Comp. Appl. Math., V. 12, no. 3, pp.199-226, 1993
© Sociedade Brasileira de Matemática Aplicada e Computacional, printed in U.S.A.

ACTIVE SET METHODS FOR PROBLEMS IN COLUMN BLOCK ANGULAR FORM

J.M. STERN[1] and S.A. VAVASIS[2]

[1]Departamento de Ciência da Computação
Universidade de São Paulo
Cx.P. 20570, CEP 01498
São Paulo, Brasil
and
[2]Computer Science Department
Cornell University
Ithaca, NY 14853, USA

ABSTRACT: *We study active set methods for optimization problems in Block Angular Form (BAF). We begin by reviewing some standard basis factorizations, including Saunders' orthogonal factorization and updates for the simplex method that do not impose any restriction on the pivot sequence and maintain the basis factorization structured in BAF throughout the algorithm. We then suggest orthogonal factorization and updating procedures that allow coarse grain parallelization, pivot updates local to the affected blocks, and independent block reinversion. A simple parallel environment appropriate to the description and complexity analysis of these procedures is defined in Section 5. The factorization and updating procedures are presented in Sections 6 and 7. Our update procedure outperforms conventional updating procedures even in a purely sequential environment.*

RESUMO: *MÉTODO DE CONJUNTO ATIVO PARA PROBLEMAS EM FORMA ANGULAR DE BLOCOS. Fatorizações básicas são revistas, incluindo a fatorização ortogonal de Saunders. Sugerem-se então fatorizações ortogonais e procedimentos de atualização em ambiente de paralelismo.*

PALAVRAS-CHAVE: Programação linear, método simplex.

[1] Supported by Fundação de Amparo à Pesquisa do Estado de São Paulo grant 87-0540-9. Part of this work was done while the author was visiting Argonne National Laboratory for the Summer Institute in Parallel Programming 1990. Part of this work was also supported by the School of Operations Research and Industrial Engineering at Cornell University.

[2] Part of this work was done while the author was visiting Sandia National Laboratories, Albuquerque, New Mexico, supported by the U.S. Department of Energy under contract DE-AC04-76DP00789. Part of this work was also supported by the Applied Mathematical Sciences Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under grant DE-FG02-86ER25013.A000 and in part by the National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research, through NSF grant DMS 8920550.

#278/92. Received: 20/VII/92. Accepted: 30/VI/93.

1. ACTIVE SET METHODS

Consider the linear program

$$LP : \min\{fx : x \geq 0 \text{ and } Ax = d\}.$$

If A is $m \times n$, a (nondegenerate) vertex of the feasible region has $n-m$ active inequalities constraints, i.e. $n-m$ variables are set to 0. These are the residual variables for that vertex. Permuting the vector x to separate its basic, i.e. nonzero, and residual entries, and partitioning f and A accordingly, we can write LP as:

$$\min \begin{bmatrix} f^b & f^r \end{bmatrix} \begin{bmatrix} x^b \\ x^r \end{bmatrix}, \; x \geq 0 \mid \begin{bmatrix} B & R \end{bmatrix} \begin{bmatrix} x^b \\ x^r \end{bmatrix} = d.$$

Using the basis inverse, B^{-1}, we can isolate the basic variables of this vertex

$$x^b = \tilde{d} - \tilde{R}x^r \equiv B^{-1}d - B^{-1}Rx^r,$$

and the value of the objective function at this vertex is

$$\phi = \zeta - zx^r \equiv f^b\tilde{d} + (f^r - f^b\tilde{R})x^r\,.$$

If we make a single element of x^r positive, $x^r_j > 0$, the value of x^b, the basic solution, becomes

$$x^b = \tilde{d} - \tilde{R}x^r = \tilde{d} - x^r_j\tilde{R}_{\bullet,j}$$

and remains feasible if nonnegative. This suggests the *simplex method* for going from a feasible vertex to a better feasible vertex. In one step of the simplex we:

- Look for a residual index j such that $z_j < 0$.
- Compute $i = \text{Argmin}_{k \mid \tilde{R}_{j,k} > 0} \{\tilde{d}_k / \tilde{R}_{k,j}\}$.
- Make variable x^r_j basic, and x^b_i residual.
- Compute the new basis inverse.

The simplex method cannot proceed if $z > 0$ in the first step, or if it takes the minimum of an empty set in the second step. The second case corresponds to an unbounded LP, and in the first case the current vertex is an optimal solution. Swapping the basic/residual status of a pair of variables is called (to) *pivot*. At every pivot we have to update $\tilde{R}$, i.e. recompute the fundamental basis $\begin{bmatrix} I & -\tilde{R}^t \end{bmatrix}$ of the null space of $[R\ B]$. Historically the first version of the simplex algorithm, the tableaux simplex, did exactly that, updating the tableaux matrix $\begin{bmatrix} I & \tilde{R} & \tilde{d} \end{bmatrix}$ at every pivot. But we really do not need to carry the fundamental null space basis explicitly. It suffices to have a factorization of B that allows us to compute the one column of $\tilde{R}$, i.e. the one fundamental feasible direction that we need at every iteration: see [46]. This is the revised simplex method of [13].

We can generalize this simple strategy to problems with nonlinear objective functions, see [3] and [29], or even to problems with nonlinear constraints, see [31] and [8]. These are called active set methods (ASMs) and they all need, in explicit or implicit form, the fundamental null space basis of $[R \ B]$. The best form in which to maintain and update the fundamental null space basis is highly dependent on the form and structure of the problem. In this paper we examine this question for problems whose constraint matrix, A, is in column block angular form. Minor variations of all our considerations and algorithms apply to problems in row block angular form.

2. THE COLUMN BLOCK ANGULAR FORM

Optimization problems in BAF are very common in practice, like multiperiod control, scenario investment planning, stochastic programming, truss or circuit optimization, economic stabilization, etc. [2], [15], [19], [26], [27], [30], [38], [39], and [43].

A matrix in CBAF is a block matrix A, with $b \times (b+1)$ blocks, $A^{k,l}$, $1 \le k \le b$, $1 \le l \le b+1$, where only the diagonal blocks, $D^k = A^{k,k}$, and the (column) angular blocks, $E^k = A^{k,b+1}$, contain nonzero elements (NZE's). Block D^k has dimension $m(k) \times na(k)$, and block E^k has dimension $m(k) \times na(b+1)$.

We define the concatenations, shown in Figure 1,

$$A^{\bullet,k} = \left[A^{1,k}, \dots, A^{b,k}\right] \text{ for } 1 \le k \le b+1$$
$$A^{k,\bullet} = \left[A^{k,1}, \dots, A^{k,b+1}\right] \text{ for } 1 \le k \le b.$$

The element in row i and column j of the matrix D^k is $D^k_{i,j}$, and $D^k_{i,\bullet}$ and $D^k_{\bullet,j}$ are, respectively, the block's ith row and jth column. In the same way, $A^{\bullet,k}_{\bullet,j}$ and $A^{k,\bullet}_{i,\bullet}$ are the jth column and the ith row of $A^{\bullet,k}$ and $A^{k,\bullet}$, respectively.

In an ASM we always have a special square and nonsingular matrix of columns of A, the basis B. The nonbasic columns of A form the residual matrix, R. We always assume that the CBAF structure is maintained in B and R, as illustrated in Figure 1. If the kth diagonal block of B, B^k, has $n(k)$ columns of D^k, and the kth diagonal block of R has the remaining $da(k) = na(k) - n(k)$, we define $bp^k(1 \dots n(k))$ and $rp^k(1 \dots da(k))$ as the basic and residual column indices of D^K, in the order they appear, respectively, in B^k and in the kth diagonal block of R. So bp^k and rp^k, for $1 \le k \le b+1$, are a compete characterization of B and R: The diagonal blocks of the basis are $B^k = D^k(:, bp^k(1 : n(k)))$ and the corresponding angular blocks of the basis are $C^k = E^k(:, bp^{b+1}(1 : n(b+1)))$. In the same way, the diagonal and angular blocks of the residual matrix are $D^k(:, rp^k(1 : da(k)))$ and $E^k(:, rp^{b+1}(1 : da(b+1)))$. In the last equations the colon (:) is to be interpreted as an index range operator, $1 : n = [1, 2, \dots, n]$, and if A is $m \times n$, $A(:,:) = A(1 : m, 1 : n)$ [12].

We define $d(k) = m(k) - n(k)$, and since we assumed B to be nonsingular, $d(k) \ge 0$. Also, since B is square, $n(b+1) = \sum_1^b d(k)$.

3. BASIS FACTORIZATIONS

The basic numerical operation in an ASM is to compute and update the inverse, or

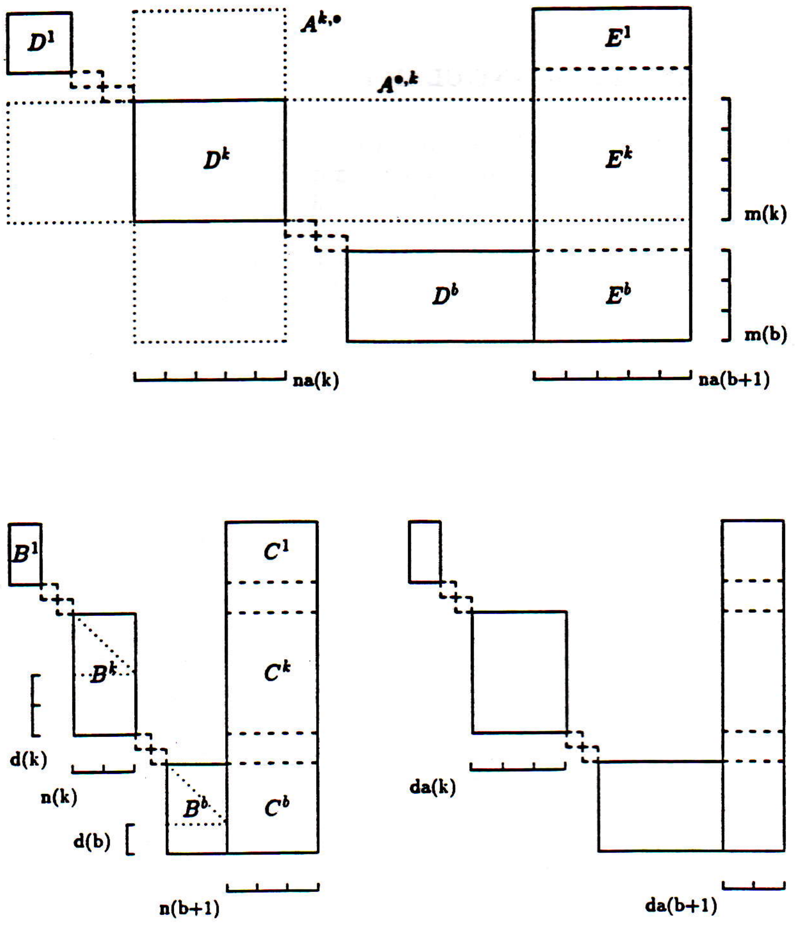

Figure 1. The Column Block Angular Form.

a factorization, of the current basis B. The factorization most commonly used is the

Gaussian, $LU = B$, that allows us to easily compute $B^{-1}x = U^{-1}(L^{-1}x)$, from the lower and upper triangular factors L and U. In fact, we usually have the LU factorization of QBP, where Q and P are (row and column) permutation matrices. We need or can use the permutations Q and P in order to:

- Maintain numerical stability.
- Preserve factors' sparsity.
- Preserve factors' structure.

Our main goal in this paper is to preserve the block structure in the factorization. We want to use the CBAF structure to parallelize independent "block operations" in our algorithms, as explained in the next sections. Preserving structure is also a first step to preserve sparsity, i.e., structure can be seen as a block scale or "macroscopic" sparsity: see [5] and [25]. Structure and sparsity are aspects of a combinatorial nature, whereas stability is an analytical one. Not surprisingly, the criteria for choosing P and Q that would optimize the combinatorial and the analytical properties of the factorization are conflicting. Let us examine this point more carefully:

In order to preserve the CBAF structure in the factorization, we shall restrict the choice of P, allowing column permutations only within each diagonal block, or within the angular columns, refer to Figure 1. So doing we can implicitly give the permutation P by the vectors bp^k, rp^k, $1 \leq k \leq b+1$, as they are defined in Section 2.

The row permutation Q will now divide each diagonal block in two:

- an *upper* square block, chosen to be nonsingular, to be factored.
- a *lower* rectangular block, of dimension $d(k) \times n(k)$.

As the ASM progresses, we have to *pivot*, i.e. we have to replace a column of B by a column of R, and then update the factorization. If we could guarantee that, at each diagonal block, the upper block remains nonsingular; then it would be easy to do these updates preserving the CBAF structure of the factors. Unfortunately, no such guarantee exists. In fact, in order to achieve numerical stability, we will need to permute upper and lower rows: see [20] and [44].

These permutations between upper and lower rows, in successive updates, lead to a degeneration of the CBAF structure in the factorization. Some strategies have been suggested to preserve block angular structures in the LU factorization, see [5] and [45], and they all have to forbid this type of permutations. Some remedies are suggested to preserve stability, nevertheless, some pivots cannot be handled. This is very inconvenient for, as explained in Section 1, we want the pivot sequence to be determined by the ASM, and have nothing to do with the details of how we are carrying the implicit fundamental null space basis of $[R\ \ B]$.

4. THE ORTHOGONAL FACTORIZATION

In order to preserve the CBAF structure of B in the factorization, we will use the orthogonal factorization $QU = B$, where Q is orthogonal and U upper triangular. The orthogonal factorization of B is uniquely defined, up to a choice of signs [24]. Furthermore a permutation matrix, P, is itself orthogonal. So the upper triangular factor, U, in the orthogonal factorization $QU = PB$ or $(P^tQ)U = \tilde{Q}U = B$ must be independent of the row permutation.

Once we have the QU factorization of B, only the U factor needs to be stored. What we need is a factorization of the inverse, and instead of using

$$B^{-1} = U^{-1}Q^t$$

we can use

$$Q^t = U^{-t}B^t$$

and

$$B^{-1} = U^{-1}U^{-t}B^t.$$

Now, since $(QU)^tQU = U^tQ^tQU = U^tU = B^tB$, we can also compute U by the Cholesky factorization of the symmetric matrix B^tB. So the orthogonal factor, Q, never has to be explicitly computed: see [22].

We can see that U is itself in CBAF, with b $n(k) \times n(k)$ upper triangular diagonal blocks, V^k, b corresponding $n(k) \times n(b+1)$ rectangular angular blocks, W^k, and the final south-east $n(b+1) \times n(b+1)$ triangular block, S.

Bodewig has already considered the idea of symetrizing a matrix in order to solve a linear system, i.e. solve $B^tBx = B^td$ instead of $Bx = d$, in order to get a more stable procedure that is independent of the row permutation, [7]. For the same reasons Saunders considered the use of the QU factorization in the simplex method, and observed that the $B = LQ$ factorization would preserve the block angular structure of problems in row block angular form, [40].

5. BLOCK OPERATIONS

In this and the next sections we derive some procedures to compute and update the Cholesky factor, U, for bases in CBAF using only a few simple block operations. Moreover our procedures allow many of these block operations to be performed in parallel. Our procedures have better complexity bounds than factorizations and updates that do not explicitly use the block structure of the basis, as [22], [24] or [40]. Our update procedure will give us much better bounds even in a purely sequential environment.

Let us consider an efficient direct *block QR factorization*, procedure $bqr()$ illustrated in Figure 2, that takes advantage of the CBAF structure of B, in order to parallelize several steps in the basis' factorization:

1. Compute (in parallel) the QU factorizations of the b diagonal blocks,

$$\begin{bmatrix} V^k \\ 0 \end{bmatrix} \equiv (Q^k)^tB^k.$$

2. Apply (in parallel) the orthogonal transformations to the angular blocks, computing

$$\begin{bmatrix} W^k & Z^k \end{bmatrix} \equiv (Q^k)^tC^k.$$

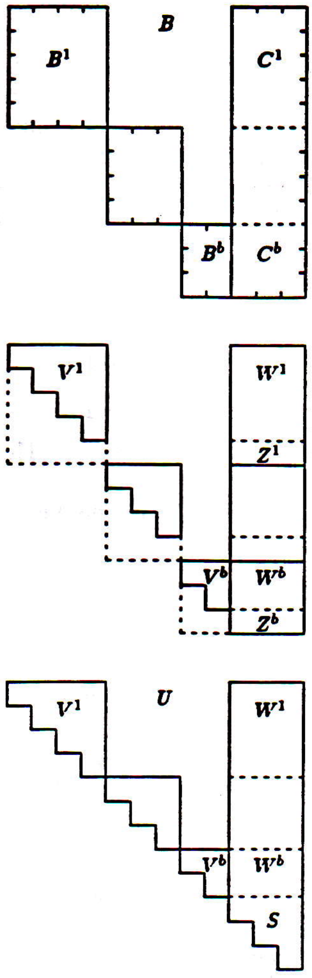

Figure 2. The QR Factorization.

3. Form and factor the "south-east" block Z, i.e.,

$$S \equiv (Q^{b+1})^t Z\,, \quad Z \equiv \begin{bmatrix} Z^1 \\ \vdots \\ Z^b \end{bmatrix}.$$

As we can see, almost all the work in $bqr()$ consists of the repetitive application of some simple block matrix operations. In order to take advantage of this block modularity in the procedures presented in the next sections, we now define a few simple "block operations". A detailed analysis of each one of the basic block operations we need, and the number of floating point operations (FLOPs) they require, can be found in [24]:

1. Compute the *partial Cholesky factorization*, eliminating the first n columns of the block matrix
$$\begin{bmatrix} F & G \\ G^t & 0 \end{bmatrix}$$
to get
$$\begin{bmatrix} V & W \\ 0 & Z \end{bmatrix}$$
where $F = F^t$ is $n \times n$, and G is $n \times l$. This requires $(1/6)n^3 + (1/2)n^2 l + (1/2)nl^2 + O(n^2 + l^2)$ FLOPs.
2. Compute the *partial back transformation*, i.e. u, in
$$\begin{bmatrix} V & W \\ O & I \end{bmatrix}^t \begin{bmatrix} u^1 \\ u^2 \end{bmatrix} = \begin{bmatrix} y^1 \\ y^2 \end{bmatrix}$$
where V is $n \times n$ upper triangular and W is $n \times l$, 0 and I are the zero and the identity matrices, and u and y are column vectors. This requires $(1/2)n^2 + nl + O(n + l)$ FLOPs.
3. *Reduce to upper triangular form* an upper Hessenberg matrix, i.e., apply a sequence of Givens rotations to the row pairs $\{1,2\}, \{2,3\}, \ldots \{n-1, n\}$ of the block matrix
$$[V \;\; W]$$
where V is $n \times n$ upper Hessenberg, and W is $n \times l$, in order to reduce V to upper triangular. This requires $2n^2 + 4nl + O(n^2 + l^2)$ FLOPs.
4. *Reduce to upper triangular form* a column—upper triangular block matrix, i.e., apply a sequence of Givens rotations to the row pairs $\{n, n-1\}, \{n-1, n-2\}, \ldots \{2, 1\}$ of the block matrix
$$[u \;\; V]$$
where u is an $n \times 1$ column vector, and V is $n \times n$ upper triangular, in order to reduce u to a single NZE in the first row, so transforming V from triangular to upper Hessenberg. This requires $2n^2 + O(n)$ FLOPs.

In the block QR factorization, many of the block operations we have to execute are independent. Therefore, the block angular structure gives us not only the possibility of preserving sparsity, but also gives us the opportunity to perform several independent block operations in parallel. In order to study the advantages of parallelizing the independent block operations required in our procedures we define a simple parallel computer. In our parallel complexity analysis we use a network environment, consisting of $b+1$ nodes, every node having a processor (CPU) and local memory. For $k = 1 \dots b$, we allocate the blocks of matrices A and U to specific nodes, as follows:

- The blocks D^k E^k, V^k and W^k are allocated at node k.
- The south-east blocks Z and S are allocated at node 0 (or $b+1$).

In the next sections we will express our complexity bounds in terms of the sum and the maximal block dimensions:

$$dbsum = \sum_1^b m(k)$$

$$dbmax = \max\{m(1), \dots, m(b), n(b+1)\}.$$

In our complexity analysis we will not only account for the processing time measured in FLOP-time units, $pTime$, but also for the necessary internode communication, INC. When b block operations, $bop^1 \dots bop^b$, can proceed in parallel (at different nodes), we bound their processing time by $\wedge_1^b flops(bop^k) = flops(bop^1) \wedge \dots \wedge flops(bop^b)$, where $\wedge$ is the maximum operator, and $flops(bop^k)$ is the number of floating point operations necessary at bop^k. In the equations that follow, $\wedge$ has lower precedence then any multiplicative or additive operator. The expressions "At node *k=1:b* compute" or "From node *k=1:b* send" mean, "At (from) all the nodes $1 \leq k \leq b$, in parallel, compute (send)". In the complexity upper bounds we give in the next sections, we will always neglect lower order terms.

6. BLOCK CHOLESKY FACTORIZATION

We can take advantage of the CBAF of B in order do the Cholesky factorization of B^tB with better performance, and parallelizing several block operations. We now give an algorithmic description of the block Cholesky factorization, $bch()$, in the simple parallel environment defined in Section 5. At each step we indicate by $pTime = \dots$ an upper bound to the required processing time, in FLOP units, and by $INC = \dots$ an upper bound to the required internode comunication. As shown in Figure 3, the steps of $bch()$ are as follows:

1. At node *k=1:b* compute the blocks $(B^k)^tB^k$, $(B^k)^tC^k$, and $(C^k)^tC^k$.
   $pTime = m(k)n(k)^2 + m(k)n(k)n(b+1) + m(k)n(b+1)^2 \leq 3dbmax^3$, $INC = 0$.
2. Send $(C^k)^tC^k$ from node k to node 0, where we accumulate $Z^0 = \sum_1^b (C^k)^tC^k$.
   $pTime = b\ n(b+1)^2 \leq b\ dbmax^2$, $INC = b\ n(b+1)^2 \leq b\ dbmax^2$
3. At node k compute the partial Cholesky factorization, eliminating the first $n(k)$

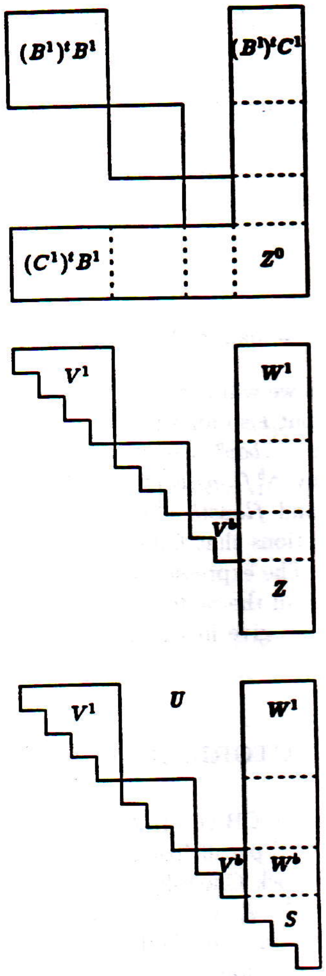

Figure 3. The Block Cholesky Factorization.

columns, of the block matrix

$$\begin{bmatrix} (B^k)^t B^k & (B^k)^t C^k \\ (C^k)^t B^k & 0 \end{bmatrix}$$

to get

$$\begin{bmatrix} V^k & W^k \\ 0 & Z^k \end{bmatrix}$$

$pTime = (1/6)n(k)^3 + (1/2)n(k)^2 n(b+1) + (1/2)n(k)n(b+1)^2 \leq (7/6)dbmax^3$, $INC = 0$.

4. Send Z^k from node k to node 0 where we accumulate $Z = \sum_0^b Z^k$.
   $pTime = b\ n(b+1)^2 \leq b\ dbmax^2$, $INC = b\ n(b+1)^2 \leq b\ dbmax^2$.
5. At node 0 factor the south-east corner $S = chol(Z)$, where $chol()$ indicates the standard Cholesky factorization.
   $pTime = (1/6)n(b+1)^3 \leq (1/6)dbmax^3$, $INC = 0$.

Theorem 6.1. *The block Cholesky factorization, $bch()$, requires no more than* $(4 + 1/3)dbmax^3 + b\ dbmax^2$ *processing time, and* $b\ dbmax^2$ *internode communication.*

At Steps 2 and 4, if the the network allows parallel internode communications, and the topology of the network is rich enough, we can "fan in" the accumulated matrices in only $log(b)$ phases, see [6] and [10]. Each phase requires $\approx b/2^l$ parallel and independent tasks, where at each task we transmit and add $dbmax^2$ reals. With this interpretation in mind we can, in Theorem 6.1, substitute b by $log(b)$.

7. BLOCK UPDATE PROCEDURE

We now address the problem of updating the QU factorization of the basis when a basic column, namely the *outj* column of the *outk* block of B, gets "out" of the basis, and the *inj* column of the *ink* block of R, a, comes "in" to the basis. Actually, as explained in Section 3, we only really need to maintain the triangular factor, U. Therefore we want an update procedure that recomputes U after a pivot. There are intuitive reasons for us to hope for an efficient update procedure:

- A pivot can be seen as two rank one modifications of B, namely we delete a column and then add a column to B. There are several procedures to update the QU factorization of B in this cases, like [22], [24] or [40]. Therefore we could use a generic "delete column, add column" two step update procedure, even without taking into consideration the block structure of B.
- We know, from Section 4, that the (block) structure of U is uniquely defined by the structure of B. Also, deleting a column from one block of B and adding a column to a second block, results in a "small" structural change, in B or U. Therefore there ought to be an efficient way to update the factor U after a pivot.

Let us now present the block update procedure, *bup*(), that explicitly uses the column block angular structure of B, in order to achieve a much better performance than a generic rank one update. The block update procedure is described in terms of the block operations defined in Section 5. In *bup*() we consider five different cases:

Case I. $ink \neq outk, \quad ink \neq b+1, \quad outk \neq b+1.$
Case II. $ink = outk, \quad ink \neq b+1.$
Case III. $ink \neq b+1, \quad outk = b+1.$
Case IV. $ink = b+1, \quad outk \neq b+1.$
Case V. $ink = outk, \quad ink = b+1.$

Let us examine in detail case I, i.e., when *ink* and *outk* are distinct diagonal blocks, as shown in Figure 4. In this case the only NZEs in the outgoing column are in $B^{outk}_{\bullet,outj} = D^{outk}(:,bp^{outk}(outj))$ and the only NZEs in the incoming column, a , are in $a^{ink} = D^{ink}(:,rp^{ink}(inj))$, see Figure 1.

Let us define $y \equiv B^t a$, and $u \equiv Q^t a = U^{-t}B^t a = U^{-t}y$. We notice that vector y has the block structure of a row in the *ink* block of B, and so does vector u. Namely the only NZEs on u are in the blocks u^{ink} and u^{b+1} ,

$$\begin{bmatrix} u^{ink} \\ u^{b+1} \end{bmatrix} = \begin{bmatrix} V^{ink} & W^{ink} \\ 0 & S \end{bmatrix}^{-t} \begin{bmatrix} y^{ink} \\ y^{b+1} \end{bmatrix}.$$

In order to update U we remove the $outj$ column of the $outk$ block, $U^{\bullet,outk}_{\bullet,outj}$, and insert u as the last column of $U^{\bullet,ink}$. Then we only have to reduce U to an upper triangular matrix, by means of orthogonal transformations. The only orthogonal transformations we use are permutations and the reductions by Givens rotations defined is Section 5. Namely we:

- Reduce $\begin{bmatrix} V^{outk} & W^{outk} \end{bmatrix}$ from upper Hessenberg to upper triangular.
- Reduce $\begin{bmatrix} u^{b+1} & S \end{bmatrix}$ to upper triangular.
- Insert the first row of $U^{b+1,\bullet}$, as the the last row of $U^{ink,\bullet}$. Then insert the last row of $U^{outk,\bullet}$ as the first row of $U^{b+1,\bullet}$.
- Reduce S from upper Hessenberg to upper triangular.

The other cases are very similar. We now give an algorithmic description of the block update procedure, *bup*(), in the simple parallel environment defined in Section 5. The steps of *bup*() are permutations, or the basic block operations defined in Section 5. At each step we indicate by $pTime = \dots$ an upper bound to the required processing time, in FLOP units, and by $INC = \dots$ an upper bound to the required internode communication.

These are the steps for case I:

1. At node *ink*, compute $y^{ink} = (B^{ink})^t a^{ink}$ and $y^{b+1} = (C^{ink})^t a^{ink}$.
   $pTime = m(ink)n(ink) + m(ink)n(b+1) \leq 2dbmax^2$, $INC = 0$.
2. At node *ink*, compute the partial back transformation

$$\begin{bmatrix} u^{ink} \\ z \end{bmatrix} = \begin{bmatrix} V^{ink} & W^{ink} \\ 0 & I \end{bmatrix}^{-t} \begin{bmatrix} y^{ink} \\ y^{b+1} \end{bmatrix}.$$

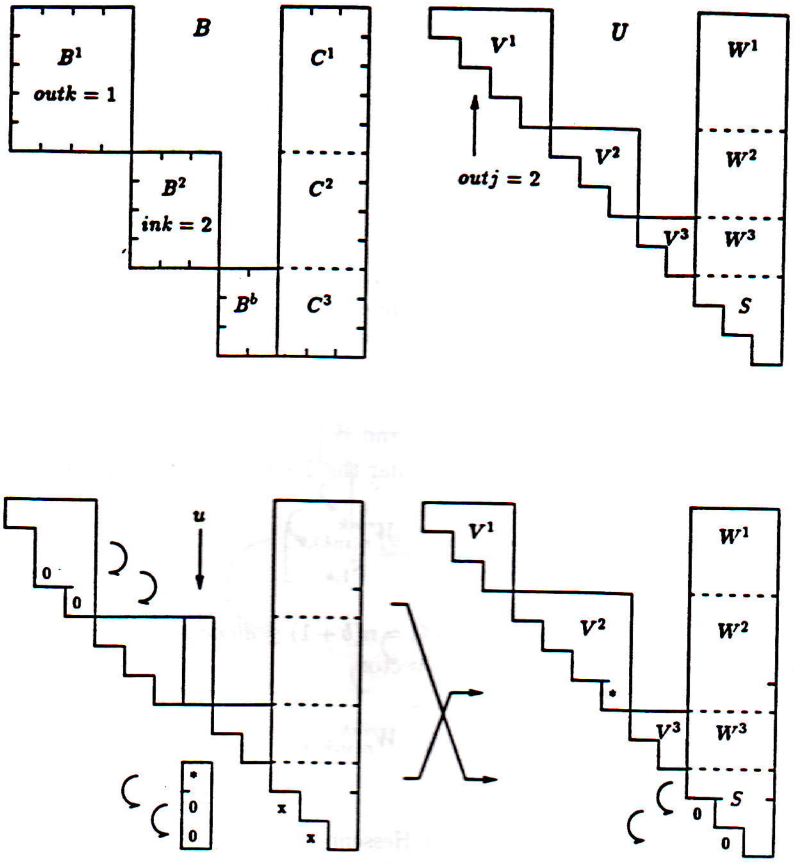

Figure 4. The Block Update Procedure, Case I.

Then insert u^{ink} as the last column of V^{ink}.
$pTime = (1/2)n(ink)^2 + n(ink)n(b+1) \leq (3/2)dbmax^2$, $INC = 0$.

3. From node ink send z to node 0.

$pTime = 0$, $INC = n(b+1) \leq dbmax$.

4. At node 0 compute $u^{b+1} = S^{-t}z$.
   $pTime = (1/2)n(b+1)^2 \leq (1/2)dbmax^2$, $INC = 0$.
5. (a) At node $outk$, remove the column $V^{outk}_{\bullet,outj}$ from V^{outk}. Then reduce $\begin{bmatrix} V^{outk} & W^{outk} \end{bmatrix}$ from upper Hessenberg to upper triangular.
   (b) At node 0, reduce $\begin{bmatrix} u^{b+1} & S \end{bmatrix}$ to upper triangular.
   Observe that the block operations at steps $5a$ and $5b$ are independent, so $pTime = 2n(ink)^2 + 4n(ink)n(b+1) \wedge 2n(b+1)^2 \leq 6dbmax^2$, $INC = 0$.
6. From node 0 send vector $S_{1,\bullet}$ to node ink, where it is inserted as the last row of W^{ink}. From node 0 send element u_1^{b+1} to node ink, where it is inserted as $U^{ink}_{n(ink)+1,n(ink)+1}$. From node $outk$ send vector $W^{outk}_{n(outk),\bullet}$ to node 0, where it is inserted as the first row of S.
   $pTime = 0$, $INC = 2n(b+1) + n(outk) \leq 3dbmax$.
7. At node 0, reduce S from upper Hessenberg to upper triangular.
   $pTime = 2n(b+1)^2 \leq 2dbmax^2$, $INC = 0$.

These are the steps for case II:

Steps 1–5 are exactly as in case I.

6. (a) From node ink send $V^{ink}_{n(ink),n(ink)}$ and $W^{ink}_{n(ink),\bullet}$ to node 0.
   (b) At node 0 reduce to upper triangular the $2 \times n(b+1)+1$ matrix

$$\begin{bmatrix} V^{ink}_{n(ink),n(ink)} & W^{ink}_{n(ink),\bullet} \\ u_1^{b+1} & S_{1,\bullet} \end{bmatrix}$$

   $pTime = 4n(b+1) \leq 4dbmax$, $INC = n(b+1) \leq dbmax$.
7. (a) From node 0 send the modified vector

$$\begin{bmatrix} V^{ink}_{n(ink),n(ink)} & W^{ink}_{n(ink),\bullet} \end{bmatrix}$$

   back to node ink.
   (b) At node 0, reduce S from upper Hessenberg to upper triangular.
   $pTime = 2n(b+1)^2 \leq 2dbmax^2$, $INC = n(b+1) \leq dbmax$.

These are the steps of case III:

Steps 1–4 are exactly as in case I.

5. (a) At node $k = 1 : b$ remove the column $W^k_{\bullet,outj}$ from W^k.
   (b) At node 0 reduce $\begin{bmatrix} u^{b+1} & S \end{bmatrix}$ to upper triangular. Then remove $S_{\bullet,outj}$ from S.
   $pTime = 2n(b+1)^2 \leq 2dbmax^2$, $INC = 0$.
6. From node 0 send to node ink, u_1^{b+1} to be inserted in V^{ink} as $V^{ink}_{n(ink)+1,n(ink)+1}$, and $S_{1,\bullet}$ to be inserted as the last row of W^{ink}.
   $pTime = 0$, $INC = n(b+1) \leq dbmax$.
7. At node 0, reduce S from upper Hessenberg to upper triangular.
   $pTime = 2n(b+1)^2 \leq 2dbmax^2$, $INC = 0$.

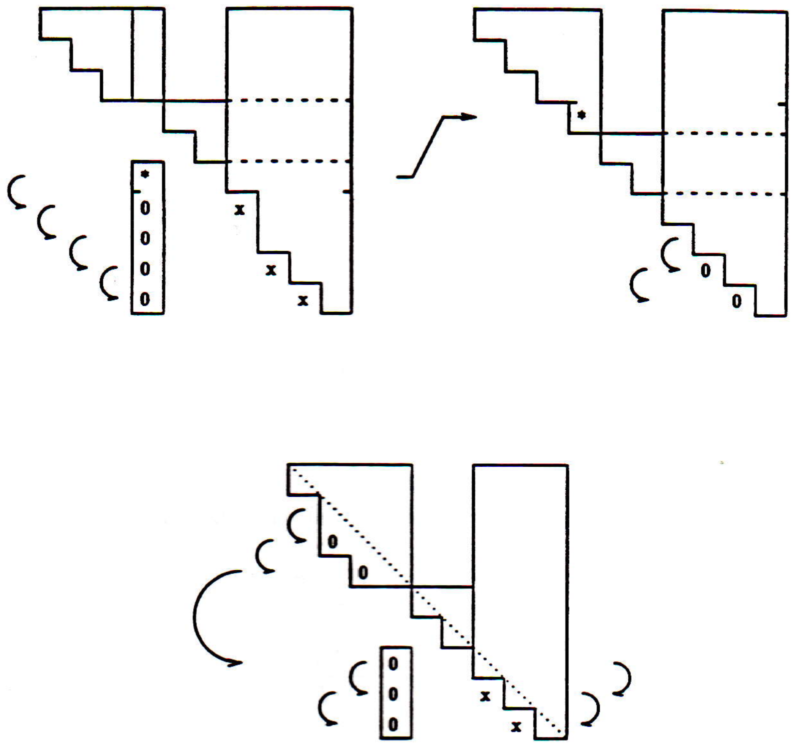

Figure 5. The Block Update Procedure, Case II and III.

These are the steps of case IV:

1. At node *k=1:b*, compute $y^k = (B^k)^t a^k$ and $x^k = (C^k)^t a^k$.
   $pTime = \wedge_1^b\ m(k)n(ink) + m(k)n(b+1) \leq 2dbmax^2$, $INC = 0$.
2. At node *k=1:b*, compute the partial back transformation

$$\begin{bmatrix} u^k \\ z^k \end{bmatrix} = \begin{bmatrix} V^k & W^k \\ 0 & I \end{bmatrix}^{-t} \begin{bmatrix} y^{ink} \\ x^k \end{bmatrix}$$

   and insert u^k as the last column of W^k.

$pTime = \wedge_1^b\ (1/2)n(k)^2 + n(k)n(b+1) \leq (3/2)dbmax^2$, $INC = 0$.

3. From node k=*1:b* send z^k to node 0, where we accumulate $z = \sum_1^b z^k$.
   $pTime = b\ n(b+1) \leq b\ dbmax$, $INC = b\ n(b+1) \leq b\ dbmax$.
4. At node 0 compute $u^{b+1} = S^{-t}z$, and insert u^{b+1} as the last column of S.
   $pTime = (1/2)n(b+1)^2 \leq (1/2)dbmax^2$, $INC = 0$.
5. Remove the column $V^{outk}_{\bullet,outj}$ from V^{outk} , and reduce $[V^{outk}\ \ W^{outk}]$ to upper triangular.
   $pTime = 2n(outk)^2 + 4n(outk)n(b+1) \leq 6dbmax^2$, $INC = 0$.
6. Send vector $W^{outk}_{n(ouk),\bullet}$ from node *outk* to node 0, where we insert it as the first row of S, and reduce S to upper triangular.
   $pTime = 2n(b+1)^2 \leq 2dbmax^2$, $INC = n(b+1) \leq dbmax$.

These are the steps of case V:

Steps 1–4 are exactly as in case IV.

5. At node k=*1:b*, remove the column $W^k_{\bullet,outj}$ from W^{outk}, and insert u^k as the last column of W^k. At node 0 remove the column $S_{\bullet,outj}$ from S, and insert u^{b+1} as the last column of S.
   $pTime = 0$, $INC = 0$.
6. At node 0, reduce S from upper Hessenberg to upper triangular.
   $pTime = 2n(b+1)^2 \leq 2dbmax^2$, $INC = 0$.

We summarize the complexity of *bup*() in the following theorem:

Theorem 7.1. *In the block update procedure, bup(), neglecting lower order terms, we have the following upper bounds for the required processing time and internode communication:*

Case	*Processing Times*	*Internode Communication*
I	$12dbmax^2$	$3dbmax$
II	$12dbmax^2$	$3dbmax$
III	$8dbmax^2$	$2dbmax$
IV	$12dbmax^2 + b\ dbmax$	$b\ dbmax$
V	$6dbmax^2 + b\ dbmax$	$b\ dbmax$

As in Theorem 6.1, if the network allows parallel internode communication, we can substitute b by $log(b)$ in our complexity expressions.

In light of Theorem 2, and the previous sections, we can compare *bup*() with other basis factorization and update techniques:

- In the standard LU factorization updates used in ASMs, the original factorization $B = LU$ is replaced, after s pivots, by a sequence $B = LL^1L^2 \dots L^sU^s$, see [1], [31], [41], and [42], where L^t is lower triangular with a single nontrivial column. So the LU factorization is maintained as a product sequence with an increasing number of factors. In our case we would also have the progressive degrading of the CBAF structure in U^s, as explained in Section 3. That makes it undesirable to continue to update the factorization for large values of s, even in the absence of numerical errors. Instead we would frequently start a fresh factorization, i.e.

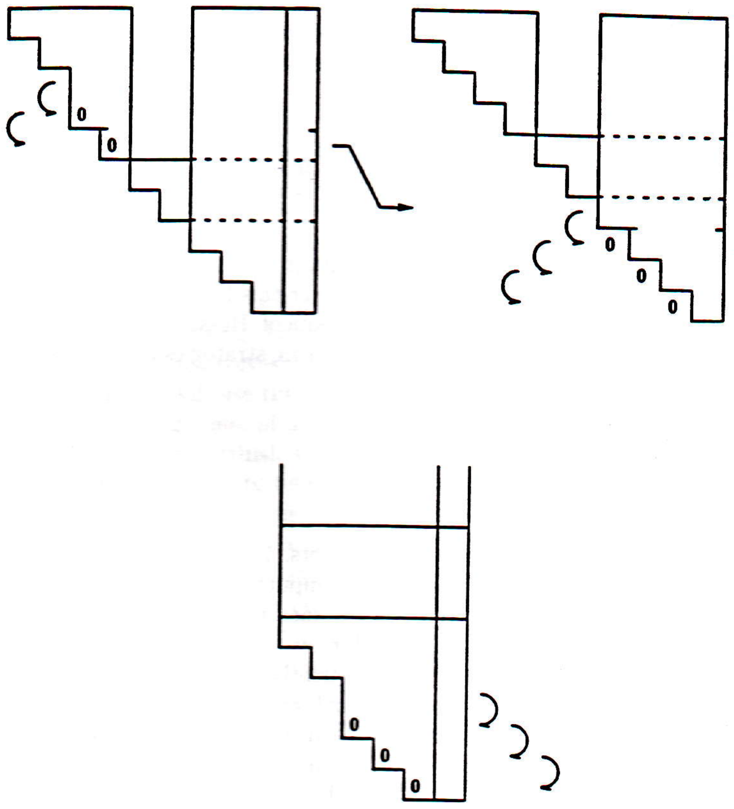

Figure 6. The Block Update Procedure, Case IV and V.

"reinvert" the Basis. In contrast, the QU factorization maintains the CBAF structure of B, and the factorization is given by a single matrix, U, instead of the product sequence in the LU factorization. Moreover we know that the QU factorization has much better numerical stability then the LU factorization: see [24] and [28].

- A generic delete-column add-column QU update as [22], [24] or [40], would require

$O(dbsum^2)$ time. Even using sparse data structures for the matrices involved, the generic updates require the rotation of all rows of U, so they would still require $O(dbsum\ dbmax)$ processing time in all the 5 pivot cases. Moreover those *dbsum* rotations have to be done sequentially. The careful use of the CBAF structure of B by *bup*(), gives us the much better bounds of Theorem 2. From computational experience with ASMs for large block angular problems, we know that in real application problems, most of the updates will be of case II, i.e. $ink = outk$, $ink \neq b+1$, see [34] and [14], where *bup*() requires only $O(dbmax^2)$ time, even in a purely sequential environment!

From the above we can expect the *bup*() to outperform the standard LU and QU updating techniques, specially if $dbsum \gg dbmax$ and $s \gg dbmax$, i.e., when the basis is much larger than its larger block, and we have to pivot many more times than there are columns in a single block.

When pivoting, by putting the entering column at the end of its block, we are imposing a particular column permutation on B. We prefer this fixed column ordering strategy for its small overhead and simplicity of the subsequent Hessenberg updates. It is however possible to consider more elaborate column ordering strategies at updates and reinversions [11], [21].

8. REINVERSIONS

In any ASM, after a given number of pivots, the accumulation of errors in the updates forces us to "reinvert" the basis, i.e. recompute the Cholesky factor U directly from B. From Section 7 we know that most of the factorization work consists of independent factorizations of the diagonal blocks B^k. Moreover, it is a well known fact that in ASMs for problems in CBAF, the basis pivots frequently replace a column from one block by a column from the same block, i.e. usually $ink = outk$, see [34] and [14]. Therefore, we will probably have some blocks that have been updated more times, and have accumulated larger errors then others. When reinverting we can take advantage of these facts by checking the accuracy of each diagonal block factorization, $(U^k)^t U^k = (B^k)^t B^k$, and only reinvert the diagonal blocks that have accumulated large errors. Of course, we always have to reinvert the final south-east block Z.

Before a reinversion, we should address the question of how to order the columns of the basis B. As explained in Section 4, the orthogonal factorization of a row and column permutation of B, QBP, is independent of the row permutation, Q. But we can still take advantage of the column permutation, P, in order to preserve sparsity. As mentioned above, at each reinversion, only a few blocks of the basis may need a fresh refactorization. Therefore we do not want to pay the time to run a column ordering algorithm for each individual block B^k to be reinverted. This situation is studied in [40]: At the beginning of the simplex, we order the columns of each rectangular block, A^k $k = 1 \ldots b$, into a "near upper triangular form" (NUTF), and then at each reinversion, order the columns in B^k as they are ordered in A^k [33].

9. NUMERICAL EXPERIMENTS

In this section we compare the performance of the simplex algorithm, using the LU or the QR factorization, when solving linear programs in CBAF. Our test problems have the structure of the b–scenarios investment problems, like the example we gave is Section 1. Our test problems have diagonal blocks of dimension $dbmax \times 2\ dbmax$, b angular columns, plus an embedded identity matrix. In order to have a large set of test problems, we use random numbers to generate (admissible) numerical values to the NZEs. The simplex always begins at the identity basis which, from the way the problem is formulated, gives us a feasible vertex.

In the following, qr-simplex is an implementation of the simplex algorithm using the QR factorization and updates, as described in the previous sections, and lu-simplex is an implementation of the simplex algorithm using a sparse LU factorization and rank one updates [1]. The two algorithms were implemented in different environments, so we can not directly compare runing times, nor do we have direct access to a FLOPs counter. However the runing time of both algorithms is dominated by the back solves of the form $Bx = d$, where B is the current basis, using the available factorization of B. But the number of FLOPs necessary for those back solves is essentially proportional to the number of NZEs in the factors. Therefore we will use the fill in the basis factors as an indirect measure of the cost, in FLOPs or runing time, of a step in the simplex. Our analysis will not take into account all the parallelism intrinsic to the qr-simplex. Even so, in a purely sequential environment, the qr-simplex seems to be a better alternative to the standard lu-simplex.

In the qr-simplex we only carry the upper triangular Cholesky factor $R = Q^t B$. So we only monitor the number of NZEs in R, $\rho = nze(R)$. In the lu-simplex we carry the upper triangular factor U, the initial lower triangular factor, L, and a sequence of rank one updates, L^1, L^2, ... , L^{cup}, where cup is the number of times we updated the basis. Each rank one update is a lower triangular matrix that only differs from the identity at one column. We keep the nontrivial columns of these rank-one transformations sequence in a $dbsum \times cup$ matrix, LSEQ, where $dbsum = b\ dbmax$. Since our starting basis is the identity, the initial lower triangular factor is trivial, and we only monitor $\upsilon = nze(U)$, $\lambda = nze(LSEQ)$, and the total fill $\tau = \upsilon + \lambda$.

Before we analyze statistical data, let us examine in detail a small example. In this example we have $dbmax = 6$ and $b = 3$. Each row in Table 1 (next page) is one pivot step. The columns in Table 1 are as follows. Column 6 is the value of the objective function. Column 2 is the fill in the Cholesky factor, ρ. Column 3, 4 and 5 are the fill in U, in $LSEQ$, and the total, i.e., υ, λ, and τ. Column 7 is the pivot's "case", as defined in Section 7. Column 1 is the order in which the vertex was visited by the simplex.

In Figure 7 we plot columns 2, 3, 4 and 5 of Table 1, ρ, υ, λ and τ, versus the pivot sequence order in column 1. The values in these four columns are plotted, respectively, with a solid line, a dashed line, a dotted line, and a dash-dotted line.

Table 1. Vertex sequence of the example in Figure 7.

Pivot	ρ	υ	λ	τ	Cost Funct.	*Case*
0	18	18	0	18	100.0000	
1	35	35	0	35	87.4386	4
2	40	39	1	40	76.7990	2
3	45	43	2	45	56.3810	2
4	49	48	4	52	47.4169	3
5	65	62	7	69	39.1938	4
6	69	65	12	77	36.1511	2
7	76	67	18	85	32.8104	2
8	82	75	24	99	31.6119	2
9	82	77	25	102	26.1915	1
10	67	62	28	90	25.6187	3
11	68	66	28	94	24.7787	2
12	75	69	34	103	23.9997	2
13	77	71	40	111	21.6801	1
14	80	75	40	115	20.7741	2
15	84	77	47	124	19.7974	2
16	72	74	53	127	19.1210	3
17	71	76	57	133	19.1169	2
18	74	73	64	137	18.4936	2
19	69	81	73	154	17.0904	3
20	68	78	73	151	16.8475	2
21	68	83	82	165	16.7994	2

Let us first compare $nze(R)$ versus $nze(U)$. There are two important effects contributing to fill the upper triangular factor, each favoring one of the factorizations:

1. Let us consider the factorization of M, a $2 \times n$ matrix. In the LU factorization we add a multiple of the first to the second row, in order to eliminate $M(2,1)$. After this "elementary row transformation" the sparsity structure of the first row remains unchanged, and the sparsity structure of the second row becomes the Boolean sum of the structures of the two rows. In the QR factorization we apply a Givens rotation to eliminate $M(2,1)$. But now the sparsity pattern of both rows become the Boolean sum of the sparsity structure of the original rows of M (except for the eliminated element, of course). From this we can see why orthogonal transformations tend to produce much more fill than elementary row transformations, which tends to favor the LU factorization.
2. The QR factorization preserves the CBAF structure of B in the Cholesky factor, R, as extensively analyzed in the previous sections. Therefore the fill in R is confined to the nontrivial blocks in its CBAF structure. On the other hand, the LU factorization progressively degenerates the CBAF structure, allowing fill to occur anywhere in U. That tends to favor the QR factorization.

We could observe, in our experiments, that the first effect is more important at the first pivots of the simplex. But as the nontrivial blocks of the Cholesky factor become denser, there is a saturation of this effect, and the the fill in R stabilizes, or grows very slowly. The second effect has a cumulative nature, and as the algorithm progresses, it tends to fill U at increasing rates. We also observe that as the simplex approaches optimality, the block structure of the basis becomes very well equilibrated, i.e. all diagonal blocks have approximately the same number of columns; this benefits the qr-simplex, but not necessarily the lu-simplex. In accordance which the comments above, we usually observe a sparser U factor at first steps of the simplex, and a sparser R at the end of the algorithm. We could also observe that the greater the ratio $dbsum/dbmax$, or just b if as in our test problems the blocks have a constant number of rows, the sooner the second effect dominates the first.

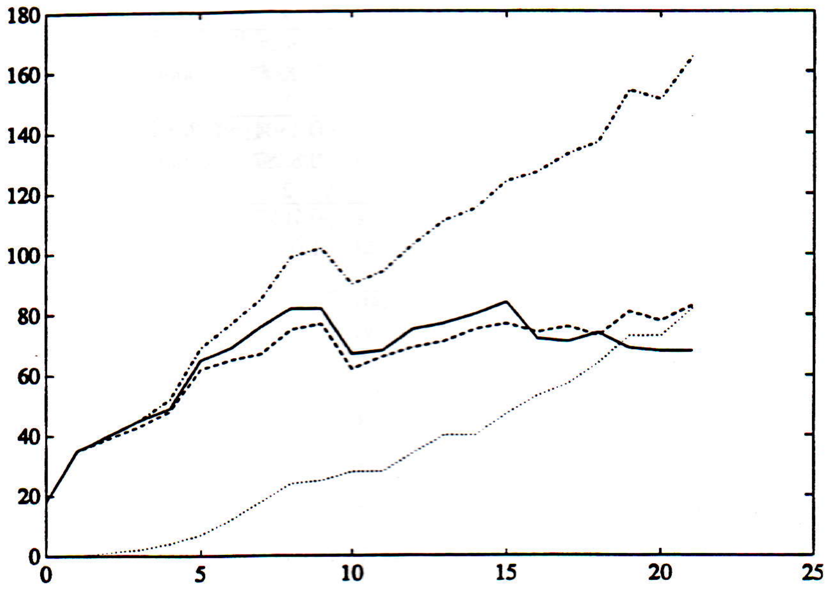

Figure 7. Fill in the factors at the first example in Table 2.

The fill in $LSEQ$ of the lu-simplex is easier to analyze; $nze(LSEQ)$ is a monotonically increasing function, beginning at zero, but growing always faster. When $nze(LSEQ)$ becomes larger then $nze(U)$, a basis reinversion is probably due, as explained in Section 8.

In our test problems that usually happens at the final steps of the algorithm. Moreover, because the non-identity part of the diagonal blocks in the investment problem are dense, and because close to optimality many identity columns have been driven out of the basis, the P3 heuristic applied to a basis at the final steps of the simplex produces almost only spikes, making the reinversion very expensive. As expected from these reasons, in our test problems, more frequent reinversions do not improve the runing time of the *lu*-simplex. For the reasons above, and in order to simplify the comparative analysis, we reinvert neither the *lu*-simplex nor the *qr*-simplex.

In Table 2 we present ratios for the total fill, ρ/τ, and for the upper triangular fill, ρ/υ, after 20%, 40%, 60%, 80% and 100% of the pivots, for some test problems. The first of these problems is the problem at Figure 7, and they all have the same structure, with $dbmax = 6$ and $b = 3$.

Table 2. Examples with 3 blocks of size 6.

τ/ρ	20%	40%	60%	80%	100%
	1.0000	0.8941	0.7282	0.5669	0.4121
1	1.0465	1.1343	1.0870	0.9730	0.8193
	2	13	4	2	0
	1.0000	1.0000	0.7925	0.8095	0.7727
2	1.0000	1.0465	0.9130	1.0000	1.0000
	0	8	2	1	0
	0.9825	0.7500	0.5414	0.4804	0.3684
3	1.0370	0.9600	0.8586	0.8687	0.7636
	6	9	4	2	0
	1.0750	1.0167	0.7922	0.7027	0.6429
4	1.1026	1.1509	1.0339	1.0196	1.0189
	1	10	3	1	0
	1.0000	0.7375	0.6489	0.5508	0.4966
5	1.0545	0.9672	1.0000	0.9420	0.8902
	2	10	4	3	0
	1.0000	0.9324	0.8667	0.6364	0.5760
6	1.0196	1.1311	1.1143	0.9844	0.9863
	3	7	4	3	0
	1.0000	0.9245	0.8590	0.7857	0.7071
7	1.0256	1.0000	1.0308	1.0476	1.0145
	3	10	1	0	2
	1.0000	0.9273	0.7581	0.8125	0.6864
8	1.0256	1.0625	0.9400	1.0833	1.0000
	2	8	3	2	1
	1.0000	1.0250	0.8590	0.6404	0.6491
9	1.0256	1.1389	1.1356	0.9344	1.1212
	1	6	5	3	1
	1.0000	0.9444	0.8148	0.7903	0.7101
10	1.0000	1.0408	1.0000	1.0426	1.0426
	0	9	2	1	0

Table 3 is similar to Table 2, only for larger test problems, with *dbmax* $= 6$ and $b = 9$. The first problem in Table 3 is the one in Figure 8.

Table 3. Examples with 9 blocks of size 6.

τ/ρ	20%	40%	60%	80%	100%
	1.0296	0.7903	0.4128	0.3179	0.2650
1	1.1767	1.0692	0.7237	0.5924	0.5619
	1	42	10	4	4
	0.9297	0.7085	0.4409	0.3148	0.2878
2	1.0620	1.0641	0.8363	0.6706	0.6588
	4	32	11	6	3
	0.9852	0.8939	0.5500	0.3288	0.2981
3	1.1050	1.1654	0.9747	0.7259	0.6602
	15	25	11	7	5
	0.8667	0.6829	0.5379	0.3699	0.2538
4	1.0428	0.9894	0.8765	0.7370	0.5651
	8	28	13	8	11
	0.9866	0.7103	0.3546	0.2887	0.2382
5	1.0889	1.0510	0.6429	0.5869	0.5272
	7	31	11	5	4
	1.0617	0.9603	0.5714	0.4980	0.4295
6	1.1570	1.2946	0.9947	0.9213	0.8281
	2	29	9	5	1
	0.9845	0.6675	0.4959	0.3758	0.3284
7	1.0641	0.9509	0.8237	0.7740	0.7696
	5	26	13	8	1
	0.9731	0.8320	0.6735	0.4325	0.3830
8	1.0824	1.1514	1.1563	0.9463	0.8968
	10	27	9	6	6
	1.0089	0.8049	0.3984	0.2277	0.1942
9	1.0900	1.1898	0.7297	0.5074	0.4709
	7	26	13	7	5
	1.0000	0.6098	0.4745	0.3037	0.2495
10	1.1565	0.9336	0.8381	0.6138	0.5482
	4	31	14	7	7

Table 4 has even larger problems, with *dbmax* $= 6$ and $b = 18$. The first problem in Table 4 is the one in Figure 9.

Table 4. Examples with 18 blocks of size 6.

τ/ρ	20%	40%	60%	80%	100%
υ/ρ	20%	40%	60%	80%	100%
Case	1	2	3	4	5
	0.9564	0.5079	0.3113	0.2681	0.2065
1	1.1573	0.8922	0.6236	0.6014	0.5339
	16	62	25	14	11
	0.9055	0.4919	0.2310	0.1925	0.1733
2	1.0950	0.9196	0.5169	0.4335	0.4247
	6	74	23	12	8
	0.9546	0.5223	0.3152	0.1955	0.1571
3	1.1434	0.9576	0.7011	0.4828	0.4270
	35	64	27	15	9
	0.9351	0.6436	0.3307	0.2615	0.2094
4	1.0925	1.0121	0.6651	0.5664	0.5139
	19	61	28	17	13
	0.9195	0.6789	0.2751	0.2122	0.1776
5	1.0462	1.0309	0.6279	0.4845	0.4512
	10	57	21	9	12
	0.9922	0.5072	0.2816	0.2045	0.1604
6	1.2098	0.9335	0.6482	0.5010	0.4219
	23	69	28	16	17
	0.8868	0.4192	0.3347	0.3173	0.2928
7	1.0583	0.8081	0.7282	0.7308	0.7053
	20	62	20	13	9
	0.9822	0.5390	0.2562	0.1858	0.1559
8	1.1156	0.9425	0.6216	0.4157	0.3907
	15	71	23	11	7
	0.8241	0.3959	0.1683	0.1353	0.1132
9	1.0898	0.7560	0.4140	0.3401	0.3077
	16	83	26	14	13
	0.9782	0.7064	0.4095	0.2569	0.2374
10	1.1681	1.0203	0.7610	0.5525	0.5600
	6	64	19	8	7

After the back solves, the most time consuming operation in a simplex step is the update of the upper triangular factor. In general this update can be as time-consuming as a back solve. However we know that for some of the pivot "cases" in the *qr*-simplex, namely cases 1, 2 or 3 but not 4 or 5, the *qr*-update is very inexpensive, involving block operations local to the blocks receiving or losing a column. We have argued that, in real problems, most of the pivots should be of case 2. In Tables 2 and 3 the third line for each test problem gives the number of pivots of each case, confirming this hypothesis.

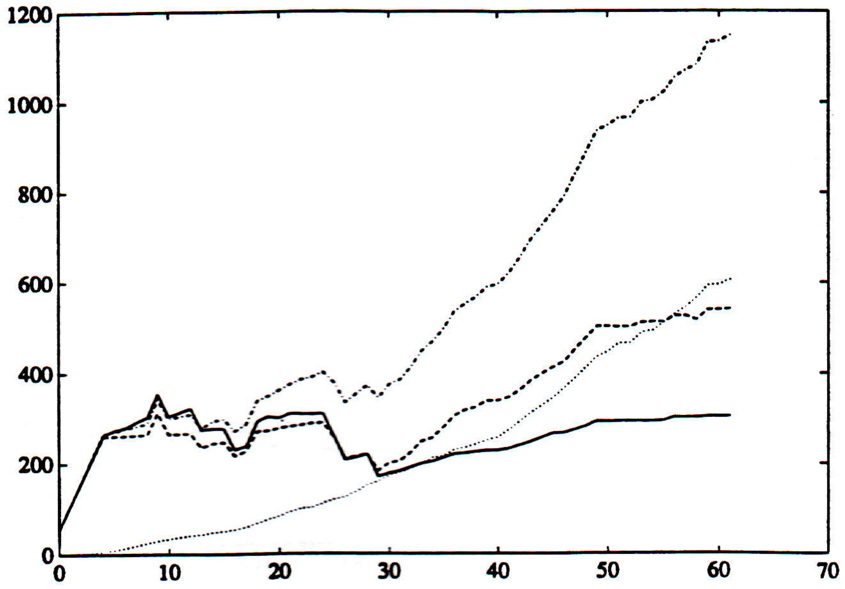

Figure 8. Fill in the factors at the first example in Table 3.

The reported numerical experiments had the QR-Simplex implemented in Sparse-Matlab (Matlab is a trademark of The Mathworks, Inc.). We are currently implementing the QR-Simplex in *C* and *PVM*, a network "Parallel Virtual Machine" process manager [4]. This implementation, on an heterogeneous Sun SPARCstation network, is intended to solve large portfolio planning financial problems [19], [43].

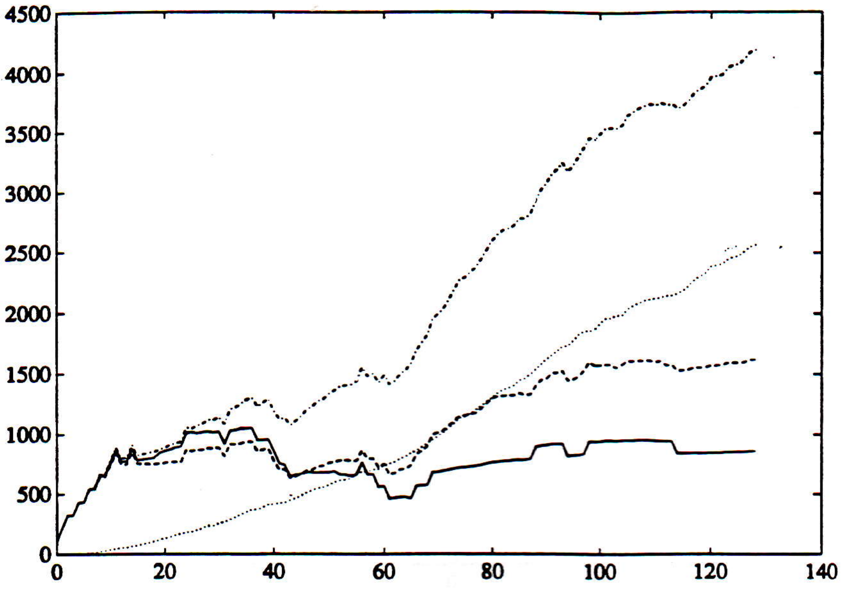

Figure 9. Fill in the factors in the first example in Table 4.

REFERENCES

[1] R.H. BARTELS and G.H. GOLUB, *The simplex method of linear programming using LU decomposition*, Comm. ACM, 12 (1969), pp. 266–268.

[2] M. BASTIAN, *Aspects of basis factorization for block angular systems with coupling rows. In [14].*

[3] M.S. BAZARAA and C.M. SHETTY, *Nonlinear Programming*, John Wiley, Chichester, 1979.

[4] A. BEGUELIN, J. DONGARRA, A. GEIST, R. MANCKEK, and V. SUNDERAM, *A Users' Guide to PVM Parallel Virtual Machine*, ORNL-TM-11826, Oak Ridge National Laboratory, 1992.

[5] J.M. BENNETT, *An aproach to some structured linear programming problems*, Operations Research, 14(4) (1966), pp. 636–645.

[6] D.P. BERTSEKAS and J.N. TSITSIKLIS, *Parallel and Distributed Computation: Numerical Methods*, Prentice Hall, Englewood Cliffs, 1989.

[7] E. BODEWIG, *Matrix Calculus*, North-Holland, Amsterdam, 1956.

[8] A.G. BUCKLEY and J.L. GOFFIN, *Algorithms for Constrained Minimization of Smooth Nonlinear Functions*, Mathematical Programming Study 16. North Holland, Amsterdam, 1982.

[9] J.R. BUNCH and D.J. ROSE, *Sparse Matrix Computations*, Academic Press, New York, 1976.

[10] G.F. CAREY, *Parallel Supercomputing*, John Wiley, Chichester, 1989.

[11] T.F. COLEMAN and A. POTHEN, *The null space problem II. algorithms*, SIAM J. Alg. Disc. Meth., 8 (1987), pp. 544–563.

[12] T.F. COLEMAN and C. VAN LOAN, *Handbook for Matrix Computations*, SIAM, Philadelphia, 1988.

[13] G.B. DANTZIG, *Linear Programming and Extensions*, Princeton University Press, Princeton, 1963.

[14] B. DANTZIG, M.A.H. DEMPSTER, and M.J. KALLIO, Eds., *Large-Scale Linear Programming*, IIASA Collaborative Proceedings Series CP-81-S1, Laxenburg, Austria.

[15] B. DANTZIG and W. GLYNN, *Parallel processors for planning under uncertainty In [38]..*

[16] I.S. DUFF and G.W. STEWART, *Sparse Matrix Proceedings*, SIAM, Philadelphia, 1978.

[17] I.S. DUFF, *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford, 1986.

[18] A.M. ERISMAN, R.G. GRIMES, J.G. LEWIS, and W.G. POOLE, *A structurally stable modification of Hellerman–Rarick's P4 algorithm for reordering unsymmetric sparse matrices*, SIAM J. Numer. Anal., 22(2) (1985), pp. 369–385.

[19] E. ERMOLIEV and R.J.B. WETS, *Numerical Procedures for Stochastic Optimization*, Springer-Verlag, Berlin, 1987.

[20] G.E. FORSYTHE and C.M. MOLER, *Computer Solution of Linear Algebric Systems*, Prentice Hall, Englewood Cliffs, 1967.

[21] J. GILBERT and M. HEATH, *Computing a sparse basis for the null space*, SIAM J. Alg. Disc. Meth., 8 (1987), pp. 446–459.

[22] P.E. GILL, G.H. GOLUB, W. MURRAY, and M. SAUNDERS, *Methods for modifying matrix factorizations*, Math. Comp., 28 (1974), pp. 505–535.

[23] P.E. GILL, G.H. GOLUB, W. MURRAY, and M. SAUNDERS, *Methods for computing and modifying LDV factorizations of a matrix*, Math. of Comp., 29 (1975), pp. 1051–1077.

[24] G.E. GOLUB and C.F. VAN LOAN, *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1983.

[25] F. GUSTAVSON, *Finding the block lower triangular form of a sparse matrix. In [9]..*

[26] D. KENDRICK, *Stochastic Control for Economic Models*, McGraw-Hill, New York, 1980.

[27] L.S. LASDON, *Optimization Theory for Large Systems*, The Macmillan Company, New York, 1970.

[28] C.L. LAWSON and R.J. HANSON, *Solving Least Square Problems*, Prentice Hall, Englewood Cliffs, 1974.

[29] D.G. LUENBERGER, *Linear and Nonlinear Programming*, Addison-Wesley, Reading Massachusetts, 1984.

[30] J.M. MULVEY and H. VLADIMIROU, *Stachastic Network Programming for Financial Planning Problems*, Report SOR-89-7, Department of Civil Engineering and Operations Research,, Princeton University, Princeton, 1989.

[31] B.A. MURTAG, *Advanced Linear Programming*, McGraw-Hill, New York, 1981.

[32] B.A. MURTAGH and M.A. SAUNDERS, *A projected Lagrangean algorithm and its implementations for sparse nonlinear constraints. In [8].*

[33] W. ORCHARD-HAYS, *Advanced Linear-Programming Computing Techniques*, McGraw-Hill, New York, 1968.

[34] A.F. PEROLD and G.B. DANTZIG, *A basis factorization method for block triangular linear programs. In [16].*

[35] S. PISSANETZKY, *Sparse Matrix Technology*, Academic Press, London, 1984.

[36] A. PREKOPA, *Studies on Mathematical Programming*, Akademiai Kiado, Budapest, 1980.

[37] D.J. ROSE and R.A. WILLOUGHBY, eds., *Sparse Matrices and Applications*, Plenum Press, New York, 71.

[38] J.B. ROSEN, ed., *Supercomputers and Large–Scale Optimization*, Baltzer AG, Basel, 1990.

[39] J.B. ROSEN and R.S. MAIER, *Parallel solution of large–scale block angular linear programs. In [38]..*

[40] M.A. SAUNDERS, *Large Scale Linear Programming Using the Cholesky Factorization*, Computer Science Department, Stanford University, 1972.

[41] M.A. SAUNDERS, *A fast, stable implementation of the simplex method using Bartels and Golub updating. In [9].*

[42] J.M. STERN, *Algoritimos Eficientes de Programa cão Linear*, Instituto de Matemática e Estatistica da Universidade de São Paulo, São Paulo, 1987.

[43] J.M. STERN, *Sparse Null Bases for Structured Optimization Problems*, Ph.D. thesis, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, 1991.

[44] J.H. WILKINSON, *The Algebric Eigenvalue Problem*, Clarendon Press, Oxford, 1965.

[45] C. WINKLER, *Basis factorization for block angular linear programs. In [36].*

[46] G. ZOUTENDIJK, *Methods of Feasible Directions*, Elsevier, Amsterdam, 1960.