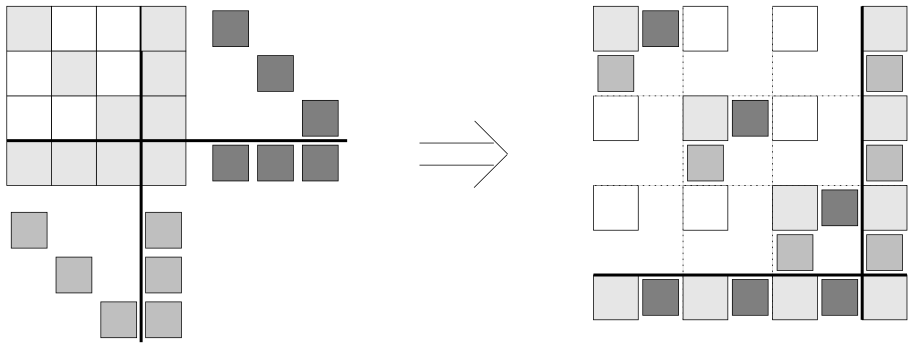
Figure 6: Dual Block-Angular *A* and implied structure of *Q* and Φ.

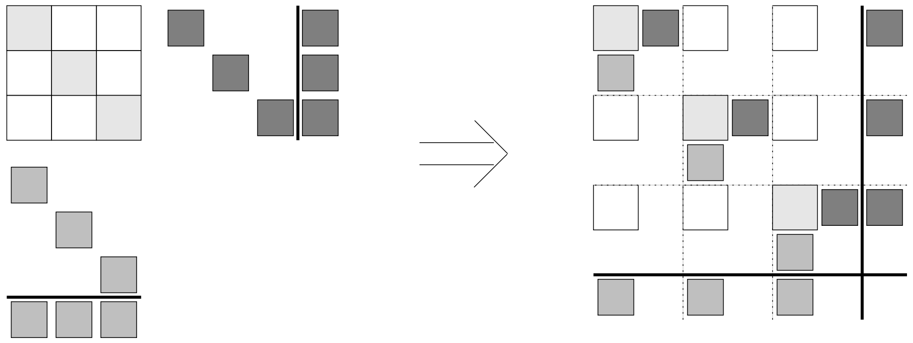
Figure 7: Primal Block-Angular *A* and implied structure of *Q* and Φ.

block-structure in *Q*, at worst every sub-block of *Q* might contain non-zero elements. However it is often possible to move non-zeros of *Q* into a more convenient block by changes of the model. Note that these are changes that improve the sparsity pattern of the augmented system matrix: they would be benefi cial for any algorithm employed to solve the system. It is therefore not a peculiar requirement of our design approach.

The second requirement puts a constraint on the block structures that can be used to construct *A*: while any block of *A* can be rectangular with very different row and column dimensions, it is suffi cient to allow only block structures that have row and column block count different by at most one to represent this block.

Figures 6-8 give examples of how a certain block structure of *A* would impose a structure on *Q* and Φ. In these examples the shaded part of the *Q* matrix indicates blocks in which nonzeros would not harm the structure of Φ that is imposed by *A*. Should nonzeros occur in other blocks of *Q* then either the problem would have to be remodelled, or *Q* could be represented as a superimposition of several structures (i.e. if *Q* had entries in the border blocks in Figure 8, *Q* could be represented as a bordered block-diagonal matrix with one diagonal block, which would then be of banded structure).

This re-ordering procedure is generic: it does not depend on the *types* of the matrices in question. It simply combines a block of the *Q* matrix with the blocks at the corresponding position in the *A* and A^T part; any blocks left over (by assumption above there will be at most one row or column) form an additional border that consists of imcomplete augmented system matrices. While the re-ordering is generic, the *type* of `Matrix` that is used to represent the structured augmented system depends on the

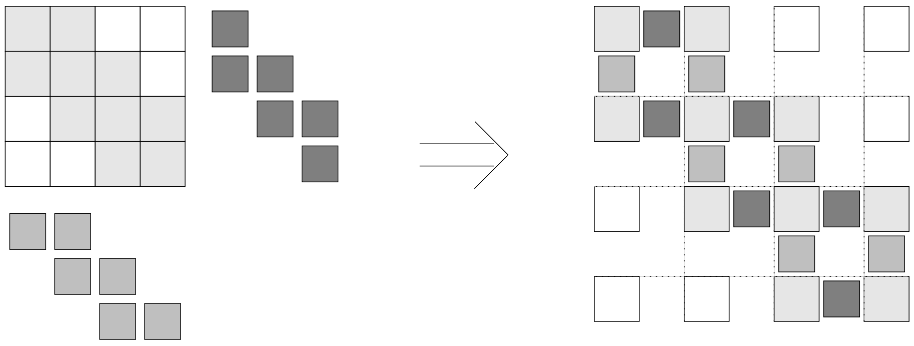

Figure 8: Banded A and implied structure of Q and Φ.

types of A and Q. The re-ordering is therefore performed by a `makeAugSystem` method of the class used to represent Q. It will do the following operations:

- From the input A, Q, B (=A^T if diagonal block of Q):
- Assert that the conditions for compatability of the matrices are satisfied.
- Determine the best combined `Matrix`-type given the types of A, Q and B.
- Create this block of the augmented System matrix, by combining one block each of the constituent matrices.

Two remarks are in order: firstly it becomes apperent, why the augmented system constructor have to assume that either of A, B or Q may not be present. Secondly the combining of the blocks is done by recursively calling the `makeAugSystem` method for the sub-blocks of Q. This way sub-augmented system blocks (like the ones in Figures 6-8) that consists themselves of structured matrices will be further re-ordered, until the whole augmented system matrix is a nested block-structured matrix. In [14] we give an example of how a block-structured augmented system with three levels of nesting is re-ordered by this process.

It is worth noting that this procedure requires no further memory to store the reordered augmented system matrix Φ. Its leaf node matrices are identical to those already present in A and Q. No physical re-ordering of memory entries is done, the procedure merely creates a new tree of matrix blocks re-using the already existing leaf-nodes.

4 Implementation

The primal-dual interior point method needs to access the system matrices A, Q and the augmented system matrix Φ. In our implementation access to these matrices is provided through two `interfaces`: `SimpleMatrix` representing a simple matrix such as A or Q and `AugSysMatrix` representing an augmented system matrix Φ. The difference between these two classes is that `SimpleMatrix` in essence only provides Matrix-Vector operations, whereas `AugSysMatrix` provides factorization and backsolve routines in addition to the Matrix-Vector operations. `AugSysMatrix` is assumed to have `SimpleMatrix` components A B and Q and a `StructuredVector` component $\Theta = (\Theta_P, \Theta_D)$ in the form

$$\begin{bmatrix} -Q-\Theta_P^{-1} & B^T \\ A & \Theta_D \end{bmatrix}.$$

An `AugSysMatrix` can either be a *diagonal block* (in which case it is symmetric and $B = A$) or *non-diagonal* in which case Θ is not present. Both the `SimpleMatrix` and `AugSysMatrix` interfaces are subinterfaces of `Matrix`.

4.1 Flow of Control

The user of our library is expected to call the constructor routines for different implementations of `SimpleMatrix` to build the matrices A and Q from their constituting blocks. After that `Q.makeAugSystem(A,B,Theta)` is called to create the augmented system matrix. `makeAugSystem` will determine from the types of its two input `SimpleMatrix` the appropriate type of the `AugSysMatrix` and construct a corresponding object by calling its constructor recursively with the appropriate children of A and Q. Note that this process merely sets up pointer structures: The actual `SparseMatrix` leaf nodes that make up Φ are identical to those that make up A and Q; these leaf nodes are re-used when building Φ this is merly done in a different order.

It would be possible and worthwhile to automate this process by the use of a modelling language. The modelling language would need to support the creation of leaf node matrices (probably from a common *core* matrix), and provide support for various structure generating processes, such as stochasticity and discretizations over time and space. Further it would need to support nonlinearities in the model. We are not aware of any modelling language that satisfi es these conditions. SMPS (the stochastic programming extension of MPS) [5] goes some way towards it, and there exists an SMPS interface to our solver.

4.2 The `SimpleMatrix` interface

The `SimpleMatrix` interface provides routines to construct the structured problem matrices A and Q and to do simple matrix-vector-type operations on them. The interface defi nes the following methods

- `SimpleMatrix Contructor(...)`
- `StructuredVector matrixTimesVector (StructuredVector)`
- `StructuredVector matrixTransTimesVector(StructuredVector)`
- `StructuredVector getColumn(int)`
- `StructuredVector getRow (int)`
- `StructuredSparseVector getSparseColumn(int)`
- `StructuredSparseVector getSparseRow (int)`
- `void setStructure(void)`
- `Tree getPrimalTree(void)`

- `Tree getDualTree(void)`
- `AugSysMatrix makeAugSystem(SimpleMatrix A, SimpleMatrix B)`

It thus includes the capability of performing matrix-vector products, retrieving a dense or sparse row or column from the matrix and to set up further structures like the primal/dual `VectorTree` (`setStructure`) and the augmented system matrix. In OOPS the following classes implement the `SimpleMatrix` interface:

`SimpleSparseMatrix`	general sparse matrix
`SimpleDenseMatrix`	general dense matrix
`SimpleNetworkMatrix`	arc-node incidence matrix for networks
`SimpleBlockDiagonalMatrix`	block-diagonal
`SimpleBorderedBlockDiagonalMatrix`	block-diagonal with dense rows and columns
`SimplePrimalBlockAngularMatrix`	block-diagonal with dense rows
`SimpleDualBlockAngularMatrix`	block-diagonal with dense columns
`SimpleRankCorrectorMatrix`	$A+VV^T$, where V has small number of columns

4.3 The `AugSysMatrix` interface

The `AugSysMatrix` interface is intended to represent an augmented system matrix of the form $\Phi = \begin{bmatrix} -Q-\Theta_P^{-1} & B^T \\ A & \Theta_D \end{bmatrix}$. It will consist of references to its constituting parts A, Q, Θ and B (identical to A if symmetric). The interface will support the same methods as `SimpleMatrix` but in addition also factorization and backsolve routines (the latter in sparse and dense modes):

- `void symbolicFactorization(void)`
- `void computeCholesky(void)`
- `StructuredVector solveCholesky(StructuredVector)`
- `StructuredVector solveL(StructuredVector)`
- `StructuredVector solveLt(StructuredVector)`
- `StructuredVector solveD(StructuredVector)`
- `StructuredVector solveSparseCholesky(StructuredSparseVector)`
- `StructuredSparseVector solveSparseL(StructuredSparseVector)`
- `StructuredSparseVector solveSparseD(StructuredSparseVector)`
- `StructuredSparseVector solveSparseLt(StructuredSparseVector)`
- `void setNewTheta(StructuredVector)`

Generally the implementations of this interface will break down the computations of matrix-vector type methods into computations on its sub-parts, calling the appropriate method of the `SparseMatrix` representing A, B, Q. `symbolicFactorization` will determine a row/column re-ordering with near-minimal fi ll-in and create data-structures that can store fi rst the re-ordered augmented-system matrix and later its factorization, allowing for the earlier determined fi ll-in. `computeCholesky` will do the numerical phase of the factorization: building the (re-ordered) augmented system matrix and fi nding a representation of its Cholesky factors. Not all implementing classes will use an implicit factorization that can be represented in the LDL^T format. Therefore some classes might not implement the `solveL/D/Lt`-methods. Accordingly some of the implementations of the methods might offer different alternatives depending on whether its children support the `solveL/D/Lt`-methods.

The `AugSysMatrix` interface is implemented in OOPS by

`SparseAugSysMatrix`	sparse leaf node augmented system matrix
`DenseAugSysMatrix`	dense leaf node augmented system matrix
`BlockDiagonalAugSysMatrix`	block-diagonal
`BorderedBlockDiagonalAugSysMatrix`	block-diagonal with dense rows and columns
`RankCorrectorAugSysMatrix`	Q of the form $\tilde{Q} + VV^T$

For both the `SimpleMatrix` and `AugSysMatrix` interface, the implementing classes can be classifi ed as either *leaf node classes* such as *dense*, *sparse* or *network* or the *complex* classes, such as *BorderBlock-Diagonal* or *RankCorrector*. The later are constituted from submatrices which themselves are of type `SimpleMatrix` or `AugSysMatrix`. The crucial idea of our library is that an effi cient implementation of all methods for a *complex* class can be reduced to a sequence of methods performed on its constituents. The top-level class here does not need to know the exact type of its constituent objects nor whether they themselves are of *leaf-node-type* or *complex*, it merely needs to know that they support the methods of the interface and assumes that they do so in a way most effi cient for their particular structure. This enables us to essentially re-create the structured matrix tree (Figure 4) with a tree of `Matrix` objects (see Figure 5).

4.4 The `VectorTree` and `StructuredVector` Classes

Most of the `Matrix` operations need to be performed on (or with) vectors. In this section when talking about *vectors* we generally mean the primal/dual iterates (x^k, y^k, s^k) of the interior point method. These will be dense vectors, hence we present this section as applicable to dense vectors. For certain subtask of the Factorization or backsolve routines, sparse vectors are preferrable: hence we have also a mirror implementation of a `SparseStructuredVector` class.

Since the implementations of the `Matrix`-methods generally break operations down to a sequence of operations on sub-blocks of matrices, we need to be able to break vectors down into sub-vectors in a compatible fashion. This is further complicated by the fact, that the implememntation should also work in parallel, where each processor only knows (and has memory allocated for) a part of the vector. For this reason only having a sequence of known components of the vector in memory is not possible, some further information is needed.

Nevertheless we need to make sure that two operations that respectively operate on a part of the vector and the whole vector, actually work with the same physical elements.

The information of what is a *compatible* vector to a particular block-structured matrix is carried in the `VectorTree` class. The `VectorTree` class is constructed from the corresponding matrix. Note that rectangular matrices will usually have different primal and dual `VectorTrees`. Figure 9 gives an example of the primal and dual tree corresponding to a block-structured matrix. Every node of the `VectorTree` is

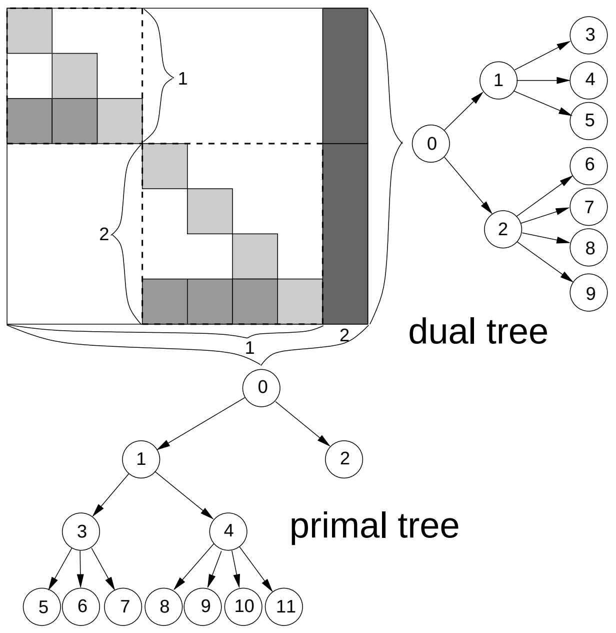

Figure 9: Primal and dual vector tree derived from structured matrix.

represented by the `VectorTree` class which carries the information on the structure of this node, as well as how this node fi ts into the complete vector. It has instance variables corresponding to

- number of children, array of children (array of `VectorTree`s),
- start and end of this node in absolute indices,
- index number (of this node in the tree),
- flag indicating whether dense data for this node is stored on a given processor.

The `VectorTree` structure is generated by the matrix. For this purpose, the `Matrix` interface defi nes a method `setStructure` that is required to recursively generate the primal and dual `VectorTree` associated with this `Matrix`. It has also `getPrimalTree`, `getDualTree` methods that will return the corresponding `VectorTree`s to the user of the Matrix.

The `StructuredVector` class represents a vector corresponding to a given `VectorTree`. That is it supports the necessary operation to access the sub-vector corresponding to every node of the `VectorTree`. Note that this is true even if the actual values of the vector are distributed among several processors. The representation of a vector as a `StructuredVector` consists essentially of two layers. The bottom layer is simply an array of `doubles` storing all the vector elements that are known on this processor. The second layer has the necessary information to access these elements by nodes in the `VectorTree`. An example of the primal `VectorTree` associated with the structured matrix in Figure 9 is displayed in Figure 10. This second layer is an array of `StructuredVector` objects (one corresponding to each node of the tree). Note that the sub-vector corresponding to a particular node of the `VectorTree` is a `StructuredVector` as well, so it is sensible to represent it by the same structure that represents the complete vector. Each `StructuredVector` object in the second layer has the following instance variables

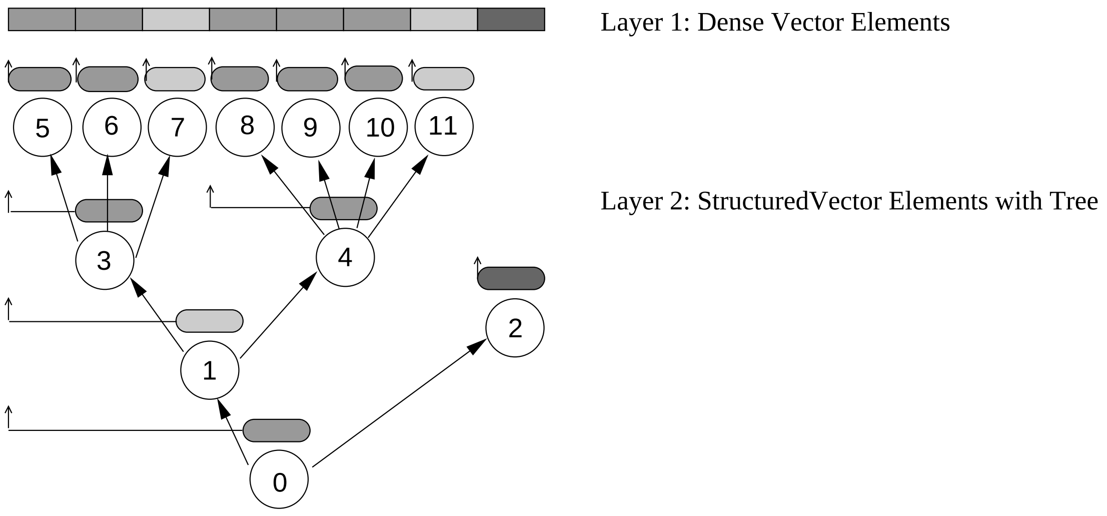
Layer 1: Dense Vector Elements
Layer 2: StructuredVector Elements with Tree
5
6
7
8
9
10
11
3
4
2
1
0

Figure 10: The two layers of the vector representation.

- node in `VectorTree` corresponding to this `StructuredVector`,
- pointer to dense element information (if on this processor),
- pointer to the complete array of `StructuredVectors`.

Note that all other information (is data on this processor, length of data corresponding to this subvector, children if any, and indices of these children in the `StructuredVectors` array) can be obtained from the corresponding node in the `VectorTree`.

Since the interior point solver OOPS is working with the augmented system we need to be able to access a primal and dual vector together as one vector structure. In this case the subvectors of the augmented system vector should not be the primal and dual vector, but again augmented system vectors corresponding to submatrices of the augmented system. This layout can be achieved by combining the equivalent nodes of the primal and dual `VectorTrees` into augmented system nodes and building a separate augmented system `VectorTree` from these (see Figure 11). Note that in our implementation we go the opposite route: The `VectorTree` corresponding to the augmented system is created first - by calling the appropriate method of the `Matrix` interface. During this process nodes are labelled depending on whether they belong to the primal or dual part of the vector. Based on this information separate `VectorTrees` can be created later to access only the primal or dual nodes of the augmented system vector.

4.5 Implementation Details

The linear algebra code is written in C. We decided to use C rather than C++ or Java due to the more direct control offered by C. While language support for object-oriented design as offered in C++ or Java would undoubtedly make the coding easier, these language features incur some overhead that might decrease the overall efficiency of the solver. It is well possible to use C for object-oriented program design at the cost of a slight increase in coding effort when compared to an object-oriented language. The bottom

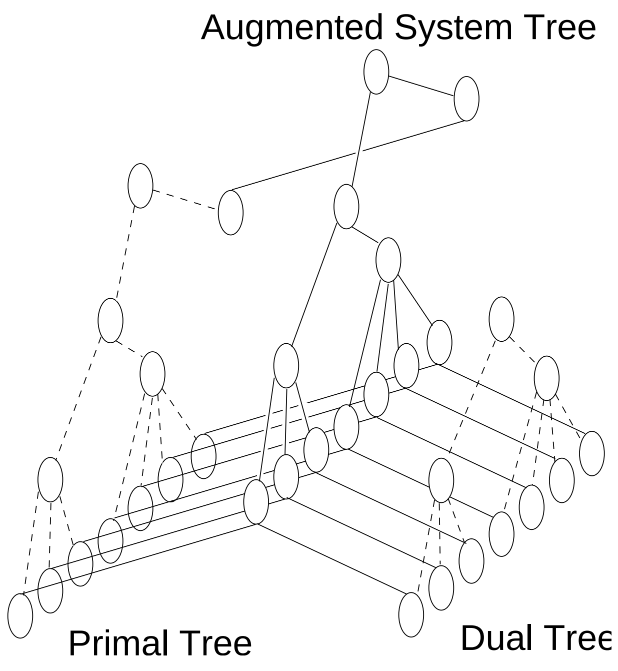

Figure 11: Building Augmented System Tree from Primal and Dual Tree.

level routines that implement the elementary sparse matrix factorization and backsolves are written in Fortran, again for effi ciency reasons.

The parallel implementation of OOPS uses MPI. This choice offers a great flexibility concerning the choice of platform and the code should run just as well on a network of PCs as on a dedicated parallel machine.

5 Implementations of the Matrix Interface

In this section we will give details of the classes that implement the `AugSysMatrix` interface.

5.1 The `BorderedBlockDiagonalAugSysMatrix` class

This class represents a augmented system matrix with symmetric bordered block-diagonal structure:

$$\Phi = \begin{pmatrix} \Phi_1 & & & & B_1^T \\ & \Phi_2 & & & B_2^T \\ & & \ddots & & \vdots \\ & & & \Phi_n & B_n^T \\ B_1 & B_2 & \cdots & B_n & \Phi_0 \end{pmatrix}, \qquad (11)$$

where $\Phi_i \in \mathcal{R}^{n_i \times n_i}, i = 0,\dots,n$ and $B_i \in \mathcal{R}^{n_0 \times n_i}, i = 1,\dots,n$. Note that since this is a *complex* class it does not use references to its constituent A, Q and Θ blocks. It therefore can represent any matrix of the above form. Matrix Φ has $N = \sum_{i=0}^{n} n_i$ rows and columns. Such blocks will be created if its components A and Q blocks are of block-diagonal and/or block-angular structure.

We can obtain a block-Cholesky type decomposition of the matrix

$$\Phi = LDL^T$$

by employing the Schur-complement mechanism as

$$L = \begin{pmatrix} L_1 & & & & \\ & L_2 & & & \\ & & \ddots & & \\ & & & L_n & \\ L_{n,1} & L_{n,2} & \cdots & L_{n,n} & L_c \end{pmatrix}, \quad D = \begin{pmatrix} D_1 & & & & \\ & D_2 & & & \\ & & \ddots & & \\ & & & D_n & \\ & & & & D_c \end{pmatrix} \tag{12a}$$

where

$$\begin{aligned} \Phi_i &= L_i D_i L_i^T & \text{(12b)} \\ L_{n,i} &= B_i L_i^{-T} D_i^{-1} & \text{(12c)} \\ C &= \Phi_0 - \sum_{i=1}^{n} B_i \Phi_i^{-1} B_i^T & \text{(12d)} \\ &= L_c D_c L_c^T & \text{(12e)} \end{aligned}$$

Formula (12b) needs some additional comments. As will become clear further down, L_i and D_i are only ever accessed in the form $L_i^{-1}b, D_i^{-1}b$, that is through Φ_i's `solveL/D/Lt` methods. The only constraint placed on the form of L_i, D_i is that the sequence of calls `solveLt, solveD, solveL` is equivalent to a call to `solveCholesky` (i.e. formula (12b) holds). Should the class representing Φ_i use an implicit factorization that does not support a `solveL` method, we can simply set $D_i = \Phi_i$ and $L_i = I$. With these settings the rest of the analysis below stays correct. For the implementation, a class (such as `MatrixAugSysRankCorrector`) that does not support `solveL` can set `solveD` as a synonym for `solveCholesky` and `solveL/Lt` as do-nothing (i.e. return the input vector). If `solveL` is supported the backsolve routine below is slightly more effi cient (requiring 3 calls to Φ_i`.solveL/Lt` rather than the equivalent of 4 (2 times `solveCholesky`) otherwise.

Representation (12) can be used to compute the solution to the system

$$\Phi x = b,$$

where $x = (x_1, \ldots, x_n, x_0)^T$, $b = (b_1, \ldots, b_n, b_0)^T$ as follows

$$\begin{aligned} z_i &= L_i^{-1} b_i, \quad i = 1, \ldots, n & \text{(13a)} \\ z_0 &= L_c^{-1}(b_0 - \sum_{i=1}^{n} L_{n,i} z_i) & \text{(13b)} \\ y_i &= D_i^{-1} z_i, \quad i = 0, \ldots, n & \text{(13c)} \\ x_0 &= L_c^{-T} y_0 & \text{(13d)} \\ x_i &= L_i^{-T}(y_i - L_{n,i}^T x_0), \quad i = 1, \ldots, n & \text{(13e)} \end{aligned}$$

Note that the matrices $L_{n,i}$ are only used in (13b, 13e) for two matrix-vector multiplications each. On the other hand the computation of $L_{n,i}$ by (12c) would require n_i solves with matrix L_i^T. In certain situations it is more effi cient not to compute $L_{n,i}$ explicitly, but evaluate (13b, 13e) as

$$z_0 = L_c^{-1}(b_0 - \sum_{i=1}^{n} B_i L_i^{-T} D_i^{-1} z_i) \tag{13b'}$$

$$x_i = L_i^{-T}(y_i - D_i^{-1} L_i^{-1} B_i^T x_0), \quad i = 1, \dots, n \tag{13e'}$$

replacing the matrix-vector product with a backsolve involving L_i.

Also the sum to compute C in (12d) is often best calculated from terms $(L_i^{-1}B_i^T)^T D_i^{-1}(L_i^{-1}B_i^T)$, which in turn are best calculated as sparse outer products of the sparse rows of $L_i^{-1}B_i^T$.

Because of this L_i, D_i, L_c, D_c can be seen as an implicit Cholesky factorization of Φ.

All these computations can be done naturally in our object-oriented environment: (12b) requires a call to `computeCholesky` for each of the diagonal parts Φ_i of Φ. The sum in (12d) is formed by `B[i].getSparseRow(...)` followed by `Phi[i].solveSparseL/D(...)` and an outer product of `SparseStructuredVector` objects to create C as a `SimpleDenseMatrix`. The backsolves can be similarly broken down into `AugSysMatrix` methods performed on Φ_i, B_i and C.

5.2 The `RankCorrectorAugSysMatrix` class

This class represents a matrix Φ that is a combination of an (easily invertible) part $\tilde{\Phi} \in \mathbb{R}^{n \times n}$ plus a low rank update VV^T, where $V \in \mathbb{R}^{n \times k}$ and k is small. Its implementation is based on the Sherman-Morrison-Woodbury formula

$$\Phi^{-1} = \tilde{\Phi}^{-1} - \tilde{\Phi}^{-1} V (I + V^T \tilde{\Phi}^{-1} V)^{-1} V^T \tilde{\Phi}^{-1}$$

which implies that the system $\Phi x = b$ can be alternatively solved by

$$W = \tilde{\Phi}^{-1} V \tag{14a}$$

$$y = \tilde{\Phi}^{-1} b \tag{14b}$$

$$C = I + V^T W \tag{14c}$$

$$x = y - W C^{-1} V^T y \tag{14d}$$

W and C^{-1} can be seen as an implicit representation of the inverse of Φ. The factorization and backsolve routine therefore consist of the following steps:

- `computeCholesky`:

```
C = DenseMatrix.identity(k,k)
A.computeCholesky
for i=1,$k$
   u = V.getSparseColumn(i)
   W[i] = A.solveCholesky(u)
   for j=1,$k$
      v = V.getSparseColumn(j)
      C[i][j] += v.dotProd(W[i])
   end
end
C.computeCholesky
```

- `solveCholesky(b)`:

  ```
  y = A.solveCholesky(b)
  tmp1 = V.matrixTransTimesVector(y)
  tmp2 = C.solveCholesky(tmp1)
  tmp3 = W.matrixTimesVector(tmp2)
  y.substract(tmp3)
  ```

As pointed out above, the implicit factorization in this class does not support the concept of separate `solveL/D/Lt` methods. As suggested `solveD` will therefore be equivalent to `solveCholesky` and `solveL/Lt` will be empty methods.

5.3 Sparse Elementary Matrices: The `SparseAugSysMatrix` class

In any sparse nested block-structured matrix the leaf nodes will be eventually represented by sparse matrices. It is therefore important to include an efficient implementation of a `SparseMatrix` class in our linear algebra library. The implementation of this class will follow very closely the way that (non-structured) sparse linear algebra for IPMs has been traditionally implemented. It will feature

- a re-ordering heuristic to find sparse Cholesky factors
- separation of symbolic and numerical factorization
- regularization to avoid two-by-two pivoting

Note that our invertible `SparseMatrix` objects are always of augmented system type: the factorization routines must therefore cope with a quasi-definite matrix.

However we do not attempt to exploit parallelism on this level. In our experience most models offer enough scope to exploit parallelism in a much more effective way, so that we see no need for a parallelisation of the `SparseMatrix` class. The re-ordering heuristic has therefore the sole aim of finding a sparse Cholesky factor, rather than to balance this aim with the requirement for a balanced elimination tree.

6 Parallelisation

Due to the block-structure of many of the classes implementing the `Matrix` interface, their methods lend themselves naturally to parallelisation. There are two main advantages in parallelisation. Firstly there is a speed gain by distributing computations among several processors. This is especially the case with block structured operations where the computations break down into subtasks that can be computed independently. The second advantage concerns memory requirement: If computations are shared between different processors, a significant amount of problem data is only required on a subset of processors. This will lead to less memory needed on each processor (thereby enabling the solution of problems that might otherwise not fit into the memory of a single machine). We would also expect that spreading the data between processors will lead to more efficient caching on every processor and hence a further speed gain.

To highlight the issues faced in the parallelisation consider the computations needed for the `computeCholesky` method from `BorderedBlockDiagonalAugSysMatrix` (Section 5.1).

Φ`.factorize`:

Φ_1`.factorize`	$V_1 = \Phi_1$`.solveL`(B_1^T)	$C_1 = V_1^T D_1^{-1} V_1$	C_1`.add`(Φ_0)	$C = \sum C_i$	C`.factorize`
$\vdots$	$\vdots$	$\vdots$	*idle*		
Φ_n`.factorize`	$V_n = \Phi_n$`.solveL`(B_n^T)	$C_n = V_n^T D_n^{-1} V_n$	*idle*		

$x = \Phi$`.solveL`(b):

$x_1 = \Phi_1$`.solveL`(b_1)	$v_1 = -\Phi_1$`.solveLt`$(D_1^{-1}x_1)$	$c_1 = B_1$`.times`(v_1)	c_1`.add`(b_0)	$c = \sum c_i$	$x_0 = C$`.solveL`(c)
$\vdots$	$\vdots$	$\vdots$	*idle*		
$x_n = \Phi_n$`.solveL`(b_n)	$v_n = -\Phi_n$`.solveLt`$(D_n^{-1}x_n)$	$c_n = B_n$`.times`(v_n)	*idle*		

The factorizations of the diagonal blocks Φ_i and the subsequent computations of matrices $C_i = B_i L_i^{-T} D_i^{-1} L_i^{-1} B_i^T$ are independent of each other, and will be distributed among processors. The computation of the Schur complement $C = \Phi_0 - \sum_i C_i$ requires communications between the processors and the results of the final factorization of C needs to be known on all processors. To save on communications the factorization of C is computed on all processors, implying that the forming of C from the C_i's and Φ_0 requires a global reduce operation.

The problem data is distributed among the processors on a 'need-to-know' basis. Once the computation tasks are assigned to processors, the appropriate distribution of problem data can be derived. In the above example obviously diagonal blocks Φ_i will be distributed among the processors. The same holds for the border blocks B_i. Φ_0 is strictly speaking only needed on one processor, however it shares the same spatial location as the Schur complement C which is needed everywhere, hence Φ_0 is also stored on all processors.

The same argument can be applied to derive the distribution of `VectorTree` nodes among the processors. Nodes x_i are distributed, whereas the node corresponding to the border blocks x_0 is stored on all processors.

The optimal distribution of child-nodes in both the matrix and vector trees among the processors assigned to the parent node is dependent on the matrix type and hence performed by a `Matrix` method `allocateProcessors`. `allocateProcessors(int first, int last)` allocates a set $\mathcal{P}(i)$ of processors to node i. It takes a range of processors and allocates them to its children in whatever way is sensible for the matrix-type that the implementing class represents. Processors are allocated to children by calling the child's `allocteProcessors` method. Obviously a parent can only allocate those processors to its children which it has been allocated itself. Where the parent can allocate more processors than it has children (nodes high up in the tree), fairly sophisticated strategies can be used that determine which child can benefit most from additional processors. Allocation of nodes to processors in a nested-structure can therefore also be performed by working recursively on the tree.

Figure 12 illustrates the allocation of problem data to processors for a nested bordered block-diagonal matrix. It should be read by comparing it with the matrix and vector tree representations from Figures 4

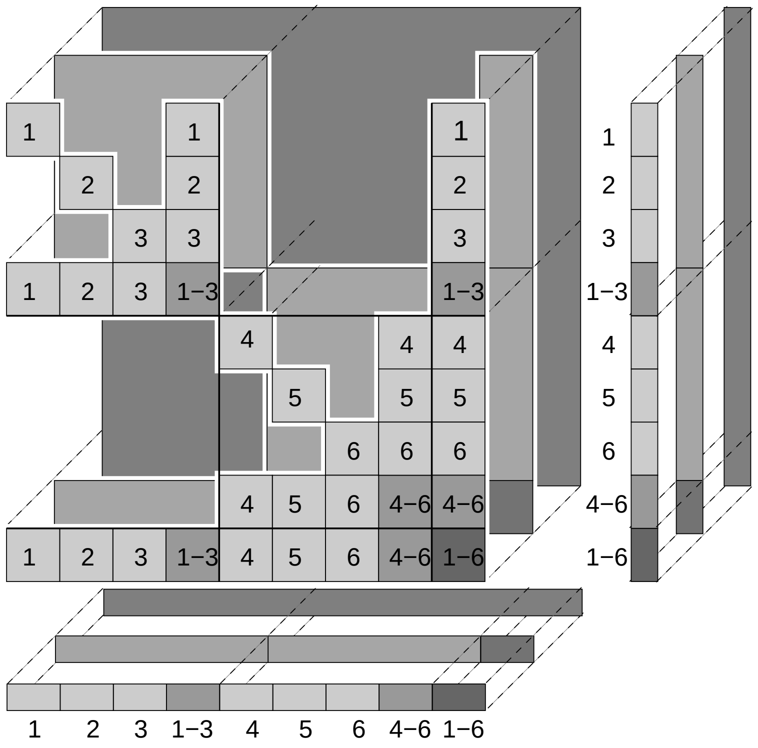

Figure 12: Allocating matrix and vector blocks to processors.

and 9. Each level of Figure 12 corresponds to one level of nodes in the trees. The bottom-most layer corresponds to the whole matrix (vector), the root node of each of the trees which is allocated to all processors. The topmost layer corresponds to the elementary matrices and vector parts. At this point we should explain what is meant by allocating a set of processors $\mathcal{P}(i)$ to a node *i*: For elementary leaf nodes the data describing the node has to be stored on every processor in $\mathcal{P}(i)$. For complex node, all `Matrix` operations on node *i* must be possible to perform by combining information held on *all* nodes in $\mathcal{P}(i)$. Hence data describing the actual elements of matrices corresponding to these nodes are *distributed* among the processors (they belong to the leaf nodes). The `Matrix` object containing the implementations of the linear algebra methods and pointers to the child nodes is present on *all* processors in $\mathcal{P}(i)$. On all processors not in $\mathcal{P}(i)$, node *i* in the matrix tree is represented by a `FakeMatrix` object. `FakeMatrix` is an implementation of the `SimpleMatrix` and `AugSysMatrix` interfaces, that defines all methods to be empty. It has no data associated with it and no children. It is a *dummy node* in the tree that causes all tree operations to stop at this point.

Using this setup most of the parallelisation of the linear algebra methods is done automatically. A computation such as (12) and (13) is coded on every processor as written (indeed as it would be in a serial implementation). Due to the `FakeMatrix` every processor only does those computations for which it has the required data. In effect a sum such as (13b) or (12d) is distributed among all processors that can perform a part of it. All that is needed differently from the serial implementation, is to sum up processor contributions at the end. When working on complex matrix trees, this layout ensures that complete branches that are allocated to a different processor are skipped, since already the top-node of the branch is a `FakeMatrix`.

Occasionally, in summations such as $C = \Phi_0 + \sum_{i=1}^{n} C_i$ we need to add an explicit test to make sure that the matrix Φ_0 is only added on one processor.

There are two types of communications needed between the processors:

- Gathering partial computational results for the same node from different processors: such as the Schur-complement sum above, or the sum needed for a parallel dot-product.
- Gathering contributions for different nodes on one processor: only at the end of the optimization to gather the final result.

The first type of communication is realised by using the list of processors associated with each node: all necessary reducing operations are performed on this set. For the second type of communication each processor must know where its bits belong in the global grid. This information is already held in the primal and dual `VectorTree` structures: `MPI` provides convenient routines that automatically retrieve the correct bits of a dense vector from the correct processors.

6.1 Loading the matrix: Parallel program flow

The nodes of the trees representing problem matrices A and Q are distributed among the processors. Clearly it is desirable that the node-specific problem data is only held on processors that will be asked to work with this data. On the other hand allocation of nodes to processors is done by a `Matrix`-method: that is the tree of `Matrix` objects needs to be build before the allocation of processors to nodes can take place. To overcome this conflict we use the following bootstrapping method:

- The model builder creates the Matrix-tree on *all* processor. Data for elementary sparse matrices is not generated yet.
- By calling `allocateProcessors` recursively nodes in the tree are allocated to processors. Nodes that should be worked on by other processors are replaced by `FakeMatrix`.
- `setStructure` will create the primal and dual `VectorTree`. They inherit their processor allocation from the matrix-Tree.
- Another recursive call `fillLeafNodes` will generate the data describing the leaf nodes on those processors that need the information.

7 Numerical Results

To emphasise the usefulness of our object-oriented algebra approach for designing an efficient interior point solver we wish to repeat some computational results for OOPS. These results have been obtained for various quadratic and nonlinear formulations of *Asset and Liability Management Problems* (ALM). Details of these results and the corresponding models can be found in [14] and [16]. Tables 1 and 2 give details of the parallel performance of OOPS on different problems. `ALM5-QP, UNS1-QP, ALM6-QP, ALM7-QP` are quadratic models, `ALM3-NLP, ALM4-NLP` involve a nonlinear constraint. These results were obtained on a SunFire 15000 with 24 processors at 900MHz and 2Gb of memory per processor. The smaller problems are solved on 1-8 processors, whereas the two larger problems could only be solved on

Problem	Total Nodes	Constraints	Variables
ALM5-QP	266305	3,461,966	9,586,981
UNS1-QP	360152	2,160,919	5,402,296
ALM6-QP	1742521	10,455,127	26,137,816
ALM7-QP	1742521	19,167,732	52,275,631
ALM3-NLP	65641	3,411,693	9,974,152
ALM4-NLP	169456	3,724,953	10,500,112

Table 1: Asset and Liability Management Problems: Problem Statistics.

Problem	1 proc		2 procs		4 procs		8 procs		16 procs	
	t	it	t	pe	t	pe	t	pe	t	it
ALM5-QP	8958	18	4566	98.0	2286	98.0	1195	93.7		
UNS1-QP	6119	27	3260	93.8	1633	93.7	872	87.7		
ALM6-QP									1470	18
ALM7-QP									8465	19
ALM3-NLP	17764	41	9028	98.4	4545	97.7	2390	92.9		
ALM4-NLP	18799	43	9391	100.0	4778	98.4	2459	95.6		

Table 2: Parallel Solution Results (t=CPU time, it=iterations, pe=parallel efficiency).

16 processors (due to memory requirements). Problems with over 50 million variables could be solved and the results demonstrate that the code parallelises very well.

To counter the possible criticism that these models are just very easy, we have compared OOPS with the industry standard QP solver CPLEX 7.0. The results are summarized in Table 3. Since we do not possess a CPLEX license for the SunFire, results are obtained on a 400MHz Ultrasparc-II processor with 2GB of memory. This explains the smaller problem sizes: for bigger problems CPLEX consistently ran out of memory (`OoM` in the table). CPLEX results for the `cALM8d` problems are extrapolated from the flop count reported by CPLEX. As can be seen OOPS outperforms CPLEX on both solution time and memory requirements, with the difference being more pronounced the larger the problem sizes are.

References

[1] A. ALTMAN AND J. GONDZIO, *Regularized symmetric indefinite systems in interior point methods*

Problem	nodes	c/s	vars	CPLEX 7.0			OOPS		
				time (s)	iter	Mb	time (s)	iter	Mb
cALM8b	1123	57,274	168,451	2838	31	637	368	16	142
cALM8c	2552	130,153	382,801	10910	29	1590	860	16	319
cALM8d	4971	253,522	745,651	(51000)	(30)	OoM	1723	17	678
QP-ALM6	3661	80,483	226,862	955	16	340	159	15	136
QP-ALM7	6481	142,503	401,662	2616	16	736	283	13	238
QP-ALM1	4971	208,713	606,322	12236	18	1731	531	17	365

Table 3: Comparison with CPLEX on QP problems.

for linear and quadratic optimization, Optimization Methods and Software, 11-12 (1999), pp. 275–302.

[2] E. D. Andersen, J. Gondzio, C. Mészáros, and X. Xu, *Implementation of interior point methods for large scale linear programming*, in Interior Point Methods in Mathematical Programming, T. Terlaky, ed., Kluwer Academic Publishers, 1996, pp. 189–252.

[3] M. Arioli, I. S. Duff, and P. P. M. de Rijk, *On the augmented system approach to sparse least-squares problems*, Numerische Mathematik, 55 (1989), pp. 667–684.

[4] S. Benson, L. C. McInnes, and J. J. Moré, *TAO users manual*, Tech. Rep. ANL/MCS-TM-249, Argonne National Laboratory, 2001.

[5] J. Birge, M. Dempster, H. Gassmann, E. Gunn, A. King, and S. Wallace, *A standard input format for multiperiod stochastic linear programs*, Committee on Algorithms Newsletter, 17 (1987), pp. 1–19.

[6] J. R. Birge and L. Qi, *Computing block-angular Karmarkar projections with applications to stochastic programming*, Management Science, 34 (1988), pp. 1472–1479.

[7] I. C. Choi and D. Goldfarb, *Exploiting special structure in a primal-dual path following algorithm*, Mathematical Programming, 58 (1993), pp. 33–52.

[8] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel Based Learning Methods*, Cambridge University Press, 2000.

[9] V. De Leone, A. Murli, P. Pardalos, and G. Toraldo (eds), *High Performance Algorithms and Software in Nonlinear Optimization*, Kluwer Academic Publisher, New York, 1998.

[10] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*, Oxford University Press, New York, 1987.

[11] M. C. Ferris and T. S. Munson, *Interior point methods for massive support vector machines*, SIAM Journal on Optimization, 13 (2003), pp. 783–804.

[12] A. George and J. W. H. Liu, *The evolution of the minimum degree ordering algorithm*, SIAM Review, 31 (1989), pp. 1–19.

[13] E. M. Gertz and S. J. Wright, *Object-oriented software for quadratic programming*, ACM Transactions on Mathematical Software, 29 (2003), pp. 58–81.

[14] J. Gondzio and A. Grothey, *Parallel interior point solver for structured quadratic programs: Application to financial planning problems*, Technical Report MS-03-001, School of Mathematics, University of Edinburgh, Edinburgh EH9 3JZ, Scotland, UK, April 2003. Accepted for publication in: *Annals of Operations Research*.

[15] ——, *Reoptimization with the primal-dual interior point method*, SIAM Journal on Optimization, 13 (2003), pp. 842–864.

[16] ——, *Solving nonlinear portfolio optimization problems with the primal-dual interior point method*, Technical Report MS-04-001, School of Mathematics, University of Edinburgh, Edinburgh EH9 3JZ, Scotland, UK, April 2004.

[17] J. Gondzio and R. Sarkissian, *Parallel interior point solver for structured linear programs*, Mathematical Programming, 96 (2003), pp. 561–584.

[18] M. D. Grigoriadis and L. G. Khachiyan, *An interior point method for bordered block-diagonal linear programs*, SIAM Journal on Optimization, 6 (1996), pp. 913–932.

[19] J. K. Hurd and F. M. Murphy, *Exploiting special structure in a primal-dual interior point methods*, ORSA Journal on Computing, 4 (1992), pp. 39–44.

[20] E. R. Jessup, D. Yang, and S. A. Zenios, *Parallel factorization of structured matrices arising in stochastic programming*, SIAM Journal on Optimization, 4 (1994), pp. 833–846.

[21] I. J. LUSTIG AND G. LI, *An implementation of a parallel primal-dual interior point method for multicommodity flow problems*, Computational Optimization and Applications, 1 (1992), pp. 141–161.

[22] J. C. MEZA, *OPT++: An object-oriented class library for nonlinear optimization*, Tech. Rep. SAND94-8225, Sandia National Laboratories, 1994.

[23] A. MIGDALAS, G. TORALDO, AND V. KUMAR, *Nonlinear optimization and parallel computing*, Parallel Computing, 29 (2003), pp. 375–391.

[24] ——, *Parallel computing in numerical optimization*, Parallel Computing, 29 (2003), pp. 373–373.

[25] G. SCHULTZ AND R. R. MEYER, *An interior point method for block angular optimization*, SIAM Journal on Optimization, 1 (1991), pp. 583–602.

[26] M. J. TODD, *Exploiting special structure in Karmarkar's linear programming algorithm*, Mathematical Programming, 41 (1988), pp. 81–103.

[27] R. J. VANDERBEI, *Symmetric quasidefinite matrices*, SIAM Journal on Optimization, 5 (1995), pp. 100–113.

[28] S. J. WRIGHT, *Primal-Dual Interior-Point Methods*, SIAM, Philadelphia, 1997.