Parallel Interior Point Solver for Structured Linear Programs

Jacek Gondzio Robert Sarkissian

December 4th, 2000
Revised February 2nd, 2002 and November 17th 2002

MS-2000-025

For other papers in this series see http://www.maths.ed.ac.uk/preprints
To appear in **Mathematical Programming**.

Parallel Interior-Point Solver for Structured Linear Programs*

Jacek Gondzio† Robert Sarkissian‡

Department of Mathematics & Statistics
The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ
United Kingdom.

December 4th, 2000, revised February 2nd, 2002 and November 17th 2002

*Supported by the Engineering and Physical Sciences Research Council of UK, EPSRC grant GR/M68169.
Accepted for publication in **Mathematical Programming**.

†Email: gondzio@maths.ed.ac.uk, URL: http://maths.ed.ac.uk/~gondzio/

‡Email: Robert.Sarkissian@hec.unige.ch, URL: http://ecolu-info.unige.ch/~logilab/sarkissian/sarkissian.html

Parallel Interior-Point Solver for Structured Linear Programs

Abstract

Issues of implementation of an object-oriented library for parallel interior-point methods are addressed. The solver can easily exploit *any* special structure of the underlying optimization problem. In particular, it allows a nested embedding of structures and by this means very complicated real-life optimization problems can be modelled. The efficiency of the solver is illustrated on several problems arising in the optimization of networks. The sequential implementation outperforms the state-of-the-art commercial optimization software. The parallel implementation achieves speed-ups of about 3.1-3.9 on 4-processors parallel systems and speed-ups of about 10-12 on 16-processors parallel systems.

1 Introduction

Optimization models often involve system dynamics, uncertainty, spatial distribution or other factors that lead to huge problems made up of small, nearly identical parts which have to be coordinated through time, uncertainty, space or other dimensions. These real-life models present a challenge to the state-of-the-art optimization software and sequential processing hardware. As a consequence, they need *specialized* optimization approaches and considerable computing facilities.

Many real-life linear programs (LPs) display some particular block structures that deserve special treatment by an optimization code. A well-known example is the block-angular structure of constraints that can be handled by a decomposition approach [4, 11]. Decomposition allows tight control of memory requirements and considerable computational efficiency, in particular through parallel computation [5].

An alternative is to use a direct solution method. There are two major approaches to the solution of linear programs: simplex method [10] and interior-point method [28]. There exist efficient implementations of both methods for *general* problems. An advantage of interior-point methods (IPMs) is that they require a number of iterations that is almost independent of the problem size [2, 22]. For very large problems they are sometimes substantially faster.

The implementations of both direct approaches might exploit the structure in many different ways. Some general purpose solvers use heuristics to detect certain structures and then take advantage of them. For example, simplex solvers look for network constraints while interior-point solvers detect dense columns.

There are several papers that address the issues of structure exploitation in the implementation of interior-point methods. A nonexhaustive list includes: [6, 9, 15, 16, 18, 19, 26, 27]. These papers describe specialized algorithms each exploiting one particular structure of the constraint matrix. Some of them [16, 19, 26] present dedicated parallel implementations of these methods.

Another approach is to incorporate in a solver a set of routines that can support *any* structure. An object-oriented implementation of such a set of routines within the context of an interior-point solver is the subject of this paper.

The approach takes advantage of inclusion polymorphism: we define a class for matrix operations from which we derive all other classes. This abstract matrix class contains a set of virtual functions (methods in the object-oriented terminology) that:

- provide all the necessary linear algebraic operations for an IPM, and
- allow self-referencing.

Among derived classes we define elementary ones for sparse, dense, network, identity and projection, as well as classes for block-structured matrices such as block-diagonal, primal block-angular, dual block-angular. Definitions of the latter use references to abstract matrix classes (place-holder objects). This self-referential property of our block-structured matrix classes allows us to represent a variety of nested structures.

We assume that the stuctured LP constraint matrix is built of a nested set of blocks. The hierarchy of blocks can be naturally represented as a tree: its root is the whole matrix, intermediate nodes are block-structured submatrices and the leaves are the elementary blocks. The tree defines how the different class objects are combined to represent the matrix structure. In this context the root corresponds to the matrix seen by an interior-point method, any leaf node corresponds to one of the elementary matrices and any intermediate node corresponds to a block-structured matrix.

Block-matrix operations are natural candidates for parallelisation. We have therefore implemented all higher-level matrix clasess in parallel. The parallelisation is coarse-grained and so is well suited to large scale linear programming applications.

With our object-oriented design of an interior-point method we expect to achieve three goals:

- reduce storage requirements of the algorithm (in consequence, to be able to solve larger problems);
- accelerate the computations by doing more block matrix operations; and
- enable easy parallelization of the algorithm.

The paper is organized as follows. In Section 2 we briefly discuss the linear algebraic operations required to implement the interior-point method. In Section 3 we introduce the tree representation of the block-structured matrices and discuss certain features of the abstract matrix class, the crucial object used in our design to support the notion of block. In Section 4 we discuss the special structures of the problems that are most often met in practical applications. In Section 5 we address issues of design and implementation of the object-oriented library used to handle linear algebraic operations of block-structured matrices. In Section 6 we describe several network optimization problems that have been used to illustrate the efficiency of our new interior-point code. In Section 7 we discuss numerical results and finally, in Section 8, we give our conclusions.

2 Linear Algebra in Interior-Point Methods

The theory [28] and the implementation [2, 22] of interior-point methods for general linear programs are very well understood. It seems that for very large LPs with unknown structure interior-point methods are often a better choice than the simplex method, at least for solving isolated problems. The efficiency of IPM strongly depends on the linear algebra.

The dominant computational task in an interior-point method [2] is the solution of the following linear equation system

$$\begin{bmatrix} -\Theta^{-1} & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} r \\ h \end{bmatrix}, \tag{1}$$

where $A \in \mathcal{R}^{m \times n}$ is the LP constraint matrix, $\Theta \in \mathcal{R}^{n \times n}$ is a diagonal scaling matrix that changes at every iteration, r and h are right hand side vectors, and Δx and Δy are Newton directions in the primal and dual spaces, respectively. The matrix of the linear system (1) is symmetric but indefinite. By substituting

$$\Delta x \;=\; \Theta A^T \Delta y - \Theta r,$$

we get the symmetric, positive definite system

$$(A\Theta A^T)\Delta y \;=\; g = A\Theta r + h. \tag{2}$$

Almost all interior-point solvers use either (1) or (2) and usually split the solution of these linear equation systems into two steps [2]:

- factorization to LL^T form,
- backsolve to compute direction Δx and Δy.

The factorization LL^T is either the Cholesky or another symmetric triangular decomposition. Although in general purpose IPM codes these linear algebraic operations are implemented with particular care, the solvers cannot in general take advantage of the special structure of matrix A in the inversion of the matrix in the system (1) or (2). An exception is the detection of dense columns in matrix A and handling them by the Schur complement mechanism, which is an option available in some interior-point solvers.

In this paper we assume that the augmented system (1) is always reduced to the normal equations (2). (It does not imply, however, that the normal equations matrix $A\Theta A^T$ is explicitly formed.) We further assume that an inverse representation of $A\Theta A^T$ is computed. This representation may be *implicit*, i.e., it does not necessarily have the form of a symmetric LL^T factorization. However, the representation provides an efficient method for the solution of equations involving $A\Theta A^T$.

An implicit inverse representation offers certain advantages over explicit Cholesky factorization. First, it often requires significantly less memory than Cholesky factorization of $A\Theta A^T$. Secondly, in many cases it better exploits the sparsity of A and leads to CPU time savings compared with the Cholesky factorization. Moreover, it exploits block matrix operations and allows natural parallelization of computations. Finally, by working with small block matrices a lot of paging can be avoided and cache memory is used more efficiently.

To implement an interior-point method for linear programming one can use the following linear algebraic operations to be done with matrix $A \in \mathcal{R}^{m\times n}$ and vectors of appropriate dimensions:

- Given A, x, y and Θ, compute Ax, A^Ty and $A\Theta A^T$,
- Given $A\Theta A^T$, compute the inverse representation (factorization) $A\Theta A^T = LL^T$, (3)
- Given L and g, solve $Lz = g$ or $L^Tz = g$.

These operations are an integral part of the abstract matrix class used in our solver. It is worth noting that an interior-point algorithm does not need to know *how* these operations are executed; it needs to access only their *results*.

We take advantage of this property to define a *polymorphic matrix class*, which is a class of algebraic operations for interior-point methods.

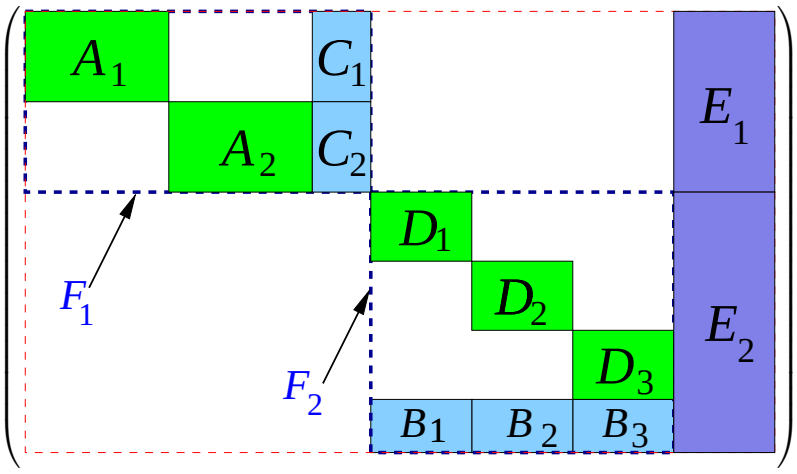

Figure 1: Example of Block-Structured Matrix A.

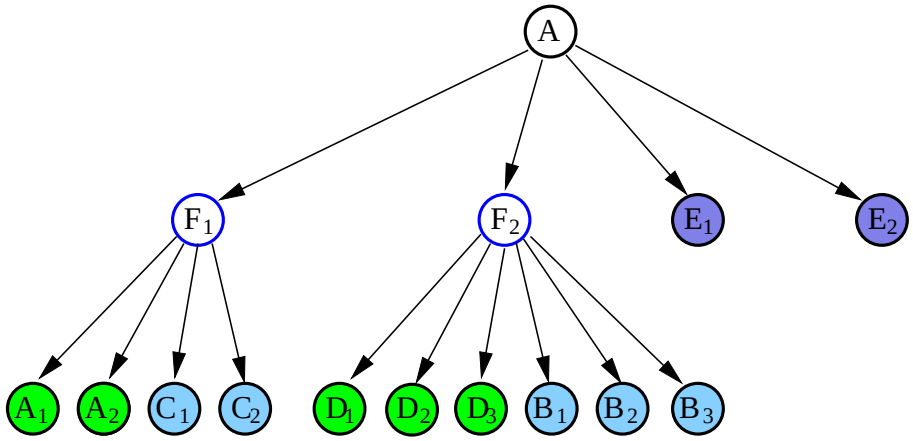

Figure 2: Tree Representation of Blocks.

3 Block-Structured Matrices

There are many structures that can be exploited by an interior-point solver. We restrict our discussion to those in which the LP constraint matrix A is built of blocks. There are two reasons for that. First, such structures are most often met in practice because they reflect the presence of dynamics, uncertainty, space distribution or other similar factors in the optimization problem. Secondly, block matrix structures can and should be exploited in any optimization algorithm that heavily involves linear algebraic operations, and the interior-point method is such an algorithm.

We assume that the special structure of A is known to the user. This is different from the approach of [12] in which an anonymous linear program (not necessarily specially structured) is reordered to a special form that can be exploited by an optimization algorithm. In [12] any matrix is reordered to a bordered block-angular form by a special reordering heuristic like the one of [21] and then the problem is solved by an optimization algorithm specialized for this particular structure.

3.1 Tree Representation of Block-Structured Matrices

With a given block-structured matrix we associate a *tree* that defines the embedding of blocks. Every node in this tree corresponds to a block of the matrix. An arc indicates the embedding of blocks. The root of the tree is the whole matrix, and its leaves correspond to elementary blocks that are not partitioned into smaller entities. Any intermediate node of the tree corresponds to some submatrix that is further partitioned into blocks; the nodes corresponding to these blocks are children of a given node. Figure 1 gives an example of the structured matrix and Figure 2 shows the corresponding tree.

The tree provides a complete description how the matrix is partitioned. In the example of Figure 1 the matrix is first partitioned into blocks F_1, F_2, E_1 and E_2 corresponding to dual block-angular structure. Block F_1 has also dual block-angular structure: it is partitioned into subblocks A_1, A_2, C_1 and C_2. Block F_2 has primal block-angular structure with three diagonal blocks D_1, D_2, D_3 and three row-border blocks B_1, B_2, B_3.

With every node of the tree we associate the *type* of the block-structure. We will say for example that A is a dual block-angular matrix with 2 diagonal blocks, F_1 is also a dual block-angular matrix with 2 diagonal blocks, F_2 is a primal block-angular matrix with 3 diagonal blocks, and all the remaining nodes are elementary matrices, e.g., sparse, dense, network, etc. Every *type* of the block structure corresponds in our implementation to a particular `Matrix` object class.

Observe that the *type* of the node determines how the linear algebraic operations for the corresponding block of the matrix should be executed. Consider one of the simplest operations such as the matrix-vector product computed with A. The type of node A in the tree implies that the operation will be split into 4 subblocks F_1, F_2, E_1 and E_2, corresponding to children of this node. In cases of nodes E_1 and E_2 that correspond to elementary blocks, this operation would invoke a routine appropriate for the given type of the node. For nodes F_1 and F_2 the same operation will have to be split further into subblocks — children of nodes F_1 and F_2. However, the same operation of matrix-vector product will be executed in a different way for nodes F_1 and F_2 because the types of these nodes are different.

3.2 Linear Algebraic Operations for Block-Structured Matrices

We have already seen that the form of the tree and the types of its nodes determine the unique way in which matrix-vector multiplication with A should be unfolded to gather the results of partial matrix-vector multiplications from all its submatrices. We also observe that to compute a matrix-vector product for any node in the tree one only needs to know the *results* of matrix-vector products computed for all children of this node; one does not have to know *how* these products are computed. This is a key feature that we shall exploit in a definition of a polymorphic abstract matrix class for linear algebra operations on block-structured matrices.

Note that a block is a submatrix of the original matrix: the operations on the original matrix induce operations on its blocks. Also, a block may itself be block-structured. Therefore, there is a recursivity which should be supported in the definition of a block. Among the blocks we will distinguish *elementary* blocks, i.e., those which are not composed of smaller entities and *derived* blocks, i.e., those which are partitioned into smaller blocks.

We now consider the set of operations that are required to implement linear algebra for nested block-structured matrices. This set of operations $\mathcal{F}_{\mathcal{X}}$ has to satisfy the *closure* condition: any operation for a derived block can be obtained by combining operations for its subblocks. We have seen that (3) was the set of operations required by an IPM, hence $\mathcal{F}_{\mathcal{X}}$ should cover (3).

To illustrate this condition better, let us observe that the matrix-vector product operation for block-structured matrices needs only matrix-vector product operations for subblocks. For efficiency reasons, certain more complicated linear algebraic operations for block-structured matrices (building $A\Theta A^T$, factorizing it, etc) may need additional operations. Below we give a complete list of these operations used in our design of an abstract matrix class. The reasons for some of them will become clearer after discussing in more detail the inverse representations used for block-angular matrices.

Below we assume that A is a block of the LP constraint matrix, the vectors x and y are parts of the vectors corresponding to this block, and the matrix Θ is a diagonal scaling matrix corresponding to this block. We also assume that the inverse representation of $A\Theta A^T$ preserves symmetry so we write it in a form $A\Theta A^T = LL^T$. However, matrix L should not be understood as the Cholesky factor of $A\Theta A^T$. It may be a part of an implicit inverse representation. The abstract `Matrix` class has to provide the following linear algebraic operations:

- compute $A\Theta A^T$ for a given Θ,
- compute the inverse representation $LL^T = A\Theta A^T$,
- compute the solution of equation $(A\Theta A^T)x = y$,
- compute the solution of equation $Lx = y$,
- compute the solution of equation $L^T x = y$,
- compute the product $Ax = y$,
- compute the product $A^T x = y$,
- retrieve column j of A,
- solve a triangular equation with the column j of A, i.e., compute $x = L^{-1}a_{\cdot j}$,
- solve a triangular equation with the ith unit vector, i.e., compute $x = L^{-1}e_i$.

We denote this set of operations by $\mathcal{F}_{\mathcal{X}}$.

4 Exploitable Structures

We allow any structure to be exploited in the linear algebraic operations of the interior-point method. We allow many different structures to be exploited within the same framework.

To make a given structure exploitable, we developed a set of routines that implement the necessary linear algebraic operations. These routines can be invoked directly by the interior-point method and also indirectly to deal with subblocks of intermediate blocks. The approach is flexible enough to allow many kinds of matrices to be included in it. Obviously, adding a new matrix class needs some programming effort. However, adding it always expands the class of problems that can be modelled within our system.

As a starting point, we have implemented a set of elementary matrix classes (general sparse matrix, general dense matrix, network adjacency matrix, identity matrix, projection matrix) and a set of block-matrix classes (block-diagonal matrix, primal block-angular matrix and dual block-angular matrix). We shall describe some of them below.

Two elementary matrix classes for general sparse and dense matrices are the major blocks the other classes are often built of. They provide all the necessary operations to solve linear programs in which matrix A is a general sparse or a general dense matrix, respectively. We have developed three more specialized elementary matrix classes for network, identity and projection matrices. Although these matrices could be treated as general sparse matrices, we found the presence of the corresponding classes useful in modeling certain network optimization problems.

A first example of a potentially exploitable block structure is the so-called *staircase* structure which is often found in mathematical programs that model dynamic processes. The simplex method can take advantage of this structure [13], both in the routines that handle the basis inverse and in the pricing. However, we have not developed a specialized matrix class for this structure because it is naturally expressed by a general sparse matrix. Indeed, Cholesky factors of $A\Theta A^T$ matrices preserve the staircase form and display moderate fill-in.

We have developed matrix classes for two block-angular structures, primal block-angular and dual block-angular. The numerical operations required by these matrices are rather straightforward; they rely on some elementary linear algebra for block structured matrices and can be found for example in [7]. These techniques are widely used in optimization and are spread across optimization literature under different names. We recall them below and use them to illustrate our developments. To simplify the discussion and the mathematical formulae we omit the diagonal scaling matrix Θ. This matrix does change the normal equations system but it does not change the sparsity pattern (structure) of this system so it can be dropped without reducing the generality of our discussion.

4.1 Primal Block-Angular Structure

Suppose a block of the constraint matrix of the linear program has the primal block-angular structure

$$A = \begin{pmatrix} A_1 & & & & \\ & A_2 & & & \\ & & \ddots & & \\ & & & A_n & \\ B_1 & B_2 & \cdots & B_n & B_{n+1} \end{pmatrix}, \tag{4}$$

where $A_i \in \mathcal{R}^{m_i \times n_i}, i = 1, ..., n$ and $B_i \in \mathcal{R}^{m_0 \times n_i}, i = 1, ..., n+1$. Matrix A has then $M = \sum_{i=0}^{n} m_i$ rows and $N = \sum_{i=1}^{n+1} n_i$ columns.

The corresponding normal equations matrix has the form

$$AA^T = \begin{pmatrix} A_1A_1^T & & & & A_1B_1^T \\ & A_2A_2^T & & & A_2B_2^T \\ & & \ddots & & \vdots \\ & & & A_nA_n^T & A_nB_n^T \\ B_1A_1^T & B_2A_2^T & \cdots & B_nA_n^T & C \end{pmatrix}, \tag{5}$$

where

$$C = \sum_{i=1}^{n+1} B_iB_i^T. \tag{6}$$

Its bordered form is preserved by the Cholesky factorization. We compute the following implicit inverse

$$\begin{array}{rcll} A_iA_i^T & = & L_iL_i^T, & i = 1,2,...,n \\ \tilde{B}_i & = & B_iA_i^TL_i^{-T}, & i = 1,2,...,n \\ S = C - \sum_{i=1}^n \tilde{B}_i\tilde{B}_i^T & = & L_SL_S^T. & \end{array} \tag{7}$$

Assume a vector $g \in \mathcal{R}^M$ is given and we want to solve equation $AA^Tz = g$. We then partition both vectors z and g accordingly to the row partition of matrix A into blocks $z = (z_1, z_2, ..., z_n, z_B)$ and $g = (g_1, g_2, ..., g_n, g_B)$ and solve the sequence of triangular equation systems:

$$\begin{array}{rcll} L_iv_i & = & g_i, & i = 1,2,...,n \\ L_Su & = & g_B - \sum\limits_{i=1}^{n} B_iA_i^TL_i^{-T}v_i & \\ L_S^Tz_B & = & u & \\ L_i^Tz_i & = & v_i - L_i^{-1}A_iB_i^Tz_B, & i = 1,2,...,n. \end{array} \tag{8}$$

It is worth noting that the order in which matrix multiplications are performed in (7) and (8) may have a significant influence on the overall efficiency of the solution of equation $AA^Tz = g$. In many cases, for example, it is advantageous to compute S in (7) as a sum of outer products of columns of matrices $\tilde{B}_i$. Similarly, $B_iA_i^TL_i^{-T}v_i$ in (8) can almost always be computed fastest if the order of multiplications is the following $B_i(A_i^T(L_i^{-T}v_i))$. An important reduction of memory requirements can be achieved by avoiding storing blocks $\tilde{B}_i$ by dealing with them in an implicit form. Indeed, even if the matrix L_i is very sparse, its inverse may be quite dense, which would inevitably lead matrix $\tilde{B}_i = B_iA_i^TL_i^{-T}$ being quite dense.

Summing up, important savings can be achieved in the storage requirements as well as in the efficiency of computations if the linear algebraic operations exploit the block structure of matrix A. Last but not least, exploiting block structure allows an almost straightforward parallelization of many of the computational steps in (7) and (8).

4.2 Dual Block-Angular Structure

Suppose a block of the constraint matrix of the linear program has the dual block-angular structure

$$A = \begin{pmatrix} A_1 & & & C_1 \\ & A_2 & & C_2 \\ & & \ddots & \vdots \\ & & A_n & C_n \end{pmatrix}, \tag{9}$$

where $A_i \in \mathcal{R}^{m_i \times n_i}, i = 1, ..., n$ and $C_i \in \mathcal{R}^{m_i \times k}, i = 1, ..., n$. Matrix A has then $M = \sum_{i=1}^{n} m_i$ rows and $N = k + \sum_{i=1}^{n} n_i$ columns.

The corresponding normal equations matrix has the form

$$AA^T = \begin{pmatrix} A_1A_1^T & & & \\ & A_2A_2^T & & \\ & & \ddots & \\ & & & A_nA_n^T \end{pmatrix} + CC^T,$$

where $C = [C_1^T, C_2^T, ..., C_n^T]^T \in \mathcal{R}^{M \times k}$ defines a rank-k corrector. The matrix AA^T may become very dense if columns of C are dense. Should this happen, the Cholesky factorization of AA^T would become prohibitively expensive. As in the previous section, it is advantageous to apply an implicit inverse. This relies on the Sherman-Morrison-Woodbury formula and requires computing the following factorization:

$$\begin{array}{rcll} A_iA_i^T & = & L_iL_i^T, & i = 1, 2, ..., n \\ \tilde{C}_i & = & L_i^{-1}C_i, & i = 1, 2, ..., n \\ S = I_k + \sum_{i=1}^{n} \tilde{C}_i^T\tilde{C}_i & = & L_SL_S^T. & \end{array} \tag{10}$$

Having defined a block-diagonal matrix

$$LL^T = \text{diag}(L_iL_i^T), \tag{11}$$

the inverse of AA^T can be expressed in the following way:

$$\begin{array}{rcl} (AA^T)^{-1} & = & (LL^T + CC^T)^{-1} \\ & = & (LL^T)^{-1} - (LL^T)^{-1}CS^{-1}C^T(LL^T)^{-1} \end{array} \tag{12}$$

with all inversions easy to compute. This formula can farther be simplified to exploit the symmetry

$$\begin{array}{rcl} (AA^T)^{-1} & = & L^{-T}(I - L^{-1}CS^{-1}C^TL^{-T})L^{-1} \\ & = & L^{-T}(I - \tilde{C}S^{-1}\tilde{C}^T)L^{-1}. \end{array} \tag{13}$$

Assume a vector $g \in \mathcal{R}^M$ is given and we want to solve equation $AA^Tz = g$. We partition both vectors z and g accordingly to the row partition of matrix A into blocks $z = (z_1, z_2, ..., z_n)$ and

$g = (g_1, g_2, ..., g_n)$ and execute the sequence of multiplications and triangular solves:

$$\begin{aligned} L_i v_i &= g_i, \qquad i = 1, 2, ..., n \\ u &= \sum_{i=1}^{n} C_i^T L_i^{-T} v_i \\ L_S t &= u \\ L_S^T v &= t \\ L_i^T z_i &= v_i - L_i^{-1} C_i v, \quad i = 1, 2, ..., n. \end{aligned} \tag{14}$$

Again it is worth noting that the order in which multiplications in some of the equations in (10) and (14) are executed may have a significant influence on the overall efficiency of the solution of equation $AA^T z = g$. In many cases, for example, it is advantageous to compute S in (10) as a sum of outer products of columns of matrices $\tilde{C}_i^T$. Similarly, the result of the multiplication $\tilde{C}_i v = L_i^{-1} C_i v$ in (14) can almost always be computed fastest if the multiplication $C_i v$ is done first and it is followed by only one triangular solve with L_i. As in the case of the primal block-angular matrix, an important reduction of memory requirements can be achieved by avoiding storing blocks $\tilde{C}_i$ and instead dealing with them in an implicit form. Finally, many steps in (10) and (14) can be executed in parallel.

In the case of primal and dual block-angular matrices all operations on subblocks could be limited to the set (3). For efficiency reasons however we extend this set to a larger one, $\mathcal{F}_\mathcal{X}$. The closure condition for $\mathcal{F}_\mathcal{X}$ has to be ensured.

We have now shown that for two particularly structured matrices, the solution of linear equations can be rearranged to take advantage of the block structure. Let us observe that many linear algebraic operations can be done with whole matrix blocks, which leads to flexible solution schemes for distributed computing. The implicit inverse representation offers significant storage savings and in some cases may also result in an improvement in the speed of computations. In a case when the available computing resources may not be sufficient to generate the whole LP constraint matrix of some very large problem, the implicit inverse representation working with parts of the problem may remain the only solution alternative. On the other hand, implicit inverse representation schemes are more sensitive to accuracy issues. We are currently working on specially adapted techniques that can be used in this context.

The presence of different matrix classes specialized for some block structure of matrices is justified whenever it can offer advantages either in the storage or in the efficiency of computations. We have used two block-angular structures to illustrate some of these advantages. As will be seen in the discussion in the next section, the framework we have developed allows the easy incorporation of many specialized matrices that can exploit different block structures.

5 Design and Implementation

The implementation has certain key features that provide a significant enhancement in exploiting block structures. Below we discuss the most important features of our design.

Though we speak of object-oriented programming, our program is written in C and we use pointers of functions and `void*` pointers to implement the desired object-oriented programming

features. For the sake of clarity, we will however use the Java programming language terminology [3] to convey the ideas behind the implementation. Programs written in a higher level language such as C++ or Java tend to be slower than those in a language like C. Given the compilers, hardware and operating systems at the time we designed the program, we accepted some loss of clarity to achieve good performance.

In terms of programming, a structure exploiting solver should provide the basic elements a user may need to take advantage of a particular structure in a matrix. The solver should be easy to extend without any modifications to existing elements. The basic elements of the solver should provide a laboratory for rapidly experimenting with the exploitation of different structures, since the best combination of basic elements is unlikely to be known in advance.

Finally, we stress that our object-oriented implementation need not add any overhead computation: if the structure is not exploited, the program should be as efficient as a general sparse IPM code. However, we intend to use our implementation for structured problems and expect to improve the performance.

5.1 Methods in the `Matrix` Class

One of the main difficulties in the design follows from the requirement for the `Matrix` object class to be self-referencial. Indeed, we want the same `Matrix` class to describe all possible blocks of the matrix, both elementary and derived. Thus the operations provided by the `Matrix` class (methods in the object-oriented terminology) must be sufficient to execute any linear algebraic operations needed by the higher level blocks. It is easier to meet this closure condition if the set of methods $\mathcal{F}_{\mathcal{X}}$ in the `Matrix` class is small. As previously observed, it would be possible to restrict $\mathcal{F}_{\mathcal{X}}$ to (3).

However, to make sure that the block-matrix operations will be executed efficiently we have to add a few functions to the matrix class. These include retrieving a single column from the matrix and solving triangular equations with this column or a unit vector as a right hand side. These are crucial for an efficient implementation of the linear algebraic operations for block-angular matrices, such as (8) or (14). Such operations are never accessed directly by the interior-point algorithm; however, they can be invoked by the matrix class that corresponds to an ancestor of a given matrix in the tree describing the block structure of the constraint matrix. We added these operations to $\mathcal{F}_{\mathcal{X}}$.

Suppose a class `Vector` that describes vectors associated with block-structured matrices has already been defined. We can define an *abstract* (or *virtual*) class `Matrix` that supports all the operations of $\mathcal{F}_{\mathcal{X}}$. It is an abstract class because its methods are not defined, leaving extended classes to provide the missing pieces:

```
abstract class Matrix {

    abstract void    ComputeAThetaAt (Vector theta);
    abstract void    Factorize ();
    abstract void    SolveAThetaAt (Vector x, Vector y);
    abstract void    SolveTriang (Vector x, Vector y);
    abstract void    SolveTransTriang (Vector x, Vector y);
```

```
    abstract void    MatrixVectProd (Vector x, Vector y);
    abstract void    MatrixTransVectProd (Vector x, Vector y);
    abstract void    GetColumnofA (Vector x, int j);
    abstract void    SolveTriangAj (Vector x, int j);
    abstract void    SolveTriangUnitVect (Vector x, int i);
    abstract Matrix ();
}
```

We can extend the `Matrix` class with *concrete* and *final* classes for each kind of block-structured matrix we want to support. Concrete and final classes do not contain any abstract method and will not be extended by another class. Although this construction is simple, it has proved to be quite powerful.

We take advantage of the *polymorphism* of the object class `Matrix` (it can represent many different objects which share the same methods) in the classes that implement a particular kind of block structure. For instance, the class `PrimalBlockAngular` can be defined as:

```
class PrimalBlockAngular extends  Matrix {

    private int NumOfBlocks;
    private Matrix[] DiagonalBlocks;
    private Matrix[] BorderedBlocks;
    ...
}
```

Polymorphism of the class `Matrix` implies that we do not make any assumptions about the classes that handle blocks of a structured matrix. `DiagBlocks` and `BorderedBlocks` correspond to blocks A_i and B_i of (4), respectively. Since by definition $\mathcal{F}_\mathcal{X}$ contains all the necessary operations to implement a block-structured operation, we know that we can include `Matrix` objects in the array elements of `PrimalBlockAngular` class. The class `Matrix` is self-referencing: it is itself a `Matrix` object and refers to other `Matrix` objects.

It is worth making a simple comparison of the set of operations $\mathcal{F}_\mathcal{X}$ in `Matrix` with the usual implementation of the set (3) in any interior-point code. For the latter, there is essentially only one way to define the problem: the LP constraint matrix is represented as a general sparse matrix. With `Matrix`, on the other hand, the user may choose among different constructors. The user may decide if he wants to process a structured matrix as a simple sparse matrix or with more dedicated treatment. Clearly, how the problem is defined and what classes are chosen will determine the way $\mathcal{F}_\mathcal{X}$ operations will be computed.

In many cases, we can take advantage of the flexibility and efficiency that result from the design of the `Matrix` class. For instance, we have developed three simple classes for network, identity and projection matrices by making minor modifications to the sparse class: the matrix-vector products were specialized for these kinds of matrices. The development of these specialized matrix classes is straightforward. They offer nonnegligible savings in storage and efficiency compared with the general sparse matrices: the network adjacency matrix has all elements equal to 1 and -1; the identity matrix does not have to be stored at all; the projection P_i onto the $n-1$ dimensional subspace of $\mathcal{R}^n$ in which the i-th component of the vector vanishes needs

storage of only one index. Multiplications of such matrices or their transpositions with vectors can achieve important savings by avoiding multiplications by ones. In the examples described below the network matrices were used as diagonal blocks in the dual block-angular factorization, in which a large amount of time is spent in matrix-vector products computed for the diagonal blocks.

5.2 Parallel Implementation

Our parallel implementation uses the standard MPI library and was run on a variety of computers: IBM SP2, cluster of Linux PCs, multiprocessor SUN HPC. For the cluster of PCs running Linux we used the MPICH implementation [17] of Argonne National Laboratory and Mississippi State University.

In the primal and the dual block-angular matrix there is substantial room for parallel computation. Most computations can be done by distributing the blocks among different processors.

A succinct sequential implementation is a good step toward a parallel one, especially in the context of message passing parallelism where the code tends to be quite verbose. Here, the polymorphism of `Matrix` is useful: if we introduce a `VoidMatrix` in which the methods do nothing, we can avoid excessive processor identity testing. Code which looks like

```
if (myid == BlockId) {
        /* Compute something with Matrix Block held
        by the processor BlockId. Otherwise do nothing. */
        ...
}
```

can be often advantageously replaced by the non testing version where the distinctions are made based on the `Matrix`. Only one processor loads a block whereas the others load an empty `VoidMatrix`.

The parallelism affects only the abstract classes that contain reference to the `Matrix` class. No attempt was made to implement elementary matrices (sparse, dense, network, identity, projection) in parallel.

6 Examples of Structured Optimization Problems

There are numerous sources of structured linear programs. Although we have developed a general structure exploiting IPM solver, the applications described in this paper are all variations of optimization problems arising in the design of telecommunication networks [23]. This choice is motivated by our knowledge of these types of problems and our experience in solving them with decomposition approaches [15, 25].

Multicommodity network flows [1] are relevant to various sectors of the economy, e.g., transportation, telecommunications, etc. In these problems, commodities in a network compete for a

common capacity. Such problems are by nature structured and they are often solved by decomposition, see e.g., [1, 15, 20]. Multicommodity flow problems can also be solved with a specialized interior-point method using an iterative solver to compute approximate Newton directions, as for example in the approach of [8] that employs the preconditioned conjugate gradient method.

In networks prone to failure, the problem is to find the best way to invest in additional arc capacity, so that in case of any breakdown in one arc, it is still possible to ship all commodities [25]. The problem also can be viewed as a two stage stochastic programming problem, with an investment decision in the first stage and routing decisions in the second stage. There exist specialized implementations of interior-point methods for two stage stochastic problems [6, 19] that use the Sherman-Morrison-Woodbury formula to take advantage of the dual block-angular structure of the LP constraint matrix.

We will solve three different classes of network optimization problems:

- Multicommodity flow problem,
- Multicommodity flow survivable network design,
- Single-Commodity flow survivable network design.

These problems are well described in the optimization literature so we give only a brief presentation of them. The reader interested in a detailed discussion of the formulation of these problems may consult [23, 25].

6.1 Multicommodity Flow Problem

Let a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be given where $\mathcal{V}$ denotes the set of its nodes and $\mathcal{E} \subset \{(i,j) : i \in \mathcal{V}, j \in \mathcal{V}, i \neq j\}$ denotes the set of its arcs. With every arc (i,j) we associate *cost* $c_{ij} \geq 0$ of shipment per unit flow through this arc and *capacity* $K_{ij} \geq 0$. Assume some *demands* $d^{(k)}, k \in \mathcal{D}$, are given. Every demand specifies a flow of a different commodity to be shipped from a *source* or to a *target*. Let vector $x = (x_{(i,j)})_{(i,j)\in\mathcal{E}} \in \mathcal{R}^{|\mathcal{E}|}$ define a flow in the network. A *feasible* flow satisfies the flow balance equation at every node i:

$$\sum_{(i,j)\in\mathcal{E}} x_{ij} - \sum_{(l,i)\in\mathcal{E}} x_{li} \;=\; d_i,$$

where d_i is a supply/demand of node i. For supply nodes $d_i > 0$, for demand nodes $d_i < 0$, and for transshipment nodes $d_i = 0$. Let N be the node-arc incidence matrix of $\mathcal{G}$. A feasible flow through a network that satisfies demand k is described by the following equation

$$Nx^{(k)} = d^{(k)}.$$

For a simple demand, vector $d^{(k)}$ has only two nonzero entries. Demands can be grouped, however, leading to a more compact formulation of the optimization problem [20]; vector $d^{(k)}$ then has several nonzero entries.

Total flow through an arc (i,j) is the sum of all single commodity flows in the arc. This cannot exceed the arc capacity:

$$\sum_{k\in\mathcal{D}} x_{ij}^{(k)} \leq K_{ij}.$$

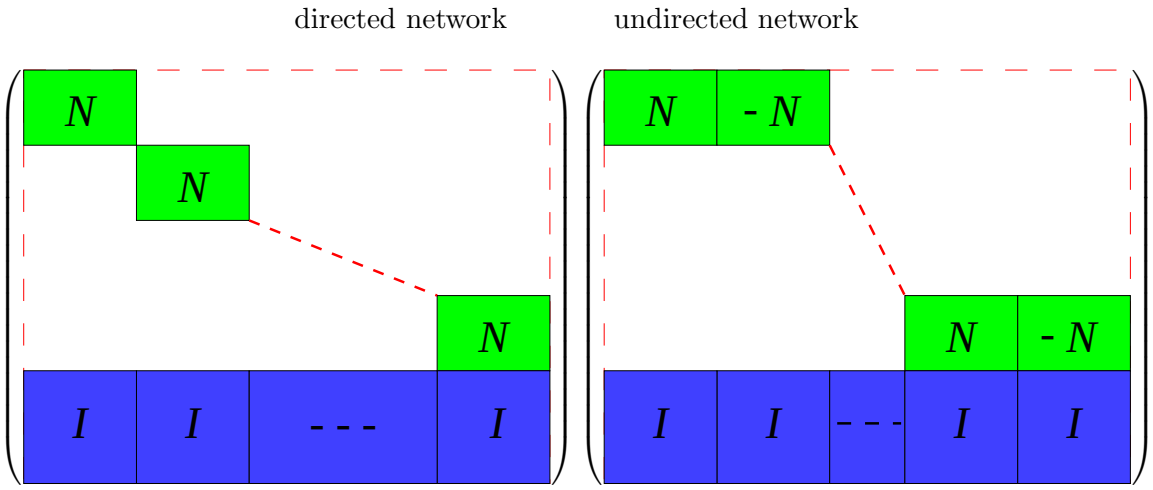

Figure 3: Structure of the minimum cost network flow problem.

Hence the minimum cost multicommodity network flow problem is the following linear optimization problem:

$$\begin{aligned} \min \quad & \sum_{(i,j)\in\mathcal{E}} c_{ij} \sum_{k\in\mathcal{D}} x_{ij}^{(k)} \\ \text{s. t.} \quad & \sum_{k\in\mathcal{D}} x_{ij}^{(k)} \le K_{ij}, && \forall (i,j)\in\mathcal{E}, \\ & N x^{(k)} = d^{(k)}, && \forall k\in\mathcal{D}, \\ & x^{(k)} \ge 0, && \forall k\in\mathcal{D}. \end{aligned} \tag{15}$$

This can be a very large linear program. Its constraint matrix has the primal block-angular structure displayed in Figure 3. Incidence matrices N are repeated for every demand and identity matrices appear in a linking constraint (arc capacity constraint). The formulation of the minimum cost network flow problem for an undirected network needs every arc to be replicated (right figure), which further increases the problem size but does not change its structure.

6.2 Multicommodity Flow Survivable Network Design

In real telecommunnication networks failures can occur and when they do customers expect that their interrupted traffic will be swiftly restored. Broadly speaking, *survivability problems* aim to ensure adequate performance of the network when some components fail. Following [25] we will say that a network is *survivable* if, for any elementary failure (i.e., removal of a single edge or node), there is a way, using some existing capacity, to rearrange the traffic assignment to meet all demands. A necessary condition for the network to be survivable is its bi-connectivity [1]. However, this topological condition is not sufficient. An additional condition is needed to ensure that there is enough capacity to reroute traffic affected by a given failure.

Consider the example network given in Figure 4 and assume that there are two commodities flowing in it: from node a to node f, 3 units pass through the edges (a,h), (h,g), (g,k) and (k,f), and from node c to node d, 5 units pass through the edges (c,h), (h,g), (g,e) and (e,d). Assume the common edge (h,g) breaks down. There are two approaches to rerouting the affected

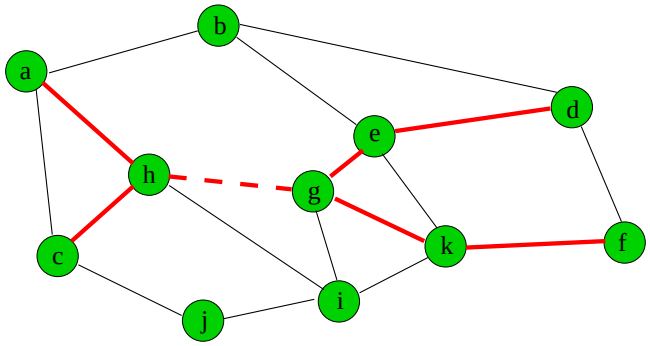

Figure 4: Arc failure in the network.

traffic.

In the *local rerouting approach* we will have one demand: send $5 + 3 = 8$ units between the endpoints h and g of the broken edge. In the *global rerouting approach* both affected routings would be considered separately. Thus we would have two demands: send 3 units between a and f, and 5 units between c and d.

The goal of the network design is to enforce the connectivity and minimise the overall cost of installing capacity.

Let an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be given and let $\mathcal{S}$ be the set of states resulting from a failure. A state of the network is characterized by an elementary failure and the rerouting demands that result from this failure. There are $|\mathcal{V}| + |\mathcal{E}|$ elementary failures of single nodes and arcs. If state s represents the failure of the node $v \in \mathcal{V}$, then $\mathcal{G}(s) = (\mathcal{V}(s), \mathcal{E}(s))$ is the graph $(\mathcal{V} \setminus v, \mathcal{E} \setminus \mathcal{E}_v)$, where $\mathcal{E}_v$ is the set of arcs adjacent to node v. If states s represents the failure of the arc $a \in \mathcal{E}$, then $\mathcal{G}(s) = (\mathcal{V}(s), \mathcal{E}(s))$ is the graph $(\mathcal{V}, \mathcal{E} \setminus a)$. For each state s, a set of demands indexed by $k_s \in \mathcal{R}_s$ between pairs of nodes should be met. In a case of node failure any demand to send a flow to or from the broken node has to be canceled. Let $K^{(s)} = (K^{(s)}_{(i,j)})_{(i,j)\in\mathcal{E}(s)}$ denote the residual capacity of the network. Assume that we install an additional capacity $y = (y_{(i,j)})_{(i,j)\in\mathcal{E}}$ with cost $c = (c_{(i,j)})_{(i,j)\in\mathcal{E}}$. In the design of a survivable network, we look for the additional capacity of least cost that allows the rerouting of the demands $k_s \in R_s$ for each state $s \in \mathcal{S}$.

Let $N(s)$ denote the incidence matrix for a graph corresponding to failure s and let $x^{(k_s)}$ denote a feasible flow in $\mathcal{G}(s)$ for demand $k_s \in \mathcal{R}_s$ in state $s \in \mathcal{S}$. Variable $x^{(k_s)} = (x^{(k_s)}_{i,j})_{(i,j)\in\mathcal{E}(s)}$ represents a flow that satisfies demand $d^{(k_s)}$.

Hence the multicommodity survivable network design problem is the following linear program

$$
\begin{aligned}
\min \quad & c^T y \\
\text{s. t.} \quad & \sum_{k_s \in \mathcal{R}_s} x^{(k_s)}_{ij} \le K^{(s)}_{ij} + y_{ij}, && \forall s \in \mathcal{S},\ \forall (i,j) \in \mathcal{E}(s), \\
& N(s)\, x^{(k_s)} = d^{(k_s)}, && \forall s \in \mathcal{S},\ \forall k_s \in \mathcal{R}_s, \\
& x^{(k_s)} \ge 0, && \forall s \in \mathcal{S},\ \forall k_s \in \mathcal{R}_s, \\
& 0 \le y \le \bar{y}, &&
\end{aligned}
\tag{16}
$$

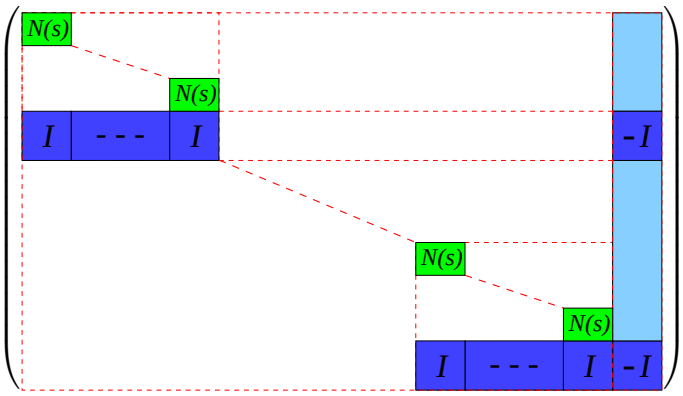

Figure 5: Structure of the multicommodity survivable network design problem.

This is again possibly a very large linear optimization problem. Its special structure is displayed in Figure 5. The structure is a nesting of the primal and dual block-angular structures. Every failure s induces a new network and associated multicommodity flow problem of rerouting the traffic affected by this failure. Incidence matrices $N(s)$ are repeated for every demand corresponding to the traffic that has to be rerouted, and identity matrices appear in linking constraints (arc capacity constraints). The new capacity installed in the network y corresponds to the block of columns that link all otherwise independent blocks. Thus the LP constraint matrix has the dual block-angular structure. Each diagonal block corresponds to a multicommodity flow problem and has primal block-angular structure.

6.3 Single-Commodity Flow Survivable Network Design

We shall now consider the problem of joint optimal investment in the base and spare capacities of the network. The capacity of an arc (i, j) is split into y_{ij} and z_{ij}. The base capacity $y = (y_{(i,j)})_{(i,j)\in\mathcal{E}}$ is used in the normal state of the network to satisfy the demands of the customers; the spare capacity $z = (z_{(i,j)})_{(i,j)\in\mathcal{E}}$ can be used for rerouting the traffic affected by any failure. Installation of base and spare capacity y_{ij} and z_{ij} on arc (i, j) costs c_{ij} and f_{ij} per unit, respectively.

Let N denote the incidence matrix of the original network and $N(s)$ denote the incidence matrix for a graph corresponding to failure $s \in \mathcal{S}$. Let $x_{ij}^{(k)}$ denote a feasible flow between the source and the target node, for a demand $k \in \mathcal{D}$. If arc s fails, the total flow through this arc creates a new demand $k_s \in \mathcal{R}_s$ to be satisfied through the residual and spare network. Note that since we deal with a local rerouting case, we have a single commodity: a new demand between the extremities of the broken arc. Let $x^{(k_s)} = (x_{i,j}^{(k_s)})_{(i,j)\in\mathcal{E}(s)}$ denote the flow in $\mathcal{G}(s)$ for a demand $k_s \in \mathcal{R}_s$ in state $s \in \mathcal{S}$. This flow can use the capacity of the spare network z.

Hence the problem of joint optimal synthesis of base and spare capacity in the network is the

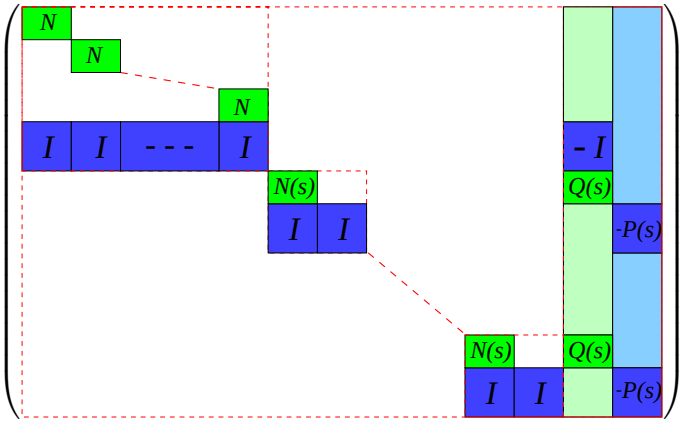

Figure 6: Structure of the single-commodity flow survivable network design problem.

following linear optimization problem

$$
\begin{array}{lll}
\min & c^T y + f^T z & \\
\text{s. t.} & \sum\limits_{k \in \mathcal{D}} x_{ij}^{(k)} \le y_{ij}, & \forall (i,j) \in \mathcal{E}, \\
& N\, x_0^{(k)} = d^{(k)}, & \forall k \in \mathcal{D}, \\
& \sum\limits_{k_s \in \mathcal{R}_s} x_{ij}^{(k_s)} \le z_{ij}, & \forall s \in \mathcal{S},\ \forall (i,j) \in \mathcal{E}(s), \\
& N(s)\, x^{(k_s)} = -Q(s) y, & \forall s \in \mathcal{S},\ \forall k_s \in \mathcal{R}_s, \\
& x_0^{(k)} \ge 0, & \forall k \in \mathcal{D}, \\
& x^{(k_s)} \ge 0, & \forall s \in \mathcal{S},\ \forall k_s \in \mathcal{R}_s, \\
& 0 \le y \le \bar{y}, & \\
& z \ge 0. &
\end{array}
\tag{17}
$$

The complicated special structure of this linear program is displayed in Figure 6. The first top level diagonal block corresponds to the multicommodity flow in a base network. The following diagonal blocks correspond to arc failures (we omit node failures in this figure). Since the flow through a broken arc is aggregated (local rerouting), there is only one commodity in each of these blocks and, consequently, only one incidence matrix $N(s)$. The last two column blocks correspond to linking variables y and z, respectively. The capacity y is used by the base network. In case of failure, the broken arc a induces a demand through matrix $Q(s)$ that has only one nonzero column with exactly two nonzero entries in rows corresponding to the extremities of arc a: $d^{(k_s)} = -Q(s)y$. Spare capacity z is available to reroute flows in all cases of failures. Matrices $P(s)$ link the single-commodity flows that use the same spare capacity. For an elementary failure of arc a, $P(s) \in \mathcal{R}^{(|\mathcal{E}|-1)\times|\mathcal{E}|}$ is a projection matrix obtained from the identity matrix by removing the row corresponding to arc a. Note that in the more complicated case of node failure (that is not displayed in this figure) several arcs would be removed from the network. Blocks corresponding to single-commodity flows could then be replaced with blocks describing multicommodity flows. The projection matrices $P(s)$ would then be obtained from identity matrices by removing all rows corresponding to the removed arcs.

Problem	Nodes	Arcs	Demands
RealNet	119	308	7021
Random6	100	300	200
Random12	300	600	1000
Random16	300	1000	1000
Random20	400	1000	5000

Table 1: Minimum Cost Network Flow Problems: Network Statistics.

Problem	Rows	Columns	Iters	Time
RealNet	14232	72996	31	77.3
Random6	8715	51300	20	27.6
Random12	88506	353400	40	933
Random16	87710	581000	39	2333
Random20	160201	799000	63	3899

Table 2: Minimum Cost Network Flow Problems: Solution Statistics.

The resulting LP constraint matrix has a dual block-angular form. Diagonal blocks correspond to multicommodity or single-commodity flows and each of these has a primal block-angular structure. The special structure is worth exploiting only for the first large block. The single-commodity blocks are too small and are treated as general sparse matrices.

7 Numerical Results

We shall discuss in this section the computational results that illustrate the efficiency of our structure exploiting interior-point code. We call it OOPS which stands for Object-Oriented Parallel interior-point Solver. Although we have already applied it to solve a number of different structured large scale optimization problems, in this paper we restrict ourselves to problems that arise in network optimization. We demonstrate the efficiency of the code on multicommodity network flow problems and the survivable network design problems presented in Section 6 and described in more detail in [25].

The first class of problems—minimum cost multicommodity network flows—has attracted a lot of attention in the mathematical programming community due to its numerous applications. Many very efficient algorithms for this class of problems exist [1] including interior-point implementations, e.g., [8]. These specialized methods usually exploit network properties better than a general linear programming approach (like ours). Thus we do not expect to offer a really competitive alternative approach for this class of problems; we use it only as an easy illustration of a well known structured problem.

Unlike the minimum cost multicommodity network flow problems, the problems of survivable network design have a more complicated structure, though they are often not so large. These problems can take full advantage of the structure exploiting methods described in this paper.

Problem	Basic data			Failure data	
	Nodes	Arcs	Routes	Failures	CondDems
PB1	30	57	150	81	1110
PB2	37	71	300	102	3266
PB3	40	77	200	109	1942
PB4	45	87	300	123	3658
PB5	65	127	400	179	7234

Table 3: Multicommodity Flow Survivable Network Design: Network Statistics.

Problem	Size		OOPS		CPLEX 6.0	
	Rows	Columns	Iters	Time	Iters	Time
PB1	22213	72514	25	122	23	143
PB2	59021	207901	34	518	35	730
PB3	54657	188266	29	407	25	518
PB4	83561	294735	33	735	31	1131
PB5	242570	886178	48	3956	34	-

Table 4: Multicommodity Flow Survivable Network Design: Solution Statistics.

7.1 Efficiency of the Sequential Code

The solver was run on a 400 MHz Ultrasparc II Sun HPC computer using Solaris 7 and on a Linux PC with a 300 MHz Pentium Pro processor. In this section we report results of sequential runs. The efficiency of the parallel code is discussed in the following section.

We start with the presentation of numerical results obtained for the multicommodity network flow problems (15). In Table 1 we report the characteristics of the networks: the numbers of nodes $|\mathcal{V}|$, arcs $|\mathcal{E}|$ and demands $|\mathcal{D}|$, respectively. In Table 2 we report the sizes of the resulting linear programs and the solution statistics: the numbers of iterations and the CPU time in seconds on the Sun HPC. Although we grouped the demands to keep the sizes of the problems as small as possible, some of them remained large and reached several hundred thousand variables. It is worth noting that the number of interior-point iterations shows moderate growth, reaching 63 on the largest problem with nearly 800,000 variables.

The multicommodity flow survivable network design problem (16) displays a nested block sparsity pattern. Its constraint matrix has dual block-angular structure. Each diagonal block in it is a multicommodity flow problem for some reduced network. We report in Table 3 several network characteristics for the test problems: graph sizes and data for original routings (Routes), the number of failures and the number of conditional demands created by them. The latter is a major factor that determines the overall size of the LP formulation (16) of this problem. The sizes of several test problems and the solution statistics for two solvers, our structure exploiting code and CPLEX 6.0 barrier code, are reported in Table 4. To facilitate the comparison we ran both solvers on the same Linux PC with 300 MHz Pentium Pro processor and 384 MB of RAM. Our IPM solver was compiled with the GNU C and FORTRAN compilers `gcc` and `g77`; options `-O3 -mpentiumpro` were used. All solution times were measured with the Linux `time` command and are given in seconds.

Problem	Nodes	Arcs	Demands
T1	12	25	66
T2	26	42	264
T3	53	79	1378
P1	25	41	300
P2	35	58	595
P3	45	91	990

Table 5: Single-commodity Flow Survivable Network Design: Network Statistics.

Problem	Size		OOPS		CPLEX 6.0 Barrier		CPLEX 6.0 Simplex	
	Rows	Columns	Iters	Time	Iters	Time	Iters	Time
T1	1021	2400	15	1.7	20	1.65	1577	2.0
T2	3414	7266	23	10.6	21	-	2852	6.8
T3	13053	26860	25	49.7	22	69.1	9112	68.0
P1	3241	6970	28	10.8	25	7.2	2474	5.2
P2	6492	13978	28	26.2	23	22.1	7829	46.3
P3	14221	32760	49	138.2	54	226.9	42520	867.0

Table 6: Single-commodity Flow Survivable Network Design: Solution Statistics.

The analysis of results collected in Table 4 reveals that the structure exploiting code is slightly faster than CPLEX on these problems. Both codes show slow increase of the number of iterations with the size of the problem. An empty space in Time column for problem PB5 indicates the case where CPLEX stopped due to numerical errors before the optimal solution was reached.

Finally, we analyse the efficiency of the new solver on the single-commodity flow survivable network design problem (17). This problem has a less regular structure, with one multicommodity block and many small single-commodity blocks. Network characteristics of the test examples are given in Table 5 and the sizes of the corresponding linear programs are given in Table 6. Since the sizes of these problems vary from small to medium we solved them also with the CPLEX default (primal) simplex method. The results for three solvers (our IPM code, CPLEX 6.0 Barrier and CPLEX 6.0 Simplex) are collected in Table 6. There is no clear winner: the simplex method is best on smaller instances and both IPM codes show a uniform behaviour, less sensitive to the problem size. CPLEX Barrier did not satisfy optimality conditions for problem T2.

The above analysis of the results justifies the conclusion that the new structure exploiting solver is flexible enough to deal with complicated nested structures and when run as a sequential code reaches the efficiency comparable to a high-quality general-purpose commercial LP solver. In the following section we discuss the ability of the new structure exploiting solver to take advantage of parallelism.

Procs	Random16		Random20		PB4		PB5	
	Time	S-up	Time	S-up	Time	S-up	Time	S-up
1	2333	1.0	3899	1.0	661	1.0	3256	1.0
2	1169	1.99	1947	2.0	361	1.83	1788	1.82
4	612	3.81	1003	3.88	194	3.41	1041	3.13
8	361	6.46	538	7.25	106	6.24	539	6.04
16	224	10.42	333	11.71	62	10.66	324	10.05

Table 7: Parallel Efficiency.

7.2 Efficiency of the Parallel Code

The structure exploiting solver has embedded routines that allow it to be run in parallel. The majority of block-matrix operations can take advantage of parallelism. This has been exploited in interior-point codes such as those documented in [16, 19, 26]. In our solver, we have parallelized only the higher level matrix classes, expecting that this will give the best granularity and will achieve good speed-ups.

In Table 7 we report results of parallel runs on an 18 processor Sun HPC 3500 system. This computer has 18 400MHz UltraSPARC II processors and 18 GB of shared memory. We have chosen the two largest test examples from each of the first two problem classes, run them on 2^k processors with $k = 0, 1, 2, 3$ and 4, and measured the parallel efficiency. The single-commodity network design problems are small so we have not run them in parallel. The data in Table 7 are self-explanatory: for every problem we report the CPU time and the speed-up (the ratio of the solution time on 1 processor and the solution time on 2^k processors).

The minimum cost multicommodity network flow problems Random16 and Random20 as well as the multicommodity flow survivable network design problems PB4 and PB5 are large and display regular structures (cf. (15) and (16) and Figures 3 and 5, respectively). The parallel implementation benefits from these features and reaches speed-ups of about 10-12 on 16 processors.

8 Conclusions

We have discussed the design and implementation of our structure exploiting interior-point solver and described several applications for which it can be used. Its object-oriented implementation uses inclusion polymorphism. We defined an abstract matrix class with a set of virtual functions that provide all the necessary linear algebraic operations for an interior-point algorithm and allow self-referencing as well. By these means we can model *any* nested structure in the linear optimization problem.

Linear algebraic operations for block-structured matrices are excellent candidates to take advantage of parallel computations [7]. Thus our solver has been designed to run in parallel. We implemented it using the message passing model of parallelism and the MPI library [17]. Our choice is motivated by the portability of the code to many different computing platforms. The solver has already been run on a Sun HPC, an IBM SP2 and a cluster of Linux PCs linked with

Ethernet as in the Beowulf project [24].

Since block-structured matrices result naturally from the modeling of real-life optimization problems, we expect the solver to be able to tackle many practical problems that are challenging for general purpose optimization software. In this paper we have focused on problems arising in network design. However, the solver has already been used to solve other structured problems and we shall report soon on this experience.

The solution of very large problems requires flexible model generation and management as well as a fast connection of the solver with the generation tool. The concept of the Structure Exploiting Tool (SET) [14] offers a practical approach to link a structure exploiting solver like the one discussed in this paper and algebraic modeling languages.

Finally, structure exploitation should improve the performance of the solver. With this respect we have shown that, run on a single processor, the new solver has an efficiency comparable to a general purpose commercial LP code. When run in parallel the new solver achieves speed-ups of 3.1-3.9 on 4 processors, 6.0-7.3 on 8 processors and 10-11.7 on 16 processors.

Acknowledgements
We are grateful to Ken McKinnon for his careful reading of the paper and his insightful comments, and to Andreas Grothey for help in running the parallel version of the solver. We are also grateful to the anonymous referee for constructive comments, resulting in an improved presentation.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows*, Prentice-Hall, New York, 1993.

[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] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language, Third Edition*, Addison-Wesley, New York, 2000.

[4] J. F. Benders, *Partitioning procedures for solving mixed-variables programming problems*, Numerische Mathematik, 4 (1962), pp. 238–252.

[5] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel an Distributed Computations*, Prentice-Hall, Englewood Cliffs, 1989.

[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] Å. Björck, *Numerical Methods for Least Squares Problems*, SIAM, Philadelphia, 1996.

[8] J. Castro, *A specialized interior-point algorithm for multicommodity network flows*, SIAM Journal on Optimization, 10 (2000), pp. 852–877.

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

[10] G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, 1963.

[11] G. B. Dantzig and P. Wolfe, *The decomposition algorithm for linear programming*, Econometrica, 29 (1961), pp. 767–778.

[12] M. C. Ferris and D. J. Horn, *Partitioning mathematical programs for parallel solution*, Mathematical Programming, 80 (1998), pp. 35–62.

[13] R. Fourer, *Staircase matrices and systems*, SIAM Review, 26 (1983), pp. 1–70.

[14] E. Fragnière, J. Gondzio, R. Sarkissian, and J.-P. Vial, *Structure exploiting tool in algebraic modeling languages*, Management Science, 46 (2000), pp. 1145–1158.

[15] J. Gondzio, R. Sarkissian, and J.-P. Vial, *Using an interior point method for the master problem in a decomposition approach*, European Journal of Operational Research, 101 (1997), pp. 577–587.

[16] 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.

[17] W. Gropp and E. Lusk, *User's guide to MPICH, a portable implementation of MPI*, Tech. Report ANL/MCS-TM-ANL-96/6, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, USA, 1996.

[18] 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.

[19] 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.

[20] K. L. Jones, I. J. Lustig, J. M. Farvolden, and W. B. Powell, *Multicommodity network flows: the impact of formulation on decomposition*, Mathematical Programming, 62 (1993), pp. 95–117.

[21] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partioning graphs*, The Bell System Technical Journal, 29 (1970), pp. 291–307.

[22] I. J. Lustig, R. E. Marsten, and D. F. Shanno, *Interior point methods for linear programming: computational state of the art*, ORSA Journal on Computing, 6 (1994), pp. 1–14.

[23] M. Minoux, *Network synthesis and optimum network design problems: Models, solution methods and applications*, Networks, 19 (1989), pp. 313–360.

[24] D. Ridge, D. Becker, P. Merkey, and T. Sterling, *Beowulf: Harnessing the power of parallelism in a pile-of-PCs*. Proceedings, IEEE Aerospace, 1997.

[25] R. Sarkissian, *Telecommunications Networks: Routing and Survivability Optimization Using a Central Cutting Plane Method*, PhD thesis, École Polytechnique Fédérale de Lausanne, CH-1205 Ecublens, November 1997.

[26] G. Schultz and R. R. Meyer, *An interior point method for block angular optimization*, SIAM Journal on Optimization, 1 (1991), pp. 583–602.

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

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