Comput Manag Sci (2009) 6:135–160
DOI 10.1007/s10287-008-0090-3

ORIGINAL PAPER

Exploiting structure in parallel implementation of interior point methods for optimization

Jacek Gondzio · Andreas Grothey

Published online: 16 December 2008

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 in truly large-scale optimization problems such as those appearing in Stochastic Programming. 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. OOPS can be run on a wide variety of architectures and has been used to solve a financial planning problem with over 10^9 decision variables. We give details of supported structures and their implementations. Further we give details of how parallelisation is managed in the object-oriented framework.

Keywords Interior point methods · Parallelism · Optimization · Structure exploitation · Object-oriented

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 (Gondzio and Grothey 2006a,b, 2007a,b). 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 fill this gap.

J. Gondzio · A. Grothey (✉)
School of Mathematics, University of Edinburgh, Edinburgh, UK
e-mail: A.Grothey@ed.ac.uk

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 our 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. Stochastic Programming is an obvious example where structure is introduced by the discretisation of the underlying probability space (Gondzio and Grothey 2007a,b). Other examples include discretisation in time or space for control problems or repetitions of matrix blocks in reliability optimization for network problems (Gondzio and Sarkissian 2003; Gondzio and Grothey 2003). As problem sizes grow, increasingly problems display a nested combination of these structures: such as network reliability problems with uncertain demands where a stochastic programming structure is superimposed on the structure of the reliability problem. 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.

OOPS provides a modular 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 operations 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.

The advantage of this object-oriented approach over traditional linear algebra implementations lies in its flexibility: It provides building blocks from which any (exploitable) combination of nested block structures present in the problem can be constructed. The layout of the package is such that this is only a concern at the modelling stage with minimum coding effort. The exploitation of the structure in the various linear algebra routines and their parallelisation will follow automatically. If additional "building bocks" representing new structures are needed they can be added easily, extending the capabilities of the solver.

A different interpretation of the object-oriented approach can be gained by introducing the concept of *elimination trees*: Elimination trees are a well known concept in the context of parallelising linear algebra operations for symmetric matrices (Duff et al. 1987; George and Liu 1989). They carry information about dependencies between rows for elimination operations of a matrix and hence guide the distribution of parts of a matrix among processors. Essentially it encodes the order of pivot operations for factoring the matrix. A balanced elimination tree makes for a more efficient exploitation of parallelism, however finding such a pivot order is non-trivial.

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. The elimination tree now encodes not only the "pivot" order but also what the applicable "pivoting" operation at each step is. For block sparse matrices these are block pivot operations, but structures such as low-rank updates require different operations. While finding an efficient elimination tree for blocks is just as difficult as for sparse elements, 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 small quadratic program, and this number does not increase significantly even for problems with many millions of variables. The linear algebra requirements boil down to factorisations and solves with the augmented system matrix of the problem. These can however be costly operations performed on huge matrices, so a highly optimised 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 (Gertz and Wright 2003), TAO (Benson et al. 2001), OPT++ (Meza et al. 2007) to name but a few (also see Gertz and Wright (2003) 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. On the other hand several developments deal specifically with exploiting stochastic programming structure in IPM (Blomvall and Lindberg 2002; Blomvall 2003; Steinbach 2000), not to mention various decomposition techniques which are also well suited to exploit parallelism (Linderoth and Wright 2003). The advantage of OOPS is added flexibility to exploit nested structures that do not fit into the usual stochastic programming frame such as stochastic network optimization.

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. Generally the `typewriter` font is used to refer to methods and structures actually present in the implementation.

The paper is organised as follows: In Sect. 2, we briefly review the linear algebra needed in interior point methods. Section 3 clarifies the concept of nested block-structured matrices and consequences to the design of OOPS. Section 4 is concerned with the details of the object-oriented implementation of the linear algebra routines, while Sect. 5 gives details of the implementations of supported matrix structures. Finally Sect. 6 is concerned with parallelisation aspects of OOPS and Sect. 7 summarises some key numerical results achieved by 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 Wright (1997) for an excellent explanation of their theoretical background and Andersen et al. (1996) 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 and convex quadratic programming

Consider the quadratic programming problem

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

where $Q \in \mathcal{R}^{n \times n}$ is a positive semi-definite matrix, $A \in \mathcal{R}^{m \times n}$ is a full rank matrix of linear constraints and vectors x, c and b have appropriate dimensions (for linear programming set $Q = 0$). The usual transformation in interior point methods consists in replacing inequality constraints with the logarithmic barriers to get

$$\min c^T x + \frac{1}{2} x^T Q x - \mu \sum_{j=1}^{n} \ln x_j \quad \text{s.t. } Ax = b,$$

where $\mu \geq 0$ is a barrier parameter.

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

where $X = \text{diag}\{x_1, \ldots, x_n\}$ ans S likewise. The interior point algorithm for quadratic programming Wright (1997) applies Newtons method to 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:

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

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

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

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

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

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

2.2 Nonlinear programming

Consider the convex nonlinear optimization problem

$$\min\ f(x) \quad \text{s.t. } g(x) \le 0,$$

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 non-negative slack variable we can formulate the associated barrier problem

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

Following the same derivations as for the convex quadratic case we arrive at the (reduced) Newton system

$$\begin{bmatrix} -Q(x,y) & A(x)^T \\ A(x) & \Theta_D \end{bmatrix} \begin{bmatrix} \Delta x \\ -\Delta y \end{bmatrix} = \begin{bmatrix} \nabla f(x) + A(x)^T y \\ -g(x) - \mu Y^{-1} e \end{bmatrix} \tag{4}$$
$$\Delta z = \mu Y^{-1}e - Ze - ZY^{-1}\Delta y,$$

where $\Theta_D = ZY^{-1}$ is a diagonal scaling matrix and $A(x) = \nabla g(x)$, $Q(x, y) = \nabla^2 f(x) + \sum_{i=1}^m y_i \nabla^2 g_i(x)$. 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 $Q(x)$ is positive semi-definite and if f is strictly convex, $Q(x)$ 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

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

2.3 Indefinite systems in interior point methods

The two systems (3) and (4) have many similarities. In (3) 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 (4) change in every iteration.

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

$$\begin{bmatrix} -Q - \Theta_P^{-1} & A^T \\ A & \Theta_D \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}. \qquad (5)$$

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 presentation we assume that we deal with convex programs hence the Hessian $Q \in \mathcal{R}^{n\times n}$ is a symmetric positive definite 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.

Note that the matrix in (5) changes numerically but not structurally at every iteration. It is therefore advantageous to separate the symbolic factorisation phase that determines a sparsity preserving pivot order from the numerical factorisation phase. The symbolic factorisation phase only needs to be done once at the beginning of the interior point algorithm. However the matrix in (5) is indefinite. The factorisation of a general indefinite matrix into LDL^T form requires the use of 2×2 block pivots which appear on the diagonal of D (Arioli et al. 1989; Duff et al. 1987). The pivot order and appearance of 2×2 pivots strongly depend on the numerical values of the pivots, preventing the separation of symbolic and numerical factorisation.

However the augmented system matrix can be transformed into a quasi-definite matrix. A quasi-definite matrix has the form $\begin{bmatrix} -E & A^T \\ A & F \end{bmatrix}$, where E and F are symmetric positive definite matrices and A has full rank. As shown in Vanderbei (1995), quasi-definite matrices are strongly factorisable, i.e., a Cholesky-like factorisation LDL^T with a diagonal D exists for any symmetric row and column permutation of the quasi-definite matrix.

We achieve this transformation by the use of a regularisation approach as in Altman and Gondzio (1999). Namely, whenever a close-to-zero pivot is encountered we add a small perturbation to the pivot. Consequently, we deal with the matrix

$$H_R = \begin{bmatrix} -Q - \Theta_P^{-1} & A^T \\ A & \Theta_D \end{bmatrix} + \begin{bmatrix} -R_P & 0 \\ 0 & R_D \end{bmatrix}, \tag{6}$$

which is quasi-definite. The diagonal positive definite 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 (regularisations) to the primal and dual objective functions, respectively. In the method of Altman and Gondzio (1999) the entries of the regularising matrices are chosen dynamically: negligibly small terms are used for all acceptable pivots and the stronger regularisation terms are used whenever a dangerously small pivot candidate appears. The use of dynamic regularisation introduces little perturbation to the original system because the regularisation concentrates uniquely on potentially unstable pivots. The use of primal and dual regularisations makes the factorisation of quasi-definite 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 Duff et al. (1987) and George and Liu (1989) we associate with this matrix an elimination tree $\mathcal{T}$, a graph with ℓ nodes $\{1, 2, \ldots, \ell\}$ 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 forest 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 fill-in elements in the Cholesky factor with f.

The tree defines 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

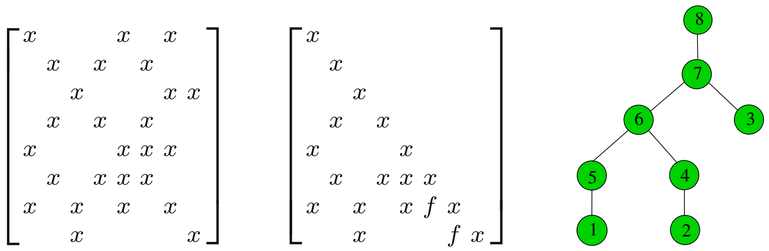


Fig. 1 Matrix Φ, its Cholesky factor L and the associated elimination tree $\mathcal{T}$

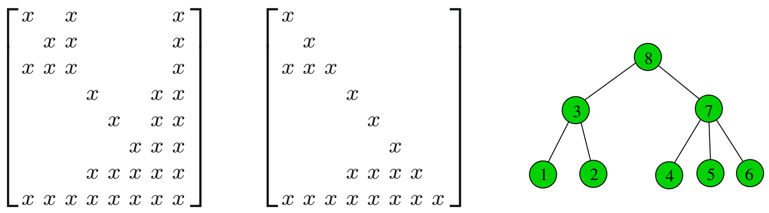


Fig. 2 Matrix Φ, its Cholesky factor L and the associated elimination tree $\mathcal{T}$

this column together 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 Fig. 2 with its corresponding elimination tree. No fill-in is created by factoring this matrix and furthermore its elimination tree is balanced. Nodes $\{1, 2, 3\}$ can be eliminated independent 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 (Fig. 3).

Alternatively, this matrix 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.

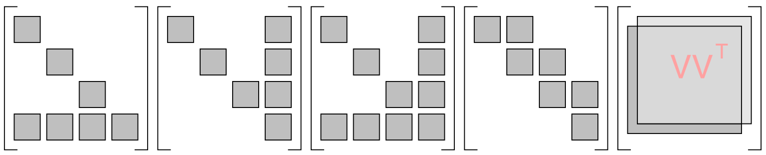


Fig. 3 Different exploitable structures: primal- and dual block-angular, bordered block-diagonal, block-banded and rank-corrector

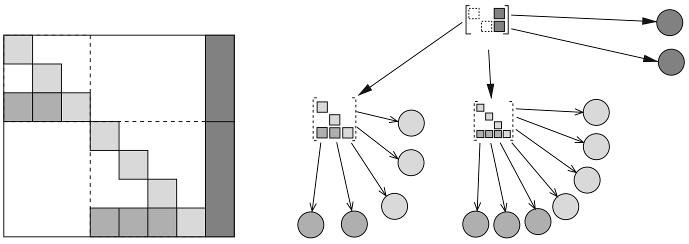


Fig. 4 Nested block-structured constraint matrix with its tree representation

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, multi-commodity network flow problems, etc) solved in telecommunications applications (Gondzio and Sarkissian 2003; Gondzio and Grothey 2003). Some formulations of Support Vector Machines (Cristianini and Shawe-Taylor 2000; Ferris and Munson 2003) have system matrices of rank-corrector structure, as have some convex reformulations of Markowitz-type financial planning problems (Gondzio and Grothey 2007a,b). 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. Adding uncertainty to an already structured problem such as in stochastic network optimization also leads to nested structure (Gondzio and Grothey 2006b; Colombo et al. 2006). In most cases the structure of the problem (or at least the process generating the structure) is known to the modeller. We therefore assume that the structure is also known to the solver, we do not try to automatically detect the structure.

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 a partitioning of associated primal and dual vectors into subvectors. The tree representation of the matrix therefore implies a tree representation of vectors in the primal and dual spaces (Fig. 4). OOPS uses `VectorTree` and `StructuredVector` classes to represent the vector tree and a vector defined on this tree. We will discuss in detail the relations between the matrix tree and the associated vector trees in Sect. 4.4.

3.3 Node-oriented linear algebra

Efficient linear algebra routines to exploit a certain known block-structure of a problem are well known and a multitude of different implementations exist (Birge and Qi 1988; Grigoriadis and Khachiyan 1996; Lustig and Li 1992). The reader interested in other parallel developments for optimization should consult (De Leone et al. 1998; Migdalas et al. 2003; Blomvall 2003; Linderoth and Wright 2003) 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 magnified, as is the multitude of different combined structures that would need to be covered.

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 matrix tree is then represented by an object of `Matrix`-type. 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 efficient 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 Fig. 4), where each node is an object of one of the classes that implement the `Matrix` interface (Fig. 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. However, this can always be done in a generic way.

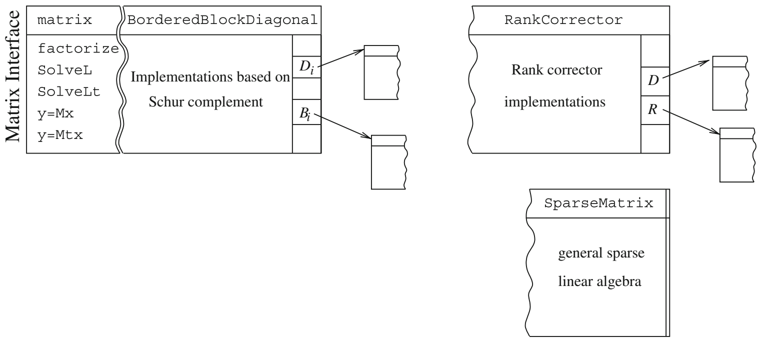


Fig. 5 The matrix interface and several implementations of it: building blocks for the tree-structure. Any of `BorderedBlockDiagonal`, `RankCorrector`, `SparseMatrix` can be used as implementations of the `Matrix` interface. `BorderedBlockDiagonal` and `RankCorrector` have submatrices which in turn are declared as `Matrix` and need an implementation specified

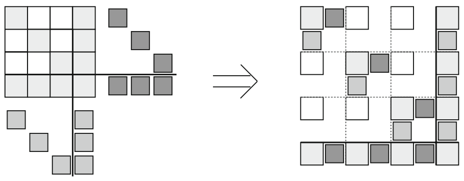


Fig. 6 Dual block-angular A and implied structure of Q and Φ

To see this, note that since A and Q by necessity have the same column dimension, we can force the use of the column vector tree implied by the nested block division of A onto Q. This implies a nested block division of Q, 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 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 (Gondzio and Grothey 2007a). Note that these are changes that improve the sparsity pattern of the augmented system matrix: they would be beneficial for any solution algorithm and are not a peculiar requirement of our design approach.

Figures 6, 7, and 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 re-modelled, or Q could be represented as a superimposition of several structures (i.e. if Q had entries in the border blocks in Fig. 8, Q could be represented as a bordered block-diagonal matrix with one diagonal block, which would then be of banded structure).

Combining the structured A and Q matrices into a structured augmented system matrix is equivalent to re-ordering the rows and columns of Φ. The result of this procedure is a matrix tree whose leaf nodes are generalised augmented system blocks of the form $\begin{bmatrix} -Q & B^T \\ A & 0 \end{bmatrix}$ where A, Q, B are unstructured sparse matrices (with $B = A$

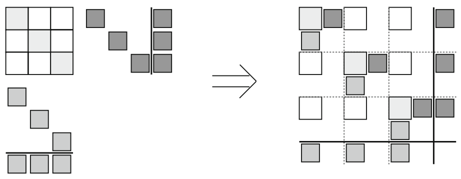


Fig. 7 Primal block-angular A and implied structure of Q and Φ

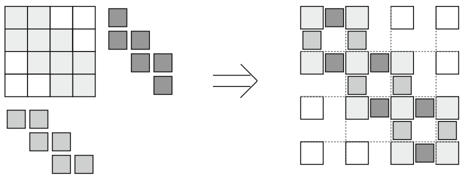


Fig. 8 Banded A and implied structure of Q and Φ

in case of a diagonal block). We use a sub-interface `AugSysMatrix` of `Matrix` to represent these blocks.

This combining 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. While the combining is generic, the *type* of `Matrix` that is used to represent the structured augmented system depends on the types of A and Q. The combining is therefore performed by a method `makeAugSystem` of the `Matrix` class. It will do the following operations:
From the input A, Q, B (=A if diagonal block).

– Determine the best combined `Matrix`-type given the types of A, Q and B.
– Create this block of the augmented System matrix, by combining the constituent matrices.

This combining of the blocks is done by recursively calling the `makeAugSystem` method for the sub-blocks of A, B and Q. This way sub-augmented system blocks (like the ones in Figs. 6, 7, 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 Gondzio and Grothey (2007a) 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 factorisation and back-solve routines in addition to the matrix–vector operations. `AugSysMatrix` is assumed to have `SimpleMatrix` components AB and Q and a `Structured-Vector` 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 sub-interfaces of `Matrix`.

4.1 Flow of control

Constructor routines exist to build the matrices A and Q from their constituting blocks for different implementations of `SimpleMatrix`. Once matrices A, Q are constructed `A.makeAugSystem(Q,B,Theta)` is called to create the augmented system matrix. `makeAugSystem` determines from the types of its two input `SimpleMatrix` the appropriate type of the `AugSysMatrix` and constructs 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.

It would be possible and worthwhile to automate the construction by the use of a modelling language that allows the modeller to encode information about the problem structure into the model. 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 discretisations over time and space. Further it would need to support nonlinear problems. We are not aware of any modelling language that satisfies these conditions. SMPS (Birge et al. 1987) (the stochastic programming extension of MPS) goes some way towards it, and an SMPS interface to our solver exists.

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 defines the following methods

```
– SimpleMatrix Constructor(...)
– StructuredVector matrixTimesVector(StructuredVector)
– StructuredVector matrixTransTimesVector
  (StructuredVector)
– StructuredVector getColumn/Row(int)
– StructuredSparseVector getSparseColumn/Row(int)
– VectorTree getPrimal/DualTree(void)
– AugSysMatrix makeAugSystem(SimpleMatrix Q,
  SimpleMatrix B, StructuredVector theta)
```

It thus includes the capability of performing matrix-vector products, retrieving a dense or sparse row or column from the matrix and setting up further structures like the primal/dual 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
`SimpleBlockDiagonalMatrix`	block-diagonal
`SimpleBorderedBlockDiagonalMatrix`	block-diagonal with dense row and column border
`SimplePrimalBlockAngularMatrix`	block-diagonal with dense row border
`SimpleDualBlockAngularMatrix`	block-diagonal with dense column border
`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 consists of references to its constituting parts A, Q, Θ and B (identical to A if symmetric). The interface supports the same methods as `SimpleMatrix` but in addition also factorisation and back-solve routines (the latter in sparse and dense modes):

```
– void symbolicFactorization(void)
– void computeCholesky(void)
– StructuredVector solveCholesky(StructuredVector)
– StructuredVector solveL/D/Lt(StructuredVector)
– StructuredVector solveSparseCholesky
  (StructuredSparseVector)
– StructuredSparseVector solveSparseL/D/Lt
  (StructuredSparse-Vector)
```

Generally the implementations of this interface breaks down the computations of matrix–vector type methods into computations on its sub-parts, calling the appropriate method of the `SparseMatrix` representing A, B and Q. The method `symbolicFactorization` determines a sparsity preserving row/column re-ordering and creates data-structures to store the re-ordered augmented system matrix and its factorisation. `computeCholesky` performs the numerical phase of the factorisation: building the (re-ordered) augmented system matrix and finding a representation of its Cholesky factors. Not all implementing classes use an implicit factorisation 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. In addition some implementations (such as those using an iterative solver) might not use a Cholesky-type factorisation at all. In this case `computeCholesky` builds a preconditioner for the system and `solveCholesky` performs the iterations of a preconditioned conjugate gradient scheme.

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 row and column border
`RankCorrectorAugSysMatrix`	Q of the form $\tilde{Q} + VV^T$

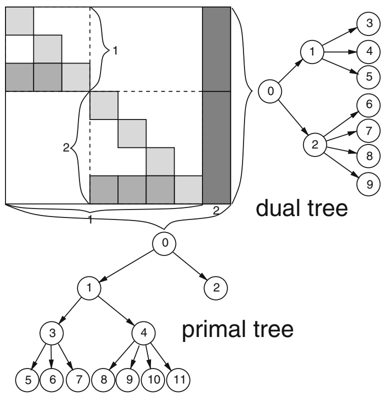
For both the `SimpleMatrix` and `AugSysMatrix` interface, the implementing classes can be classified as either *leaf node classes* such as `Dense, Sparse` or `Network` or the *complex* classes, such as `BorderBlockDiagonal` or `Rank-Corrector`. The latter are constituted from sub-matrices which themselves are of type `SimpleMatrix` or `AugSysMatrix`. The crucial idea on which the design of our library is based is that an efficient 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 efficient for their particular structure (Fig. 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 (represented by the `StructuredVector` class) For certain sub-tasks of the factorisation or back-solve routines, sparse vectors are preferable: hence we have also a mirror implementation of a `StructuredSparseVector` 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 requirement that the implementation should also work in

Fig. 9 Primal and dual vector tree derived from structured matrix



parallel, where each processor only knows (and has memory allocated for) a part of the vector.

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` by its `getPrimalTree`, `getDualTree` method. Note that rectangular matrices 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`carries information on the structure of this node and on how this node fits into the complete vector:

1. number of children, array of children (array of `VectorTrees`),
2. start and end of this node in absolute indices,
3. index number (of this node in the tree).

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. Keeping all the dense elements of the vector consecutively in memory has obvious cache advantages. The second layer has the necessary information to access these elements by nodes of the `VectorTree`. An example of the primal `VectorTree`associated with the structured matrix in Fig. 9 is displayed in Fig. 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

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

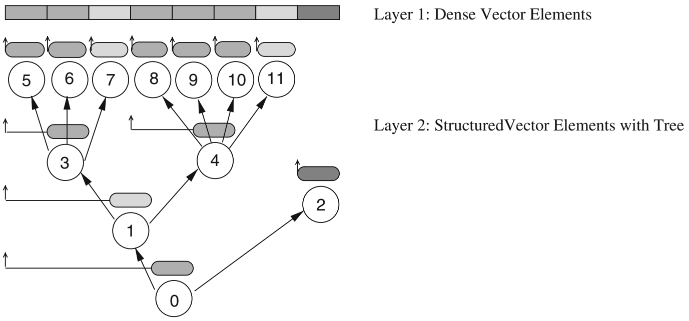
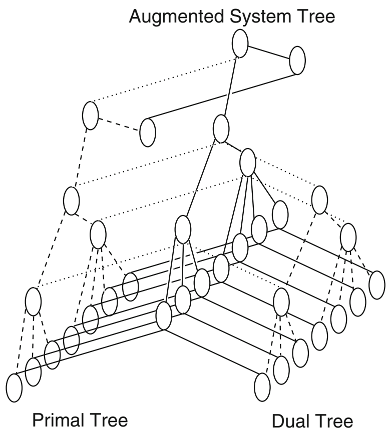

Layer 1: Dense Vector Elements
Layer 2: StructuredVector Elements with Tree
5
6
7
8
9
10
11
3
4
2
1
0

Fig. 10 The two layers of the vector representation: the ovals represent the `StructuredVector` nodes together with a pointer to the start of this node's dense elements in memory

Fig. 11 Building the augmented system tree from primal and dual tree: *solid lines* show the augmented system tree and *dashed lines* the primal/dual trees



Note that all other information (such as 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 the primal and dual vectors together as one vector structure. In this case the subvectors of the augmented system vector should not be the primal and dual vectors, but again augmented system vectors corresponding to submatrices of the augmented system consisting of interleaved primal and dual vector parts. 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 (Fig. 11).

Note that in our implementation we go the opposite route (for reasons of cache efficiency): 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 when needed.

OOPS is written largely in C/C++. Some bottom level routines that implement the elementary sparse matrix factorisation and back-solves are written in FORTRAN for efficiency reasons.

The parallel implementation of OOPS is targeted at a distributed memory architecture and uses message passing via MPI. This choice offers flexibility concerning the choice of platform. OOPS has been run on a variety of platforms ranging from a network of PCs to dedicated massively parallel machines.

5 Implementations of the `Matrix` interface

5.1 The `BorderedBlockDiagonalAugSysMatrix` class

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

$$\Phi = \begin{pmatrix} \Phi_1 & & & B_1^T \\ & \ddots & & \vdots \\ & & \Phi_n & B_n^T \\ B_1 & \cdots & B_n & \Phi_0 \end{pmatrix}, \tag{7}$$

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. Blocks of this structure are created when merging the components A and Q of mixed 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

$$L = \begin{pmatrix} L_1 & & & \\ & \ddots & & \\ & & L_n & \\ L_{n,1} & \cdots & L_{n,n} & L_c \end{pmatrix}, \quad D = \begin{pmatrix} D_1 & & & \\ & \ddots & & \\ & & D_n & \\ & & & D_c \end{pmatrix} \tag{8a}$$

where

$$\Phi_i = L_i D_i L_i^T \tag{8b}$$

$$L_{n,i} = B_i L_i^{-T} D_i^{-1} \tag{8c}$$

$$C = \Phi_0 - \sum_{i=1}^{n} B_i \Phi_i^{-1} B_i^T = L_c D_c L_c^T \tag{8d}$$

Formula (8b) needs some additional comments. As will become clear further down, L_i and D_i are only ever accessed in the form $L_i^{-1}b$, $D_i^{-1}b$, $L_i^{-T}b$, 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 `solveL, solveD, solveLt` is equivalent to a call to `solveCholesky` (i.e. formula (8b) holds). Should the class representing Φ_i use an implicit factorisation 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 `RankCorrectorAugSysMatrix`) 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 back-solve routine below is slightly more efficient (requiring 3 calls to Φ_i . `solveL/Lt` rather than the equivalent of 4 (2 times `solveCholesky`) otherwise.

Representation (8) can be used to compute the solution to the system

$$\Phi x = b,$$

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

$$z_i = L_i^{-1} b_i, \quad i = 1, \dots, n \tag{9a}$$

$$z_0 = L_c^{-1}\left(b_0 - \sum_{i=1}^{n} L_{n,i} z_i\right) \tag{9b}$$

$$y_i = D_i^{-1} z_i, \quad i = 0, \dots, n \tag{9c}$$

$$x_0 = L_c^{-T} y_0 \tag{9d}$$

$$x_i = L_i^{-T}\left(y_i - L_{n,i}^T x_0\right), \quad i = 1, \dots, n. \tag{9e}$$

Note that the matrices $L_{n,i}$ are only used in (9b, 9e) for two matrix-vector multiplications each. On the other hand the computation of $L_{n,i}$ by (8c) would require n_i solves with matrix L_i^T. In certain situations it is more efficient not to compute $L_{n,i}$ explicitly, but evaluate (9b, 9e) as

$$z_0 = L_c^{-1}\left(b_0 - \sum_{i=1}^{n} B_i L_i^{-T} D_i^{-1} z_i\right) \tag{9b$'$}$$

$$x_i = L_i^{-T}\left(y_i - D_i^{-1} L_i^{-1} B_i^T x_0\right), \quad i = 1, \dots, n \tag{9e$'$}$$

replacing the matrix–vector product with a back-solve involving L_i. Because of this L_i, D_i, L_c, D_c can be seen as an implicit Cholesky factorisation of Φ.

Further the sum to compute C in (8d) 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 $L_i^{-1}B_i^T$.

Formulae (8) and (9) rely on the ability to factorize Φ_i and the resulting Schur-complement matrix L_c in a stable manner. This is guaranteed within the IPM framework by the use of primal–dual regularization as in (6). See also Altman and Gondzio (1999) for a discussion of these issues.

All these computations are done naturally in our object-oriented environment: (8b) requires a call to `computeCholesky` for each of the diagonal parts Φ_i of Φ. The sum in (8d) is formed by `B[i].getSparseRow(...)` followed by `Φ[i].solveSparseL/D(...)` and an outer product of `StructuredSparseVector` objects to create C as a `SimpleDenseMatrix`. The back-solves can also be broken down into `AugSysMatrix` methods performed on Φ_i, B_i and C.

5.2 The `RankCorrectorAugSysMatrix` class

This class represents a matrix $\Phi = \tilde{\Phi} + VV^T$ 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

$$W = \tilde{\Phi}^{-1}V \tag{10a}$$
$$C = I + V^TW \tag{10b}$$
$$y = \tilde{\Phi}^{-1}b \tag{10c}$$
$$x = y - WC^{-1}V^Ty \tag{10d}$$

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

```
computeCholesky:

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

```
solveCholesky(b):

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

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

5.3 Sparse elementary matrices: the `SparseAugSysMatrix` class

In any sparse nested block-structured matrix the leaf nodes are 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 follows very closely traditional sparse linear algebra implementations for interior point methods including separation of symbolic and numerical factorisation and regularisation to avoid two-by-two pivoting for augmented systems (Sect. 2.3).

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 sub-tasks 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 leads 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). Spreading the data between processors further leads to more efficient caching on every processor and hence a further speed gain.

In OOPS parallelism is implemented as follows: Every node i of the matrix tree has a set of processors $\mathcal{P}(i)$ assigned to it. These processors between them share all the work needed to perform any of the `Matrix` methods on this node and its children. Data is organised in memory in such a manner that the processors in $\mathcal{P}(i)$ between them have all the data necessary to perform these operations. OOPS addresses both these issues in a manner consistent with its object-oriented design: the responsibility for distribution of computations and data lies with each class implementing the `Matrix` interface.

The distribution of processors to child nodes is performed by a method `allocateProcessors` which is part of the `Matrix` interface. `allocateProcessors(int[] procs)` 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, by calling the child's `allocateProcessors` method.

If the number of processors allocated to a node does not exactly match the number of blocks/children, the `Matrix` object in question will decide on how to pool resources and computations in an optimal way for the required tasks.

Where the parent can allocate more processors than it has children (nodes high up in the tree) more 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.

Communications between processors are coordinated by the parent class (since only this class has the necessary information on which processors are allocated to which parts of the task).

Consider the example of the `computeCholesky` and `solveL` methods from the `BorderedBlockDiagonalAugSysMatrix` class discussed in Sect. 5.1.

Φ`.factorise`:

$\Phi_1.\mathtt{factorise}$	$V_1 = \Phi_1.\mathtt{solveL}(B_1^T)$	$C_1 = V_1^T D_1^{-1} V_1$	$C_1.\mathtt{add}(\Phi_0)$	$C = \sum C_i$	$C.\mathtt{factorise}$
$\vdots$	$\vdots$	$\vdots$	*idle*		
$\Phi_n.\mathtt{factorise}$	$V_n = \Phi_n.\mathtt{solveL}(B_n^T)$	$C_n = V_n^T D_n^{-1} V_n$	*idle*		

$x = \Phi.\mathtt{solveL}(b)$:

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

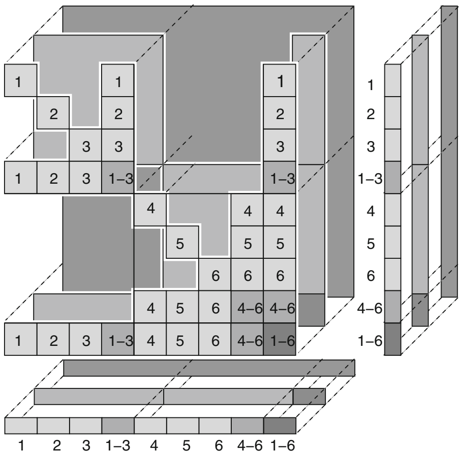
The factorisations of the diagonal blocks Φ_i and the subsequent computations of matrices $C_i = B_i L_i^{-T} D_i^{-1} L_i^{-1} B_i^T$ are independent of each other, and are distributed among available processors. The computation of the Schur complement $C = \Phi_0 - \sum_i C_i$ requires communications between the processors and the result of the final factorisation of C needs to be known on all processors allocated to the node. To save on communications the factorisation 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.

Once the computation tasks are assigned to processors, the appropriate distribution of problem data and child nodes can be derived on a 'need-to-know' basis. In the above example diagonal blocks Φ_i are 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 allocated to all processors.

The distribution of `VectorTree`nodes follows the distribution of the corresponding matrix nodes: Nodes x_i are distributed, whereas the node corresponding to the border blocks x_0 is stored on all processors.

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 Figs. 4 and 9. Each level of Fig. 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 leaf nodes describing elementary matrices and vector

Fig. 12 Allocating matrix and vector blocks to processors



parts. Since the matrix and vector data is held only in the leaf nodes this layer also indicates on which processors different parts of the problem data are kept.

6.1 Loading the matrix: parallel program flow

Since the allocation of processors to nodes is done by a `Matrix` method, there is an obvious bootstrapping problem: For the recursive allocation of processors to nodes, the *whole* `Matrix` tree needs to be known to all processors. On the other hand as little problem data as possible should be kept on each processor. To overcome this OOPS implements the following bootstrapping procedure:

The complete matrix tree is kept on all processors. This includes the `Matrix` objects containing the implementations of the linear algebra methods and pointers to the child nodes (but **no** information about the problem data). Since this information consists exclusively of pointers, very little memory is required.

On all processors not in $\mathcal{P}(i)$, node i in the matrix tree is replaced by a `Fake Matrix` 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.

In a second stage a recursive call to `fillLeafNodes` sets up the required problem data (i.e. sparse matrix entries, right-hand side and objective vectors) on each leaf node. Due to the use of `FakeMatrix` this is done only on those processors that need the data. This of course implies that every part of the problem data *could* be generated on every processor. Either appropriate C/C++-routines must be present on all processors, or a modelling system is required to run on all processors. This is clearly not desirable either. In a separate project (Grothey et al. 2009) we suggest a parallelisable modelling system that generates problem data only on those processors that need the data.

A useful side-effect of this setup is that most of the parallelisation of the linear algebra methods is done automatically. A computation such as (8) and (9) is coded on every processor as written (indeed as it would be in a serial implementation). The implementation trick used in `FakeMatrix` guarantees that every processor only does those computations for which it has the required data. In effect a sum such as (9b) or (8d) 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 using the provided MPI Communicators. 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.

Below we give an overview of the bootstrapping phase:

1. Build the Matrix-tree on *all* processors. Data for sparse leaf matrices is not generated yet.
2. Call `allocateProcessors` recursively to allocate nodes in the tree to processors. On processors not in $\mathcal{P}(i)$ the `Matrix` object is replaced by `FakeMatrix`.
3. Create primal and dual `VectorTree`recursively. They inherit their processor allocation from the corresponding `Matrix` object in the associated augmented system tree.
4. Another recursive call to `Matrix` method `fillLeafNodes` generates the data describing matrix and vector parts in the leaf nodes on the appropriate processors only.
5. Start interior point algorithm.

7 Numerical results

The power of the structure exploiting interior point solver has been demonstrated in a wide range of applications. OOPS has been used to solve multistage stochastic portfolio optimization problem on the UK High Performance computing facility HPCx and the BlueGene/L machine at EPCC, Edinburgh with 1280 and 1024 processors respectively. The largest of these problems solved had over 12 million scenarios and 1.01×10^9 decision variables (Gondzio and Grothey 2006a).

Good parallel scaling and superiority over commercial non structure exploiting solver CPLEX has been demonstrated on a range of nonlinear variants of the portfolio optimization problem (Gondzio and Grothey 2007a,b). Further OOPS has been used to solve stochastic utility distribution problems (Gondzio and Grothey 2006b) and stochastic network optimization problems (Colombo et al. 2006).

Acknowledgments We wish to thank the anonymous referee for the valuable comments. This paper was supported by the Engineering and Physical Sciences Research Council of UK, EPSRC grant GR/R99683/01.

References

Altman A, Gondzio J (1999) Regularized symmetric indefinite systems in interior point methods for linear and quadratic optimization. Optim Methods Softw 11(12):275–302

Andersen ED, Gondzio J, Mészáros C, Xu X (1996) Implementation of interior point methods for large scale linear programming. In: Terlaky T (ed) Interior point methods in mathematical programming. Kluwer Academic Publishers, New York, pp 189–252

Arioli M, Duff IS, de Rijk PPM (1989) On the augmented system approach to sparse least-squares problems. Numerische Mathematik 55:667–684

Benson S, McInnes LC, Moré JJ (2001) TAO users manual. Tech. Rep. ANL/MCS-TM-249, Argonne National Laboratory

Birge J, Dempster M, Gassmann H, Gunn E, King A, Wallace S (1987) A standard input format for multiperiod stochastic linear programs. Comm Algorithms Newslett 17:1–19

Birge JR, Qi L (1988) Computing block-angular Karmarkar projections with applications to stochastic programming. Manage Sci 34:1472–1479

Blomvall J (2003) A mulitstage stochastic programming algorithm suitable for parallel computing. Parallel Comput 29:431–445

Blomvall J, Lindberg PO (2002) A Riccati-based primal interior point solver for multistage stochastic programming. Eur J Oper Res 143:452–461

Colombo M, Gondzio J, Grothey A (2006) A warm-start approach for large-scale stochastic linear programs. Technical Report MS-06-004, School of Mathematics, University of Edinburgh, Edinburgh EH9 3JZ, Scotland, UK, August

Cristianini N, Shawe-Taylor J (2000) An introduction to support vector machines and other kernel based learning methods. Cambridge University Press, London

De Leone V, Murli A, Pardalos P, Toraldo G (eds) (1998) High performance algorithms and software in nonlinear optimization. Kluwer Academic Publisher, New York

Duff IS, Erisman AM, Reid JK (1987) Direct methods for sparse matrices. Oxford University Press, New York

Ferris MC, Munson TS (2003) Interior point methods for massive support vector machines. SIAM J Optim 13:783–804

George A, Liu JWH (1989) The evolution of the minimum degree ordering algorithm. SIAM Rev 31:1–19

Gertz EM, Wright SJ (2003) Object-oriented software for quadratic programming. ACM Trans Math Softw 29:58–81

Gondzio J, Grothey A (2003) Reoptimization with the primal–dual interior point method. SIAM J Optim 13:842–864

Gondzio J, Grothey A (2006a) Direct solution of linear systems of size 10^9 arising in optimization with interior point methods. In: Wyrzykowski R (ed) Parallel Processing and Applied Mathematics. Lecture Notes in Computer Science, vol 3911. Springer, Berlin, pp 513–525

Gondzio J, Grothey A (2006b), Solving distribution planning problems with the interior point method. Technical Report MS-06-001, School of Mathematics, University of Edinburgh, Edinburgh EH9 3JZ, Scotland, UK, February

Gondzio J, Grothey A (2007a) Parallel interior point solver for structured quadratic programs: Application to financial planning problems. Ann Oper Res 152:319–339

Gondzio J, Grothey A (2007a) Solving nonlinear portfolio optimization problems with the primal–dual interior point method. Eur J Oper Res 181:1019–1029

Gondzio J, Sarkissian R (2003) Parallel interior point solver for structured linear programs. Math Program 96:561–584

Grigoriadis MD, Khachiyan LG (1996) An interior point method for bordered block-diagonal linear programs. SIAM J Optim 6:913–932

Grothey A, Hogg J, Woodsend K, Colombo M, Gondzio J (2009) A structure-conveying parallelisable modelling language for mathematical programming. In: Ciegis R, Henty D, Kågström B, Žilinskas J (eds) Parallel scientific computing and optimization: advances and applications. Springer optimization and its applications, vol 27. Springer, Berlin, pp 147–158

Linderoth J, Wright SJ (2003) Decomposition algorithms for stochastic programming on a computational grid. Comput Optim Appl 24:207–250

Lustig IJ, Li G (1992) An implementation of a parallel primal–dual interior point method for multicommodity flow problems. Comput Optim Appl 1:141–161

Meza J, Oliva R, Hough P, Williams P (2007) OPT++: An object oriented toolkit for nonlinear optimization. ACM Transactions on Mathematical Software 33, p. 12. Article 12, 27 pages

Migdalas A, Toraldo G, Kumar V (2003) Parallel computing in numerical optimization. Parallel Comput 29:373–373

Steinbach M (2000) Hierarchical sparsity in multistage convex stochastic programs. In: Uryasev S, Pardalos PM (eds) Stochastic optimization: algorithms and applications. Kluwer Academic Publishers, New York, pp 363–388

Vanderbei RJ (1995) Symmetric quasidefinite matrices. SIAM J Optim 5:100–113

Wright SJ (1997) Primal–dual interior-point methods. SIAM, Philadelphia